An overview on the Static Code Analysis approach in Software Development

Ivo Gomes[1], Pedro Morgado[1], Tiago Gomes[1], Rodrigo Moreira[2],

[1] Software Testing and Quality, Master in Informatics and Computing Engineering,
[2] Software Testing and Quality, Doctoral Program in Informatics Engineering,
Faculdade de Engenharia da Universidade do Porto,
Rua Dr. Roberto Frias 4200-465, Porto, Portugal
{ei05021, ei05051, ei05080, pro08007}@fe.up.pt

Abstract. Static analysis examines program code and reasons over all possible behaviors that might arise at run time. Tools based on static analysis can be used to find defects in programs. Recent technology advances has brought forward tools that do deeper analyses that discover more defects and produce a limited amount of false warnings. The aim of this work is to succinctly describe static code analysis, its features and potential, giving an overview of the concepts and technologies behind this type of approach to software development as well as the tools that enable the usage of code reviewing tools to aid programmers in the development of applications, thus being able to improve the code and correct errors before an actual execution of the code.

Keywords: static analysis, code review, code inspection, source code, bugs, dynamic analysis, software testing, manual review.

1 Introduction

The use of analytical methods to review source code in order to correct implementation bugs is, and has been, one of the backbone pillars behind software development.

In the beginning of software development there was no conscience on how necessary and effective a review might be, but in the 1970's, formal review and inspections were recognized as important to productivity and product quality, and thus were adopted by development projects [1]. This new approach to software development acknowledges defect removal in the early stages of the development process proved to produce more reliable and efficient programs. Fagan's definition of error detection efficiency is as follows [2]:

$$\text{Error Detection Efficiency} = \frac{\text{Errors found by an inpection}}{\text{Total errors in the product before inspection}} \times 100\,. \qquad (1)$$

So, as far as source code is concerned, it is in the best interest of the programmer to take advantage of static analysis. Although this does not imply that other forms of software analysis should be discouraged, on the contrary, the best way to certify that an implementation has the least amount of errors or defects is by combining both the static and the dynamic measures of analysis.

The static analysis approach is meant to review the source code, checking the compliance of specific rules, usage of arguments and so forth; the dynamic approach is essentially executing the code, running the program and dynamically checking for inconsistencies of the given results. This means that testing and reviewing code are separate and distinguishable things, but it is unadvised that one should occur without the other, and it is also arguable as to what should be done first, testing or reviewing software [3].

This work focuses on the description of the static methods of analysis, with a special attention to the available tools in the market that provide this kind of service.

This paper is organized in the following sections: Section 1, this current section, introduced the static analysis approach; Section 2 will describe a relative brief overview of static analysis, followed by the description of the most common methods of code reviewing done by humans: self review, walkthrough, peer review, inspection and audit. In order to ascertain the truly fundamental qualities of static code analysis and more importantly, to distinguish them from the dynamical testing approaches, Section 3 will describe the advantages and disadvantages regarding static analysis. A comprehensive comparison between code review and testing shall explain why the usage of just one of them is discouraged; Section 4 will summarize a listing of the most popular software tools that are capable of performing this type of code analysis which shall be followed by a comparison between some aspects of these tools; a further evaluation of these tools is described in Section 5; in Section 6 will feature some possible enhancements to be performed on such tools; and finally Section 7 will express a discussion over static code analysis tools in software development.

2 Overview of the Static Analysis approach

Static code analysis is the analysis of computer software which is performed without the actual execution of the programs built from that software, as opposite of dynamic analysis (testing software by executing programs). For the majority of cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated software tool, with human analysis being called program understanding, program comprehension or code inspection.

It can be argued that software metrics and reverse engineering are forms of static analysis, but such discussion is not the aim of this work.

Programmers make little mistakes all the time, like a missing semicolon here, an extra parenthesis there, and so on. Most of the time these gaffes are inconsequential,

the compiler notes the error, the programmer fixes the code, and the development process continues. However, this quick cycle of feedback and response normally does not apply to most security vulnerabilities, which can lie dormant for an indefinite amount of time before discovery. As explained earlier, the longer a defect on the software lies dormant, the more expensive it can be to fix [4].

The promise of static analysis is to identify many common coding problems automatically before a program is released. Static analysis aims to examine the text of a program statically, without attempting to execute it. Theoretically, static analysis tools can examine either a program's source code or a compiled form of the program to equal benefit, although the problem of decoding the latter can be difficult [4].

2.1 Manual Review

Manual reviewing or auditing is a form of static analysis, very time-consuming, and to perform it effectively, human code auditors must first know what type of errors there are supposed to find before they can rigorously examine the code.

The reviewing of an application's code can be done in any phase of software development, but the best results are when this is done at an early stage, because the costs and risk of detecting and correcting security vulnerabilities and quality defects late in the software development process can be high. When those bugs escape into the market and are discovered by customers, the fallout can affect the bottom line and damage reputations [5].

