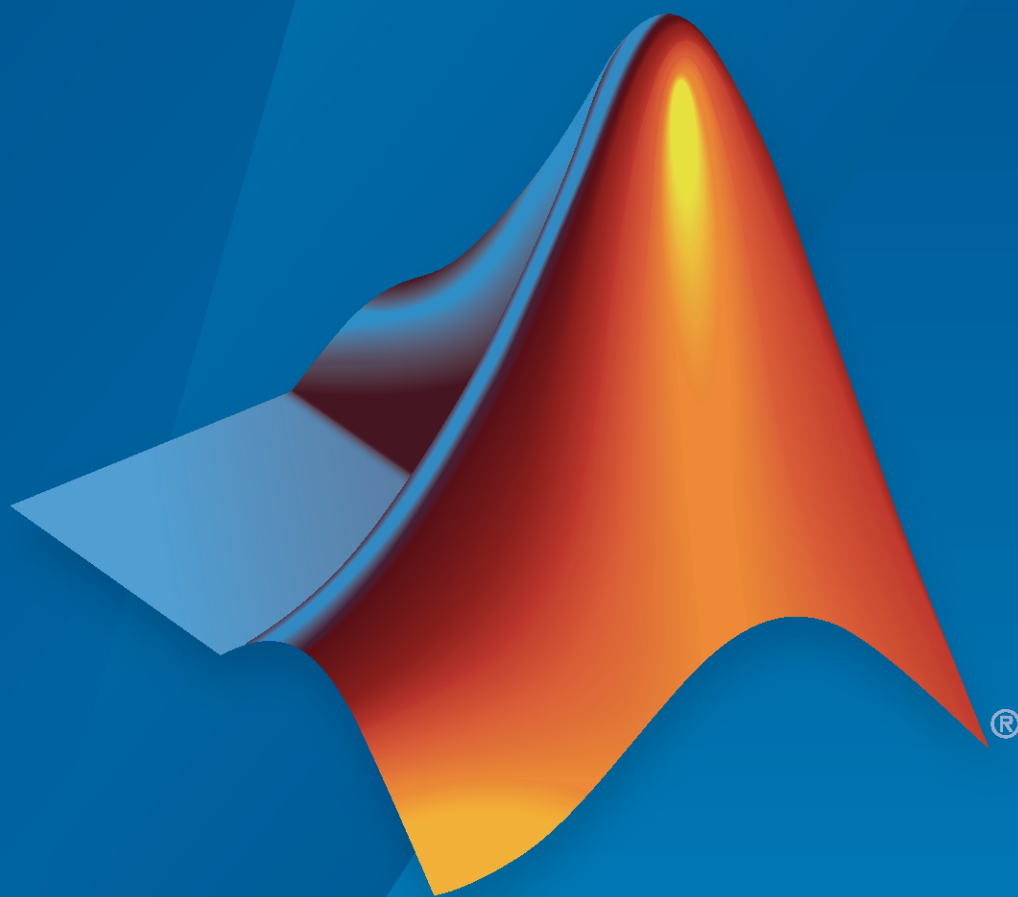


Polyspace® Code Prover™

Getting Started Guide



R2023a



How to Contact MathWorks



Latest news: www.mathworks.com
Sales and services: www.mathworks.com/sales_and_services
User community: www.mathworks.com/matlabcentral
Technical support: www.mathworks.com/support/contact_us



Phone: 508-647-7000



The MathWorks, Inc.
1 Apple Hill Drive
Natick, MA 01760-2098

Polyspace® Code Prover™ Getting Started Guide

© COPYRIGHT 2013–2023 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See www.mathworks.com/trademarks for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

Patents

MathWorks products are protected by one or more U.S. patents. Please see www.mathworks.com/patents for more information.

Revision History

September 2013	Online only	Revised for Version 9.0 (Release 2013b)
March 2014	Online Only	Revised for Version 9.1 (Release 2014a)
October 2014	Online Only	Revised for Version 9.2 (Release 2014b)
March 2015	Online Only	Revised for Version 9.3 (Release 2015a)
September 2015	Online Only	Revised for Version 9.4 (Release 2015b)
March 2016	Online Only	Revised for Version 9.5 (Release 2016a)
September 2016	Online Only	Revised for Version 9.6 (Release 2016b)
March 2017	Online Only	Revised for Version 9.7 (Release 2017a)
September 2017	Online Only	Revised for Version 9.8 (Release 2017b)
March 2018	Online Only	Revised for Version 9.9 (Release 2018a)
September 2018	Online Only	Revised for Version 9.10 (Release 2018b)
March 2019	Online Only	Revised for Version 10.0 (Release 2019a)
September 2019	Online Only	Revised for Version 10.1 (Release 2019b)
March 2020	Online Only	Revised for Version 10.2 (Release 2020a)
September 2020	Online Only	Revised for Version 10.3 (Release 2020b)
March 2021	Online Only	Revised for Version 10.4 (Release 2021a)
September 2021	Online Only	Revised for Polyspace Code Prover Version 10.5, Polyspace Code Prover Server Version 10.5, and Polyspace Code Prover Access Version 2.5 (Release 2021b)
March 2022	Online Only	Revised for Polyspace Code Prover Version 10.6, Polyspace Code Prover Server Version 10.6, and Polyspace Access Version 4.0 (Release 2022a)
September 2022	Online Only	Revised for Polyspace Code Prover Version 10.7, Polyspace Code Prover Server Version 10.7, and Polyspace Access Version 4.1 (Release 2022b)
March 2023	Online Only	Revised for Polyspace Code Prover Version 10.8, Polyspace Code Prover Server Version 10.8, and Polyspace Access Version 4.2 (Release 2023a)

Introduction

1

About This Getting Started Guide	1-2
Polyspace Code Prover Product Description	1-3

Code Prover Analysis on Desktop

2

Run Polyspace Code Prover on Desktop	2-2
Example Files	2-2
Run Polyspace in User Interface	2-3
Run Polyspace on Windows or Linux Command Line	2-4
Run Polyspace in Eclipse	2-5
Run Polyspace in MATLAB	2-6
Review Polyspace Code Prover Analysis Results	2-8
Example Files	2-8
Interpret Results	2-8
Address Results Through Bug Fix or Comments	2-10
Manage Results	2-11

Code Prover Analysis on Server

3

Quick Start Guide for Polyspace Server and Access Products	3-2
Installation	3-2
Setting Up Polyspace Analysis	3-3
Run Polyspace Code Prover on Server and Upload Results to Web	
Interface	3-6
Prerequisites	3-6
Check Polyspace Installation	3-7
Run Code Prover on Sample Files	3-7
Sample Scripts for Code Prover Analysis on Servers	3-9
Specify Sources and Options in Separate Files from Launching Scripts ...	3-9
Complete Workflow	3-10

View Assigned Results in Polyspace Access Web Interface	3-12
View Assigned Findings by Using the Polyspace Access Project Explorer and Dashboard	3-12
Triage and Assign Results in Polyspace Access Web Interface	3-14
Navigate the Polyspace Access Web Interface Dashboard	3-14
Navigate the Results List, Result Details, and Source Code Panels	3-16
Filter Polyspace Access Results	3-17
Assign Status and Owner to Results	3-19
Send Email Notifications with Polyspace Code Prover Server Results ..	3-20
Creating E-mail Notifications	3-20
Prerequisites	3-21
Export Results for E-mail Attachments	3-22
Assign Owners and Export Assigned Results	3-22

Offloading Code Prover Analysis from Desktop to Server

4

Send Code Prover Analysis from Desktop to Locally Hosted Server	4-2
Client-Server Workflow for Running Bug Finder Analysis	4-2
Prerequisites	4-3
Configure and Start Server	4-3
Configure Client	4-5
Send Analysis from Client to Server	4-5

Deploy Polyspace Code Prover

5

Source Code Verification with Polyspace Code Prover	5-2
How Polyspace Verification Works	5-2
Value of Polyspace Code Prover Verification	5-3
Polyspace Products and Software Development Workflows	5-5
Using Polyspace Products in Software Development	5-5
Coordinating Pre-Submit and Post-Submit Usage of Polyspace	5-6
Polyspace Products for Ada Code	5-7
Differences Between Polyspace Bug Finder and Polyspace Code Prover	5-8
How Bug Finder and Code Prover Complement Each Other	5-8
Workflow Using Both Polyspace Bug Finder and Polyspace Code Prover	5-14

Introduction

- “About This Getting Started Guide” on page 1-2
- “Polyspace Code Prover Product Description” on page 1-3

About This Getting Started Guide

This Getting Started Guide covers all Polyspace Code Prover products:

- Polyspace Code Prover™
- Polyspace Code Prover Server™
- Polyspace Access™

Depending on how you set up a Code Prover run, you might be running an analysis from one of these locations:

- **Desktop:** If you are running an analysis and reviewing the results on your desktop, you use Polyspace Code Prover. To get started, see “Code Prover Analysis on Desktop”.
- **Server:** If you are running an analysis on a server or reviewing the results from a server run on a web browser, you use:
 - Polyspace Code Prover Server to run the analysis.
 - Polyspace Access to host the analysis results (for review on a web browser).

To get started, see “Code Prover Analysis on Server”.

The Code Prover analysis engine underlies all Code Prover products. Chapters that do not mention a particular platform typically describe the underlying Code Prover analysis engine and apply to both platforms.

Polyspace Code Prover Product Description

Prove the absence of run-time errors in software

Polyspace Code Prover is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and other run-time errors in C and C++ source code. It produces results without requiring program execution, code instrumentation, or test cases. Polyspace Code Prover uses semantic analysis and abstract interpretation based on formal methods to verify software interprocedural, control, and data flow behavior. You can use it to verify handwritten code, generated code, or a combination of the two. Each code statement is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

Polyspace Code Prover displays range information for variables and function return values, and can prove which variables exceed specified range limits. Code verification results can be used to track quality metrics and check conformance with your software quality objectives. Polyspace Code Prover can be used with the Eclipse™ IDE to verify code on your desktop.

Support for industry standards is available through IEC Certification Kit (for IEC 61508 and ISO 26262) and DO Qualification Kit (for DO-178).

Code Prover Analysis on Desktop

- “Run Polyspace Code Prover on Desktop” on page 2-2
- “Review Polyspace Code Prover Analysis Results” on page 2-8

Run Polyspace Code Prover on Desktop

Polyspace Code Prover is a sound static analysis tool that proves the absence of overflow, divide-by-zero, out-of-bounds array access, and certain other run-time errors in C and C++ source code. A Code Prover analysis produces results without requiring program execution, code instrumentation, or test cases. Code Prover uses semantic analysis and abstract interpretation based on formal methods to determine control flow and data flow in the code. You can use Code Prover on handwritten code, generated code, or a combination of the two. In the analysis results, each operation is color-coded to indicate whether it is free of run-time errors, proven to fail, unreachable, or unproven.

