Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 266 (2010) 17–32
www.elsevier.com/locate/entcs

Fade to Grey: Tuning Static Program Analysis

Ansgar Fehnker, Ralf Huuck, Sean Seefried and Michael Tapp

National ICT Australia Ltd. (NICTA)
Locked Bag 6016
University of New South Wales
Sydney NSW 1466, Australia [1]

Abstract

Static program analysis complements traditional dynamic testing by discovering generic patterns and relations in source code, which indicate software deficiencies such as memory corruption, unexpected program behavior and memory leaks. Since static program analysis builds on approximations of a program's concrete behavior there is often a trade-off between reporting potential bugs that might be the result of an over-approximation and silently suppressing those defects in that grey area. While this trade-off is less important for small files it has severe implications when facing large software packages, i.e., 1,000,000 LoC and more. In this work we report on experiences with using our static C/C++ analyzer Goanna on such large software systems, motivate why a flexible property specification language is vital, and present a number of decisions that had to be made to select the right checks as well as a sensible reporting strategy. We illustrate our findings by empirical data obtained from regularly analyzing the Firefox source code.

Keywords: Source code analysis, static analysis, C/C++, false positive reduction, case study, Firefox.

1 Introduction

Traditional software testing is an integral part of the software quality assurance process to validate the overall design, to find bugs in the implementation and to increase trust in the correctness of a system. The advantage of traditional testing is that *functional* behavior of the software can be examined for the real implementation on the real hardware. This stands in contrast to, e.g., purely model-based approaches. The disadvantage is that testing explores typically only a limited number of program behaviors. Even if all program paths are explored, only a limited set of inputs can be examined, and significant manual effort is required to find the appropriate test cases. One of the most significant disadvantages is that testing does

[1] National ICT Australia is funded by the Australian Governments Department of Communications, Information Technology and the Arts and the Australian Research Council through Backing Australias Ability and the ICT Research Centre of Excellence programs.

1571-0661 Crown Copyright © 2010 Published by Elsevier B.V. Open access under CC BY-NC-ND license.
doi:10.1016/j.entcs.2010.08.046

not scale well to large code bases. There are typically an overwhelming number of test cases to consider to achieve a satisfactory coverage.

Static program analysis [1,2] can alleviate some of the aforementioned disadvantages. In contrast to traditional testing, static program analysis does not execute the implementation, but analyzes the source code for known dangerous programming constructs, for combinations of those and their causal relationships, and the impact of potentially tainted input. Typical examples in C/C++ are null pointer dereferences, accessing freed memory, memory leaks, or creating exploits through buffer overruns. These types of bugs are only found to the extend in that they affect the functional behavior, and since traditional testing is focussed on checking the functional behavior of the system, finding these types of bugs is more often a welcomed side-effect, rather than intentional. Because static program analysis can pinpoint those software deficiencies directly, and because it is scalable to large code bases and can be run fully automatically, it is a way to complement traditional testing.

Static program analysis cannot always be precise, since it does not execute the real code, but examines syntactic relations within the source code. This means, over-approximations are used to estimate the actual program behavior. This almost inevitably leads to false alarms, i.e., warnings which are spurious and do not correlate to any actual execution. In addition to these false alarms, there are warnings that pinpoint code where the programmer bends the rules of the C/C++ standard, often to achieve efficiency. From the programmers' perspective these warnings are often also considered to be false alarms. The art of static program analysis is to minimize this grey area of potential false alarms of either type. There are three options: Not reporting any bugs that might be the result of over-approximations, adding semantic information to the analysis to make the approximation more precise, or reverting to an under-approximation all together, i.e., only consider definite bugs in the analysis.

In this paper we present a number of results and experiences of developing a static program analyzer from a tool builder's point of view. In particular we look at common software bugs and potential false alarms, which we discovered in large source code bases. False alarms in this work comprise false positives as defined by formal language semantics, as well as unnecessary or superfluous alarms from a tool user's point of view. Based on practical observations we present a number of dimensions to classify properties and bugs, and give detailed explanations why we believe those dimensions to be appropriate and substantiate this by several example deficiencies found in a large existing code base.

Moreover, we suggest practical measures and analysis techniques to improve static program analysis results in general. To support our claims, we implemented some of those measures in our static analyzer *Goanna* and report on the qualitative results we obtained from analyzing the source code of Firefox, which has around $2,500,000$ LoC after preprocessing.

The paper is structured as follows: In the subsequent Section 2 we summarize the different classes of static program analysis techniques and typical approaches

by modern static analysis tools. We briefly describe the underlying technology of our own analyzer Goanna in Section 3. The focus of this work is Section 4 where we discuss the dimensions for classifying bugs, give example deficiencies, as well as measures for improvement. Section 5 presents empirical data based on analyzing Firefox, before Section 6 concludes the paper.