Reviewing includes not only the code, but all documentation, requirements and designs the developer produces, everything is susceptible of being review, because there can be errors hidden in every step of software development.

Basically, static code analysis performed by humans can be divided in two major categories: self reviews and 3rd party reviews, which are tightly related to the Personal Software Process and the Team Software Process [6].

In the picture below, the initial phase shows the actual implementing of the code, which obviously isn't any type of static analysis. Following is the *self review* of the written code, where the programmer tries to evaluate and correct by himself the code he implemented. The *walkthrough* focuses on the presentation to an audience of the code in question by its programmer. The *peer review* is when the programmer presents his code to a colleague to review. Finally the *inspection* and *audit*, which is usually done by a third party of evaluators, the *audit* being the highest formal review [5].

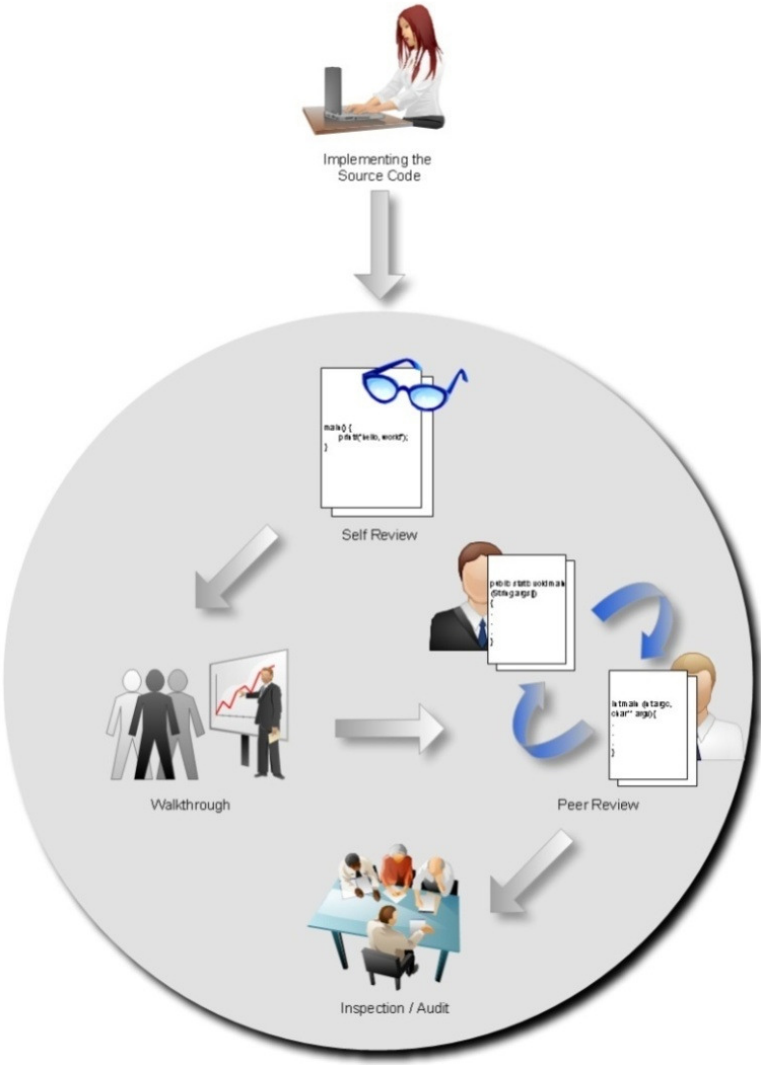

Fig. 1. Flow of types of reviews that increase formality.

The best way to detect and correct bugs in an early stage of development is when the programmer himself performs the review and tries to find and correct problems in his code, this is commonly known as self review.

In every programmer there should be a sense of personal responsibility in his implementations, and as such, it is always a good idea to try and keep track of the most common mistakes he does. This way in time it will become easier to prevent repeating them once again.

There are some guidelines as to how to perform a proper self review: producing reviewable items (code, design, specifications, etc.); trying not to review code on screen, to circumvent the tendency to correct bugs as they are found; not reviewing

the code right after it is written; to follow a structured review process; create personal checklists of the most common mistakes; taking enough time to review the code, so as to be certain that everything is as it should be (usually half the time it was required to write the code is more than enough to properly review it) [7].

The team review process can be a bit more complex, and there several different steps in reviewing software as a group of people. An interesting method is the walkthrough, in which the developer explains his code and ideas to an audience, being subject to their criticism. In addition, there are formal requisites to perform static reviews of code.

This kind of group review can be achieved with a before-after technique, meaning there is a necessity of a review plan prior to the review (assembled by the leading reviewer) and a review report that contains all the results.

The components of a formal review plan are: the review goals, the collection of items being reviewed, a set of preconditions for the review, roles, team size, participants, training requirements, review steps and procedures, checklists and other related documents to be distributed to participants, the time requirements, the nature of the review log and summary report, and rework and follow-up criteria and procedures.

