Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Elias Penttilä

Improving C++ Software Quality with Static Code Analysis

Master's Thesis
Espoo, May 16, 2014

Supervisor: Professor Lauri Malmi
Advisor: Kaj Björklund, M.Sc. (Tech.)

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF MASTER'S THESIS

Author:	Elias Penttilä		
Title: Improving C++ Software Quality with Static Code Analysis			
Date:	May 16, 2014	**Pages:**	96
Major:	Software Systems	**Code:**	T-106
Supervisor:	Professor Lauri Malmi		
Advisor:	Kaj Björklund, M.Sc. (Tech.)		

Static code analysis is the analysis of program code without executing it. Static analysis tools are therefore a useful part of automated software analysis. Typical uses for these tools are to detect software defects and otherwise suspect code. Several algorithms and formal methods are available specializing in code analysis. Token pattern matching is used by simpler tools, while more in-depth tools prefer formal methods such as abstract interpretation and model checking. The choice of algorithms thus depends on the preferred analysis precision and soundness.

We introduced the practical problems facing static analysis, especially in the context of C++ software. For static analysis to work in a satisfiable way, the tool must understand the semantics of the code being analyzed. Many tools, particularly open-source ones, have deficiencies in their capabilities of code understanding due to being unable to correctly parse complex C++. Furthermore, we examined the difficulty of handling large numbers of warnings issued by these tools in mature software projects. As a summary, we presented a list of five open-source and six commercial static analysis tools that are able to analyze C++ source code.

To find out the viability of integrating static analysis tools in real-world projects, we performed a two-part evaluation. The first part was a measurement of the detection accuracy of four open-source and two commercial tools in 30 synthetic test cases. We discovered that Clang excels in this test, although each tool found different sets of defects, thus reaffirming the idea that multiple tools should be used together. In the second part of the evaluation, we applied these tools on six consecutive point releases of DynaRoad. While none of the tools were able to detect any of the crash defects known in these releases, they proved to be valuable in finding other unknown problems in our code base. Finally, we detailed the integration effort of three static analysis tools into our existing build process.

Keywords:	static code analysis, software quality, evaluation, automation
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN TIIVISTELMÄ

Tekijä:	Elias Penttilä		
Työn nimi: C++-ohjelmien laadun parantaminen staattisella koodianalyysillä			
Päiväys:	16. toukokuuta 2014	**Sivumäärä:**	96
Pääaine:	Ohjelmistojärjestelmät	**Koodi:**	T-106
Valvoja:	Professori Lauri Malmi		
Ohjaaja:	Diplomi-insinööri Kaj Björklund		

Staattisella koodianalyysilla tarkoitetaan ohjelmakoodin analysointia suorittamatta sitä. Tämä tekee siitä hyödyllistä ohjelmien automaattista analyysia varten. Tyypillisiä käyttökohteita ovat ohjelmavirheiden havaitseminen sekä tyylitarkastuksien tekeminen. Analyysityökalujen toteuttamiseen on useita algoritmeja sekä formaaleja menetelmiä. Yksinkertaisemmat työkalut turvautuvat merkeistä koostuvien hahmojen etsimiseen lähdekoodista. Toteutustavan valinta riippuu pitkälti halutusta analyysin tarkkuudesta.

Työssä esiteltiin C++-ohjelmien analyysiin kohdistuvia ongelmia. Staattisen analyysityökalun on toimiakseen ymmärrettävä analysoitava koodi riittävän hyvin, jotta analyysin tuloksista olisi hyötyä. Monella analyysityökalulla on vaikeuksia ymmärtää monimutkaista lähdekoodia, mikä koskee erityisesti avoimen lähdekoodin ohjelmia. Työssä käsiteltiin lisäksi syitä miksi laajojen ohjelmien analysointi on hankalaa suurten varoitusmäärien takia. Lopuksi listattiin viisi avoimen lähdekoodin analysointiohjelmaa sekä kuusi kaupallista ohjelmaa.

Työn tarkoituksena oli selvittää mahdollisuuksia integroida staattisia analyysiohjelmia olemassa oleviin kehitysprosesseihin suorittamalla ohjelmilla kaksiosainen arviointi. Ensimmäinen arviointi koostui 30:stä synteettisestä testistä, joissa mitattiin analyysityökalujen tarkkuutta havaita ennalta määriteltyjä ohjelmavirheitä. Clang-kääntäjä suoriutui parhaiten näistä testeistä. Kaikki analyysityökalut havaitsivat kuitenkin eri virheitä, mikä vahvistaa käsitystä siitä, että mahdollisimman monen työkalun käyttö on suositeltavaa. Toisessa arvioinnissa tutkittiin valituilla analyysityökaluilla kuutta eri DynaRoadin julkaisuversiota. Saaduilla tuloksilla pystyttiin vertailemaan analyysityökalujen pätevyyttä havaita ohjelmasta raportoituja kaatumisvikoja. Analyysityökalut eivät tunnistaneet yhtään tunnettua vikaa, mutta osoittivat hyödyllisyytensä löytämällä muita tuntemattomia vikoja. Työn lopuksi käytiin läpi kolmen analyysityökalun integrointi olemassa oleviin kehitysprosesseihin.

Asiasanat:	staattinen koodianalyysi, ohjelman laatu, arviointi, automaatio
Kieli:	Englanti

Acknowledgments

I would like to thank my advisor, Kaj Björklund, for giving me the original inspiration and motivation for writing this thesis. This thesis would also not be what it is now without the excellent guidance by my supervisor, Professor Lauri Malmi. Many thanks go to my employer, DynaRoad, for having the greatest patience and providing a fantastic environment during this whole process.

Finally, I would like to thank my family for making it possible for me to be where I am now. Without them, I would never have accomplished the things I have and had the capability of writing this thesis.

Helsinki, May 16, 2014

Elias Penttilä

Contents

1 Introduction **8**
1.1 Background 8
1.2 Thesis Motivation 9
1.3 Thesis Goals 10
1.4 Thesis Structure 11

2 Introduction to Static Analysis **12**
2.1 Benefits of Static Analysis 12
2.2 Practical Issues 13
2.3 Uses of Static Analysis 15
2.3.1 Type Checking 15
2.3.2 Style Checking 16
2.3.3 Finding Bugs 18
2.3.4 Improving Security 19
2.3.5 Program Understanding and Visualization 20
2.3.6 Program Verification 22

3 Technical Aspects of Static Analysis **23**
3.1 Transforming Source Code for Analysis 23
3.2 Levels of Analysis 26
3.3 Source Code and Bytecode 28
3.4 Analysis Methods 28
3.4.1 Pattern Matching 29
3.4.2 Data-flow Analysis 29
3.4.3 Abstract Interpretation 31
3.4.4 Symbolic Execution 33
3.4.5 Model Checking 35
3.5 Custom Rules and Checks 36

4 Analyzing Modern C++ Software **40**
4.1 Understanding Code 40
4.1.1 Language Extensions 41
4.1.2 Project Configuration 42
4.2 Scalability and Precision 43
4.3 Managing the Results 43
4.4 Defect Classification 44
4.4.1 Common Weakness Enumeration 45
4.5 Static Analysis Tools 46
4.5.1 Open-Source Tools 46
4.5.1.1 GCC 47
4.5.1.2 Clang 47
4.5.1.3 Cppcheck 49
4.5.1.4 Cpplint 50
4.5.1.5 Flint 51
4.5.2 Commercial Tools 52
4.5.2.1 Coverity SAVE 52
4.5.2.2 Klocwork Insight 53
4.5.2.3 LLBMC 53
4.5.2.4 PC-lint 55
4.5.2.5 PVS-Studio and CppCat 55
4.5.2.6 Visual Studio 2013 56

5 DynaRoad Software **59**
5.1 Features 59
5.2 Technical Overview 61
5.3 Build Process 63
5.4 Existing Analyzers 63
5.5 Potential Improvements 64

6 Static Analysis Tool Evaluation **66**
6.1 Synthetic Tests 67
6.1.1 Results 68
6.2 Differential Analysis 71
6.2.1 Results 72
6.3 Build Process Integration and Reporting 73

7 Conclusion **76**
7.1 Future Work 78

A Defect Descriptions **86**

B DynaRoad Crash Defects **92**
B.1 Version 5.3.0 92
B.2 Version 5.3.1 92
B.3 Version 5.3.2 93
B.4 Version 5.3.3 94
B.5 Version 5.3.4 94
B.6 Version 5.3.5 95

Chapter 1

Introduction

Static analysis is the analysis of program code without its execution. Programs known as static analysis tools perform the analysis on existing source code. These tools provide developers with useful information about the presence of possible defects and vulnerabilities in source code. Fixing these issues typically increases the quality of the software [21, 34, 61].

Developing software with zero bugs is a respectable albeit unrealistic goal. Using static analysis tools in the software engineering process can help reduce the number of software defects. Unfortunately, tools often report a large number of superfluous warnings known as false positives that hinder tool adoption. Applying these tools on existing large code bases can escalate the number of false positives even further. Reducing the effect of these unwanted reports is a large part of getting useful information out of these tools. [5, 6, 37, 61]

As most static analysis tools are able to find different sets of issues, we will evaluate and use as many tools as possible. Using multiple tools results in large reports with many warnings both true and false positives. To circumvent the large number of reports and still get useful information out of these tools, we will integrate each tool in our automated build process. This will enable us to get defect information for each build automatically, and more importantly, only be alerted when new defects are introduced.

1.1 Background

Static analysis tools come in many forms that fulfill different purposes. Some tools are specialized in finding security vulnerabilities while others analyze code for stylistic issues. Many different things can be tracked and analyzed by these tools. In modern compilers, types and sometimes even values are

followed and checked against predetermined constraints and requirements. This allows compilers to perform aggressive optimizations by tracking things such as compile-time constant values and unreachable code [45]. Advanced static analysis tools strive to make even more thorough analyses than compilers. As some forms of analysis can take a considerable amount of time depending on the size of the code base, it is useful to separate it from compilation.

Looking inside these tools, we see that they can have vastly different implementations. Algorithms for statically analyzing code include formal methods, such as abstract interpretation [19] and model checking [28, 58, 68]. Other implementations use more straightforward methods like bug pattern matching [18] and value tracking [30].

Many static analysis tools exist today, but not all are suitable for analyzing large-scale production software, especially software programmed in C++. The C++ programming language presents many difficulties for these analyzers due to being a multi-paradigm general-purpose programming language that maintains backwards compatibility with the low-level C language. Creating a fully standards compliant parser is therefore not an easy task. Libraries providing a functional parser do exist, but their standards compliance is not always sufficient. Fortunately, new frameworks are being built making it easier to implement static analyzers. One recent example is the LLVM[1] compiler framework and its Clang[2] compiler. New analyzers such as Parfait [17] and LLBMC [26, 48, 58] have sprung up that use these existing frameworks and can therefore focus on the analysis instead of code parsing.

DynaRoad[3] is a Windows desktop software for road construction scheduling [39]. The software itself is implemented with the C++ programming language and the user interface uses the Microsoft Foundation Classes[4] (MFC) UI framework. MFC is a vendor-dependent library that requires the use of language extensions developed by Microsoft. The analysis of DynaRoad source code is thus not as straightforward as it would be for purely standards compliant code [9].

1.2 Thesis Motivation

Software is seldom bug-free, which makes it essential to discover ways of finding bugs as soon as possible. Static analyzers provide one of the many components to fulfill this task. Performing static analysis on the source

[1]http://llvm.org/
[2]http://clang.llvm.org/
[3]http://www.dynaroad.com/
[4]http://msdn.microsoft.com/en-us/library/d06h2x6e(v=vs.120).aspx

code of software can help in pointing out possible defects and vulnerabilities. Dynamic analysis, such as unit testing, is also useful as the two forms of analysis complement each other by finding different types of problems at different points in the software engineering process. [16]

In addition to the problems faced when analyzing existing large-scale code bases with non-standard language extensions, the tools are in most parts *unsound* and therefore report a large number of false positives. False positives are warnings and defects that are incorrect due to the analyzer not being fully *context-sensitive* and *path-sensitive*. These imperfections are the result of tools using mere approximations for their program model due to the analysis problem being *undecidable* [15, 43]. No tool is therefore perfect, which is why it is recommended to use as many tools as possible [59, 66].

Excluding style checking, the DynaRoad development process currently lacks any meaningful form of static analysis. Thus, introducing several new tools to this process increases the number of warnings considerably. Systems have to be put in place to manage the analysis reports to prevent overwhelming the developers with low-impact warnings. At Google, a systematic effort to track and review defects was introduced, which provided developers with feedback on new defects found at each point in the development process [5]. Giving developers feedback and making defect reports manageable in the DynaRoad development process is therefore a valuable objective.

1.3 Thesis Goals

The goal of this thesis is to evaluate existing C++ static analysis tools and to research the feasibility of performing analysis on the DynaRoad code base. It is therefore our objective to find as many tools to evaluate as possible that can handle code that has evolved for over 10 years. Open-source and free tools have no immediate cost associated to them and are thus easier to evaluate and integrate. Commercial tools are often very expensive and have time-limited or otherwise restricted evaluations.

Additionally, we want to find out how each tool can be integrated into our existing automatic build process. These tools being a part of that process means that developers receive immediate feedback when new defects are introduced into the code. It is important for developers to react to these defect reports, but forcing the build to fail in case of the presence of any low-impact report is unrealistic. As these tools can report hundreds, if not thousands, of defects for a large software code base, fixing all those issues slows down the adoption of these tools. We will thus evaluate the feasibility of reporting only the differences between defect reports. This makes it easier

for developers to focus on new defects that they are able to fix more easily due to having recent knowledge of the relevant code.

1.4 Thesis Structure

This thesis begins by introducing the concept of static analysis and its capabilities in Chapter 2. The introduction continues by detailing the various practical issues and use cases of static analysis tools.

Chapter 3 follows the introductory chapter with a more detailed explanation of the various technicalities involved in the static analysis process. The similarities with modern compilers is examined in addition to the various levels on which static analyzers can function. A more detailed follow-up on the different levels comes thereafter by introducing the specifics of some analysis algorithms and their capabilities. The extension facilities and support for in-code annotation languages of some tools are also investigated. Finally, the technical hurdles facing these tools when analyzing larger-scale software in production are considered.

The C++ language is briefly introduced in Chapter 4 along with various vendor-specific language extensions. Problems arising when analyzing complex programming languages and some possible solutions are introduced thereafter. The chapter continues by explaining the importance of defect classification and giving an example of an existing classification. Ending with a final listing of various existing C++-specific static analysis tools, the chapter gives a comprehensive overview of the capabilities of each tool.

The thesis continues in Chapter 5 by giving a short summary of the features and technical details of the DynaRoad software. Existing practices in the build process and used static analysis tools are also explained. Furthermore, the potential improvements to the software engineering process are discussed in the final part of the chapter.

In Chapter 6, a handful of tools are evaluated in two different ways. A small set of test cases are devised and the capabilities of each tool is then determined. To evaluate the tools on larger software, they are run on a predetermined set of consecutive release versions of DynaRoad. The differences between these runs are then compared to known bugs and defects, thus giving us a performance metric on the effectiveness of each tool. The results of these evaluations are given in addition to the effort required to integrate these tools into the existing build process.

Finally, Chapter 7 discusses the results obtained in the previous chapters and how these should be interpreted. Future improvements to the process are also investigated and considered.

Chapter 2

Introduction to Static Analysis

The goal of this chapter is to give a broad overview of what static analysis is and how it helps us improve software quality. We will begin by explaining why static analysis is used, and what types of problems it can detect. Furthermore, this chapter will also detail the practical issues facing static analysis and how they can be solved.

Static analysis is the analysis of the software source code, bytecode, or occasionally even binary executables. This is the opposite to unit testing and other dynamic analysis methods, where the analysis focuses on the runtime execution of software. These methods sometimes even complement each other as shown by Aggarwal and Jalote [1], where both forms of analysis are used together to improve the precision of detecting buffer overflow vulnerabilities.

One useful analogy for explaining static analysis is that it can be thought of as source code spell checking [16]. Static analysis tools have existed since the 1970s in the form of "lint" tools. One such notable tool is the lint program by Johnson [38] for checking C source code, where it augments the C compiler by enforcing stricter type rules than the compiler.

Even though static analysis can never completely prove that a program is bug-free, it can still provide us with useful information regarding security, regressions, and performance. It is therefore often used for finding bugs and security vulnerabilities in software. Other practical uses include style checking [6], dead code detection [27], and the enforcement of coding conventions.

2.1 Benefits of Static Analysis

Programmers are not perfect and they will eventually make mistakes. These mistakes are typically the result of small typing errors and the lack of understanding certain programming principles. One way of addressing these

problems is by doing code review, but that typically requires manual review by other programmers. Additionally, a programmer might be biased and consider some parts of the program more important thus mistakenly letting other defective parts trough the review. A static analysis tool treats all parts of the source code equally and will thus always analyze every part of it. Due to this unbiased and deterministic nature of static analysis, it gives programmers an efficient measure against these types of errors. Another typical problem with manual review is the size of the program. Programmers reviewing hundreds of thousands, even millions of lines of code is practically unfeasible, but for a static analysis such code sizes poses no problems. Whole program analysis for any large program is an undeniable benefit, and static analysis tools make this practical. [15]

As static analysis is performed at compilation time, it provides a great opportunity for software engineers to integrate static analysis tools early in the build process. The earlier an issue is detected, the easier and more cost efficient it is to fix. Early detection also means that the programmers receive immediate feedback on the code they are writing. This in turn enables the programmers to iterate on their code faster and possibly notice similar problems in other parts of the code. Thus, programmers making these mistakes learn to handle them in the future. [16]

The use of static analysis tools has been shown to provide cost reductions when maintaining mature software products. On average, maintenance cost reductions of 17% were observed in a benchmark by Baca, Carlsson, and Lundberg [7]. These tools provide an efficient way of improving software quality as detecting even one severe defect is enough to make a tool cost-effective [61].

2.2 Practical Issues

Rice's theorem states that any nontrivial analysis applied to a program can be reduced to the halting problem, thus rendering all static analysis problems undecidable in the worst case. This means that all static analysis tools are imperfect and have to settle on using approximations and making assumptions about the code they are analyzing. Using approximations in static analysis tools does not make them completely useless, though, as we can still gain useful and relevant information from the results they provide. Typically, the issues pertaining to static analysis are more practical in nature. [15, 43]

As the tools themselves work based on approximate representations of the analyzed code, some of the issues facing them are related to this approximate nature of the whole process. The results of an analysis tool can be categorized

into four classes as shown in Table 2.1. A *true positive* is an actual defect correctly detected by the analysis. A *false positive* is a defect found by the analysis that is not an actual defect. A *false negative* is an actual defect not found by the analysis. A *true negative* is correct code found correct by the analysis, i.e., everything not included in the previous categories.

Table 2.1: Result classification.