2 Dimensions of Static Program Analysis

Static program analysis is a term that was coined by the compiler community for a set of techniques to investigate program properties without actually executing the program. In recent years those techniques have become popular not only for compiler optimization, but to find certain patterns in programs, that indicate bugs or, more generally, software deficiencies.

In the latter half of this paper we introduce a number of categories that can be used to classify warnings, which in turn help to select appropriate analysis techniques. In this section we briefly touch on the different types of static analysis techniques. The simplest type of analysis is just searching for keywords, potential dangerous library calls and the like without considering any structural or additional semantic information of the program. The more sophisticated the analysis becomes the more computation is typically required, potentially slowing the analysis down. We consider the following classes of analysis techniques:

Flow-sensitive analysis takes into account the control flow of a program while a flow-insensitive analysis does not. E.g., taking loops and branching behavior into account are characteristics of a flow-sensitive analysis while typical text searches are insensitive.

Path-sensitive analysis considers only valid program paths. This means, more program semantics is considered like variable values conditionals that enables the analysis to distinguish between feasible and infeasible paths.

Context-sensitive analysis takes the calling context of a function such as the states of input parameters and global variables into account. It is a special case of *inter-procedural* analysis, because it not only considers whole-program information, but the actual different program states in which a function is called.

Naturally, the more information that is available, the better the analysis results become. However, from a practical point of view collecting and computing semantic information can result in excessive computation, and slow down the analysis to a point where it is not scalable to larger programs. In most circumstances a static analysis tool is only regarded as useful if the analysis time is roughly in the same order of magnitude as the compilation process and not several orders of magnitude higher.

Apart from the semantic depth of the analysis there is a classification on the type of approximation. We distinguish between *may-* and *must-analyses.*

May-analysis considers over-approximations of program behavior. May analysis, for example, might return as a result for a loop that indicates a specific variable

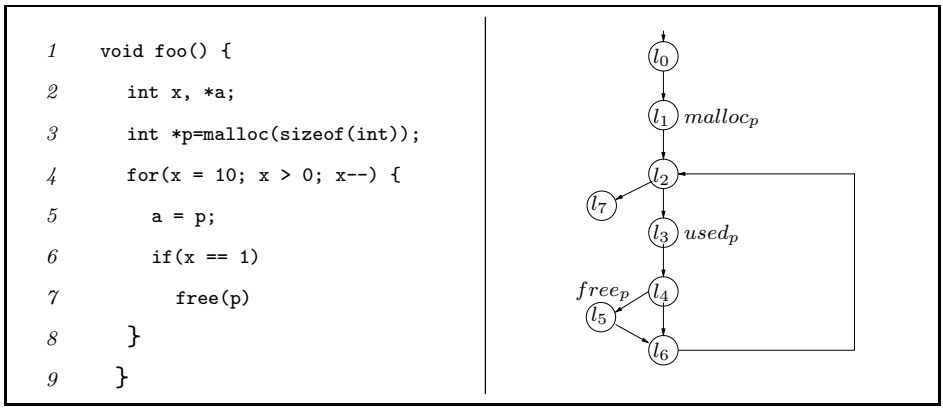

Fig. 1. Example program and labeled CFG for use-after-free check.

is written after the loop, even if the analyzer itself cannot decide if this loop ever terminates.

Must-analysis considers under-approximations of program behavior. Must analysis will not return, for the loop example above, that the same variable is written, as it only considers those effects that are guaranteed to happen.

While may-analysis can turn up significantly more bugs, the detected bugs may not exist in the actual program behavior due to infeasible paths or infeasible data dependencies. In this context these warnings are called *false positives* (or *false alarms*). Must-analysis, however, might miss bugs due to the nature of under-approximation. We call these *false negatives*.

To complicate matters further, modern static program analyzers often mix over- and under-approximations within the same analysis. For instance, the semantics of pointer arithmetic might be under-approximated and the semantics of a loops over-approximated. While not *sound*, those frameworks have proven to be most effective in turning up many bugs without generating many false alarms at the same time [3]. Another complication is that the term *false alarm* is often used in a different way, motivated from the point of view point of a developer, or from the point of view of a tester. Section 4 discusses these alternative uses.

3 The Goanna Approach to Static Analysis

In this work we use an automata based static analysis framework that is implemented in our tool *Goanna*. In contrast to typical equation solving approaches to static analysis, the automata based approach [4,5,6] defines properties in terms of temporal logic expressions over annotated graphs. The validity of a property can then be checked automatically by graph exploring techniques such as model checking [7,8]. Goanna [2] itself is a close source project, but the technical details of the approach can be found in [9].

