Technical reports in Computer and Information Science

Report number 2008:3

A Comparative Study of Industrial Static Analysis Tools (Extended Version)

by

Pär Emanuelsson and Ulf Nilsson
par.emanuelsson@ericsson.com, ulfni@ida.liu.se

January 7, 2008

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

A Comparative Study of Industrial Static Analysis Tools (Extended Version)

Pär Emanuelsson
Ericsson AB
Datalinjen 4
SE-583 30 Linköping, Sweden
par.emanuelsson@ericsson.com

Ulf Nilsson
Dept of Computer and Info. Sci.
Linköping University
SE-581 83 Linköping, Sweden
ulfni@ida.liu.se

January 7, 2008

1 Introduction

Almost all software contains *defects*. Some defects are found easily while others are never found, typically because they emerge seldom or not at all. Some defects that emerge relatively often even go unnoticed simply because they are not perceived as errors or are not sufficiently severe. Software defects may give rise to several types of *errors*, ranging from logical/functional ones (the program sometimes computes incorrect values) to runtime errors (the program typically crashes), or resource leaks (performance of the program degrades possibly until the program freezes or crashes). Programs may also contain subtle security vulnerabilities that can be exploited by malicious attackers to gain control over computers.

Fixing defects that suddenly emerge can be extremely costly in particular if found at the end of the development cycle, or worse: after deployment. Many simple defects in modern programming languages can be found by modern compilers, e.g. in statically typed languages. But the predominating method for finding defects is *testing*. 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. 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. This paper is a survey and comparison of three market leading static analysis tools: PolySpace Verifier, Coverity Prevent and Klocwork K7. The list is by no means exhaustive, and the list of competitors is steadily increasing, but the three tools represent state-of-the-art in the field at the moment.

The main objective of this study is (1) to identify significant static analysis functionality provided by the tools, but not addressed in a normal compiler, and (2) to survey the underlying supporting technology. The goal is not to provide a ranking of the tools; nor is it to provide a comprehensive survey of all functionality provided by the tools. Providing such a ranking is problematic for at least two reasons: Static analysis is generally only part of the functionality provided by the tool; for instance, Klocwork K7 supports both refactoring and software metrics which are not supported by the two other tools. Even if restricting attention only to static analysis functionality the tools provide largely non-overlapping functionality. Secondly, even when the tools seemingly provide the same functionality (e.g. detection of dereferencing of null pointers) their solutions are often not comparable; each tool typically finds defects which are not found by any of the other tools.

Studying the internals of commercial and proprietary tools is not without problems—in particular, it is impossible to get full information about technical solutions. However, *some* technical information is publicly available in manuals and white papers; some of the tools also originate from academic tools which have been extensively described in research journals and conference proceedings. While technical solutions may have changed since then, we believe that such information is still largely valid. We have also consulted representatives from all three providers with the purpose to validate our descriptions of the tools. Still it must be pointed out that the descriptions of suggested technical solutions is subject to a certain amount of guessing in some respects.

The rest of the report is organized as follows: In Section 2 we define what we mean by the term static analysis and survey some elementary concepts and preconditions; in particular, the trade off between precision and analysis time. Section 3 contains a description of basic principles of static analysis. In Sections 4–6 we survey the static analysis functionality provided by PolySpace Verifier/Desktop, Coverity Prevent and Klocwork K7 focusing in particular on the support for the C and C++ programming languages. Section 7 addresses the complementary issue of programming guidelines such as those of The Motor Software Reliability Association (MISRA). Section 8 contains a qualitative comparison of the three tools summing up their relative merits and shortcomings. The section also surveys several industrial evaluations of the tools over time at Ericsson, in particular involving the products from Coverity and Klocwork. Section 9 contains conclusions.

2 Static analysis

Languages such as C and, to a lesser extent, C++ are designed primarily with efficiency and portability in mind[1], and therefore provide little support to avoid or to deal with runtime errors. For instance, there is no checking in C that read or write access to an array is within bounds, that dereferencing of a pointer variable is possible (that the variable is not null) or that type casting is well-defined. Such checks must therefore be enforced by the programmer. Alternatively we must make sure that the checks are not needed, i.e. guarantee that the error conditions will never occur in practice.

By the term *static analysis* we mean *automatic* methods to reason about runtime properties of program code without actually executing it. Properties

[1]Or so it is often claimed; in fact, even in ANSI/ISO Standard C there are many language constructs which are not semantically well-defined and which may lead to different behavior in different compilers.

that we consider include those which lead to premature termination or ill-defined results of the program, but precludes for instance purely syntactic properties such as syntax errors or simple type errors.[2] Nor does static analysis address errors involving the functional correctness of the software. Hence, static analysis can be used to check that the program execution is not prematurely aborted due to unexpected runtime events, but it does not guarantee that the program computes the correct result. While static analysis *can* be used to check for e.g. deadlock, timeliness or non-termination there are other, more specialized, techniques for checking such properties; although relying on similar principles. Static analysis should be contrasted with *dynamic analysis* which concerns analysis of programs based on their execution, and includes e.g. testing, performance monitoring, fault isolation and debugging.

Static analysis is useful in several respects. It can be used to detect certain types of software runtime errors—e.g. division by zero, arithmetic overflows, array indices out of bounds, buffer overflows etc—without actually executing the code. However, static analysis does not in general guarantee the absence of runtime errors. While static analysis can reduce the need for testing or even detect errors that in practice cannot be found by testing, it is not meant to replace testing.

In addition to finding errors, static analysis can also be used to produce more efficient code; in particular for "safe" languages like Java, where efficiency was not the primary goal of the language designers. Many runtime tests carried out in Java programs can in practice be avoided given certain information about the runtime behavior. For instance, tests that array indices are not out-of-bounds can be omitted if we know that the value of the indices are limited to values in-bounds. Static analysis can provide such information.

Static analysis can also be used for type inference in untyped or weakly typed languages or type checking in languages with non-static type systems [22]. Finally static analysis can be used for debugging purposes (see e.g. [1]), for automatic test case generation (see e.g. [17]), for impact analysis (see e.g. [26]), intrusion detection (see e.g. [28]) and for software metrics (see e.g. [29]). However, in this paper we focus our attention on the use of static analysis for finding defects and software vulnerabilities which typically would not show up until the code is executed.

2.1 Detecting runtime errors

The following is a non-exhaustive list of runtime problems that typically cannot be detected by a traditional compiler, but which can potentially be detected by static analysis:

- Improper resource management: Resource leaks of various kinds, e.g. dynamically allocated memory which is not freed, files, sockets etc. which are not properly deallocated when no longer used;
- Illegal operations: Division by zero, calling arithmetic functions with illegal values (e.g. non-positive values to logarithm), over- or underflow in

[2]The borderline is not clear; some checks done by compilers, such as type checking in a statically typed language, are closer to runtime properties than syntactic ones. In fact, in a sufficiently rich type system some type checking must be done dynamically.

arithmetic expressions, addressing arrays out of bounds, dereferencing of null pointers, freeing already deallocated memory;

- Dead code and data: Code and data that cannot be reached or is not used. This may be only bad coding style, but may also signal logical errors or misspellings in the code;
- Incomplete code: This includes the use of uninitialized variables, functions with unspecified return values (due to e.g. missing return statements) and incomplete branching statements (e.g. missing cases in `switch` statements or missing `else` branches in conditional statements).

Other problems checked for by static analysis include non-termination, uncaught exceptions, race conditions etc.

2.2 Precision and time of analysis

Most interesting properties checked by static analyses are *undecidable*, meaning that it is impossible, even in theory, to determine whether an arbitrary program exhibits the property or not. As a consequence static analyses are inherently *imprecise*—they typically infer that a property (e.g. a runtime error) *may* hold. This implies that

1. if a program has a specific property, the analysis will usually only be able to infer that the program *may* have the property. In some cases the analysis may also be able to infer that the program *must* have the property.
2. if the program does not have the property, there is a chance that (a) our analysis is actually able to infer this (i.e. the program *must not* have the property), but it may also happen that (b) the analysis infers that the program *may* have the property.

If the property checked for is a defect then we refer to case 2(b) as a *false positive*. Hence, if the analysis reports that a program may divide by zero we cannot tell in general whether it is a real problem (item 1) or if it is a false positive (item 2(b)). The precision of the analysis determines how often false positives are reported. The more imprecise the analysis is, the more likely it is to generate false positives.

Unfortunately precision usually depends on analysis time. The more precise the analysis is, the more resource consuming it is, and the longer it takes. Hence, precision must be traded for time of analysis. This is a very subtle trade-off—if the analysis is fast it is likely to report many false positives in which case the alarms cannot be trusted. On the other hand a very precise analysis is unlikely to terminate in reasonable time for large programs.

One way to avoid false positives is to filter the result of the analysis, removing potential errors which are unlikely (assuming some measure of likelihood). However, this may result in the removal of positives which are indeed defects. This is known as a *false negative*—an actual problem which is not reported. False negatives may occur for at least two other reasons. The first case is if the analysis is too optimistic, making unjustified assumptions about the effects of certain operations. For instance, not taking into account that `malloc` may return null. The other case which may result in false negatives is if the analysis is incomplete; not taking account of all possible execution paths in the program.