	Reported	Not reported
Defect	True positive	False negative
Non-defect	False positive	True negative

A static analysis tool that finds all possible issues is said to be *sound*. In practice, soundness also means that the tool reports a high number of false positives. This is no doubt an undesirable trait, which is why most static analysis tools do not strive for soundness, but rather a balance between soundness and usability. [59, 66]

When analyzing source code in specific programming languages, soundness is often further defined respective to something. In the case of C and C++, their support for arbitrary pointer arithmetic makes sound analysis challenging. This problem can be circumvented by making assumptions about the memory model of programs. The SLAM static analysis toolkit by Ball and Rajamani [8] creates sound boolean abstractions of C programs by making simplifying assumptions about their memory model. This model is described as a "logical memory model" where related pointers, such as `*p` and `*(p+1)`, are assumed to point to the same object.

In addition to soundness, the term *complete* is used to define an analyzer that never produces false positives. On the other hand, the analyzer may erroneously leave some actual problems unreported. This means that the analyzer produces false negatives. [59]

The static analysis process is furthermore defined with the help of certain properties. These properties of interest are *flow-sensitivity*, *path-sensitivity*, and *context-sensitivity*. Flow-sensitivity describes the sensitivity of an analysis to remember the order of instructions. In case the order is not remembered, the analysis is said to be flow-insensitive. Path-sensitivity on the other hand concerns itself with the ability of an analysis to remember the path taken through program branches. If the analysis considers all paths possible without avoiding impossible paths, it is said to be path-insensitive. Lastly, the most involved property, context-sensitivity, describes the ability of an analysis to consider its context, e.g., remembering call graphs in interprocedural analysis.

An analysis is context-insensitive if it fails to understand the side-effects of the interprocedural analysis, i.e., the analysis is merely intraprocedural. [59]

Another important facet for a static analyzer is to understand the semantics of the code being analyzed. This means that the static analyzer has to have similar functionality to a compiler. For complex programming languages this can be difficult to implement due to the number of different and slightly incompatible compilers that use their own language dialects.

2.3 Uses of Static Analysis

Static analysis tools come in many different forms. Some tools can automatically validate coding conventions and check for style issues. Other tools specialize in finding serious security problems and other non-stylistic issues in software. Many static analysis tools combine style checking and bug finding thus supporting several forms of analysis. In the following sections we explore the various ways static analysis tools can improve the quality of software.

2.3.1 Type Checking

Most modern compilers of type-safe programming languages perform type checking. Type checking makes it easy to prevent programming errors such as the assignment of incompatible types, as show in Listing 2.1, and the passing of invalid arguments to functions.

Listing 2.1: Example of an assignment with incompatible types.

```
1 float x = 1.0;
2 int* y = x;
```

Some programming languages support implicit conversions between types which may be undesirable. Listing 2.2 shows that in C++ one can convert from a floating point value to an integer implicitly. These types of conversions are called *type coercions*. In this particular case the floating point value is truncated before being converted to an integer.

Listing 2.2: Example of type coercion.

```
1 float x = 1.0;
2 int y = x;
```

Both cases are relevant, but only the first one is an actual type violation. A programmer could consider the second case as a false positive if it were reported as an issue by at static analysis tool, because there is no loss of data. For other values of x data loss could occur, which again would make the example case a true positive.

To improve the type checking done by the compiler, some static analysis tools are able to read annotations written in the source code. Often by adding these annotations, the complexity of the analysis is dramatically decreased, thus making even hard problems feasible [66]. The use of such annotations augment the existing compiler type system by giving programmers the necessary tools to tell the analyzer about specific type and value semantics. Shankar et al. [57] experimented with giving variables a *taintedness* property in order to enforce security-related data-flow policies.

2.3.2 Style Checking

Style checking tools are a simple form of static analysis tools analyzing source code for the purpose of finding coding style violations. Today, modern compilers come equipped with various types of built-in style and coding convention checks. These checks are typically enabled by explicitly enabling them or by setting a suitably high “warning level.”

In Listing 2.3 the whitespace usage is not consistent, which could be detected by a static analysis tool as a whitespace violation. Compilers do not traditionally detect whitespace issues, but implementing such checks in external tools is often trivial using regular expressions.

Listing 2.3: Example of a whitespace style violation.

```
1 if (x== 2)
2     return;
```

Regular expressions are not aware of language semantics and will therefore typically report false positives as seen in Listing 2.4. In this case the style convention dictates that there should only be a single statement per line. The check does not know that the multiple statements are part of a for loop, and would thus be considered a false positive, because for loops are typically written on a single line.

Maintaining non-whitespace related coding conventions can be more challenging. These checks typically require support from the compiler itself due to their semantic nature. One such useful convention is the unused function parameter check demonstrated in Listing 2.5.

Listing 2.4: Example of a whitespace false positive.

```
1 for (int x = 0; x < 10; ++x)
2 {
3     ...
4 }
```

Listing 2.5: Example of an unused parameter.

```
1 int add(int x, int y)
2 {
3     return x + x;
4 }
```

In addition to simpler style and whitespace checks, we can perform larger-scale analysis on source code in order to find *code smells*. Code smells are non-technical issues related to maintainability and program design. Some automatic tools have been developed for automatic inspection of such issues, such as the jCosmo program by Emden and Moonen [22] and the Detex program by Moha et al. [50].

Another non-technical static analysis application is the detection of duplicate code, also known as *clone detection*. Duplicate code is often the result of programmers copy-pasting similar code, which then leads to code defects when the copy-pasted code is not updated along with the original code. One way to eliminate duplicate code is by refactoring the offending code, but this manual process requires finding every instance of the duplicate code in addition to making error-prone modifications. Li et al. [46] implemented a static analysis tool, CP-Miner, for automating the discovery of duplicate code.

Another relatively well-known clone detection tool is PMD, which provides a "copy-paste detector"[1] for the purpose of finding clones. This tool supports both Java and C/C++ source code and provides results as a HTML report. Duplicates are found with the Rabin-Karp string matching algorithm, which makes the task of finding substrings within large code bases feasible.

One recent application of clone detection methods is ClonEvol [32], which uses the source code documentation generator Doxygen[2] together with the similarity analysis tool Simian[3]. Doxygen is used as a simple static

[1]http://pmd.sourceforge.net/pmd-4.2.5/cpd.html
[2]http://www.doxygen.org/
[3]http://www.harukizaemon.com/simian/

analyzer that parses source code and provides semantic meaning to it. This information is used to track the evolution of code in a version control system as illustrated in Figure 2.1. Simian is a general-purpose similarity analyzer that is in this case used to find clones and track their movement and behavior throughout the code base history.

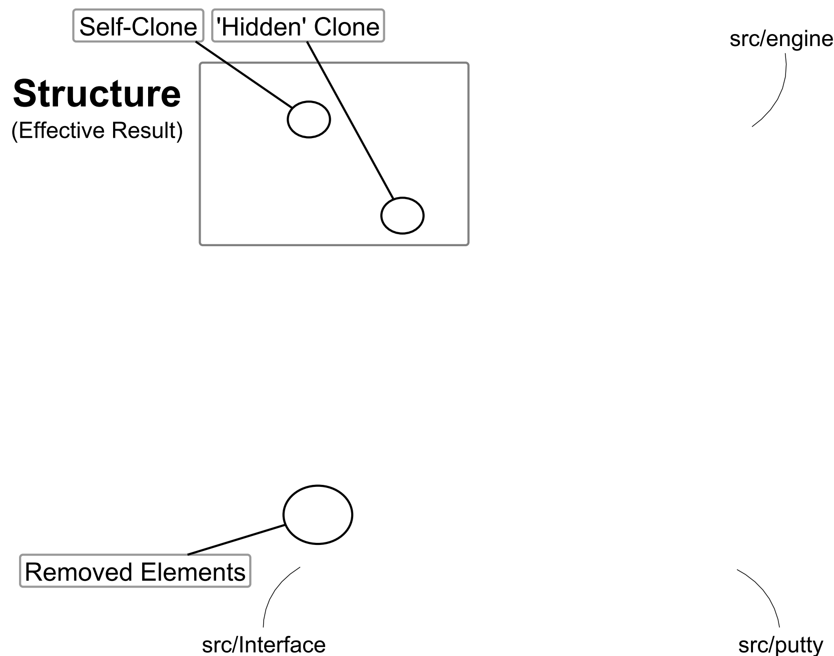

Figure 2.1: ClonEvol output for the open-source FileZilla application. [32]

2.3.3 Finding Bugs

Contrary to style checking, bug finding tools look for patterns which may lead to behavior unintended by the programmer. Sometimes style checks, such as unused variables and parameters, can help in detecting these kinds of issues, but bug checks are often more specific in nature. The distinction between style checking and bug finding tools is that the purpose of the latter is to detect problems in the source code which might lead to runtime issues [34]. Fixing problems reported by style checking tools do not necessarily result in any runtime differences.

A typical problem in C++ programs is the mistake of assigning to a variable instead of comparing to a variable. This manifests itself as a completely valid and syntactically valid program, but is obviously a mistake as seen in Listing 2.6.

Listing 2.6: Example of an unintended assignment.

```
1 if (x = 100)
2     f(x);
```

Unfortunately, sometimes the assignment is actually intended as in Listing 2.7, which makes this problem harder to detect unambiguously. In this case, though, the distinction is that the assignment is a pointer assignment and that the pointer dereference is dependent on the result of the assignment. A simple tool might report both cases as errors, which is obviously wrong for the second case, and thus lead to a false positive.

Listing 2.7: Example of an intended assignment.

```
1 if (int* x = get_pointer())
2     f(*x);
```

FindBugs[4] is a popular and widely researched static analysis tool for Java-based software. FindBugs works by analyzing Java bytecode looking for specific bug patterns, such as infinite recursions and guaranteed null pointer exceptions [18]. The tool has since been further improved by adding in support for more complex data- and control-flow analysis capabilities in order to more reliably detect null pointer dereferences [36]. The null pointer analysis uses an approximation of the static single assignment (SSA) form by trying to detect distinct values among local variables and operands. Forward data-flow analysis is then performed using the approximate SSA to find all null pointer dereferences. Finally, a backward data-flow analysis is done for each possible null pointer dereference to actually verify its existence [35].

2.3.4 Improving Security

Programming errors, such as buffer overflows, are a serious and, unfortunately, a recurring issue in software. Static analysis tools can find these problems in a similar fashion to the previously mentioned bug finding tools. In the case of security issues, though, the number of false negatives should be minimized and scrutinized through manual code review.

Splint[5] is an open-source static analyzer for C programs specializing itself in detecting security vulnerabilities. The analyzer can detect buffer

[4] http://findbugs.sourceforge.net/
[5] http://www.splint.org/

overflows, format string problems, and other programming mistakes often attributed to reduced software security [25].

In C and C++ programs, format string vulnerabilities represent a real threat due to the lack of type checking. User input is sometimes used directly as a format string, deliberately or by mistake, which leads to serious problems. In Listing 2.8, if the string input by the user is `%s`, the `printf` function will proceed to read characters directly off the program stack. Here though, the stack could contain sensitive information which would then be compromised. Attackers can also use format strings to write data with `printf` using the `%n` format specifier, which writes the current count of output characters to a variable. [57]

Listing 2.8: Example of a format string vulnerability.

```
1 printf(buf);
```

When copying strings with unsafe C functions, such as `strcpy` demonstrated in Listing 2.9, a static analysis tool can mistakenly report the copy as a possible buffer overflow. A complete analysis of the possible inputs to `strcpy` is required to understand that no buffer overflow is possible. This case requires the sizes of the source and destination to be statically known at compile-time, but without this knowledge, a naive analysis would give a false positive.

Listing 2.9: Example of a non-overflowing copy.

```
1 const char* x = "Hello";
2 char buf[256];
3 strcpy(buf, x);
```

2.3.5 Program Understanding and Visualization

Source code can also be statically analyzed to gather data- and control-flow information for visualization graphs. These graphs help programmers understand the multiple ways a program can work at runtime. Many integrated development environments (IDEs) also contain features implemented using static analysis techniques. A common feature is source code navigation enabling programmers to find all references to a symbol or to go directly to function definitions in other files.

Static analysis tools will sometimes output reports in graphical formats using HTML or a graphical user interface. This gives the tool a way of visualizing reported issues by giving detailed data- and control-flow information. Clang can output analysis reports in HTML format as shown in Figure 2.2.

```
1  char f(int i)
2  {
3      char buf[10];
4      return buf[i];
           2  ← Access out-of-bound array element (buffer overflow)
5  }
6
7  int main()
8  {
9      f(20);
       1  Calling 'f' →
10 }
```

Figure 2.2: Example of a buffer overflow found by Clang.

Chang, Jo, and Her [13] implemented an exception propagation visualization tool using a set-based control-flow framework. This tool gives users additional information regarding the types of exceptions a specific Java method throws and the paths traveled by the exception through a control-flow graph. With this information users of the tool are able to more specifically decide what exceptions to catch and where. A sample display of the tool can be seen in Figure 2.3.

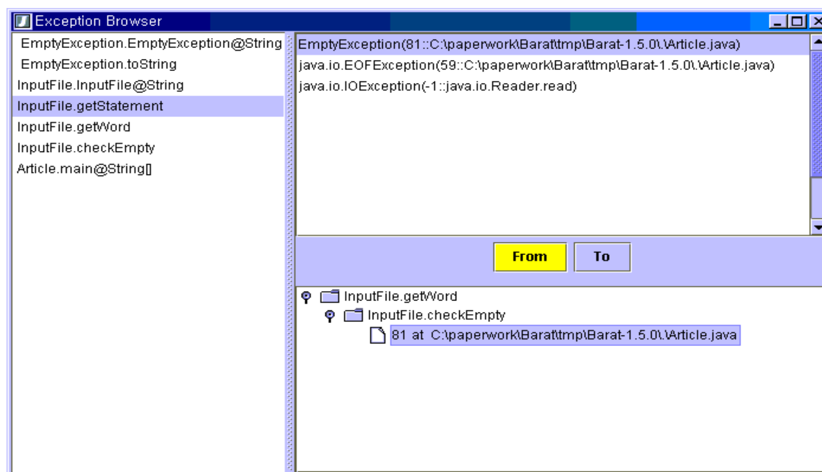

Figure 2.3: Example of an exception propagation path. [13]

2.3.6 Program Verification

Program verification is the verification of a program using a preexisting specification. This type of verification is useful in case of mission-critical programs where no failure is accepted. If the failure of a program has great economical and social downsides, then having a complete specification available can be cost-efficient. Ideally, the program specification would specify the functionality and operation of a program completely, but in practice a thorough specification like this is often infeasible. This type of a complete verification is called *equivalence checking*, but no tools exist for equivalence checking large-scale software due to the problem size.

The problem can be made feasible by limiting the specification to a partial specification. The verification of this partial specification is often called *property checking*. Property checking verifies the properties of a program through *temporal safety properties*. These properties specify the order in which a set of instructions are disallowed to be executed. The typical use-after-free bugs can thus be detected by implementing a property checking rule disallowing the use of pointers after they have been freed.

Program verification and property checking tools are sound with respect to the specification, i.e., they will never report false negatives. Although, such tools will often place restrictions on the types of programs that can be analyzed. Problematic constructs, such as function pointers and arbitrary pointer arithmetic is therefore not allowed in order to maintain the promised soundness. These tools are implemented using model checkers or by logical inference. [16]

Chapter 3

Technical Aspects of Static Analysis

In this chapter we will continue the description of static analysis by examining the more in-depth technical aspects of static analysis tools and methods. To begin, we will explain how similarly static analyzers and compilers function at the beginning of the analysis process when they transform source code into data structures. We will also describe a few of the current state-of-the-art algorithms used by static analyzers. The chapter will continue by showing how custom rules and checks are typically implemented and what they can provide developers in terms of improving analysis precision and efficiency.

3.1 Transforming Source Code for Analysis

Simple static analysis tools work by finding predefined string patterns in source code. More sophisticated tools do not look for patterns in the source code itself, but instead analyze its more complex representations. In order to get to this advanced representation, the code is processed similarly to a compiler, but with the addition of analysis-specific customizations. As seen in Figure 3.1, a static analysis tool builds a model, e.g., a syntax tree and control-flow graph, and then does the actual analysis using its predefined rules.

Many steps in this process are similar to the way compilers work, but it should be noted that modern C++ compilers do not have clearly defined boundaries between lexical analysis, parsing, and semantic analysis. Such an intertwined arrangement is required in order to support the otherwise ambiguous constructs of the C++ language and is thus also a requirement for static analysis tools in order for them to be fully standards compliant.

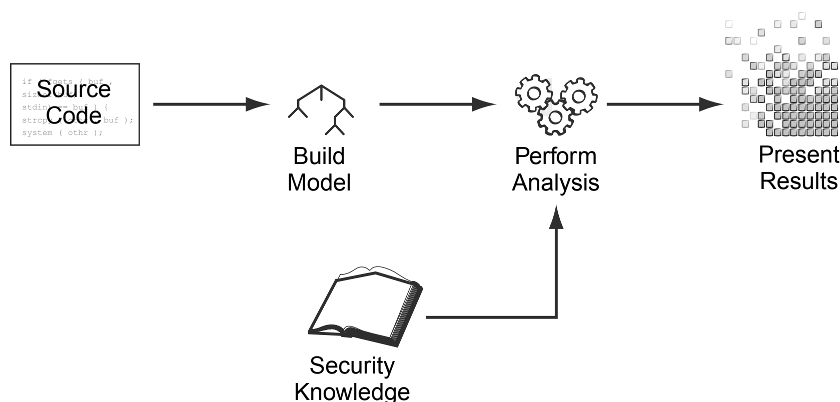

Figure 3.1: High-level representation of the inner workings of a security-focused static analysis tool. [16]

As with compilers, static analyzers begin by performing a lexical analysis on the source code. The purpose of this lexical analysis is to tokenize the input stream of characters acquired from reading the source code. Next, the stream of tokens is parsed by a parser into a parse tree, which in turn is transformed into an *abstract syntax tree* (AST). An example AST created by Clang from Listing 3.1 is depicted in Figure 3.2.

Listing 3.1: Source code used for AST example.

```
1 int main()
2 {
3     int x = 10;
4
5     if (x == 10)
6         return 0;
7
8     return 1;
9 }
```

The AST is useful during analysis due to it being a normalized version of the parse tree without extraneous type constructs whose only purpose is to make the parsing unambiguous. A normalized AST could, for example, be constructed using a smaller version of the language, e.g., by converting all for-loops into while-loops. This in turn is beneficial for analysis as then the tool only is only required to support one type of loop construct thus making checks simpler.

The next step in the transformation process is to build a symbol table

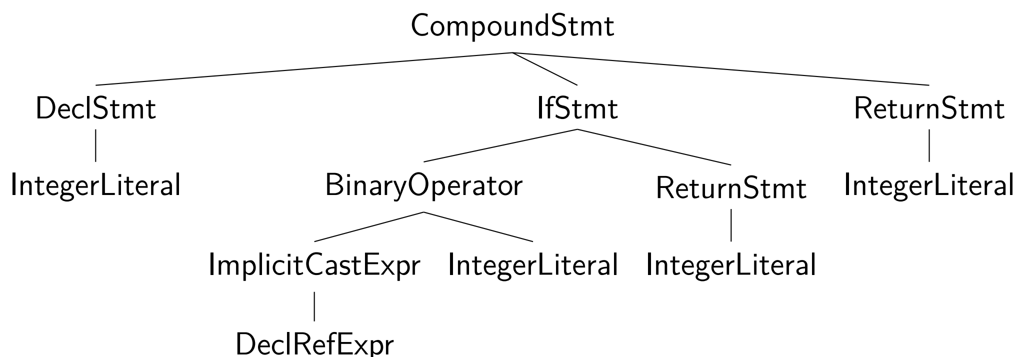

Figure 3.2: Example AST produced by Clang.

with semantic analysis. This is the part where variable names and other identifiers are placed in a symbol table along with their corresponding types. With this information, a compiler can check identifier types and warn about their improper use. Similarly, for static analysis, this information is of great use when searching for bug patterns. With perfect knowledge of all identifier types, patterns can be constructed to match only specific types, such as pointers or integers.