The basic idea of our approach is to map a C/C++ program to its corresponding

[2] http://www.redlizards.com

control flow graph (CFG), and to label the CFG with occurrences of syntactic constructs of interest. The CFG together with the labels can easily be mapped to the input language of a model checker or directly translated into a Kripke structure for model checking.

A simple example of this approach is shown in Fig. 1. Consider the contrived program `foo` which is allocating some memory, copying it a number of times to `a`, and freeing the memory in the last loop iteration.

One example of a property to check is whether after freeing some resource, it still might be used. In our automata based approach we syntactically identify program locations that allocate, use, and free resource p. We automatically label the program's CFG with this information as shown on the right hand side of Fig. 1. This property can then be checked by the following property in Computation Tree Logic (CTL):

$$AG\ (malloc_p \Rightarrow AG\ (free_p \Rightarrow \neg EF\ used_p)),$$

where AG stands for "for all paths and in all states" and EF for "there exists a path and there exist a state". This means that whenever there is `free` after `malloc` for a resource p, there is no path such that p is used later on.

One advantage of this automata based approach is that properties can be modified easily to express stronger/weaker requirements by changing the path quantifier, i.e., changing an A to an E and vice versa. The following property

$$AG\ (malloc_p \Rightarrow AG\ (free_p \Rightarrow \neg AF\ used_p)),$$

is only violated if the resource p is used on all paths after being freed. While this relaxed property does not pick up as many bugs as the previous one, it also does not create as many false alarms. This is one way to tune a static program analyzers.

For properties defined in temporal logic a model checker can automatically check if they are true. The first property of the example does not hold for the labeled CFG, while the associated requirement – a resource is not used on any path following a `free` – does hold for the program. This is obviously a false alarm. The reason is that the model only contains the CFG and the labels, but does not reflect the semantic fact that p is only freed in the last loop operation and never accessed afterwards. The second property, however, will be true, as there is always at least one path, namely exiting the loop after the free-operation, where there is no access to p after free.

Tuning the strictness of different checks can be one way to improve the bug/false-alarm ratio, another option is to add more semantic information and reflect the fact that it is semantically impossible to access p after a free operation. We implemented a *false path elimination* strategy based on interval constraint solving that does exactly this. Hence, our implementation can use the first check to find all possible bugs and still rule out many false positives automatically.

We implemented the whole framework in our tool Goanna. Goanna is able to handle full C/C++ including compiler dependent switches for the GNU gcc compiler and uses the open source model checker NuSMV [10] as its generic analysis engine. The run-times are typically in the order of the compilation, i.e., we experience an

average overhead of 2 to 3 times the compilation time.

With respect to the dimensions introduced in Section 2, Goanna's analysis can easily be tuned from a may- to a must-analysis (and vice versa) by changing the path quantifier of a property. Moreover, Goanna falls into the category of flow-sensitive, path-sensitive, but unsound program analyzers. This means, Goanna takes the structure of the program into account by always analyzing along the paths in the CFG, it reduces the set of feasible paths by false path elimination and approximates the semantics of different C/C++ constructs differently. This is very much in line with other modern static program analysis tools [11,12,13]. Although Goanna is not yet analyzing inter-procedurally the examination of bug classes and tuning options in the latter remains largely the same.

4 Classification of Properties and Bugs

The most commonly asked question about static analysis tools is to quantify the false positive rate. This is a seemingly straightforward question, but the answer depends very much on the context.

From a developer's point of view *false positives* refer to warnings that are useless clutter, while *true positives* are warnings that compel the developer to fix the code. The bug tracking software Fogbugz, for example, distinguishes between "Must Fix" and "Fix If Time" bugs. The decision in which category a warning falls, may depend on the application area, the life cycle of a project, or the maturity of the code, but is to some extent subjective. A deficiency in mature, well tested code might not be fixed unless there is a serious potential for malfunction, out of a concern that tampering with the code might introduce new bugs. In contrast, even minor deficiencies are likely be fixed in new code, to ensure maintainability of the project in the future.

From the perspective of testing it matters if a deficiency occurs always or under certain circumstances only. Bugs that occur predictably in every run are easy to find. The only penalty for finding them through testing is that valuable resources have to be spent. In contrast, deficiencies that are data or configuration dependent are much harder to find, since it requires a developer to find specific test cases that expose the bug. Since exhaustive testing is often prohibitive, these kinds of bugs can go undetected for a long time before they manifest themselves. Data-dependent deficiencies are also a common cause for security exploits. Bugs that occur always are also called *Bohrbugs*, while bugs that occur only incidentally are called *Heisenbugs* [14]. In static analysis there is a third category of warnings, namely those that do not cause any functional misbehavior, but reflect bad coding practice. Deficiencies in this category cannot be found through testing, but only through code inspection or static analysis.

