ABSTRACT

Software vulnerabilities are added into programs during its development. Architectural flaws are introduced during planning and design, while implementation faults are created during coding. Penetration testing is often used to detect these vulnerabilities. This approach is expensive because it is performed late in development and any correction would increase lead-time. An alternative would be to detect and correct vulnerabilities in the phase of development where they are the least expensive to correct and detect. Source code audits have often been suggested and used to detect implementations vulnerabilities. However, manual audits are time consuming and require extended expertise to be efficient. A static code analysis tool could achieve the same results as a manual audit but at fraction of the time.

Through a set of cases studies and experiments at Ericsson AB, this thesis investigates the technical capabilities and limitations of using a static analysis tool as an early vulnerability detector. The investigation is extended to studying the human factor by examining how the developers interact and use the static analysis tool.

The contributions of this thesis include the identification of the tools capabilities so that further security improvements can focus on other types of vulnerabilities. By using static analysis early in development possible cost saving measures are identified. Additionally, the thesis presents the limitations of static code analysis. The most important limitation being the incorrect warnings that are reported by static analysis tools. In addition, a development process overhead was deemed necessary to successfully use static analysis in an industry setting.
Automated static code analysis
- A tool for early vulnerability detection

Dejan Baca
Automated static code analysis
- A tool for early vulnerability detection

Dejan Baca

Department of Systems and Software Engineering
School of Engineering
Blekinge Institute of Technology
SWEDEN
You can’t defend. You can’t prevent. The only thing you can do is detect and respond.

—Bruce Schneier
Abstract

Software vulnerabilities are added into programs during its development. Architectural flaws are introduced during planning and design, while implementation faults are created during coding. Penetration testing is often used to detect these vulnerabilities. This approach is expensive because it is performed late in development and any correction would increase lead-time. An alternative would be to detect and correct vulnerabilities in the phase of development where they are the least expensive to correct and detect. Source code audits have often been suggested and used to detect implementations vulnerabilities. However, manual audits are time consuming and require extended expertise to be efficient. A static code analysis tool could achieve the same results as a manual audit but at fraction of the time.

Through a set of cases studies and experiments at Ericsson AB, this thesis investigates the technical capabilities and limitations of using a static analysis tool as an early vulnerability detector. The investigation is extended to studying the human factor by examining how the developers interact and use the static analysis tool.

The contributions of this thesis include the identification of the tools capabilities so that further security improvements can focus on other types of vulnerabilities. By using static analysis early in development possible cost saving measures are identified. Additionally, the thesis presents the limitations of static code analysis. The most important limitation being the incorrect warnings that are reported by static analysis tools. In addition, a development process overhead was deemed necessary to successfully use static analysis in an industry setting.
Acknowledgements

First and foremost, I would like to thank my supervisors PhD Lars-Ola Damm, Docent Bengt Carlsson and Professor Lars Lundberg for their support, especially for valuable feedback on papers and other research related advice.

I would like thank my colleges and developers at Ericsson that put up with my interference and I apologize for the delays and headaches I have caused you. I especially want to thank, Pierre Börjesson, Ibrahim Manjusak, Anders Henriksson, Murali Radhakrishnan and Lars Stavholm for helping me to conduct my case studies at Ericsson, Kai Petersen for writing papers together with me, Martin Bolt for our security discussions, Sigrid Eldh for our discussions and future papers, Martin Hylerstedt for helping me to spell check my work and Peder Enhörning for inspiring me to investigate the economical aspect of static code analysis.

Finally, I would like to thank 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 included as chapters in this thesis.


Chapter 5. Dejan Baca and Bengt Carlsson, 'Static analysis as a security touch point: An industry case study’, *Submitted journal manuscript*, 2009

*Dejan Baca* is the main author of Chapter 3-5, i.e. based on advisory support from the co-authors, he has outlined and written these papers. While on Chapter 2 *Dejan Baca* is the main research result and analysis contributor and *Bengt Carlsson* is the main author.

Papers that are related to but not included in this thesis.


# Table of Contents

1 Introduction .............................................. 1
   1.1 Concepts ............................................... 2
      1.1.1 Software Development .......................... 2
      1.1.2 Software Vulnerabilities ........................ 4
      1.1.3 Early Vulnerability Detection ....................... 6
      1.1.4 Implementation Vulnerabilities ................... 6
      1.1.5 Automated Static Code Analysis .................... 10
   1.2 Research Approach .................................... 12
      1.2.1 Research Questions ............................... 12
      1.2.2 Research Methods .................................. 13
      1.2.3 Research Validity .................................. 14
      1.2.4 Research Environment ............................ 15
      1.2.5 Vulnerability Taxonomy ............................ 17
   1.3 Outline and Contributions ........................... 17
      1.3.1 Chapter 2 ......................................... 18
      1.3.2 Chapter 3 ......................................... 19
      1.3.3 Chapter 4 ......................................... 20
      1.3.4 Chapter 5 ......................................... 21
   1.4 Conclusions ........................................... 22
   1.5 Future Work ............................................ 24

2 Software Security Analysis – Execution Phase Audit .................. 25
   2.1 Introduction ........................................... 25
   2.2 Securing unsecured software .......................... 26
   2.3 Product and development process ........................ 28
   2.4 The investigation .................................... 29
   2.5 Results ............................................... 30
# Table of Contents

2.5.1 The Static analysis tools .................................................. 30  
2.5.2 Manual Examination of the Automated tools findings ................ 31  
2.5.3 Security vulnerabilities .................................................. 32  
2.5.4 Proof of concepts ....................................................... 32  
2.5.5 Return on investment .................................................... 34  
2.6 Discussion ........................................................................... 35  
2.7 Conclusions ........................................................................ 36  

3 Evaluating the Cost Reduction of Static Code Analysis for Software Security .................................................. 39  
3.1 Introduction ........................................................................... 39  
3.2 Related work ........................................................................ 41  
3.3 Research methodology ......................................................... 42  
3.3.1 Taxonomy ........................................................................ 42  
3.3.2 Vulnerabilities ................................................................. 43  
3.3.3 Development and SAT Process ........................................... 43  
3.3.4 Coverity Prevent checkers ................................................... 44  
3.4 Case study ............................................................................. 46  
3.4.1 Examining the tools output .................................................. 47  
3.5 Results ................................................................................... 49  
3.5.1 Product A ....................................................................... 49  
3.5.2 Product B ....................................................................... 51  
3.5.3 Product C ....................................................................... 53  
3.5.4 All products ..................................................................... 55  
3.5.5 Code quality improvement .................................................. 56  
3.6 Discussion ............................................................................. 57  
3.6.1 Conclusion ....................................................................... 58  

4.1 Introduction ........................................................................... 61  
4.2 Background and Related Work ................................................. 63  
4.3 Research Method ................................................................. 63  
4.3.1 Variables ....................................................................... 63  
4.3.2 Subjects ......................................................................... 64  
4.3.3 Instrumentation ................................................................. 65  
4.3.4 Operation ....................................................................... 66  
4.3.5 Threats to Validity ............................................................. 66  
4.3.6 Static analysis tool output examples .................................... 67
## Table of Contents

4.4 Data Analysis ......................................................... 68
4.5 Discussion ............................................................. 71
4.6 Conclusion .............................................................. 73

5 Static analysis as a security touch point: – An industry case study 75
5.1 Introduction ............................................................. 75
5.2 Background and Related Work ........................................ 76
  5.2.1 Security touchpoint .............................................. 76
  5.2.2 Taxonomy ............................................................ 78
  5.2.3 Capabilities of a Static Analysis Tool .......................... 79
5.3 Research method ....................................................... 81
  5.3.1 Case Study Context .............................................. 81
  5.3.2 Research Questions and Propositions .......................... 81
  5.3.3 Case Selection and Units of Analysis .......................... 82
  5.3.4 Data Collection Methods ....................................... 84
  5.3.5 Threats to Validity ............................................... 87
5.4 Results ................................................................. 88
  5.4.1 Evaluation of Adoption Strategies ............................ 88
  5.4.2 Vulnerability detection ........................................... 89
  5.4.3 Vulnerability identification ...................................... 92
  5.4.4 Vulnerability correction ......................................... 93
  5.4.5 Feedback & Observations ....................................... 95
5.5 Discussion ............................................................. 97
5.6 Conclusions ............................................................. 98

List of Figures .......................................................... 107

List of Tables ............................................................ 108

List of Code Examples .................................................. 110
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, some companies view security as a requirement that is taken care of at the end of a products development. Instead of integrating security in every phase of development some development organizations hope security can be added cheaply at the end. This is often done by performing penetration testing release ready software. The disadvantage with this approach is the higher cost of correcting the vulnerabilities so late in the 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 with the lowest cost possible, 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 were 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).
Chapter 1. Introduction

It is therefore necessary to evaluate and study the different methods and tools to determine their advantages and problems. Studies have also indicated that implementation vulnerabilities are the most common type of vulnerability, although these studies base their conclusions on vulnerabilities databases (Baker et al. 1999). Thus, it is possible that design vulnerabilities are more numerous and harder to find. Even so, this thesis focuses on vulnerabilities that are created during implementation in the software’s source code. To detect these implementation vulnerabilities, a technique called static code analysis is used. Several research and commercial tools perform variable degrees of sophisticated static analysis.

This thesis evaluated the potential of using static code analysis as an early vulnerability detector. Several case studies were conducted in an industry setting to investigate the capabilities and limitations of static code analysis. Our research questions are explained in greater detail in Section 1.1.

In section 1.1 the background knowledge of software development, vulnerabilities, and static analysis is explained. The section is mostly based on related work, except for section 1.1.3 that explains our view of early vulnerability detection. Then in section 1.2 the thesis research method is explained in detail. What methods are used, how the results are validated, and what taxonomy is used in the rest of the sections is explained in this section. Section 1.3 then summarizes the contributions of the thesis.

1.1 Concepts

In this section related work and the concepts of vulnerabilities, early detection and static analysis are explained.

1.1.1 Software Development

During development of software faults and flaws are introduced either from the implementation 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 availability. Unfortunately, testing mainly focuses on verifying the intended functionality and not on detecting vulnerabilities.

Figure 1.1 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 1.1, or the implementation might add unwanted functionality, labeled as B in Figure 1.1. Testing and validation has most of its 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 penetration 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 5.1 shows a layout of the security touchpoint development process (McGraw 2006).

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). Several of these processes recommend some sort of source code audit during the implementation phase of development. 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 any existing 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 products entire development lifecycle (McGraw 2006).
Chapter 1. Introduction

1.1.2 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 2002). 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 behavior or error. While the static analysis tool is searching for the fault or bug, testers are looking for the propagation of the fault. Figure 1.3 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.

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 labeled
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.

Figure 1.3: Origin and propagation of vulnerabilities.
Chapter 1. Introduction

1.1.3 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 countless times 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 are 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.

1.1.4 Implementation Vulnerabilities

All software projects produce at least one common artifact, 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.

**Buffer Overflows**

A buffer overflow is often the undesired result of incorrect string manipulation in the standard C library. When reading strings from the user it is the programmer’s responsibility to ensure that all character arrays are large enough to accommodate the length of the strings. Unfortunately, programmers often use functions such as `strcpy()` and `strcat()` without verifying that the buffer will fit. This may lead to user data overwriting memory past the end of an array. Below is an example of a buffer overflow that is caused by unsafe `strcpy()` usage:

```
Listing 1.1: Buffer overflow example.
1: char dst[64];
2: char *s = read_string();
3: strcpy(dst, s);
```

The string `s` is read from the user on line 2 and can be of any length. The `strcpy()` function copies it into the `dst` buffer. If the length of the user string is greater than 64, the `strcpy()` function will write data past the end of the `dst[]` array. If the array is located on the stack, a buffer overflow can be used to overwrite a function return address and execute code specified by the attacker. However, the mechanics of exploiting software vulnerabilities are outside the scope of this thesis and will not be discussed further. For the purposes of vulnerability detection, it is sufficient to assume that any vulnerability can be exploited by a determined attacker.

**Format String Bugs**

Format string bugs have been found in mature well known products, this include the Apache web server, wu-ftpd FTP server, OpenBSD kernel and many others. The format string bug arises when data received from an attacker is passed as a format string argument to one of the output formatting functions in the standard C library, commonly known as the `printf` family of functions. These functions produce output as specified by directives in the format string. Some of these directives allow for writing to a memory location specified in the format string. If the format string is under the control of the attacker, the `printf()` function can be used to write data to an arbitrary memory location. An attacker can use this to modify the control flow of a vulnerable program and execute code of his or her choice. The following example illustrates this bug:
Chapter 1. Introduction

Listing 1.2: Format string example.

1: char *s = read_string();
2: printf(s);

The string s is read from the user and passed as a format string to printf(). An attacker can use format specifier such as "%s" and "%d" to direct printf() to access memory at an arbitrary location. The correct way to use printf() in the above code would be printf("%s", s), using a static format string.

Integer Overflows

A third kind of memory overflow vulnerability is the integer overflow. These vulnerabilities are harder to exploit than buffer overflows and format string bugs. However, they have been discovered in OpenSSH, Internet Explorer and the Linux kernel. There are two kinds of integer issues: sign conversion bugs and arithmetic overflows. Sign conversion bugs occur when a signed integer is converted to an unsigned integer. On most modern hardware a small negative number, when converted to an unsigned integer, will become a very large positive number. Consider the following C code:

Listing 1.3: Signed/Unsigned integer overflow example.

1: char buf[10];
2: int n = read_int();
3: if (n < sizeof(buf))
4:  memcpy(buf, src, n);

On line 2 we read an integer n from the user. On line 3 we check if n is smaller than the size of a buffer and if it is, we copy n bytes into the buffer. If n is a small negative number, it will pass the check. The memcpy() function expects an unsigned integer as its third parameter, so n will be implicitly converted to an unsigned integer and become a large positive number. This leads to the program copying too much data into a buffer of insufficient size and is exploitable in similar fashion to a buffer overflow.