Now, with the help of the AST and the symbol table, the static analysis tool can proceed to the next step of analyzing the control- and data-flows of the program. These two analysis methods are essential in finding the possible paths taken by a program and if those paths cause potential problems. To give an example, let us consider the function `f` in Listing 3.2 which takes a single integer as an argument and uses this integer as an array index on a 10 element array. If this function is then called from another function with the argument 20, then only by analyzing the data-flow can it be irrefutably shown that the argument causes an array index out of bound error. The output of Clang demonstrating this is shown in Listing 3.3. In this case the analysis is an interprocedural analysis, meaning that the whole program is analyzed while considering the interactions between all functions. The other form of analysis, intraprocedural analysis, concerns itself only with the analysis of paths within a single function.

To make data-flow analysis easier, a specific transformation called *static single assignment* (SSA) is used. The transformation interprets every modification made to a variable as a new variable with a new name, which in turn makes analyzing data-flow easier due to unambiguity. In literature these new variables are often named using subscripts. Thus, a simple addition of variables shown in Listing 3.4 can be represented in SSA form as shown in Listing 3.5.

Listing 3.2: Example of a buffer overflow found by interprocedural analysis.

```
char f(int i)
{
    char buf[10];
    return buf[i];
}

int main()
{
    f(20);
}
```

Listing 3.3: Output from CLANG detecting a buffer overflow with the help of data-flow analysis.

```
$ clang -cc1 -fsyntax-only -analyze -analyzer-checker=alpha.security buffer.cpp
buffer.cpp:4:12: warning: Access out-of-bound array element (buffer overflow)
    return buf[i];
           ^~~~~~
1 warning generated.
```

In case the program branches and each branch assigns a variable different values, the now differently named variables will finally merge after the branches join. This merging provides a way for SSA to work in unambiguously even for branching paths. What essentially happens, is that a new variable is created using a ϕ function to that receives the value of any of the branched values. The use of this function is demonstrated in Listing 3.6.

3.2 Levels of Analysis

Static analysis algorithms can be grouped into roughly three different categories: *lexical*, *syntactic*, and *semantic*. Some examples of the various types

Listing 3.4: Variable addition.

```
x = 1;
y = 2;
x = x + y;
```

Listing 3.5: Variable addition in SSA form.

```
1 x1 = 1;
2 y1 = 2;
3 x2 = x1 + y1;
```

Listing 3.6: ϕ function joining two branches.

```
1 if (y > 0)
2     x1 = 1;
3 else
4     x2 = 2;
5 y1 = φ(x1, x2);
```

of defects belonging to these categories are shown in Table 3.1.

Lexical tools include simple pattern matching analyzers, such as GREP. Pattern matching tools directly analyze the character stream for any unwanted patterns. More flexibility is achieved by using matching constructs such as regular expressions.

Syntactic tools, such as CPPCHECK and FLINT, work with the token stream. These tools can use the token stream directly or build *abstract syntax trees* (AST) out of them. If an AST is built, it can be traversed by the tool while performing specific checks on the tree nodes. The checker can also access other nodes reachable from the node being analyzed so gain additional knowledge of the context.

Semantic tools build upon the syntactic tools by gaining even more understanding of the analyzed code. Analysis is so in-depth that the tool has full understanding of the context and interprocedural flow. This is especially evident in compilers, such as CLANG, but also observed in more advanced static analysis tools, like CPPCHECK with its data-flow tracking. Some analyzers may even have in-depth knowledge of the language being analyzed, and thus take into account such language features when tracking variable values. One example of this is the C++ operator `new`, which can never return a null pointer due to it always throwing an exception in case of memory allocation failures. An analysis tool can then use this information and gain the knowledge that a newly allocated pointer can never be null nor is it ever aliased.

Table 3.1: The classifications of various defects into analysis levels.

Defect class	Lexical	Syntactic	Semantic
Use of dangerous function	✓		
Result of assignment as condition	✓		
Non-virtual destructor in virtual class		✓	
Non-explicit single-argument constructor		✓	
Context-sensitive memory leak			✓
Null pointer dereference			✓

3.3 Source Code and Bytecode

Static analysis tools can be designed to analyze source code, intermediate bytecode, and even binaries. Typically, in C and C++ software, the analysis is focused on the source code as analyzing machine-specific instructions loses valuable semantic information. In programming languages providing their own form of intermediate bytecode, such as Java, an analyzer can utilize this data directly and thus rely completely on the Java compiler itself to parse the source code correctly. This makes it easier for tool implementors to build these tools and focus on the more important task of code analysis.

Unfortunately, relying on parsing the source code of C++ programs can cause parsing problems due to imperfect standards compliance. To circumvent this obstacle, Merz, Falke, and Sinz [48] implemented a low-level bounded model checker (LLBMC[1]) extending the LLVM compiler framework. LLBMC uses the intermediate bytecode representation output by the LLVM compiler to create an even more simplified intermediate form that is after various transformations passed on to a model solver.

3.4 Analysis Methods

Static analysis tools are often built by separating the detection engines from the actual rules and patterns which define the problem cases we are looking for. The detection engines can be built using various techniques, such as simple pattern matching or more complex data-flow and model checking methods. Tools can also contain multiple detection techniques, which is the case, for example, in FindBugs, where pattern detectors can be implemented with and without control- and data-flow sensitivity [34]. In the following sections we will describe several of the techniques used in these tools.

[1] http://llbmc.org/

3.4.1 Pattern Matching

Pattern matching is the simplest form of static analysis. Basically, the idea is to look for predefined patterns within a character or token stream. The GREP command line utility, found in many Unix-like operating systems, can be considered as a very simple pattern matcher. With this utility, we can create regular expressions in order to find unwanted patterns within source code.

We can, for example, search for the pattern "gets," which lets us know of possible uses of the unsafe standard C function `gets`. The problem with this, though, is that any string containing the substring "gets" would be reported as a positive match. Let us consider the case in Listing 3.7 where even if the search is refined to only match against full words, strings within comments would still be reported. Fundamentally, the issue here is that the "gets" pattern only considers the character stream and so lacks any detailed information about semantics or context from where it was found [15, 16, 59]. The CPPLINT[2] tool is basically a character stream pattern matcher, which uses regular expressions to discover possible source code defects.

Listing 3.7: Example case where simple pattern matching can mistakenly report use of the unsafe `gets` function.

```
// don't use gets here
char buf[10];
fgets(buf, sizeof(buf), stdin);
```

To get around the problem of context-insensitivity, an improvement upon this method is possible. Instead of matching against a character stream, we match against the token stream given by lexical analysis. This is how many tools, such as ITS4 [62] and CPPCHECK[3], work for cases involving solely pattern matching. FINDBUGS [34] uses a similar method to scan for patterns within the bytecode stream.

3.4.2 Data-flow Analysis

Data-flow analysis methods have been thoroughly researched in the context of compilers and compiler implementations [2]. The same methods can be applied to static analysis tools. Some examples of tools implementing data-flow analysis techniques are FINDBUGS [34] and CPPCHECK.

[2] http://google-styleguide.googlecode.com/svn/trunk/cpplint/cpplint.py
[3] http://cppcheck.sourceforge.net/

The data-flow analysis problem is typically solved with an iterative graph algorithm [41]. In compilers, this method is often used to find dead code and to perform various types of optimizations, such as *constant propagation*. Constant propagation is an optimization method where expressions that have the same value everywhere are replaced with the same constant value [2].

Data-flow analysis is a general method where a set of *data-flow values* are associated with each program execution point. Specifically, the values in this set are abstracted depending on the analysis. This makes the analysis more efficient as we only need to keep track of data relevant to it.

The way data-flow analysis knows what values variables and function parameters can contain is called *value range propagation* [53]. Value range propagation helps the analyzer in tracking the range of values going through functions. Another related value tracking functionality is *taint propagation*. It detects what values can be received from the user and are thus potential security problems, e.g., `printf` format strings read from user input.

One typical example of a data-flow analysis is the analysis of *reaching definitions*. For this purpose, we define the data-flow values to be the latest assignments to every variable seen from a specific program execution point. This allows us to know what each variable contains at that particular point in the execution. Using this information we can perform many useful checks, such as detecting buffer overflows and null pointer dereferences. [2]

A simple example of using reaching definitions to find buffer overflows is demonstrated in Listing 3.8. Here, a buffer of 10 elements is defined at line 1, which is later accessed using an index retrieved from an external source on line 3. The problem occurs when the index is exactly 10 due to the incorrect conditional expression at line 5. This is a typical off-by-one error where the last valid index is `sizeof(buf)-1` due to being zero-based. Thus, knowing what possible values the variable `i` can have at line 8 provides us with an opportunity to warn about this defect at compile-time.

Listing 3.8: Detecting a buffer overflow with data-flow analysis.

```
1  char buf[10];
2
3  int i = get_index_from_user();
4
5  if (i > sizeof(buf))
6      return '';
7
8  return buf[i];
```

Programming languages featuring arbitrary pointer arithmetic can make practical data-flow analysis considerably more difficult due to *pointer aliasing*. With typical pointer arithmetic, a programmer can create pointers and point them to any conceivable addresses limited only by the address space. This makes analysis difficult as two pointers can therefore point to the same address, or any address for that matter, which means that a program can modify memory however it pleases. Cases like these are especially important when tracking pointer use after being freed, because aliasing makes it difficult to know if and when pointers become invalid. Pointer analysis is not a solved problem, though, as scalability and precision issues are prevalent these types of pointer analysis tasks. [33]

3.4.3 Abstract Interpretation

Abstract interpretation is a formal framework for making semantic approximations of programs for the purpose of static analysis. The basic idea behind the method is to choose a suitable abstraction and apply that abstraction to the program code. This operation will remove the uninteresting parts of the program in order to make the analysis more manageable. The freedom to choose an abstraction provides other useful benefits, such as ensuring that an infinite analysis problem becomes finite and thus solvable. An abstraction like this is certainly less precise than the concrete problem, but it still provides us with useful information and in feasible time. Another interesting feature of abstract interpretation is that the analysis is sound with respect to the abstraction, i.e., it will find all problems evident in the abstracted program. [19]

Using abstract interpretation requires one to define the abstraction with the help of an abstract domain. The abstract domain is defined as $D = (L, \leq, \bot, \top, \sqcup, \sqcap)$, where

- $(L, \leq)$ is a partially ordered set with the relation $\leq$,
- $\bot$ is the least element ($\sqcap L = \bot$),
- $\top$ is the greatest element ($\sqcup L = \top$),
- $\sqcup$ is the join (or least upper bound) operator, and
- $\sqcap$ is the meet (or greatest lower bound) operator.

This definition forms a *complete lattice*, which is a partially ordered set where each subset $X \sqsubseteq L$ has a least upper bound $\sqcup X$ and a greatest lower bound $\sqcap X$.

The join operator is conveniently analogous to the semantics of program branching. That is, a branch in the program can be interpreted with a join operator, thus ensuring that the resulting element after either branch is the smallest element greater or equal to both original elements. The meet operator can be defined, but is not required for our examples. Due to the join operator, these abstractions are flow-insensitive, which means that the control sequence is ignored and thus do not require complex control-flow handling. Instead, all possible sequences are considered, which may lead to false positives in case of implausible control-flows. [16, 59]

Let us consider the example in Listing 3.9. This example uses a simple abstraction of value signs to detect the possibility of a division by zero. We use the same abstract domain as Slaby [59] to represent the signs of values. The abstract domain elements are thus $L = \{\oplus, \ominus, 0, \top, \bot\}$. This abstraction is similar to the one introduced by Rosendahl [54], except it explicitly introduces the elements for uninitialized ($\bot$) and unknown values ($\top$). This abstract domain forms the lattice illustrated in Figure 3.3. The lattice is ordered using the relation of subsets ($\sqsubseteq$), i.e., the least element is the empty set ($\bot = \varnothing$), and the greatest element contains all values of the domain ($\top = \{\oplus, \ominus, 0\}$).

With this simple abstraction, we can try to answer the question if a division by zero occurs in our example. Essentially, we pose the question: is $n = 0$? Through abstract interpretation, we arrive at an answer that maybe it is, i.e., $n = \top$, which in our chosen abstraction represents an unknown but initialized value. In this particular case we are uncertain if a division-by-zero defect is present, but a static analysis tool making this analysis could report this as a potential issue.

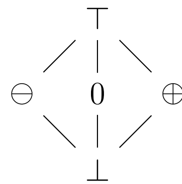

Figure 3.3: Lattice of the sign abstract domain.

More complex cases, such as arbitrarily long loops, can be modeled with the help of *fixed point* calculations. In addition to a fixed point, abstractions with infinite height require the use of a so called *widening operator* (∇). This operator was introduced by Cousot and Cousot [19] to help solve the infinity problem by ensuring loop iterations terminate. For example, for integer intervals, the infinite sequence $([0, 0], [0, 1], \dots, [0, \infty])$ could be modified to be finite, e.g., $([0, 0], \dots, [0, 100], [0, \infty])$. Use of the widening operator makes

Listing 3.9: Example program containing a possible division by zero.

```
1  int f(bool c)
2  {
3      int n = 0; // n = 0
4      int a = 10; // a = ⊕
5      int b = −20; // b = ⊖
6
7      if (c)
8          n = n + a; // n = 0 + ⊕ = ⊕
9      else
10         n = n ∗ b; // n = 0 · ⊖ = 0
11
12     // n = ⊕ ⊔ 0 = ⊤
13
14     return a / n; // Is n = 0?
15 }
```

analysis more imprecise, but presents a reasonable trade-off to ensure the analysis is completable in finite time.

The development of new numerical abstract domains has been of interest to academic researchers. In addition to the original interval abstraction by Cousot and Cousot [19], several new abstractions have been constructed that have clear advantages over it. These new abstractions are more precise while still retaining a sensible runtime requirement. One of these is the octagon abstraction [49], which is able to capture relations between two variables thus being useful for array bounds checking. Another even more recent abstraction is the pentagon abstraction [47]. This abstraction is notable for having better complexity guarantees than the octagon abstraction.

3.4.4 Symbolic Execution

Symbolic execution, or *symbolic simulation* [16], is a method of simulating every possible execution path of a program. Instead of simulating the execution by considering every possible variable value, the variables are represented as symbolic expressions. The analysis tool can then answer queries based on these symbolic expressions using *satisfiability module theorem* (SMT) solvers. This effectively enables the tool to consider every possible value for a variable with a single program execution thus reducing computational requirements.

To demonstrate symbolic execution, let us consider the function `sum` in Listing 3.10 that computes a sum of three variables. The values of each

variable for a single execution of this function are shown in Table 3.2. The same case when performing a symbolic execution is detailed in Table 3.3, which shows the symbolic values that are stored in a *symbolic memory* [59]. In this case, symbolic execution gives us the necessary insight about the variables and their relations, so that we can instantly see that the function returns the sum of $a + b + c$.

Listing 3.10: Function returning the sum of three values.

```
1 int sum(int a, int b, int c)
2 {
3     int x = a + b;
4     int y = b + c;
5     int z = x + y − b;
6     return z;
7 }
```

Table 3.2: Normal execution of `sum(1,3,5)`. A question mark denotes an uninitialized value and a dash denotes an unchanged value.

Line	x	y	z	a	b	c
1	?	?	?	1	3	5
3	4	-	-	-	-	-
4	-	8	-	-	-	-
5	-	-	9	-	-	-
6	return 9					

For more difficult cases, such as branches and loops, it is not sufficient to consider only the variable states. Branching is handled by remembering all previously encountered branches and the paths taken at those points. The branch choices are stored as a *path condition*. The path condition is a boolean expression over the symbolic input that is updated at every branching point. [42, 59]

In practice, symbolic execution of larger programs becomes difficult due to the size of the problem space. Symbolic execution suffers from path explosion similarly to other execution exploration methods due to program branching.

Table 3.3: Symbolic execution of `sum`(α_1,α_2,α_3). A question mark denotes an uninitialized value and a dash denotes an unchanged value.

Line	x	y	z	a	b	c
1	?	?	?	α_1	α_2	α_3
3	$\alpha_1 + \alpha_2$	-	-	-	-	-
4	-	$\alpha_2 + \alpha_3$	-	-	-	-
5	-	-	$\alpha_1 + \alpha_2 + \alpha_3$	-	-	-
6			return $\alpha_1 + \alpha_2 + \alpha_3$			

Loops are even more troublesome as they can make the execution tree infinite. Several solutions have been proposed in order to minimize the effect of these issues. Path explosion can, for example, be alleviated by executing branching paths in parallel. Another solution is to handle interprocedural symbolic execution by summarizing each function. [59]

3.4.5 Model Checking

Model checking is the analysis of a program represented as a finite-state automaton. This formal method for performing analysis is closely related to program verification where aspects of the program are compared to known specifications. For the purpose of model checking, the finite-state automaton is considered as a model.

Many defects can be detected with model checking. To name an example, the double-free problem, i.e., the freeing of allocated memory twice, is detectable with a model checking rule. An example of this rule is illustrated in Figure 3.4, where the state machine enters an erroneous state in case a memory address is freed twice. In this simplified example, the model begins at state q_0 where two transitions are possible. If the execution path contains the `free` operation, the state transitions to q_1. Otherwise, the state remains the same. At q_1 the transitions are the same as in q_0, thus when a second `free` is encountered the state transitions to q_2. Finally, being at q_2 means that a memory address has been freed twice and therefore an error is triggered.

The previous simplified example is only a small part of the whole model. The task of building a model can be difficult as knowing what aspects of the programs should be incorporated is not straightforward. One approach by Fehnker et al. [28] is to constructor a labeled graph from the control-flow graph. An implementation based on this approach was found to be feasible

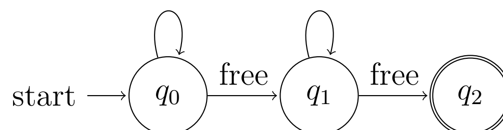

Figure 3.4: A finite-state machine detecting the case of double-freed memory. [16]

even on very large code bases [27].