From the perspective of formal language theory *true positives* are warnings that refer to actual behavior as defined by the program semantics, while *false positives* refer to warnings that are logically impossible. This classification can either be made locally, by flow- or path-sensitive analysis, or globally, by context-sensitive analysis.

```
   /*file:  netwerk/streamconv/converters/ParseFTPList.cpp */
   92   unsigned int numtoks = 0;
   ...
   98   if (carry_buf_len) /* VMS long filename carryover buffer */
   99    {
   100    tokens[0] = state->carry_buf;
   101    toklen[0] = carry_buf_len;
   102    numtoks++;
   103   }
   104
   105  pos = 0;
   106  while (pos<linelen && numtoks<(sizeof(tokens)/sizeof(tokens[0])) )
   107   {
   ...
   111   if (pos < linelen)
   112    {
   ...
   117    if (tokens[numtoks] != &line[pos])
   118     {
   119      toklen[numtoks] = (&line[pos] - tokens[numtoks]);
   120      numtoks++;
   121     }
   122    }
   123   }
   124
-> 125  linelen_sans_wsp=&(tokens[numtoks-1][toklen[numtoks-1]])-tokens[0];
```

Table 1
Warning for a potential out-of-bounds violation on array subscript `numtoks` in line 125.

With local path-sensitive analysis a warning is a *true positive* if there exist input to the function, such that the error path becomes possible. With a context-sensitive analysis a warning is only a *true positive*, if the rest of the system is actually able to provide such an input. If the rest of the system can guarantee that such an input is impossible, the subsystem can use this as an assumption, and guarantee correct behavior under this assumption.

For the development of effective static analysis tools neither of these categories alone are sufficient to consider. It is quite easy to produce warnings that are logically possible, refer to deficiencies that manifest itself always, but will not compel a seasoned developer to change the code. On the other hand, there are warnings that motivate a developer to fix code, even if in the particular instance it has no functional consequences, since fixing increases robustness and extensibility of the code base.

As tool developers we identified three dimensions to judge the quality of warnings.

Severity: high, medium, low.

Incidence: always, sometimes, never.

Correctness: correct, intra-procedural over-approximation, inter-procedural over-approximation.

In the remainder we will discuss each of these categories in detail and provide details to illustrate that the categories are not necessarily correlated, e.g that severity can be independent from incidence. All examples are taken from the Firefox code base [15].

```
   /*file:  /mozilla/js/src/fdlibm/e_asin.c */
   127   if(ix<0x3e400000) {   /* if |x| < 2**-27 */
   128       if(really_big+x>one) return x;
   129   } else
   130       t = x*x;
-> 131       p = t*(pS0+t*(pS1+t*
                    (pS2+t*(pS3+t*(pS4+t*pS5)))));
   132       q = one+t*(qS1+t*(qS2+t*(qS3+t*qS4)));
   133       w = p/q;
   134       return x+x*w;
```

Table 2
Warning for an uninitialized variable caused by an accidental "braceless" else-branch

```
   /* file:/obj/dist/include/xpcom/nsWeakReference.h */
   90  inline
   91  nsSupportsWeakReference::~nsSupportsWeakReference()
   92    {
-> 93     ClearWeakReferences();
   94    }
```

Table 3
Warning for non-virtual destructor `ClearWeakReferences()` in an abstract class.

4.1 Severity

The severity of a warning can be either high, medium or low, and this is typically how developers categorize bugs. Security flaws, even if they have benign causes or only occur under very specific circumstances, can be more severe than errors that have an immediate impact on functionality. A warning might have medium or high severity, even if close manual inspection cannot establish conclusively, whether there is an actual execution producing the bug. Just the chance for the deficiency to create a run-time bug is sufficient reason to change the code.

Out-of-bounds errors are typical example of deficiencies that are considered harmful. Table 1 depicts an example. The first line comment gives the file name and path in the Firefox code base [15]. The code fragment uses an unsigned (positive) integer `numtoks` as array subscript, which is initialized to 0 in line 92. The subscript may be incremented in line 102 and 120, depending on whether certain conditions are true. Variable `numtoks` will not be updated in line 102, if the condition `(carry_buf_len)` in line 92 is false. If the while-condition in line 106 evaluates to false, `numtoks` will not be incremented, and neither of the if-conditions in 111 or 117 evaluate to false. Thus, it is possible that `numtoks` is not incremented at all before line 125. If this case happens `tokens[numtoks-1]` will access an element outside of the array leading to a potential buffer overrun.