Arithmetic overflows occur when a value larger than the maximum integer size is stored in an integer variable. The C standard says that an arithmetic overflow causes "undefined behavior", but on most hardware the value wraps around and becomes a small positive number. For example, on a 32-bit Intel processor incrementing 0xFFFFFFFF by 1 will wrap the value around to 0. The main cause of arithmetic overflows is addition or multiplication of integers that come from an untrusted source. If the result is used to allocate memory or as an array index, an attack can overwrite data in the program. Consider the following example of a vulnerable program:
Listing 1.4: Integer overflow example.

```c
1: int n = read_int();
2: int *a = malloc(n * 4);
3: a[1] = read_int();
```

On line 1 we read an integer n from the user. On line 2 we allocate an array of n integers. To calculate the size of the array, we multiply n by 4. If the attacker chooses the value of n to be 0x40000000, the result of the multiplication will overflow the maximum integer size of the system and will become zero. The malloc() function will allocate 0 bytes, but the program will believe that it has allocated enough memory for 0x40000000 array elements. The array access operation on line 3 will overwrite data past the end of the allocated memory block.

**Null dereferences**

A null-pointer dereference occurs when a pointer with a value of NULL is used as if referring to a valid memory address. This operation causes a null-pointer exception or a program segmentation fault. The effect of null dereferences are often denial of service attacks but in some cases the software’s exception handling might reveal sensitive information.

Listing 1.5: Null-pointer dereference example.

```c
1: char *var = getenv("VARIABLE");
2: strcpy(buffer, getenv("VARIABLE"));
```

On line 1 we use `getenv` to read an environment variable and then we store its memory location in the char pointer var. If the environment variable is not set, `getenv` will return a NULL. In line 2 we do not verify that var is not NULL and then use it in a `strcpy` operation. The software would segmentation fault and terminate.

**Time of check to time of use**

Time of check to time of use are a type of race conditions that occur between the time a resource is checked and the time the resource is used. The time frame can be very small and still be exploitable. The vulnerability has often been used in UNIX systems to escalate local user privileges.

Listing 1.6: Time of check time of use example.

```c
1: if (access(file, R_OK) != 0)
```
This vulnerability has often been used in combination with setuid programs, program that are executed with higher privileges than the user that started the program. On line 1 the `access` operation verifies that the actual user has permission to access the file and if that check is true line 2 is ignored. Then on line 3 the program opens the file. There is however a slight delay between the verification and the opening of the file. During this short time-span the checked file can be swapped with a symlink that points to a privileged file. The program would then open and use the privileged file instead.

### 1.1.5 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 use 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 flow-sensitive analysis uses the programs control flow while performing an analysis, while a flow-insensitive analysis does not. A flow-sensitive analysis is usually more precise, it may infer that $x$ and $y$ may be aliased only after certain line, while a flow-insensitive analysis only infers that $x$ and $y$ may be aliased anywhere. However, a flow-sensitive analysis is usually more time consuming.

A path-sensitive analysis considers only valid paths through the program. It takes into account of values of variables and boolean expressions in conditionals and loops to prune execution branches that are not possible. A path-insensitive analysis takes into account all execution paths including infeasible ones. Path-sensitivity usually implies higher precision but usually requires longer analysis times.

A context-sensitive analysis takes the context, e.g. global variables and actual parameters of a function call, into account when analyzing a function. This is also known as inter-procedural analysis in contrast to intra-procedural analysis, which analyses a function without any assumptions about the context. Intra-procedural analyses are much faster, but suffer from greater imprecision than inter-procedural analyses. Path- and context-sensitivity rely on the ability to track possible values of program variables. If we do not know the values of the variables in the boolean expression of a conditional, then we do not know whether to take the 'then' branch or the 'else' branch to get the correct data flow.

Another important issue is aliasing. When using pointers or arrays the value of a variable can be modified by modifying the value of another variable. Without a careful value and aliasing analyses we will typically have large numbers of false positives, or one has to do ungrounded, optimistic assumptions about the values of variables. The un-decidability of runtime properties implies that it is impossible to have an analysis
which always finds all defects and produces no false positives.

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 than research tools do.

Because of the focus of this thesis in a specific type of faults, vulnerabilities, we categorize 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 untrue fact. These are often caused by 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 1.1.2.

- **Security Positive**: Warnings that are correct and can be exploited into any of the four effects in section 1.1.2.

- **False Negative**: Known vulnerabilities that the static analysis tool did not report. 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.

## 1.2 Research Approach

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

### 1.2.1 Research Questions

Software development departments in Ericsson AB aim to improve their software’s security before it is sent to testing and verification. To achieve this goal, static analysis was proposed as a solution. However, to determine if static analysis is a plausible solution this thesis intends to answer some crucial research questions:

1. *What are the capabilities and limitations of static code analysis, for early vulnerability detection?*
For a technique to be an effective and efficient as an early vulnerability detector, it is required not only to detect vulnerabilities it is preferred to do so in a short time-span using uncompleted code. Because this is our primary research question all Chapters contribute to its answer.

2. To what extent does static analysis reduce existing security maintenance cost?
Software security is seldom a sellable feature and is instead expected. It is therefore important that the process changes either have a minimal cost impact, in lead-time or direct financial benefit, or alternatively the tools can have a cost saving effect that nullifies any added overhead to the development process. This question is answered in Chapter 3 were we compare static analysis to known vulnerabilities.

3. How should an early vulnerability detector be integrated into an already existing development process and what additional integration problems are created in the process?
A static analysis tool requires human involvement and therefore the developers’ interaction and judgment creates a problem source. The tool can be deployed in several different ways and its data can be used and analyzed differently. The human factor is explored in Chapter 4 and in Chapter 5. In Chapter 5 we also examine the deployment of a static analysis tool in several industry projects.

This thesis answers the above question through case studies at Ericsson AB. By answering the above questions, the strengths and weaknesses of static analysis as an early vulnerability detector should be explored.

1.2.2 Research Methods
The research objective of this thesis is to study and evaluate methods and tools in practical industrial context, in this case the usefulness of static code analysis as an early vulnerabilities detector. 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.

Survey: 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 Chapter 4 surveys were used to collect developer experience data and in Chapter 5 surveys were used to
Chapter 1. Introduction

determine how the different projects had deployed and used the static analysis tool in a real setting.

Experiment: Experiments are controlled studies that are designed to test one specific 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 typically address a limited scope (Kitchenham et al. 1995).

Controlled experiments were used in both Chapter 4 and Chapter 5 to examine a specific variable. In Chapter 4 the developers' experience was compared to their ability to correctly classify warnings reported by the tool. In Chapter 5 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).

In Chapter 2 and in 3, 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.

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 Chapter 3 to examine how effective static analysis could detect already known vulnerabilities and what cost savings would have been possible if static analysis had been used. In Chapter 5 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.

1.2.3 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 change. 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 of static analysis results 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, throughout 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. However, the results still focus heavily on server software written in the C and C++ language.

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. However, in Chapter 4 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.

### 1.2.4 Research Environment

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 1.4

*Design and Implementation:* 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
Chapter 1. Introduction

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. While the length of the implementation phase got shorter, the impact on static analysis was negligible or nonexistent. The most significant impact with a shorter implementation phase is the need to have a short analysis time, for static analysis the analysis time is often the same as the compile time. As such, it is not heavily effected by time shortage, dynamic analysis that require more time would be more effected by the change of development process.

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.

The case studies only focus on software written in the C and C++ language. There are two reasons for this limitation, first the available security checkers for Java programs are limited in comparison to the C/C++ checkers, and secondly the nature of the Java code in Ericsson products is often as a GUI. Because all the programs functionality resides server side, the GUI did not have any code that could create implementation vulnerabilities.
1.2.5 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.

1.3 Outline and Contributions

This section presents the contributions of this thesis. Each subsection represents a publication and its contributions. In short, Subsection 1.3.1 presents the possibility of early vulnerability detection in even mature well-tested software. In Subsection 1.3.2 the capabilities of detecting known vulnerabilities are explored. Subsection 1.3.3 focuses on the developers and their needed experience to utilize the static analysis tool and in Subsection 1.3.4 the result of an extended industry case study shows the strengths and weaknesses of actually using a static analysis tool as an early vulnerability detector.
Chapter 1. Introduction

1.3.1 Chapter 2

This chapter presents a proof of concept, where early and simpler static analysis tools are used to examine a mature well-tested product. The purpose of the chapter is to determine if basic static analysis technology is useful as an early vulnerability detector and to partly answer research question 1. Instead of using laboratory code, the study was done on a released product that had no known security faults. Therefore, the product first had to be examined with several static analysis tools. Then the output from the tools’ (RATS, ITS4, Flawfinder) was analyzed to determine how similar they were. In this case, there was considerable similarity in the tools’ result and at least for the simpler tools’ it is clear that the choice of tool had little impact on the result. Because the product had no known faults, all the reported warnings had to be examined and a group of possible vulnerabilities was created. To verify that the tools’ findings were actual vulnerabilities and not only possible threats, several proof of concept attacks were performed on the examined product. This resulted in three possible attacks on the product:

1. A remote buffer overflow that was exploitable into remote shell access.

2. The possibility to sabotage the systems through a race condition that destroyed log files and history data in the system.

3. A decryption attack that used poor random number generators to break encrypted messages sent between the products and legitimate clients.

While the study showed the benefit of even the simplest static analysis tools, it also showed the weaknesses in the technology. Because static analysis operates on the source code, the tool has to determine if the warning is a real threat or not, early tools did no verification and therefore have a high number of false positives. In this study less than 10% of the reported warnings were useful. Developer would have to discard 9 out of 10 warnings as false positives before finding one real threat. Those numbers are not acceptable as most developers would just ignore the tool’s output. The second problem was the sheer number of warnings, in just 100.000 lines of code there were more than 800 warnings. The warnings also presented very little information and analyzing the result was time consuming. It was calculated that a minimum of 16 working days was put into examining the warnings.

However, in this chapter it is shown that even simple static analysis tools can detect exploitable vulnerabilities and it can detect them in mature well-tested software. It is shown that theoretically, static code analysis can be used as an early vulnerability detector and that industry can benefit financially and with more secure and robust products.
1.3.2 Chapter 3

The main contribution in this chapter is the comparison between a static analysis tool and known reported vulnerabilities. Therefore, answering research question 2 and contributing to question 1. To improve on the weaknesses of the previous chapter a state of the art static analysis tool is used instead. The tool (Coverity Prevent) was selected from an initial review where most of the leading tools were examined. In this pre-study it was determined that the result of most tools were similar even if the initial warnings looked different. In addition, the number of false positives varied between the tools. The study also used several products to generalize the results and increase the validity.

The study was performed on source code that was at least one year old and had since release several known vulnerabilities in the code. These vulnerabilities also had an average cost associated to them so a possible cost reduction was calculated. By analyzing the code and examining the warnings we determined how many of the known vulnerabilities could be detected during implementation and therefore save the paid maintenance cost. We also found new vulnerabilities that were previously unknown and laid dormant in the code. Because the static analysis tool detected more than just security faults, we also examined all the functional warnings but did not compare them to known bugs.

In the three examined products, we saw that the number of false positives was much lower in state of the art tools compared to the previous chapter. On average, the tool produced a false positive rate of about 20%. For an early vulnerability detector this false positive rate is theoretically acceptable because it would not deter the developers in a significant way. The false positive rate could further be lowered by removing weak checkers or by modeling the tool to better understand the code. However, the majority of the detected warnings were not security faults but coding improvement and functional faults. Only about 5% of the warnings were security related. The percentage of security warnings could be increased by removing checkers that could not detect security faults. Because faults and vulnerabilities look the same for a static analysis tool, getting the detection rate to 100% security faults is probably not possible.

An interesting finding in this study was the large number of dormant vulnerabilities that were verified as user exploitable vulnerabilities. Most of them were memory related, either from input validation errors or from API abuses. In the end, 59 new previously unknown vulnerabilities were discovered. Of the already known vulnerabilities only about 30% were detected by the tool and all of them were memory related. This showed the strength of the tool, but also its weakness in detecting non-memory related vulnerabilities. Even so, the study showed the tools’ usefulness as early vulnerability detector. If these three projects had used the tool during implementation, they would have spent about 28% less time on bug correction and at the same time would have had
Chapter 1. Introduction

a more robust and higher quality product.

### 1.3.3 Chapter 4

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 3. Because the tool requires that developers 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 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 then 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.

1.3.4 Chapter 5

This chapter examines two years of industry experience with static analysis tools (Coverity Prevent) and contributes to research question 1 and 2. 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, 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 vulnerability detector.

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
Chapter 1. Introduction

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 then 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.

1.4 Conclusions

Research question 1: Capabilities
Throughout this thesis, we have shown that static analysis is capable of detecting some types of implementation vulnerabilities. Most of the detected vulnerabilities were classified as input validation and representation faults and a majority was caused by different memory overruns. Static analysis was capable in detecting vulnerabilities that required a precise execution flow to be exploitable. However, we identified several instances where known implementation vulnerabilities where not detected by the static analysis tool. At the same time the tool did detect new vulnerabilities that where not previously known. Because the tools are often not sound e.g., there can be false negatives, a specific type of vulnerabilities cannot be excluded and penetration tests still
have to look for all types of vulnerabilities. Tool vendors have seen this as a necessary
sacrifice to ensure that the tool has an acceptable analysis time. The analysis time is
indeed an important factor for an early vulnerability detector and is more important
than the completeness of the analysis. False negatives therefore seem to be a necessary
limitation with static analysis tools.

Research question 1: Shortcoming
From the simpler static analysis in Chapter 2 to the more sophisticated analysis in the
subsequent chapters, the false positive keep creating problems. First, they create an
overhead by having to examine warnings that are no faults at all. But more disturbing
the observation in Chapter 5 were the correction of false positives actually created new
vulnerabilities in code that was previously secure, raises new concerns about the tech-
nologies capability as early vulnerability detector. Further studies should be conducted
to determine the full impact of the false positives.

Research question 2: Cost saving
Within the context of our studies we have also shown that static analysis could lower
costs by detecting the vulnerabilities during implementation instead of later phases of
development. With the simpler static analysis tools we show in Chapter 2 that the ex-
amined project could have detected 54 security improvements during implementation.
Because the faults could have been detected early they would have been less costly
to detect and correct compared to later phases of development. More concretely, in
Chapter 3 we explicitly evaluated the tools cost saving and its effectiveness in detect-
ing known vulnerabilities. In the three examined products the tool detected 30% of the
known vulnerabilities, when comparing tool usage cost with the known average cost to
correct the vulnerabilities, we saw a 17% possible cost reduction for reported vulner-
abilities. Also new vulnerabilities that had not been detected by testers or customers
where found by the tool. These dormant vulnerabilities could have been exploited by
malicious users and damage the company’s reputation.