In practice, models can be represented either explicitly or in a symbolically. Explicit models contain all program states that are discovered through enumeration. Using explicit models can be impractical due to the demanding memory requirements. To overcome this issue, one possible solution is to store states symbolically. This means that interdependent values are represented as formulaic equivalences, e.g., $a \Leftrightarrow b$ means that state variables a and b will always contain the exact same value. [59]

Bounded model checking is similar to model checking, but instead of completely evaluating the model, a bounded approach is taken. This method is orthogonal to unbounded model checking as both methods can provide information about different kinds of program properties. A bounded model checker works by restricting the state-space exploration to a predefined depth. If that depth is reached during analysis, one of two separate contingencies may take place: either the maximum depth is increased and the exploration continues, or the exploration is simply terminated. In case the maximum depth is reached and the exploration is terminated, the analysis is typically left incomplete. The low-level bounded model checker project (LLBMC) is an implementation of a bounded model checker for intermediate program bytecode. [59]

3.5 Custom Rules and Checks

Static analysis tools can be divided into extensible and non-extensible tools. Many simple tools will only check for rules implemented by the author of the tool, which is useful, but limits the usefulness of the tool in case of more specific needs. Some of the more advanced static analysis tools provide ways of implementing additional custom checks using tool-specific rules [16]. Separating the analysis algorithms from the actual rules being checked makes static analysis tools extensible and thus more versatile. In addition to customized rules, some tools support annotations that are embedded in the source code itself. Both methods, source annotations and custom checks,

provide useful information to the static analyzer in order to reduce false positives and to improve the results by checking for increasingly relevant aspects of the code [66].

The precursor to the Coverity static analysis tool, the XGCC extensible compiler, uses a high-level state-machine language called METAL to give users a way of writing custom checks [23]. Due to being a high-level language purposefully created for the purpose of analyzing code, it provides users of the tool a more straightforward way of implementing custom checks compared to implementing them directly as C extensions to the XGCC compiler. These custom checks are linked together with the main XGCC tool in order to be executed along with it. When the tool is run, it translates source code functions into its own representation and applies all checks on every possible execution path.

The METAL language is designed to be very extensible and to be a convenient way of implementing custom rules and checks without requiring detailed knowledge of compiler technology [31]. Listing 3.11 shows an example of a custom checker written in the METAL language detecting the use of freed memory and double freeing. In this example, the tool represents the call to `kfree(v)` as the initial state that then transitions to the state `v.freed`. If after the latter state the same pointer is then dereferenced (`*v`), the state transitions to `v.stop` while simultaneously giving an error about a use after free. The double free check works in an equivalent way.

Having an extensible framework gives users of the system the possibility of checking for a variety of different problems in source code. Even though it supports custom checks, using them does not get rid of false positives. To combat them, XGCC implements pruning of non-executable paths and ranking of results to improve the quality of output given to users. [31]

Listing 3.11: Free checker implemented with the METAL language.

```
1 state decl any_pointer v;
2
3 start: { kfree(v) } ==> v.freed;
4
5 v.freed: { *v } ==> v.stop,
6     { err("using \%s after free!", mc\_identifier(v)); }
7 | { kfree(v) } ==> v.stop,
8     { err("double free of \%s!", mc\_identifier(v)); }
9 ;
```

These tools, the XGCC compiler and the METAL language, have been used

for finding non-security as well as security related problems in operating system kernels [4]. A major part of the security related checking was a range checker tracking the input originating from the user through kernel execution paths. Tracking this type of input is useful as it can often be passed to system APIs without sanitizing it. Thus analyzing these situations and finding issues in them result in many security improvements.

The Microsoft C++ compiler has integrated support for many types of static analysis checks. Furthermore, the compiler supports an elaborate source annotation language[4] (SAL) that is used by Microsoft in the system libraries. Larochelle and Evans [44] experimented with a similar solution of annotating system libraries in order to improve the buffer overflow detection rate in their LCLint static analyzer. A few examples of these system header annotations are shown in Listing 3.12 where the parameters of the various character output functions are annotated.

All the example functions use the `_Check_return_opt` annotation specifying that checking the return value is optional. This helps the static analyzer reduce false positives by letting it skip the checking of missing return value checks for these functions. The `printf` function takes a format string input parameter, as annotated by `_In_z_` and `_Printf_format_string`, and a variable number of arguments containing the values of said format string placeholders. Differing somewhat from the previous function, the `putc` function outputs a character to a `FILE` handle. In this case, though, the handle serves both as an input and as an output parameter, as seen from the `_Inout_` annotation, due to the function mutating the file handle.

Listing 3.12: Annotations as used in the Microsoft `stdio.h` header file.

```
1 _Check_return_opt_ _CRTIMP int __cdecl printf(_In_z_
      _Printf_format_string_ const char * _Format, ...);
2 _Check_return_opt_ _CRTIMP int __cdecl putc(_In_ int _Ch, _Inout_ FILE *
      _File);
3 _Check_return_opt_ _CRTIMP int __cdecl putchar(_In_ int _Ch);
4 _Check_return_opt_ _CRTIMP int __cdecl puts(_In_z_ const char * _Str);
```

In addition to the aforementioned annotations, SAL supports many additional types of annotations for a multitude of purposes. As can be seen in the previous example, these annotations are very useful for improving program security and robustness. Annotations like these make checking for buffer overflows in system libraries practically free from the user point of view. The

[4]http://msdn.microsoft.com/en-us/library/ms182032.aspx

same annotations are also available for 3rd parties and are documented in the MSDN article referenced earlier.

The Clang compiler has support many of the language extensions implemented by GCC. One very relevant extension for static analysis is the attribute system which gives programmers the ability to specify custom attributes for functions and variables. The `nonnull` function attribute is used by the compiler for simple cases where obvious null pointers are passed to these functions. What is interesting is that the static analyzer also understands this attribute and thus uses it in more complex null pointer checks. Other attributes, such as `noreturn` defining a function as never returning, can be used to help out the static analyzer to perform even better.

Chapter 4

Analyzing Modern C++ Software

This chapter will introduce some of the problems associated with the analysis of large-scale modern C++ software. These problems include aspects such as language complexity, tool scalability, and management of the results. We will also go through a selection of tools and briefly mention their capabilities. A summary of their capabilities and features will be presented at the end of the chapter.

4.1 Understanding Code

C++ is a multi-paradigm programming language originally developed by Bjarne Stroustrup[1] in the early 1980s. The language supports object-oriented programming, generic template programming, and functional programming paradigms. C++ is also backwards compatible with C, which does not help its role in the analysis either. As the language supports arbitrary pointer arithmetic and very lax control-flow manipulation using instructions such as `goto`, it prevents analysis tools from making accurate representations of the semantics [62]. Thus, due to the complex nature of C++, it is not an easy target for static analysis. This is unfortunate, because the ability to understand the analyzed code is critical to the tools success [16].

The complexity of parsing C++ has led to the apparent lack of fully standards compliant language analysis tools. This is evident at least in the world of open-source tools, where several tools face problems parsing everything correctly. In languages where introspection is possible, tools do not have to parse source code themselves. Instead, tool authors can focus on

[1] http://www.stroustrup.com/

making tools with the help of existing language facilities. Python provides a module for manipulating the abstract syntax tree directly, thus making tools such as PYFLAKES[2], PYLINT[3], and PYCHECKER[4] possible.

Currently, existing C++ tools mostly use their own parsers, which is not ideal as these parsers often lack the means to handle some parts of the standard language. This can be an issue when using heavily template-based libraries like BOOST[5] and the standard template library (STL) itself. Parsing issues can often mean that the tool misinterprets the syntax and semantics of a program. Thus, full compatibility with the language standard is very useful for a tool in order for it to be able to analyze large-scale software.

Recently, the LLVM project[6] and the CLANG[7] compiler, have become an interesting development in the world of compilers and static analysis tools. CLANG provides a C++ compiler with full standards compliance[8]. In addition to a standards compliant compiler, it comes integrated with a static analyzer[9]. This analyzer is built on a modular framework and is capable of performing analysis on C++ source code. What this means, is that we now have a way of analyzing code with the precision of a compiler in addition to having a framework for implementing custom checks for our domain-specific needs. [17, 45, 67]

Another way of approaching the language barrier is to not analyze the source code. Instead, some tools are able to directly analyze intermediate bytecode generated by compilers. In the case of LLVM, a static analysis tool called LLBMC is able to perform its analysis on the intermediate bytecode produced by the compiler CLANG family of compilers. This enables to tool to be generic in a sense and support many programming languages at once. [58]

4.1.1 Language Extensions

While there is generally only one standard language specification, compiler vendors also provide non-standard extensions with their compiler. These extensions are sometimes required due to the platform-specific implementation details of bundled standard libraries. Another common reason is the desire to include features included in a working draft of the next version of a language

[2]https://pypi.python.org/pypi/pyflakes
[3]http://www.pylint.org/
[4]http://pychecker.sourceforge.net/
[5]http://www.boost.org/
[6]http://llvm.org/
[7]http://clang.llvm.org/
[8]http://clang.llvm.org/cxx_status.html
[9]http://clang-analyzer.llvm.org/

specification. This is often implemented by the compiler vendor as language extensions for the current compiler. Unfortunately, these extensions are not always perfectly compatible with the future standard as working drafts tend to change.

The Microsoft C++ compiler supports many extensions due to its reliance on platform-specific Windows libraries. Unfortunately, sometimes the use of these extensions leak outside their required areas, which in turn makes using only standard language features more difficult. This has created a problem with software vendors developing cross-platform products. As an example, the Microsoft compiler supports the `sealed` keyword, which is identical in functionality to the `final` keyword in the next C++ standard. Another example is the lack of Unicode file name support in standard `fstream` classes due to the reliance on UTF-16 on Windows platforms.

The creators of the COVERITY products have extensive experience with these issues. Bessey et al. [9] explain the many issues faced by them during their development of static analysis tools. The fact is that users of these tools are not often incentivized to change their code to conform with any form of standard language, and it is therefore the responsibility of the tool vendor to support the very divergent nature of different C++ compilers. [9]

4.1.2 Project Configuration

Introducing static analysis tools into old code bases is a source of even more practical obstacles. One specific problem is the organization of files and the management of file inclusions across compilation units. Building software requires the specification of compiler and linker flags in order to aid them in locating files within the file system [59]. In some cases, such as the Microsoft C++ compiler, the static analysis tool comes built-in with the compiler. This alleviates the need to specify compiler flags multiple times for both the compiler and the analysis tool and so makes the tool deployment significantly more effortless. In any case, static analysis tools should be able to support any build system in place at large software projects. Otherwise, tools would have to be executed manually, which makes day-to-day use impractical and a burden on developers.

In order to fully analyze production software, static analysis tools should also be able to analyze 3rd party libraries and headers. Without the information and implementation details of these libraries, the analysis tool is unable to completely understand the program structure. Some help can be provided by the way of annotations, which is the case in the Microsoft C++ system headers, but annotations can only provide information related to the boundaries between system and non-system code. In practice, these issues are

completely ignored due to the unavailability of system level source code and the difficulty of modifying such code when available [59]. If system libraries generate warnings, they are typically not going to be fixed, only suppressed.

4.2 Scalability and Precision

In mature software, the code base is typically relatively large and complex. The size of the code base can become a problem in case it is meant to be analyzed. Depending on the depth of the analysis and the algorithms used by the static analysis tool, the large size of the code base may prevent the tool from finishing its analysis at all. This is often the case with model checkers and program verifiers, which require a somewhat complete mathematical model of the program. Other types of tools use internal data structures, such as abstract syntax trees, that represent the parsed information. The size of these structures may also lead to considerable and sometimes impractical memory usage patterns. [59]

While memory usage is often a problem with large code bases, so are computational costs. The type and thoroughness of the analysis largely determines its computational cost and complexity. Global analysis, i.e., analysis that takes into account intra- and interprocedural data- and control-flows, are considerably more computationally expensive than simpler analysis methods, such as pattern matching with regular expressions. The benefit of a more complex analysis is that it often detects issues that are impossible to detect otherwise. Static analysis tools are thus left with the choice between scalability and precision. [14, 16]

4.3 Managing the Results

Using these tools in new projects is only a matter of tool configuration and decisions on conventions. Configuring such tools to run automatically make them a constant part of the build process. In a new project, finding new types of warnings only mean that they are either disabled or made part of the existing coding conventions. These additions are then part of the natural evolution of the project-specific coding conventions. The use of many different tools is also encouraged in new projects due to the small number of warnings. To give these warnings visibility, they should be combined and presented in reports suitable for developers. [63]

When static analysis tools are integrated into mature projects, the large number of warnings often overwhelm developers. As is often the case with

old software, its code may have been developed using many different coding conventions and styles. Thus, introducing static analysis tools into this process will initially give many superfluous results, and the process of reviewing and filtering them is a manual and laborious task. Prioritizing these issues is important in order to avoid developers being discouraged to use static analysis tools. What often follows from this prioritization process is that many of these issues are found to be low-impact, but they should still be reviewed. However, sometimes these issues can not be fixed and should thus be removed from further analysis reports. [5, 37]

One way to work around the influx of the initial warnings is to make them the baseline reference point to which any future results are compared. Approaching the problem like this makes future results useful as we are only informed of the new and fixed warnings in a code base. Although, it should be noted that this is essentially a stop-gap measure to make the initial deployment easier. Continuous integration tools provide a convenient platform for result gathering and report generation.

To further reduce the number of unimportant warnings and false positives, many tools can be configured to ignore specific issues. Static analysis tools often have command line options that specify which checks are to be made. More specific configurations can be done for tools that support source code annotations. These annotations are inserted as comments or custom attributes and they give the tools more details about the code semantics. Specific checks can often also be enabled or disabled through these annotations. Some tools even have graphical interfaces for suppressing specific warnings. In Visual Studio 2013, the user can navigate the results and choose which warnings to disable and a `#pragma` directive is automatically inserted at the correct place in the source code.

4.4 Defect Classification

It is useful to classify problems found in software as these classifications make it easier to document and manage those issues. On a broader scale, Chess [16] splits defects into two categories: *general defects* and *context-specific defects*. General defects are the ones that do not depend on the business logic and specific semantics of a program. One prominent example in this category is the buffer overflow, which can appear regardless of the type and purpose of a program. Context-specific defects depend on the program semantics and are thus harder to detect with general-purpose static analysis tools.

Even though these broad categories provide useful information, they are not always enough. More fine-grained classifications are thus needed in

order to convey more specific information about defects and vulnerabilities. Tsipenyuk, Chess, and McGraw [60] devised a taxonomy for defects called the *Seven Pernicious Kingdoms*. In this taxonomy defects are categorized into

1. Input validation and representation
2. API abuse
3. Security features
4. Time and state
5. Error handling
6. Code quality
7. Encapsulation
8. Environment

where the eighth category, environment, is actually not part of the source code at all, but instead contains all vulnerabilities attributed to outside influences.

4.4.1 Common Weakness Enumeration

The MITRE Corporation, a non-profit organization established to aid the government with technology and engineering, have created the Common Weakness Enumeration (CWE)[10]. CWE is a list of well known and defined software weaknesses, more often used by commercial software security analysis tools to advertise certain feature sets. In our case it provides a useful reference for describing a list of typical problems plaguing C and C++ software.

To complement the CWE list, the National Institute of Standards and Technology (NIST) maintains several test suites of with known vulnerabilities and defects. This collection of test suites goes by the name of the Software Assurance Metrics And Tool Evaluation (SAMATE) project[11]. Each test is constructed by having a good case, without a defect, and a bad case, with the defect being tested. Both cases are purposefully very similar in order to find out if a static analysis tool mistakes a good case as bad, thus reporting a false positive. CWE codes are given for all tests in the test suite in case one needs further information about the defect.

[10] http://cwe.mitre.org/
[11] http://samate.nist.gov/

4.5 Static Analysis Tools

Many static analysis tools exist that are able to analyze C++ source code. These tools come in both open-source and commercial forms. From our standpoint, this separation is useful as we can make general observations regarding tools within these categories. The major features of these tools are summarized in Table 4.1.

Table 4.1: Overview of static analysis tools.

Name	License	Languages	Type
Clang	Open	C, C++, Obj-C	Semantic
Coverity SAVE	Commercial	C, C++, C#, Java	Semantic
Cppcheck	Open	C, C++	Semantic
Cpplint	Open	C, C++	Lexical
Flint	Open	C, C++	Syntactic
GCC	Open	C, C++, Obj-C, Java	Semantic
Klocwork Insight	Commercial	C, C++, Java	Semantic
LLBMC	Commercial	C, C++, Obj-C	Semantic
PC-lint	Commercial	C, C++	Semantic
PVS-Studio	Commercial	C, C++	Semantic
Visual Studio 2013	Commercial	C, C++, .NET	Semantic

4.5.1 Open-Source Tools

In the world of static analysis, the available tools have typically been commercial. This is especially true for C++ software, where useful and functional open-source tools have been lacking. Lately, the field of open-source static analysis has seen some resurgence.

According to a benchmark by Chatzieleftheriou and Katsaros [14], where four open-source tools and two commercial tools were analyzed, only one of the open-source tools had similar precision to the commercial tools. This precision came with a significant cost, as the average runtime of the open-source tool, Frama-C[12], was the highest of all tested tools. Since Frama-C does not support the C++ language, it is not considered in this thesis.

Historically, compilers have not been very good in providing precise diagnostics about potential non-syntactical issues in source code. Today, after significant improvements to modern compiler technology, they are able to

[12] http://frama-c.com/

process source code with the necessary detail to analyze it for very specific issues. These analysis methods are typically enabled through compiler warning flags that are passed to the compiler during the build process.

4.5.1.1 GCC

The GNU Compiler Collection (GCC[13]) provides a suite of several compilers for different programming languages. Its C and C++ compilers are widely used in the open-source community and they come installed by default in practically every Linux distribution. It is also available for Windows platforms through distributions such as MinGW[14] and Cygwin[15]. The major difference between these distributions is the fact that MinGW builds native executables without the need for 3rd party runtime libraries. This benefit does come with the downside of a lack of full POSIX compatibility.

These compilers come with several documented flags for reporting potential issues in source code. For example, the `-wformat` flag enables format string verification for `printf`, `scanf`, etc. Another warning, which is actually enabled by default, is the compile-time division by zero check. The compiler also supports source code annotations in the form of custom attributes. One very interesting attribute is the `nonnull` attribute, which can be used on function parameters. Functions where these non-null parameter exist are checked by the compiler in case they are called with null arguments.

One of the more interesting compiler flags for analysis purposes is the `-fsyntax-only` flag. With this flag the compiler is instructed to only check for syntax errors. In addition to this, the compiler also reports warnings with this flag enabled. This is very useful for analysis as the compiler can then be used as a pure static analyzer without code generation. Unfortunately, GCC does not instantiate C++ templates when this flag is enabled, thus making it less than ideal as a static analyzer.

4.5.1.2 Clang

Clang is an open-source C, C++, and Objective-C front-end for the LLVM compiler framework. The compiler has by design a very similar set of options as GCC due to it being intentionally built as a drop-in replacement. In addition to the normal warning levels and reporting options provided by the compiler, it also features a more in-depth static analysis component. This static analyzer supports creating checks that work on the *abstract syntax tree*

[13]http://gcc.gnu.org/
[14]http://www.mingw.org/
[15]http://www.cygwin.com/

(AST) provided by the compiler. The checkers are implemented as visitors that traverse the AST and visit the nodes in it.

