CHALMERS

Static Code Analysis For Embedded Systems

Master of Science Thesis in Computer Science and Engineering

MAGNUS ÅGREN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Göteborg, Sweden, August 2009

The Author grants to Chalmers University of Technology and University of Gothenburg the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.
The Author warrants that he/she is the author to the Work, and warrants that the Work does not contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology and University of Gothenburg store the Work electronically and make it accessible on the Internet.

Static Code Analysis For Embedded Systems

MAGNUS ÅGREN

© MAGNUS ÅGREN, August 2009.

Examiner: PATRIK JANSSON

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden August 2009

Abstract

Much software for embedded systems is written in languages such as C. This is known to be error prone, because of manual memory management and similar insecurities. A countermeasure against such problems is static code analysis. This thesis presents an evaluation of techniques for static code analysis, focusing on methods of fault detection. A number of different analysis tools have been tested, at Ascom Wireless Solutions, a developer of embedded system for wireless communication, on production code. The tools were able to detect real faults, but with significant manual interaction required.

Acknowledgements

This thesis presents the project "Static Code Analysis For Embedded Systems", for the degree of master of science in computer science and engineering, at Chalmers University of Technology. The work was carried out at Ascom Wireless Solutions, in Gothenburg, during the spring of 2009.

I want to thank my examiner at Chalmers, Patrik Jansson, for taking on this project and guiding me though it.

I also wish to thank my supervisors at Ascom, Jonas Estberger and Martin Karlsson, for many fruitful discussions and continuous support.

Lastly, I thank the developers in the Amazon project at Ascom for putting up with me running much of the analysis software on their compilation server.

Contents

1 Introduction **1**
- 1.1 Ascom Wireless Solutions 1
- 1.2 Objectives 1
- 1.3 Scope 2

2 Analysis **3**
- 2.1 Software measures 4
- 2.2 Tools 5

3 Pilot study **6**
- 3.1 Tools evaluated 6
- 3.2 Simplified faults 8

4 Results **13**
- 4.1 Lint 13
- 4.2 Cppcheck 13
- 4.3 Saturn 14
- 4.4 Coverity Prevent 15
- 4.5 Klocwork Insight 16

5 Discussion **25**
- 5.1 Conclusions 25

Bibliography **28**

Chapter 1

Introduction

Writing fault-free software is a notoriously difficult task. As software has become ubiquitous, methods for addressing the issue of software faults are of high importance. Static code analysis is the "analysis of source code carried out without execution of that software"[21]. Ideally, because the code does not need to be executable, the analysis can be applied at an early stage of development. How well this works in practise is discussed in chapter 5.

Early feedback allows developers to correct potential faults before code is committed into central repositories. The merit of this approach lies in the difficulty of deriving from a failure the fault that caused it. The process of tracking down the source of a failure is well-known to be costly.

1.1 Ascom Wireless Solutions

Ascom Wireless Solutions is a developer of products and systems for wireless communication. The end users work within industry, medical care, correctional institutions etc. The section Mobile Software Design (MSWD) is responsible for the development of software for wireless telephones. Requirements on the software are high reliability and usability in safety critical systems. Static code analysis is proposed as a means to improve the development work and the feedback to the developers.

1.2 Objectives

The purpose of this work is to evaluate different methods of static code analysis. The objective is threefold:

- The definition of a number of different methods of analysis and discussions of their strengths, limitations and theoretical underpinnings.
- A prototype toolkit for static code analysis, based on these findings. Its implementation shall permit a high degree of automation and the toolkit shall be used on code developed by MSWD.
- A recommendation for how MSWD shall proceed with static code analysis.

To achieve this, four guiding question will be answered:

- What methods of static code analysis are available?
- What are the inherent limitations of static code analysis?
- Which measurements correlate well to software quality?
- What tools are readily available?

1.3 Scope

The code that is to be analysed is written in a combination of C and C++. Tool front-ends must therefore be able to handle both C and C++ constructs. The main intended use case of the toolkit is as an aid during everyday development work. The amount of manual interaction should therefore be minimised. The target of the analysis shall be detection of faults.

There are numerous tools for static code analysis available. To avoid reinventing the wheel, implementation work will focus on combining existing tools, as well as processing their output to ease use.

The subject area of software testing is closely related to static code analysis. Both practises are concerned with defect detection. Testing however, usually requires executable code, and hence becomes applicable later in the development cycle. While testing in general is beyond the scope of this report, there is no clear-cut distinction between test and static analysis.

General quality improvement activities, such as coding guidelines, are beyond the scope of this work.

Organisation of this Thesis Chapter 2 details the notion of software faults and discusses methods for static code analysis. Chapter 3 introduces an evaluation of different tools, and chapter 4 presents the results of this evaluation. Finally, chapter 5 contains conclusions.

Chapter 2

Analysis

Defects in software can result in unintended, erroneous behaviour. More formally, a *defect* or *fault* is defined as:

> A flaw in a component or system that can cause the component or system to fail to perform its required function e.g. an incorrect statement or data definition. A defect, if encountered during execution, may cause a failure of the component or system [21].

Consequently, a *failure* is defined to be:

> Deviation of the component or system from its expected delivery, service or result [21].

Thus, not all faults lead to failures. The terms fault and defect are used interchangeably in this thesis.

To achieve an understanding of software defects, and to create a nomenclature, taxonomies of faults have been created. One taxonomy, that combines results of previous efforts with new contributions, is the Common Weakness Enumeration (CWE) [3], maintained by MITRE, an American, non-profit federal research corporation. The focus of CWE is on security flaws, and thus, many of the faults listed belong to the subset of vulnerabilities: faults that can be exploited through malicious intent. The classical example of a security vulnerability is the buffer overflow: through a carefully crafted string, that is written to memory past the end of an input buffer, an attacker alters the function of a program.

