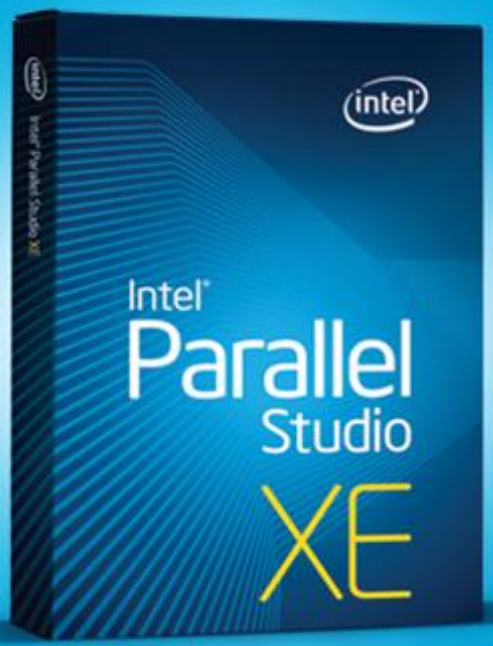
Improve Fortran Code Quality with Static Security Analysis (SSA)

with Intel® Parallel Studio XE



This document is an introductory tutorial describing how to use static security analysis (SSA) on C++ code to improve software quality, either by eliminating bugs or making the code more secure, or both. It provides a brief description of the goals of the product feature and walks through an end-to-end example showing how it is used.

Static Security Analysis is available in the Intel® Parallel Studio XE and Intel® Cluster Studio XE suites.

Introduction

This document is an introductory tutorial describing the static security analysis feature of the Intel® Parallel Studio XE. It provides a brief description of the goals of the product feature and walks through an end-to-end example showing how it is used.

What is Static Security Analysis (SSA)?

Static security analysis (SSA) attempts to identify errors and security weaknesses through deep analysis of source code. SSA is primarily aimed at developers and QA engineers who wish detect software defects early in the development cycle in order to reduce time, cost and increase ROI. It also assists developers in hardening their application against security attack. SSA provides an effective way to discover defects, especially in code that is hard to exercise thoroughly with tests. SSA can also detect race conditions resulting from misuse of parallel programming frameworks such as OpenMP and Intel Cilk Plus.

Many coding errors and patterns of unsafe usage can be discovered through static analysis. The main advantage of static analysis over dynamic analysis is that it examines all possible execution paths and variable values, not just those that are provoked during testing. This aspect of static analysis is especially valuable in security assurance, since security attacks often exercise an application in unforeseen and untested ways.

SSA can detect over 250 different error conditions. They include buffer overflows and boundary violations, misuse of pointers and heap storage, memory leaks, use of uninitialized variables and objects, unsafe/incorrect use of C/C++ or Fortran features and libraries.

How does Static Security Analysis Work?

Static security analysis is performed by the Intel® C++ and the Intel® Fortran Compilers operating in a special mode. In this mode the compiler dedicates more time to error analysis and bypasses the instruction generation process entirely. This allows it to find errors that go undetected during ordinary compilation. SSA requires your code to compile without serious errors using the Intel Compiler, but you do not execute the results of this compilation. You do not need to use Intel compiler to create your production binaries in order to take advantage of SSA. We will begin the tutorial with showing how to prepare an application for SSA.

SSA can be done on a partial program or a library, but it works best when operating on an entire program. Each individual file is compiled into an object module, and the analysis results are produced at the link step. The results are then viewed in Intel® Inspector XE.

The results of static analysis are often inconclusive. The tool results are best thought of as potential problems deserving investigation. You will have to determine whether a fix is required or not. The Intel Inspector XE GUI is designed to facilitate this process. You indicate the results of your analysis by assigning a state to each problem in the result. Intel Inspector XE saves the state information in its result files.

Over time the source will change and you will want to repeat the analysis. The first time you open your new analysis result in Intel Inspector XE, it automatically calculates a correspondence between the previous result and the new result. It uses this correspondence to initialize the state information for the new result. This means you do not have to investigate the same issues over again.