As a practical example, a custom checker[16] using the CLANG static analysis framework exists for the well-known Heartbleed vulnerability. This vulnerability is the consequence of a buffer over-read in the open-source cryptography library OPENSSL[17]. The checker identifies uses of the POSIX byte order conversion functions `ntohl`/`ntohs` and taints the data returned by them. Then, any unconstrained use of this tainted data in functions like `memcpy` are reported. This checker demonstrates the versatility of the CLANG static analysis framework for finding more specific vulnerabilities.

Another extension mechanism is with the help of the LIBTOOLING[18] library. This library is designed to be the basis of more full-featured tools. It provides a simple way of accessing the compiler front-end and its configuration. Several tools have been built with this library that come as part of the source distribution. `clang-check` is a command line application that functions as a simple syntax checker and interface to the static analyzer. Other useful tools, like the source code formatter `clang-format`, use the library as well.

A script, `scan-build`, works as a wrapper around the compiler front-end. This script enables one to use existing project Makefiles as the wrapper intercepts calls to the compiler replacing them with calls to the analyzer. After the code has been analyzed, the script writes an HTML file for each issue it finds. These HTML reports give clear and structured interpretations of the context leading to each specific issue.

Even though the suggested usage is through this script, one can also use the compiler directly by passing the relevant command line flags to it. This is necessary for using the analyzer on platforms where the wrapper does not work correctly, such as Windows. It should be noted that this method of invocation is currently unsupported.

The static analyzer has many built-in checks ready to use. Currently these are grouped into categories that can be enabled through command line options. These categories are

- *alpha* for experimental checks,
- *core* for general purpose checks,
- *cplusplus* for C++-specific checks,

[16]http://blog.trailofbits.com/2014/04/27/using-static-analysis-and-clang-to-find-heartbleed/
[17]http://www.openssl.org/
[18]http://clang.llvm.org/docs/LibTooling.html

- *deadcode* for detecting dead code,
- *osx* for Objective-C and Apple SDK related checks,
- *security* for insecure API usage checks, and
- *unix* for Unix and POSIX checks.

4.5.1.3 Cppcheck

Originally created by Daniel Marjamäki, Cppcheck is an open-source static analysis tool that has been part of several benchmarks [14, 24, 40, 59]. The software comes in both a command line version as well as a graphical version, shown in Figure 4.1. Both versions use the same analysis engine and thus have identical checks, although the command line version is more suitable for automated analysis due to it being easier to run in non-interactive environments.

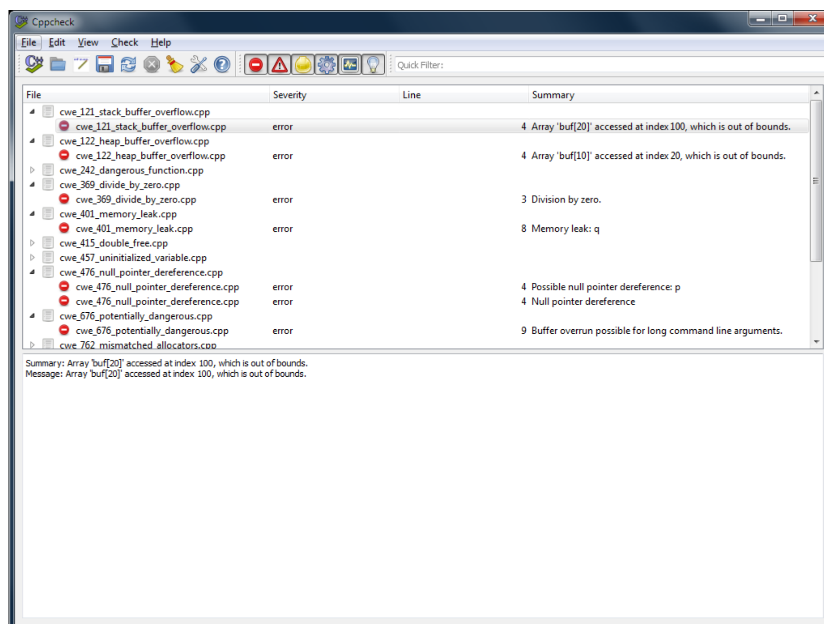

Figure 4.1: An example of the Cppcheck graphical user interface showing several warnings.

The analyzer builds a simplified *abstract syntax tree* (AST) using its own custom parser and lexical analysis, and may therefore not always conform to the latest C++ standard due to bugs or lack of features. To support more elaborate checks, the latest version (1.64) implements a generic data-flow

tracking framework. This new framework supports general-purpose context-sensitive interprocedural data-flow analysis and can be used by individual checkers. Many checkers have been modified to use this new system, although some old checkers still use their own specific data-flow tracking. The framework also performs abstract interpretation when tracking values in loops.

There are two very different extension mechanisms in CPPCHECK for creating custom checks. One way to create them is by modifying the source code to create new C++ classes containing the desired checks. These classes work as visitors on the tokenized AST stream. In addition to this method, custom checks can also be created by specifying regular expression in an XML formatted configuration file. The regular expressions are used find defects by matching them against the token stream. Although convenient, these regular expressions do not provide the same capabilities as the source modifying method.

The analyzer comes with many different checkers. These checkers are categorized by their *severity*. Program options are available for enabling or disabling checks by their severities. These severities are

- *error* for more severe issues like syntax errors,
- *warning* for suggestions about possible problems,
- *style* for dead code and other stylistic issues,
- *performance* for some common performance related suggestions,
- *portability* for 64-bit and general platform portability issues, and
- *information* for informational messages about problems with the analysis.

CPPCHECK is also able to give even more warnings if enabled with the `--inconclusive` flag. This flag enables, as the name suggests, inconclusive checks where the analyzer might not be completely sure about the existence of a problem. By enabling this flag, some previously hidden problems might be detected while significantly increasing the number of false positives.

4.5.1.4 Cpplint

Originating from Google, CPPLINT[19] is a code analysis and style checking program that enforces the Google C++ Style Guide[20]. This program, written

[19] http://google-styleguide.googlecode.com/svn/trunk/cpplint/cpplint.py
[20] http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml

in Python, uses a mixture of regular expression matching and other line based heuristics to detect various problems in the analyzed code. It mostly detects style issues, whitespace irregularities, and various other potential problems. Due to its simplicity, it analyzes even large projects relatively fast.

As it is a style convention checking tool created for Google projects, like CHROMIUM[21], its use is somewhat limited for general-purpose analysis. Fortunately, irrelevant checks can be disabled through the use of command line flags. In addition to individual checks, whole categories of checks can be enabled or disabled. The currently existing categories are

- *build* for build and preprocessing issues,
- *legal* for missing copyright messages,
- *readability* for correct but unreadable code,
- *runtime* for runtime related issues, and
- *whitespace* for whitespace usage conventions.

4.5.1.5 Flint

FLINT[22] is a static analysis tool recently open-sourced by Facebook. It was previously developed in C++, but has since been converted to the D programming language[23]. This resulted in smaller code and better runtime efficiency due to the use of compile-time code generation features specific to D. Instead of building a tree, the tool converts source code into an array of tokens. Checkers can then traverse this token array and navigate it looking for specific tokens.

The tool is equipped with several preexisting checkers, e.g,

- *checkBlacklistedSequences* checks for blacklisted token sequences such as `volatile` while still allowing `asm volatile`,
- *checkBlacklistedIdentifiers* checks the use of blacklisted identifiers like `strtok`,
- *checkDefinedNames* checks identifier naming for illegal use of implementation reserved underscores,

[21]http://www.chromium.org/

[22]https://code.facebook.com/posts/729709347050548/under-the-hood-building-and-open-sourcing-flint/

[23]http://dlang.org/

- *checkIncludeGuard* checks for missing include guards in either `#pragma` or `#ifdef` form, and
- *checkMemset* checks for the correct order of the `memset` function parameters.

These are only five of the 25 available checkers in the initial open-sourced version.

4.5.2 Commercial Tools

Several commercial static analysis tools have been made since the creation of the first lint-type tools. These commercial tools are typically either smaller standalone tools or parts of a larger suite of tools. With the significant costs of some of these tools comes useful benefits like extensive on-site support, tool customization, and project-specific tuning for the customer. It is also typical that the tools have a more security focused marketing while also supporting many style- and performance-oriented checks. The more affordable tools listed here are smaller self-contained utilities that have simple command-line interfaces. Often times they can also be accessed from within integrated development environments.

4.5.2.1 Coverity SAVE

Coverity Static Analysis Verification Engine (Coverity SAVE[24]), is a static analysis tool developed by Coverity Inc. This analyzer is a part of a larger product suite supporting languages such as C, C++, C#, and Java. Their static analysis tool is a based on the Stanford Checker developed at Stanford University by Hallem et al. [31]. It is notable for being used by several open source projects through a common web interface[25]. The open source analysis effort was contracted by the United States Department of Homeland Security in order to improve the security of software essential to the foundation of the Internet. According to Bessey et al. [9], their tool supports a multitude of languages and vendor-specific language variations. This is important in order to analyze many legacy and proprietary software systems as they are often implemented with non-standard programming language versions.

Although the algorithms used in Coverity SAVE are proprietary and thus unknown to us, its precursor, the Stanford Checker uses a depth-first search to construct a control-flow graph by traversing the abstract syntax

[24] http://www.coverity.com/products/coverity-save/
[25] http://scan.coverity.com/

tree of the source code. The Stanford Checker speeds up interprocedural analysis by caching block level state changes. This enables the tool to skip over already analyzed blocks by using the cached state as a summary of how the block modifies the state. [31]

The tool itself is *unsound*, i.e., it does not prove the absence of bugs and problems, but instead strives to find as many bugs as possible. Even though unsoundness is an unfavorable feature in the theoretical sense, it is commonplace in static analysis tools as it makes problematic language constructs and the inevitable false positives easier to handle. It also allows tool developers to focus primarily on finding new types of defects even if the analyzer is sometimes mistaken. Earlier versions of the static analysis tool by Coverity, Coverity Prevent, used function-level summaries in its interprocedural data-flow analysis. Additionally, it also applied statical inference methods to detect possible problems in code. [3]

4.5.2.2 Klocwork Insight

Klocwork Insight[26] is a product suite made by Klocwork Inc. for code analysis, metrics, reporting, and refactoring. The tools in the suite are provided as generic desktop applications and as extensions for various integrated development environments, such as Visual Studio and Eclipse. Languages supported include C, C++, and Java, and all these can be used in the same project due to its multi-language analysis.

A previous version of the product, K7, was used to find problems in the open source Samba project. Together with the Coverity static analysis tool, they found a handful of issues, which were later fixed. Notably, K7 found more than 150 possible vulnerabilities after other tools had already analyzed the code. [29, 52]

K7 is also known to have interprocedural analysis capabilities and very useful extension mechanisms. By making extensions, customers are able to create new checks specific to their needs. In addition to the static analysis features, the product suite can also help with architectural analysis. Using code metrics and complexity analysis, the tools are able to give developers valuable information about the status of their code. This information is helpful for program understanding through various visualizations. [64]

4.5.2.3 LLBMC

LLBMC (or low-level bounded model checker) is a static analysis tool that analyses the bytecode produced by the Clang compiler. This means that

[26] http://www.klocwork.com/products/insight/

it is language independent, although, many language-specific features need explicit modeling work so that the bounded model checker is able to analyze them. By analyzing bytecode, the tool bypasses the need to specifically understand high-level programming languages, which is a useful property especially considering complex languages like C++. The analyzed bytecode is in the LLVM-IR format, which is an intermediate abstract assembler code output by the LLVM family of compilers, such as CLANG.

Before this bytecode can be analyzed, it has to be transformed into a suitable form for model checking as illustrated in Figure 4.2. First, the control flow is simplified by unrolling loops and inlining functions. Then, the bytecode is encoded into ILR, an intermediate logic representation used by LLBMC to represent programs in the form of bit-vectors and arrays. This representation is essential as it gives us state objects for memory contents. Additional state objects describe the state of the memory allocation system in order to encode logical dependencies between instructions accessing memory. This way, memory access patterns incorporating the original LLVM read/write operations are explicitly encoded in the IRL in an ordering-independent fashion. Finally, the ILR is furthermore simplified by rewriting memory access patterns in order to reduce them into bit-vector formulas for correctness evaluation through the use of a *satisfiability modulo theory* (SMT) solver. [58]

LLBMC uses a flat array-based memory model which enables it to properly support weakly typed languages, like C and C++, where a pointer can be cast to represent a pointer of basically any other type. It is thus very proficient in detecting buffer overflows, integer overflows, and memory errors [58]. To make solving the program model feasible, loops are unrolled to a predetermined iteration count and recursive function call depths are limited. The LLBMC tool has been successfully used in the software verification competition SV-COMP 2013, where it took second place in four categories [26].

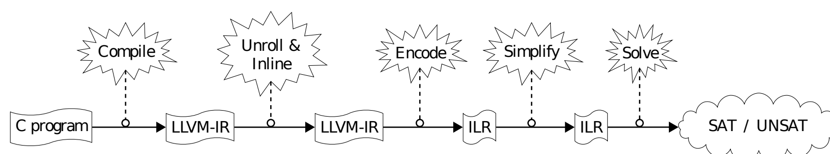

Figure 4.2: The source code transformation pipeline of LLBMC. [48]

4.5.2.4 PC-lint

PC-LINT[27], by Gimpel Software, was created in 1985 to be a DOS-based equivalent to the Unix lint tool. Initially, its purpose was to do basic type checking by comparing the used function arguments to their respective function parameters. Since then, the tool has gained several new advanced features, such as strong type checking with dimensional analysis and global analysis capabilities with interprocedural value tracking.

In the later versions, message suppression was improved by giving the users of the tool ways to hide messages in specific contexts. As a large number of warnings can be detrimental for the eventual adoption of a static analysis tool, this development proved to be very beneficial. Messages can be suppressed with command line options or within source code comments. These suppressions can be tailored to consider only specific contexts, such as C macros.

In order to provide interprocedural value tracking, PC-LINT builds a global call graph by scanning the whole source code. This is built as a multi-pass analysis by having a primary *general walk* go through the call graph while simultaneously recording every possible function argument value. In the next step *specific walks* are performed on the same call graph using the previously recorded values. Thus, each specific walk is essentially performing a simplified data-flow analysis.

PC-LINT is notable for its long history and its support for a wide variety of C and C++ language dialects. It also has built-in options for enabling detection of issues related to the MISRA[28], CERT[29], and other secure programming specifications. Additionally, a version of the tool called FLEXELINT is provided in source form to make it as portable as possible. [30]

4.5.2.5 PVS-Studio and CppCat

PVS-STUDIO and CPPCAT are both static analysis products developed by the Product Verification Systems[30] company. PVS-STUDIO consists of three distinct analyzers: 64-bit portability, multiprocessing, and general analysis. The former two are older products previously developed as separate products VIVA64 and VIVAMP. This tool can be used either through a standalone graphical interface, illustrated in Figure 4.3, or as an extension to the VISUAL STUDIO integrated development environment.

[27] http://www.gimpel.com/html/pcl.htm
[28] http://www.misra.org.uk/
[29] https://www.securecoding.cert.org/
[30] http://www.viva64.com/

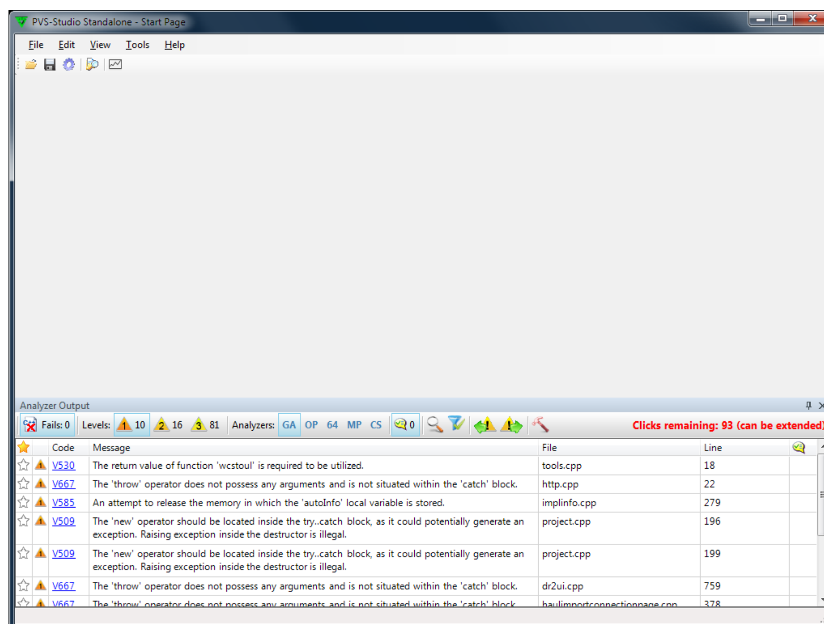

Figure 4.3: The PVS-Studio standalone user interface showing issues from DynaRoad.

PVS-Studio uses a heavily modified C++ front-end called OpenC++[31]. This C++ front-end handles the lexical analysis and parsing of the source code. Furthermore, two different choices for preprocessing source code are supported: the Microsoft compiler and Clang. It is noteworthy to understand that this choice only decides which compiler is used for preprocessing and not for general parsing. The Clang preprocessor is faster, but may in some cases fail to handle Microsoft-specific language extensions. In that case PVS-Studio will fall back to using the slower Microsoft compiler.

CppCat is a simplified version of the same product. Its purpose is to provide a straightforward interface to some of the same underlying analyzers that are contained within the PVS-Studio package. Notable changes are the omissions of the 64-bit portability and multiprocessing analyzers.

4.5.2.6 Visual Studio 2013

Visual Studio 2013[32] is an integrated development environment (IDE) developed by Microsoft supporting multiple programming languages for Windows development. Among other tools, it comes bundled with a static analyzer

[31] http://opencxx.sourceforge.net/
[32] http://www.visualstudio.com/

for C++ and C# projects. This tool was first introduced in Visual Studio 2005 and was a continuation of the PREfast static analysis tool created at Microsoft Research [16]. A sample output of the static analyzer in Visual Studio 2013 is illustrated in Figure 4.4.

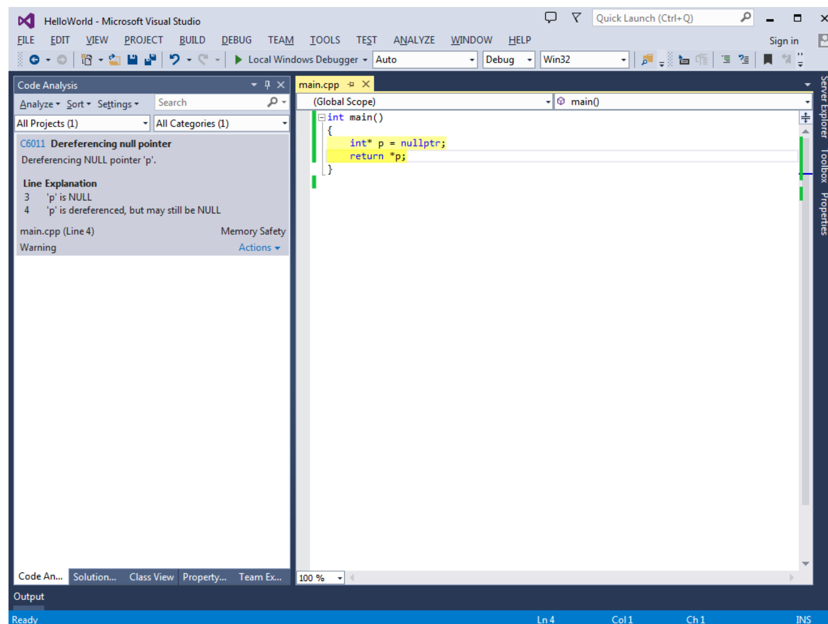

Figure 4.4: The Visual Studio 2013 static analyzer detecting the presence of a null pointer dereference.