Most relevant to this report are faults that are not vulnerabilities; since the failures most relevant to prevent are those that occur during normal program execution. Given below is a selection of faults, listed in CWE, that during normal program execution, without malicious intent, can lead to failures, and that can be identified through static code analysis.

- Resource exhaustion; an unbounded amount of resources are allocated, but never released.
- Memory leak; allocated memory is not freed before the last reference is removed.
- Double free; one memory address is consecutively freed twice.

- Use after free; memory is referenced after it has been deallocated.
- Freeing unallocated memory, the `free` function is invoked on a pointer not referencing dynamically allocated memory.
- Null pointer dereference.
- Array index out of bounds.
- Data race; data can be altered between two operations, when it is expected to remain constant.
- Dead code: code that, because of its surrounding context, cannot be executed.

2.1 Software measures

One specific method of static code analysis is the collection of different software measures[1]. This effort aims to quantify qualitative aspects of software; such as complexity, testability, reliability, and maintainability. The obtained values have two different uses. Either to function as basis for resource estimation, by serving as an assessment of required effort. E.g., a value of testability may function as basis for determining the number of testers to employ. Alternatively, the measurement results can serve to predict defect density. Although this may be valid for entire software products, note however, that Fenton and Neil [11] did not find correspondence between complexity metrics and the number of faults of a software module. Voas argue that, although complexity itself is not an error, it increases the probability of latent, hidden errors [22]. In conclusion, note also that, in their critique of software defect prediction models [10], Fenton and Neil remark that:

> Despite the many efforts to predict defects, there appears to be little consensus on what the constituent elements of the problem really are.

2.1.1 Common measures

Below, three complexity measures, commonly used for prediction of defect density, are given. The common practise is to assume that the number of defects of a program, is proportional to the value of these measures.

Lines of code (LOC)

Sometimes given in thousands, then written as kLOC, with k for kilo. There is no universally agreed upon definition, but entities typically summed are: all lines of text, all non-blank lines, or all non-comment and non-blank lines.

[1] Zuse [23] argues that the term measure should be used, rather than the term metric; because, informally, a measure denotes a mapping from an object to a value, whereas a metric denotes a criterion to determine the distance between two different objects.

McCabe's cyclomatic complexity

Defined by McCabe in 1976 [17]. It is calculated on the control-flow graph of a program. For most practical purposes it can be computed as the number of decision nodes plus one.

Halstead's complexity measures

Halstead [13] defined a program P as consisting of a number of operators and operands, "and of nothing else". From this he derived the following:

$$\begin{aligned}
\eta_1 &= \text{Unique operator count} \\
\eta_2 &= \text{Unique operand count} \\
\eta &= \eta_1 + \eta_2 \\
N_1 &= \text{Total number of operators} \\
N_2 &= \text{Total number of operands} \\
N &= N_1 + N_2 \\
V &= N \log_2 \eta
\end{aligned}$$

where η is called the vocabulary of P, N is called the length of P, and V the volume of P.

All of the above measures are programming language dependent. They cannot be used to compare programs performing the same function, but written in different languages. For a comprehensive overview with comparison of a vast number of different metrics, see Zuse [23].

2.2 Tools

The first link in a tool-chain for static code analysis is the compiler. The lexical analyser, parser, and type checker of the compiler can typically catch simple mistakes, such as syntax errors, undeclared variables, and assigning floating-point values to integer variables. Although advances in optimisation technology has led to an increased amount of program analysis being performed by the compiler [1], the compiler still, in the words of Johnson: "concentrates on quickly and accurately turning the program text into bits which can be run" [15]. This warrants the need for specialised analysis tools, continuing where the compiler leaves off.

Chapter 3

Pilot study

As previously noted, there are many tools available for static code analysis. Evaluating a large number of tools would be infeasible, and beyond the reach of a single masters thesis project. To get an overview of the field, a pilot study was carried out, testing tools with different theoretical foundations. Five applications were selected: a simpler commercial tool already in use at Ascom, a simpler free tool, a more theoretical, research oriented tool, and two more advanced commercial tools. As raw data, to feed to the tools, simplified versions of three faults found in code developed at Ascom, one entire code base, and one self-contained module, from a project at Ascom, was used. The larger code base, henceforth referred to as "Amazon", constitutes the entire firmware for a DECT[1]-telephone. A specific version from mid-development, with known errors, was selected for the pilot study. The size is approximately 300 kLOC. The module, implementing alarm handling functionality for the phone, has wrapper stubs for the rest of the surrounding system. Its size is approximately 9 kLOC.

3.1 Tools evaluated

3.1.1 Lint

One of the earliest examples of a program written solely for code analysis is Lint, introduced by Stephen C. Johnson in 1978 [15]. It targeted questionable constructs: "bugs and obscurities", for programs written in the C programming language. Examples include: variables and functions declared but never used, unreachable code, and peculiarities, such as the statement of dereferencing a pointer, but ignoring the value.

```
*p;
```

The first version of Lint ran on the Unix system of the time. Lint-like programs for other systems and languages, e.g. JLint and Splint [14, 20], have since been devised . Ascom currently employs the PC-lint program, version 9.00b [18].

[1]DECT: Digital Enhanced Cordless Telecommunications is a standard used for digital cordless phones.

3.1.2 Cppcheck

Cppcheck [5] is a free software tool for static analysis of C and C++ code. According to the tool webpage, bugs checked for include: memory leaks, use of references after deallocation, and out of bounds errors; with the goal of no false positives. Further checks, with known false positives, for bugs such as buffer overruns and array indexing out of bounds, can be enabled. Furthermore, checks for conditions always evaluating to the same value, as well as checks for dead code in the form of unused functions, are optionally available.

Cppcheck was chosen for the pilot study for a rough estimate of what to expect from free, appearingly simpler, tools.