The list of components of a formal review report: checklist will all items checked and commented, list of defects found, list of attendees, review metrics (time and effort spent, size of the item being reviewed in lines of code or pages, number of defects found and ratios of defects/time, defects/size and size/time), status of the reviewed item (if it is accepted or to be re-inspected, depending on the number and gravity of defects found), estimate of rework effort and date for completion [7].

2.2 Usage of automated tools for static analysis

Static analysis tools compare favorably to manual reviews because they're faster, which means they can evaluate programs much more frequently, and they encapsulate some of the knowledge required to perform this type of code analysis in a way that it isn't require the tool operator to have the same level of expertise as a human auditor. Just as a programmer can rely on a compiler to consistently enforce the finer points of language syntax, the operator of a good static analysis tool can successfully apply that tool without being aware of the finer points of the more hard to find bugs. Furthermore, testing for errors like security vulnerabilities is complicated by the fact that they often exist in hard-to-reach states or crop up in unusual circumstances. Static analysis tools can peer into more of a program's dark corners with less fuss than dynamic analysis, which requires actually running the code. Static analysis has also the potential to be applied before a program reaches a level of completion at which testing can be meaningfully performed [4].

Good static analysis tools must be easy to use, this means that their results must be understandable to normal developers who might not know much about security and that they educate their users about good programming practice. Another critical feature is the kind of knowledge (the rule set) the tool enforces. The importance of a good rule set can't be overestimated. In the end, good static checkers can help spot

and eradicate common security bugs. Static analysis for security should be applied regularly as part of any modern development process.

That being said, static analysis tools cannot solve all of the security problems, mainly because these tools look for a fixed set of patterns, or rules, in the code. Although more advanced tools allow new rules to be added over time, if a rule hasn't been written yet to find a particular problem, the tool will never find that problem [4].

The output of static analysis tools still requires human evaluation. There's no way for a tool to know exactly which problems are more or less important for the programmer automatically, so there is no way to avoid studying the output and making a judgment call about which issues should be fixed and which ones represent an acceptable level of risk.

A tool can also produce false negatives (the program contains bugs that the tool doesn't report) or false positives (the tool reports bugs that the program doesn't contain). False positives cause a problem because of the time it may take the developer to understand there is no error after all, but false negatives are much more dangerous because they lead to a false sense of security. A good tool for static analysis is one that, although sometimes shows a false positive, never lets a false negative pass [4].

A further study of these tools can be found in sections 4 and 5 of this document.

3 The advantages and disadvantages of Static Code Analysis

The testing of a software application has many points of procedure, in order for it to be considered in conformance with the designated specifications of performance and usability. Static analysis can only gain meaning if the other forms of analysis are made, because nobody can only use this technique and be sure that the software is defect proof, which can be seen as a huge disadvantage. On the other hand, there is no way of ever being sure that the implementation is error free...

There are basically two types of software analysis: dynamic and static. As it was explained in section 2 of this document, static analysis is performed without actually executing programs built from that software, however dynamic analysis is performed by executing programs on a real or virtual processor. Although dynamic analysis checks the functional requirements of a software project, static analysis can decrease the amount of testing and debugging necessary for the software to be deemed ready.

The disadvantage of dynamic analysis is that the results produced are not generalized for future executions. There is no certainty that the set of inputs over which the program was run is characteristic of all possible program executions. Applications that require correct inputs (such as semantics-preserving code transformations) are unable to use the results of a typical dynamic analysis, just as applications that require precise inputs are unable to use the results of a typical static analysis [9].

Dynamic analysis can be as fast as program execution. Some static analyses run quite fast, but in general, obtaining accurate results requires a great deal of computation and long waits, especially when analyzing large programs. Typically, static analysis is conservative and sound. Soundness guarantees that analysis results are an accurate description of the program's behavior, no matter on what inputs or in what environment the program is run. Conservatism means reporting weaker properties than may actually be true; the weak properties are guaranteed to be true, preserving soundness, but may not be strong enough to be useful [9].

Software design is also prone to the existence of mistakes, such as the need to improve errors messages, badly structured specifications and models, and so on. These problems are difficult to detect via testing, mostly because most problems have their origin in requirements and design of software. Requirements and design artifacts can be reviewed but not executed and tested [6]. On the other hand, if the focus is set on the job of different developers or testers' teams, either it is impossible to find design related problems or problems of slow code development because of poorly organized and unstructured code.

One of the advantages of the static analysis approach during development is that the code is forcefully directed in a way as to be reliable, readable and lees prone to errors on future tests. This also influences the verification of the code after it is ready, reducing the number of problems found in further implementations that code.

A good example of the advantages of static analysis over the other types of analysis is this study: "Subject Project Study - Analysis Technique Comparison" [8]. The goal was to present "Code Reading versus Functional Testing versus Structural Testing". Comparing them in respect to fault detection effectiveness and cost classes of faults detected.