Initially only bundled with the Windows Driver Development Kit (DDK), the PREfast analyzer supports a simple pattern matching and some data-flow sensitive checks, such null pointer dereference checking. PREfast and thus the Visual Studio 2013 static analyzers are only capable of local intraprocedural analysis, i.e., they are unable to analyze data-flow across function calls. To find simple programming mistakes, the tool detects the presence of predefined bug patterns in the abstract syntax tree. Both tools understand the Standard Annotation Language (SAL), which helps them gain additional information about program semantics. [56]

PREfast is a "faster" version of the similarly named PREfix tool. The main difference being that PREfix is able to perform interprocedural analysis thus rendering it more capable in finding software defects. This does mean that the analysis takes significantly more time, which is why the tool is meant to be run server-side. [51]

To remove the need for repeated checking of the same functions, PREfix symbolically executes leaf functions and caches function-level summaries. The

symbolic execution state is stored in a virtual machine, which is then analyzed for potential defects. Once leaf functions have been summarized, the analysis continues upwards the calling graph to the callers of these functions. [12]

Chapter 5

DynaRoad Software

This chapter introduces the target of the analysis in this thesis, the DynaRoad software. We will make a brief overview of the major features and capabilities of the software. The following sections will elaborate on the technical aspects and architecture of the software. Finally, we will go over the current build process and how we feel it can be improved.

5.1 Features

DynaRoad is a project scheduling and management software developed by DynaRoad Oy aimed mainly at road construction projects. It has seen use in several major road construction projects around the world. It is also used in schools and universities as a teaching tool for civil engineers.

The software itself is a Windows desktop application providing the user with a graphical user interface. The user is able to create new construction projects by importing mass volume data, also called a *bill of materials*, directly from preexisting Excel files. These files are generated by exporting the data from other construction design programs. When a project has been created and embedded with sufficient data, such as material types, tasks, and resources, a project designer can then use automated tools for calculating optimized mass haul plans.

The software is comprised of three distinct modules: *planning*, *scheduling*, and *controlling*. Planning module gives project designers the tools to construct a project from non-temporal tasks. Non-temporal tasks are tasks that lack scheduling information. Such tasks are adequate for mass haulage planning. The scheduling module augments these tasks with scheduling information like start times, finish times, and dependencies. With this the designer is able to plan a schedule for the whole project taking into account resource usage

and costs. When the project is finally scheduled, the third module comes into play. This module is focused on the controlling aspects of projects in progress. Visualizations of aspects such as forecasts, scheduling conflicts, and task completion degrees help project managers steer their projects closer to their plans.

Optimization is a major part of the software, which is why DynaRoad gives users two distinct ways of optimizing their project. The two methods are separate due to their use of different data for their optimizations. The first method uses linear programming to find optimal mass hauls plans by minimizing haul costs. This method ignores scheduling data, such as task timing and resource production rates, as this information is not necessary for plan calculation. Mass haul plans contain mass haul quantities and destinations for each task. Sometimes mass requires processing before it can be used and is therefore hauled through multiple locations, thus creating haul chains.

The second optimization method is very similar to the first, but instead of ignoring scheduling data, it is included. Due to scheduling data being taken into account, the optimal plan now establishes a realizable project schedule in addition to minimizing its mass haul costs. Previously, DynaRoad supported an optimization method implemented with a genetic algorithm, but that method has since been replaced with the linear optimization method explain above. [39]

Several graphical views provide the user with useful insight on the current and future status of a project:

Mass haul view
A view for visualizing the mass distribution crosscut along a road.

Mass curve view
Displays the cumulative sum of mass along a road.

Time-location view
Shows the correlation between location-based task progress and time. A demonstration of the time-location view can be seen in Figure 5.1.

Gantt view
A view showing the start and finish times of each task in a task hierarchy. This view is very common in project management software.

Resource view
Shows resource usage at various points of time. Is useful for resource overallocation prevention.

Map view

Overlays the project on a geographical map while positioning tasks and other locations at their actual real-world coordinates. Figure 5.2 shows an example output of the map view for a sample project.

Control view

Used for visualizing the status of an in-progress project. Shows tasks and their status compared to their planned schedule, i.e., is the task late, in time, or not started.

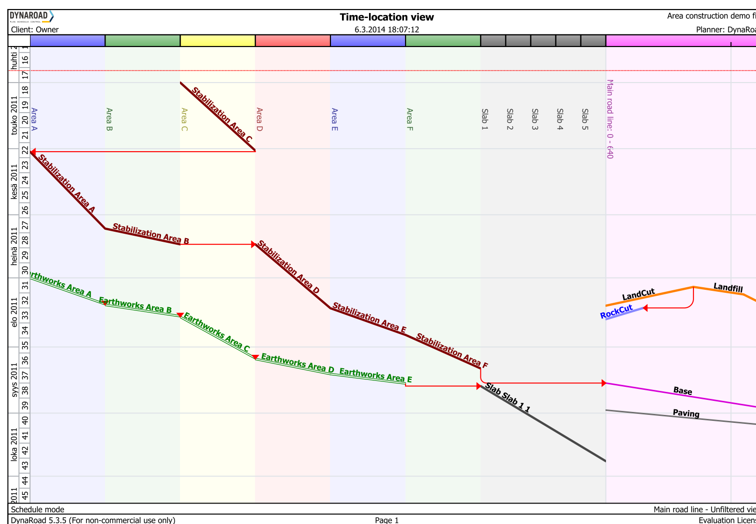

Figure 5.1: DynaRoad displaying the time-location view of a simple project.

5.2 Technical Overview

The software is comprised of two main architectural modules: the kernel and the user interface (UI). The kernel implements the domain model of the software. Each domain concept is implemented as a collection of C++ classes and other data structures, e.g., class `RoadLine` for project road lines. One factor that has remained important during the implementation of kernel functionality, is the aversion of adding UI and platform-specific library dependencies. This lack of UI dependencies is very useful in light of the fact that it makes these classes testable using automatic non-interactive testing

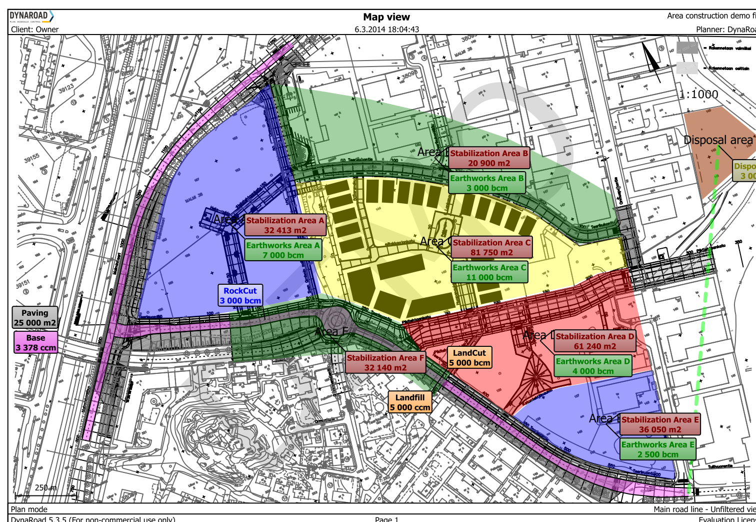

Figure 5.2: DynaRoad output of the map view showing construction areas and their locations on a real-world map.

tools. Thus, the kernel presents a self-contained public facing API to other parts of the architecture, most notably the UI.

The software is developed in C++ as a Windows desktop application. The user interface is implemented with the Microsoft Foundation Class[1] (MFC) library. MFC is an application framework for Windows programs providing helper classes for common tasks, such as 2D graphics, printing, and document serialization. MFC has a long history and is still supported and maintained by Microsoft. Newer, more modern libraries have been developed by Microsoft, but not for C++ software. Since the introduction of the C# language, libraries such as WinForms and WPF have been introduced as being more approachable ways for developing Windows applications. Due to the prevalence of legacy applications and the continuing interest in C++ application development from a performance standpoint, many applications retain the dependency to MFC. Currently, most of the interface visible to the user is implemented with MFC. The Cairo[2] graphics library is used to draw 2D graphics for vector graphics exporting needs.

In addition to the two main architectural tiers, a separate service layer provides technical services, including license management and Excel file manipulation. The service layer is tested as part of the automated testing

[1]http://msdn.microsoft.com/en-us/library/fe1cf721(v=vs.110).aspx
[2]http://www.cairographics.org/

suite. Due to the design of the libraries in this layer, they are relatively easy to test automatically.

5.3 Build Process

The current software build process is implemented as a single-part build pipeline in an instance of the JENKINS[3] continuous integration software. A normal run of the pipeline goes as follows:

1. A developer commits new source code to the source code repository.
2. The JENKINS server polls the repository and notices the new commit thus updating its local working copy.
3. JENKINS begins the build process by executing a build command with the MSBUILD project file residing in the working copy.
   (a) MSBUILD starts by compiling and linking the automated unit testing executable.
   (b) The automated test executable is run and the results are gathered by JENKINS for later reporting.
   (c) In case the tests are successful, the main executable is compiled and linked.
   (d) The main executable is packaged along with any required external data files in a Microsoft Installer (.msi) file and as a 7-ZIP[4] archive.
4. The resulting output files are archived in storage.
5. Build warnings, errors, and test results are then reported in the JENKINS build system.

5.4 Existing Analyzers

The DYNAROAD code base is analyzed by a collection of SUBVERSION[5] post-commit scripts and nightly shell scripts. The Subversion post-commit hooks are responsible for checking coding style conventions and documentation spelling errors. Due to the post-commit hooks blocking the commit procedure,

[3] http://jenkins-ci.org/
[4] http://www.7-zip.org/
[5] http://subversion.apache.org/

these particular tasks have to be done relatively quickly. Lengthy analysis scripts are run during the night when computer resources are required the least. These nightly scripts can thus analyzed code more thoroughly than scripts run as post-commit hooks. Currently, only one static analysis tool, CPPCHECK, is run nightly.

One of the most useful existing post-commit script is a script written in Perl. Perl is a programming language well-known for its usefulness in handling textual data. Basically, this script contains a list of predefines regular expressions that are matched against the source code. Any matches found by the script are reported as warnings. These warnings are accumulated in a text file, which is then e-mailed nightly to all developers. Reoccurring e-mail messages provide a great incentive for fixing any remaining issues.

A notable part of the current analysis setup is the treatment of compiler warnings in DYNAROAD. Compilers provide helpful information on suspect and possibly defective code. Modern compilers have a large number of warnings that can be enabled using command line options. DYNAROAD is compiled with the highest warning level possible and those warnings are treated by the build system as errors. This makes it impossible to introduce code that produces warnings and makes for a very clean code base.

5.5 Potential Improvements

Our existing checks have shown to be of great use for detecting potential issues in coding style and correctness. Unfortunately, using post-commit hooks is not future-proof as it makes moving between version control systems more difficult. In addition to this, it is very useful to systematically store the results of the static analysis tools for historical inspection. These historical records can be used to generate warning count trends and graphs, which helps the inspection of the software development process. A logical solution is to move the contents of all post-commit hooks into the source tree. To do this, the build files have to execute the analysis tools themselves. Additionally, due to build files typically being stored in a version control systems, the command line options for these tools are therefore also conveniently version controlled. Having the static analysis tool execution information in the build file makes it possible for developers to launch the analysis tools themselves on their local computer. This improves productivity by avoiding the need for developers to wait for the execution of post-commit hooks.

Moving the static analysis tool execution to the build file itself makes it possible to specify exactly which checks are being run and when. A continuous integration server is suitable for this task as it can be configured to execute

builds with specific command line options by creating *build jobs*. Build jobs are either run after a commit or scheduled to run at specific times in a similar fashion to the UNIX Cron daemon. This centralizes the execution of external static analysis tools onto the continuous integration server.

Further improvements to the software engineering process are possible by increasing the number of used static analysis tools. The best case scenario is to run as many tools as possible while still considering the practical time and scalability limitations of these tools. Tools having long execution times that provide very little useful information can easily be ignored.

Chapter 6

Static Analysis Tool Evaluation

In this thesis we evaluated several freely available static analysis tools in order to find out about their practical usefulness in our software engineering process. This chapter is divided into three parts: an evaluation of the detection accuracy of several tools in synthetic tests, an evaluation of real-world detection capabilities, and an investigation into build process improvements.

We chose four open-source and two commercial static analysis tools. All tools were chosen on the basis of having some level of C++ support and being either open-source or free to evaluate. The chosen tools and their respective versions are listed in Table 6.1. Note that we chose to leave some potential tools out of this evaluation due to their lack of features or unavailability for our required platforms. PC-lint currently lacks support for the latest C++ standard. CppCat is a stripped down version of PVS-Studio. LLBMC is not available on Windows platforms.

Table 6.1: List of the evaluated tools, their versions, and download locations.

Name	Version	Location
Clang	r203645	http://clang.llvm.org/
Cppcheck	1.64	http://cppcheck.sourceforge.net/
Cpplint	r119	http://google-styleguide.googlecode.com/
Flint	g9cefb19	https://github.com/facebook/flint/
PVS	5.16.10324.3	http://www.viva64.com/en/pvs-studio/
VS2013	12.0.21005.1	http://www.visualstudio.com/

In order to evaluate the defect detection efficiency of each tool, we built a synthetic test suite specifically designed to point out what each tool was able to detect. After that, we set out to gather information of the detection efficiency

in a real-world setting by doing a differential analysis on our code base. A large-scale evaluation like this requires, in addition to a well-functioning detection engine, that the tool is scalable enough to handle large amounts of source code. Finally, for each tool the possibility of integrating it into our continuous integration environment was investigated.

6.1 Synthetic Tests

Creating a suitable test suite of synthetic tests for the purpose of benchmarking static analysis tools has been shown to be a nontrivial task. The reason for this is the large variety of detection mechanisms an algorithms used in these tools, which lead to vastly different end results in detection accuracy. Ideally, such tests would contain every possible variation of the defect being tested. Unfortunately, creating such large test suites would lead to a combinatorial explosion in number of test cases. To give an example, the latest version (1.2) of the Juliet Test Suite for C/C++[1] contains 61,387 test cases that contain only 118 different defects. This means that there are, on average, over 500 test cases for each defect with different data- and control-flow patterns. [11, 20]

For our evaluation we settled for a much smaller set of defects and test cases to keep the evaluation manageable and in order to meet time constraints. We chose 19 defects from the MITRE Common Weakness Enumeration (CWE) list and 11 rules and recommendations from the CERT C and C++ Secure Coding Standards. Many of these defects were also used in the benchmark by Chatzieleftheriou and Katsaros [14], which is a similar comparative evaluation where the features of several static analysis tools were compared and tested with a synthetic test suite. An evaluation with known test cases provides valuable information on the actual detection accuracy in different defect categories.

A detailed description of each test case is presented in Appendix A. The 30 defects and programming mistakes were each contained within their individual C++ source files. These files only tested the ability of the static analysis tools to detect the presence of problems, not their absence. According to Schmeelk [55], small self-contained tests like these are *microbenchmarks*, i.e., synthetic test cases not lifted from real-world programs. As such, they only measure defect detection accuracy and not, for example, runtime performance.

The test suite was designed to contain many typical security and C++-specific problems. Instances of them have been found in our own code base in the past. These are typical problems, such as use-after-free, double

[1] http://samate.nist.gov/SRD/testsuite.php

free, memory leaks, buffer overflows, use of dangerous functions, and race conditions. Issues like these exist in both C and C++ software, where memory management is left to the programmer and strings are basically character arrays of arbitrary length. C++ provides alternatives that alleviate these exact cases, but their use is not required by the language.

It should be noted that none of these test cases contain syntax errors, i.e., standards-compliant compilers should be able to successfully compile each one of them. Both Clang and the VS2013 compilers were able to compile each case without syntax errors. Although, in some cases the compiler refused to compile obvious statically detectable problems, such as integer divisions by zero. Each tool was run with most warnings enabled as summarized in Table 6.2.

Table 6.2: Options used in the synthetic test evaluation.

Name	Options
Clang	`-Weverything -pedantic -std=c++11,` `-analyzer-checker=core,cplusplus,deadcode,security`
Cppcheck	`--enable=all --library=std.cfg`
Cpplint	`--filter=-whitespace,-legal`
Flint	N/A
PVS	All warnings (1, 2, 3) and analyzers (GA, OP, 64, MP, CS)
VS2013	Warning level 4, Microsoft All Rules preset

6.1.1 Results

A summary of the results for the CWE-part of the synthetic tests is show in Table 6.3. Another summary, shown in Table 6.4, contains the results of the synthetic tests for the CERT secure coding test cases. By looking at the results, we noticed that no tool was able to find every possible problem.

The compilers and their integrated analyzers, Clang and VS2013, together with the more complex analysis tools, Cppcheck and PVS-Studio, were able to find most of the CWE defects. Cpplint, being mostly a style and coding convention checking tool, was not able to detect most of the CWE defects. Flint does not come bundled with many checkers out-of-the-box and therefore did not perform too well in this evaluation.

Table 6.3: Defects detected by the evaluated tools. (Common Weakness Enumeration)

ID	Description	Clang	Cppcheck	Cpplint	Flint	PVS	VS2013
CWE-121	Stack buffer overflow	✓[1]	✓			✓	✓
CWE-122	Heap buffer overflow		✓			✓	✓
CWE-134	Uncontrolled format string	✓[1]				✓	
CWE-170	Improper null termination						✓
CWE-190	*Integer overflow**						
CWE-242	Dangerous function	✓	✓				✓[2]
CWE-327	Use of `rand`			✓			
CWE-369	Division by zero	✓	✓			✓	✓[1]
CWE-401	Memory leak		✓				
CWE-413	*Improper resource locking**						
CWE-415	Double free	✓	✓			✓	✓
CWE-416	Use after free	✓					✓
CWE-457	Uninitialized variable	✓[1]	✓			✓	✓[1]
CWE-476	Null pointer dereference	✓	✓			✓	✓
CWE-561	Dead code		✓				
CWE-590	Deallocation of non-heap pointer	✓					
CWE-663	Non-reentrant function	✓[2]		✓	✓		✓[2]
CWE-676	Potentially dangerous function	✓	✓	✓			✓[2]
CWE-762	Mismatched memory routines		✓[3]		✓[4]	✓	✓[3]
Total		11	11	3	2	8	12

[1] Warning issued by compiler.
[2] Function deprecated in library implementation.
[3] Does not detect the `unique_ptr` allocation mismatch.
[4] Detects only the `unique_ptr` allocation mismatch.
* Not detected by any tool.