Research question 3: Developer interaction
As an early vulnerability detector, the tools proved capable in finding some vulnerabil-
ities and any developer could, in the majority of cases, correct the security fault cor-
rectly. However, classifying the vulnerabilities proved harder than expected and many
developers did not understand that the warning was actually a security threat, increasing
the number of false negatives and faulty corrections. Similarly, the developers could
not classify the false positives and often corrected faults that did not exist and in ex-
treme cases created new vulnerabilities. We did not observe any improvement in using
the tool with increased general software development experience, instead only specific
experience in using the static analysis tool improved the developer’s results. Because
of the complexity of understanding the code and the warning, developers could not
determine how accurate there interoperation of a warnings was and we found no corre-
lation between the correct interpretation and developers perception.

Research question 3: Deployment overhead
The deployment of the tool also created problems, as can be expected for all develop-
ment tools. Even though the tool could provide benefits to developers it was often seen
as an obstacle and something developers avoided. As such, the tool’s true potential that
was shown in early case studies was not replicated in post- mortem analysis after the
tool was used in development. For a successful deployment the majority of the exam-
ined projects had to use overhead operations to assure that the tool is used correctly.
This lessened static analysis efficiency as an early vulnerability detector and increased
the cost of using the tool.

1.5 Future Work

This thesis has focused on C language code that was written for server software; it
could easily be expanded to other types of software to determine how effective static
code analysis would be in a different context.

The false positives are creating severe problems with the static analysis tools, they
create overhead but more important they introduce the notion that the tool can be in-
correct. Developers therefore assume that already corrected code that still produces a
warning is a false positive. However, the observed behavior has shown that the correc-
tion often is faulty and the warnings are real vulnerabilities. Added with the possibility
that a false positive correction can create new vulnerabilities more research should be
put into removing or minimizing the false positive rate even further. Early static anal-
ysis tools showed an about 90% of the warnings as false positives with more advanced
tools lover it to about 20%. This is, however, still to large.

Most static analysis checkers focus on implementation faults. Further research
could examine if static analysis techniques can be used to detect design flaws. Even
if detecting design flaws after implementation might not sound like a good early vul-
nerness technique it has some benefits. Current design vulnerability methods are
time consuming and expert dependent. The results from the design analysis are not
always valid as the implementation of the design might vary greatly from the initial de-
sign. Therefore, the examination of the design with the aid of static analysis techniques
might be more cost effective and accurate than an early design vulnerability method.
Chapter 2

Software Security Analysis – Execution Phase Audit

Bengt Carlsson and Dejan Baca

2.1 Introduction

During recent years software developers have changed focus from only reliability measurement to include aspects of security threats and risks. Both security tests and function tests look for weaknesses but not necessarily at the same time. By definition a reliability threat or test will sooner or later manifest itself, while a security threat may remain undetected. Recently static analysis tools have been used to automate source-code security analysis by measuring the amount of weaknesses or vulnerabilities at hand. As a result, improved error-free code for future versions of the application may be developed.

Manual audit done by experienced programmers is a time consuming but otherwise efficient method for conducting secure code revision of software. The main reasons for introducing automated auditing tools are to decrease manual audit time and to integrate automatic tools as part of a revision update. The first issue has its background in program checkers (Johnson 1978) followed by several generations of automated auditing tools, from rule based to more flexible context based. Static analysis tools use a database of keywords to find vulnerabilities and output a vulnerability report by doing a syntactic matching (Wagner et al. 2000). These tools report a large number of false pos-
itives since they lack a deepened context analysis, and therefore manual examinations to exclude the false positives are necessary. More recent global analysis tools perform an analysis of program semantics (for an overview see (Chess and McGraw 2004a)) in an effort to minimize the amount of false positives. The second issue is the possibility to integrate automated tools into the software development life cycle as improvement and effectiveness factors. Development cost savings are part of a more general return on investment (ROI) calculation for the investigated project. ROI involves factors for identifying the context of the life cycle, techniques for setting up the measurement, the actual requirement and comparison of the software, and finally the set of measures involved. So, the obvious calculation of number of bug fixes should be supplemented by calculating revision time, to complete the development cost discussion.

This study discusses what sort of security flaws that have been found, how they can be avoided and how to use automated tools to increase the level of security during software revision. In Section 2, different security exploits are described and in Section 3 the product and its development process is explained. The investigation, described in Section 4, uses different automated auditing tools combined with manual auditing applied on a real telecom application. The results, in Section 5, compare the effectiveness of different automated auditing tools applied on the mobile telecom server software. The development cost is discussed as an improvement on the original investigation. In the discussion section open and close source software are compared followed by a conclusion.

2.2 Securing unsecured software

Like most commercial applications, the code investigated is a closed source software project. The main issue in general is to protect the software from being copied, consequently it is difficult for an outsider to find and correct security vulnerabilities. In contrast to close source, open source is released publicly and interested users can perform audits and post corrections.

Information about closed source security bug-tests is insufficiently discussed in the literature whereas classification of various security taxonomies is abundantly discussed (see (McGraw 2004) for an overview). Some security vulnerabilities occur repeatedly on both the examined application and when exploring different security sites. Recent software auditing of security vulnerabilities in Windows NT with a susceptibility matrix classification is presented by Jiwnani and Zelkowitz (Jiwnani and Zelkowitz 2004). Unlike this investigation, the result presented here is based on a limited subset of all detectable security flaws.

Four common types of security flaws that often arise in software are described
Buffer overflow - a study presented by Wagner et al. (Wagner et al. 2000), showed that as much as 50% of all large exploits were buffer overflows. Buffer overflows occur when unsafe programming languages like C and C++ are used. The insecurity of C and C++ are a consequence of not performing any bound checks on arrays or pointer references. A buffer overflow occurs when the amount of data exceeds the size of the buffer. There is a lack of bound checks to confirm that the data will fit in the buffer. This can result in several dangerous function calls in the C programming language and lead to a buffer overflow.

Misplaced trust - when developing new software there is always the question of whom to trust and what input to consider safe. Misplaced trust creates many security risks and is often the base of exploitable code. During the investigation most of the misplaced trusts was found in the input validation. Often, the developers consider input data from in-house software as safe and therefore no input validation is performed. They may try to hide information inside the code, e.g. cryptographic keys and other in-house authentications (Viega and McGraw 2002). With programs like IDApro (ida 2004) that helps reverse engineer binaries, an attacker can recreate the design of the software and find any hidden algorithms. Placing the trust in client software and not on the server side also creates a security issue, i.e. it is possible for the end user to circumvent the client and any client validation.

Race condition - in a multi thread environment there is always a probability of a race condition. Because of multi threading, the program can perform the same or different tasks at the same time. Most race conditions are not security vulnerabilities but instead stability issues. Race conditions are becoming a bigger problem and they are often hard to correct even after they have been found. To avoid race conditions the developer has to remove the time between verifying a resource and using it. Even in microscopic timeframes a race condition can be exploited (Viega and McGraw 2002).

Poor random number generator - a simple routine like generating a random number is a daunting task for a computer. Developers use non-true random calls where the results can be predicted. From a security point of view, random numbers are important when dealing with encryption. Without a good random number the cryptography can be predicted and broken. Most UNIX systems have their own built in random number generator.
Chapter 2. Software Security Analysis – Execution Phase Audit

2.3 Product and development process

The results of this paper were obtained through an investigation at a software development department at Ericsson AB. The most recent stable version of a telecom product was used as a case study. The examined product had a component-based architecture and was built on a shared platform, which allows use and reuse of the same component architecture. Only the C++ components were examined during the investigation but the product also used HTML and Java for GUI. For communication the product and the components used a common socket connection interface and all the data was transmitted in XML format. The product runs on a SPARC/Solaris 8 platform, a server application composed of approximately 100,000 lines of code. During every production cycle the product passes several tests but no source code security analysis had been undertaken. Figure 2.1 describes the principal steps within the investigated project.

The development department had a project cycle of 1-1.5 years with an average of 50 participants. After an initial demand the design structures were decided. If security gaps, as a result of design errors, were undetected no later analysis would typically capture these deficiencies. After the design (feasibility) was completed, the product was divided into different function parts. Execution and function tests were then done before being reunited at the system level again. No security specific tests had been done through the development process. During every phase of the development cycle measures were taken to detect faults in the code, e.g. basic tests during the execution and automated tests, TTCN (tte 2005), during function and system test. Damm et al. (Damm et al. 2004) performed a case study at the same development department and concluded that a fault had almost a twenty fold increase in cost if it was found during system tests instead of the execution test. Since neither function nor system test had any specific security tests, any security flaw introduced during the execution would likely pass unnoticed, e.g. it will not be detected until the product had ended its development cycle.

Figure 2.1: Program progress and security analysis
2.4 The investigation

During the investigation both an automatic and a manual audit of the source code were performed. The static analysis tools were used in the execution part as pure code revision tools. The possible exploits that appeared in the audit were confirmed or dismissed in the practical test environment. The proof of concept investigation used the compiled code at the system level to show the effect of the security risks. The purpose of the manual source code auditing was to reveal the security risks in the code. Reading and understanding the source code was time demanding and relied on the analysts knowledge in the programming language and the system the software is used on. In order to find these security vulnerabilities the analyst had to understand the source code and make sure it did what it was designed for and nothing more. Since no prior security test had been undertaken, the automatic auditing tools were used to examine vulnerabilities from a large database rather than performing a semantics analysis. The comprehensive task was to find methods and rules for integrating security analysis into program revision on closed source software. The automatic auditing of the source code was performed with RA TS, a tool for static analysis. To confirm the findings of RA TS, the tools Flawfinder and ITS4 were used. Any differences between the tools findings were noted. Confirmed threats and possible security issues were examined and categorized. The outcome was divided into the following categories:

1. False positives. These warnings, generated by the automated auditing tool, do not pose any risk at all.

2. False negative. All vulnerabilities, found during the manual auditing, not reported by the automated auditing tool.

3. Possible security improvement. A possible security risk without confirmed consequence, i.e. the code is not properly written but other parts of the program will take care of the vulnerability.

4. Security risk with consequence. A security risk that had a consequence and was confirmed in the test environment.

The purpose with these categories was to separate different types of threats, aiming to find the most risky ones. To remove the false positive warnings, each warning were examined and determined if it could pose a threat or not. All inspected threats found during the manual examination were labeled as possible security improvements. Finally, all security warnings confirmed in the practical test environment were reexamined to be certain that no new or further exploitation was possible.
Chapter 2. Software Security Analysis – Execution Phase Audit

The security warnings were also examined at a system level; why they had occurred, how they could have been avoided and what kind of threats they posed. This re-examination was used to create a guideline for how the threats could be avoided in the future and how to avoid insecurities while developing new software. Besides checking all the warnings that were generated by the audit tools, a manual audit of selected components was conducted. The combined size of these components represents 10% of the overall source code used by the product. The purpose of the manual audit was to examine the effectiveness of the static analysis tools. Three security risks were chosen and proof of concept programs were written to show the effect of the security risks. The three security risks were buffer overflows, race conditions and poor random generators. The three proofs of concept programs had different goals in their attack; siege the system, sabotage or espionage.

2.5 Results

The findings were divided into four parts. First the efficiency of the automated tools, was investigated. Secondly, these findings were applied on the telecom product where the security risks are categorized into four groups. Third, different proof of concepts programs illustrated the possibility to create attacks. Finally an estimation of revision time and effectiveness was done, i.e. a ROI factor was decided.

2.5.1 The Static analysis tools

The different tools showed a mixed list of warnings depending on what level they were configured to report, lowest, standard or highest. Table 2.1 below, showed the results from the tools at the different levels. At the lowest level every warning was reported compared to the highest level where only the most dangerous warnings should be reported.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Lowest</th>
<th>Standard</th>
<th>Highest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flawfinder</td>
<td>587</td>
<td>587</td>
<td>0</td>
</tr>
<tr>
<td>ITS4</td>
<td>504</td>
<td>310</td>
<td>36</td>
</tr>
<tr>
<td>RATS</td>
<td>520</td>
<td>306</td>
<td>242</td>
</tr>
</tbody>
</table>

Flawfinder was the tool that showed the highest number of warnings at the standard level, in fact, it showed all the warnings it could find. Flawfinder found one unique
warning that neither RATS nor ITS4 found. Flawfinder did not find a single insecurity dangerous enough to be labelled at the highest level. ITS4 had the lowest overall report of warnings and worst correct findings of security risks. At the highest level, ITS4 reported 36 warnings where none of them was reported by the other tools at this level. All the warnings reported at the highest level in ITS4 were correctly reported by the other tools at a lower level of danger. RATS was the tool with the least configurable levels of warnings. As a result RATS had the largest number of warnings, 242, at the highest level. All the warnings at the highest level were also unique when compared to the other tools at the highest level. RATS reported altogether 520 warnings of which 101 were unique.

2.5.2 Manual Examination of the Automated tools findings

All warnings reported by the automated auditing tools were manually examined. By doing a manual auditing on entire components, instead of auditing warnings separately, the automated tools insecurities could be better verified. During this manual component audit no new security risks were found, i.e. the security risks had already been reported by the automatic tools. So, from the reported warnings by the three tools, a manual examination determined if the warnings were possible security improvements or if they were false alarms. Together the tools reported 823 unique warnings. The manual examination of these warnings was a time consuming task, resulting in the findings in Figure 3. In all there existed 59 possible security improvements (7.2% of the total warnings) when combining the different automatic tools with manual auditing. Of these 25 represented a security risk with consequences, i.e. almost half of the possible risks were real security threats. For the entire application there existed one possible security improvement per 1700 lines of source code. For every 4000 lines of source code there existed one security risk that could be exploited.

RATS had the best results with just one false negative; this false negative was found by ITS4. RATS was also the only tool that found possible security improvements at its highest level, where 80% of all the possible security improvements were found.