3.1.3 Saturn

Saturn [19] is a research platform from Stanford University [2]. The goal of the project is to develop analysis techniques with high scalability, without sacrificing precision. The platform computes summaries of all the functions used in a program; these summaries are then used in place of the actual code, for all further analysis. Analyses are expressed as systems of constraints, written in Calypso, a special-purpose logic programming language. One such analysis, of null-pointer dereferencing, is included in the Saturn distribution.

The intention behind including Saturn in the pilot study was to gain insight into how much work it would be to adapt a less "mature" tool.

3.1.4 Coverity Prevent

Coverity Prevent [4] is a commercial tool, spun of from Dawson Englers work on metacompilation [9, 12].

Prevent is structured as a suite of command line tools, and a database with a GUI, running as a local http-server. The separate applications perform different individual tasks, such as, configuration, build, analysis, and commit to the database. For a given project, the typical workflow is as follows: First, a configuration is created for each compiler used to build the project. Second, a regular project build is executed, albeit inspected by Prevent, tracking each compiler invocation. Thus, all source files of interest are processed for analysis, and their respective dependencies are resolved. After this, the actual analysis is performed, and the results are committed to the database. Finally, the results can be inspected, either in a web browser, or in the Eclipse IDE [7], via a plugin.

A recent study by Emanuelsson and Nilsson [8] presents Coverity as among market-leaders, and Prevent as current state-of-the-art, within the field of static analysis tools. An evaluation of Coverity Prevent can be seen as giving one kind of upper bound of the current capabilities of static code analysis.

3.1.5 Klocwork Insight

Klocwork Insight [16] is a commercial tool, comparable to Prevent [8]. Being each others foremost competitor, Insight and Prevent show much similarity. Insight is also structured as a suite of applications, mostly command line tools, and a database with a GUI, running as a local http-server. Tasks separated into different applications are configuration, build, analysis, etc. The workflow

is also largely similar, and the results can be inspected either in a web browser, or in Eclipse, via a plugin.

3.2 Simplified faults

Given as source code below, are the three simplified versions of the faults mentioned above. The intention when simplifying has been to retain, as far as possible, the original form of the fault, while cropping away all superfluous information. This treatment can make the examples appear somewhat contrived.

3.2.1 Data race

This is an example of a potential race condition. The function `hazard` uses the variable `accum`, declared as static on line 14, for internally performed work. The function `f` in turn, uses `hazard`, to compute the sum of a range of integers. Neither usage is protected by any locks. When `f` is forked of as two threads on lines 45 and 46, the loop on lines 31 to 34 will not correctly sum the integers (1 to 10000 and 2000 to 3000 respectively), if the threads are interleaved.

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <pthread.h>
 4
 5 typedef struct {
 6    char* name;
 7    int from;
 8    int to;
 9 } args_t;
10
11
12 int hazard(int x)
13 {
14    static unsigned long int accum = 0;
15
16    if (x == 0) {
17       accum = 0;
18    }
19    else {
20       accum += x;
21    }
22
23    return accum;
24 }
25
26 void* f(void* args)
27 {
28    args_t* a = (args_t*) args;
29
30    hazard(0);
```

```
31    for (int i=a->from; i < a->to; i++)
32    {
33          printf("%s: %d\n", a->name, hazard(i));
34    }
35
36    return NULL;
37 }
38
39 int main()
40 {
41    pthread_t foo, bar;
42    args_t foo_args = {"foo", 1, 10000};
43    args_t bar_args = {"bar", 2000, 3000};
44
45    pthread_create(&foo, NULL, f, (void*) &foo_args);
46    pthread_create(&bar, NULL, f, (void*) &bar_args);
47
48    pthread_join(foo, NULL);
49    pthread_join(bar, NULL);
50    return 0;
51 }
```

3.2.2 Reference after removal

This is an example of reference to a node, in a linked list, after its removal. The list nodes are of type `list_t`, defined on lines 10 to 13. All available nodes are allocated statically at line 15. The function `init_list` creates a list from four arbitrary nodes, whereas the function `remove_list_entry` removes one node from the list. The reference to a node in the list, on line 66, immediately after its removal from the list, on line 64, lacks well-defined meaning. Thus, the else-branch on line 69 will always be taken.

It should be noted that `list_t` contains no pointer to any data, but only the pointer structure of the list.

```
 1 #define LENGTH     10
 2 #define FREE_ENTRY 0xFF
 3
 4 // shuffle the list
 5 #define E1 2
 6 #define E2 7
 7 #define E3 6
 8 #define E4 4
 9
10 typedef struct {
11    unsigned char next;
12    unsigned char prev;
13 } list_t;
14
15 list_t list[LENGTH];
16 unsigned char first_entry = FREE_ENTRY;
```

```
17
18 void remove_list_entry(unsigned char i)
19 {
20    if (i == first_entry) { // first in list
21       if (i == list[i].next) { // and last in list
22          first_entry = FREE_ENTRY;
23       }
24       else { // first but not last
25          first_entry = list[i].next;
26          list[list[i].next].prev = list[i].next;
27       }
28    }
29    else if (i == list[i].next) { // last in list
30       list[list[i].prev].next = list[i].prev;
31    }
32    else { // somewhere in the list
33       list[list[i].prev].next = list[i].next;
34       list[list[i].next].prev = list[i].prev;
35    }
36
37    list[i].next = FREE_ENTRY;
38 }
39
40 void init_list()
41 {
42    for (int i=0; i < LENGTH; i++) {
43       list[i].next = FREE_ENTRY;
44       list[i].prev = FREE_ENTRY;
45    }
46
47    first_entry = E1;
48    list[E1].prev = E1;
49    list[E1].next = E2;
50
51    list[E2].prev = E1;
52    list[E2].next = E3;
53
54    list[E3].prev = E2;
55    list[E3].next = E4;
56
57    list[E4].prev = E3;
58    list[E4].next = E4;
59 }
60
61 int main()
62 {
63    init_list();
64    remove_list_entry(E3);
65
66    if (list[E3].next == E3) {
```

```
67        return 0; // unreachable
68    }
69    else {
70        return 1;
71    }
72 }
```

3.2.3 Memory leak

This is an example of a resource leak; there is a path, from the allocation at line 17, through the branch at line 24, via the subsequent call at line 26, finally not taking the branch at line 5, where all the references to the allocated memory is lost, but it is never freed. A higher-level description is that `Get` and `Release` are intended to work pairwise as allocation–deallocation functions. Not all paths are, however, well-behaved in this respect.

As an extra test, for the evaluation of analysis tools, at line 7, deallocation is made dependent on the value at the allocated memory. This behaviour was added to the example after simplification, to see to what extent different tools handle dynamic memory.