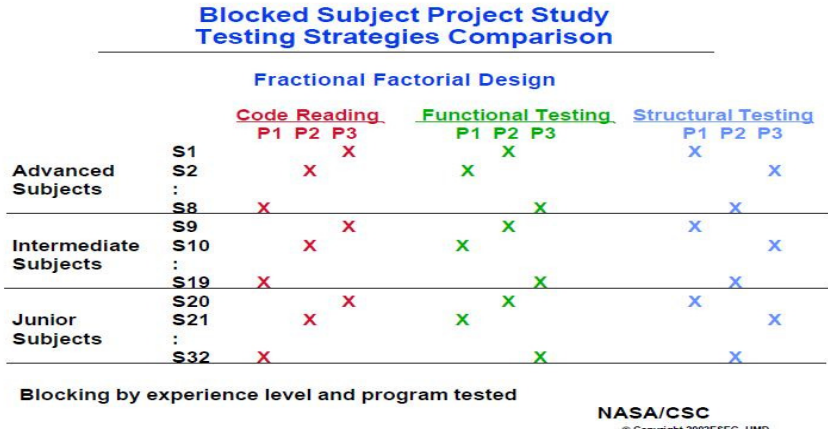

Fig. 2. Results of the NASA/CSC study on Code Reading versus Functional Testing versus Structural Testing.

This experience involves different groups of people, from the more experienced programmers to the junior ones, divided into several teams in order for each team to test a specific program separately, and each team uses a different method for evaluating the applications.

The results of this study by NASA/CSC prove that code reading can be "more effective than functional testing and more efficient than functional or structural testing" [8].

4 Tools for Static Code Analysis

Tools based on static analysis can be used to discover defects in programs. Several tools have been developed through the years in order to aid this process. The tools build on static analysis and can be used to find runtime errors as well as resource leaks and even some security vulnerabilities statically, i.e. without executing the code [10].

This section describes some of the most popular tools for static code analysis. These tools can be classified in the following categories: Microsoft .NET, Java, C/C++ and Multi-Language. In addition, these tools are either open-source or commercial ones.

4.1 Microsoft .NET

One static analysis tool within the Microsoft .NET Framework is **FxCop** [11], which is a free tool created by Microsoft. FxCop analyzes the intermediate code of a compiled .NET assembly and provides suggestions for design, security, and performance improvements. By default, FxCop analyzes an assembly based on the rules set forth by Design Guidelines for Developing Class Libraries. The design guideline rules are divided into nine categories, including design, globalization, performance, and security, among others. Furthermore, FxCop not only displays more than 200 rules that are used when analyzing an assembly but also allows the user to turn off existing rules and add custom ones. FxCop is intended for class library developers but is also useful as an educational tool for people who are new to the .NET Framework. This tool is available as a standalone application and includes a command-line implementation that makes it easy to plug into an automated build process.

Another free static code analysis tool from Microsoft is **StyleCop** [12]. Whereas FxCop evaluates design guidelines against intermediate code, StyleCop evaluates the style of C# source code in order to enforce both a set of style and consistency rules. Style guidelines are rules that specify how source code should be formatted. They dictate whether spaces or tabs should be used for indentation and the format of for loops, if statements and other constructs. Some StyleCop rules include: the body of for statements should be wrapped in opening and closing curly brackets; there should be white space on both sides of the = and != operators; and calls to member variables within a class must begin with "this".

One powerful but commercial static code analysis tool is **CodeIt.Right** [13] from vendor SubMain. It takes static code analysis to the next level by enabling rule violations to be automatically refactored into conforming code. Like FxCop, CodeIt.Right ships with an extensive set of predefined rules, based on the design guidelines mentioned earlier, with the ability to add custom rules. But CodeIt.Right makes it much easier to create and use custom rules, and is also capable of automatically fix the code issues it finds.

4.2 Java

In the Java world, there are many high-quality static analysis tools available for free. One recognized static analysis tool for Java code is **FindBugs**. It uses a series of ad-hoc techniques designed to balance precision, efficiency, and usability. One of its main techniques is to syntactically match source code to known suspicious programming practice [14].

PMD [15] is another static analysis tool that, like FindBugs, performs syntactic checks on program source code, but does not have a data flow component. In addition to some detection of clearly erroneous code, many of the "bugs" PMD looks for are stylistic conventions whose violation might be suspicious under some circumstances. For instance, having a try statement with an empty catch block might indicate that the caught error is incorrectly discarded. Since PMD includes many detectors for bugs that depend on programming style, PMD includes support for selecting which detectors or groups of detectors should be run. Additionally, PMD is easily extensible by developers, who can write new bug pattern detectors using either Java or XPath.

A further open source tool that enforces coding conventions and best practice rules for Java code is known as **CheckStyle**. It works by analyzing Java source code and reporting any breach of standards. It can be integrated in an IDE as a plug-in, so that developers can immediately see and correct any breaches of the official standards. In addition, it can also be used to generate project-wide reports that summarize the breaches found. Checkstyle includes more than 120 rules and standards, and deals with issues that range from code formatting and naming conventions to code complexity metrics [16].