There are a number of well-established techniques that can be used to trade-off precision and analysis time. A *flow-sensitive* analysis takes account of the control flow graph of the program while a *flow-insensitive* analysis does not. A flow-sensitive analysis is usually more precise—it may infer that `x` and `y` may be aliased (only) after line 10, while a flow-insensitive analysis only infers that `x` and `y` may be aliased (anywhere within their scope). On the other hand, a flow-sensitive analysis is usually more costly.

A *path-sensitive* analysis considers only valid paths through the program. It takes account of values of variables and boolean expressions in conditionals and loops to prune execution branches which are not possible. A path-insensitive analysis takes into account all execution paths—even infeasible ones. Path-sensitivity usually implies higher precision but is usually more time consuming.

A *context-sensitive* analysis takes the context—e.g. global variables and actual parameters of a function call—into account when analyzing a function. This is also known as *inter-procedural* analysis in contrast to *intra-procedural* analysis which analyses a function without any assumptions about the context. Intra-procedural analyses are much faster but suffer from greater imprecision than inter-procedural analyses.

The undecidability of runtime properties implies that it is impossible to have an analysis which always finds all defects and produces no false positives. A framework for static analysis is said to be *sound* (or *conservative* or *safe*) if all defects checked for are reported, i.e. there are no false negatives but there may be false positives.[3]Traditionally, most frameworks for static analysis have aimed for soundness while trying to avoid excessive reporting of false positives. However, most commercial systems today are not sound (i.e. they will not find all actual defects) and also typically produce some false positives.

3 Basic principles

In this section we survey some aspects of the underlying machinery needed to build static analysis tools and where the different vendors typically make their own design decisions; thus affecting the performance—precision as well as analysis time/scalability—of static analysis tools.

3.1 Computing with sets of states

Some properties checked by static analysis tools can be carried out by relatively straightforward pattern matching techniques. However, most properties are more challenging and require much more sophisticated analysis techniques. It is

[3]Soundness can be used in two completely different senses depending on if the focus is on the reporting of defects or on properties of executions. In the former (less common) sense soundness would mean that all positives are indeed defects, i.e. there are no false positives. However, the more common sense, and the one used here, is that soundness refers to the assumptions made about the possible executions. Even if there is only a small likelihood that a variable takes on a certain value (e.g. `x=0`) we do not exclude that possibility. Hence if the analysis infers that `X` *may* be zero in an expression `1/x`, there is a possibility that there will be a runtime error; otherwise not. This is why a sound analysis may actually result in false positives, but no false negatives.

often claimed that static analysis is done without executing the program, but for nontrivial properties this is only partially true. Static analysis usually implies executing the program—not in a standard way, but on an abstract machine and with a set of abstract non-standard values replacing the standard ones. The underlying concept is that of a *state*; a state is a collection of program variables and the association of values to those variables. State information is crucial to determine if a statement such as `x=x/y` may result in division by zero (it *may* do so if `y` *may* have the value zero at the time when the division is made).

In the case of an intra-procedural analysis the state takes account only of local variables while a context-sensitive analysis must take account also of global variables plus the contents of the stack and the heap. The program statements are state transformers and the aim of static analysis is to associate the set of all possible states with all program points. Such sets of states are typically infinite or at least very large and the analysis must therefore resort to some simplified description of the sets representing only some of the relationships between the program variables, e.g. tracking an interval from which a variable *may* take its value.

For instance, instead of computing with the integers we may compute with values that describe some property of the integers; we may as a simple example replace the domain of integers with the finite domain $\{\ominus, 0, \oplus, ?\}$ where "$\ominus$" designates a negative integer (i.e. the interval $]-\infty, -1]$), "0" designates the integer 0 (the interval$[0, 0]$), "$\oplus$" designates a positive integer (the interval $[1, \infty[$) and "?" designates any integer (the interval $]-\infty, \infty[$). Operations, such as addition, which normally operate on the integers must be redefined over the new domain and in such a way that the abstract operation mimics the concrete one in a faithful way. For instance, we may replace addition of integers with "abstract addition" where for instance the addition of two negative integers is a negative integers, while the addition of a positive and negative integer can result in any integer. The abstract operation of addition can be defined as follows:

$+$	$\ominus$	0	$\oplus$	?
$\ominus$	$\ominus$	$\ominus$	?	?
0	$\ominus$	0	$\oplus$	?
$\oplus$	?	$\oplus$	$\oplus$	?
?	?	?	?	?

Such abstractions leads to loss of information which influences the precision of the analysis; if we know that `x = 4` and `y = -3` then we know that `x + y` is positive, but if we only know that `x =` $\oplus$ and `y =` $\ominus$ then we can only safely infer that `x + y` is an integer.

Some vendors provide very sophisticated machinery to track the set of possible values taken by variables and take precautions not to make unjustified assumptions about the result of operations. However, most vendors take a more pragmatic approach, for instance associating intervals with variables and sometimes do not even provide sound approximations of the operations.

3.2 Analysis of incomplete code

It is sometimes claimed that static analysis can be applied to incomplete code (individual files and/or procedures). While there is some truth to this, the quality of such an analysis may be arbitrary bad. For instance, if the analysis does not know how a procedure or subprogram in existing code is called from outside it must, to be sound, assume that the procedure is called in an arbitrary way, thus analyzing executions that probably cannot occur when the missing code is added. This is likely to lead to false positives. Similarly incomplete code may contain a call to a procedure which is not available, either because it is not yet written, or it is a proprietary library function. Such incomplete code can be analyzed but is also likely to lead to a large number of false positives and/or false negatives depending on if the analysis is sound or not.

On the positive side, it is often not necessary to provide complete code for missing functions or function calls. It is often sufficient to provide a stub or a top-level function that mimics the effects of the properties checked for. As an example, a partial function that maps an integer to a natural number and may fail due to division by zero can be mimicked by the following stub (where `random_int()` is assumed to return a random integer):

```
int foo(int n)
{
  if (rand()) return random_int();
  else return 0/0;
}
```

The tools studied in this report adopt different approaches to deal with incomplete code and incremental analysis when only some code has been modified.

3.3 Aliasing

Two pointer variables are said to be aliased if they point to the same location. The existence of aliasing means that we can change the value pointed to by a variable through a second variable. For example

```
int x = 17;
int *y;
y = &x; // Aliasing
x = 1;  // Also changing the value of y
```

Aliasing is not only a major source of programming errors, it is also a major source of imprecision; infering whether two variables are aliased is an undecidable problem so we cannot in general tell whether two variables are aliased or not. Hence, if the analysis infers that two variables *may* be aliased, we must assume that a change in one variable *may* also affect the value of the other variable.

A sound static analysis of programs using pointers would be almost impossible without a reasonable aliasing analysis. The emergence of commercial systems for static analysis is largely due to the development of efficient aliasing analyses; e.g. the inter-procedural analyses of A. Deutsch [12] and the very efficient and flow-insensitive analysis of B. Steensgaard [27].

An accurate and fast aliasing analysis is necessary to draw sound conclusions about runtime behavior of code. Unfortunately, most tools do not do a full aliasing analysis, and often make unjustified assumptions about the non-aliasing of pairs of variables. This is often done in the name of efficiency (and perhaps to reduce the number of false positives) but, as pointed out, there are very efficient aliasing analyses, and the lack of aliasing analysis may lead to many false negatives in code that make use of pointers.

4 PolySpace Verifier and Desktop

PolySpace Technologies provides a suite of tools including PolySpace Verifier and PolySpace Desktop—static analysis tools based on techniques of A. Deutsch and the work of P. Cousot's group. Our study is based on version 2.5 of Verifier and Desktop. PolySpace Verifier is based on classic lattice-theoretic static analysis techniques known as *abstract interpretation* [5]—the underlying analyzer relies on a sound approximation of the set of all reachable states. PolySpace Desktop relies on similar techniques but is intended for file/unit analysis and lacks some of the functionality of Verifier. Judging from available documentation the tools rely on so-called convex polyhedra to encode sets of states, see P. Cousot and N. Halbwachs [7].[4]

The properties checked by PolySpace Verifier are in many cases similar as those checked e.g. by other commercial systems, but the analysis is more sophisticated taking account of non-trivial relationships between variables (taking advantage of convex polyhedra) while other static analysis tools seem to cater only for simple relationships (e.g. equalities between variables and variables being bound to constant values or intervals of values). Given a complete code base, the PolySpace analysis is sound implying that it computes a superset of all reachable states of the program; it is also flow-sensitive, inter-procedural, context-sensitive and supports aliasing (probably relying on an aliasing analysis developed by A. Deutsch [12]). Based on the analysis the code is "colored" as follows:

- Red: code that *must* lead to runtime problems in all executions,
- Orange: Code which may, or may not, lead to errors,
- Grey: Dead code, i.e. code that will never be executed,[5]
- Green: Code that cannot contain any of the runtime errors checked for.[6]

Code may be colored orange for two reasons: it will be colored orange if there are some executions which do not fail, while some others do. But code may also be orange because of imprecision: the analysis may involve executions which cannot

[4]A convex polyhedron is an n-dimensional geometric shape where for any pair of points inside the shape the straight line connecting the points is also inside the shape. In practice this implies that the shape is given by a set of m linear inequalities of the form $Ax \leq B$ where A is an $m \times n$ matrix and B is an m-vector of reals. A convex polyhedron can be used to represent complex relationships between numerical program variables, e.g. $x - y \leq 0, -y \leq 0, x \leq 3$ represents all states where y takes a value between 0 and 3 and x takes a value between 0 and the value of y.