```
 1 #define NULL 0
 2
 3 void Release(int* a_pDynamic)
 4 {
 5     if (a_pDynamic != NULL)
 6     {
 7         if (*a_pDynamic != 42)
 8         {
 9             delete a_pDynamic;
10             a_pDynamic = NULL;
11         }
12     }
13 }
14
15 void Get(int a_i)
16 {
17     int* pVal = new int;
18     *pVal = 42;
19
20     if (a_i >= 2)
21     {
22         Release(pVal);
23     }
24     else if (a_i <= 0)
25     {
26         Release(NULL);
27     }
28     else
29     {
30         delete pVal;
```

```
31     }
32 }
33
34 int main(void)
35 {
36     Get(2);
37     Get(0);
38
39     return 0;
40 }
```

Chapter 4

Results

4.1 Lint

The PC-Lint version used at Ascom was run on the three simplified example faults. It could identify two of them:

For the data race example 3.2.1, warnings were raised at lines 45 and 46, stating that the return values from `pthread_create` were ignored. PC-Lint can, however, be given additional information about the semantics of a function, through the means of annotations in the code. When an annotation was provided, declaring the function `f` to be a thread, warnings were raised for the calls to `hazard`, at lines 30 and 33, stating that the use of `accum` was unprotected.

For the reference after removal example 3.2.2, noting was reported.

For the memory leak example 3.2.3, three warnings were raised: one after the `Get` function, at line 32, stating that the pointer `pVal` is not deallocated along all paths, one after the `Release` function, at line 13, stating that the argument pointer `a_pDynamic` is never deallocated, and one in the the `Release` function, at line 10, stating that the last value assigned to `a_pDynamic` is not used.

The Amazon code base contains 1757 suppressed lint warnings. Suppression indicates that the warning has been manually reviewed and classified as a false positive. Data on how many lint warnings that have led to fixes are not available.

The alarm handling module contains 51 suppressed lint warnings.

4.2 Cppcheck

When fed with the simplified example faults, Cppcheck reported nothing, regardless of what checks where enabled. When run on Amazon, with no extra check enabled, two faults were found. Both were calls to the `sprintf` library function — that writes the contents of one character string to another — with the same pointer used for both input and output. According to C99, the behaviour is undefined if the input and output buffers overlap.