It is interesting to note that out-of-bounds accesses are not necessarily functional errors. Using the array subscript -1 to refer to the last element of the previous array in multidimensional arrays is a common and accepted practice. An out-of-bounds warning for such common use would have a very low severity. This is however not the case in this example. Variable `numtoks` is an unsigned integer, which suggests the index should be bounded to positive integers, and it is unclear if the predecessor array for `tokens` is defined in any meaningful way. For this reason this warning was classified to have medium severity.

Out-of-bound errors illustrate that warnings of the same category, can in a certain context be permissable or even common practice. For tool developers this introduces the problem that it is not sufficient to look at the program semantics in isolation, but it is also necessary to look at the programmers' intent.

```
   /*file:  /js/src/fdlibm/e_sqrt.c */
   142  double z;
   ...
   218  if((ix0|ix1)!=0) {
   219      z = one-tiny; /* trigger inexact flag */
   ...
   229  }
   ...
-> 234  u.d = z;
   235  __HI(u) = ix0;
   236  __LO(u) = ix1;
   237  z = u.d;
   238  return z;
```

Table 4
Warning for use of an uninitialized variable `z` in line 234, caused by a conditional initialization in line 219.

Table 2 shows as an example of an uninitialized variable. In this example, variable `t` will only be initialized in the else-branch in line 130. In line 131 variable `t` will however be used regardless of whether it was initialized. From indentation it appears that the entire block from line 130 to line 134 was intended to be in the else-branch. In this case, the actual error, missing braces, was caught because it caused the use of an uninitialized variable. This deficiency had been present in the code for several years, but it was not found through testing or bug reports by users, since this code fragment is only invoked for a rare platform. Although obviously a bug, it was classified to have a low severity.

Table 3 shows a warning for using a non-virtual destructor in an abstract class. The problem with this construct is that even if a subclass defines its own destructor, it will use the destructor of the abstract class. This might have unexpected consequences – unexpected from the developers point of view, consequences that may include memory leaks. Although the use of a non-virtual destructor in an abstract class may not result in unexpected behavior if all developers are carefully aware of this, it might become a legacy problem. Even if there is no effect, and even if the associated misbehavior never occurs, this warning is considered important enough for developers to change the code.

4.1.1 Tuning Implications.

One of the conclusions to be drawn from the above observations is that it is necessary to embed different severity levels for warnings in a static program analyzer. While, e.g., null pointer dereferences, out-of-bounds errors and potential memory leaks should get a relatively high severity level, code cleanliness such as unused values or dead code should generally get a lower severity level. However, since different stages of product development might have different requirements, it is advisable to make those categories configurable and let the user decide which properties should go into which severity level.

Goanna is highly configurable, users can place different properties into different severity levels and they can also create their own groups. For instance, having a group for ongoing development and a group for legacy code checks. Goanna can be adapted according to the user's needs which are not fixed a priori.

```
   /* file:dom/src/offline/nsDOMOfflineLoadStatusList.cpp */
   495  NS_IMETHODIMP
   496  nsDOMOfflineLoadStatusList::Observe(nsISupports *aSubject,
   497                                      const char *aTopic,
   498                                      const PRUnichar *aData)
   499  {
-> 500   nsresult rv;
   501   if (!strcmp(aTopic, "offline-cache-update-added")) {
   502    nsCOMPtr<nsIOfflineCacheUpdate> update=do_QueryInterface(aSubject);
   503   if (update) {
   504     UpdateAdded(update);
   505    }
   506   }else if (!strcmp(aTopic, "offline-cache-update-completed")) {
   507    nsCOMPtr<nsIOfflineCacheUpdate> update=do_QueryInterface(aSubject);
   508    if (update) {
   509      UpdateCompleted(update);
   510    }
   511   }
   512
   513   return NS_OK;
   514  }
```

Table 5
Warning for an unused variable `rv` in line 500.

4.2 Incidence

The incidence of a warning refers to whether the associated behavior/bug happens always, or only for some runs. Finding a bug through testing gets harder if the bug depends on the configuration and/or input. An example of a input dependent bug is given in Table 4. Variable `z` will be used uninitialized if the bit-wise comparison `(ix0|ix1)` in line 218 evaluates to false. Finding this deficiency through testing requires the construction of a test input such that both variables `ix0` and `ix1` are equal to 0. Static analysis in contrast, applies an abstraction to the possible input, and is therefore capable of pinpointing the error. A drawback of this approximation through abstraction is that static analysis might flag deficiencies that cannot occur, either because a certain combination of conditions never occurs, or because a certain combination of input never occurs. In this example we only have a single condition, and from the comment `trigger inexact flag` in line 219, we conclude that the condition can be false. Otherwise the "inexact" case would be the only case.