[5]To be more precise, code may also be grey when preceded by red code, i.e. code that follows after a defect will be considered unreachable.

[6]To be more precise, defective code may be green when preceded by orange code—for instance, if there is a potential out-of-bounds access `a[i]` to an array then that particular access will be orange but in subsequent code it will be assumed that the index `i` is in-bounds so a subsequent access to the same array with the same index will be green.

actually occur, but which are anyway considered because of approximations made when representing relationships between variables.

Soundness of the analysis implies that any error checked for must be either in red or orange code. That is, there can be no false negatives.[7] However, there may be code which should be red which is orange, and there may be code which is red or green which should be grey. More importantly perhaps, non-erroneous code may be marked as orange. That is, there may be false positives; in the documentation [24] it is claimed that programs consisting of 50KLoC (50,000 lines of code) may have between 400 to 8,000 orange warnings. For smaller code bases the percentage of oranges is typically between 2 to 15 per cent according to PolySpace representatives. However, the ratio obviously depends both on code size and, in particular, coding style. An optimal code size with respect to the rate of oranges is between 20–50KLoC according to the documentation. The rate of oranges also depends on the accuracy of the analysis—it is possible to tune the analysis by specifying one out of a handful of levels of precision where the most sophisticated level typically has a very low level of oranges but is not recommended for code exceeding 1KLoC and where the coarsest analysis can cope with more than 1,000KLoC but with a very high rate of code colored orange.

The documentation of PolySpace Verifier and Desktop is vague in some respects. It seems that the tools are unable to deal well with certain language constructs. The manual mentions specifically gotos with backward jumps, but there is an ad-hoc trick to circumvent the shortcoming.

PolySpace provides an automatic stubbing mechanism. Automatically generated stubs are (sometimes incorrectly) assumed not to have effect on global data by default. To avoid false positives, and more seriously false negatives, it is recommended to do manual stubbing. There are pragmas that can be used to express global effects of the stubs. There is also a very useful mechanism based on *assertions* which can be used to express local as well as global effects. Some predefined functions are available as stubs but PolySpace does not seem to provide as extensive support as e.g. Coverity.

There is support for dealing with concurrency. The tool produces a *variable dictionary* and *concurrent access graph* that depicts shared variables and whether they are protected or not. Shared data may be colored orange (data subject to race conditions) or green (protected data). The sharing analysis takes possible aliasing and atomicity into account, and subsequent analysis of concurrent programs is possible assuming that shared data is indeed protected. If there are possible race conditions it is still possible to model shared memory by use of the type qualifier `volatile`.

PolySpace Verifier supports analysis of programs in C, C++ and Ada. While probably feasible to analyze general purpose code, PolySpace recommends using the tool in particular for embedded systems in automotive, avionics and space industry, telecommunication systems and medical devices. PolySpace's sophisticated mechanism for tracking linear relationships between variables makes it particularly useful for analyzing arithmetic operations. However, there is no support at all for e.g. memory leaks or stack overflow. The complexity of tracking relationships between variables also implies that the tool does not deal well

[7]The absence of false negatives obviously only concerns the properties checked for; runtime errors not checked for by PolySpace Verifier—like memory leaks—will not be detected.

with very large programs unless the code is manually partitioned; in the documentation 50KLoC is mentioned as an approximate upper limit, but the measure obviously depends e.g. on coding style and the precision of the analysis.

The philosophy and functionality of the PolySpace tool set has a great deal in common with the coding standards of MISRA (Motor Industry Software Reliability Association) (MISRA-C:2004 [21]). PolySpace even provides a tool for checking MISRA compliance (the MISRA Checker). Conversely, adopting a coding standard such as MISRA-C is likely to influence static analysis positively, both in analysis time and in precision. See also Section 7.

4.1 Checkers

PolySpace Verifier analyses violation of the following runtime properties for the C language. For further information on the checkers and examples see the documentation of PolySpace for C [24]:

- `COR` (Array conversion must not extend range) Checks that an array variable is not assigned the value of another array variable pointing to a smaller array. Such an operation is likely to lead to future problems when the array is accessed.[8]

- `OBAI` (Array index within bounds) Checks that an array index is within bounds when used to access an array.

- `IRV` (Initialized Return Value) Checks that functions return well-defined values. The less likely reason is that the function returns a value of an uninitialized variable. The more likely reason is that the function contains some execution paths that are not terminated by a return statement. This would typically happen in a function defined only for non-negative integers which is called with a negative integer. The error is not signaled if the value is not used by the caller.

- `NIV` (Non-Initialized Variable) Checks that variables are initialized before being accessed. Also compound data (arrays and structures) are checked. The use of non-initialized variables may result in unpredictable results, but not necessarily a runtime error.

- `NIP` (Non-Initialized Pointer) Checks that pointers are initialized (i.e. point to some well-defined location) before being dereferenced the first time.

- `ASRT` (User Assertion) Tries to establish whether a user assertion/invariant is valid or not. The check relies on the standard library `assert.h`. Normally, when such an assertion is encountered during runtime the assertion is checked on-the-fly; if the assertion evaluates to true the execution continues but if the assertions fails program execution aborts (i.e. results in a runtime error). The checker tries to verify the assertion statically. If the assertion is green it is known to hold in all executions. The use of assertions makes it possible, in theory at least, to prove more general properties of programs, including functional properties.

[8]It is not mentioned in the manual but the same problem arises for all type casts of pointers, where a pointer is set to point to data which is too small according to the type of the pointer.

- `UNFL` (Scalar and Float Underflows) Checks if an arithmetic expression (integer or float) leads to underflow.
- `OVFL` (Scalar and Float Overflows) Checks if an arithmetic expression (integer or float) leads to overflow.
- `ZDV` (Scalar or Float Division by zero) Checks if the denominator of division is zero.
- `SHF` (Shift amount in 0..31 or 0..63) Checks if the result of a shift (left or right) is greater than the size of an integer respectively a long integer.
- `SHF` (Left operand of left shift must be positive) Checks if a left shift is applied to a signed number. Shifting a signed number to the left will corrupt sign information.
- `COR` (Function pointer must point to a valid function) Checks if a function pointer actually points to a function, or to a function with a valid prototype.
- `COR` (Wrong type for argument) Checks if the actual arguments passed to a function match the formal arguments of the function. Checking this is not possible in general in the presence of function pointers.
- `COR` (Wrong number of arguments) Checks if the number of actual arguments passed to a function (through a function pointer) matches the number of formal arguments.
- `IDP` (Pointer within bounds) Checks if a dereferenced pointer is still within bounds of the object pointed to.
- `NTC` (Non Termination of Call) Checks if a procedure call returns. The check reports a defect when the procedure loops infinitely or if the procedure leads to a runtime error, or if the procedure relies on other procedures which may fail to terminate. There is a mechanism k-NTC (Known Non Termination of Call) which facilitates not coloring procedures which are meant not to terminate and which can be provided by command line arguments.
- `NTL` (Non Termination of Loop) Checks if a loop (for, do-while or while) terminates. According to the documentation NTL, as well as NTC, can only be red, and never orange, suggesting that it may find a definite loop but may fail to find calls/loops which may terminate in some executions.
- `NTC` (Arithmetic expressions) Checks if arithmetic expressions functions are used with valid arguments; e.g. square-root and logarithm must be positive.
- `UNR` (Unreachable Code) Checks if "code snippets" (assignments, returns, conditional branches and function calls) may be reached.

5 Coverity Prevent

Coverity was founded in 2002 and is a spin-off company from a research group led by Dawson Engler at the Stanford Computer Science Department [13]. The products Coverity Prevent [10] and Coverity Extend [8] are developments of the academic systems xgcc and Metal respectively. Descriptions of the principal underlying technology of xgcc and Metal can be found in a series of research papers, see e.g. [15]. Our survey of Prevent and Extend is based on available documentation (version 2.4), other publicly available reports and information from company representatives. We believe that Prevent and Extend still rely largely on technology developed for xgcc and Metal but do not have full insight in present technology.

In 2006 Coverity and Stanford were awarded a substantial grant from the U.S. Department of Homeland Security to improve Coverity tools to hunt for bugs and vulnerabilities in open-source software. During the first year 5,000 defects were fixed in some 50 open source projects. Updated results of the analyses can be found on the web [11].