When you decide a problem does require fixing you should report it into your normal bug tracking system, just as you would report a defect detected by an executable test. Intel Inspector XE is not a bug tracking system. The state information tracks your progress in investigating the result of analysis. What you do with your conclusions is outside the scope of SSA and the Intel Parallel Studio XE.

Setting up for SSA

The set up process is usually fairly simple, but it can be quite complicated, depending on the situation. For Microsoft* Windows* when using Microsoft* Visual Studio*, set up is automated with a simple menu/dialog driven approach that automatically configures your solution for SSA. Details on how to use this and how it works are in the tutorial that follows.

In Linux* OS or when building with a make file or command line script in Windows we recommend you modify your build procedure (whatever it is) to create a new build configuration for SSA. The term build configuration refers to a mode of building your application, using specific compiler options and directing the intermediate files to specific directories. Most applications have at least two build configurations: debug and release. You will want to create one more for SSA. The SSA configuration must build with the Intel compiler with additional options set to enable SSA.

Please note the distinction between creating a new build configuration and building an existing configuration with additional options. If you build, say, your debug configuration with the additional options that enable SSA then you will get analysis results (as long as your debug configuration builds with the Intel compiler). This is a perfectly good way to do an initial product evaluation. However, it's awkward to work this way on an ongoing basis, because the object modules produced by SSA will overwrite your debug-mode object modules every time you run SSA. Just as you want to keep debug object modules separate from release object modules, you will want to keep debug object modules separate from SSA object modules. If you are going to use SSA on an ongoing basis, you will want to get set up properly.

The process for creating a new build configuration will be different for each application. The sample applications used in this tutorial can be built under Visual Studio on Microsoft Windows OS or with a make file on Linux OS. We will walk through the steps needed to update this make file for SSA.

If your application build is very complex and you don't feel confident that it can be modified safely, there is an alternative set up method. You can execute your normal build under a "watcher" utility called inspxe-inject. This application intercepts process creations and recognizes all the compilation and link steps performed during your build. It records this information in a build specification file. This file can be supplied as input to another utility, inspxe-runsc, which invokes the Intel compiler to repeat the same build steps as your original build did. These utilities are not explained in this tutorial.

Fortran Tutorial

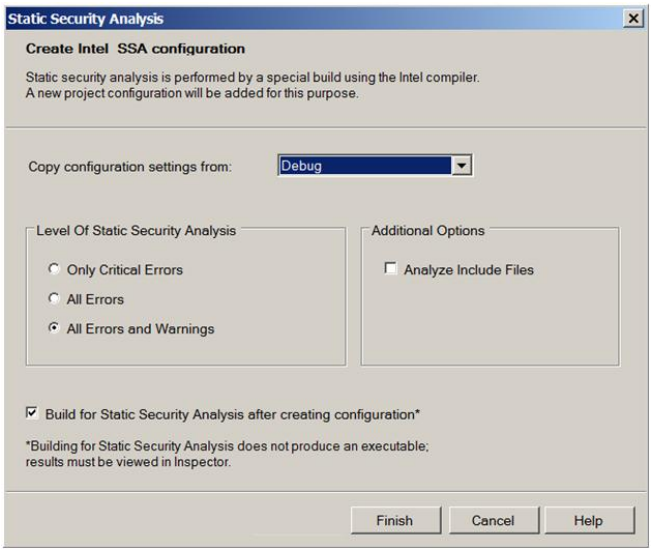
We will be using a sample application called "FortranDemo_ssa". You can find it below the Intel Inspector XE root install directory, below the "samples" subdirectory. Unzip this application to directory on your disk.

We will start by getting set up for SSA. This is done differently on Windows and Linux OS. For Windows OS read the following section. For Linux OS, go to Setting up and Running SSA using a make file on Linux OS.

Setting up and Running SSA Using Visual Studio on Windows OS

We will go through the setup process using Visual Studio solution FortranDemo.sln that is supplied with the sample program. Start by opening this solution in Visual Studio. The process for setting up for SSA is quite simple. If you are using Visual Studio 2008 or Visual Studio 2010 then Visual Studio will need to convert for solution file from Visual Studio 2005 format. Let it go ahead and do this conversion.