Table 6.4: Defects detected by the evaluated tools. (CERT Secure Coding Standard)

ID	Description	Clang	Cppcheck	Cpplint	Flint	PVS	VS2013
ARR32-CPP	Use of invalidated iterator		✓				
DCL17-CPP	Passing large object by value					✓	
ERR09-CPP	Heap-allocated exception				✓		
ERR37-CPP	Deprecated throw specifications	✓[1]			✓		✓[1]
EXP05-CPP	Use of C-style cast	✓[1]		✓			
EXP31-CPP	*Assert with side effects**						
EXP35-C	Return address of temporary	✓	✓			✓	✓[1]
FLP30-CPP	Floating point loop counter	✓					
INT11-CPP	Cast between pointer and integer					✓	
OOP32-CPP	Non-explicit constructor			✓	✓		
OOP34-CPP	Non-virtual destructor		✓				
Total		4	3	2	3	3	2

[1] Warning issued by compiler.
* Not detected by any tool.

All tools were capable of finding instances of at least some violations of the CERT secure coding guidelines being tested. These tests require a deeper knowledge of the C++ language and are thus difficult to detect without dedicated checkers. It should be noted that some of the guidelines are more stylistic in nature and therefore do not have widespread support among static analysis tools.

6.2 Differential Analysis

The second part of the evaluation was a comparison of the tool reported warnings against actual defects found and fixed in various revisions of DynaRoad. For this evaluation, we chose six consecutive versions of the stable branch of DynaRoad. All versions were minor point releases of the 5.3 version and all of them contained various types of bug fixes and minor feature additions. For each version the number of fixed known crash defects was recorded and kept as a reference.

The issues reported by each tool were logged by a version-by-version basis. Comparing the logged issues from release version to release version gave us insight in how these tools are able to detect real-world problems. Similarly to the synthetic evaluation, Table 6.5 shows that our goal was to enable as many checkers as possible for each tool.

Table 6.5: Options used in the differential analysis evaluation.

Name	Options
Clang	`-Wall, -analyzer-checker=core,cplusplus,deadcode, security`
Cppcheck	`--enable=warning,style,performance,portability --library=std.cfg --library=windows.cfg`
Cpplint	runtime
Flint	N/A
PVS	All warnings (1, 2, 3) and analyzers (GA, OP, 64, MP, CS)

For each tool, the differences in the generated reports were searched for known crash defects in specific program versions. A similar version-by-version evaluation was done by Wedyan, Alrmuny, and Bieman [65] for Java-based static analysis, where only 3% of reported issues were detected by the tools. In our case, this type of evaluation was beneficial as it did not require us to have perfect knowledge of every existing problem in the code base. This is especially useful for mature large-scale software, such as DynaRoad.

So, we settled for an imperfect knowledge evaluation by analyzing different version of the same software. The information we gathered gave us a general feel on how useful these tools are for finding actual bugs. We then classified these results into *true positives* and *false negatives*. True positives were defects detected by the tool and fixed in the next program version. False negatives were defects not detected by the tool that were fixed in the next program version. One should note that we purposefully left out problems found by the tools that were not fixed as we needed a clear picture on how these tools could have eliminated known issues during development.

6.2.1 Results

All tools reported many warnings of varying severities as shown in Table 6.6. No results were available for VS2013 because the code base would have required modifications to accommodate the newer compiler. Comparing the output of the tools, we found that none of them was able to detect any of the crash defects fixed between versions. No previously unknown crash defects were found either. The number of known crash defects in each version is summarized in Table 6.7 with detailed descriptions of each bug in Appendix B.

Table 6.6: Number of warnings for each DynaRoad version.

Version	Clang	Cppcheck	Cpplint	Flint	PVS
5.3.0	150	77	236	264	3032
5.3.1	152	77	237	266	3041
5.3.2	152	77	239	266	3046
5.3.3	154	82	241	264	3049
5.3.4	154	82	242	264	3051
5.3.5	159	91	250	267	3071

To find out what the most frequent warning issued by each tool was, we grouped and sorted the warnings from DynaRoad version 5.3.0 by their category. As can be seen in Table 6.8, the most frequent warnings were not very relevant to our analysis. Clang warned mostly about class constructor initialization lists being in the wrong order. The most frequent warning issued by Cppcheck was about function parameters being passed by value instead of constant references. Cpplint being a style-focused checker warned mostly about the use of C-style casts. Flint was previously modified to suppress the association header warning due to our use of precompiled headers, but despite the modification, it was still the most frequent warning. Interestingly, PVS-Studio gave many warnings about code fragments not being analyzed, which

Table 6.7: Number of known crash defects in each DYNAROAD version.

Version	Known Crash Defects
5.3.0	3
5.3.1	3
5.3.2	4
5.3.3	1
5.3.4	1
5.3.5	6

according to the vendor, are the result of erroneous template instantiations, errors in the C++ preprocessor, or a problems in the analyzer itself.

6.3 Build Process Integration and Reporting

We integrated three static analysis tools into our build: CPPCHECK, CPPLINT, and FLINT. CLANG integration was postponed as eliminating all the syntax errors reported by it would have required several modifications to our code base. For similar reasons we also left out VS2013, which would have required us to upgrade our code base from an older version of the compiler. PVS-STUDIO was not integrated as its use in production requires a commercial license.

To begin, we modified the project build file by adding a custom build target for these checks named `Check`. This target depends on three other targets that each execute their respective static analysis tool. One specific detail that had to be noted between these tools was their capability of doing global analysis. If a tool was capable of such analysis, like CPPCHECK, then it was executed by giving it paths instead of running it separately for each file. This decision also affects the execution time due to the way a new process of the tool is spawned for each file if analyzed separately. All tools were configured to finally output their results into a common report directory.

Our build process was based on the JENKINS continuous integration software where we already had several static analyzers in use for other non-C++ projects. The DYNAROAD build was configured in an existing job called `DynaRoad-trunk`. This job is responsible for building, testing, and creating installation archives of the DYNAROAD development version. A new build job `DynaRoad-analyze` was created that runs the newly created `Check` build

target thus executing all static analyzers. This job was scheduled to run after the main job as a dependency. As the dependency is only notified in case of a successful build, the analysis is only done when needed.

The reports of each tool are gathered by a specialized JENKINS ANALYZER COLLECTOR[2] plug-in capable of generating graphical trends and reports. What made this plug-in even more useful for our integration was its feature of reporting the warning differences between runs. This feature was further configured to fail the build in case too many new warnings occurred. To help developers notice these warnings, e-mail notifications are sent that also aid them in fixing the warnings.

[2]https://wiki.jenkins-ci.org/display/JENKINS/Analysis+Collector+Plugin

Table 6.8: Most frequent warnings issued for DynaRoad version 5.3.0.

Name	Warning	Count
Clang	Field X will be initialized after field Y.	57
	Calling convention X ignore for this target.	36
	Missing X prior to dependent type name.	16
	Suggest braces around initialization of sub-object.	7
	Private field X is not used.	6
Cppcheck	Use const reference for X to avoid unnecessary data copying.	19
	Function parameter X should be passed by reference.	18
	Unused variable.	9
	Variable X is assigned a value that is never used.	8
	`BOOST_FOREACH` caches the end iterator. It is undefined behavior if you modify the container inside.	5
Cpplint	Using deprecated casting style.	115
	Single-argument constructors should be marked explicit.	26
	Line ends in whitespace.	16
	Redundant blank line at the end of a code block should be deleted.	15
	Failed to find complete declaration of class.	15
Flint	The associated header file of .cpp files should be included before any other includes.	214
	`volatile` does not make your code thread-safe.	11
	Heap-allocated exception: `throw new X();`.	11
	`boost::shared_ptr` should be replaced by `boost::make_shared`.	6
	Implicit conversion to X may inadvertently be used.	6
PVS	A code fragment from X cannot be analyzed.	527
	Memsize type is used in the struct/class.	447
	Decreased performance. Consider creating a pointer/reference to avoid using the same expression repeatedly.	253
	Be advised that the size of the type `long` varies between LLP64/LP64 data models.	218
	Dangerous magic number X used.	173

Chapter 7

Conclusion

In this thesis we introduced the concept of static code analysis and focused on its contribution to software quality. We began by explaining what the benefits of static analysis are and how such analysis is limited by theoretical and practical problems. Various practical use cases were expanded upon and illustrated with examples.

We also examined the various methods and algorithms that static analyzers are implemented with. Theoretical limits ensure that static analyzers can never perfectly analyze all programs. They still provide useful benefits in terms of defect finding due to their easy automation and use. The choice of what methods to use depends on how deep of an understanding of the code is desired. Static analysis tools can be used to find problems in source code ranging from style issues to race conditions. Typical style checkers can work very well with only a minimal understanding, thus only lexical pattern matching is needed for the analysis.

We continued by divulging into the various problems facing static analysis tools when analyzing C++ source code. These problems include aspects such as: understanding complex and non-standard code, precision and scalability, and how to manage false positives. To further our efforts in comprehending the specifics of software defects, we introduced the notion of defect classifications, which help us distinguish instances of defects. A few existing classifications are detailed, in particular the security focused Common Weakness Enumeration (CWE). Since a major part of our thesis was to evaluate existing static analysis tools, we researched several such tools and produced an feature summary of them.

The DynaRoad software, being the target of our improvements, was briefly introduced by giving an overview of its features and capabilities. A technical overview of it then followed giving insight into the architectural choices made during development. The build process and the existing static

analysis tools used as part of it were furthermore detailed. Finally, as one of the objectives of this thesis, we examined the possible improvements to the aforementioned process.

We concluded the thesis by performing a comprehensive evaluation of four open-source and two commercial tools. To find out what types of defects the tools are capable of finding, we began by evaluating them using a set of synthetic tests. Our set of tests included 30 different tests ranging from buffer overflows to the use of floating point loop counters. None of the tools were able to detect all defects, which clearly exemplifies the importance of running many analyzers together. In practice, due to the varying detection accuracies of these tools in different control-flow configurations, the tools are able to complement each other quite well.

The best performer, Clang, positively detected defects in 15 out of all 30 tests. The next best accuracies were observed from Visual Studio 2013 and Cppcheck. Both were able to detect 14 defects, but not all were the same ones. We found it interesting that the best accuracy for a non-compiler tool was by Cppcheck as it really shows that static analysis tools do not have to be compiler-level language parsers. The only commercial non-compiler we evaluated, PVS-Studio, came in close second detecting 11 different defects. Cpplint and Flint, being mainly style checkers, were unsurprisingly the worst performers with five detected defects each.

The second evaluation we performed was a differential analysis where the defects detected by each tool were compared against the known crash defects in our software. We retrieved the source code for six different point releases of the stable version of our software. Each tool was then instructed to analyze these versions individually. Then, the changes in the reports between each version were compared against the information in our bug tracking software which gave us a picture of their real-world accuracy. To our dismay, we found that none of the tools were able to detect any of the known crash defects. Granted, most of the crash bugs were not easy cases for static analyzers to find. The tools did not possess the required capabilities to analyze deep flow-related problems and domain-specific semantic behavior. Fortunately, it was encouraging to see that many of the tools found other issues that had been in the code for years.

As the final contribution of this thesis, we investigated the integration of some of the evaluated tools into our own software engineering processes. We used Jenkins as a continuous integrator that automatically built and analyzed our software. This meant that we were receiving continuous and automatic reports of the quality of our software. By integrating static analysis tool with our existing automated dynamic tests, we found that both types of testing complemented each other well. The main problem with introducing

static analyzers to mature software was that the initial warning counts were very large. Thus, the use of differential reports became relevant and aided our adoption of these tools by notifying developers in case of new warnings. This meant that we could ignore the original several thousand warnings at least for a while.

7.1 Future Work

Our evaluation of Clang along with its static analyzer showed us that it has excellent detection capabilities. What excites us even more is the great potential of the framework and the future checkers[1] being created with it. Unfortunately, due to incompatibilities with our current code base, we were unable to integrate it in our build process completely. It would be very interesting to see how they would perform if the code was modified to make Clang cleanly compile it.

As our objective was to put as many static analyzers into use as possible, the idea of using commercial tools intrigues us. For us, the high price of most commercial tools has been prohibitive, but it could be alleviated by evaluating them in our environment beforehand. Having tools that can easily be automated is the key, though, which is why the large graphical user interfaces of these commercial tools have not interested us as much.

Another interesting prospect is the development and use of custom checkers. Many project- and domain-specific issues can realistically only be found through custom checks that have increased semantic knowledge of the code base. One typical problem that we have noticed in DynaRoad is the omission of certain functions calls required by our serialization library. Specifically, a `NotifyChange` function must be called in every domain model class before the model mutates itself [10]. Detecting the lack of this call and many similar problems would be a perfect job for a static analyzer.

The way the static analyzers are currently automated in our build process can also be improved. Our other projects, which are much younger than DynaRoad, have been introduced with systematic code review. The code review process integrates perfectly with automated static analysis and it can use the analysis results to automatically down-vote defect introducing changes. A similar process, which streamlines the work-flow and lessens the burden on developers, has also been in use at Google [6] and is a possible future for DynaRoad as well.

[1] http://clang-analyzer.llvm.org/potential_checkers.html

Bibliography

[1] A. Aggarwal and P. Jalote. ”Integrating Static and Dynamic Analysis for Detecting Vulnerabilities”. In: *Proceedings of the 30th Annual International Computer Software and Applications Conference*. 2006. DOI: 10.1109/COMPSAC.2006.55.

[2] A. V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley, 2006.

[3] A. Almossawi, K. Lim, and T. Sinha. *Analysis Tool Evaluation: Coverity Prevent*. Technical report. Carnegie Mellon University, 2006.

[4] K. Ashcraft and D. Engler. ”Using programmer-written compiler extensions to catch security holes”. In: *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. 2002. DOI: 10.1109/SECPRI.2002.1004368.

[5] N. Ayewah et al. ”Evaluating static analysis defect warnings on production software”. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 2007. DOI: 10.1145/1251535.1251536.

[6] N. Ayewah et al. ”Using Static Analysis to Find Bugs”. In: *IEEE Software* 25 (2008), pages 22–29. DOI: 10.1109/MS.2008.130.

[7] D. Baca, B. Carlsson, and L. Lundberg. ”Evaluating the cost reduction of static code analysis for software security”. In: *Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security*. 2008. DOI: 10.1145/1375696.1375707.

[8] T. Ball and S. K. Rajamani. ”The SLAM project: debugging system software via static analysis”. In: *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 2002. DOI: 10.1145/565816.503274.

[9] A. Bessey et al. ”A few billion lines of code later: using static analysis to find bugs in the real world”. In: *Communications of the ACM* 53 (2010), pages 66–75. DOI: 10.1145/1646353.1646374.

[10] K. Björklund. "A Serialization Library with Undo Support". Master's thesis. Helsinki University of Technology, 2005. URL: `http://www.iki.fi/kbjorklu/mthesis/`.

[11] T. Boland and P. E. Black. "Juliet 1.1 C/C++ and Java Test Suite". In: *IEEE Computer* 45 (2012), pages 88–90. DOI: `10.1109/MC.2012.345`.

[12] W. R. Bush, J. D. Pincus, and D. J. Sielaff. "A static analyzer for finding dynamic programming errors". In: *Software: Practice and Experience* 30 (2000), pages 775–802. DOI: `10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H`.

[13] B.-M. Chang, J.-W. Jo, and S. H. Her. "Visualization of exception propagation for Java using static analysis". In: *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*. 2002. DOI: `10.1109/SCAM.2002.1134117`.

[14] G. Chatzieleftheriou and P. Katsaros. "Test-Driving Static Analysis Tools in Search of C Code Vulnerabilities". In: *Proceedings of the 35th Annual Computer Software and Applications Conference Workshops*. 2011. DOI: `10.1109/COMPSACW.2011.26`.

[15] B. Chess and G. McGraw. "Static analysis for security". In: *IEEE Security & Privacy* 2 (2004), pages 76–79. DOI: `10.1109/MSP.2004.111`.

[16] B. Chess. *Secure programming with static analysis*. Addison-Wesley, 2007.

[17] C. Cifuentes and B. Scholz. "Parfait: designing a scalable bug checker". In: *Proceedings of the Static Analysis Workshop*. 2008. DOI: `10.1145/1394504.1394505`.

[18] B. Cole et al. "Improving your software using static analysis to find bugs". In: *Companion to the 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications*. 2006. DOI: `10.1145/1176617.1176667`.

[19] P. Cousot and R. Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 1977. DOI: `10.1145/512950.512973`.

[20] P. Cuoq, F. Kirchner, and B. Yakobowski. "Benchmarking Static Analyzers". In: *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems*. 2012. DOI: `10.1.1.416.5890`.

[21] Z. Ding, H. Wang, and L. Ling. "Practical strategies to improve test efficiency". In: *Tsinghua Science and Technology* 12 (2007), pages 250–254. DOI: 10.1016/S1007-0214(07)70119-0.

[22] E. van Emden and L. Moonen. "Java quality assurance by detecting code smells". In: *Proceedings of the 9th Working Conference on Reverse Engineering*. 2002. DOI: 10.1109/WCRE.2002.1173068.

[23] D. Engler et al. "Checking system rules using system-specific, programmer-written compiler extensions". In: *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation*. 2000.

[24] A. Ermakov and N. Kushik. "Detecting C Program Vulnerabilities". In: *Proceedings of the 5th Spring/Summer Young Researchers' Colloquium on Software Engineering*. 2011.

[25] D. Evans and D. Larochelle. "Improving security using extensible lightweight static analysis". In: *IEEE Software* 19 (2002), pages 42–51. DOI: 10.1109/52.976940.

[26] S. Falke, F. Merz, and C. Sinz. "LLBMC: Improved Bounded Model Checking of C Programs Using LLVM". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-36742-7_48.

[27] A. Fehnker and R. Huuck. "Model checking driven static analysis for the real world: designing and tuning large scale bug detection". English. In: *Innovations in Systems and Software Engineering* 9 (2013), pages 45–56. DOI: 10.1007/s11334-012-0192-5.

[28] A. Fehnker et al. "Model Checking Software at Compile Time". In: *Proceedings of the 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. 2007. DOI: 10.1109/TASE.2007.34.

[29] C. Frye. *Klocwork static analysis tool proves its worth, finds bugs in open source projects*. 2006. URL: http://searchsoftwarequality.techtarget.com/news/1196981/Klocwork-static-analysis-tool-proves-its-worth-finds-bugs-in-open-source-projects.

[30] J. Gimpel. "Software That Checks Software: The Impact of PC-lint". In: *IEEE Software* 31 (2014), pages 15–19. DOI: 10.1109/MS.2014.13.

[31] S. Hallem et al. "A system and language for building system-specific, static analyses". In: *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2002. DOI: 10.1145/543552.512539.

[32] A. Hanjalic. ”ClonEvol: Visualizing software evolution with code clones”. In: *Proceedings of the 1st IEEE Working Conference on Software Visualization*. 2013. DOI: 10.1109/VISSOFT.2013.6650525.

[33] M. Hind. ”Pointer analysis: haven’t we solved this problem yet?” In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 2001. DOI: 10.1145/379605.379665.

[34] D. Hovemeyer and W. Pugh. ”Finding bugs is easy”. In: *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 2004. DOI: 10.1145/1052883.1052895.