You can run Code Prover on C/C++ code from the Polyspace user interface, in a supported development environment (IDE) such as Eclipse or using scripts. See:

- “Run Polyspace in User Interface” on page 2-3

If this is your first time using Polyspace, you might want to start from the Polyspace user interface. You can get help from features such as a project setup wizard, assisted configuration and summarized analysis log.

- “Run Polyspace on Windows or Linux Command Line” on page 2-4

Once you set up a project in the Polyspace user interface and complete a few trial runs, you can export the configuration to scripts that you run automatically or on-demand. You can also run a Polyspace analysis directly from the command line in your operating system. You can then save the commands in batch files (Windows) or shell scripts (Linux) for later runs. If you are running Polyspace Server products using continuous integration tools such as Jenkins, you can reuse your scripts from the Polyspace desktop products.

- “Run Polyspace in Eclipse” on page 2-5

Once you are familiar with running Polyspace from the command line, you can create menu items in your IDE that run your scripts and launch a Polyspace analysis in one click. In Eclipse and Eclipse-based IDEs, you can install a Polyspace plugin that does not require any additional setup at all. When you run Polyspace from the Eclipse plugin, the analysis configuration is created directly from your Eclipse project.

- “Run Polyspace in MATLAB” on page 2-6

If you have a MATLAB installation, it is particularly easy to write scripts to run a Polyspace analysis. You get all the benefits of scripting in the MATLAB environment, for instance, automatic help on function syntaxes. After analysis, you can create your own visualization of the results using MATLAB graphics and visualization tools.

Example Files

To follow the steps in this tutorial, copy the files `example.c` and `include.h` from `polyspaceroot\polyspace\examples\cxx\Code_Prover_Example\sources` to another folder. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2020a`.

Run Polyspace in User Interface

Open Polyspace User Interface

Double-click the `polyspace` executable in `polyspaceroot\polyspace\bin`. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2020a`. See also “Installation Folder”.

If you set up a shortcut to Polyspace on your desktop or the **Start** menu in Windows®, double-click the shortcut.

Add Source Files

To run a verification, you have to create a new Polyspace project. A Polyspace project points to source and include folders on your file system.

On the left of the **Start Page** pane, click **Start a new project**. Alternatively, select **File > New Project**.

After you provide a project name, on the next screens:

- Add your source folder.

In this tutorial, add the path to the folder in which you saved the file `example.c`. Click **Next**.

- Add your include folder.

In this tutorial, add the path to the folder in which you saved the file `include.h`. This folder can be the same as the previous folder. Click **Finish**.

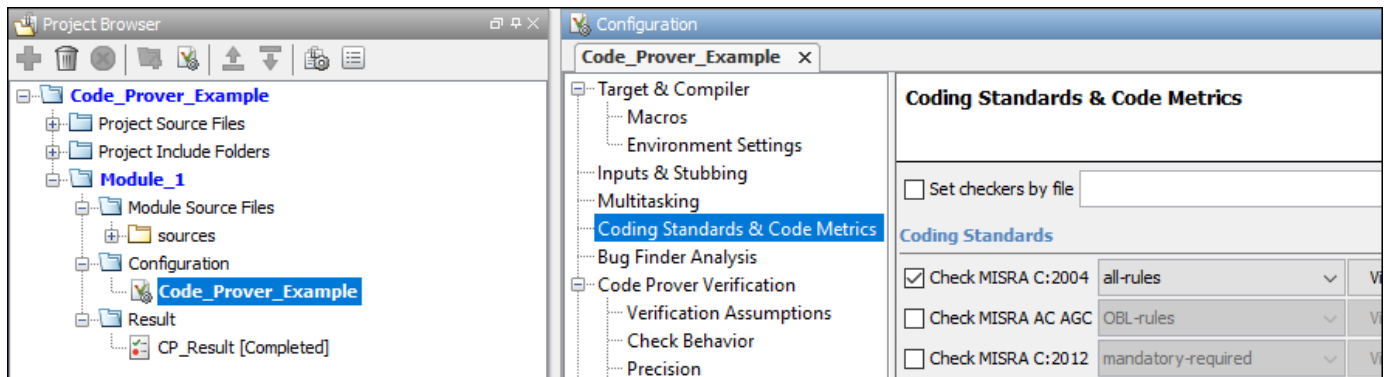
The screenshot shows a dialog box titled "Project - Properties" with a close button (X) in the top right corner. The main heading is "Define project". Below this, there are three input fields: "Project name" with the value "polyspace_project", "Version" with "1.0", and "Author" with "userName". A checkbox labeled "Use default location" is checked. Below it, the "Location" field contains the path "iangopa\Documents\Polyspace_Workspace\polyspace_project" and has a folder selection icon to its right. Under the "Project configuration" section, there are three unchecked checkboxes: "Use template", "Create from build command", and "Create from AUTOSAR specification". At the bottom of the dialog, there are four buttons: "Back", "Next", "Finish", and "Cancel".

After you finish adding your source and include folders, you see a new project on the **Project Browser** pane. Your source folders are copied to the first module in the project. You can right-click a project to add more folders later. If you add folders later, you must explicitly copy them to a module.

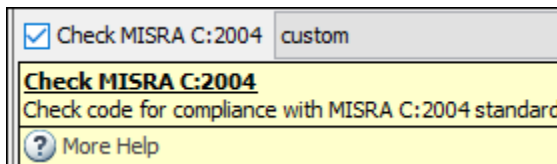
Configure and Run Polyspace

You can change the default options associated with a Polyspace analysis.

Click the **Configuration** node in your project module. On the **Configuration** pane, change options as needed. For instance, on the **Coding Rules & Code Metrics** node, select **Check MISRA C:2004**.



For more information, see the tooltip on each option. Click the **More help** link for context-sensitive help on the options.



To start verification, click **Run Code Prover** in the top toolbar. If the button indicates Bug Finder, click the arrow beside the button to switch to Code Prover.

Follow the progress of verification on the **Output Summary** window. After the verification, the results open automatically.

Additional Information

See:

- “Add Source Files for Analysis in Polyspace Desktop User Interface”
- “Run Analysis in Polyspace Desktop User Interface”

Run Polyspace on Windows or Linux Command Line

You can run Code Prover from the Windows or Linux® command line with batch (.bat) files or shell (.sh) scripts.

Use the `polyspace-code-prover` command to run a verification.

To save typing the full path to the command, add the path `polyspaceroot\polyspace\bin` to the `Path` environment variable on your operating system. Here, `polyspaceroot` is the Polyspace installation folder, for instance, `C:\Program Files\Polyspace\R2023a`.

Navigate to the folder where you saved the files (using `cd`). Enter the following:

```
polyspace-code-prover -sources example.c -I . -results-dir . -main-generator
```

Here, `.` indicates the current folder. The options used are:

- `-sources`: Specify comma-separated source files.
- `-I`: Specify path to include folder. Use the `-I` flag each time you want to add a separate include folder.
- `-results-dir`: Specify the path to the folder where Polyspace Code Prover results will be saved.

Note that the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

- `Verify module or library (-main-generator)`: Specify that a main function must be generated if not found in the source files

After verification, the results are saved in the file `ps_results.pscp`. You can open this file from the Polyspace user interface. For instance, enter the following:

```
polyspace ps_results.pscp
```

Instead of specifying comma-separated sources directly on the command line, you can list the sources in a text file (one file per line). Use the option `-sources-list-file` to specify this text file.

Additional Information

See:

- “Run Polyspace Analysis from Command Line”
- `polyspace-code-prover`

Run Polyspace in Eclipse

If you develop code in Eclipse or an Eclipse-based IDE, you can run Code Prover directly from your IDE.

After installing the Eclipse plugin, you can run Polyspace directly on the files in your Eclipse projects.

In the **Project Explorer** pane in Eclipse, select your project. To use Code Prover for the analysis, select **Polyspace > Code Prover**. To start the analysis, select **Polyspace > Run** (Ctrl + R).

After analysis, the results open automatically in Eclipse.

Additional Information

See “Run Polyspace Analysis on Eclipse Projects”.

Run Polyspace in MATLAB

Before you run Polyspace from MATLAB®, you must link your Polyspace and MATLAB installations. See “Integrate Polyspace with MATLAB and Simulink”.

To run an analysis, use a `polyspace.Project` object. The object has two properties:

- **Configuration:** Specify the analysis options such as sources, includes, compiler and results folder using this property.
- **Results:** After analysis, read the analysis results to a MATLAB table using this property.

To run the analysis, use the `run` method of this object.

To run Polyspace on the example file `example.c` in `polyspaceroot\polyspace\examples\cxx\Code_Prover_Examples\sources`, enter the following at the MATLAB command prompt.

```
proj = polyspace.Project

% Configure analysis
proj.Configuration.Sources = {fullfile(polyspaceroot, 'polyspace', 'examples', ...
    'cxx', 'Code_Prover_Example', 'sources', 'example.c')};
proj.Configuration.EnvironmentSettings.IncludeFolders = {fullfile(polyspaceroot, ...
    'polyspace', 'examples', 'cxx', 'Code_Prover_Example', 'sources')}
proj.Configuration.TargetCompiler.Compiler = 'gnu4.9';
proj.Configuration.ResultsDir = fullfile(pwd, 'results');
proj.Configuration.CodeProverVerification.MainGenerator = true;

% Run analysis
cpStatus = proj.run('codeProver');

% Read results
resObj = proj.Results;
cpSummary = getSummary(resObj, 'runtime');
cpResults = getResults(resObj, 'readable');
```

After verification, the results are saved in the file `ps_results.pscp`. You can open this file from the Polyspace user interface. For instance, enter the following:

```
resultsFile = fullfile(proj.Configuration.ResultsDir, 'ps_results.pscp');
polyspaceCodeProver(resultsFile)
```

Additional Information

See:

- “Run Polyspace Analysis by Using MATLAB Scripts”
- `polyspace.Project`
- `polyspace.Project.Configuration` Properties

See Also

Related Examples

- “Review Polyspace Code Prover Analysis Results” on page 2-8
- “Run Polyspace Analysis on Code Generated with Embedded Coder”

Review Polyspace Code Prover Analysis Results

Polyspace Code Prover checks C/C++ code exhaustively and proves the absence of certain types of run-time errors (static analysis or verification). Whatever means you use for running the analysis, afterwards, you open the results in the Polyspace user interface (or if you ran the analysis in Eclipse, the results open in Eclipse).