Set up for SSA by selecting **Build > Build Solution for Intel Static Security Analysis** menu item. This brings up the 'Create configuration' dialog which is used to choose an existing baseline configuration and specify the SSA options.



Click 'Finish' to accept the default settings. This creates a new Intel_SSA configuration. It copies the properties from the baseline (Debug) configuration and then changes it to enable the selected SSA-related options. Since the checkbox next to "Build for Static Security Analysis after creating configuration" is checked, it will immediately build this configuration, which is equivalent to running SSA.

You are now set up for SSA and have launched your first analysis.

For information on setting up applications built using a command line script or make file, see the following section explaining the setup process on Linux OS..

Setting up and Running SSA using a make file on Linux OS

The sample application comes with a make file that is used to build the application on Linux OS. The included make file will build the application using the Intel Fortran compiler and includes a build target, SSA. The SSA build target is just like the debug build target except for these changes:

1. It adds the additional option "-diag-enable sc3"
2. It puts the intermediate files in the SSA subdirectory.

This illustrates the changes that are needed to update a make file to prepare for SSA.

Now that you are set up, all you have to do is build your SSA build configuration to perform analysis. You can do this by performing the following steps:

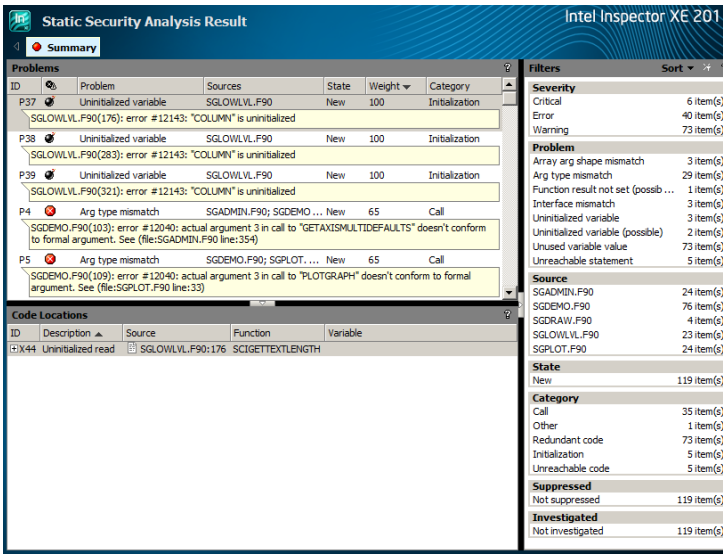
1. Open a command shell
2. Set the environment variables for the Intel compiler by executing the `ifortvars.sh` script in the compiler bin directory, supplying the `ia32` option.
3. Execute `make ssa`

TIP: keep this command window open for later operations.

Interacting with the Analysis Results

On Windows OS, the Intel Inspector XE automatically opens a new result as soon as the build completes. On Linux OS, type `inspxe-gui` to invoke Intel Inspector XE and then use the **File > Open** menu to open the result. By default, the file you want to open is called `r000sc.inspxe`, and is contained in a directory named `r000sc` below the root directory of the **FortranDemo** project.
The remainder of the Fortran demo is almost identical for Windows and Linux OS. The main difference is that the Intel Inspector XE GUI is embedded within Visual Studio on Windows OS, while on Linux OS the GUI runs as a stand-alone program. The look of the individual Intel Inspector XE windows is almost identical on Windows and Linux OS.

The initial window you will see will look something like this. (Note: you might have to drag the scrollbar up to the top to get the right line on top.)



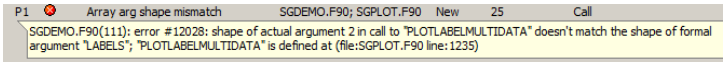
This window consists of three main areas. The upper left pane is the table of Problem Sets. This is your "to do" list, the things you need to investigate. The lower left pane shows the code locations corresponding to the currently selected problem set. The right pane shows the filters. It controls what problem sets are displayed and which are hidden.