Jlint [17] is a free static analysis tool. It will check Java code and find bugs, inconsistencies and synchronization problems by performing data flow analysis and building lock graph. Jlint performs local and global data flow analyses, calculating possible values of local variables and catching redundant and suspicious calculations. Except for deadlocks, Jlint is able to detect possible race condition problem, when different threads can concurrently access the same variables. Regarding message reporting, it uses a smart approach - all messages are grouped in categories, and it is possible to enable or disable reporting messages of specific category as well as concrete messages. Jlint is capable of remember reported messages and it won't report them once again when Jlint runs a second time. Nevertheless, Jlint is not easily expandable.

One more tool based on theorem proving, performs formal verification of properties of Java source code. The **ESC/Java**, Extended Static Checking system for

Java, is designed so that it can produce some useful output even without any specifications. In order to use ESC/Java, the developer needs to add preconditions, post conditions, and loop invariants to source code in the form of special comments. In addition, ESC/Java uses a theorem proofer to verify that the program matches the specifications. Its approach to finding bugs is notably different from the other mentioned tools [18].

4.3 C/C++

Lint [19] was the name originally given to a particular program that flagged suspicious and non-portable constructs (likely to be bugs) in C language source code. It can be used to detect certain language constructs that may cause portability problems. In addition, Lint can be used to check C programs for syntax and data type errors. It checks these areas of a program much more carefully than the C compiler does, displaying many messages that point out possible problems. Lint checks language semantics and syntax errors, considering areas such as: program flow; data type checking; variable and function checking; portability; and inefficient coding style.

A further commercial static analysis tool, **CodeSonar**, is a sophisticated source code tool that performs a whole-program, interprocedural analysis on C/C++ code and identifies complex programming bugs that can result in system crashes, memory corruption, and other serious problems. CodeSonar pinpoints problems at compile time that can take weeks to identify with traditional testing. Like a compiler, CodeSonar does a build of the code, but instead of creating object code it creates an abstract representation of the program. After the individual files are built, a synthesis phase combines the results into a whole-program model. The model is symbolically executed and the analysis keeps track of variables and how they are related. Warnings are generated when anomalies are encountered. CodeSonar does not need test cases and works with the existing build system [20].

Another static analysis tool for C and C++ programs is called **HP Code Advisor**. This commercial tool reports various programming errors in the source code. This tool enables programmers to identify potential coding errors, porting issues, and security vulnerabilities. HP Code Advisor leverages the advanced analysis capabilities of HP C and HP C++ compilers available on the HP Integrity systems. HP Code Advisor is a powerful static code analysis tool that automatically diagnoses various issues in a source program. HP Code Advisor leverages advanced cross-file analysis technology from HP compilers. It stores the diagnosed information in a program database. With the built-in knowledge of system APIs, HP Code Advisor looks deep into the code and provides helpful warnings with fewer false positives. HP Code Advisor detects a wide range of coding errors and potential problems such as memory leaks, used after free, double free, array/buffer out of bounds access, illegal pointer access, uninitialized variables, unused variables, format string checks, suspicious conversion and casts, out of range operations, C++ coding style warnings [21].

Mygcc [22] is an extension of the gcc compiler, supporting user-defined checks written in a simple formalism that can be checked efficiently. It can be customized very easily by adding user-defined checks for detecting for instance, memory leaks,

unreleased locks, or null pointer dereferences. User-defined checks are performed in addition to normal compilation, and may result in additional warning messages. Path queries can be run on the control-flow graph of functions, specifying a start node, a stop node, and constraints on the path in between. Gcc already includes many built-in checks such as uninitialized variables, undeclared functions, format string inspection. Mygcc allows programmers to add their own checks that take into account syntax, control flow, and data flow information. The implementation of mygcc as a lightweight patch to gcc, and is based on the disruptive concept of *unparsed pattern matching*, which make the patch easily portable.

Splint is a tool for statically checking C programs for security vulnerabilities and coding mistakes. With minimal effort, Splint can be used as better lint. If additional effort is invested adding annotations to programs, Splint can perform stronger checking than can be done by any standard lint. In addition, Splint checks unused declarations, type inconsistencies, use before definition, unreachable code, ignored return values, execution paths with no return, likely infinite loops, and fall through cases [23].

Another tool worth mentioning is **PolySpace Verifier**. It enables embedded software developers to detect run-time errors in C, C++ before they compile and run the code and prove automatically which operations are error-free. Overflows, out of bounds array index and divide-by-zero errors, amongst others, are easily detected by PolySpace which models the flow of data through the code. PolySpace Verifier can also be integrated into Model-Based Design tools to trace back errors to their root cause in the model. PolySpace's Verifier is used extensively for Embedded Software development and more especially in the Transportation, Defense, Aerospace and Automotive industries where there is a high expectation of safety-critical systems [24].