Coverity Prevent is a dataflow analysis tool that relies on inter-procedural analysis techniques. The analysis is neither sound nor complete, that is, there may be both defects which are not reported and there may be false alarms. A substantial effort has however been put on eliminating false positives, and the rate of these is clearly low (reportedly around 20 per cent). It is less clear what the rate of false negatives is, but case-studies described later indicate that the tool fails to report a number of known faults; at least when used out-of-the-box. It is also unclear if the presence of false negatives is a consequence of too aggressive filtering of error messages or if it is due to a partial analysis of the source code—probably a combination of the two. Judging from non proprietary information there is a probabilistic technique based on clustering of error messages and Bayesian learning which can be used to detect inconsistent use of programming constructs and which may also be used to filter error messages, see e.g. [18]. It is claimed that the tool investigates all possible code paths [9], but this does not necessarily imply that all possible data values are taken into account. The analysis is path sensitive but Coverity does not track complex relationships between variables as is done by PolySpace, but keeps track only of intervals and simple relationships between variables. Moreover Prevent does not track values of global variables at all; it is unclear if this source of imprecision is dealt with pessimisticly, in which case impossible paths would have to be analyzed, or optimistically, in which case possible paths are not analyzed at all.

The user can partly influence to what extent the analysis should suppress potential errors, e.g. by specifying a bound on the number of iterations of loops.

Coverity Prevent implements an *incremental* analysis which means that the system automatically infers what parts of the source code that have to be re-analyzed after the code has been modified. This typically reduces the analysis time substantially, but may of course imply a complete re-analysis in the worst case.

Coverity Prevent provides a useful meta-language for writing stubs and a somewhat primitive mechanism, based on XML, to mask out certain events in functions or stubs. It is also possible to add annotations (as C/C++ comments) to the code in order to avoid some false positives. The system comes with an extensive library of models of standard libraries for various platforms.

The version we have studied (2.4) supports analysis of C and C++ only but Java support is on the way. There is build and analysis support for most platforms (Linux/Unix clones, Windows and Mac OS X), numerous C and C++ compilers and the tool integrates with standard IDEs. The interface relies largely on command line input and scripts which makes it very flexible albeit somewhat crude, but there is a multi user GUI based on HTML for analyzing the error reports.

The user documentation is good, the functionality of the tool and how to use it is well described and there are numerous examples.

5.1 Coverity C checkers

Coverity Prevent version 2.4 provides the following checkers for C source code:

- `RESOURCE_LEAK` Detects leaks of memory, files or sockets. Resource leaks may result in degraded performance and even premature termination of the program. There are options to analyze fields of structures and aliasing between variables.

- `USE_AFTER_FREE` Detects dereferencing or deallocation of memory already deallocated. This includes both trying to free memory which has already been freed, as well as trying to dereference freed memory.

- `UNINIT` An intra-procedural checker that detects the use of uninitialized variables; something that may lead to unpredictable results. This includes also dynamically allocated memory by use of e.g. `malloc`.

- `OVERRUN_STATIC` An inter-procedural checker that detects overruns of constant-sized, stack-allocated arrays. Overruns on the stack a particularly troublesome since it may be used by an intruder to change the return address of a stack frame.

- `OVERRUN_DYNAMIC` This checker is similar to the previous one but detects overruns of dynamically allocated heap buffers.

- `SIZECHECK` Because of C's liberal treatment of pointers it is possible to point to a piece of memory which is smaller than the size of the type that the variable should be pointing to. The checker attempts to detect such cases which may lead to subsequent overruns.

- `STACK_USE` The checker detects stack overruns or, to be more precise, when the stack exceeds a certain specified threshold. The checker does not account of compiler optimizations such as tail recursion optimization. The checker is not activated by default.

- `UNUSED_VALUE` Detects assignments of pointer values never used.

- `DEADCODE` Detects code which is never executed. In the best case the code is just unnecessary, but it may also be because of a logical error e.g. in a conditional.

- `FORWARD_NULL` This checker detects cases where a pointer is first checked if null and is then subsequently dereferenced.

- `REVERSE_INULL` Detects null checks after dereferencing has taken place. This may result from unnecessary checks but it may also be a logical error where dereferencing may involve a null pointer.
- `NEGATIVE_RETURNS` This checker detects misuse of negative integers and functions that may return negative integers e.g. to signal an error; potential problems involve e.g. accessing arrays or assignment to an unsigned integer.
- `REVERSE_NEGATIVE` Detects negative checks after potentially dangerous use. Either the check is unnecessary or the use is dangerous.
- `RETURN_LOCAL` Detects functions returning pointer to local stack variable. Since the data is on the stack its value is undetermined once the function returns. Using the data is likely to lead to a crash or serious memory corruption.
- `NULL_RETURNS` Detects unchecked dereferences of functions that may return null return values.
- `CHECKED_RETURN` Detects inconsistent use of return values. The checker performs a statistical analysis in an attempt to find diverging treatment of return values.
- `BAD_COMPARE` A highly specialized checker that looks for cases where pointers to functions are compared to 0.

5.2 Coverity C++ checkers

In addition to the C checkers, the following C++ checkers are provided:

- `BAD_OVERRIDE` Detects errors in overriding virtual functions.
- `CTOR_DTOR_LEAK` The checker detects leaks due to missing destructors to constructors, similar to `RESOURCE_LEAK`.
- `DELETE_ARRAY` Detects bad deletion of arrays. In C++ structures are deallocated with `delete` while dynamic arrays are deallocated with `delete[]`.
- `INVALIDATE_ITERATOR` Detects STL iterators that are either invalid or past-the-end.
- `PASS_BY_VALUE` Detects function parameters that are too big (more than 128 bytes). Large parameters are normally passed by reference rather than by value.
- `UNCAUGHT_EXCEPT` Detects uncaught exceptions. This checker is currently in beta stage.

5.3 Concurrency checkers

There is restricted support for analyzing concurrent programs, dealing mainly with locking order of synchronization primitives:

- `LOCK` Detects double locks, and missing lock releases.
- `ORDER_REVERSAL` Detects cases of incorrect lock ordering, potentially leading to deadlock.
- `SLEEP` Detects blocking functions where locks may be held too long.

5.4 Security checkers

As of version 2.3.0 Coverity Prevent includes checkers specifically directed towards security vulnerabilities.

- `OPEN_ARGS` Warns of incorrect use of the system call to `open` with the flag `O_CREAT`. If the file does not exist and permissions are not also specified the file my have unsafe permissions.
- `SECURE_CODING` Warns of possibly dangerous use of certain system calls such as `gets`, `strcpy`, etc.
- `BUFFER_SIZE` Warns of potential memory corruption due to incorrect size argument to string or buffer copy operation.
- `STRING_OVERFLOW` Warns of potential string overflow in system calls involving writing to strings. Failure to check that the destination is sufficiently large may lead to serious memory corruption.
- `CHROOT` Warns of incorrect use of the `chroot` system call. The call can be used to re-root a program to a directory so that it cannot name files outside of that directory, thus providing a sandbox mechanism. However, the call may be unsafe if the directory specified is outside of the current working directory.
- `TOCTOU` (Time-Of-Check-Time-Of-Use) Detects a set of race conditions where a resource is first checked by one system call, and then later used by another system call (e.g. `access` and `chmod`). An intruder may be able to modify the resource inbetween the check and the use.
- `SECURE_TEMP` Warns of the use of insecure functions for creating temporary files/file names, e.g. where it is easy to guess the names of temporary files.

There are five security checkers dealing with so-called *tainted data*. Most programs rely on data being read from files, sockets, command line arguments, environment variables etc. Failure to check the validity of such data before use, may lead to serious security holes. The checkers track entry points of such data ("sources"), where it is used ("sinks"), intermediate operations on such data ("transitivity") and validity checks ("sanitizers").

- `STRING_NULL` Detects entry and subsequent use of non null-terminated strings.

- `STRING_SIZE` Detects the use of strings whose size has not been checked.
- `TAINTED_SCALAR` Detects the use of scalars without proper checking of bounds, e.g. in indexing of arrays, loop bounds, or in specific function calls.
- `TAINTED_STRING` Warns of the use of strings without first checking the validity of them, e.g. in calls to `system`, `exec`, or if used in formatted input/output.
- `USER_POINTER` Detects the use of userland pointers.

5.5 Extensions

Coverity Extend [8] builds on ideas from the meta language Metal [3], although using a significantly different syntax. Metal was a meta language for specifying new checkers using a dedicated textual notation borrowing concepts from finite state automata; however, Coverity Extend essentially uses C++ syntax with some additional syntactic sugar. The tool provides support for parsing and building abstract syntax trees. There is support for automatically traversing the abstract syntax trees including a hook and a pattern matching mechanism enabling the user to take appropriate actions on spotting specific patterns. There is a also a notion of an abstract store facilitating assigning abstract values to variables and expressions. The automatic traversal of the syntax trees facilitates pruning infeasible branches and a fixpoint mechanism to stop traversal of loops. There is also an explicit backtracking feature that makes it possible for the user to prune infeasible branches not spotted by the underlying framework.