You can sort the table of problem sets by clicking on any of the column headers. By default the problem set table is sorted by **weight**. The weight is a value between 1 and 100 which reflects how interesting a problem is. Problems that can do more damage have higher weight. Problems that are more likely to be true problems (as opposed to false positives) are also given higher weight. So the weight provides a natural guidance for your order of investigation.

Look at P1, Array arg shape mismatch, and see what we can learn about this problem.

Investigating a Problem

So far all we know about the problem is summarized in the table entry:

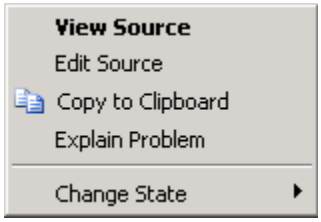


Array arg shape mismatch is the short description of the problem. The full description is shown in the shaded area. The **New** entry in the state column indicates that this problem was discovered for the first time in this analysis run and has not yet been investigated. The 25 weight value indicates the weight. The category of problem is **Call**, which means it is related to the assignment of an actual array argument to a dummy argument of a different size.

Click on this problem to select it. The lower pane refreshes itself to show the source code locations related to this problem:

Here we see two source locations. The first is where the function PLOTLABELMULTIDATA was called by the subroutine LINEDEMO ("Call site") and the other is where PLOTLABELMULTIDATA is defined.

One way we could get more information about a problem is to read an explanation of what that problem type means. Some SSA errors are pretty technical and require explanation. To see the explanation for this problem type, right click on the problem. That brings up this pop-up menu:



Select **Explain Problem**. This brings up a help topic that explains this problem in detail. This is the help topic this particular problem type:

Array parameter shape mismatch

The type of an actual argument does not match the corresponding formal parameter at a subroutine call.

Specifically, this error occurs when both the actual and formal parameter types are arrays but the arrays have different shape.

This same kind of error can also happen when a FORTRAN dummy argument of type subroutine is invoked. That is, the subroutine that is invoked through a dummy argument might exhibit the same problem as can occur in a direct call. In this case, the problem may or may not occur, depending on what subroutine was passed to the dummy argument of subroutine type. There will be an additional observation in such cases that identifies the call site where the subroutine argument was passed in.

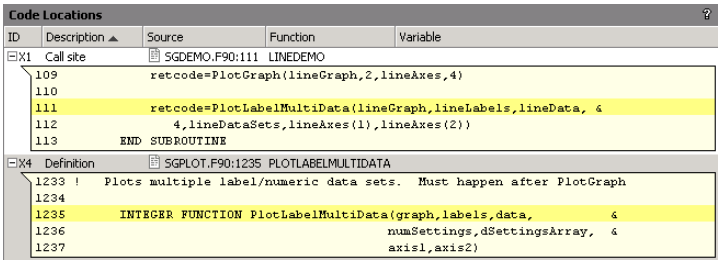
ID	Observation	Description
1	Call site	The actual argument that was passed
2	Definition	The definition of the called procedure and shows the declaration of the formal parameter

Example

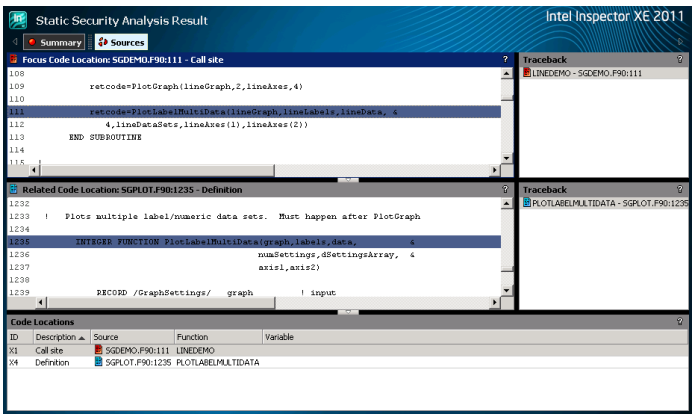
```
subroutine mysub(m)
type mytype1
    integer f1
    real f2
end type mytype1
type (mytype1), dimension(2,3) :: m
print *, m
end

type mytype2
    integer g1
    real g2
end type mytype2
type (mytype2), dimension(3,2) :: n
n%g1 = 1
call mysub(n)
! shapes of dummy argument and actual argument are different.
print *, n%g1
end
```


As you can see, the problem type reference material explains more fully what the precise error condition is, its potential consequences, and provides an example that demonstrates the problem.