4.4 Multi-Language

Coverity Prevent is the leading automated approach for ensuring the highest quality, most reliable software at the earliest phase of the development lifecycle. The most accurate static code analysis solution available today, it automatically scans C/C++, Java and C# code bases with no changes to the code or build system. Because it produces a complete understanding over the build environment and source code, Prevent is the tool of choice for developers who need flexible, deep, and accurate source code analysis. Hundreds of development organizations worldwide use Prevent to automatically analyze large, complex code bases and root out the critical, must-fix defects that lead to system failures, runtime exceptions, security vulnerabilities, and performance degradation. Coverity Prevent offers the following benefits: automatically find critical defects that can cause data corruption and application failures; improve development team efficiency and speed time to market for critical applications; and improve software integrity and end-user satisfaction [25].

Most recently Klocwork announced the debut of a new static analysis tool that aims to ensure quality and security in the code development process, both at the level of the desktop and organization wide – **Klockwork Insight**. It applies complex static analysis techniques to C, C++, and Java and C# to automatically locate critical

programming bugs and security vulnerabilities in source code. By applying inter-procedural control flow, data flow, value-range propagation and symbolic logic evaluation, this tool can find hundreds of errors on well-validated, feasible execution paths. Furthermore, Insight is designed to fit within existing development process and is scalable to large organizations due to role-based access control and extended analysis capabilities such as parallelization, distributed and incremental analysis. Klocwork Insight is a groundbreaking approach to source code analysis that has been proven in some of the most demanding software development environments in the world [26].

Hammurapi is a versatile automated code review solution. This tool establishes code ascendancy processes in organizations by injecting automated code review "hooks" into development and build processes. Its main features include: robustness, by not failing on source files with errors and also don't fail if some inspectors throw exceptions; extensibility, since Hammurapi is a modular solution, as a result different pieces of the solution can be independently extended or even replaced; power, due to the fact that the tool uses a rules engine, to infer violations in the source code, allowing to implement non-trivial logic in inspectors; and is capable of reviewing sources in multiple programming languages **-** in modern applications, where pieces of Java or other programming language code are glued together with XML descriptors and where client-side JavaScript invokes server-side actions (AJAX, Flex), it is very important to have a holistic view of the application. Reviewing only Java or C# sources is not enough to ensure health of the application. Mesopotamia can parse and store source files written in different programming languages. Hammurapi works on Mesopotamia object model and as such can review sources in multiple programming languages, perform cross-language inspections, and generate a consolidated report [27].

RATS - Rough Auditing Tool for Security - is an open source tool developed and maintained by Secure Software security engineers. RATS is a tool for scanning C, C++, Perl, PHP and Python source code and flagging common security related programming errors such as buffer overflows and TOCTOU (Time Of Check, Time Of Use) race conditions. RATS scanning tool provides a security analyst with a list of potential trouble spots on which to focus, along with describing the problem, and potentially suggest remedies. It also provides a relative assessment of the potential severity of each problem, to better help an auditor prioritize. This tool also performs some basic analysis to try to rule out conditions that are obviously not problems. As its name implies, the tool performs only a rough analysis of source code. It will not find every error and will also find things that are not errors. Manual inspection of the code is still necessary, but greatly aided with this tool [28].

Understand is a commercial static code analysis software tool produced by SciTools. It is primarily used to reverse engineer, automatically document, and calculate code metrics for projects with large code-bases. Understand helps programmers to quickly comprehend, measure, maintain and document their source code. In addition, is fast and easy to use, it is a programmer's IDE oriented at maintenance tasks. It supports C/C++/C# and Java. Its most significant features are: semantic change analysis; advanced metrics; multi-scenario source code maintenance estimation; combined language analysis; custom architecture creation; and creation of code analysis snapshots [29].

5 Tool Comparison

Testing has the potential of finding most types of defects, however, testing is costly and no amount of testing will find all defects. Testing is also problematic because it can be applied only to executable code, i.e. rather late in the development process. Alternatives to testing, such as dataflow analysis and formal verification, have been known since the 1970s but have not gained widespread acceptance outside academia, that is, until recently; lately several commercial tools for detecting runtime error conditions at compile time have emerged [10].