Example Files

To follow the steps in this tutorial, run Polyspace using the steps in “Run Polyspace Code Prover on Desktop” on page 2-2. Alternatively, in the Polyspace user interface, open example results using **Help > Examples > Code_Prover_Example.psprj**. If you have loaded the example results earlier and made some changes, to load a fresh copy, select **Help > Examples > Restore Default Examples**.

Interpret Results

Review each Polyspace result. Find the root cause of the issue.

Start from the list of results on the **Results List** pane.

- If the **Results List** pane covers the entire window, select **Window > Reset Layout > Results Review**.
- If you do not see a flat list of results, but instead see them grouped, from the  list, select **None**.

Click the **Family** column header to sort the results based on how critical they are. Select the red **Illegally dereferenced pointer** check in the file `example.c`. A red check indicates that the error happens on all execution paths considered in the analysis.

The screenshot displays the Polyspace interface with three main panes:

- Results List:** A table showing analysis results. The selected entry is:

Family	Check	File
*	Illegally dereferenced pointer	example.c
- Result Details:** Shows the error description:

Illegally dereferenced pointer
 Error: pointer is outside its bounds
 Dereference of local pointer 'p' (pointer to int 32, size: 32 bits):
 Pointer is not null.
 Points to 4 bytes at offset 400 in buffer of 400 bytes, so is outside bounds.
 Pointer may point to variable or field of variable:
 'array', local to function 'Pointer_Arithmetic'.

 Below the description is an event table:

Event	File	Scope	Line
1 Entering function 'RTE'	main.c	main()	38
2 Entering function 'Pointer_Arithmetic'	example.c	RTE()	218
3 Error: pointer is outside its bounds	example.c	Pointer_Arithmetic()	101
- Source:** Shows the C code snippet:


```

92 int i, *p = array;
93
94 for (i = 0; i < 100; i++) {
95     *p = 0;
96     p++;
97 }
98
99 if (get_bus_status() >= 0) {
100     if (get_oil_pressure() >= 0) {
101         *p = 5; /* Out of bounds */
102     } else {
103         i++;
104     }
105 }
      
```

See the source code on the **Source** pane and further information about the result on the **Result Details** pane.

For the **Illegally dereferenced pointer** result, the message on the **Result Details** pane indicates that the pointer `p` has an allowed buffer of 400 bytes. It points to a location that begins at 400 bytes from the beginning of the buffer and points to a data type of 4 bytes.

To investigate further and find the root cause of the issue, right-click the variable `p` on the **Source** pane and select **Search For All References**. Click each search result to navigate to the corresponding location on the source code. At each location, place your cursor on the variable `p` to see a tooltip that describes the variable value at that point in the code.

The screenshot shows a tooltip for the variable `p` in the source code. The tooltip text is:

Local pointer 'p' (pointer to int 32, size: 32 bits):
 Pointer is not null.
 Points to 4 bytes at offset multiple of 4 in [0 .. 396] in buffer of 400 bytes, so is within bounds (if memory is allocated).
 Pointer may point to variable or field of variable:
 'array', local to function 'Pointer_Arithmetic'.

Press 'F2' for focus

You see that the pointer variable `p` is initialized to a 100-element `int` array. The pointer traverses the array in a `for` loop with 100 iterations and points to the end of the array. On the line with the red **Illegally dereferenced pointer** check, this pointer is dereferenced, resulting in dereference of a memory location outside the array.

Additional Information

See:

- “Interpret Code Prover Results in Polyspace Desktop User Interface”
- “Code Prover Result and Source Code Colors”
- “Complete List of Polyspace Code Prover Results”

Address Results Through Bug Fix or Comments

Once you understand the root cause of a Polyspace finding, you can fix your code. Otherwise, add comments to your Polyspace results to fix the code later or to justify the result. You can use the comments to keep track of your review progress.

Right-click the variable `p` on the **Source** pane. Select **Open Editor**. The code opens in a text editor. Fix the issue. For instance, you can make the pointer point to the beginning of the array after the `for` loop. Changes to the code are highlighted below.

```
...
int i, *p = array;

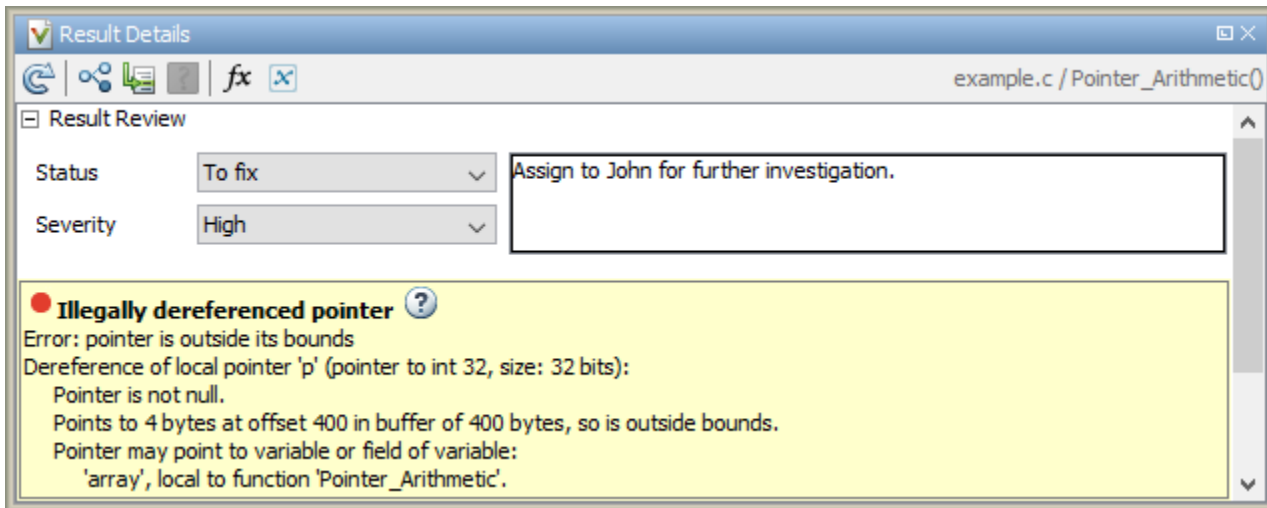
for (i = 0; i < 100; i++) {
    *p = 0;
    p++;
}

p = array;

if (get_bus_status() > 0)
    ...
```

If you rerun the analysis, you do not see the red **Illegally dereferenced pointer** check.

Alternatively, if you do not want to fix the defect immediately, assign a status **To investigate** to the result. Optionally, add comments with further explanation.



If you assign a status **No action planned**, the result does not appear in subsequent runs on the same project.

Additional Information

See:


- “Address Results in Polyspace User Interface Through Bug Fixes or Justifications”
- “Annotate Code and Hide Known or Acceptable Results”

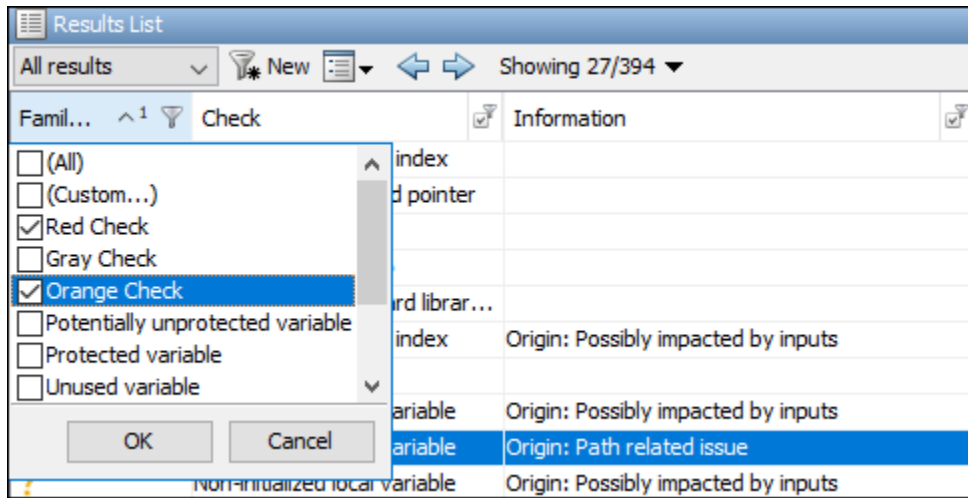
Manage Results

When you open the results of a Code Prover analysis, you see a list of run-time checks, coding rule violations or other results. To organize your review, you can narrow down the list or group results by file or result type.

For instance, you can:

- Review only red and critical orange checks.

Click the **Family** column header to sort checks by color. Alternatively, you can filter out results other than red and orange checks. To begin filtering, click the  icon on the column header.



You can review only the path-related orange checks because they are likely to be more critical. To filter out other checks, use the filters on the **Information** column. Clear the **All** filter and then select the filter **Origin: Path related issue**.

- Review only the new results since the last analysis.

On the **Results List** pane toolbar, click the **New** button.

- Review results in certain files or functions.

On the **Results List** pane toolbar, from the  list, select **File**.

Additional Information

See:

- “Filter and Group Results in Polyspace Desktop User Interface”
- “Prioritize Check Review”

Code Prover Analysis on Server

Quick Start Guide for Polyspace Server and Access Products

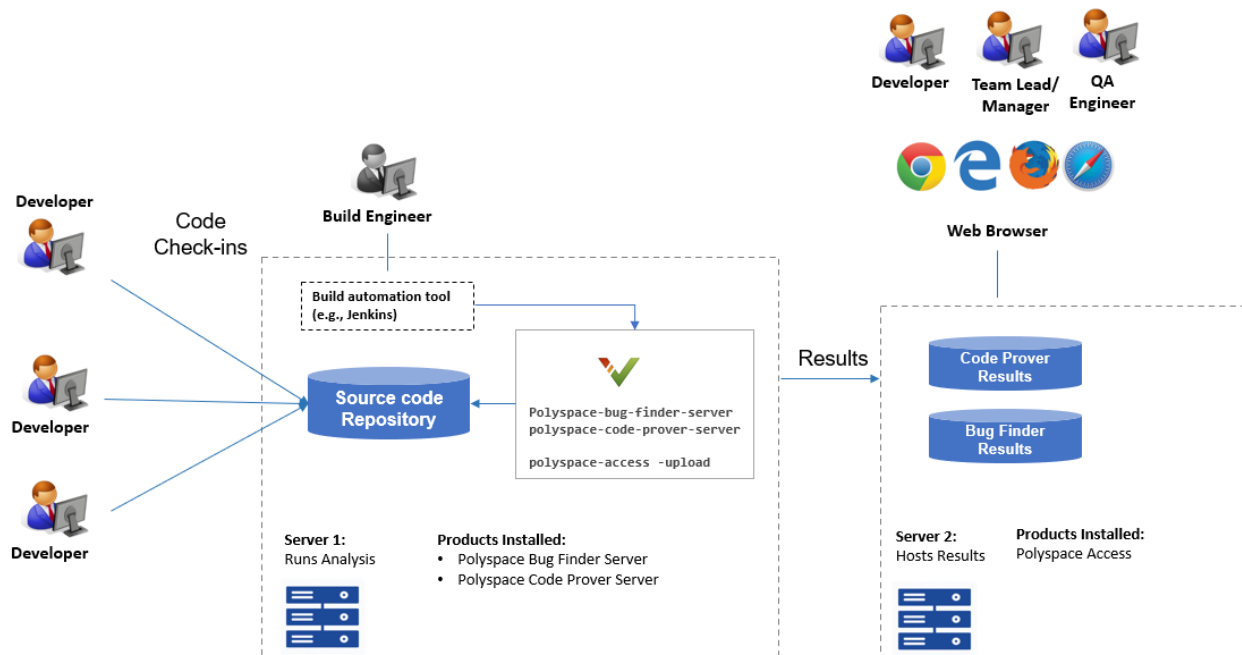
To avoid finding bugs late in the development process, run static analysis by using Polyspace products.