6 Klocwork K7

Klocwork K7 comes in two variants—the defect+security suite for finding (1) code defects, (2) security vulnerabilities and (3) interface problems, and the development suite which, in addition, provides support for code metrics and customized analyses. Klocwork K7 supports analysis of C, C++ and Java. We have studied version 7.1 of the tools.

The philosophy of Klocwork is similar to that of Coverity. The checkers address very similar defects. However, experiments indicate that the tools from Coverity and Klocwork tend to find different defects. Klocwork, like Coverity, also provides a range of security checkers with similar functionality as those of Coverity. In contrast to Coverity Prevent it is hard to find public documentation on the underlying technology used by Klocwork. However, we conjecture that Klocwork builds largely on similar principles; the analysis is not sound but it does some sort of inter-procedural analysis facilitating pruning of some impossible paths and tracks some, but far from all, aliasing information. It can record simple relationships between variables and probably tracks possible values of variables by intervals. Finally it uses some heuristic techniques to rank error reports. Like Coverity Prevent there is a repository for storing results of previous analyses, and what error reports to visualize.

In response to Coverity's mission to find bugs in open-source software Klocwork applied their tool to some of the open-source projects and claim that they found many defects not found by Coverity's tool.

As already mentioned Klocwork K7 is not limited to finding defects. It also provides support for refactoring and code metrics etc.

6.1 Code defects

There are checkers for C and C++ and a separate set of checkers for Java and we limit attention to the former. There are a large number of very specific checkers for finding code defects, which are categorized in a completely different way than by Coverity, but the Klocwork checkers can roughly be categorized as follows:

- Incorrect freeing of memory: This indicates a possible memory leak. Checkers includes e.g. the use of mismatched allocation/deallocation constructs (`malloc` should be matched by `free`, `new` by `delete` and `new[]` by `delete[]`). The category also includes freeing of non-heap memory or unallocated memory, or freeing of memory after pointer arithmetics. There is also a checker for detecting functions which free memory in some, but not in all, execution paths.
- Suspicious return value: A range of checkers that detect suspicious use (or non-use) of return values, e.g. a function returning a pointer to local data, or some branch of a function that does not return a value. There are also checkers for functions declared as void that return values and non-void functions that do not return explicit value. There is also a checker for calls to side-effect free functions where the return value is not used.
- Null pointer dereference: There are a large number of checkers for dereferencing of null pointers in various situations.
- Use memory after free: There are checkers to detect cases where freed memory may be dereferenced leading to unpredictable results.
- Uninitialized data: Detects cases where data is implicitly cast to a smaller data type which may lead to loss of precision.
- Unreachable code or unused data: There are several checkers for detecting unreachable program constructs and unused data. This includes e.g. unreachable `break`- and `return`-statements, unused functions, local variables and labels, unused variables and parameters. There are checkers for statement without effects and values assigned to variables which are not subsequently used. The presence of unreachable code or unused data does not have to be a (serious) problem, but it may be due to serious logical errors or misspellings.
- Incorrect allocation size: Checks for defects due to dynamic memory allocation of data of insufficient size. This is likely to lead to subsequent buffer overruns. There is also a checker for invalid pointer arithmetics that may cause similar problems.
- Use of container iterator: Flags suspicious use of iterators for STL-containers.
- Memory leak: There are checkers that detect when previously allocated memory is not freed.

- Buffer overruns: Flags array bounds violations due to non-null terminated strings. See also security checkers.

Some checkers come in two flavors—"might" and "must" checkers—signaling problems which *may* occur and problems which *must* occur. For instance there is a FNH.MIGHT and a FNH.MUST checker for detecting freeing of non-heap memory. This corresponds roughly to orange and red code in PolySpace.

Klocwork also detects some cases of syntactically correct code which is sometimes considered inappropriate or dangerous and may be the result of misspellings:

- Assignment in conditions: Flags the use of assignment in conditions which is allowed and frequently used in C-programs, but may be due to misspellings (writing assignment instead of equality).
- Misplaced semicolon: Flags suspicious use of semicolon in syntactically correct code.

6.2 Security vulnerabilities

Like Coverity Prevent there a range of checkers dedicated to finding security vulnerabilities in C/C++ and Java code. Only C/C++ checkers are discussed here. There are approximately 50 checkers for C/C++. Some vulnerabilities are generic (e.g. buffer overruns and tainted data) but many of the checkers are very specific; checking the use of specific system calls, known to be potential sources of security holes. Klocwork provide checkers for security vulnerabilities such as:[9]

- Buffer overruns: Reading or writing data out of bounds is a very common type of attack in languages such as C and C++. This includes use of unsafe format string for formated printing and scanning. Similar (or even worse effects) may arise because of non-constant format strings or the use of format strings injected from outside (see tainted data). There are checks for use of non-null terminated strings and dangerous string copy operations.
- Command injection: System calls such as `system`, versions of `exec` etc may be used to spawn dangerous processes, in particular in combination with untrusted input strings.
- Use of tainted (untrusted) data: A serious security hole in software is data provided from the outside, such as user input, environment variables, sockets, files, command line arguments etc. If such data is not verified or sanitized before being used it can result in malicious attacks; in particular in combination with buffer overrun attempts or command injection. A well-known example is use of the standard function `gets`.
- Access problems: A number of system functions require special system privileges (admin/super-user etc) to be performed, e.g. allocation of ports and access to certain entries in the Windows registry. Eliminating the use of such functions entirely from the code is not possible, but it may be

[9]There is an even greater number of checkers for Java, which are not covered here.

possible to perform the functions e.g. during start-up and then revert to lower privileges and/or to perform the function externally e.g. through a service instead. There are several checkers that flag the use of operating system functions requiring special privileges. Another problem checked for is ignoring return values from system functions to lower priority of the current process. Finally there are checkers to detect system functions that may leak critical data through the use of absolute path names.

- Time-of-check/time-of-use (TOCTOU): There are a number of system calls for checking the status of resources (e.g. files) before using/modifying them, e.g. `access`, `chown`, `chgrp` and `chmod`. It is of course vital that the resource is not replaced/modified inbetween by an attacker. There are checkers that signal such critical checks/uses.

- File access: Files may be subject also to other vulnerabilities, e.g. the use of insufficient (e.g. relative) file names of dynamic load libraries. Problems may also arise because of improper sequencing of calls to resources, and improper use of temporary file names resulting from unsafe use of the `CreateFile` system call. Another problem checked for is poor naming of files for critical data or executable code (dynamic link libraries etc).

- Poor encryption: There are checks to warn of use of old, i.e. unsafe, algorithms for encryption of data.

- Unintended copy: It is typically not clear what happens to memory after a process stops executing. Sensitive data or even code may be exposed to other processes that reuse the same memory unless the memory is explicitly overwritten. This concerns in particular system calls like `realloc`, `fork` and `vfork`.

In addition to the defects and security checkers Klocwork also provides a support for checking interface problems and for code metrics:

- The interface problems addressed by Klocwork are essentially syntactic checks involving dependencies but no semantic analysis. The checks include issues such as finding potential cycles in `#include`-directives, multiple declarations of an object in different files, unintentional reuse of names variable or function names, declarations which are not used, transitive use of `#include`-directives etc.

- The metrics supported by Klocwork are also syntactic in nature. There are metrics on file-level as well as class- and function-/method-level. Metrics range from simple ones like the number of lines of code, number of declarations, operands, comments, includes etc., accesses to global variables number of conditionals and nesting of control structures and so forth. But there are also more sophisticated yet well-established measures like McCabe cyclomatic complexity [19] and Halstead program volume metrics [16].

Interface problems as well as metrics are of course very relevant in development of correct software but is an orthogonal issue to static analysis (at least as we use the term).

7 Programming guidelines

Many software errors can be avoided by limiting the freedom of designers and programmers. The standard example is the use of gotos, available in most programming languages and useful in certain situations, but usually considered harmful. Similarly many runtime errors can be avoided by not using e.g. pointers and dynamic allocation of memory unless absolutely necessary.

Restricting the use of certain language constructs or limiting the design space can also be beneficial for static analysis; both with regard to precision as well as analysis time. For instance, restricting the use of pointers, global variables, dynamic memory allocation and recursion will improve precision and reduce the analysis time substantially.

The Motor Industry Software Reliability Association (MISRA) has issued a set of guideline for the use of C language code in vehicular embedded systems ([20] and [21]). The guidelines consist of 100+ rules intended to reduce the number of errors in vehicle based software by considering a subset of the C language. There are "required" rules, which must be adhered to, and "advisory" rules which should normally be followed. In the first version of the guidelines there were 93 required rules and 34 advisory rules. The rules concern what version of C to use and how to embed code written in other languages. There are rules on which character sets to use, how to write comments, identifiers and constants. Some of the more significant rules concern

- what *types* to use and how to use them;
- the naming and scope of *declarations and definitions*;
- the *initialization* of variables before use;
- how to write expressions involving certain *operators*;
- how to do explicit and implicit type *conversions*;
- how to write *expressions*;
- which *control flow* statements that should be used and how to use them;
- how to declare and define *functions* and various constraints on them, such as return values and formal parameters (e.g. disallowing the use of recursive functions and functions with a variable number of arguments);
- the use of *pre-processing directives* and macros;
- the use of *pointers and arrays* (e.g. avoiding pointer arithmetics and dereferencing of null-pointers);
- the use of *structures and unions* where, in particular, the latter is a source of hard-to-find errors;
- the use of *standard libraries.* In particular, ensuring that standard functions are called in the way they are intended to. There are also rules disallowing the use of certain standard functions and libraries (e.g. dynamic allocation of memory).