Next, determine whether this problem is really present in our application. To do that, look at the source code. The fastest way to do that is to expand the code reference in the lower pane to expose a small snippet at the referenced location. There are two ways to do this. One is to click the plus sign in the ID column. The other is to right-click on the item in the lower pane and select **Expand All Code Snippets** from the pop-up menu. You will see this:

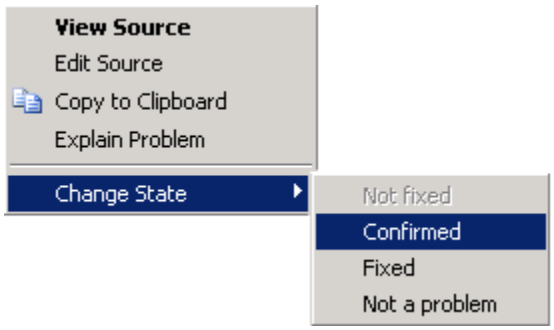


In this case, the code snippets do not really show enough as to what issue is. Double click the first snippet to open the **Sources** view.

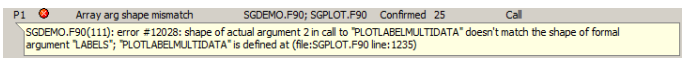


You can see in the bottom source window that the second argument of PlotLabelMultiData, labels, is a single element array. If you were to scroll up in the top source window you would see that the actual argument, lineLabels, is defined as a six element array, hence the error.

Since you have confirmed that this is a real error, record our conclusion. Right click the problem and select **Change State > Confirmed** from the pop-up menu.



You can see the state is updated in the problem set table:



Reducing Clutter with Filtering

Filters allow you to focus on the problems you are interested in and hide the problems you want to ignore.

Once a problem has been investigated there is usually no reason to look at it again. One of the nicest uses for filters is to hide all the problems you are finished investigating.

Go all the way to the bottom of the filter window and click the **Not investigated** item inside the **Investigated** filter.

When you do that, the filter item redraws to indicate that it is active:

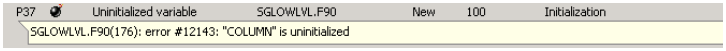
Notice how that problem we marked as "Confirmed" disappeared from the table of problem sets when we did that. It a good idea to keep this filter set like this while you work on analyzing results.

The first several filters, Severity, Problem, Source, State, and Category, correspond to columns in the table of problem sets. You can hide all rows in the table that do not match a specific value in some column. Click on the second line in the Source filter (SGDEMO.F90). It will look like this:

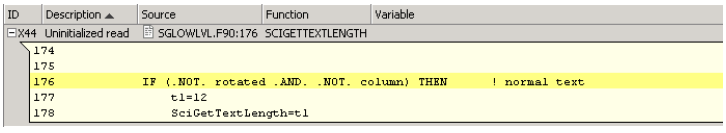
Notice how the problem set table has also redrawn to show only problems in this source file. You can turn off a filter by clicking the **All** box, but leave it turned on for now.