- **Polyspace Bug Finder™** checks C/C++ code for bugs, coding standard violations, security vulnerabilities, and other issues.
- **Polyspace Code Prover** performs exhaustive checks for divide by zero, overflow, array access out of bounds, and other common types of run-time errors.

See also “Differences Between Polyspace Bug Finder and Polyspace Code Prover” on page 5-8.

If you run Polyspace checkers regularly as part of continuous integration, you can protect against regressions from new code check-ins. To run Polyspace on a server during continuous integration, use **Polyspace Bug Finder Server** and **Polyspace Code Prover Server**. To host the Polyspace analysis results, use **Polyspace Access**.

A typical workflow looks like this figure.



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

Installation

Prerequisites

Depending on the needs of your project, team or organization, you have decided to obtain a certain number of licenses of Polyspace Server and Polyspace Access products. This guide helps you to install individual instances of these products on a machine.

Install Polyspace Server

To install Polyspace Server products, download and run the MathWorks installer. Enter a license for the Polyspace Server products (or request a trial license). See also Request a Trial License. The Polyspace Server products are installed in a separate folder from other MathWorks products. See also “Install Polyspace Server and Access Products”.

Install Polyspace Access

Before installing Polyspace Access, consider the number of users who will potentially review Polyspace results simultaneously. The system requirements depend on the number of simultaneous reviewers. See also “System Requirements for Polyspace Access”.

Polyspace Access consists of several services: a user manager to authenticate user logins, an issue tracker to integrate your bug tracking tool with Polyspace, a database to manage results, a web server to show results, and a gateway to handle communications. The services are deployed in Docker containers. You can start the services from a common interface called the Cluster Admin.

To install Polyspace Access:

- Download the installer as a zip file.
- Unzip the file and start the Cluster Admin. From the Cluster Admin interface, start the various services. See “Install Polyspace Access for Web Reviews”.

After installation, to see uploaded results, you and other reviewers can log in to:

`https://hostName:portNumber/metrics/index.html`

Install Network License Manager

Both Polyspace Server and Polyspace Access use licenses that require communication with a network license manager for license checkouts.

- To install, configure and start the network license manager for Polyspace Server, see “Administer Network Licenses”.
- To install, configure and start the network license manager for Polyspace Access, see “Manage Polyspace NNU Licenses”.

Setting Up Polyspace Analysis

Prerequisites

You or your IT department in your organization must install the required number of Polyspace Server and Polyspace Access instances. This guide helps you to set up a Polyspace analysis as part of continuous integration using a single instance of Polyspace Server and Polyspace Access.

To check that your Polyspace Server and Polyspace Access installations can communicate with each other, see “Check Polyspace Installation”.

Run Polyspace Server and Upload Results to Polyspace Access

You can run the Polyspace Server products at the command line of your operating system:

- To run the analysis, use the `polyspace-bug-finder-server` and `polyspace-code-prover-server` executables.

- To upload analysis results, use the `polyspace-access` executable. You can also use this executable to export the results from Polyspace Access as text files for archiving or email attachments.

You can run all Polyspace executables from the `polyspace/bin` subfolder of the Polyspace installation folder (for instance, `/usr/local/Polyspace Server/R2023a`, see also “Installation Folder”). To start running Polyspace Server by using sample C source files and sample scripts, see:

- “Run Polyspace Code Prover on Server and Upload Results to Web Interface” on page 3-6
- “Send Email Notifications with Polyspace Code Prover Server Results” on page 3-20

You can also preconfigure the Polyspace analysis options from your build command (makefile), and then append a second options file with analysis specifications such as checkers. See “Create Polyspace Analysis Configuration from Build Command (Makefile)”.

If you have an installation of the Polyspace desktop products, you can prepare the analysis configuration in the user interface of the desktop products. You can then generate Polyspace options files to run during continuous integration. See “Configure Polyspace Analysis Options in User Interface and Generate Scripts”.

Include Polyspace Runs in Continuous Integration by Using Tools Such as Jenkins

Once you have working scripts to run a Polyspace analysis, you can run those scripts at predefined intervals using continuous integration tools such as Jenkins and Bamboo. In Jenkins, you can use a Polyspace plugin to point to your Polyspace installations and send email notifications to developers after the analysis, based on criteria such as new defects.

From within the Jenkins interface, search for and install the Polyspace plugin. For a quick start on using the Jenkins plugin and sample scripts, see the Polyspace plugin GitHub repository. For the full workflow with Jenkins, see “Sample Scripts for Polyspace Analysis with Jenkins”.

Create a Workflow for Result Reviewers

Depending on tools that you already use, you can set up a convenient workflow for result reviewers. For example:

Reviewers receive alerts for new results and log into Polyspace Access

- When new results are available, the continuous integration tool alerts a group of users. The email alert contains the Polyspace Access URL of the project where the results are uploaded.
- In the Polyspace Access interface, a reviewer can open this project URL, filter results based on files, and fix the issues or set a status for the results. See also:
 - “Filter and Sort Results in Polyspace Access Web Interface”
 - “Address Results in Polyspace Access Through Bug Fixes or Justifications”

Reviewers get customized email alerts with results in attachment

- Before upload to Polyspace Access, using the `-set-unassigned-findings` option of the `polyspace-access` executable, the continuous integration (CI) tool assigns owners to new analysis results based on file or component ownership or another criteria.
- After upload, using the `-export` option of the `polyspace-access` executable, the CI tool exports analysis results for each owner to a separate text file. The tool then sends the text file in

an email attachment to the owner. The text file contains results with the corresponding URLs in the Polyspace Access interface.

If you use Jenkins as your CI tool, the Polyspace plugin in Jenkins directly supports this workflow. See “Sample Scripts for Polyspace Analysis with Jenkins”.

- On receiving the email, the owner opens the attached text file, copies the URL of each result to their web browser and reviews the result.

Reviewers open tickets from bug tracking tools

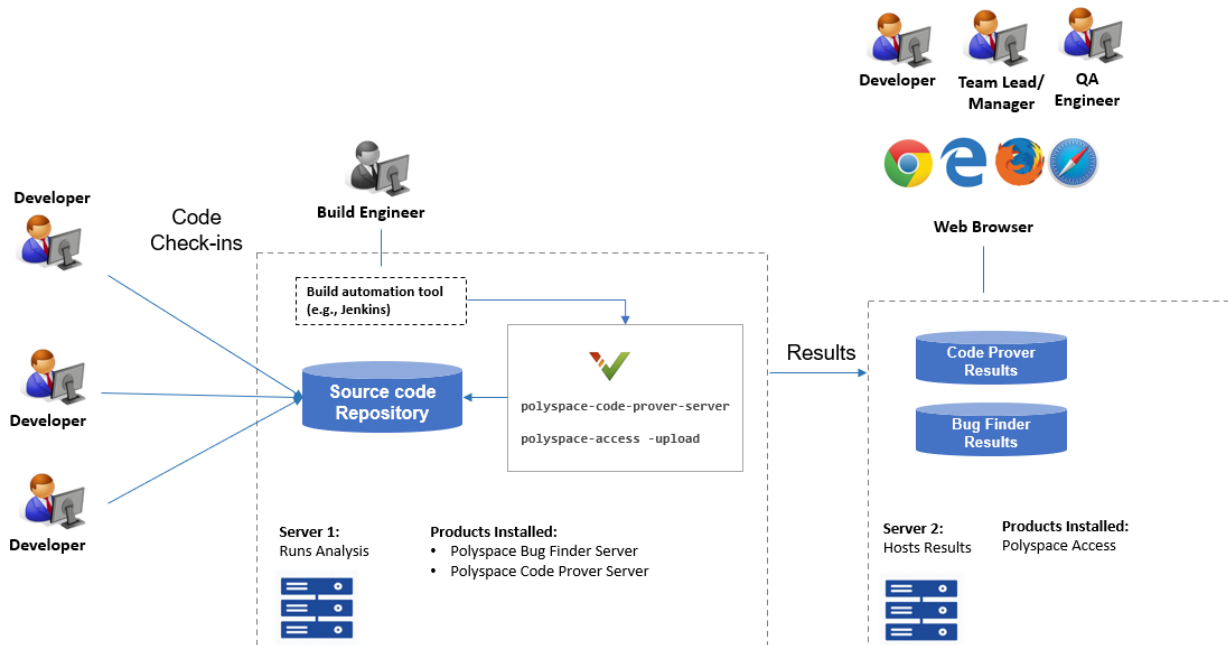
- A reviewer, such as a quality engineer, reviews all new results and creates JIRA tickets for developers. See “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface”.
- Developers open each JIRA ticket and navigate to the corresponding Polyspace result in the Polyspace Access interface.

Run Polyspace Code Prover on Server and Upload Results to Web Interface

Polyspace Code Prover Server proves the absence of run-time errors in C/C++ code, and then uploads findings to a web interface for code review.

You can run Code Prover as part of continuous integration. Set up scripts that run a Code Prover analysis at regular intervals or based on new code submissions. The scripts can upload the analysis results for review in the Polyspace web interface and optionally send emails to owners of source files with Polyspace findings. The owners can open the web interface to review only the new findings from their submission, and then fix or justify the issues.

In a typical project or team, Polyspace Code Prover Server runs periodically on a few testing servers and uploads the results for review. Each developer and quality engineer in the team has a Polyspace Access license to view the results in the web interface for investigation and bug fixing.



Note: Depending on the specifications, the same computer can serve as both Server 1 and Server 2.

Prerequisites

To run a Code Prover analysis on a server and review the results in the Polyspace Access web interface, perform this one-time setup:

- To run the analysis, install one instance of the Polyspace Code Prover Server product.
- To upload results, set up the components required to host the web interface of Polyspace Access.
- To view the uploaded results, you and each developer reviewing the results must have a Polyspace Access license.