Table 5 shows an unused variable. Even though it might seem counterintuitive, this warning points to a deficiency that occurs *always*; each run of the program declares the variable, but never uses it. It should be noted that unused variables are almost never of medium or high severity. These deficiencies will only be removed, if at all, to clean up the code. This is an example of a warning that reports a deficiency that manifests itself always, is semantically correct, but still has a low severity.

4.2.1 Tuning Implications.

Catching bugs depending on their incidence can best be tuned by the analysis algorithms used. Must-analysis is good at picking up bugs that will always occur, while may-analysis picks up potential bugs on single executions. Moreover, a path-sensitive analysis can filter out those combinations of conditions that are infeasible in the execution. To get a better understanding of inputs from other parts of the program and to increase the precision of the analysis a full context-sensitive approach should be used.

Goanna supports the fine tuning of properties by reasoning about some program path or all paths as described in Section 3. Moreover, Goanna can automatically eliminate infeasible paths that are caused by a set of excluding conditions. Goanna does not perform full context-sensitive analysis, yet. However, one can argue that every function should be implemented defensively, i.e., inputs should be checked before being used to avoid unexpected crashes.

4.3 Correctness

For a given warning we need to decide if this warning corresponds to an actual execution with respect to the program semantics. A warning can be spurious, i.e., the associated behavior can be logically impossible for the following two reasons: First, because of an over-approximation of the control flow, and second, because the context in which a function is used was over-approximated. Table 6 gives an example of two warnings that are caused by an over-approximation of the control flow, in particular of the short-circuit operator `&&`. While the C and C++ standard leaves the evaluation order for some operators explicitly undefined, i.e., it is compiler depended, it defines that the short-circuit operators are executed from left to right. The concatenation of short-circuit operators in line 894 in Table 6, evaluates first the variable `doc`, assigns then a value to variable `shell`, which is then used in the third expression. To fix the spurious warnings it is sufficient to locally refine the control flow graph to reflect this behavior.

Spurious warnings involving the context are much harder to fix. A typical example is null-pointer analysis. Pointer arithmetic and aliasing are known to be hard problems, and depending on the precision of the analysis do not scale to large code bases. Pointers can be passed through several functions and modified in each of them, which requires an inter-procedural and context-sensitive analysis. On the other hand there is an acceptable level of spurious warnings for a null-pointer analysis, given that bugs that are caused by null-pointer dereferences are often severe.

4.3.1 Tuning Implications.

Reducing false alarms resulting from intra- or inter-procedural over-approximations can best be addressed by finer abstractions and specialized analysis algorithms. In the example above creating a fine grained CFG for the analysis of short circuit operators can help. Moreover, a specialized inter-procedural pointer analysis taking aliasing into account can also aid in reducing context-sensitive over-approximations. Often, incorrect warnings are caused by one of the many exceptions, and corner cases that exist in C and C++. In this case it is usually sufficient to refine the syntactic description of the properties on an intra-procedural level. This technique is also effective to cover coding practices that are not according to standard, but nevertheless common.

Goanna has some experimental features for creating refined CFGs on demand and tracking pointer aliases through several functions in a modular way. It is, however, not fully implemented yet and the results in the subsequent section are obtained without them.

```
    /* file:/dom/src/base/nsLocation.cpp */
    890  nsCOMPtr<nsIDocument> doc(do_QueryInterface(...));
    891
    892  nsIPresShell *shell;
    893  nsPresContext *pcx;
->  894  if (doc && (shell = doc->GetPrimaryShell()) &&
    895      (pcx = shell->GetPresContext())) {
    896    pcx->ClearStyleDataAndReflow();
    897  }
```

Table 6
The condition on line 894 generates two spurious warnings. An uninitialized variable warning, and an unused assignment warning, both of variable `shell`.

5 Empirical Results

Our implementation is written in OCaml and we use as the back-end model checker NuSMV 2.3.1. The current implementation is an early version consisting mostly of intra-procedural analyses for full C/C++. At the moment we have implemented 18 different classes of checks. These checks cover, among others, the correct usage of malloc/free operations, use and initialization of variables, potential null-pointer dereferences, memory leaks and dead code. The CTL property is typically one to two lines in the program and the description query for each atomic proposition is around five lines because of the need to cover many exceptional cases.

We evaluate our tool with respect to run-time performance, memory usage, and scalability, by running it on a regular basis over nightly builds of the Mozilla code base for Firefox. The code base has 1.43 million lines of of pure C/C++ code, which becomes 2.45 million non-empty lines after preprocessing. The analysis time including the build process for Firefox is 234 minutes on a DELL PowerEdge SC1425 server, with an Intel Xeon processor running at 3.4 GHz, 2 MiB L2 cache and 1.5 GiB DDR-2 400 MHz ECC memory.