```
sprintf(buf, "%s", buf);
```

A summary of the reports from running Cppcheck on Amazon, with all checks enabled, are given in table 4.1.

Fault	Occurrences
Undefined behaviour	2
False memory leaks	56
Condition always evaluating to true	1
Unused functions	1590
Unused struct or union members	37

Table 4.1: Cppcheck run on Amazon, with all checks enabled.

The cases of undefined behaviour are the same as the ones detailed above. The false memory leaks refer to pointer typed class instance variables, in class declarations. This is a known false positive [6].

The reported number of unused functions, 1590, may seem high. Table 4.2 further details their locations in the code. As can be seen, most of the reportedly unused functions are located in third party libraries; the implementation of which are outside of Ascom's control. Although it is normal for library code to have an extensive API, not used in its entirety within a single application, the reports can also contain false positives. False positives could result from Cppcheck failing to handle function pointers, macro expansion, or inheritance correctly. The occurrences reported for code developed by Ascom could be cases of dead code.

Similarly to the unused functions, the cases of unused struct or union members are all found in libraries.

Location	Occurrences
DECT Stack	71
Third party libraries	1388
GUI Implementation	35
Other modules	96

Table 4.2: Location of reported unused functions.

Because the check for unused functions is global, it is not relevant when checking a single module. Cppcheck was, however, run with all other checks enabled, on the alarm handling module. One class without constructor, one case of inheriting from a class with a non-virtual destructor, and four cases of the known false positive memory leak, was found.

4.3 Saturn

Saturn can, unfortunately, only handle C, not C++. When picking tools for the pilot study, this was overlooked. Concerning the simplified fault examples, a null-pointer analysis is of interest with regard to the example memory leak 3.2.3. This is, however, written in C++. Substantial parts of Amazon is, nevertheless, written in C.

Before the C source code can be analysed by Saturn, it needs to be run through the C preprocessor. Among other things, this expands macros and

resolves the inclusion of header-files. The library header files for the operating system used by Amazon contains some extensions to C, to facilitate better compiler optimisation for the target microprocessor. Saturn could not handle these extensions. Dealing with the library dependencies, so that Saturn would be able to analyse the code, may require lots of work. Furthermore, any results might become invalid, if library dependencies are swept under the carpet. This, in combination with the applicability being limited to parts of the source code, led to Saturn being dropped from further evaluation.

4.4 Coverity Prevent

Prevent is a commercial tool, distributed under a licence that requires purchase. Time-limited trial licences are, however, available without cost, and one such was acquired. Coverity's trial process consisted of a number of steps, that followed the tool workflow: First, a configuration was setup for the compiler, and the project to be analysed was built. Second, Ascom obtained a trial licence, and the analysis was carried out. After that, a web conference was held with Coverity, during which the commit of the results, to the database, was remotely administered. The results where then manually inspected, and categorised as either faults, false positives, or constructs working as intended. A typical example of such a construct is an unreachable default case, often containing output of error information, in a switch statement. Because the trial process limited the possibility of committing analysis results to the database, and thus limited the possibility of manually reviewing each reported defect, the large Amazon code base was chosen over the medium sized single module, in hope of detecting more faults; thereby achieving a more thorough evaluation.

The simplified example faults were analysed. For the memory leak 3.2.3, one resource leak was reported. Owing to the constraints mentioned above, this was not investigated further. Analysis of the other two examples yielded no reports of defects.

4.4.1 Amazon

Building the Amazon code base with Prevent was problematic. The documentation provided with Prevent lists the compiler used for Amazon as supported. The standard C and C++ libraries, provided by the compiler vendor, however, contains template implementations that Prevent was unable to parse. The problem affects roughly 150 files, of the circa 1300 that constitute Amazon, but originates in a small number of base classes, using includes from the problematic parts of the system library. These problems have a clear negative effect on the accuracy of the analysis; for example the implementation of the GUI could not be analysed. There were also some more parse problems, pertaining to a custom syntax, supported by the compiler vendor, for binding symbols to memory locations. This is used to create references to memory mapped resources. Around 30 files are affected, and again the source of the incompatibility is located in a common header file.

The default options were used. The different *checkers*, the detection mechanism for each fault type, can be individually tweaked, however. Table 4.3 shows

the initial report. Table 4.4 shows the categorisation of the results reported, after manual inspection.