This section demonstrates two evaluations performed on two different sets of static analysis tools. The first evaluation refers to ARCHER, BOON, PolySpace C Verifier, Splint, and UNO tools (some of these tools weren't mentioned in previous section due to space limitations). Four are open-source tools (ARCHER, BOON, SPLINT, UNO) and one is a commercial tool (PolySpace C Verifier). These tools have been evaluated using source code examples containing 14 exploitable buffer overflow vulnerabilities found in various versions of Sendmail (SM), BIND, and WU-FTPD. Each code example included a "BAD" case with and an "OK" case without buffer overflows. Buffer overflows varied and included stack, heap, bss and data buffers; access above and below buffer bounds; access using pointers, indices, and functions; and scope differences between buffer creation and use. Buffer overflow vulnerabilities often permit remote attackers to run arbitrary code on a victim server or to crash server software and perform a denial of service (DoS) attack. PolySpace and Splint detected a substantial fraction of buffer overflows while the other three tools generated almost no warnings for any model program. Boon had two confusions (detections combined with false alarms), one on each of SM and FTPD. Archer had one detection on SM and no false alarms. UNO generated no warnings concerning buffer overflows. The detection rates of three of the five systems tested were below 5% when tested on C source code modeled after those sections of open-source C WU-FTPD, Sendmail, and BIND server software that contain known and exploitable buffer overflows.
Even though two static analysis tools (Splint and PolySpace) had high detection rates of 87% and 57%, they are not without problems. These tools would have detected some in-the-wild buffer overflows, but warnings generated by them might have been ignored by developers annoyed by high false alarm rates. The false alarm rate measured just on the patched lines in the model programs was 43% and 50%. More concerning, perhaps, is the rate of false alarms per line of code, which for these tools is 1 in 12 and 1 in 46. Additionally, these tools do not appear to be able to discriminate between vulnerable source code and patched software that is safe, making them of little use in an iterative debugging loop. The tool with the best performance, PolySpace is slow enough to preclude common use; it takes days to analyze a medium-sized program of 100,000 lines [32].

The second evaluation refers to three market leading static analysis tools in 2006/07: PolySpace Verifier, Coverity Prevent and Klocwork K7 (now called Insight). The main objective of this study was to identify significant static analysis functionality provided by the tools, but not addressed in a normal compiler, and to survey the underlying supporting technology. The goal was not to provide a ranking of the tools; nor was it to provide a comprehensive survey of all functionality provided by the tools. While all three tools have much functionality in common, there

are noticeable differences, in particular when comparing PolySpace Verifier against Coverity Prevent and Klocwork K7. The primary aim of all three tools obviously is to find real defects, but in doing so any tool will also produce some false positives. While Coverity and Klocwork are prepared to sacrifice finding all bugs in favor of reducing the number of false positives, PolySpace is not; as a consequence the former two will in general produce relatively few false positives but will also typically have some false negatives. Coverity claims that approximately 20 to 30 per cent of the defects reported are false positives. Klocwork K7 seems to produce a higher rate of false positives, but stays in approximately the same league. PolySpace, on the other hand, does in general mark a great deal of code in orange color which means that it may contain a defect, as opposed to code that is green (no defects), red (definite defect) or grey (dead code). If orange code is considered a potential defect then PolySpace Verifier produces a high rate of false positives.

All three tools rely at least partly on inter-procedural analyses, but the ambition level varies significantly. PolySpace uses the most advanced technical solution where no execution sequences are forgotten, but some impossible execution paths may be analyzed due to the approximations made. Coverity Prevent and Klocwork K7 account only of interval ranges of variables in combination with "simple" relationships between variables in a local context with the main purpose to prune some infeasible execution paths, but do not do as well as PolySpace. Global variables and nontrivial aliasing are not accounted for or treated only in a restricted way. As a consequence neither Coverity nor Klocwork take all possible behaviors into account which is one source of false negatives. It is somewhat unclear how Coverity Prevent and Klocwork K7 compare with each other, but impression is that the former does a more accurate analysis. While PolySpace appears to be aiming primarily for the embedded systems market, Klocwork and Coverity have targeted in particular networked systems and applications as witnessed, for instance, by a range of security checkers.

Klocwork and Coverity address essentially the same sort of security issues ranging from simple checks that critical system calls are not used inappropriately to more sophisticated analyses involving buffer overruns (which is also supported by PolySpace) and the potential use of so-called tainted (untrusted) data. Coverity supports incremental analysis of a whole system, where only parts have been changed since last analysis. Results of an analysis are saved and reused in subsequent analyses. An automatic impact analysis is done to detect and, if necessary, re-analyze other parts of the code affected indirectly by the change. Such an incremental analysis may take significantly less time than analyzing the whole system from scratch. With the other tools analysis of the whole system has to be redone.

None of the tools provide very sophisticated support for dealing with concurrency. Klocwork currently provides no support at all. Coverity is able to detect some cases of mismatched locks but does not take concurrency into account during analysis of concurrent threads. The only tool which provides more substantial support is PolySpace which is able to detect shared data and whether that data is protected or not [10].

6 Proposed Improvements on Existing Tools

Tools used to identify bugs in source code often return large numbers of false positive warnings to the user. True positive warnings are often buried among a large number of distracting false positives. By making the true positives hard to find, a high false positive rate can frustrate users and discourage them from using an otherwise helpful tool. The use of historical data mined from the source code revision history can be useful in refining the output of a bug detector by relating code flagged by the tool to code changed in the past. Examining the code changes and the state of the code before and after the change may allow matching previous code changes to warnings produced by a bug finding tool. Warnings could be matched to code changes in a number of ways. The functions invoked the location in the code (module, API or function) or the control or data flow may be used to link the flagged code to the code from the repository. Warnings that flag code similar to code snippets that have been changed in the past may be more likely to be true positives [30].