See “Install Polyspace Server and Access Products”.

Check Polyspace Installation

To check if Polyspace Code Prover Server is installed:

- 1 Open a command window. Navigate to *polyspaceserverroot*\polyspace\bin. Here, *polyspaceserverroot* is the Polyspace Code Prover Server installation folder, for instance, C:\Program Files\Polyspace Server\R2023a. See also “Installation Folder”.
- 2 Enter:

```
polyspace-code-prover-server -help
```

You should see the list of options allowed for a Code Prover analysis.

To check if the Polyspace web interface is set up for upload:

- 1 Navigate again to *polyspaceserverroot*\polyspace\bin.
- 2 Enter:

```
polyspace-access -host hostName -port portNumber -create-project testProject
```

Here, *hostName* is the name of the server hosting the Polyspace Access web server. For a locally hosted server, use localhost. *portNumber* is the optional port number of the server. If you omit the port number, 9443 is used.

If the setup was complete, a project called `testProject` is created in the Polyspace web interface.

You are prompted for your login and password each time you use the `polyspace-access` command. To avoid entering login information each time, provide the login and an encrypted version of your password with the command. To create an encrypted password, enter:

```
polyspace-access -encrypt-password
```

Enter your login and password. Copy the encrypted password and provide this encrypted password with the `-encrypted-password` option when using the `polyspace-access` command.

- 3 In a web browser, open this URL:

```
https://hostName:portNumber/metrics/index.html
```

Here, *hostName* and *portNumber* are the host name and port number from the previous step.

In the **Project Explorer** pane on the Polyspace web interface, you should see the newly created project `testProject`.

Run Code Prover on Sample Files

To run Code Prover, in your operating system, open a command window.

- 1 To run a Code Prover analysis, use the `polyspace-code-prover-server` command.

- 2 To upload the results to the Polyspace web interface, use the `polyspace-access` command.

To avoid typing the full path to the command, add the path `polyspaceserverroot\polyspace\bin` to the Path environment variable on your operating system.

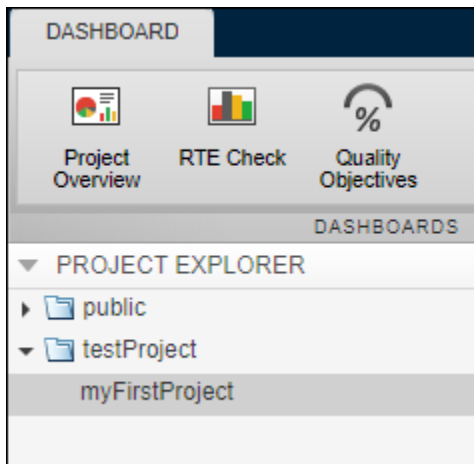
Try out the commands on sample files provided with your Polyspace installation.

- 1 Copy the sample source files from `polyspaceserverroot\polyspace\examples\cxx\Code_Prover_Example\sources` to another folder where you have write permissions. Navigate to this folder at the command line.
- 2 Enter:


```
polyspace-code-prover-server -sources example.c,single_file_analysis.c
-I . -main-generator -results-dir .
polyspace-access -host hostName -port portNumber
  -login username -encrypted-password pwd
  -create-project testProject
polyspace-access -host hostName -port portNumber
  -login username -encrypted-password pwd
  -upload . -project myFirstProject -parent-project testProject
```

Here, *username* is your login name and *pwd* is the encrypted password that you created previously. See “Check Polyspace Installation” on page 3-7.

Refresh the Polyspace web interface. You see the newly uploaded results under the `testProject` folder in the **Project Explorer** pane.



To see the results in the project, click **Review**. For more information, see “Review Polyspace Code

Prover Results in Web Browser”. You can also access the documentation using the  button in the upper right of the Polyspace Access interface.

The options used with the `polyspace-code-prover-server` command are:

- `-sources`: Specify comma-separated source files.
- `-I`: Specify path to include folder. Use the `-I` flag each time you want to add a separate include folder.
- `-results-dir`: Specify the path to the folder where Polyspace Code Prover results will be saved.

Note that the results folder is cleaned up and repopulated at each run. To avoid accidental removal of files during the cleanup, instead of using an existing folder that contains other files, specify a dedicated folder for the Polyspace results.

- **Verify module or library (-main-generator):** Specify that a main function must be generated if not found in the source files

For the full list of options available for a Code Prover analysis, see “Complete List of Polyspace Code Prover Analysis Options”. To open the Code Prover documentation in a help browser, enter:

```
polyspace-code-prover-server -doc
```

Sample Scripts for Code Prover Analysis on Servers

To run the analysis, instead of typing the commands at the command line, you can use scripts. The scripts can execute each time that you add or modify source files.

A sample Windows batch file is shown below. Here, the path to the Polyspace installation is specified in the script. To use this script, replace `polyspaceserverroot` with the path to your installation. You must have already generated the encrypted password for use in the scripts. See “Check Polyspace Installation” on page 3-7.

```
echo off
set POLYSPACE_PATH=polyspaceserverroot\polyspace\bin
set LOGIN=-host hostName -port portNumber -login username -encrypted-password pwd
"%POLYSPACE_PATH%\polyspace-code-prover-server"
-sources example.c,single_file_analysis.c -I .^
-main-generator -results-dir .
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -create-project testProject
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -upload . -project myFirstProject
-parent-project testProject
pause
```

A sample Linux shell script is shown below.

```
echo off
set POLYSPACE_PATH=polyspaceserverroot\polyspace\bin
set LOGIN=-host hostName -port portNumber -login username -encrypted-password pwd
"%POLYSPACE_PATH%\polyspace-code-prover-server"
-sources example.c,single_file_analysis.c -I .^
-main-generator -results-dir .
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -create-project testProject
"%POLYSPACE_PATH%\polyspace-access" %LOGIN% -upload . -project myFirstProject
-parent-project testProject
pause
```

Specify Sources and Options in Separate Files from Launching Scripts

Instead of listing the source files and analysis options within the launching scripts, you can list them in separate text files.

- Specify the text file listing the sources by using the option `-sources-list-file`.
- Specify the text file listing the analysis options by using the option `-options-file`.

By removing the source files and analysis option specifications from the launching scripts, you can modify these specifications as required with new code submissions while leaving the launching script untouched.

Consider the script in the preceding example. You can modify the `polyspace-code-prover-server` command to use text files with sources and options. Instead of:

```
polyspace-code-prover-server -sources example.c,single_file_analysis.c -I .  
-main-generator -results-dir .
```

use:

```
polyspace-code-prover-server -sources-list-file sources.txt  
-options-file polyspace_opts.txt -results-dir .
```

Here:

- `sources.txt` lists the source files:

```
example.c  
single_file_analysis.c
```
- `polyspace_opts.txt` lists the analysis options in separate lines:

```
-I .  
-main-generator
```

Typically, your source files are specified in a build command (makefile). Instead of specifying the source files directly, you can trace the build command to create a list of source specifications. See `polyspace-configure`.

Complete Workflow

In a typical continuous integration workflow, you run a script that executes these steps:

- 1 Extract Polyspace options from your build command.

For instance, if you use makefiles to build your source code, you can extract analysis options from the makefile. The command below first executes `make` and then determines the analysis options from the processes executed.

```
polyspace-configure -output-options-file compile_opts make
```

See also:

- `polyspace-configure`
 - “Create Polyspace Analysis Configuration from Build Command (Makefile)”
- 2 Run the analysis with the previously created options file. Append a second options file that contains the remaining options required for the analysis.

```
polyspace-code-prover-server -options-file compile_opts -options-file run_opts
```

See “Specifying Multiple Options Files”.

- 3 Upload the results to Polyspace Access.


```
polyspace-access login -upload resultsFolder -project projName  
-parent-project parentProjName
```

Here, *login* is the combination of options required to communicate with the web server that is hosting Polyspace Access:

```
-host hostName -port portNumber -login username -encrypted-password pwd
```

resultsFolder is the folder containing the Polyspace results. *projName* and *parentProjName* are names of the project and parent folder as they would appear in the Polyspace Access web interface.

- 4 Optionally, send email notifications to developers with new results from their code submission. The email contains attachments with links to the results in the Polyspace Access web interface.

See “Send Email Notifications with Polyspace Code Prover Server Results” on page 3-20.

See examples of scripts executing these steps in “Sample Scripts for Polyspace Analysis with Jenkins”.

See Also

polyspace-access | polyspace-code-prover-server

More About

- “Send Email Notifications with Polyspace Code Prover Server Results” on page 3-20
- “Send Code Prover Analysis from Desktop to Locally Hosted Server” on page 4-2
- “Complete List of Polyspace Code Prover Analysis Options”

View Assigned Results in Polyspace Access Web Interface

In a typical collaborative review workflow, results in Polyspace Access are assigned to individuals for further analysis, fixing, or justification. Three common ways to access your assigned results are:

- You receive a direct link to a finding or set of findings.
- You open Polyspace Access and navigate to your assigned results from the **Project Overview** dashboard.
- You open Polyspace Access and navigate to your assigned results using the **Review** button in the taskbar and the **Assigned to me** filter from the filters drop-down list.

If you receive a link to your results, click the link to view your assigned results. Links are commonly sent through email or a bug tracking tool ticket.

View Assigned Findings by Using the Polyspace Access Project Explorer and Dashboard

Before you start reviewing your assigned results in Polyspace Access, make sure that:

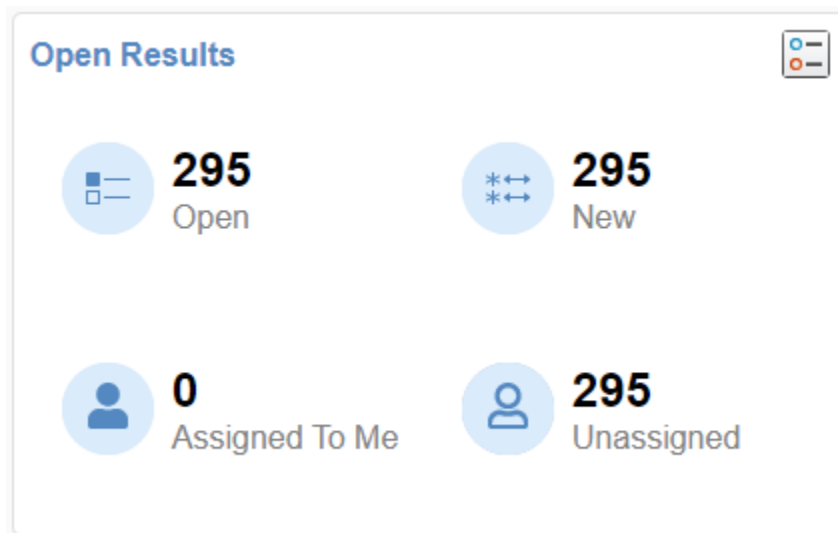
- You have a valid login for Polyspace Access.
- You know the URL for your company's Polyspace Access Web Interface. If you do not know the URL, contact your Polyspace Access administrator.
- The results must be uploaded to Polyspace Access.