Fault	**Occurrences**
Singleton pointer used as array	9
Inconsistent check of return value	14
Dead code	9
Dereference after positive check for null	28
Break statement missing in switch statement	1
Statement with no effect	2
Unchecked return value possibly null	2
Heap allocated array indexed out of bounds	1
Stack allocated array indexed out of bounds	43
Dereference before check for null	7
Use of uninitialised variable	23
Value assigned but never used	1

Table 4.3: Initial report from Prevent, for Amazon.

Category	**Number**
Faults	67
False positives	56
Working as intended	17

Table 4.4: Prevent, manual categorisation of the reported defects.

Tables 4.5, 4.6, and 4.7 shows faults, false positives, and working as intended, respectively, by location in the code. The code has been divided into different logical blocks, *components*. Figure 4.1 shows the ratios between component sizes, and faults in each component.

There are known workarounds for certain causes of false positives; the behaviour of system functions for assertions and exceptions can be modelled. The limited possibility of committing analysis results to the database, however, was a hindrance to experimentation with this.

4.5 Klocwork Insight

Like Prevent, Klocwork Insight is distributed under a licence that requires purchase. Time-limited trial licences are, however, available for Insight as well. Klocwork's trial process and licence terms, on the other hand, differs significantly from Coverity's. A technician from a Klocwork reseller visited Ascom, and installed and set up Insight on site. The trial licence permitted two weeks of unrestricted use.

The simplified example faults were analysed. The memory leak 3.2.3 was correctly identified, but the other two examples yielded no reports of defects.

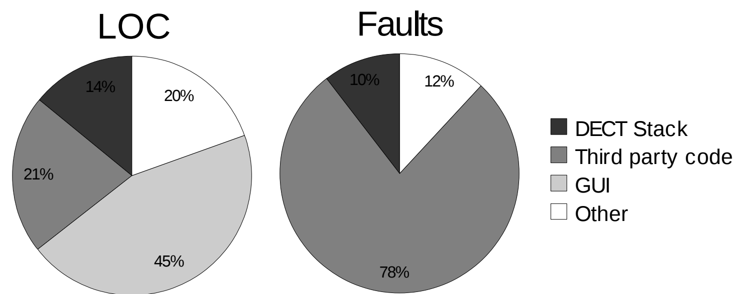

Figure 4.1: Prevent, component size vs. faults.

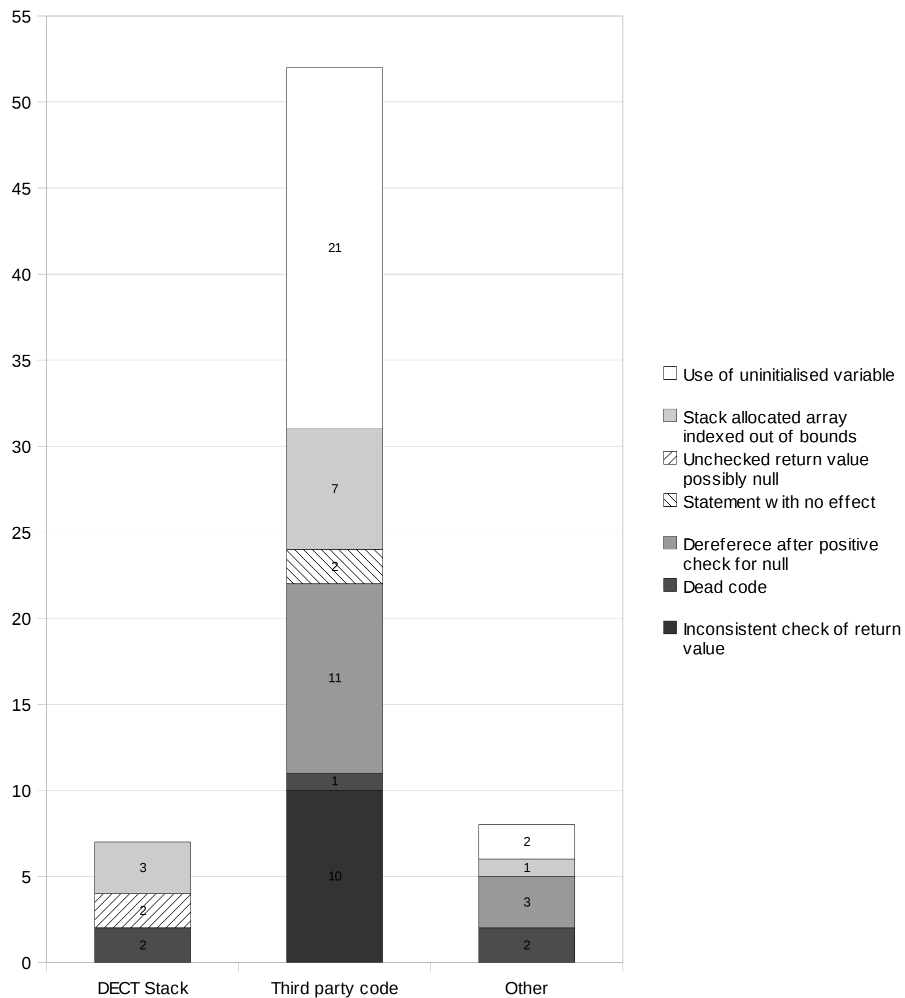

Table 4.5: Prevent, faults by component.

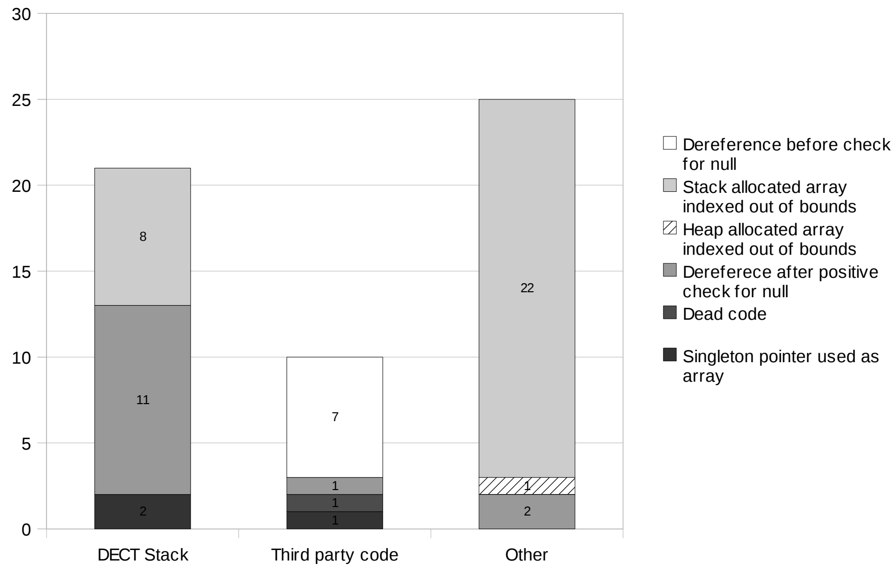

Table 4.6: Prevent, false positives by component.

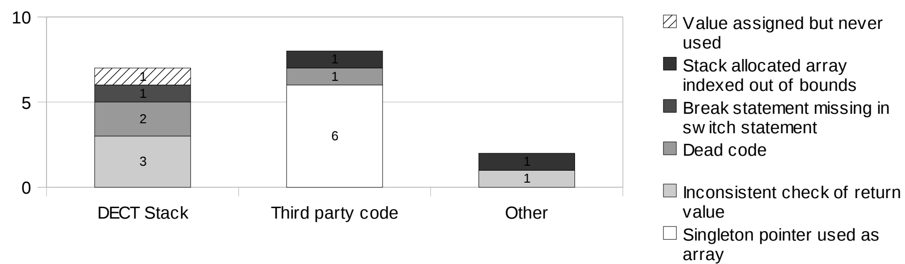

Table 4.7: Prevent, working as intended by component.

4.5.1 Amazon

No real problems where encountered when building Amazon with Insight. Furthermore, as the number of builds and commits to the database was not restricted by the trial licence, experimentation with modelling the system functions for assertions and exceptions was possible. Using such models eliminated numerous false positives. All results presented below are taken from a run with the models in use.

Table 4.8 shows the initial report. Table 4.9 shows the categorisation of the results reported, after manual inspection.

Fault	**Occurrences**
Array index out of bounds	382
Local array index out of bounds	9
Case labels mixing different enum types	14
Memory leak, might	1
Memory leak, must	1
Fault in assignment operator or copy constructor	
Not all data members assigned	8
No assignment operator defined	59
No copy constructor defined	59
Null-pointer dereference	
Positively checked, used in call, might	3
Positively checked, used in call, must	2
Positively checked, might	9
Positively checked, must	1
Unchecked return value, might	4
Unchecked return value, must	19
Pointer passed as argument, might	2
Pointer passed as argument, must	1
Local pointer dereference, might	1
Local pointer dereference, must	1
Check after dereference	3
Use of uninitialised symbol	
Local array, might	1
Local array, must	3
Local variable, might	7
Local variable, must	1

Table 4.8: Initial report from Insight, for Amazon.

Tables 4.10, 4.11, and 4.12 shows faults, false positives, and working as intended, respectively, by component. Figure 4.2 shows the ratios between component sizes, and faults in each component.

Category	**Number**
Faults	205
False positives	288
Working as intended	98

Table 4.9: Insight, manual categorisation of the reported defects.

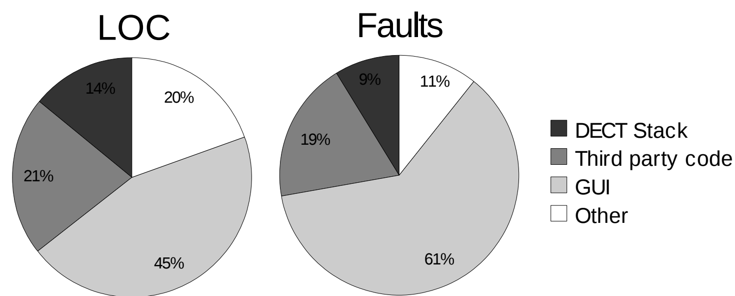

Figure 4.2: Insight, component size vs. faults.

	DECT Stack	Third party code	GUI	Other
Array index out of bounds	13	13	3	4
Local array index out of bounds	-	-	-	3
Case labels mixing different enum types	-	13	-	-
Memory leak, might	-	-	-	1
Memory leak, must	-	-	1	-
Fault in assignment operator or copy constructor				
Not all data members assigned	-	-	1	-
No assignment operator defined	-	-	57	2
No copy constructor defined	-	-	57	2
Null-pointer dereference				
Positively checked, used in call, might	-	-	-	-
Positively checked, used in call, must	-	1	-	1
Positively checked, might	-	2	-	3
Positively checked, must	-	1	-	-
Unchecked return value, might	1	-	-	-
Unchecked return value, must	4	4	6	2
Pointer passed as argument, might	-	-	-	1
Pointer passed as argument, must	-	-	-	1
Local pointer dereference, might	-	-	-	-
Local pointer dereference, must	-	-	-	-
Check after dereference	-	-	-	2
Use of uninitialised symbol				
Local array, might	-	-	-	-
Local array, must	-	-	-	-
Local variable, might	-	5	1	-
Local variable, must	-	-	-	-

Table 4.10: Insight, faults by component.

	DECT Stack	Third party code	GUI	Other
Array index out of bounds	183	8	26	48
Local array index out of bounds	1	-	-	1
Case labels mixing different enum types	-	-	-	-
Memory leak, might	-	-	-	-
Memory leak, must	-	-	-	-
Fault in assignment operator or copy constructor				
Not all data members assigned	-	-	6	-
No assignment operator defined	-	-	-	-
No copy constructor defined	-	-	-	-
Null-pointer dereference				
Positively checked, used in call, might	-	-	-	3
Positively checked, used in call, must	-	-	-	-