In order to understand an error report, users must develop a way to take the information in the report and relate it to the potential problem with the code. Moreover, to decide whether an error report is a false positive, the user has to realize something about the sources of imprecision in the analysis. Therefore, they can create their own ad-hoc, inconsistent procedures that neglected some sources of imprecision. This situation can be addressed by encoding a triaging procedure as a checklist that enumerates the steps required to triage a specific report [31].

7 Discussion

Good static analysis tools must be easy to use. This means that their results must be understandable to normal developers so that they educate their users about good programming practices. Another critical feature is the kind of knowledge (the rule set) the tool enforces. The importance of a good rule set can't be overestimated. In the end, good static checkers can help spot and eradicate common security bugs. This is especially important for languages such as C, for which a very large corpus of rules already exists. Static analysis should be applied regularly as part of any modern development process.

Static code analysis tools provide a fast, automated way to ensure that source code remains to predefined design and style guidelines. Following such guidelines helps produce more uniform code and also can point out potential security, performance, interoperability, and globalization shortcomings. Static code analysis tools are not a replacement for human-led code reviews. Rather, they can generate a first pass of the code base and highlight areas that require more attention from a senior developer.

References

1. Kan, Stephen H.: Metrics and Models in Software Quality Engineering. Pearson Education, Inc., Boston (2003)

2. Fagan, M. E.: Design and code inspections to reduce errors in program development. IBM (1976)
3. Faria, J. P..: Inspections, revisions and other techniques of software static analysis. Software Testing and Quality, Lecture 9, FEUP, Porto (2008)
4. McGraw, G., Chess, B.: Static Analysis for Security. IEEE Computer Society (2004)
5. Klocwork. Early bug detection, comprehensive coverage. Klocwork Inc. (2008) http://www.klocwork.com/solutions/defectDetection.asp
6. Humphrey, W.: The Personal Software Process^{SM} (PSP^{SM}). Carnegie Mellon University, Massachusetts (2000)
7. Faria, J. P.: Software Reviews and Inspections. FEUP, Porto (2008)
8. Basili, V.: Experimentation in Software Engineering. Experimental Software Engineering Group, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland and Fraunhofer Center for Experimental Software Engineering, Maryland (2002)
9. Ernst, M.: Static and dynamic analysis: synergy and duality. MIT Lab for Computer Science, Cambridge, Workshop on Dynamic Analysis, ICSE'03 International Conference on Software Engineering Portland, Oregon (2003)
10. Emanuelsson, P.,Nilsson, U.: A Comparative Study of Industrial Static Analysis Tools. Linköping University Electronic Press (2008)
11. FxCop, http://msdn.microsoft.com/en-us/library/bb429476.aspx (2008)
12. Microsoft StyleCop, http://code.msdn.microsoft.com/sourceanalysis (2008)
13. CodeIt.Right, http://submain.com/?nav=products.cir (2007)
14. Static Analysis, http://weblogs.java.net/blog/gsporar/archive/talking_about_s.html (2006)
15. PMD, http://pmd.sourceforge.net/ (2008)
16. CheckStyle, http://checkstyle.sourceforge.net/ (2007)
17. Jlint, http://www.download.com/Jlint/3000-2218_4-10213979.html (2003)
18. ECS/Java, http://en.wikipedia.org/wiki/ESC/Java (2008)
19. Lint, http://en.wikipedia.org/wiki/Lint_programming_tool (2008)
20. CodeSonar Source-Code Analyser, http://www.scl.com/grammatech/codesonar (2008)
21. HP Code Advisor, http://en.wikipedia.org/wiki/HP_Code_Advisor (2007)
22. mygcc, http://mygcc.free.fr/ (2008)
23. Splint analysis, http://rozinov.sfs.poly.edu/presentations/splint_analysis.pdf (2003)
24. PolySpace Verifier, http://www.softdevtools.com/modules/weblinks/singlelink.php?lid=116 (2007)
25. Coverity -A Higher Code, http://www.coverity.com/library/pdf/coverity_prevent.pdf (2008)
26. Klocwork - Static source code analysis, http://www.klocwork.com/products/insight.asp (2008)
27. Automated code review system – Hammurapi, http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/hammurapi-group/products/hammurapi/ (2007)
28. RATS, http://www.fortify.com/security-resources/rats.jsp (2007)
29. Understand – Source Code Analysis, http://www.scitools.com/products/understand/ (2008)
30. Williams, C.C., Hollingsworth, J.K.: Using Historical Information to Improve Bug Finding Techniques, Workshop on the Evaluation of Software Defect Detection Tools (2005)
31. Phang, K.Y., Foster, J.S, Hicks, M., Sazawal, V.: Path Projection for User-Centered Static Analysis Tools, University of Maryland, College Park (2008)
32. Zitser, M., Lippmann, R., Leek, T.: Testing Static Analysis Tools using Exploitable Buffer Overflows from Open Source Code (2004)