To view your assigned findings:

- 1** Log into your company's Polyspace Access Web Interface by using a web browser.
- 2** Open the **Project Explorer** on the left side and select your project run. Projects are listed in a file-folder organization system. A project folder can contain additional sub-folders or individual project runs. After you select your results, the **Project Overview** dashboard opens, displaying your results.

If you select the folder, the dashboard shows an aggregate of the statistics for all the project runs in that folder.

- 3** Click **Assigned To Me** on the Summary card of the **Project Overview** dashboard. The **REVIEW** window opens and shows the **Results List** filtered to all the results that are assigned to you. If you select a folder instead of a project run, **Assigned To Me** is not clickable.



You can reassign your results to another user if needed. See “Triage and Assign Results in Polyspace Access Web Interface” on page 3-14. At this point, all results assigned to you are displayed and you can begin investigating your results.

See Also

More About

- “Create Custom Filter Groups in Polyspace Access Web Interface”
- “Review Polyspace Code Prover Results in Web Browser”
- “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface”

Triage and Assign Results in Polyspace Access Web Interface

Polyspace Access offers a centralized database where you can store Polyspace analysis results for sharing with your team and performing collaborative reviews. Once analysis results are uploaded to Polyspace Access, a common next step is to filter and assign results to team members. Use the Polyspace Access dashboards and links to access results. Use filters to review and sort the results that you want to assign.

- 1 Log into the Polyspace Access web interface by using a web browser.
- 2 Open the **Project Explorer** on the left side and select your project run. Projects are listed in a file-folder organization system. A project folder can contain additional sub-folders or individual project runs. You can use the filter at the top of the **Project Explorer** to search for uploaded results. After you select your results, the **Project Overview** dashboard opens, displaying your results.

If you select a folder, the dashboard shows an aggregate of the statistics for all the project runs in that folder.

Navigate the Polyspace Access Web Interface Dashboard

After you select your project run in the **Project Explorer**, the **Project Overview** dashboard opens for those results. The **Project Overview** dashboard shows a snapshot of the project including:

- What findings currently exist.
- The type and status of the findings.
- Tracking of open findings over time.

The dashboard is split into multiple sections:

- **Summary**

The **Summary** is the main section of the **Project Overview** dashboard and shows a snapshot of the project. This section contains cards showing **Open Issues**, **Code Metrics**, **Quality Objectives**, **Defects** (Bug Finder only), **Run-time Checks** (Code Prover only), and **Coding Standards**.

- **Trends**

The **Trends** section uses a graph to show defects over time.

- **Details**

The **Details** section enables users to take a closer look at the project in a table. This table shows the total number of coding standards violations and their status. The status and number of defects is shown (Bug Finder only). The status and number of global variables and red, gray, orange, and green checks is shown (Code Prover only). Click any table entry to view the corresponding results in the **Results List**.

Summary Section Overview

The Summary section contains the cards listed in this table:

Card	Description
Open Issues	Shows the total number of open issues, new issues compared to the previous run, the number of open issues that are assigned to the current user, and the total number of unassigned issues. Click any of these links to view the corresponding results in the Results List .
Code Metrics	Shows the total number of subprojects, number of files, number of lines without comments, and the biggest cyclomatic complexity value of the code. Click the “Code Metrics” link to open the Code Metrics dashboard in a new tab. See “Code Metrics Dashboard in Polyspace Access Web Interface”
Quality Objectives	Shows the completion percentage of all quality objectives and the remaining quality objectives as defined by the current threshold. A label next to the percentage bar shows the analysis status. For example, the label reads Incomplete if checkers required for the selected threshold were not activated in the analysis. Click the “Quality Objectives” link to open the Quality Objectives dashboard in a new tab. You can create user-defined thresholds for quality objectives. See “Quality Objectives Dashboard in Polyspace Access”
Defects (Bug Finder only)	Shows the current number of open defects along with their status. Click the “Defects” link to open the Defects dashboard in a new tab. This dashboard shows a more detailed breakdown of all open defects and provides the ability to view defects by category or by file. See “Defects”
Run-time Checks (Code Prover Only)	Shows the current number of red, orange, gray, and green run-time checks. Click the "Run-time Checks" link to open the Run-time Checks dashboard in a new tab. This dashboard shows a more detailed breakdown of open run-time checks and provides the ability to view run-time checks by category or by file. See “Run-Time Checks”

Card	Description
Coding Standards	<p>Shows the current number of open coding standard issues and their status. Click the “Coding Standards” link to open dashboards for the coding rules in new tabs. These dashboards can include the Custom Rules dashboard, the Guidelines dashboard, and the dashboards for whichever coding standards are activated for the project such as MISRA C:2012 or SEI CERT C. See “Coding Standards”</p> <p>The different coding standards dashboards enable you to view a more detailed breakdown of all open coding standard issues including the ability to view coding standard issues by category or by file.</p>

Clicking any link within the tables takes you to the **REVIEW** page with the relevant filters applied.

Navigate the Results List, Result Details, and Source Code Panels

In many cases, clicking a link on the **Project Overview** dashboard opens the **REVIEW** page. The **REVIEW** page is separated into three major panes:

- **Results List**
- **Result Details**
- **Source Code**

To view additional panes available in the Polyspace user interface, including **Review History** and **Call Hierarchy**, on the toolstrip, click **Window** and select a pane. See also “Interpret Results”.

Results List

The **Results List** contains all the results matching the filters that are set. No other issues are displayed unless you remove these filters. Click the pink eraser icon next to the filters to remove all filters. Place your cursor over an individual filter to open the option to remove the filter.

The **Results List** is organized in a table format. You can sort each column by clicking the column title. You can further filter results at an item level. For the item that you want to filter, right-click the row of the item in the column you want to filter by. This shows the options to filter out or show only the value in the cell. You can also set the **Show only** and **Filter out** values in the **Filters** section on the toolstrip. See also “Results List in Polyspace Access Web Interface”

Result Details

Result Details shows detailed information about individual results, including additional information about the result, links to relevant documentation, and review information such as status, severity, and comments. Select a result in the **Results List** to display the result information in the **Result Details** pane. See “Result Details in Polyspace Access Web Interface”

When applicable, the trace of events shows the events that lead to the error. Click the event to highlight the relevant line of code in the **Source Code** pane.

You can also create a bug tracking ticket and assign an owner to a result. See “Assign Status and Owner to Results” on page 3-19

Source Code

The **Source Code** pane shows the location of the result in the source code. You cannot make edits in the **Source Code** pane. Select a result in the **Results List** to see it in the **Source Code** pane. Right-click in the **Source Code** pane to:

- Quickly navigate to a line in the file.
- Search for all references of a variable.
- Copy the file path to your clipboard.
- Expand or collapse macros.

If multiple results are at the same location in the code, right-click the relevant code to select one of the results to focus on with the **Select Results** option. See “Source Code in Polyspace Access Web Interface”

```

272 void bug_declmismatch() {
273     extern bigstruct_diff
274     S_for_programming; /* Defect: Struct has different definition than in previous definitio
275
276     read_pstruct_diff(&S_
277 }
278
279
280 void corrected_declmismatch
281     extern bigstruct S_fc
282     Using same struct definition */
283     read_pstruct(&S_for_programming);

```

Filter Polyspace Access Results

The toolbar displays several additional options for navigation and filtering.

Custom Filters

Apply and create custom filters. See also “Create Custom Filter Groups in Polyspace Access Web Interface”.

Family Filters

Quickly apply filters by result type. For instance, clicking **Defects** filters to show only defect type results. Clicking the arrow next to the **Defects** filter specifies viewing high, medium, or low defects. Similarly, Run-time Checks, Coding Standards, and Code Metrics enable further narrowing the scope of your review with additional options in the drop down list.

Filters

The Filters section contains quick filters as listed in this table:

Filter	Value
Workflow	<ul style="list-style-type: none"> • Open - Findings with the status 'Unreviewed', 'To fix', 'To investigate', or 'Other'. • To Do - Findings with the status 'Unreviewed'. • In Progress - Findings with the status 'To fix', 'To investigate', 'Other'. • Done - Findings with the status 'Justified', 'Not a defect', or 'No action planned'. • Annotated - Findings with a status, severity, or comment assigned from the source code.
Resolution	<ul style="list-style-type: none"> • New - Findings discovered in current run. • Unresolved - Baseline findings still open in current run. • Resolved - Baseline findings fixed or done in current run. • Fixed - Baseline findings no longer present in current run. <p>See also “Compare Results in Polyspace Access Project to Previous Runs and View Trends”</p>
Assignee	<ul style="list-style-type: none"> • Unassigned • Assigned To Me
Status	<ul style="list-style-type: none"> • Unreviewed • To Investigate • To Fix • Justified • No Action Planned • Not A Defect • Other
Severity	<ul style="list-style-type: none"> • High Severity • Medium Severity • Low Severity
Software Quality Objectives	<p>Individual filters for SQO1 through Exhaustive</p> <p>See also “Evaluate Polyspace Code Prover Results Against Software Quality Objectives”</p>

Use the **Show only** filter to show results associated with the keyword, file name, or comment in the **Show only** field. Use **Filter out** to remove results associated with the keyword, file name, or comment in the **Filter out** field.

You can apply the **Show only** and **Filter out** filters by right-clicking the **Results List** table. Each column allows for the filtering of different parameters. Right-click in the cell containing the phrase that you want to filter and select **Show only** or **Filter out** to apply the filter.



Assign Status and Owner to Results

You can set up email alerts so that component owners get notified when Polyspace results appear in their components. See “Send Email Notifications with Polyspace Code Prover Server Results” on page 3-20

To assign a user to a result, select the result that you want to assign from the **Results List**. In the **Result Details** pane, use the **Assigned to** drop-down list to select the user you want to assign the results to. Alternatively, begin typing the user name and select them from the autocomplete list. To unassign a user, click the x icon to the right of the **Assigned to** drop-down list.

To assign a status, severity, or comment, in the **Result Details** pane, choose a **Status** and **Severity** from the drop-down lists. Comments are entered in the text field to the right of the drop down.

To select multiple results, hold the **Ctrl** key and click each result. If you want to select a group of results, click the first result, then hold the **Shift** key and click the last result to select all results.