All automated tools checked had false negatives because of deficiencies in the vulnerability database. Two reasons were found, either the risks were not present in the vulnerabilities database or the tools where not capable to perform the analysis required for finding these false negatives. The tool with the lowest number of false negatives was RATS. A missing registry in its database caused RATS single false negative. The false negatives in Flawfinder were also missing vulnerabilities in the database combined with deficiency inside the database, Flawfinder did not report known vulnerabilities. The false negatives from ITS4 were also missing vulnerabilities, but the larger portions were deficiency reporting fixed local buffers that were not used properly.
2.5.3 Security vulnerabilities

The security risks in the code were categorized into four groups: buffer overflow, misplaced trust, race condition and poor random generators, see Figure 2.3. Security risks with unsafe buffer handling were placed in the buffer overflow group. Incorrect input validations and the lack of validations were placed in the misplaced trust group. The race conditions were security risks where an attacker would be able to alter the information the software was going to use. Predictable random generators in a security function were placed in the poor random generator group. This was also the smallest of all groups.

2.5.4 Proof of concepts

As already mentioned, three security risks were chosen and proof of concept programs were written to show the effect of the security risks.

- Siege the system with a buffer overflow: the buffer overflow created a opportunity to crash the program and create a denial-of-service attack. It also created the possibility of remote shell access, a far more dangerous risk. The proof of
concept program created a remote shell. By overwriting the return address, the software was redirected to the injected machine code and a shell was bound to a specific port. With this shell access, an attacker would have had access to the system and gateway into the rest of the systems in the network. Because of the dangerous results from buffer overflows they were considered as the most severe security risks.

- Sabotaging the system with a race condition: the race condition made it possible for an attacker to damage the system and more importantly, the ability to cover any track of the intrusion. The proof of concept program exploited a race condition present in the product to force the software to overwrite and delete log files. It is the software itself that destroys the log files, an Intrusion Detection System (IDS) would not detect the intrusion.

- Conducting espionage on the system due to poor random generators: the poor random generator created the possibility to break the encryption, thus enabling an attacker to read any information important enough to encrypt. The proof of concept resulted in a decryption and encryption tool that may be used for breaking whatever encrypted data the attacker wants. The exploit made it possible to

Figure 2.3: Proportions of different vulnerabilities
read other customers encrypted data and change them.

2.5.5 Return on investment

Since the automated tools used during the audit were open source tools, there was no significant investment cost to acquire the automated tool. There are more powerful tools that are not free and are better at finding security issues but for this article these highend tools were excluded. The remaining investment cost is usage of the tool and learning the tool. The manhours used to audit the findings of the tool are the only large investment cost worth calculating. So, the ROI depends on the ease of use and the targeted users, e.g. making the learning cost negligible compared to the overall project study costs.

<table>
<thead>
<tr>
<th></th>
<th>All warnings</th>
<th>RATS total</th>
<th>RATS standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warnings</td>
<td>843</td>
<td>520</td>
<td>306</td>
</tr>
<tr>
<td>Work days</td>
<td>42</td>
<td>27</td>
<td>16</td>
</tr>
<tr>
<td>Security warnings</td>
<td>100%</td>
<td>98%</td>
<td>91.5%</td>
</tr>
<tr>
<td>Improvement/day</td>
<td>2.38%</td>
<td>3.36%</td>
<td>5.72%</td>
</tr>
</tbody>
</table>

In the investigation 843 warnings were examined, to fully comprehend the warning the surrounding code had to be understood. A total of 42 workdays were spent on manually investigating the warnings. This figure can be reduced if only one tool is used at a higher level. By only using RATS and at a standard level, RATS would have found 91.5% of all of the possible security improvements with 16 workdays of examination done by developers unfamiliar with security issues. The improvement per day was more than doubled compared to investigating all warnings. For a relative small increase of time during the execution phase, it would be possible to lower the amount of undetected security faults. On the examined product the execution phase lasted approximately 720 days, when the time spent by all developers is combined. With an extra 16 days, used to review the source code with the assistance of RATS, a maximum of 54 security related faults were detected, resulting in decreased development costs. Since the 54 faults found by RATS were security related, neither function nor system tests were able to detected them. Consequently, the faults would have occasionally been detected by the product users, resulting in a twenty fold cost increase to correct the discovered faults. The code reviews during execution are done by the developers themselves, and there are no records or calculations of the corrections made. Therefore it was not possible to determine the exact gain from introducing static analysis tools.
into the execution phase.

### 2.6 Discussion

Manual audit done by experienced programmers is a time consuming but efficient method for conducting secure code revision of software. By comparing manual and automatic audit it was possible to estimate both the degree of audit tool improvements, and the deficiencies of program developers. The chosen rule based audit tools were not effective in finding security risks, but instead they showed where the manual auditing should be focused. The tools saved time and eased the burden on those performing the security analysis. When looking at all 823 unique warnings from the three investigated tools, 7.2% probable security warnings were found. RATS was the tool with the least false negatives and the most unique possible security improvements. During highest level report, 80% of these improvements were found which constituted a success rate of 19.5%. Despite the low success rate, the cost to perform the audit with static analysis tools, was much lower then the fault correction cost the security issues would have costed if they were found later in the development process. When looking at all the warnings from all the three tools, the manual audit did not find any security risks that the tools had not already reported.

The open source project Apache (apa 2004) performed a similar test (apa 2002) with the tool RATS for about 110000 lines of code. The Ericsson application had after several years of use and bug fixes a total of 59 warnings that where related to security risks while the Apache project had, 12 warnings related to security risks. The apache project benefited from a large user base that has examined the code and made source code corrections. The Ericsson product had been restricted to the time available to the developers and had not undergone specific security testing, i.e. the initial security quality should for a true comparison be measured after the conducted test.

The security arguments for and against closed source code compared to open source code, are decreased exposure to risks and fewer persons involved in security improvements. At best, closed source developers have a security team or persons that examines the code. In principle, under the assumption of the standard reliability growth model, opening a system enables the attacker to discover vulnerabilities more quickly, but it helps the defenders exactly as much (Anderson 2002). More efforts on both sides imply exploring and correcting more vulnerability, but the probability of finding a new security failure is not altered. The mean time to failure depends only on the initial quality of the code and the time spent testing it thus far. This has been discussed for general code revision (Bishop and Bloomfield 1996) and security vulnerabilities (Rescorla 2005). The static analysis tool could help a project to discover its vulnerabilities by adding
Chapter 2. Software Security Analysis – Execution Phase Audit

a security audit to the project code review. In the investigated products, the Ericsson product benefited more from integrating static analysis tools to its development process. Industrial software projects have time and resource constraints, i.e. delivering on time with a restricted amount of money. Improving security revisions at the same time as ongoing function tests fulfill these requirements. By using RATS standard settings about 90% of all security warnings found, could be corrected. Due to programmers experience in the source code and recent exposure, the audit time could have been further reduced if conducted as part of an execution process.

Furthermore, if more examinations would be performed on the source code it is possible that even more security risks could be found at a greater cost than the 42 workdays. With large projects, the development of new source code could be too expensive for a full source code security analysis. With very large projects, even a good enough result could be too hard to achieve with the sparse resources spent on security. For a more precise ROI a new audit should be done during the execution phase on the softwares next project cycle. As a rule of thumb for an Ericsson project, the estimated cost for correcting vulnerability is increased twenty fold during a system test. This growth is mainly dependent on new persons involved for a function test and a new department involved for the system test, both generating heavy loads of people involved, new documentations and other overheads. Today the customers, buying the product, may also demand security tests aside of function tests before the product is given global availability. It is, as shown above, much cheaper to do this during an ongoing execution phase. The automated tools generate warnings where insecure code exists and the developer can then examine the warnings and correct the code or determine the warnings as false positives. If this is performed during the execution phase there will not be a great number of warnings to examine, compared to a complete examination of a finished product. The developer also learns from warnings and may avoid insecure code in the future. Since automated tools are fast, tests could be performed every day by sending new warnings to the developers for examination.

2.7 Conclusions

The audit of the Ericsson product showed one possible security improvement per 1700 lines of source code. Furthermore, there existed one security improvement that could be exploited every 4000:th lines of code. In all 823 potential security vulnerabilities were examined during 42 days of work. Half of the found threats in the product were buffer overflows followed by race condition, misplaced trust and to minor degree poor random generators. Static analysis tools were used to speed up the revision process and to integrate security tests into the project process. Different tools with different levels
of security warnings were examined where RATS at standard level setting produced the most efficient result. The results showed that RATS found 91.5% of all real security threats spending less than 40% of total time for finding them.

Bigger industrial projects usually include different subgroups for developing specific functions of the program in progress, before merging into the final system. All vulnerabilities with an action plan at the execution level are cheaper to handle than those vulnerabilities at the function level or exceedingly at the system level. Today, customers demand security tests aside of function tests, making an action plan for correcting vulnerabilities indispensable. This work shows that proportionately small investments improve the program code security, by integrating an automatic auditing tool into the execution of source code revision. For the softwares next project cycle a more precise return on investment calculation should be done focusing on optimizing the ratio between found vulnerabilities and time spent during the execution phase.
Chapter 3

Evaluating the Cost Reduction of Static Code Analysis for Software Security

Dejan Baca, Bengt Carlsson and Lars Lundberg

3.1 Introduction

Flaws in software design and faults/bugs in software code are constantly being introduced into programs during their development. If certain conditions are present, these faults and flaws can propagate during runtime into failures that might be exploited as vulnerabilities (Viega and McGraw 2002), see also Figure 3.1. Failures and especially vulnerabilities increase the cost for the developers and require more time to be spent on maintenance instead of new features. Many developers rely on automated testing tools to verify their software and at the same time reduce development time (Pretschner et al. 2005). Unfortunately, most of the testing is done to verify functionality and not to find vulnerabilities.

Today, telecom developers are trying to decrease development time and shorten time to market. It is therefore of interest to detect and correct a fault before the software has been tested. The earlier an fault is detected the less it costs to correct the fault (Damm et al. 2006) (Briand et al. 2000). So, developers that want to lower devel-
Chapter 3. Evaluating the Cost Reduction of Static Code Analysis for Software Security

Figure 3.1: The relation between source code faults and design flaws resulting in visible failures that might propagate into known or unknown vulnerabilities.

opment time and reduce the quantity of faults have to depend on early fault detection tools and methods; one of these is automatic static code analysis.

Automatic static analysis has been an area of early fault detection research for many years and has more recently been added to security engineering processes as an early tool for detecting vulnerabilities (Lipner 2004). Researchers aim to minimize the human time and complexity needed during source code inspections, i.e. to automate source code reviews. The quest to automate and aid the source code review processes started with simple program checkers (Johnson 1978) 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 developer to create more stable, reliable and secure code. The tools represent a varying degree of sophistication and some utilize complicated techniques such as abstract interpretation to achieve their goal of better automation. Different commercial vendors specialized their tools in areas such as security.

Several published papers have concluded that static code analysis tools (SAT) can detect code vulnerabilities (Chess and McGraw 2004b). However, the data are often laboratorial or based on assumptions that the reported warnings are indeed vulnerabilities. In this paper we have instead applied static analysis with reported vulnerabilities in large software systems and examined how effectively they are detected. In this case study we are trying to answer how effective static code analysis tools are in detecting vulnerabilities. These detected vulnerabilities present a cost saving opportunity for
the developers. We also examine what is the correlation between SAT findings and reported vulnerabilities.

Section 2 presents previous studies that have examined static code analysis for vulnerability detection. Section 3 explains how the static analysis tool works and the development process used by the examined software, clarifying the need for early fault detection. Section 4 explains how the case study was performed and what data had been analyzed. In Section 5 all the results are presented. Section 6 discusses the effectiveness of the static analysis tool, the trouble report data mining and the possible cost and quality benefits of detecting the faults earlier.

### 3.2 Related work

There are several related articles examining the effectiveness of static code analyzers, both as a security tool and as a general fault finder. For most of these articles non-commercial analysis tools are used or no industrial reference data are examined.

In Evans and Larochelle (Evans and Larochelle 2002) a case study with the free tool Splint, the open source software wu-ftp was used, where the discovered faults and false positive numbers were presented. They concluded that a static analysis tool can detect security faults but did not specify the content of the faults. Schuh (Schuh 16) concluded that while today’s code analysis tools are better it is still the expertise of the examiner that is the most important factor. He conducted the study on both C++ and Java code with the aid of three commercial checkers. The study used lab code with known vulnerabilities deliberately inserted, giving some clues about detected vulnerabilities. He also presented a high number of false positive from all of the investigated tools forming one of the most complete studies describing limitations with static code analysis. Carlsson and Baca (Carlsson and Baca 2005) looked at mature telecom graded software, analyzed with the help of open source checkers, ITS4, Flawfinder and RATS. The study showed that a security static code analyzer could find security vulnerabilities but also created a large number of false positives. They did not compare their results with trouble reports or any maintenance cost with data from industry. The study also did a short survey into the return on investments associated with the work of using a static code analysis. Okun et al. (Okun et al. 2007) used the static code analysis tool Coverity Prevent and data from open source projects to determine if the projects using SAT have had any improvement in the number of vulnerabilities. Unfortunately, they did not produce any strong conclusions because there were too many interfering factors. This paper will try to answer the same question by investigating if already reported vulnerabilities could have been detected if SAT was used during development.
3.3 Research methodology

This section first explains how a fault was classified as a vulnerability. This makes it easy to determine what the most common effect of reported vulnerabilities is. A taxonomy to determine the cause of the vulnerabilities is then presented, followed by a process and software description and an in-detail explanation of the case study.

3.3.1 Taxonomy

We used the “Seven Pernicious Kingdoms” taxonomy from Katrina Tsipenyuk to group the vulnerabilities. This taxonomy focuses on implementation faults and is especially useful for static code analysis tools (Tsipenyuk et al. 2005). The taxonomy explains the cause of a vulnerability, but not necessarily the effect of it. It is divided into the following eight groups:

- **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 receiver. 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. incorrect encryption setup.

- **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 and therefore opens the door for race conditions.

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

- **Code quality** - Poor code quality leads to unpredictable behavior that 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 things and setting up barriers between them.

- **Environment** - Environment includes everything outside the code that is still critical to the security of the software.
3.3.2 Vulnerabilities

A failure is labeled as security vulnerable (Landwehr 1981) if, under any conditions or circumstances, it results in denial of service, unauthorized disclosure, unauthorized destruction of data, or unauthorized modification of data. These represent the propagation or effect of an exploit.