About 22% of the time is spent on generating the properties including NuSMV model building, 55% on model checking itself, and about 16% on parsing and 9% on supporting static analysis techniques such as interval constraint solving techniques. Some more run-time and scalability details can be found in [9].

The results of the regular runs over the Firefox code base are also used to evaluate the quality of the warnings, and to document the evolution of the checks over time. To do this we take a statistically significant random sample of the warnings for each check, and analyze the warnings contained in the sample manually. By addressing the weaknesses, we have reduced the number of warnings from over 48000 to about 8000. But since this is still very much ongoing work these figures are subject to significant changes. The reductions were not uniform over the 18 different classes, and the different checks also have a different potential for further reductions. The remainder of this section discusses the experiential outcome for a selected group of checks.

Unused Parameter and Variables.

In the initial experiments unused parameters and variables accounted for more than 31000 warnings. An inspection of the warnings revealed that most of them were caused by a structural use of unused parameters and variables throughout

Firefox. This was obviously an accepted coding practice, to achieve consistency throughout the code base, rather than being accidental omissions or errors. It is interesting to note that all of these warnings were semantically correct, pointing to behavior that occurs always, and thus are *true positives* from a language semantics point of view. However, from a user perspective these warnings are of a very low severity.

We divided the previous check into two checks to improve the significance: one that flags all occurrences of unused variables, and a second that only flags those that are likely to be unused inadvertently. The first check is by default disabled, and only using the second check reduced the number of warnings to 113, which is a 99.65% reduction. This was achieved by a slight modification of the associated syntactic requirements. This change removed about 64% of the 48000 warnings, i.e., we removed a significant number of *true positives* that happened always, were semantically correct, but of little interest.

Null Pointer Dereference.

One of the most common causes of bugs in C and C++ is the incorrect handling of pointers. The first experiments resulted in more than 9000 warnings for dereferences of potential null pointers. Many of those were caused by a conservative over-approximation of the possible paths. By improving the syntactic requirements the number of warnings was reduced to 1200. This number is still too high, and further improvements will have to come from an improved path-sensitive analysis, but more importantly from an improved context-sensitive analysis.

To identify a lower bound of null pointer warnings, we divided the one check into four checks, distinguishing between definite paths and potential paths on one side, and between potential and likely null pointers as detectable ny our analysis on the other hand. First experiments show that a further reduction in warnings of about 75% is possible, but only a context-sensitive analysis could provide definite answers.

Dead Code Analysis.

Dead code warnings are typically of a low severity, and our analysis tries to be very careful about flagging dead code. From the early experiments on the warning for dead code were almost always correct. However, we noticed that in a fair number of cases the dead code was inserted on purpose. A typical example are switch statements that have a return in each of the cases, which is followed by an additional return statement. Some compilers complain about a missing return statement, even if there is one in each case of the switch statement. Hence, developers add code which is actually dead. This and similar cases of dead code warnings account for about 25% of the warnings, and can be effectively suppressed by changing the syntactic requirement. Moreover, programmers often add a comment stating that some code is dead, but still keep it around.

Virtual Function Call in Destructor.

While a virtual function call in a destructor does not necessarily cause a problem, it might result in memory leaks, and is considered bad coding practice, regardless of the effects. Our analysis of the Firefox code base found 114 instances of such usages. From those warnings, 105 refereed to the same macro that was used over and over again, and thus caused as many warnings after preprocessing. This points to another way to improve the usability of static analysis tools, namely to present to the user with a concise set of warnings. Except for this duplication of warnings, the check itself was kept unchanged, as all of the warnings are considered valuable.

Summary.

Overall, we achieved an 83% reduction in false alarms and very low severity warnings by mostly changing the precision of our rules. This was achieved by taking care of particular programming styles and strengthening our CTL properties. Encoding checks as CTL properties and switching from a may- to a must-analysis easily by changing the path quantifier proved crucial to quickly adjust the granularity where needed.

Currently, our reported defect density is around 3.2 warnings per 1000 LoC for the Firefox code base. The best state-of-the art checkers provide a defect density of around 0.35 for Firefox. While there appears to be a significant gap between those two numbers, when neglecting low impact properties such as unused values and taking into account must-properties only, we achieve a defect density of 0.36. However, those numbers can only serve as a rough comparison, because we do not know about the exact checks commercial tools are implementing, their reporting strategy or the exact techniques used. Nonetheless, understanding programming styles, severity of bugs and the importance of scalability are in a first analysis step more important than deep semantic analysis techniques.