If your bug tracking tool is integrated with Polyspace Access, you can use the **Ticket** section to create a ticket based on the result. Click  to create a ticket or click  to link an existing ticket. See “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface”

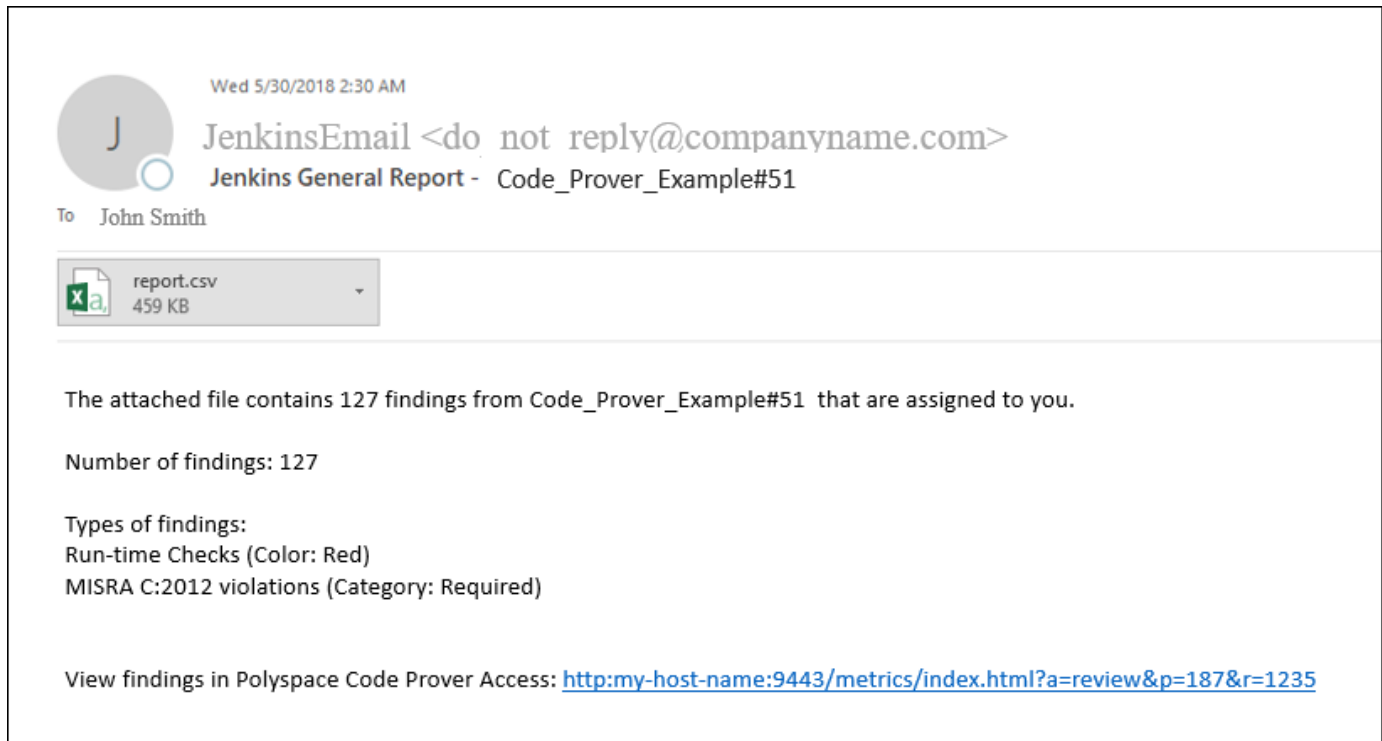
See Also

More About

- “Create Custom Filter Groups in Polyspace Access Web Interface”
- “Review Polyspace Code Prover Results in Web Browser”
- “Create Bug Tracking Tool Tickets from the Polyspace Access Web Interface”

Send Email Notifications with Polyspace Code Prover Server Results

If you run a Polyspace analysis as part of continuous integration, each new code submission produces new results. You not only see new results in components that were modified but also in components that depended on the modified components. You can set up e-mail alerts so that component owners get notified when new Polyspace results appear in their components.



Creating E-mail Notifications

To create e-mail notifications:

- 1 Export new analysis results to a tab-delimited text file (.tsv format).
Apply filters to export specific types of results, for instance, defects with high impact. If required, you can also apply additional filters to the exported files using search and replace utilities. See “Export Results for E-mail Attachments” on page 3-22.
- 2 Send an email with the results file in attachment. For each result, the attachment contains links to open the result in the Polyspace Access web interface.

For instance, if you use an e-mail plugin in Jenkins, you can create a post-build step to send an e-mail after the analysis is complete.

If you use the Polyspace plugin in Jenkins, you can use Polyspace helper utilities for the entire e-mail notification process. See “Sample Scripts for Polyspace Analysis with Jenkins”

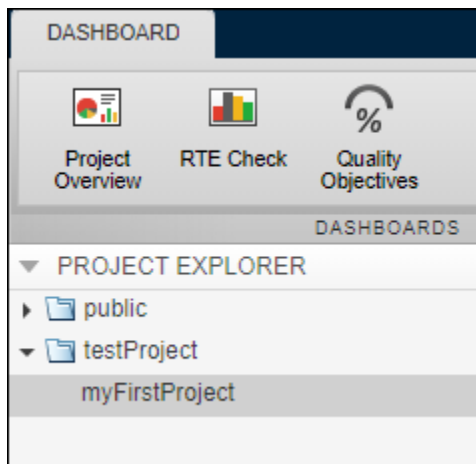
Alternatively, results can be directly assigned to owners based on their file paths. You can set up email notifications that exports a separate results file per owner and sends an email to each owner

with the corresponding results file in attachment. See “Assign Owners and Export Assigned Results” on page 3-22.

Prerequisites

To run this tutorial:

- You must have uploaded some result to the Polyspace Access web server. If you complete the tutorial “Run Polyspace Code Prover on Server and Upload Results to Web Interface” on page 3-6, you should see a folder `testProject` on the **Project Explorer** pane of the Polyspace Access web interface. The folder contains one project `myFirstProject`.



To see the results in the project, with `myFirstProject` selected, click the **Review** button. You see a list of run-time checks. The **Type** column shows the color of the checks. In this tutorial, only red checks will be exported for e-mail attachments.

- You must be able to interact with the Polyspace Access interface from the command line. For instance, navigate to `polyspaceserverroot\polyspace\bin` and enter:

```
polyspace-access login -list-project
```

Here, `polyspaceserverroot` is the Polyspace Code Prover Server installation folder, for instance, `C:\Program Files\Polyspace Server\R2023a`. The variable `login` refers to the following combination of options. You provide these options with every use of the `polyspace-access` command.

```
-host hostName -port portNumber -login username -encrypted-password pwd
```

Here, `hostName` is the name of the Polyspace Access web server. For a locally hosted server, use `localhost`. `portNumber` is the optional port number of the server. If you omit the port number, 9443 is used. `username` and `pwd` refer to the login and an encrypted version of your password. To create an encrypted password, enter:

```
polyspace-access -encrypt-password
```

Copy the encrypted password and provide this password with later uses of the `polyspace-access` command.

Export Results for E-mail Attachments

You can export all results in a project or only certain types of results.

Open a command window. Navigate to the folder where you want to export the results.

- To export all results, enter the following:

```
polyspace-access login -export testProject/myFirstProject -output .\result.txt
```

- To export only red checks, enter the following:

```
polyspace-access login -export testProject/myFirstProject  
-rte Red -output .\result_red_checks.txt
```

Open each text file in a spreadsheet viewing utility such as Microsoft® Excel®. In the first file, you see all results but in the second file, you only see the red run-time checks. Instead of `-rte Red`, you can apply other filters.

- To see only new results compared to the previous analysis of the same project, use the option `-new-findings`.
- To apply a more fine-grained set of filters, you can use software quality objectives (SQOs). The software quality objectives are specified through a progressively stricter set of SQO levels, numbered from 1 to 6. You can customize the requirements of each level in the Polyspace Access web interface, and then use the option `-open-findings-for-sqo` with the level number to export only those results that must be reviewed to meet the requirements. See also “Evaluate Polyspace Code Prover Results Against Software Quality Objectives”.

To see all filtering options, enter:

```
polyspace-access -h -export
```

You can configure your e-mail utility to send these exported files in attachment.

If required, you can also apply additional filters to the exported files using search and replace utilities. For instance, use search and replace utilities on the results file to include results only from specific files and functions. In Linux, you can use `grep` and `sed` to retain only results in specific files.

Instead of exporting to text files, you can also generate reports in PDF or Word using predefined report templates. For more information, see `polyspace-report-generator`.

Assign Owners and Export Assigned Results

You can assign owners to results in specific files or folders. You can then export one result file per owner and send an email to each owner with the corresponding file in attachment.

You can assign owners in the Polyspace Access web interface or at the command line.

In this tutorial, assign all results in the file `example.c` to `jsmith` and all results in the file `single_file_analysis.c` to `jboyd`.

```
polyspace-access login
  -set-unassigned-findings testProject/myFirstProject
  -owner jsmith -source-contains example.c
polyspace-access login
  -set-unassigned-findings testProject/myFirstProject
  -owner jboyd -source-contains single_file_analysis.c
```

After assignment, export one results file per owner.

```
polyspace-access -host login
  -export testProject/myFirstProject -output .\results.txt -output-per-owner
```

These files contain the exported results:

- `results.txt` contains all results.
- `results_jsmith.txt` and `results_jboyd.txt` contains results assigned to `jsmith` and `jboyd` respectively.
- `results.txt.owners.list` contains the list of owners, in this case:

```
jsmith
jboyd
```

Before assigning owners to results, use the option `-dryrun` to perform a dry run of the assignments. Without performing the assignment, the option shows the files with results that are assigned and the owner that the results are assigned to.

See Also

`polyspace-access`

Offloading Code Prover Analysis from Desktop to Server

Send Code Prover Analysis from Desktop to Locally Hosted Server

You can perform a Polyspace analysis locally on your desktop or offload the analysis to one or more dedicated servers. This topic shows a simple server-client configuration for offloading the Polyspace analysis. In this configuration, the same computer acts as a client that submits a Polyspace analysis and a server that runs the analysis.

You can extend this tutorial to more complex configurations. For full setup and workflow instructions, see related links below.

Note The versions of Polyspace on the client and server machines must match.

Client-Server Workflow for Running Bug Finder Analysis

After the initial setup, you can submit a Polyspace analysis from a client desktop to a server. The client-server workflow happens in three steps. All three steps can be performed on the same computer or three different computers. This tutorial uses the same computer for the entire workflow.

- 1 Client node:** You specify Polyspace analysis options and start the analysis on the client desktop. The initial phase of analysis till compilation runs on the desktop. After compilation, the analysis job is submitted to the server.