- **Denial of Service** - Preventing the intended usage of the software. The most common Denial of Service attacks today are network based that exhaust system resources. Source code that does not always properly release a system resource can 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.

3.3.3 Development and SAT Process

The SAT Coverity Prevent was used during the case study. Prevent, derived from the Stanford Metal/xgcc research project, uses a combination of inter-procedural data flow analyses and statistical analysis to detect faults (Hallem et al. 2002).

![Workflow Diagram](image)

Figure 3.2: This figure explains the Prevent work flow [13].

After the tool has gathered the necessary data it utilizes checkers to detect faults. Each checker tries to match a specific category of potential faults. Figure 3.2 shows Coverity Prevents workflow were the tool first collects source code data and then relies
on checkers to detect faults. The tool can also incorrectly claim that a fault is present (false positive).

Figure 3.3 shows a simplified iteration of the products development process and the intended use of SAT. Because SATs are used early in development and are partially automated they can provide early fault detection. Two of the requirements for an early fault detection tool are fast execution time and the ability to run on non-executable code. Because of these two requirements, dynamic checkers were excluded from the companies initial investigation where Coverity Prevent was deemed the best tool for the job. After the software has been completed it is released to customers. Any bugs that are found by the users are then reported back as trouble reports or bug reports (TR). A maintenance survey on the same development process suggests a 17 times increase in man hours to correct a TR compared to correct the fault during implementation (Damm et al. 2006). Other studies have suggested up to a hundred fold increase in total cost (Briand et al. 2000).The higher cost supports the use of early fault detection tools to prevent a high number of TR’s.

### 3.3.4 Coverity Prevent checkers

The SAT is continuously improving and developing new checkers. At the time of the investigation these checkers stood out and were especially useful. Other studies have explained the tool and checkers more (Boehm 1981).

- **NULL_RETURNS**: A function that can return NULL must be checked before it is used. An attacker might be able to force the function to return an unexpected NULL and cause a segmentation fault.

- **FORWARD_NULL**: A program will normally crash when a NULL pointer is dereferenced. One situation this can happen is when the pointer has been checked against NULL and is dereferenced later. This check identifies such situation
by checking all possible paths where such NULL dereferences can occur. The checker prevents denial of service attacks.

- **REVERSE_NULL**: Another situation this can happen is when the pointer is dereferenced before it has been checked against NULL. If the dereference is NULL, the check programmer should be warned to place the check against NULL before dereference.

- **REVERSE_NEGATIVE**: Sometimes a negative value is not advisable to use. One way to avoid such use is to check for negative value after a possible dangerous use. Attackers can effect loop iterations and copy operations if they can insert a negative values.

- **SIZECHECK**: Incorrect amount of memory allocation can lead to undetermined behavior and program crashes. By checking for inadequate memory allocations for a specific object it prevents memory out of bound errors.

- **RESOURCE_LEAK**: A memory leak can lead to program crashes. A leak of file descriptors, and socket can cause crashes and also have other harmful effects on the program. Apart from usual memory leak checks, it checks for interesting situations like aliasing as well.

- **USE_AFTER_FREE**: Heap values should not be used after they have been deallocated, as it might lead to non-deterministic results when it is used. This also includes checks for double freeing of a pointer.

- **UNINIT**: The use of un-initialized variables can often result in nondeterministic behavior. Under some situations, it can also cause security vulnerabilities.

- **OVERRUN_STATIC**: This checker identifies invalid accesses to a static array, as it can cause buffer overruns that can ultimately lead to security vulnerabilities and program crashes.

- **OVERRUN_DYNAMIC**: Instead of examining static arrays this checker examines dynamic arrays. But the detected faults are of the same characteristic.

- **NEGATIVE>Returns**: If a value that is returned from a function can be negative and is used inappropiately, it can cause multiple errors such as memory corruption, crashes, infinite loops and so on.
3.4 Case study

The results are based on a case study that was performed on both mature telecom graded software and active open source projects. The software systems (A, B and C) are C++ based and are used as servers. The projects have different development processes. The case study was conducted on older versions of the product that had not used SAT’s before. The source code also contained several already known vulnerabilities that were known through trouble reports. These vulnerabilities have been reported from outside the development team. The static analysis tool Coverity Prevent was chosen by a telecommunication and data communication systems manufacturer after an extensive internal investigation where several commercial and open source tools were compared. As a result of the internal investigation the tool Coverity Prevent is now deployed throughout the research and development units of the company. The newest version of the SAT was not used but instead the stable version from when the products were developed. This was necessary to ensure that our study will be as authentic as possible.

The products below have all passed several revisions and can be considered mature. Because of its size and age several developers have worked with the code in product A and B while in C a more dedicated small group of developers has been involved.

Product A had about 600,000 lines of analyzed code. The code base includes some third party code. The source code also includes a frame work which is included in the case study because TR’s on the frame work are reported to the product. Product A receives, retrieves and stores data. It also handles and processes the user data. This is also the oldest product and has source code that was written several years ago.

Product B had about 300,000 lines of analyzed code and handles large amount of user data without doing any heavy processing. Its primary function is to shuffle data between different endpoints.

Product C had about 50,000 lines of analyzed code that serves and processes data. Compared to product A and B, it can not handle the same amount of possible input combinations. Product C mostly serves data to end users. None of the examined versions of the products have had a formal security review as part of their ongoing development. All the product versions have been released and accumulated TR’s for a minimum of one year.

The static analysis tool (SAT) ran on an older already released version with reported faults. Therefore we had to analyze three variables of data, see Figure 3.4:

\[ a = \text{The correctly reported warnings from SAT not reported by TR’s.} \]

\[ b = \text{The intersection between SAT and the security warnings within TR, i.e. those} \]
Figure 3.4: A fictive data collection with vulnerabilities from both the static analysis tool (SAT) and trouble reports (TR).

TR warnings found by SAT.

\[ c = \text{The reported security TR not detected by SAT.} \]

The dormant vulnerabilities (a) were either corrected in newer product versions or reported as TR after the case study. Any vulnerability that were dubious or hard to understand were also confirmed by testing them on the running product. These new faults present quality improvements done by the tool.

The intersecting faults (b) are possible cost improvements were the fault can be found earlier in the development process and therefore save time and money. These were found by examining what lines of code a TR had changed and then comparing if the SAT had issued a warning on these lines.

The known vulnerabilities (c) have all been reported, verified and corrected but were not detected by the SAT. If the TR results into changed code its origin can either been from a SAT detectable code fault or a more obscure design flaw.

### 3.4.1 Examining the tools output

The results from the SAT are divided into three groups.

- **False positives** - These are any warnings that are incorrectly reported by the tool.
- **Security faults** - This group represents warnings that are security related. It is also later on split up into two sub groups.
- **Functional faults** - All the remaining correct warnings that did not fit into the either of the above groups were instead put here.

Security faults were split up into two sub groups. Vulnerabilities are warnings that could propagate into at least one of the four vulnerability classifications. The most contributing factor for the code quality group was rules and checks outside the source code that prevented exploitations. The source code was exploitable but the end product was not. These are therefore split into an own group, separate from the vulnerabilities and is not part of SAT (a) group in our model, Figure 4.
Chapter 3. Evaluating the Cost Reduction of Static Code Analysis for Software Security

In the exploit tables in section 5, the total amount can be higher then the total amount of report warnings. This is possible because vulnerabilities sometimes can be exploited in more the one way.

**Early TR detection effectiveness**

The SAT’s effectiveness is depends on the tools capability to detect known vulnerabilities. Its effectiveness is easily calculated with the gathered data. i.e. an increased value indicates increased efficiency of the tool.

\[ E = \frac{a}{b + c} \]  

A \( E \) value of 1 shows that all TR are detected by SAT while 0 means none are found. \( E \) is easiest understood as percentage values, e.g. \( E = 0.2 \) means that the SAT detects 20% of the TR’s.

**Project cost improvement**

To calculate the actual cost improvement we have to compare the cost of buying and using the tool divided by the cost the company paid to solve the TR’s that the SAT also found, \( b \) in Figure 3.4.

The projects new \( TR_{cost} \) with the SAT as an percentage compared to the old cost would then be

\[ TR_{reduction} = \frac{\left( \frac{\text{Warnings}}{\text{Time factor}} \times Hour_{cost} \right) + \left( c \times TR_{cost} \right)}{TR_{cost} (b + c)} \]  

\( TR_{reduction} \) shows the percentage difference between the projects new maintenance cost and old maintenance cost.

\( TR_{cost} \) is the average total cost of a reported TR for that product. This value is the same for all three products and represents how much the companies development department on average has to spend per reported TR.

\( Hour_{cost} \) represents the hourly developer cost and is constant for all three projects.

\( Tool_{cost} \), the SAT has a, per lines of code, license. The products have different amount of code and therefore have different tool costs. But the license cost per line of code is the same.

\( Time_{factor} \) is the number of warnings a developer can, on average, verify per hour. This value was determined from time reports and a follow up study. For all products 34 warnings can be examined per hour.
Warnings are the total of SAT warnings for the product, including false positives. While warnings and \( Time_{factor} \) are published values, the other three have to be kept hidden due to company secrecy.

**SAT Quality improvement**

In this case quality is measured as the ability to prevent future TR’s by detecting faults before testing, e.g. early fault detection. But to calculate quality we have to know the total amount of vulnerabilities.

To know the total amount of vulnerabilities \( V_{tot} \) all unknown vulnerabilities have to be counted also. Counting unknown vulnerabilities is impossible but we will try to estimate \( V_{tot} \) by comparing known vulnerabilities with new vulnerabilities that the SAT discovers. But SAT can only detect implementation vulnerabilities. Therefore all the design flaws have to be removed from the c group before calculating \( V_{tot} \). Because we are using two different methods of detecting vulnerabilities it might be possible to use similar methods as capture, recapture (Briand et al. 2000) to determine what the total amount of vulnerabilities is.

By multiplying \( a + b \), the total number of vulnerabilities found by the tool, with \( b / (b + c) \) we may calculate \( V_{tot} \):

\[
V_{tot} = \frac{(a + b)(b + c)}{b}
\]  

(3.3)

### 3.5 Results

We divided the results between the three products and present them one by one. The first figure shows the SAT's output while the first table presents the cause of the vulnerabilities and compare the SAT results with TR’s. The second table shows the effect of the vulnerabilities where vulnerabilities sometime can be exploited in more than one way. Therefore the second table can have a higher total amount of vulnerabilities than the first table.

#### 3.5.1 Product A

From the total amount of warnings 371 (22.1%) were false positive warnings and did not require any correction in the source code. 85 (5.1%) were classified by the author as security related and 48 (2.9%) were classified as vulnerabilities that could be exploited while the remaining were not exploitable but were bad code quality. The SAT found a
new vulnerability every 12,500 lines of code. The product had 8 known vulnerabilities, 5 of them were implementation based.

![Figure 3.5: The total SAT results on product A, right circle explains security findings in more detail.](image)

The SAT found all reported input validations. But surprisingly did not find all possible implementation vulnerabilities. Two TR’s that were deemed as implementation faults were not detected by the SAT. The Time and state was most likely not detected do to missing checkers. The API abuse was very unclear. The tool had examined the file and the vulnerability was confirmed in a practical experiment. Also faults of a similar nature had been reported by the SAT before. The most likely conclusion is that the SAT misinterpreted the code and labeled the vulnerability as a false positive.

Due to the nature of the Coverity tool and its available checkers, the majority of the vulnerabilities are stack or heap specific, mostly buffer overflows. Only three vulnerabilities could not be used to repeatedly crash or lock up resources but they did instead present the possibility to leak information.

In product A the majority of vulnerabilities were not detected by TR’s but instead newly found by the SAT. These were mostly local exploits and therefore unlikely to be reported as TR’s.

Three reported vulnerabilities were detected by the SAT. Using eq. 3.1 the SAT had an \( E=37.5\% \) TR detection effectiveness. The cost for the three vulnerabilities in the b group could have been reduced to a \( TR_{reduction}=85\% \) of the original total maintenance cost. If the product would have used SAT and assuming the developers had spend the
Table 3.1: Divided into the cause of the vulnerability, the finding of SAT or reports from TR’s. Third column shows how many were detected by both methods.

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>SAT (a+b)</th>
<th>TR (b+c)</th>
<th>SAT∩TR (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input validation and representation</td>
<td>35</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>API abuse</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Security features</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Time and state</td>
<td>12</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Errors</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code quality</td>
<td>37</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Environment</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3.2: The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more then one way.

<table>
<thead>
<tr>
<th>Failure</th>
<th>SAT (a+b)</th>
<th>TR (b+c)</th>
<th>SAT∩TR (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denial of Service</td>
<td>45</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>Unauthorized destruction</td>
<td>21</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Unauthorized modification</td>
<td>23</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Unauthorized disclosure</td>
<td>17</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

time to correct all the warnings, product A would have had 15% less in maintenance cost for security TR’s and 45 unreported vulnerabilities would not have been present in the released product.

### 3.5.2 Product B

In product B only 5.3% of the total warnings were false positives. 12% of the warnings were security related witch is higher then Product A. But the number of vulnerabilities per lines of code is much smaller. The SAT tool found a new vulnerability every 42.850 lines of code. There were 7 known vulnerabilities, of these only 2 were implementation based.

All the vulnerabilities were memory related and involved the writing or reading of Null values. An attacker could exploit these vulner-abilities with special crafted input messages. Some of the vulnerabilities required a special order of requests before the
program entered an unsafe state. The majorities of TR’s was design based and not do to poor implementation. Four design flaws were missing input validation and one flaw in the products API to outside programs.

Table 3.3: Divided into the cause of the vulnerability, the finding of SAT or reports from TR’s. Third column shows how many were detected by both methods.

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>SAT (a+b)</th>
<th>TR (b+c)</th>
<th>SAT∩TR (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input validation and representation</td>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>API abuse</td>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Security features</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Time and state</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Errors</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code quality</td>
<td>9</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Environment</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Product B had only known DoS attacks and the SAT only found DoS vulnerabilities. This is probably because the product mostly does not process the data. There were seven reported TR’s and the SAT found seven vulnerabilities. Two vulnerabilities were
identified by both detection methods.