Positively checked, might	1	-	-	3
Positively checked, must	-	-	-	-
Unchecked return value, might	-	-	-	2
Unchecked return value, must	-	-	-	-
Pointer passed as argument, might	-	-	-	-
Pointer passed as argument, must	-	-	-	-
Local pointer dereference, might	-	-	-	-
Local pointer dereference, must	-	1	-	-
Check after dereference	-	-	-	-
Use of uninitialised symbol				
Local array, might	-	-	-	1
Local array, must	-	-	2	1
Local variable, might	1	-	-	-
Local variable, must	-	-	-	-

Table 4.11: Insight, false positives by component.

	DECT Stack	Third party code	GUI	Other
Array index out of bounds	20	7	47	10
Local array index out of bounds	4	-	-	-
Case labels mixing different enum types	-	-	1	-
Memory leak, might	-	-	-	-
Memory leak, must	-	-	-	-
Fault in assignment operator or copy constructor				
Not all data members assigned	-	-	1	-
No assignment operator defined	-	-	-	-
No copy constructor defined	-	-	-	-
Null-pointer dereference				
Positively checked, used in call, might	-	-	-	-
Positively checked, used in call, must	-	-	-	-
Positively checked, might	-	-	-	-
Positively checked, must	-	-	-	-
Unchecked return value, might	1	-	-	-
Unchecked return value, must	3	-	-	-
Pointer passed as argument, might	-	-	-	1
Pointer passed as argument, must	-	-	-	-
Local pointer dereference, might	-	-	-	1
Local pointer dereference, must	-	-	-	-
Check after dereference	-	-	-	1
Use of uninitialised symbol				
Local array, might	-	-	-	-
Local array, must	-	-	-	-
Local variable, might	-	-	-	-
Local variable, must	-	-	-	1

Table 4.12: Insight, working as intended by component.

4.5.2 Alarm handling module

As mentioned earlier, the trial licence for Insight permitted analysis of both Amazon and the alarm handling module. Four defects were reported for the alarm handling module. After manual inspection three of these were classified as faults, and one to be working as intended

4.5.3 Comparison with Prevent

Figure 4.3 shows the ratios between different categories, for Prevent and Insight. Although Insight generated more reports than Prevent, 591 versus 170, Prevent showed a higher hit-rate. The overlap between faults found was limited: 15 of the faults occurred in reports from both tool. This confirms the observation by Emanuelsson and Nilsson [8] that: “Even if the tools look for the same categories of defects . . . the defects found in a given category by one tool can be quite different from those found by another tool.”

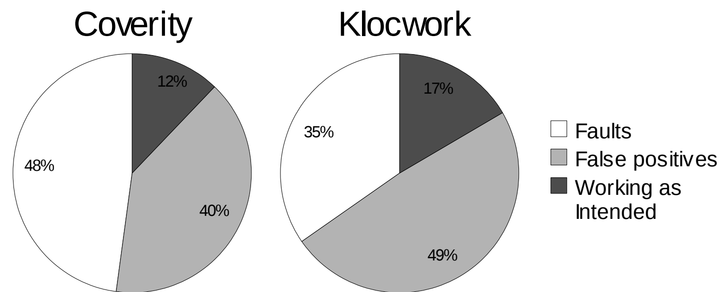

Figure 4.3: Coverity Prevent vs. Klocwork Insight; relative category rates.

Chapter 5

Discussion

In the introduction to this thesis it was claimed that static code analysis can be applied early during development, to non-runnable code. The difficulties in getting Coverity Prevent and Saturn to run on the code refutes this claim, as do the false positives reported e.g. for unmodelled assertions. These experiences, however, together with the analysis results, lead to a number of requirements to place on analysis tools.

- The manual post-processing of fault reports, to determine whether there is a fault or not, should be limited. A tool should not only generate few false positives, it must also be clear from the reports wherein the alleged fault actually lies.
- The tool should integrate well with the build process, and support the compiler and libraries used in the code under analysis. The C preprocessor should also be dealt with correctly; macros must be expanded properly, and conditional compilation must be handled.

5.1 Conclusions

The first objective of this work was the to define and discuss a number of different methods of static code analysis. Apart from the exposition given in previous chapters, the four guiding questions posed are answered below.

What methods of static code analysis are available? Chapter 2 covers different methods of analysis. From a practitioners point of view, static code analysis is mainly available through different tools.

What are the inherent limitations of static code analysis? All analysis must make some simplifying abstraction. Software measures can be seen as the extreme of simplification, reducing functions, modules or entire programs, to a single number. The biggest limitation of software measures, however, is the absence of a measure for predicting fault density.

Among commercial tool vendors, information about underlying abstractions and simplifications made is usually only available, if available at all, to the

customer as marketing material. In this respect, the market for static analysis tools is a sellers market, which is another kind of limitation.

Which measurements correlate well to software quality? To answer this question is to define software quality. One possible definition is that quality lies in few faults reported by some chosen tool. An objection to this definition is that different tools detect different defects, and that it is hard to predict what defects will lead to failures. From the user perspective, software quality is usually closely related to few failures, whereas, from the developer perspective, software quality can be more abstract characteristics of the source code, such as readability or elegance. Certain specific qualities, e.g. testability, relate closely to complexity. Low complexity depends on few paths through the code, which requires fewer test cases for coverage.