Investigating a Second Problem

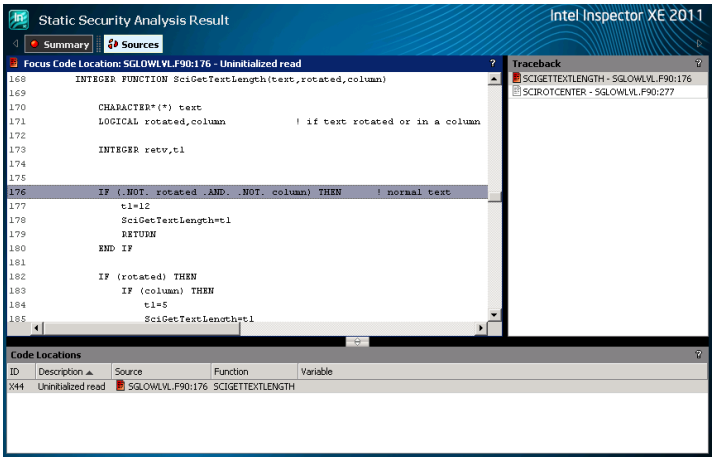
Take a look at another problem in the solution. Scroll down to P37 which marked critical in the table of problem sets:



To investigate this issue, select this problem set and look at the lower pane to see the related source code:



This shows us the use of column, but it doesn't tell us enough to understand the problem. To see the actual source, right click the code snippet and select **View source**. This is what you will see:



This is the **Sources** view. It contains one pair of windows, showing a section of source code and (on the right) a Traceback. Up at the top left you see a highlighted box that says **Sources**. To the left of that you see another box that says **Summary**. You can click this box to go back to the summary view we were looking at earlier.

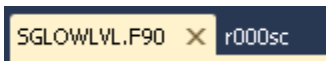
The source shows that "column" is an input argument to the function SciGetTextLength. So where is SciGetTextLength being called? If you remember, SSA performs a whole-program, cross file analysis. It analyzes the flow of data values through procedure calls, even across files. Traceback information answers this question.

If you look at the Traceback, you will see that SciGetTextLength is called by SciRotCenter. Click SCIROTCENTER in the Traceback window and you will see the call to SciGetTextLength in the source code window. Scrolling up through the source code window you can see that 'column' has not been initialized prior to calling SciGetTextLength.

Fixing a Problem and Rescanning

Let's fix this problem by entering the statement "column = .true." at source line 276. To bring up your normal source editor, you can either double click that line the sources view window, or right click that line and select **Edit Source** from the pop-up menu. On Linux OS this will activate whatever editor is selected by the EDITOR environment variable. On Windows OS this opens the Visual Studio source editor.

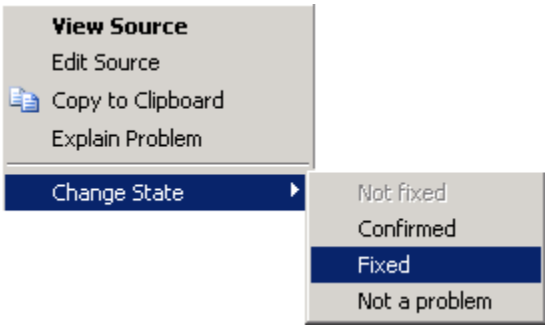
On Windows OS, the editor window will open a new tab in the same tab group window that contains the results. So opening the source for editing will hide the result. To view the results again, click the **r000sc** tab.



After making the edit, save the result, and close the editor but do not rebuild the application yet.

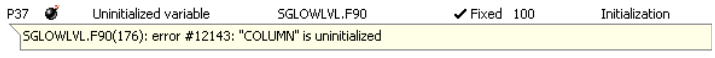
Now, go back into the summary view. On Visual Studio you may need to click on the **r000sc** tab to get back to the Intel Inspector GUI. To get from the **Sources** view to the **Summary** view, click on the box that says **Summary** at the upper left of the **Sources** view.

Use the right-click pop-up menu to change the state of P37 to say **Fixed**.



Notice that as soon as you do this, the problem disappears, since you set the filter set to show only uninvestigated problems. To see that problem again, go to **Filters** pane, go down to the bottom and turn off the Investigated filter by clicking on All:

Now you can see the problem again, and see that it really is marked as Fixed.