Table 3.4: The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more than one way.

<table>
<thead>
<tr>
<th>Failure</th>
<th>SAT (a+b)</th>
<th>TR (b+c)</th>
<th>SAT∩TR (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denial of Service</td>
<td>7</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>Unauthorized destruction</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unauthorized modification</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Unauthorized disclosure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Product B did not process the data in any large amount and was therefore not receptive to a large variety of attacks. Only DoS attacks had been reported and all new vulnerabilities were also DoS attacks. But even the dormant unknown vulnerabilities were remotely exploit-able. All implementation faults were found by the SAT but none of the five design vulnerabilities. The SAT had an effectiveness of $E=28.6\%$ to detect vulnerabilities TR’s. With eq. 3.2 the product would have had a new maintenance cost of $TR_{reduction}=83\%$, a 17\% reduction. At the same time five undetected DoS vulnerabilities would also have been avoided.

3.5.3 Product C

Product C was the smallest of the products and had therefore a small development team. 2 (6\%) of the warnings were false positive and 9 (27\%) where security related. But a majority were just code quality issues and not exploitable. The SAT tool found a new vulnerability every 16.700 lines of code. The vulnerability detection rate was more similar to product A than B. The product had 8 known vulnerabilities, 2 of these were implementation based.

The SAT did not detect any other vulnerabilities than DoS attacks in this product. Because the product was relative small the number of detected vulnerabilities were also few. In this product the SAT managed to detect incorrect encapsulation of data. The two worse TR’s were design based with one lacking necessary input validation and the other allowing access by pass do to weak API.

Eight TR’s had been reported and all of them could be used to launch DoS attacks. Two TR’s where especially bad and could be exploit in all four ways. But the SAT did not find any new vulnerability except for one new DoS attack.

The SAT and had an effectiveness of $E=25\%$ in detecting TR’s. The SAT also found one new vulnerability and provided a new $TR_{reduction}=77\%$. Showing this studies best
Chapter 3. Evaluating the Cost Reduction of Static Code Analysis for Software Security

Figure 3.7: The total SAT results on product C, right circle explains security findings in more detail.

Table 3.5: Divided into the cause of the vulnerability, the finding of SAT or reports from TR’s. Third column shows how many were detected by both methods.

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>SAT (a+b)</th>
<th>TR (b+c)</th>
<th>SAT∩TR (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input validation and representation</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>API abuse</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Security features</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Time and state</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Errors</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code quality</td>
<td>6</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Environment</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

cost reduction with 23%.

This product had the least benefit of the SAT in finding new or already known TR’s, but at the same time saved the most money. The SAT also failed in detecting any of the non-DoS vulnerability. But it did detect all known implementation faults and at the same time one previously unknown. This product had the best saving potential because of its smaller size and therefore tool license cost, combined with a relative large amount
Table 3.6: The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more than one way.

<table>
<thead>
<tr>
<th>Failure</th>
<th>SAT (a+b)</th>
<th>TR (b+c)</th>
<th>SAT ∩ TR (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denial of Service</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Unauthorized destruction</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Unauthorized modification</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>Unauthorized disclosure</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

of detected TR’s compared to its size.

3.5.4 All products

Because the three products are different and have different development processes their results are not 100% compatible. But we combine them anyway to better show the similarities and differences between SAT and TR’s.

The SAT managed to find some TR but did not locate all implementation faults even if it technically should be possible. The SAT was most effective at detecting Input validations. These are often memory related vulnerabilities. The SAT also found several new vulnerabilities that TR’s had not. Only Input validations and API abuses were detected by both methods.

Table 3.7: Divided into the cause of the vulnerability, the finding of SAT or reports from TR’s. Third column shows how many were detected by both methods. Summary of all vulnerabilities from the three products.

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>SAT (a+b)</th>
<th>TR (b+c)</th>
<th>SAT ∩ TR (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input validation and represen</td>
<td>39</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>API abuse</td>
<td>6</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>Security features</td>
<td>0</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>Time and state</td>
<td>12</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Errors</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Code quality</td>
<td>52</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Encapsulation</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Environment</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

In this case study the SAT had an average of 30.4% effectiveness in catching secu-
rity TR’s early. For the total project with all three products the SAT could have lowered the maintenance costs with 18% and at the same time detected 59 new vulnerabilities and prevented customers from having to report 7 security TR’s.

Table 3.8: The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more then one way. This table shows the combines results from all products.

<table>
<thead>
<tr>
<th>Failure</th>
<th>SAT (a+b)</th>
<th>TR (b+c)</th>
<th>SAT∩TR (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denial of Service</td>
<td>55</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>Unauthorized destruction</td>
<td>21</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Unauthorized modification</td>
<td>23</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Unauthorized disclosure</td>
<td>17</td>
<td>6</td>
<td>1</td>
</tr>
</tbody>
</table>

3.5.5 Code quality improvement

To calculate the possible quality improvement we have to estimate the total amount of exploitable code faults. We assume that the ratio between dormant vulnerabilities and implementation vulnerabilities that have both been reported and found by the SAT is constant. This assumption is based on the idea that for the SAT tool to detect the remaining implementation vulnerabilities its checkers needs to be improved or new added and would therefore find even more dormant vulnerabilities.

Figure 3.8: Total amount of implementations vulnerabilities in Product A. Using eq. 3.
In this case study there were still some known implementation based TR’s that were not detected. One was relatively easy to understand and it was the lack of a concurrency checker that was responsible, newer version of the SAT has introduced concurrency checkers. But because the newer version was not released during the products development the vulnerability would not have been detected. On the second undetected implementation vulnerability it was more unclear. Similar faults had been reported before but this partial vulnerability was not. The most likely reason is that the tool decided that the fault was not significant enough and therefore did not report it. The SAT tried to minimize the false positive rate. Only Product A had undetected implementation TR’s. If we assume that new or better checkers would find these and more vulnerabilities then according to Eq. 3.3 a new total amount of implementation vulnerabilities would have been 80 in stead of the currently known 50. Product A had therefore only 60% of its possible code quality improvement, e.g. an improved SAT would have found even more vulnerabilities.

3.6 Discussion

When observing the results from the case study it is clear that in the particular case a static code analysis tool can detect faults that propagate into full vulnerabilities. The false positive rate of free/open source tools have been criticized in previous papers (see section 3.2 related works) and often discussed as a cause why developers do not use static code analysis. In this case study the false positive rate varied from 5-22%. Even the highest false positive rate is acceptable compared to previous studies. The tool managed to find several vulnerabilities in mature, well tested and stable products. From these vulnerabilities the majority were stack or heap related. This shows that the tools security detection capabilities mostly rely on software’s handling of memory, e.g. in the form of string based buffer overflows or memory problems due to tainting. 59 of the now known vulnerabilities were new vulnerabilities that the SAT found.

For this study the most important warnings were the intersecting (b) faults. They were both detected by SAT and reported as TR’s. These faults have already cost the project money and time to verify and correct. Table 3.7 shows that 7 TR’s were detected by the SAT (b). The TR’s were either classified as Input validation or as API abuse. Both detection methods had found more categories but only these two had intersecting vulnerabilities. As for most of the vulnerabilities the effect was mostly DoS attacks. But there were two that could be exploited in more sinister ways.

Detecting design based failures is highly improbable, due to the data the tool uses, 16 (91%) of the undetected TR’s were design based. But there were still 2 (9%) that should have been detected (in Table 3.1 there are two implementation vulnerabilities
found by TR that SAT had not found), i.e. unlike design errors there were no technical restriction. Missing or faulty checkers is the most probable cause why the remaining implementation vulnerabilities were not found. The checker developer has to balance his checker in such a way that it does not provide too many false positives, as such some checkers would not be possible or hard to implement. So, the drawback of user friendly SATs is that some checkers do not catch all vulnerabilities of their type, e.g. the open source tool RATS may probably detect more vulnerabilities but with a much higher rate of false positives.

When comparing SAT and TR it is clear that their detection capability varies. SAT focuses mostly on memory handling and therefore detects mostly Input validations. TR’s on the other hand detect vulnerabilities caused by both implementation and design. TR’s from server software seem also to focus on remote exploits. But with a detection rate of about 30%, SAT still presents a partially automated method in detecting real vulnerabilities early. In the examined project with these three products, the maintenance cost for security TR’s could have been lowered by 17% if a SAT would have been used. The SAT effectiveness and cost reduction are not in this study related to each other. Product A that had the best effectiveness result at the same time had the worse cost saving. Showing that the products size and complexity, assuming it increase the number of warnings, affect the cost of using the tool more than the added effectiveness saved money.

During this case study we have used TR’s as a measurement for SAT effectiveness. As shown in the results TR’s themselves are not a method that detects all vulnerabilities. But they do show vulnerabilities that have had a measurable cost and are therefore a good reference for cost saving actions.

3.6.1 Conclusion

We have looked at real-life trouble reports from three large software systems, consisting of approximately 1,000,000 lines of C++ code. There is a significant cost associated with handling the security related trouble reports in these systems. Our research study showed that, by using a static analysis tool a 17% cost reduction for reported security bugs would have been possible. One product even showed a 23% cost reduction. The cost reduction includes all costs associated with the SAT. Most of the vulnerabilities found were stack or heap related, i.e. the security detection capabilities mostly rely in the software’s handling of memory. No design based vulnerabilities were detected and more implementation failures should have been possible to detect, i.e. when the static analysis tool is lowering the rate of false positives some vulnerabilities are dismissed. Almost 70% of the TR’s were not found by Coverity, i.e. for these there were no reduction of costs. Product A had the best effectiveness with 37.5% of the TR detected
while Product C had the worse with 25%, but because ??? The SAT also found dormant vulnerabilities that were not reported as TR’s. A total of 2.6 times more vulnerabilities were detected by the SAT compared to the TR’s. This means that static code analysis does not only reduce the cost. There is also a significant quality improvement due to the detection of dormant vulnerabilities.
Chapter 3. Evaluating the Cost Reduction of Static Code Analysis for Software Security
Chapter 4

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

Dejan Baca, Kai Petersen, Bengt Carlsson and Lars Lundberg

4.1 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-
Chapter 4. Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?

bilities. 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. 2004), 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-
nerness 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.
4.2 Background and Related Work

The first static analysis tools (SAT) were simple program checkers (Viega et al. 2000) 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.

4.3 Research Method

4.3.1 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 4.1).

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

- **False positive** An incorrectly reported warning.
Chapter 4. Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?

Figure 4.1: 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 SA T. 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.

### 4.3.2 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 answered a simple yes or no question.

4.3.3 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 1998). 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 positives 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 to ensure 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 1998).
- Null-pointer exceptions which allow to user-induced segmentation faults (Krsul 1998).
• Initialization checkers determine if allocated resources are used before they are properly initialized causing information leaks or segmentation faults (Krsul 1998).

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

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.

4.3.4 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.

4.3.5 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
threat is whether the results are applicable for the industry or not. As this experiment entirely includes industrial developers, the results can be generalized to industry developers in common. Overall, the external validity of the study thus can be considered as high.

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.