Another reflection on complexity is that it could lead to the introduction of faults, simply because developers will have difficulty understanding how the code is supposed to work. In contrast, the opposite could hold; when code is complex it must be thoroughly understood before any change can be made at all. In the face of complexity, developers may take greater care to ensure that changes are indeed correct; whereas straightforward code could lead to sloppiness. The conclusions of Fenton and Neil indicate that neither of the above explanations are generally true, but that each can be valid in the single case.

What tools are readily available? The majority of available tools are commercial and non-free. Free software tools, such as Cppcheck or Splint, are of a relatively simple kind, on par with lint; there is no free software alternative as advanced as Prevent or Insight. Considering the high availability of free software alternatives to non-free software in general, this may seem surprising. It could have to do with the free software philosophy's take on faults; Linus's law, as formulated by Eric Raymond, states that "given enough eyeballs, all bugs are shallow". Thus, it could be that the demand for "bug hunting" software is lower among developers of free software, than among developers of non-free ditto.

5.1.1 Prototype toolkit

A prototype toolkit, the second objective of this work, has not been implemented. An initial hypothesis was that different tools would target different kinds of faults. Through evaluation, a set of tools complementing each other would be identified, and for these a unifying structure could then be implemented, creating a coherent toolkit. Although different tools find different faults, the categories of faults targeted are largely the same. At the same time, the diversity of the categories places each individual tool in the role intended for the toolkit.

Another expectation, also unmet, was that software measures could form part of a toolkit as a guide of analysis effort. Part of the code likely to have many errors would be identified and subjected to more thorough analysis. As previously mentioned, though, the use of software measures as indicators of fault density is, at best, questionable.

5.1.2 Recommendation

The third objective concludes this report; a recommendation for how MSWD shall proceed with static code analysis.

The results show that static code analysis is a feasible way of finding actual faults in production code. In search of good defect detection, one possibility is to evaluate more tools. As the supposed market leading tools, however, target faults of similar kind, I recommend instead that these — Coverity Prevent and Klocwork Insight — be evaluated with regard to aspects complementary to the defect detection; such as cost and return of investment.

Regardless of what tool is used, I recommend actively using its features for modelling certain semantics of the code under analysis: with PC-Lint, use code annotations, with Prevent or Insight, provide models for e.g. assertions and memory allocation functions. Proactively using such features has several benefits; annotations and models can serve as documentation of the intended use of a function, the tools report fewer false positives, but will also detect usage not in line with these given specifications.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett, and Peter Hawkins. An overview of the saturn project. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 43–48, New York, NY, USA, 2007. ACM.

[3] Common weakness enumeration. `http://cwe.mitre.org`. Retreived 2009-05-31.

[4] Coverity Prevent. `http://www.coverity.com/html/coverity-prevent-static-analysis.html`. Retreived 2009-03-14.

[5] Cppcheck - a tool for static C/C++ code analysis. `http://cppcheck.wiki.sourceforge.net/`. Retreived 2009-03-09.

[6] Cppcheck, memory leaks. `http://cppcheck.wiki.sourceforge.net/Memory+Leaks`. Retreived 2009-03-09.

[7] Eclipse IDE. `http://www.eclipse.org/`. Retreived 2009-06-01.

[8] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools (extended version). Technical report, Department of Computer and Information Science, Linköping University, January 2008.

[9] Dawson Engler, Benjamin Chelf, and Andy Chou. Checking system rules using system-specific, programmer-written compiler extensions. pages 1–16, 2000.

[10] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25:675–689, 1999.

[11] Norman E. Fenton and Martin Neil. Software metrics. *Journal of Systems and Software*, 1999.

[12] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *In Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 69–82. ACM Press, 2002.

[13] Maurice H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[14] Jlint 3.0. http://artho.com/jlint/. Retreived 2009-03-15.

[15] Stephen C. Johnson. Lint, a C program checker. In *Comp. Sci. Tech. Rep.* Murray Hill, 1978.

[16] Klocwork Insight. http://www.klocwork.com/products/insight.asp. Retrieved 2009-06-02.

[17] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[18] Pc-lint for C/C++. http://www.gimpel.com/html/pcl.htm. Retreived 2009-03-09.

[19] The Saturn software analysis project. http://saturn.stanford.edu/. Retreived 2009-03-14.

[20] Splint. http://www.splint.org/. Retreived 2009-03-15.

[21] Erik van Veenendaal. *Standard glossary of terms used in Software Testing*. International Software Testing Qualifications Board, version 2.0 edition, December 2007. http://www.istqb.org/downloads/glossary-current.pdf.

[22] Jeffrey M. Voas and Keith W. Miller. Software testability: The new verification. *IEEE Software*, 12:17–28, 1995.

[23] Horst Zuse. *Software Complexity*. Walter de Gruyter, 1991.