Now go ahead and rebuild the application to create a new analysis result and open the new result, r001sc. (P37, P38, and P39 were all related issues and generally you would have wanted to investigate all three prior to rescanning, but for the sake of this example go ahead and rescan now). On Windows OS you can do all this by selecting the **Build>Rebuild solution for Static Security Analysis...** menu item. You will want to close the Visual Studio Output window when the build completes. On Linux OS you should repeat the steps you used earlier to build (rerun the make file in your command windows, then use **File > Open** in the Intel Inspector GUI to open **r001sc**).

The first thing you will notice is that most problems now say **Not fixed** instead of **New** in their state. This demonstrates the way that Intel Inspector XE automatically initializes the state in the new result, r001sc, from the previous result, r000sc. Problems that were seen before are no longer considered **New**. They are simply not investigated yet, which is what the **Not fixed** state indicates.

You will also notice that the problem we investigated earlier, the array parameter shape mismatch problem, is still in a **Confirmed** state, just as we marked it in r000sc.

You will see that the number of critical errors has been reduced from 6 to 5, however, the error eliminated was P38, one of the related errors pointed out earlier. P37 which was marked as **Fixed** in r000sc is now marked as a **Regression**. This indicates that the problem still exists. **Regression** is considered an uninvestigated state, so this problem would still be seen if you set the filter to show only problems whose investigation state is **Not investigated**. If you were to investigate P38 (P39 in r000sc) you would see that there is another instance of SciGetTextLength being called without "column" being initialized. Fixing this issue will then also resolve P37.

Summary and review

To review, we have seen:

1) The table of problem sets summarizes the problems found by SSA.
2) Selecting a problem set shows the associated code locations in the lower pane.
3) Dig into a problem by viewing code snippets, going to the Sources view (where you can see tracebacks), or going to your source editor.
4) Get a full explanation of what a problem type means by using the "Explain Problem" menu item to access the problem type reference material.
5) Set the state of a problem set to record the results of your investigation.
6) Use filters to hide problems that you are done investigating, or to focus on particular source files or particular classes of problems.
7) SSA automatically carries state information forward from result to result. It keeps track of what problems are new, what problems were investigated, and verifies that fixed problems are really fixed.

Additional Resources

Intel Learning Lab – Technical videos, whitepapers, webinar replays and more.

Intel Parallel Studio XE product page – How to videos, getting started guides, documentation, product details, support and more.

Evaluation Guide Portal – Additional evaluation guides that show how to use various powerful capabilities.

Intel® Software Network Forums – A community for developers.

Intel® Software Products Knowledge Base – Access to information about products and licensing,

Download a free 30 day evaluation

Improve Fortran Code Quality with Static Security Analysis (SSA)

Purchase Options: Language Specific Suites

Several suites are available combining the tools to build, verify and tune your application. All the suites listed below offer Static Security Analysis. Single or multi-user licenses and volume, academic, and student discounts are available.

	Suites >>	Intel® Parallel Studio XE	Intel® C++ Studio XE	Intel® Fortran Studio XE	Intel® Cluster Studio XE	Intel® Cluster Studio
Components	Intel® C / C++ Compiler	●	●		●	●
	Intel® Fortran Compiler	●		●	●	●
	Intel® Integrated Performance Primitives[3]	●	●		●	●
	Intel® Math Kernel Library[3]	●	●	●	●	●
	Intel® Cilk™ Plus	●	●		●	●
	Intel® Threading Building Blocks	●	●		●	●
	Intel® Inspector XE	●	●	●	●	
	Intel® VTune™ Amplifier XE	●	●	●	●	
	Static Security Analysis	●	●	●	●	
	Intel® MPI Library				●	●
	Intel® Trace Analyzer & Collector				●	●
	Rogue Wave IMSL* Library[2]					
	Operating System[1]	W, L	W, L	W, L	W, L	W, L

Note: (1)[1] Operating System: W=Windows, L= Linux, M= Mac OS* X. (2)[2] Available in Intel® Visual Fortran Composer XE for Windows with IMSL*
(3)[3] Not available individually on Mac OS X, it is included in Intel® C++ & Fortran Composer XE suites for Mac OS X

Notices

Optimization Notice

Notice revision #20110804

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.