[35] D. Hovemeyer and W. Pugh. ”Finding more null pointer bugs, but not too many”. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 2007. DOI: 10.1145/1251535.1251537.

[36] D. Hovemeyer, J. Spacco, and W. Pugh. ”Evaluating and tuning a static analysis to find null pointer bugs”. In: *Proceedings of the 2005 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*. 2005. DOI: 10.1145/1108768.1108798.

[37] B. Johnson et al. ”Why don’t software developers use static analysis tools to find bugs?” In: *Proceedings of the 35th International Conference on Software Engineering*. 2013. DOI: 10.1109/ICSE.2013.6606613.

[38] S. C. Johnson. *Lint, a C program checker*. Technical report. Bell Laboratories, 1977.

[39] J. Kemppainen et al. ”Lean Construction Principles in Infrastructure Construction”. In: *Proceedings of the 12th Annual Conference of the International Group for Lean Construction*. 2004.

[40] W. M. Khoo et al. *Hunting for vulnerabilities in large software: the OpenOffice suite*. Technical report. University of Cambridge, 2010.

[41] G. A. Kildall. ”A Unified Approach to Global Program Optimization”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. 1973. DOI: 10.1145/512927.512945.

[42] J. C. King. ”Symbolic Execution and Program Testing”. In: *Communications of the ACM* 19 (1976), pages 385–394. DOI: 10.1145/360248.360252.

[43] W. Landi. "Undecidability of static analysis". In: *ACM Letters on Programming Languages and Systems* 1 (1992), pages 323–337. DOI: 10.1145/161494.161501.

[44] D. Larochelle and D. Evans. "Statically Detecting Likely Buffer Overflow Vulnerabilities". In: *Proceedings of the 10th USENIX Security Symposium*. 2001.

[45] C. Lattner. "LLVM and Clang: Next generation compiler technology". In: *Proceedings of the BSD Conference*. 2008.

[46] Z. Li et al. "CP-Miner: finding copy-paste and related bugs in large-scale software code". In: *IEEE Transactions on Software Engineering* 32 (2006), pages 176–192. DOI: 10.1109/TSE.2006.28.

[47] F. Logozzo and M. Fähndrich. "Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses". In: *Proceedings of the 2008 ACM Symposium on Applied Computing*. 2008. DOI: 10.1145/1363686.1363736.

[48] F. Merz, S. Falke, and C. Sinz. "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR". In: *Verified Software: Theories, Tools, Experiments*. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-27705-4_12.

[49] A. Miné. "The octagon abstract domain". English. In: *Higher-Order and Symbolic Computation* 19 (2006), pages 31–100. DOI: 10.1007/s10990-006-8609-1.

[50] N. Moha et al. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Transactions on Software Engineering* 36 (2010), pages 20–36. DOI: 10.1109/TSE.2009.50.

[51] N. Nagappan and T. Ball. "Static Analysis Tools As Early Indicators of Pre-release Defect Density". In: *Proceedings of the 27th International Conference on Software Engineering*. 2005. DOI: 10.1145/1062455.1062558.

[52] V. Okun et al. "Effect of static analysis tools on software security: preliminary investigation". In: *Proceedings of the 3rd ACM Workshop on Quality of Protection*. 2007. DOI: 10.1145/1314257.1314260.

[53] J. R. C. Patterson. "Accurate Static Branch Prediction by Value Range Propagation". In: *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*. 1995. DOI: 10.1145/223428.207117.

[54] M. Rosendahl. *Introduction to abstract interpretation*. Technical report. University of Copenhagen, 1995.

[55] S. Schmeelk. ”Towards a Unified Fault-detection Benchmark”. In: *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. 2010. DOI: 10.1145/1806672.1806684.

[56] R. C. Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, 2005.

[57] U. Shankar et al. ”Detecting Format String Vulnerabilities with Type Qualifiers”. In: *Proceedings of the 10th USENIX Security Symposium*. 2001.

[58] C. Sinz, F. Merz, and S. Falke. ”LLBMC: A Bounded Model Checker for LLVM’s Intermediate Representation”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-28756-5_44.

[59] J. Slaby. ”Automatic Bug-Finding Techniques for Large Software Projects”. PhD thesis. Masaryk University, 2013.

[60] K. Tsipenyuk, B. Chess, and G. McGraw. ”Seven pernicious kingdoms: a taxonomy of software security errors”. In: *IEEE Security & Privacy* 3 (2005), pages 81–84. DOI: 10.1109/MSP.2005.159.

[61] M. Vestola. ”Evaluating and enhancing FindBugs to detect bugs from mature software: Case study in Valuatum”. Master’s thesis. Aalto University, 2012.

[62] J. Viega et al. ”ITS4: a static vulnerability scanner for C and C++ code”. In: *Proceedings of the 16th Annual Computer Security Applications Conference*. 2000. DOI: 10.1109/ACSAC.2000.898880.

[63] S. Wagner et al. ”An Evaluation of Two Bug Pattern Tools for Java”. In: *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation*. 2008. DOI: 10.1109/ICST.2008.63.

[64] M. Webster. *Leveraging static analysis for a multidimensional view of software quality and security: Klocwork’s solution*. Technical report. Klocwork Inc., 2005.

[65] F. Wedyan, D. Alrmuny, and J. M. Bieman. ”The Effectiveness of Automated Static Analysis Tools for Fault Detection and Refactoring Prediction”. In: *Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation*. 2009. DOI: 10.1109/ICST.2009.21.

[66] Y. Xie et al. "Soundness and its role in bug detection systems". In: *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools*. 2005.

[67] Z. Xu, T. Kremenek, and J. Zhang. "A memory model for static analysis of C programs". In: *Proceedings of the 4th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation*. 2010.

[68] J. Yang et al. "Using model checking to find serious file system errors". In: *ACM Transactions on Computer Systems* 24 (2006), pages 393–423. DOI: 10.1145/1189256.1189259.

Appendix A

Defect Descriptions

CWE-121: Stack-based buffer overflow

This defect specifically defines the case where a stack-allocated buffer is overflown. What makes this case especially problematic is that an attacker could execute arbitrary code by modifying function return addresses and other similarly critical information.

CWE-122: Heap-based buffer overflow

When copying a buffer to another, the destination buffer should be large enough to be able to contain the data of the source buffer. If this is not the case, the destination buffer will overflow. In case the destination buffer has been allocated on the heap, it is a heap-based buffer overflow.

CWE-134: Uncontrolled format string

Use of `printf` and other similar standard C functions support passing a format string. A format string defines the way input or output is to be handled and formatted, including types of values. In case the format string can be directly manipulated by an attacker, the program can be exploited to read and write unintended memory locations. Thus, it is important that the format string is not provided by the user.

CWE-170: Improper null termination

In C and C++ strings are encoded with a terminating null character. This allows strings lengths to be calculated by counting the number of characters preceding the null character. In case the null character is missing, these length calculations and other iterative character-based operations will fail to correctly detect the end of the string. Such strings can be the result of `strncpy` calls, which can end in a string without null termination.

CWE-190: Integer overflow

Integers can contain a finite range of values. Integers operations can thus overflow when being added or multiplied together. Problems arise when a program manages resources or execution using these possibly overflown integers.

CWE-242: Use of inherently dangerous function

This vulnerability concerns the use of functions that are always dangers no matter their use. One such function is the standard C function `gets`. The purpose of this function is to return a string containing the characters read from the user. Unfortunately, the caller can never be sure how many characters were read, thus being unable to accommodate and correctly allocate a buffer large enough. Any buffer is therefore easily overflown by a malicious user.

CWE-327: Use of a broken or risky cryptographic algorithm

The standard C/C++ function `rand` is often implemented with a cryptographically unsafe random number generator. Even if cryptographic safety is not a requirement, better randomness is provided by other standard or general purpose libraries. For C++ programs, the standard library has implementations for many random number generators, such as Mersenne Twister.

CWE-369: Divide by zero

Dividing integers with zero is undefined behavior and should thus be prevented. Algorithms dividing integers by unknown values should therefore check for zero divisors. This affects handling of user input as well in case the user may directly invoke a division by zero. Floating point numbers have a defined division-by-zero behavior.

CWE-401: Memory leak

Allocated memory should be deallocated after it is no longer needed, but if it is not, the memory is leaked. If the program allocates a large amount of memory repeatedly, leaking memory could lead to out-of-memory errors. This can be used by attackers for denial of service attacks.

CWE-413: Improper resource locking

Multi-threaded environments give rise to completely new problems for programmers. One such typical issue is the lack of resource locking when exclusive access to it is intended. For simple data types, locks can be replaced with faster atomic operations.

CWE-415: Double free

Memory freed with the `free` or `delete`-family of functions should not be freed again. Freeing memory twice, i.e., double freeing, is undefined behavior. In C++, use of smart pointers greatly decreases the prevalence of these issues.

CWE-416: Use after free

In C and C++ a memory pointer is not set to null after it has been freed. The existence of freed non-null pointer means that they can be dereferenced after being freed. This leads to a use after free defect due to the pointer still pointing to its former memory address.

CWE-457: Use of uninitialized variable

Variables in C and C++ can be left uninitialized. This can be seen as a marginal performance optimization, but may lead to unintended behavior if not correctly maintained. It is thus often safer to explicitly initialize variables as that makes the code more maintainable due to not relying on the variable being initialized on every execution path.

CWE-476: Null pointer dereference

Pointers in C and C++ can point to any conceivable memory address only restricted by the address space. Typically, an invalid pointer will point to the null address. Dereferencing a null pointer leads to undefined behavior and is thus discouraged.

CWE-561: Dead code

Dead code is code that is never executed. This can mean that the code contains a bug that leads to parts of the code never being executed. Another typical reason is that the code is old and due to refactoring is purposefully never executed. In any case, dead code should be eliminated as that increases the maintainability and quality of the code base.

CWE-590: Free of memory not on the heap

It is possible to use deallocation operations, like `free` and `delete`, on memory addresses pointing to non-heap memory. Such calls lead to undefined behavior and in many cases will crash the program. One example of this is the deallocation of a pointer pointing to a variable allocated on the stack.

CWE-663: Use of non-reentrant function in concurrent context

Some standard C library functions are non-reentrant, i.e., they are not safe to use in multi-threaded environments. Typical examples of

functions like these are `strtok` and `rand`. The reason for non-reentrancy can often be attributed to global structures being manipulated by these functions. To overcome these issues, compiler vendors can provide non-portable reentrant versions of these functions.

CWE-676: Use of potentially dangerous function
This weakness describes the use of known problematic API functions, such as `strcpy`. These functions can be used safely in some cases, but incorrect use could lead to vulnerabilities. In the case of `strcpy`, the problem manifests itself when the string to be copied is of undetermined length, thus resulting in a possible buffer overflow.

CWE-762: Mismatched memory management routines
C++ introduces several new keywords related to memory management. These are the `new`/`delete` and `new[]`/`delete[]` pairings of keywords. The former allocates memory for a single C++ object calling its constructor at allocation and destructor at deallocation. The latter form allocates memory for an array of C++ objects once again calling their respective constructors and destructors. If these forms are mismatched with each other or the C `malloc`/`free` functions, problems arise.

ARR32-CPP: Use of invalidated iterators
Iterators are basically pointers to elements in C++ containers. If the container is modified it will in some cases invalidate all iterators. This mostly happens when the container wants to reallocate its memory. Use of invalidated iterators leads to undefined behavior.

DCL17-CPP: Pass large objects as const references
In addition to passing values by value or as pointers, C++ has support for reference semantics. References are essentially non-null pointers and provide a convenient way of passing around large objects. Passing by const reference is typically faster than by passing by value as that avoids the copying of the object. This is mostly a performance improvement.

ERR09-CPP: Throw temporaries and catch by reference
Exceptions can be allocated on the stack or on the heap. Unfortunately, throwing exceptions allocated on the heap leaves responsibility of the deallocation to the catcher. Stack-allocated exceptions on the other hand can be thrown by value, thus creating a copy, or by taking the address of it. Throwing while using the address-of operator essentially throws a pointer. All this confusion can be solved by only throwing anonymous temporaries and catching them as (const) references.

ERR37-CPP: Use of deprecated function exception specification

Exception-specifications have been deprecated in the latest C++ standard due to implementation and performance problems. The new standard specifies the `noexcept` keyword that can be used to tell the compiler and users of that function that no exceptions will be thrown from it. It should be noted that the secure coding standard originally mandated the correct use of the now deprecated `throw()` keyword.

EXP05-CPP: Do not use C-style casts

Variables and values in C can be cast to basically any other type. Due to being backwards compatible with C, C++ maintains the same syntax for type casts. It also introduces new keywords for the various new casts in the language: `static_cast`, `dynamic_cast`, `const_cast`, and `reinterpret_cast`. The use of these keywords is encouraged as they are more explicit about the intention of the cast. They also have different semantics to C-style casts.

EXP31-CPP: Avoid side effects in assertions

The C standard library macro `assert` can be used to make runtime assertions about conditions in the program code. To only enable this macro in debug builds, the macro is defined to be a no-op if `NDEBUG` is defined. If assertions are made that modify the program state in any way, the assertion is said to have side effects. This is a problem if the program is ever built without defining `NDEBUG` as those side effects will not exist. Thus, assertions should never have side effects.

EXP35-C: Do not modify objects with temporary lifetime

The typical instance of this is when a function returns the address to a local variable declared inside the function. Dereferencing that address and modifying the variable may lead to undefined behavior.

FLP30-CPP: Use of floating point variables as loop counters

Because floating point numbers are unable to represent all fractions precisely, their use as loop counters is frowned upon. Loops with such counters can result in infinite never-ending loops. As floating point numbers are hardware-dependent, exact behavior of such loops is difficult to predict.

INT11-CPP: Converting between pointers and integers

Care should be taken when casting integers to pointers and vice versa. Mistakes are easy to make if wrong integer types are used or the integer values are manipulated in some way. This leads to less portable code

and should be avoided if possible. The C and C++ standard libraries provide portable type definitions for pointer address values in case any casting is required.

OOP32-CPP: Single-argument constructors should be "explicit"
Class constructors in C++ are by default implicit, i.e., they will be used for implicit type conversions. The reason for this is backwards compatibility with the C language. Often, these conversions are unintended and may lead to bugs that are difficult to find. Making all single-argument constructors explicit, except when implicit conversions are actually needed, makes the code more robust and less prone to bugs.

OOP34-CPP: Ensure proper destructor for polymorphic objects
Polymorphic objects can be deleted via pointers to their base class. For such deletions to be well-defined, the base class is required to have a virtual destructor.

Appendix B

DynaRoad Crash Defects

B.1 Version 5.3.0

Listing B.1: Integer division-by-zero error.

```
1 int days = get_work_days(start_date, finish_date, calendar);
2 double total_amount = 0;
3
4 if (total_amount > 0)
5 {
6     // days is zero
7     average_amount = total_amount / days;
8 }
```

Listing B.2: Unhandled exception due to erroneous format string in translation.

```
1 // format string missing variable placeholder
2 const string str = (format("String") % error_str).str();
```

Listing B.3: Null pointer dereference due to Windows system setting disabling recent file history.

```
1 // file_list is null
2 if (index >= file_list->size())
3     ...
```

B.2 Version 5.3.1

Listing B.4: Use of floating point NaN as function argument due to division-by-zero caused by user input.

```
1 // portion_begin equals portion_begin
2 return get_value(p->get_location(), (portion - portion_begin) / (portion_end -
      portion_begin));
```

Listing B.5: Unhandled exception due to logic error caused by user input.

```
1 const map_t::const_iterator i = m.find(p);
2
3 if (i == m.end())
4 {
5     // not handled
6     throw std::logic_error("not found");
7 }
```

Listing B.6: Use of freed memory due to logic error.

```
1 for (auto p : objects)
2 {
3     // p is already freed
4     if (!p->get())
5         ...
6 }
```

B.3 Version 5.3.2

Listing B.7: Null pointer dereference due to invalid `dynamic_cast` in iterator adaptor.

```
1 for (iterator i = task.begin(), i_end = task.end(); i != i_end; ++i)
2 {
3     const task_t& t = **i;
4     // t is a dereferenced null pointer
5     const location_t& l = t.get_location();
6     ...
7 }
```

Listing B.8: Thread using freed memory due to missing thread termination.

```
1 void static_thread_function(void* param)
2 {
3     dialog_t& dlg = *reinterpret_cast<dialog_t*>(param);
```

```
4       ...
5       dlg.thread->solve();
6       ...
7       dlg.thread->stop(); // dlg has been destructed during the call to solve
8       return 0;
9   }
```

Listing B.9: Use of invalid array index.

```
1   // get_array returns wrapper to empty array
2   if (e.get_array().get_at(0).get_number() > 0)
3       ...
```

Listing B.10: Out-of-range `vector` insertion position due to invalid negative parameter.

```
1   // row is -1
2   void model::insert_row(const int row, const int count)
3   {
4       if (count == 0)
5           return;
6       ...
7       const int index = calculate_index(*this, row, 0);
8       items_.insert(items_.begin() + index, count * columns_, 0);
9       ...
10  }
```

B.4 Version 5.3.3

Listing B.11: Use of a reference bound to a temporary `auto_ptr`.

```
1   // convert returns a temporary auto_ptr
2   const bitmap_t& converted = (bitmap.get_format() == FORMAT_XYZ) ?
        bitmap : *bitmap.convert(FORMAT_XYZ);
3   return std::auto_ptr<Bitmap>(new Bitmap(BitmapInfo(converted.get_width(),
        converted.get_height(), converted.is_alpha(), converted.get_bpp(), converted.
        get_data())));
```

B.5 Version 5.3.4

Listing B.12: Null dereference of wrapped pointer retrieved from an external library.

```
1 // get_object returns an object wrapping a null pointer
2 object_t o = element.get_object();
3
4 if (o.get_type() == type_xyz)
5     ...
```

B.6 Version 5.3.5

Listing B.13: `vector` insertion throws unhandled exception due to an out-of-memory error.

```
1 // not enough memory for vector insertion
2 items_.insert(items_.begin() + index, count * columns_, 0);
```

Listing B.14: Null pointer dereference due to missing handling of deleted pointers.

```
1 // get_source returns a dereferenced freed pointer
2 set.insert(&u.get_source().get_site());
```

Listing B.15: Unhandled `std::bad_alloc` exception due to excessive copying.

```
1 bitmap_t::bitmap_t(const int width, const int height, const int bytes_per_line,
      const format_t fmt)
2 {
3     // not enough memory for vector resize operation
4     data_.resize(bytes_per_line * height);
5 }
```

Listing B.16: Passing null pointers to `std::string::assign`.

```
1 if (loaded && p)
2 {
3     // UserName is null
4     user_name.assign(p->UserName);
5     // CredentialBlob is null
6     const wchar_t* const q = reinterpret_cast<const wchar_t*>(p->
          CredentialBlob);
7     password.assign(q, q + p->CredentialBlobSize / sizeof(wchar_t));
8 }
```

Listing B.17: Unhandled exception due to logic error.

```
1 // unhandled exception
2 throw std::logic_error("not started"));
```

Listing B.18: Unhandled exception due to logic error.

```
1 // unhandled exception
2 throw calendar_exception();
```