4.3.6 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 their 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 4.1: A simple off by one warning.

```
Event overrun-local: Overrun of static array "buff"
of size 32 at position 32 with index variable "32"
30: buff[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 an denial of service attack.

Listing 4.2: 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 4. 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;

4.4 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 4.2 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 4.3 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
exactly in the opposite way. That is, the results are better when they developers are uncertain in their answers. Security vulnerabilities vary within the chance range and no improvement claims can be made with confidence.

Because Figure 4.4 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 noteworthy between the three groups. The security vulnerabilities also stay within the chance range.

Figure 4.4: Classifications grouped by the developers general experience.

Figure 4.5 examines the two most important specific experiences. Here, 15 developers with security experience are compared to 19 without. Furthermore, 14 developers with prior knowable in SAT are compared to the remaining 20 developers with no knowledge in SAT. While the security group is better then the non security group in detecting vulnerabilities the results is only just better then 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 4.6 shows developers with both SAT and security experience compared to the developers with no specific or only one specific experience.
Figure 4.5: Classifications based on two specific experience groups.

Figure 4.6: Correct classifications from developers with a combined specific experience.

4.5 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
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 4.2 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 developer’s 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 tool’s 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 their 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.

4.6 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 then 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 4. Static Code Analysis to Detect Software Security Vulnerabilities – Does Experience Matter?
Chapter 5

Static analysis as a security touch point: – An industry case study

Dejan Baca and Bengt Carlsson

5.1 Introduction

During the development of software products mistakes are inevitably made, either in its planning, design or in the implementation. These mistakes often delay and add extra cost to the products’ development, in some cases the mistakes may propagate into costly failures reported by customers. This has inspired research in early fault detection where the correct techniques are applied on the appropriate phase of development to catch as many of the faults as early as possible. Unfortunately, security vulnerabilities are often not detected during daily usage by the end user. Instead malicious users detect the vulnerabilities during abnormal use and often do not report back. Initially early fault detection research did not focus on vulnerabilities and security was not part of the development process. Instead security problems were addressed in the end of the development by security assurance teams. However, with the increased awareness of security vulnerabilities and a higher cost to correct the vulnerabilities late in the process, companies and researchers started to examines possibilities to introduce
security thinking as early as possible, the reason being that it is a generally accepted fact that faults found later in the development process are much more costly to fix than faults found early (Boehm 2002). From this several security development processes and early detection techniques were created. Among the most prominent improvements are the Microsoft secure development lifecycle (Lipner 2005) and Cigital’s software security touchpoints (McGraw 2006). Both of these try to detect vulnerabilities as early as possible to reduce development cost and lead-time. They also have in common that they recommend code reviews during implementation, more specifically automated reviews that would aid the programmer during development. However, for an automated review to be effective there are some requirements that have to be fulfilled, the output has to be useful, the developers have to understand the output and the review has to be integrated seamlessly so the developers want to use it.

So far, few empirical studies have been conducted which show problems and benefits that can be achieved with automated reviews in an industrial context. This is, however, important to understand the challenges and opportunities associated with the approach in a software security context to further improve it. Therefore, we conducted an industrial case study at Ericsson AB, Sweden. The case study makes the following contributions:

- A deployment strategy for static code analysis tools.
- Provide an understanding of which kind of vulnerabilities static code analysis tools effectively detect. This for example gives an indication of whether the tool should be complemented with other methods.
- An evaluation of the usage of the tool by developers to judge warnings provided by the tool, and the support of the tool when correcting vulnerabilities.

5.2 Background and Related Work

5.2.1 Security touchpoint

The company is using the concept of security touch points to improve the security of the end product. Security touch points are based on good software engineering practices and involve quality assurance of software security throughout the software lifecycle (Mead and McGraw 2005).

A security touch point represents activities throughout the whole development lifecycle which should assure high quality of the product from a software security perspective. Figure X shows a process and security touch points related to different phases in
the process. The goal is to detect security vulnerabilities at the earliest touch point possible. This means understanding how to work with security engineering into requirements, architecture, design, coding, testing, validation, measurement and maintenance.

At the code level, the focus is on implementation bugs, especially those that static analysis tools discovered while scanning source code for common vulnerabilities. Manual code review is a necessary practice, but not sufficient for achieving secure software (Porter et al. 1995). This method has some serious drawback; it takes much time away from development and is very knowledge dependent. With a large code base and several third party libraries, code reviews become even more problematic. An alternative is instead computer analysis of the code using either dynamic analysis, done during runtime, or static analysis that uses the actual source code for its analysis. The two techniques have different merits but dynamic checkers have a poor scalability and are often excluded from projects because they take too long time performing the analysis. Static checkers are fast but with a limited ability to detect errors that are more complicated (Ernst 2004). With touch points the authors suggests that a static analysis tool should be used early in the development to detect vulnerabilities (McGraw 2006). The tool should be useful for both security experts and daily use by the developer writing the code. The tool therefore should be integrated into the development process.
and/or IDE, it should not have a high false positive rate, it should provide useful input to all users and it should be security aware. The tool would then be used by the developers on a daily or instant basis and correct vulnerabilities as soon as they are detected, thus reducing the vulnerability detection cost.

5.2.2 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 a vulnerabilities but not necessary the effect of it. The following eight groups are present in the taxonomy:

- **Input validation and representation** - Metacharacters, 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, state 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 things and setting up barriers between them.

- **Environment** - Environment includes everything outside the code that is still critical to the security of the software.
5.2.3 Capabilities of a Static Analysis Tool

With the term static analysis, we mean an automated process for accessing 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. However, static analysis cannot determine the functional correction of the examined code. A contrast to static analysis is dynamic analysis that does its analysis during runtime on a live system. However, dynamic analysis requires that the code is executed with sufficient test cases to execute all code paths and it slows down the test cases substantially. 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 article, we will focus on static analysis as it provides the better results early in software’s development. While static analysis can detect vulnerabilities early, most tools do not guarantee the absence of runtime errors. Static analysis can aid penetration testing but it cannot replace security specific testing, it should be viewed as a compliment that can detect some of the vulnerabilities early and save time and money for the developers. To detect the vulnerabilities, static analysis tools use predefined rules or checkers that explains 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. How often the false positives reported determine the precision of the analysis. 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 and analysis time has 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 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.

A framework for static analysis is sound if all defects checked for 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 actual defects and they still produce false positives. The effectiveness and correctness is also
Chapter 5. Static analysis as a security touch point: – An industry case study

depended on the quality of the checker. The most important groups of security aware checkers are:

- Memory corruption is the largest group and also the most common C++ coding fault. Many security vulnerabilities also arise from memory faults. Each checker specializes in one kind of a fault or common criteria that can identify a group of faults. This group verifies that no invalid access are done to static arrays (OVERRUN_STATIC), all buffer manipulation operations are also verified that they have a correct size allocation (BUFFER_SIZE_WARNINGS) and source buffers are examined so they are not larger then the intended target buffer (STRING_OVERFLOW).

- Reference checkers verify that references and pointer are used correctly. They identify when it is possible that a NULL is used or forwarded instead of a reference (FORWARD_NULL), besides forwarding they also examine that return values really are there (MISSING_RETURN, CHECKED_RETURN). Checks that detect usage of freed memory are also done by this group (USE_AFTER_FREE).

- Initialization checkers determine if allocated resources are used before they are properly initialized (UNINIT, UNINITCTOR). In some cases the allocated resource is never used (UNREACHABLE).

- Race conditions and concurrency faults are a minority but still important checkers. They verify that temp files have a secure usage (SECURE_TEMP) and detect local file based race conditions (TOCOU). Deadlocks and possible infinite loops are also detected (INFINITE_LOOP).

All of leading SATs present there results in a similar manner. The tools usually display their warnings in their own GUI together with the source code that was analyzed. Because the tools use the same presenting technique and style, the results from the study should be generalizable for all leading SATs and not only Coverity Prevent. Below is an example of how these SATs present there warnings, this example is an complex denial of service attack were three conditions first have to be meet before that attack is made possible.

Listing 5.1: Complex denial of service attack that has three conditions.

```c
Event alloc_fn: Called allocation function
"operator new (unsigned int)"
7859: daData = new DdrSet(daTagNo);
At conditional (1): "StringTokenizer::hasMoreTokens()"
```
5.3 Research method

To fulfill the goals of this study we first had to investigate how the tool was used since deployment, this was done by interviewing project leaders and observing how running projects were using the tool. To determine the tools effectiveness we examined the tools historical data to determine what types of faults had been detected and corrected. Finally, to determine how developers use the tool we conducted a study were developers first had to classify and then correct a list of warnings.

5.3.1 Case Study Context

Ericsson AB is a leading and global company offering solutions in the area of telecommunication and multimedia. This study was performed at one of the company’s sites that collaborates heavily with other offshore sites, as such the investigated products and developers are located all over the world.

5.3.2 Research Questions and Propositions

The goal of the study is to provide a comprehensive evaluation of static code analysis for improving the quality of software products from a security perspective. This goal
leads to the following main research question:

- **How effective and efficient is static code analysis as an early vulnerability detector?**

  Effective means that we find the right faults that make the product exploitable. Efficient means that one finds the fault where they are easiest to find and correct.

**Proposition:** Static code analysis is effective and efficient as it has been when looking at faults from a general perspective without security focus. Though, this is worth investigating as it is much harder to understand security vulnerabilities compared to normal faults. Vulnerabilities often require that the software enters a certain state or that the data flow follows a specific path, these requirements are 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”.

1. **What are successful employment strategies for an organization wanting to use static code analysis to detect vulnerabilities?**

   When introducing a new method or process there is always a high risk that this introduction might fail, e.g. due to resistance of change. In order to facilitate the successful introduction we looked at different strategies for introducing static code analysis and evaluate them.

2. **What types of vulnerabilities can be detected with static code analysis?**

   Previously we presented a taxonomy of vulnerabilities. In order to have a secure product it is important to have a good coverage of those vulnerabilities in fault detection.

3. **How effective are developers in using the tool’s output for the purpose of:**

   (a) identifying the vulnerabilities
   (b) correcting the vulnerabilities

   To address the warnings reported by the tool the developers have to judge whether it is a real fault or false positive. If it is a real fault the developer then has to judge if the warning is a security vulnerability. Thereafter, the developers (based on their judgment) decide if and how to correct the vulnerability.

**5.3.3 Case Selection and Units of Analysis**

Ericsson AB is a leading and 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 results presented here are gathered from real industry projects at the company that have introduced and used the tool as part of their development. The company has used the principals of security touch points to improve its software’s security. The tool was introduced into the development process two years prior to the 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.

- Product A had about 600,000 lines of analyzed code. The code base includes some third party code. It has passed several revisions and can be considered mature. Because of its size and age several developers have worked with the code. The source code also includes a framework. The framework is also included in the case study because bugs on the framework are reported to the product. Product A receives retrieves and stores data. It also handles and processes the user data.

- Product B had about 300,000 lines of analyzed code. It has passed several revisions and can be considered mature. Because of its size and age several developers have worked with the code. Product B handles large amount of user data but does not do any heavy processing. Its primary function is to shuffle data between different endpoints.

- Product C had about 50,000 lines of analyzed code. It has passed several revisions and can be considered mature. Because of its size it has a more dedicated small group of developers. Product C serves and processes data but compared to product A and B, it can not handle the same amount of possible input combinations. Product C mostly serves data to end users.

- Product D had about 800,000 lines of analyzed code. The product is very mature and has been developed for many years and in several different countries. The product's main role is as a statistics data gatherer. Even though it does handle large quantities of data it does minor processing and focuses more on throughput and storage.

The products share the characteristic that they have been developed using C++ and follow a server-client architecture.

In this study, we have chosen the SAT, Coverity Prevent. Coverity Prevent is one of the state of the art SATs and has been used in other studies to detect customer reported
bugs and vulnerabilities [13]. This tool was chosen after an internal study showing that a low number of false positive was combined with a high number of reported faults. Because the tool is automated and has a low percentage of false positive (10-20%), industry often uses the SAT as a drop in tool without any training. A more security focused SAT, Fortify, was excluded due to its higher false positive rate and other "free-ware" tools were too simple in an industrial setting.

5.3.4 Data Collection Methods

In this study three data collection methods have been used, namely post-mortem-analysis, industrial experiment, and open-ended interviews. The post-mortem-analysis was used to answer research question 2 by studying historical analysis data and the vulnerabilities that had been detected. The industrial experiment examined how developers classified and corrected vulnerabilities and contributes to research question 3. To answer research question 1 and to verify our post-mortem-analysis we used open-ended interviews. Furthermore, the validity threats to the study are discussed.

Post-Mortem-Analysis

Post-mortem analysis is a powerful tool to analyze findings of completed projects (Myllyaho et al. 2004). The paper suggests four different steps to conduct post-mortem analysis for research projects:

- *Plan the post-mortem analysis (methods and tools):* In this step we identified the products that have been using static code analysis and the repositories showing the fault statistics and reports from Coverity. Ericsson has internal systems keeping track of the required data for this study.

- *Conduct data collection:* We looked at the warnings from the tool and applied the taxonomy. In particular, to classify the warnings we used a taxonomy that is more specific towards implementation faults, because static analysis only detects implementation faults. The outcome is the number of detected faults in each category of vulnerabilities in the classification.

- *Reflection of data with a few key people:* Having collected and classified the data we had informal discussions with developers coding the software that was analyzed. The purpose was to validate our findings in terms of whether the classification is valid on a sample of the classified vulnerabilities. In the first product (A) we did this for all vulnerabilities, for the other products it was approximately...
20%. As for those vulnerabilities the classification was accurate we can assume that the remaining vulnerabilities are also classified correctly.

- **Analysis and Synthesis of findings**: The authors analyzed and reasoned about the findings and people from the company have reviewed the findings as well.

### Industrial Experiment

Experiments are useful whenever one wants to evaluate how well people perform certain tasks in a controlled environment (as is required to answer research question 1.1.3). That means, the researcher is able to manipulate the relevant independent and dependent variables. Furthermore, other variables that influence the outcome, but we are not interested in, can be controlled.

The purpose of an experiment can be described using the template proposed in (Wohlin et al. 2003):

- Analyze static code analysis for the purpose of evaluation,
- With respect to effectiveness in terms of classifying and correcting real software vulnerabilities in software code
- From the point of view of developer
- In the context of industrial developers evaluating warnings of coverity from one real product developed at different sites at Ericsson AB.

**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. In the experiment presented in (Baca et al. 2009) the software developers are grouped based on their experience and the impact of the experience on their performance is investigated. In this paper though we concentrate on presenting the overall effectiveness of all developers. 32 out of the initial 34 developers also answered how they corrected the warnings.

**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
Chapter 5. Static analysis as a security touch point: – An industry case study

security fault). Security faults are further explained in the instructions following the definitions of vulnerabilities (Krsul 1998). 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 (cov ). The initial classification was important to evaluate in this experiment because for most projects there were only a few developers that initially examined the warnings and determined what should be corrected. Therefore vulnerabilities could pass through as false positives if the initial classification was wrong. In a perfect early vulnerability scenario every developer should use the SAT and immediately try to correct the warning, but this was not the case for any of the examined projects.

Operation: The study was conducted in four distinct steps.

1. 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.

2. 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.
3. In the third step the developers corrected the faults that they have classified as true warnings.

4. In the fourth step, we analyzed the impact of the correction in terms of whether the fault was fixed, and if new faults occur. Furthermore, we evaluated whether those new faults are vulnerabilities. Minor or semantic errors were ignored because the developers could not compile their code, instead the code was examined was pseudo code.

Open-Ended Interviews

Besides the quantitative evaluation of the tool we wanted to gather feedback on the perception of the developers regarding the tool. Therefore, we asked them to provide feedback on two aspects:

1. Deployment of the tool: Initially the study had to investigate how the different projects had 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 to fill in a questioner about the tool deployment and usage. Besides answering the questioner, the projects also had to provide 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 how effective the tool actually had been. However, it is also important to know how useful the developers feel the tool had been. By interviewing developers that had used the tool, we gathered their experience and at the same time verified our own results by asking developers if they agreed with our own analysis.

Overall, we talked to more than 15 developers and questioned 11 project managers on their tool usage and deployment. While conducting the interviews we took notes.

5.3.5 Threats to Validity

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
Chapter 5. Static analysis as a security touch point: – An industry case study

results in section 5.4.2 do not represent the full capacity of the tool they to show the result four industry project managed to achieve with the tool.

The studies of this paper are restricted to the usage of Coverity Prevent and Ericsson AB. We believe that Coverity Prevent is equally good as any other state of are static analysis tool. Ericsson is also a global enterprise and the studies where conducted at several different site around the world to try and generalize the results.

5.4 Results

5.4.1 Evaluation of Adoption Strategies

From the initial study three distinguishable approaches for using the tool was identified, also four products with the most historical data were chosen for further investigation. All of the chosen products had C++ code and none of the Java products were examined further. The Java products were not examined further because the chosen SAT, Coverity Prevent version 3.1, did not have any security checkers for Java products that had a possibility of being detected in these types of products.

Industry does not always agree with literature methods and often change or adapt the base idea to suit their need. In this case, the company wanted to introduce the static analysis tool as an early vulnerability detector with as little overhead as possible. Each project could decide by their own how to introduce the tool, because of this there were deployment and usage variation within the company. Three distinctly different approaches to deploy and use the tool have been identified at Ericsson.

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

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

3. A configuration management driven approach. These projects integrated the tool into their bug report system and had one or several developers responsible to classify and report any warnings.

The first method, No overhead, had a large number of failed deployments. Most often the project had paid for the tool, but in the end never actually used it on their source code or in some cases, they had done one test run and never integrated it into
their development process. Developers complained about the lack of time to use or integrate the tool.

In the second method, champion approach, 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 a expired license and was not discovered before this study was done. In these projects, the responsibility to correct warnings is usually on the champion. Most developers assumed the champion will correct all warnings and few developers ever use the tool. These projects often ignored their legacy warnings and only corrected new warnings.

The third method, configuration management integration, had the largest overhead and the tool was often not used as an early fault detector. 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 to correction to the bug handling system. The project either had one champion for the entire project or one developer per component that was responsible to report 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 then 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.

Most of the qualitative data for this paper was acquired by projects using the third method, because they were the only projects that had used the tool in such extent that useful data could be collected.

5.4.2 Vulnerability detection

According to our used taxonomy only vulnerabilities from four groups were detected in our examined products. Each table below explains in more detail how many of the vulnerabilities that were detected and what subgroup the vulnerability belongs to.

<table>
<thead>
<tr>
<th>Table 5.1: Detected vulnerabilities caused by lack of input validation.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input validation and representation</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Buffer Overflow</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Format String</td>
</tr>
<tr>
<td>String Termination Error</td>
</tr>
<tr>
<td>Setting Manipulation</td>
</tr>
</tbody>
</table>
Chapter 5. Static analysis as a security touch point: – An industry case study

Table 5.1 is the largest group and every product had most of its vulnerabilities in this category. Most of the vulnerabilities in this category are related to array memory management and the vulnerabilities can often be exploited severely. Buffer overflows are buffers that can be overwritten by none validated input data. Format String also often result into 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 string that the attacker can insert as non terminated, any read from these string would continue reading until a termination was found.

<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 5.2 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 parties’ libraries affect this category greatly. Missing Check against Null are instances were 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 an possible Null pointer is used or it statistically shows that calls to this API is often verified and there are some instances were this verification is not done. Unchecked Return Value function 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 is ignored exception throws.

<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 5.3 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 base race conditions. File System Access are file base race condition were the file is first verified and then without any locking used and assumed nothing has changed. Missing Lock are possible deadlocks or instances were a variable should have been locked before it is changed.

<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>

Table 5.4 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 telecom providers were uptime is a contract-based requirement. Memory Leak are user induced memory or resource leaks were an user can through legitimate or faked request waist system resource 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 are 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.

In Figure 5.2 the four groups are shown without there phylum. It is easily seen that input validation and representation is the largest group, this group includes buffer overflow and it often create problems for C++ products. Product D also stands out as the product with most variation in its vulnerabilities, it is therefore selected as the product for further investigation. 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 fault, 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 examine code, it might be possible that the tool and the examination might be more costly then to detect the vulnerability later in development.

5.4.3 Vulnerability identification

When classifying the warnings the developers had three choices, either the warning was a false positive, a true positive or a security related warning. The destination between true positive and security warnings might be important to the way they later correct the warning. For an early vulnerability detector to be effective even non-security develop-
ers should be able to detect vulnerabilities with the tool.

In Table 5.5 we see the false positive and how the developers classified the warnings. The majority identified few of the warnings as false positives and it seems that in this case the developers overestimate 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>
<thead>
<tr>
<th>False positives</th>
<th>False positive</th>
<th>True positive</th>
<th>Security threat</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>
</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 security threat. Only on the buffer overflows did 1/3 of the developers correctly classify a warning as a security threat. 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 security threat.

The security classification is not as important as the next subsection were 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.

5.4.4 Vulnerability correction

We first examine how the developers corrected the false positives. These warnings were incorrect and should not be corrected at all. In Table 5.7 the false positives and how many developers corrected them is 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, 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 case of the buffer overflow, seven of the correction 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 been detected during normal testing and it is therefore considered more severe.

Table 5.7: Developers response to false positives reported by the SAT. Most severe at the bottom.

<table>
<thead>
<tr>
<th>False positives</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>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>11</td>
<td>21</td>
<td>0</td>
</tr>
<tr>
<td>Uninitialized Variable</td>
<td>8</td>
<td>24</td>
<td>0</td>
</tr>
<tr>
<td>Exception Handling</td>
<td>6</td>
<td>26</td>
<td>0</td>
</tr>
<tr>
<td>Null Dereference</td>
<td>21</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Erroneous Synchronization</td>
<td>20</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Use After Free</td>
<td>10</td>
<td>0</td>
<td>22</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>7</td>
<td>18</td>
<td>7</td>
</tr>
</tbody>
</table>

Table 5.8 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 through 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 classify the warnings as a security threat. 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.

<table>
<thead>
<tr>
<th>Vulnerabilities</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>
</tr>
<tr>
<td>Unchecked Return Value</td>
<td>6</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>Null Dereference</td>
<td>6</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>Null Dereference</td>
<td>7</td>
<td>0</td>
<td>25</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>1</td>
<td>9</td>
<td>22</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>0</td>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>Insecure Temporary File</td>
<td>2</td>
<td>9</td>
<td>21</td>
</tr>
<tr>
<td>Buffer Overflow</td>
<td>4</td>
<td>13</td>
<td>15</td>
</tr>
<tr>
<td>File System Access</td>
<td>6</td>
<td>17</td>
<td>8</td>
</tr>
</tbody>
</table>

5.4.5 Feedback & Observations

The projects that integrated the tool into the development cycle thought the task was easy, however they also believed the tool lacked in IDE interrogation and would have preferred a solution were the tools database would be part of there existing bug tracker instead of an own GUI. The false positive rate was also considered manageable and according to the developers did not hinder the tools usage. Several developers had

95
expected the tool to find more severe faults than what they perceived the tool had.

In this example the SAT had 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 pseudo code:

Listing 5.2: Required data flow example.