The character of the rules varies substantially. Some rules are quite loose like the advisory rule stating that provisions should be made for appropriate runtime checking, since C does not provide much builtin support. This is an obvious case where static analysis can be of great help. However, most of the rules are essentially syntactic and can be easily checked with simple pattern matching techniques like the rule which bans the use of functions with a variable number of arguments, or the use of recursion. There are also specific rules which are not easily checked by humans; e.g. requiring that values of expressions should be the same under any evaluation order admitted by the C standard and the rule which requires that there must be no dead code.

Programming guidelines, such as the MISRA rules, and static analysis can benefit from each other. On the one hand static analysis can be used to verify certain non-trivial rules/recommendation suggested by the coding rules. For instance, PolySpace provides a separate tool to verify MISRA compliance. On the other hand, doing static analysis can be greatly simplified by adhering to a more disciplined programming style; in fact, PolySpace recommends using a subset of the MISRA rules [24] to reduce the amount of orange marked code (code that may lead to errors).

8 A comparison of the tools

Shallow static analysis tools based on pattern matching such as FlexeLint [14] have been around since the late 1980s. Lately several sophisticated industrial-strength static analysis tools have emerged. In this report we study tools from three of the main providers—PolySpace, Coverity and Klocwork. There are several other static analysis tools around, including PREfix/PREfast from Microsoft [2], Astree [6], which are not as widely available. An interesting newcomer is CodeSonar from Grammatech, founded by Tim Teitelbaum and Tom Reps, which is similar in style and ambition level to Coverity Prevent and Klocwork K7. Even if we focus here on tools intended for global and "deep" (=semantic) analysis of code, more lightweight tools like FlexeLint may still be useful in more interactive use and for local analysis.

There are also dynamic tools that aim for discovering some of the kinds of defects as the static analysis tools do. For example Insure++ [23] and Rational Purify [25] detect memory corruption errors.

A rough summary of major features of the three systems studied here can be found in Table 1. Such a table is by necessity incomplete and simplistic and in the following sub-section we elaborate on the most important differences and similarities.

8.1 Functionality provided

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 (i.e. false alarms). 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 (defects which are not reported). It is almost impossible to quantify the rate of false negatives/positives; 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. However, the rate of false positives obviously depends on the quality of the code. The rate of false negatives is even more difficult to estimate, since it depends even more on the quality of the code. (Obviously there will be *no* false negatives if the code is already free of defects.) According to Coverity the rate of defect reports is typically around 1 defect per 1-2KLoC.

PolySpace, on the other hand, does in general produce a great deal of orange code. *If* orange code is considered a potential defect then PolySpace Verifier produces a high rate of false positives. However, this is a somewhat unfair comparison; while Coverity and Klocwork does not even give the developer the opportunity to inspect all potential defects, PolySpace provides that opportunity and provides instead a methodology in which the developer can systematically inspect orange code and classify it either as correct or faulty. In other words, Coverity and Klocwork are likely to "find some bugs", but provide no guarantees—the rest of the code may contain defects which are not even reported by the tool. PolySpace on the other hand can provide guarantees—if all code is green (or grey) it is *known* not to contain any bugs (wrt the properties checked for, that is). On the other hand it may be hard to eliminate all orange code.

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 relationships between variables are approximated by convex polyhedra and all approximations are sound—that is, no execution sequences are forgotten but some impossible execution paths may be analyzed due to the approximations made. Coverity Prevent and Klocwork K7 accounts 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 (fully) 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 has the more accurate analyses.

Another consequence of the restricted tracking of arithmetic values of variables in Coverity Prevent and Klocwork K7 is that the products are not suitable for detecting arithmetic defects, such as over- and underflows or illegal operations like division by zero. The products do not even provide arithmetic checkers. PolySpace on the other hand does provide several arithmetic checkers, setting it apart from the others.

While PolySpace is the only tool that provides arithmetic checkers, it is also the only one among the three which does not provide any checkers for resource leaks; in particular there is no support for discovering defects in dynamic management (allocation and deallocation) of memory. As a consequence there are also no checkers e.g. for use-after-free. This lack can perhaps be explained by PolySpace's focus on the embedded systems market, involving safety or life critical applications where no dynamic allocation of memory is possible or allowed.

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 data. The focus on networked application also explains the support for analyzing resource leaks since dynamic management of resources such as sockets, streams and memory is an integral part of most networked applications.

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 analysis. 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 takes significantly less time than analyzing the whole system from scratch. With the other tools analysis of the whole system has to be redone.

All the tools provide the possibility to analyze a single file. However such an analysis will be much more shallow than analyzing a whole system where complete paths of execution can be analyzed.

Both Klocwork and Coverity provide means for writing user defined checkers and integrating them with the analysis tools. However, the APIs are non-trivial and writing new checkers is both cumbersome and error prone. There are no explicit guidelines for writing correct checkers and no documented support for manipulation of abstract values (e.g. interval constraints). There is also no support for reusing the results of other checkers. Termination of the checker is another issue which may be problematic for users not familiar with the mathematical foundations of static analysis.

All three tools support analysis of the C programming language and C++. When this study was initiated only Klocwork supported analysis of Java but Coverity has recently announced a new version of Prevent that also supports analysis of Java. Only PolySpace supports analysis of Ada. Klocwork is the only provider which claims to be able to handle mixed language applications (C/C++/Java).

The downside of PolySpace's sophisticated mechanisms for tracking variable values is that the tool cannot deal automatically with very large code bases without manual partitioning of the code. While Coverity Prevent and Klocwork K7 is able to analyze millions of lines of code off-the-shelf and overnight, PolySpace seems to reach the complexity barrier already at around 50KLoC with the default settings. On the other hand PolySpace advocates analyzing code in a modular fashion. Analysis time is typically not linear in the number of lines of code—analyzing 10 modules of 100KLoC is typically orders of magnitude faster than analyzing a single program consisting of 1,000KLoC. However this typically involves human intervention and well-defined interfaces (which obviously may be beneficial for other quality reasons...)

Coverity and Klocwork seem to have given priority to largely similar concerns, with focus on general purpose software, in particular communication centric software in Internet environments. They both offer checkers for security vulnerabilities and checkers for management of resource leaks which are missing in PolySpace Verifier. PolySpace has a sophisticated machinery for tracking re-

Table 1: Summary of features of Coverity Prevent, Klocwork K7 and PolySpace Verifier

Functionality	Coverity	KlocWork	PolySpace
Coding style	No	Some	No
Buffer overrun	Yes	Yes	Yes
Arithmetic over/underflow	No	No	Yes
Illegal shift operations	No	No	Yes
Undefined arithmetic operations	No	No	Yes
Bad return value	Yes	Yes	Yes
Memory/resource leaks	Yes	Yes	No
Use after free	Yes	Yes	No
Uninitialized variables	Yes	Yes	Yes
Size mismatch	Yes	Yes	Yes
Stack use	Yes	No	No
Dead code/data	Yes	Yes	Yes (code)
Null pointer dereference	Yes	Yes	Yes
STL checkers	Some	Some	No?
Uncaught exceptions	Beta (C++)	No	No
User assertions	No	No	Yes
Function pointers	No	No	Yes
Nontermination	No	No	Yes
Concurrency	Lock order	No	Shared data
Tainted data	Yes	Yes	No
Time-of-check Time-of-use	Yes	Yes	No
Unsafe system calls	Yes	Yes	No
MISRA support	No	No	Yes
Extensible	Yes	Some	No
Incremental analysis	Yes	No	No
False positives	Few	Few	Many
False negatives	Yes	Yes	No
Software metrics	No	Yes	No
Language support	C/C++	C/C++/Java	C/C++/Ada

lationships between variables, in particular integer and floating point variables, which makes it useful for arithmetic embedded applications such DSPs (Digital Signal Processors) and dataflow architectures typically used in avionics and automotive systems with hard limitations on memory (typically no dynamic allocation of memory) and the range of numerical data (over-/underflows may be catastrophic).

On the more exotic side Coverity provides a checker for *stack use*. It is unclear how useful this is since there is no uniform way of allocating stack memory in different compilers. Klocwork is claimed to provide similar functionality but in a separate tool. PolySpace set themselves aside from the others by providing checkers for non termination, both of functions and loops. Again it is unclear how useful such checkers are considering the great amount of research done on *dedicated* algorithms for proving termination of programs. Coverity has a checker for uncaught exceptions in C++ which was still a beta release. PolySpace provides a useful feature in their support for writing general assertions in the code. Such assertions are useful both for writing stubs and may also be used for proving partial correctness also of functional properties.