6 Conclusions

In this work we presented our experiences in tuning static program analysis for large software systems. In particular we gave a classification of properties and bugs based on our practical experiences in keeping the warning rate down to report relevant issues. We advocate that a crucial first step is to get familiar with programming styles and the oddities in real code and to fine tune syntactic checks accordingly instead of applying an expensive semantic analysis right from the beginning. Typically static program analysis returns a manageable set of warnings. In a future step, only this set should be subjected to full context-sensitive analysis to keep the overall analysis scalable.

Related to our work is an evaluation and tuning of static analysis for null pointer exceptions in Java as described in [16]. The authors show that many null pointer bugs can be found on a syntactic level without sophisticated semantic analysis. They show that fine tuning syntactic checks is the key for good analysis results. A similar conclusion has been drawn in [17] where the authors studied coding patterns and

use contextual information based on domain knowledge to reduce the false positive rate. Ayewah et al. evaluate in [18] the static analysis tool Findbugs, on Java source code. Their test bench contains the code for Sun's Java 6 JRE, Sun's Glassfish JEE server, and portions of Google Java code base. Their findings are similar to ours, but they report from a user's perspective. In our paper we describe the heuristics that can be used by developers of static analysis tools to increase the quality of the warnings.

More information about prominent commercial static analyzers can be found in [19]. The authors compare a number of tools and point out their strength and weaknesses as well as their experiences of using them within Ericsson. A more general comparison of some static program analyzers including a discussion on bugs can be found in [21,20,22].

Acknowledgement

NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program.

References

[1] Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques and Tools. Addison-Wesley (1986)

[2] Nielson, F., Nielson, H.R., Hankin, C.L.: Principles of Program Analysis. Springer (1999)

[3] Engler, D., Chelf, B., Chou, A., Hallem, S.: Checking system rules using system-specific, programmer-written compiler extensions. In: Proc. Symposium on Operating Systems Design and Implementation, San Diego, CA. (October 2000)

[4] Holzmann, G.: Static source code checking for user-defined properties. In: Proc. IDPT 2002, Pasadena, CA, USA (June 2002)

[5] Dams, D., Namjoshi, K.: Orion: High-precision methods for static error analysis of C and C++ programs. Bell Labs Tech. Mem. ITD-04-45263Z, Lucent Technologies (2004)

[6] Schmidt, D.A., Steffen, B.: Program analysis as model checking of abstract interpretations. In: Proc. SAS '98, Springer (1998) 351–380

[7] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons for branching time temporal logic. In: Logics of Programs Workshop. Volume 131 of LNCS., Springer (1982) 52–71

[8] Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Proc. Intl. Symposium on Programming, Turin, April 6–8, 1982, Springer (1982) 337–350

[9] Fehnker, A., Huuck, R., Jayet, P., Lussenburg, M., Rauch, F.: Model checking software at compile time. In: Proc. TASE 2007, IEEE Computer Society (2007)

[10] Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: Intl. Conf. on Computer-Aided Verification (CAV 2002). Volume 2404 of LNCS., Springer (2002)

[11] Coverity: Prevent for C and C++. http://www.coverity.com

[12] Klocwork: K7. http://www.klocwork.com/

[13] Fortify: Fortify static code analysis. http://www.fortifysoftware.com/

[14] Gray, J.: Why do computers stop and what can be done about it? In: Symposium on Reliability in Distributed Software and Database Systems. (1986) 3–12

[15] Mozilla: Source code for Firefox, nightly build. Avaiable at: ftp://ftp.mozilla.org/pub/mozilla.org/firefox/nightly/

[16] Hovemeyer, D., Spacco, J., Pugh, W.: Evaluating and tuning a static analysis to find null pointer bugs. In: PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, New York, NY, USA, ACM (2005) 13–19

[17] Reimer, D., Schonberg, E., Srinivas, K., Srinivasan, H., Alpern, B., Johnson, R.D., Kershenbaum, A., Koved, L.: Saber: smart analysis based error reduction. In: ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM (2004) 243–251

[18] Ayewah, N., Pugh, W., Morgenthaler, J.D., Penix, J., Zhou, Y.: Evaluating static analysis defect warnings on production software. In: PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, New York, NY, USA, ACM (2007) 1–8

[19] Emanuelsson, P., Nilsson, U.: A comparative study of industrial static analysis tools. In: 3rd International Workshop on Systems Software Verification (SSV 08), Submitted for publication (2008)

[20] Kratkiewicz, K.: Evaluating static analysis tools for detecting buffer overflows in C code. Master's thesis, Harvard University, Cambridge, MA (2005)

[21] Cousot, P., Cousot, R., Feret, J., Mine, A., Mauborgne, L., Monniaux, D., Rival, X.: Varieties of static analyzers: A comparison with ASTREE. In: TASE '07: Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, IEEE Computer Society (2007) 3–20

[22] Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. In: SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering, New York, NY, USA, ACM (2004) 97–106