```
(1) Memory us allocated
Com::Message aMessage = new Com::Message();

(2 – n) Takes a false path (several of these existed)
If Input is of type X
Store aMessage as such type

(3) Takes a true path
If Input is of type Y
Does not store aMessage because type Y should not have a message

(4) Takes a false path
Else
No message type found, free memory aMessage

(5) Memory goes out of scope without freeing
Return;
```

In this example, 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 is 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 is 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 were all different types of message, both successful an unsuccessful constructs were executed. The information in the bug report was very vague compared to
the SAT showed 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.

5.5 Discussion

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 as example Java. This is further shown as none of the examined Java products had any reported warnings that we could classify according to our taxonomy. There are numerous studies 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 sporadic, but this is probably because these vulnerabilities are more uncommon and not based on the tools ability to detect them.

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 directly try and correct it. Unfortunately we have observed that the procedure is more often that the warnings is first classified as a true positive or false positive and in some cases prioritize. 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 accurate 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 exists. While the majority of the corrections were harmless by most developers, some warnings created almost only harmful correction. Even through 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 were 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 32 developers did 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 worse 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 it rather obscure warning message.

SAT usefulness was further hindered by its lack of adoption in the company; just stating that SAT should be used was not enough to achieve wide adoption. Only 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 were developers started on their own initiative to examine the tools output. This was caused by the burden of answering bug report from SAT output, as the output was often easier to correct then answering the bug report. But the tools output was also the cause why the tool was not always used. The fault presented by the tool was often seen, by developers, as unimportant and scenarios that actually happened. The warnings were therefore ignored, but at the same time this study observed several cases were low prioritized warnings were indeed exact solutions to obscure bug reports that had not been resolved. Developers without prior SAT experience also ignored more warning then experienced developers.

Because of these shortcomings, some traits are extra important for SAT, if it is intended to be used as an early vulnerability detector. First, the false positive rate should be very low or preferably zero. It is not acceptable that inexperience developers might create new vulnerabilities when they try to correct a false positive. 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. We have also seen a trend that not all developers should correct or classify all types of security warnings, as it may add cost to development it may be more sensible to use a more general SAT as Coverity Prevent during development. After that a security assurance team may verify the results, before or during testing, with a more security oriented SAT that has a higher false positive rate but less chance of false negatives, for example Fortify from Cigital.

5.6 Conclusions

In the four examined products, static code analyses have been 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. Correcting the
reported warnings created some problems and 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 static analysis tool showed in detail how a vulnerability could be triggered. The detail showed to be both a strength and weakness, if the developers understood the warning the correction of the fault was easy and less time consuming than vague bug reports. However, when developers did not understand the warnings the correction created more problems and failed test cases and the warning was often labeled as a false positive.

We believe that for a SAT to be effective as an early vulnerability detector it has to have a low false positive rate. The tool also has to be integrated into the development process in such a way that developers want to use the tool, e.g. seamless IDE integration or by providing direct time saving benefits, some projects achieve this by creating an artificial time penalties through bug reports on SAT warnings.
Chapter 5. Static analysis as a security touch point: – An industry case study
References

Coverity Prevent manual.


REFERENCES


REFERENCES


the year 2000 Network and Distributed system Security symposium, San Diego, CA, USA.


# List of Figures

1.1 Difference between design and actual implementation. .................. 3  
1.2 Security touchpoints development process. ............................ 4  
1.3 Origin and propagation of vulnerabilities. ............................ 5  
1.4 Waterfall Development process at Ericsson AB. ....................... 16  
2.1 Program progress and security analysis ............................... 28  
2.2 Auditing results after manual examination ........................... 32  
2.3 Proportions of different vulnerabilities .............................. 33  
3.1 The relation between source code faults and design flaws resulting in visible failures that might propagate into known or unknown vulnerabilities. ........................................ 40  
3.2 This figure explains the Prevent work flow [13]. .................... 43  
3.3 The intended usage of SAT in an example development cycle. ....... 44  
3.4 A fictive data collection with vulnerabilities from both the static analysis tool (SAT) and trouble reports (TR). ................................. 47  
3.5 The total SAT results on product A, right circle explains security findings in more detail. .................................................. 50  
3.6 The total SAT results on product B, right circle explains security findings in more detail. .................................................. 52  
3.7 The total SAT results on product C, right circle explains security findings in more detail. .................................................. 54  
3.8 Total amount of implementations vulnerabilities in Product A. Using eq. 3. ................................................................. 56  
4.1 Independent and dependent variables. ..................................... 64  
4.2 Correct classifications per warning type. .............................. 69
4.3 Correct answers per confident group. ......................................... 69
4.4 Classifications grouped by the developers general experience. .... 70
4.5 Classifications based on two specific experience groups. .......... 71
4.6 Correct classifications from developers with a combined specific expe-
    rience. ............................................................................ 71

5.1 Security touchpoints development process. ........................... 77
5.2 Detected vulnerability ......................................................... 92
List of Tables

2.1 Number of security warnings .............................................. 30
2.2 Security warnings investigated ........................................... 34

3.1 Divided into the cause of the vulnerability, the finding of SAT or reports from TR’s. Third column shows how many were detected by both methods. .................................................. 51
3.2 The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more than one way. .......................................................... 51
3.3 Divided into the cause of the vulnerability, the finding of SAT or reports from TR’s. Third column shows how many were detected by both methods. .................................................. 52
3.4 The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more than one way. .......................................................... 53
3.5 Divided into the cause of the vulnerability, the finding of SAT or reports from TR’s. Third column shows how many were detected by both methods. .................................................. 54
3.6 The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more than one way. .......................................................... 55
3.7 Divided into the cause of the vulnerability, the finding of SAT or reports from TR’s. Third column shows how many were detected by both methods. Summary of all vulnerabilities from the three products. .......................................................... 55
3.8 The effects or exploits made possible by the vulnerabilities. Vulnerabilities might be exploitable in more than one way. This table shows the combines results from all products. .......................................................... 56

5.1 Detected vulnerabilities caused by lack of input validation. ........... 89
5.2 Detected vulnerabilities caused by API abuse. ................................. 90
5.3 Detected vulnerabilities caused by Time and state. .................. 90
5.4 Detected vulnerabilities caused by Code quality. .................... 91
5.5 Developers identification of false positivies. ....................... 93
5.6 Developers identification of security threats. ....................... 94
5.7 Developers response to false positives reported by the SAT. Most severe at the bottom. .......................... 94
5.8 Developers correction of vulnerabilities reported by the SAT. .... 95
Listings

1.1 Buffer overflow example. ........................ 7
1.2 Format string example. ............................. 8
1.3 Signed/Unsigned integer overflow example. .............. 8
1.4 Integer overflow example. ............................ 9
1.5 Null-pointer dereference example. ........................ 9
1.6 Time of check time of use example. ...................... 9
4.1 A simple off by one warning. .......................... 67
4.2 More complex denial of service attack were three conditions have to be met before the attack is possible. .................. 67
5.1 Complex denial of service attack that has three conditions. 80
5.2 Required data flow example. ........................... 96
Security is always going to be a cat and mouse game because there’ll be people out there that are hunting for the zero day award, you have people that don’t have configuration management, don’t have vulnerability management, don’t have patch management.

–Kevin Mitnick
ABSTRACT

Software vulnerabilities are added into programs during its development. Architectural flaws are introduced during planning and design, while implementation faults are created during coding. Penetration testing is often used to detect these vulnerabilities. This approach is expensive because it is performed late in development and any correction would increase lead-time. An alternative would be to detect and correct vulnerabilities in the phase of development where they are the least expensive to correct and detect. Source code audits have often been suggested and used to detect implementations vulnerabilities. However, manual audits are time consuming and require extended expertise to be efficient. A static code analysis tool could achieve the same results as a manual audit but at fraction of the time.

Through a set of cases studies and experiments at Ericsson AB, this thesis investigates the technical capabilities and limitations of using a static analysis tool as an early vulnerability detector. The investigation is extended to studying the human factor by examining how the developers interact and use the static analysis tool.

The contributions of this thesis include the identification of the tools capabilities so that further security improvements can focus on other types of vulnerabilities. By using static analysis early in development possible cost saving measures are identified. Additionally, the thesis presents the limitations of static code analysis. The most important limitation being the incorrect warnings that are reported by static analysis tools. In addition, a development process overhead was deemed necessary to successfully use static analysis in an industry setting.