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 in 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 (via the data dictionary).

Both Coverity and Klocwork have developed lightweight versions of their tools aimed for frequent analysis during development. These have been integrated with Eclipse IDEs. However the defect databases for Coverity and Klocwork have not been integrated into Eclipse IDEs or TPTP. PolySpace has integrated with the Rhapsody UML tool to provide a UML static analysis tool. It analyzes generated code and links back references to the UML model to point out where defects have been detected. Besides that PolySpace has its general C++ level advantages with a sound analysis (no false negatives) and presumably problems with analyzing large code bases (larger than 50-100 KLoC)—a restriction which should be more severe in the UML situation compared to hand-coded C++.

8.2 Experiences at Ericsson

A number of independent evaluations of static analysis tools have been performed by development groups at Ericsson. Several of the evaluations included tools studied in this report; Coverity has been evaluated by several groups. Klocwork has also been subject to evaluations but not quite as many. There has been one attempt to use PolySpace for one of the smallest applications, but the evaluation was not successful and it is not clearly known why. It would have been very interesting to compare results from PolySpace, which is sound, to results from Klocwork and Coverity. Perhaps that would give a hint on the false negative rate in Klocwork and Coverity.

Some general experiences from use of Coverity and Klocwork were:

- The tools are easy to install and get going. The development environment is easy to adapt and no incompatible changes in tools or processes are needed.

- The tools are able to find bugs that would hardly be found otherwise.

- It is possible to analyze even large applications with several million lines of code and the time it takes is comparable to build time.
- Even for large applications the number of false positives is not a major problem.
- Several users had expected the tools to find more errors and errors that were more severe.
- On the other hand, several users were surprised that the tools found several bugs even in applications that had been tested for a long time. Perhaps there is a difference in what users find reasonable to expect from these tools. There might also be large differences in what different users classify as a false positive, a bug and a severe bug.
- It is acceptable to use tools with a high false positive rate (FlexeLint) if the tool is introduced in the beginning of development and then used continuously.
- It is unacceptable to use tools with a high false positive rate if the product is large and the tool is introduced late in the development.
- Many of the defects found could not cause a crash in the system as it was defined and used at the moment. However if the system would be only slightly changed or the usage was changed the problem could happen and cause a serious crash. Therefore these problems should be fixed anyway.
- Even if the tools look for the same categories of defects, for instance memory leaks, addressing out of array bounds etc, the defects found in a given category by one tool can be quite different from those found by another tool.
- Handling of third party libraries can make a big difference to analysis results. Declarations for commercial libraries that come with the analysis tool can make the analysis of own code more precise. If source for the library is available defects in the library can be uncovered, which may be as important to the quality of the whole application as the own code.
- There are several aspects of the tools that are important when making a tool selection that has not been a part of the comparison in this paper; such as pricing, ease of use, integration in IDEs, other functionality, interactiveness etc.

Below follows some more specific results from some of the evaluations. We do not publish exact numbers of code sizes and found bugs etc for confidentiality reasons since some of the applications are commercial products in use.

Evaluation 1 - Coverity and FlexeLint: An application that had been thoroughly tested, both by manually designed tests and systematic tests that were generated from descriptions.

FlexeLint was applied and produced 1,200,000 defect reports. The defects could be reduced to about 1,000 with a great deal of analysis and following filtering work. These then had to be manually analyzed.

Coverity was applied to the same piece of code and found about 40 defects; there were very few false positives and some real bugs. The users appreciated the low false positive rate and thought that the defects that were found would hardly have been found by regular testing.

The users had expected Coverity to find more defects. It was believed that there should be more bugs to be found by static analysis techniques. It was not known if this was the price paid for the low false positive rate or if the analyzed application actually contained only a few defects.

The users also expected Coverity to find more severe defects. Many of the findings were not really defects, but code that simply should be removed, such as declarations of variables that were never used. Other defects highlighted situations that could not really happen since the code was used in a restricted way—which was not known to the analysis tool.

Evaluation 2 - Coverity: A large application was analyzed with Coverity. Part of the code had been previously analyzed with FlexeLint. The application had been extensively tested.

Coverity was easy both to install and use, and no modifications to existing development environment was needed. The error reports from the analysis were classified as follows

- 55 per cent were no real defects but only bad style,
- 2 per cent were false positives,
- 38 per cent were considered real bugs, 1 per cent were considered severe.

The users appreciated that a fair number of defects were found although the code had been thoroughly tested previously.

Evaluation 3 - Coverity and Klocwork: An old version of an application that was known to have some memory leaks was analyzed using Coverity and Klocwork.

In total Klocwork reported 32 defects including 10 false positives and Coverity reported 16 defects including 1 false positive. Only three defects were common to both tools! Hence Klocwork found more defects, but also had a larger false positive rate. Although the tools looked for the same kind of defects the ones actually found were largely specific to each tool. This suggests that each of the tools fails to detect many defects.

Looking at only the memory leaks the results were similar. Klocwork reported 12 defects of which 8 were false, totaling 4 real defects and Coverity reported 7 defects all of which were true defects. None of the tools found any of the known memory leaks.

Evaluation 4 - Coverity and Klocwork Two old and released C++ products were analyzed using Coverity and Klocwork. The products came with trouble reports that had been generated during previous tests. One purpose was to compare how many faults each of the tools would find. Another purpose was to get an estimation of how many of the faults that had been discovered in testing could be found by the static analysis tools.

Coverity found significantly more faults and also had significantly less false positives than Klocwork. One of the major reasons for this was the handling of third party libraries. Coverity analyzed the existing source code for the libraries and found many faults in third party code! Klocwork did not analyze this code and hence did not find any of these faults. Besides that the analysis of the libraries that Coverity did resulted in fewer false positives in the application code since it could be derived that certain situations could not occur.

The time of analyses was about the same as build time for both tools. That is good enough for overnight batch runs but not for daily, interactive use during development.

Both tools lacked integration with CM tool Clearcase, the source code had to be copied into the repository of the analysis tools. There was no way to do inspection of analysis results from an IDE, but the reviews had to be done in the GUI of the analysis tools.

Coverity was preferred by the C++ developers. It had incremental analysis that would save time and it could easily analyze and report on single components.

Although the main part of the evaluation was on old code some studies were done on programs during the development. The development code had more warnings and most of them were real faults; most of these were believed to have been found during function test. It had been anticipated that more faults would be found in low level components, but these components proved to be stable and only a few defects were discovered. More faults were however found in high level components with more frequent changes.

Evaluation 5 - Coverity, Klocwork and CodePro A Java product with known bugs was analyzed. A beta version of Coverity Prevent with Java analysis capabilities was used.

None of the known bugs were found by the tools. Coverity found more real faults and had far less false positives than Klocwork. For Coverity one third of the warnings were real bugs.

Klocwork generated many warnings—7 times the number of warnings that Coverity did. The missing analysis of the third party library seemed to be the major reason. However, Klocwork does a ranking of the potential defects and when only the four most severe levels of warnings were considered the results were much better—there were few false positives.

CodePro Analytix (developed and marketed by Instantiations) is a tool aimed for analysis during development. It is integrated into the Eclipse IDE and the results of an analysis cannot be persistently saved, but only exist during the development session with the IDE. The analysis is not as deep as that of Coverity or Clockwork, but is faster and can easily be done interactively during development. The tool generates a great deal of false positives, but these can be kept at a tolerable level by choosing an appropriate set of analysis rules. No detailed analysis was done of the number of faults and if they were real faults or not.

In this evaluation there was a large difference in the number of warnings generated, Coverity 92 warnings, Klocwork 658 warnings (in the top four severities 19), CodePro 8,000 warnings (with all rules activated).

9 Conclusions

Static analysis tools for detection of runtime defects and security vulnerabilities can roughly be categorized as follows

- **String and pattern matching approaches**: Tools in this category rely mainly on syntactic pattern matching techniques; the analysis is typically path- and context-insensitive. Analyses are therefore shallow, taking little account of semantic information except user annotations, if present. Tools typically generate large volumes of false positives as well as false negatives. Tools (often derivatives of the lint program) have been around for many years, e.g. FlexeLint, PC-Lint and Splint. Since the analysis is shallow it is possible to analyze very large programs, but due to the high rate of false positives an overwhelming amount of post-processing may be needed. These tools are in our opinion more useful for providing almost immediate feedback in interactive use and in combination with user annotations.

- **Unsound dataflow analyses**: This category of tools which have emerged recently rely on semantic information; not just syntactic pattern matching. Tools are typically path- and context-sensitive but the precision is limited so in practice the tools have to analyze also many impossible paths or make more-or-less justified guesses what paths are (im-)possible. This implies that analyses are unsound. Aliasing analysis is usually only partly implemented, and tracking of possible variable values is limited; global variables are sometimes not tracked at all. A main objective of the tools, represented e.g. by Coverity Prevent and Klocwork K7, is to reduce the number of false positives and to allow for analysis of very large code bases. The low rate of false positives (typically 20–30 per cent in Coverity Prevent) is achieved by a combination of a unsound analysis and filtering of the error reports. The downside is the presence of false negatives. It is impossible to quantify the rate since it depends very much on the quality of the code, but in several evaluations Coverity and Klocwork find largely disjoint sets of defects. This category of tools provide no guarantees—the error reports may or may not be real defects (it has to be checked by the user), and code which is not complained upon may still be incorrect. However, the tools will typically find *some* bugs which are hard to find by other techniques.

- **Sound dataflow analyses**: Tools in this category are typically path- and context-sensitive. However, imprecision may lead to analysis of some infeasible paths. They typically have sophisticated mechanisms to track aliasing and relationships between variables including global ones. The main difficulty is to avoid excessive generation of false positives by being as precise as possible while analysis time scales. The only commercial system that we are aware of which has taken this route is PolySpace Verifier/Desktop. The great advantage of a sound analysis is that it gives some guarantees: if the tool does not complain about some piece of code (the code is green in PolySpace jargon) then that piece of code must be free of the defects checked for.

There is a forth category of tools which we have not discussed here—namely tools based on *model checking* techniques [4]. Model checking, much like static analysis, facilitates traversal and analysis of all reachable states of a system (e.g. a piece of software), but in addition to allowing for checking of run-time properties, model checking facilitates checking of functional properties (e.g. safety properties) and also so-called temporal properties (liveness, fairness and real-time properties). There are commercial tools for model checking hardware systems, but because of efficiency issues there are not yet serious commercial competitors for software model checking.

It is clear that the efficiency and quality of static analysis tools have reached a maturity level were static analysis is not only becoming a viable complement to software testing but is in fact a required step in the quality assurance of certain types of applications. There are many examples where static analysis has discovered serious defects and vulnerabilities that would have been very hard to find using ordinary testing.

However, there is still substantial room for improvement. Sound static analysis approaches, such as that of PolySpace, still cannot deal well with very large code bases without manual intervention and they produce a large number of false positives even with very advanced approximation techniques to avoid loss of precision. Unsound tools, on the other hand, such as those from Coverity and Klocwork do scale well, albeit not to the level of interactive use. The number of false positives is surprisingly low and clearly at an acceptable level. The price to be paid is that they are not sound, and hence, provide no guarantees: they may (and most likely will) find some bugs, possibly serious ones. But the absence of error reports from such a tool only means that the *tool* was unable to find any potential defects. As witnessed in the evaluations different unsound tools tend to find largely disjoint defects and are also known not to find known defects. Hence, analyzed code is likely to contain dormant bugs which can only be found by a sound analysis.

Most of the evaluations of the tools have been carried out on more or less mature code. We believe that to fully ripe the benefits of the tools they should not be used only at the end of the development process (after testing and/or after using e.g. FlexeLint), but should probably be used throughout the development process. However, the requirements on the tools are quite different at an early stage compared to at acceptance testing. Some vendors "solve" the problem by providing different tools, such as PolySpace Desktop and PolySpace Verifier. However, we rather advocate giving the user means of fine-tuning the behavior of the analysis engine. A user of the tools today has very limited control over precision and the rate of false positives and false negatives—there are typically a few levels of precision available, but the user is basically in the hands of the tools. It would be desirable for the user to have better control over precision of the analyses. There should for example be a mechanism to fine-tune the effort spent on deriving value ranges of variables and the effort spent on aliasing analysis. For some users and in certain situations it would be acceptable to spend five times more analysis time in order to detect more defects. Before an important release it could be desirable to spend much more time than on the day to day analysis runs. In code under development one can possibly live with some false negatives and non-optimal precision as long as the tool "finds some bugs". As the code develops one can improve the precision and decrease the rate of false positives and negatives; in particular in an incremental tool such as Coverity Prevent. Similarly it would be desirable to have some mechanism to control the aggressiveness of filtering of error reports.

Finally a word of caution: Static analysis can be used to find runtime error but does not in general address the functional correctness of the program. One should be aware of the risk that an over-zealous user, in the excessive eagerness to get rid of warnings of potential runtime errors, may introduce functional incorrectness instead. The following piece of code may lead to a runtime error, unless there is a check in `initialize` that the buffer, allocated in the previous line, is not null.

```
my_buf *buf;

// Some code
buf = (my_buf *)malloc(sizeof(my_buf));
initialize(buf);
// Some more code
```

It is of course easy to get rid of a warning from a static analysis tool simply by introducing a conditional

```
my_buf *buf;

// Some code
buf = (my_buf *)malloc(sizeof(my_buf));
if(buf != NULL) {
  initialize(buf);
}
// Some more code
```

However, if the conditional is not also equipped with an else-branch it is likely that we either have a runtime error somewhere else when the buffer is used, or—if all such warnings have been suppressed—that the program does not do what it is supposed to do in the event that malloc is unsuccessful in allocating memory.

Acknowledgements: The authors are grateful to Dejan Baca, Per Flodin, Fredrik Hansson, Per Karlsson, Leif Linderstam, and Johan Ringström from Ericsson for providing tool evaluation information.

References

[1] T. Ball and S. Rajamani. The SLAM Project: Debugging System Software via Static Analysis. *ACM SIGPLAN Notices*, 37(1):1–3, 2002.

[2] W. Bush, J. Pincus, and D. Sielaff. A Static Analyzer For Finding Dynamic Programming Errors. *Software, Practice and Experience*, 30(7):775–802, 2000.

[3] B. Chelf, D. Engler, and S. Hallem. How to Write System-specific, Static Checkers in Metal. In *PASTE '02: Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 51–60, New York, NY, USA, 2002. ACM Press.

[4] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, Cambridge, MA, USA, 1999.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs by Construction Or Approximation of Fixpoints. In *Conf. Record of the Fourth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[6] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE Analyser. In M. Sagiv, editor, *Proceedings of the European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30, Edinburgh, Scotland, 2005. Springer.

[7] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Conf. Record of the Fifth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[8] Coverity. *Coverity Extend^{TM} User's Manual (2.4)*, 2006.

[9] Coverity. Coverity Prevent$^{\text{TM}}$: Static Source Code Analysis for C and C++. Product information, 2006.

[10] Coverity. *Coverity Prevent^{TM} User's Manual 2.4*, 2006.

[11] Coverity Inc. The Scan Ladder, 2007. `http://scan.coverity.com`.

[12] A. Deutsch. Interprocedural May-Alias Analysis for Pointers: Beyond k-limiting. In *Proc. Programming Language Design and Implementation.* ACM Press, 1994.

[13] D. Engler. http://www.stanford.edu/ engler/.

[14] Gimpel Software. *PC-lint/FlexeLint*, 1999. `http://www.gimpel.com/lintinfo.htm`.

[15] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A System and Language for Building System-specific, Static Analyses. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 69–82, New York, NY, USA, 2002. ACM Press.

[16] M. Halstead. *Elements of Software Science.* Operating, and Programming Systems Series Volume 7. Elsevier, 1977.

[17] J. King. Symbolic Execution and Program Testing. *Comm. ACM*, 19(7):385–394, 1976.

[18] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler. Correlation Exploitation In Error Ranking. In *SIGSOFT FSE*, pages 83–93, 2004.

[19] T. McCabe and C. Butler. Design Complexity Measurement and Testing. *Comm. ACM*, 32(12):1415–1425, 1989.

[20] Motor Industry Software Reliability Association. MISRA-C:1998 Guidelines For the Use of the C Language In Vehicle Based Software. Technical report, MISRA, 1998.

[21] Motor Industry Software Reliability Association. MISRA-C:2004 Guidelines For the Use of the C Language in Critical Systems. Technical report, MISRA, 2004.

[22] J. Palsberg and M. Schwartzbach. Object-Oriented Type Inference. In *Conf Proc Object-Oriented Programming Systems, Languages, And Applications (OOPSLA '91)*, pages 146–161, New York, NY, USA, 1991. ACM Press.

[23] Parasoft. Automating C/C++ Runtime Error Detection With Parasoft Insure++. White paper, 2006.

[24] PolySpace Technologies. *PolySpace for C Documentation*, 2004.

[25] Rational Software. Purify: Fast Detection of Memory Leaks and Access Errors. White paper, 1999.

[26] B. Ryder and F. Tip. Change Impact Analysis For Object-Oriented Programs. In *Proc. of 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis For Software Tools And Engineering (PASTE '01)*, pages 46–53, New York, NY, USA, 2001. ACM Press.

[27] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *ACM POPL*, pages 32–41, 1996.

[28] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proc. of 2001 IEEE Symp. on Security and Privacy (SP'01)*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[29] T. Wagner, V. Maverick, S. Graham, and M. Harrison. Accurate Static Estimators For Program Optimization. In *Proc. of ACM SIGPLAN 1994 Conf. on Programming Language Design And Implementation (PLDI '94)*, pages 85–96, New York, NY, USA, 1994. ACM Press.