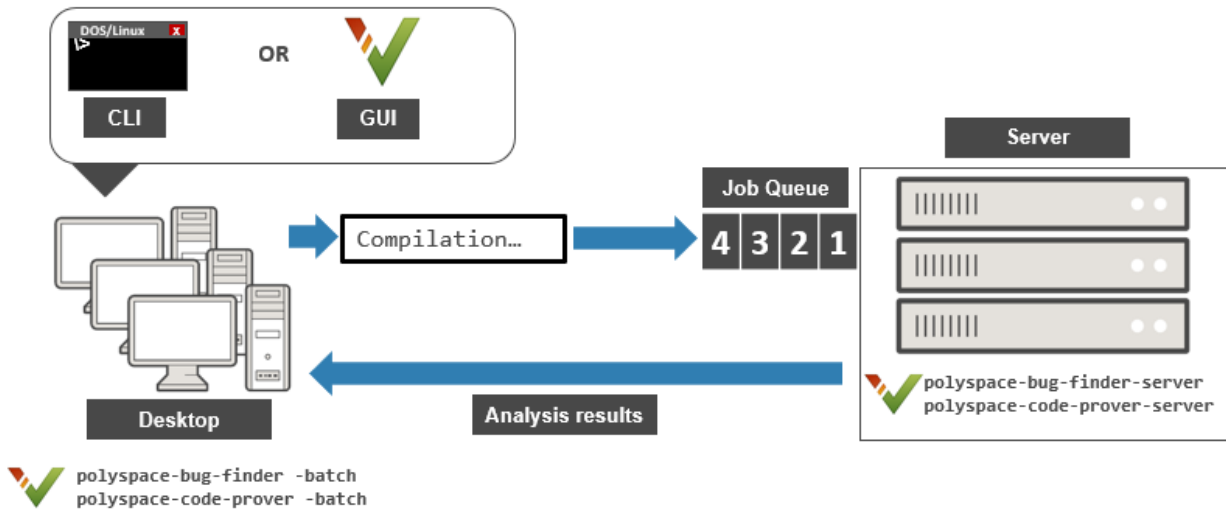
You require the Polyspace desktop product, Polyspace Bug Finder on the computer that acts as the client node.

- 2 Head node:** The server consists of a head node and several worker nodes. The head node uses a job scheduler to manage submissions from multiple client desktops. The jobs are then distributed to the worker nodes as they become available.

You require the product MATLAB Parallel Server™ on the computer that acts as the head node.

- 3 Worker nodes:** When a worker becomes available, the job scheduler assigns the analysis to the worker. The Polyspace analysis runs on the worker and the results are downloaded back to the client desktop for review.

You require the product MATLAB Parallel Server on the computers that act as worker nodes. You also require the Polyspace server products, Polyspace Bug Finder Server and Polyspace Code Prover Server, to run the analysis.



See also “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

Prerequisites

This tutorial uses the same computer as client and server. You must install the following on the computer:

- Client-side product: Polyspace Bug Finder
- Server-side products: MATLAB Parallel Server, Polyspace Bug Finder Server and Polyspace Code Prover Server.

For more information, see “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

You must know the host name of your computer. For instance, in Windows, open a command-line terminal and enter:

```
hostname
```

Configure and Start Server

Stop Previous Services

If you started the services of MATLAB Parallel Server previously, make sure that you have stopped all services. In particular, you might have to:

- Check your temporary folder, for instance, C:\Windows\Temp in Windows, and remove the MDCE folder if it exists.
- Stop all services explicitly. You do not require this step in Linux.

Open a command-line terminal. Navigate to `matlabroot\toolbox\parallel\bin` (using `cd`) and enter the following:

```
mjs uninstall -clean
```

Here, `matlabroot` is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2023a`.

If this is the first time you are starting the services, you do not have to do these steps.

Configure mjs Service Settings

Before starting services, you have to configure the `mjs` service settings. Navigate to `matlabroot\toolbox\parallel\bin`, where `matlabroot` is the MATLAB Parallel Server installation folder, for instance, `C:\Program Files\MATLAB\R2023a`. Modify these two files. To edit and save these files, you have to open your editor in administrator mode.

- `mjs_def.bat` (Windows) or `mjs_def.sh` (Linux)

Read the instructions in the file and uncomment the lines as needed. At a minimum, you might have to uncomment these lines:

- Hostname:

```
REM set HOSTNAME=myHostName
```

in Windows or

```
#HOSTNAME=`hostname -f`
```

in Linux. Remove the `REM` or `#` and explicitly specify your computer host name.

- Security level:

```
REM set SECURITY_LEVEL=
```

in Windows or

```
#SECURITY_LEVEL=""
```

in Linux. Remove the `REM` or `#` and explicitly specify a security level.

Otherwise, you might see an error later when starting the job scheduler.

- `mjs_polyspace.conf`

Modify and uncomment the line that refers to a Polyspace server product root. The line should refer to the release number and root folder of your Polyspace server product installation. For instance, if the R2023a release of Polyspace Code Prover Server is installed in the root folder `C:\Program Files\Polyspace Server\R2023a`, modify the line to:

```
POLYSPACE_SERVER_ROOT=C:\Program Files\Polyspace Server\R2023a
```

Otherwise, the MATLAB Parallel Server installation cannot locate the Polyspace Code Prover Server installation to run the analysis.

Start Services

Start the `mjs` services and assign the current computer as both the head node and a worker node.

Navigate to *matlabroot*\toolbox\parallel\bin, where *matlabroot* is the MATLAB Parallel Server installation folder, for instance, C:\Program Files\MATLAB\R2023a. Run these commands (directly at the command line or using scripts):

```
mjs install
mjs start
startjobmanager -name JobScheduler -remotehost hostname -v
startworker -jobmanagerhost hostname -jobmanager JobScheduler
               -remotehost hostname -v
```

Here, *hostname* is the host name of your computer. This is the host name that you specified in the file *mjs_def.bat* (Windows) or *mjs_def.sh* (Linux). Note that in Linux, you do not require the command `mjs install`.

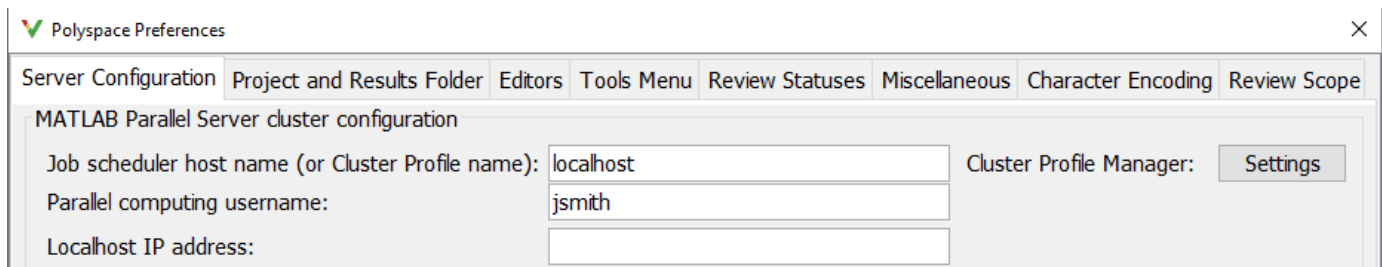
Instead of the command line, you can also start the services from the Admin Center interface. See “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”.

For more information on the commands, see “Configure Advanced Options for MATLAB Job Scheduler Integration” (MATLAB Parallel Server).

Configure Client

Open the user interface of the desktop product, Polyspace Bug Finder (or Polyspace Code Prover). Navigate to *polyspaceroot*\polyspace\bin, where *polyspaceroot* is the Polyspace desktop product installation folder, for instance, C:\Program Files\Polyspace\R2023a and double-click the *polyspace* executable.

Select **Tools > Preferences**. On the **Server configuration** tab, enter the host name of your computer for **Job scheduler host name**.



You are now set up for the server-client workflow.

Send Analysis from Client to Server

Run Code Prover on the file `example.c` provided with your installation.

Before running these steps, to avoid entering full paths to the Polyspace executables, add the path *polyspaceroot*\polyspace\bin to the PATH environment variable on your operating system. Here *polyspaceroot* is the Polyspace desktop product installation folder, for instance, C:\Program Files\Polyspace\R2023a. To check if the path is already added, open a command line terminal and enter:

```
polyspace-code-prover -h
```

If the path to the command is already added, you see the full list of options.

- 1 Copy the file `example.c` and all header files from `polyspaceroot\polyspace\examples\cxx\Code_Prover_Example\sources` to a folder with write permissions.
- 2 Open a command terminal. Navigate to the folder where you saved `example.c` and enter the following:

```
polyspace-code-prover -sources example.c -I . -main-generator  
-results-dir . -batch -scheduler hostname
```

Here, *hostname* is the host name of your computer. To run a Bug Finder analysis, use `polyspace-bug-finder` instead of `polyspace-code-prover`. Note that you can run the `polyspace-code-prover` command with a Polyspace Bug Finder license only, provided you use the `-batch` option.

After compilation, the analysis is submitted to a server and returns a job ID.

- 3 See the status of the current job.

```
polyspace-jobs-manager listjobs -scheduler hostname
```

You can locate the current job using the job ID.

- 4 Once the job is completed, you can explicitly download the results.

```
polyspace-jobs-manager download -job jobID -results-folder .  
-scheduler hostname
```

Here, *jobID* is the job ID from the submission.

The results folder contains the downloaded results file (with extension `.pscp`). Open the results in the user interface of the desktop product, Polyspace Bug Finder.

See Also

More About

- “Install Products for Submitting Polyspace Analysis from Desktops to Remote Server”
- “Send Polyspace Analysis from Desktop to Remote Servers”
- “Send Polyspace Analysis from Desktop to Remote Servers Using Scripts”

Deploy Polyspace Code Prover

Source Code Verification with Polyspace Code Prover

In this section...

“How Polyspace Verification Works” on page 5-2

“Value of Polyspace Code Prover Verification” on page 5-3

How Polyspace Verification Works

Polyspace software uses *static verification* to prove the absence of run-time errors. Static verification derives the dynamic properties of a program without actually executing it. This differs significantly from other techniques, such as run-time debugging, in that the verification it provides is not based on a given test case or set of test cases. The dynamic properties obtained in the Polyspace verification are true for all executions of the software.

What is Static Verification

Static verification is a broad term, and is applicable to any tool that derives dynamic properties of a program without executing the program. However, most static verification tools only verify the complexity of the software, in a search for constructs that may be potentially erroneous. Polyspace verification provides deep-level verification identifying almost all run-time errors and possible access conflicts with global shared data.

Polyspace verification works by approximating the software under verification, using representative approximations of software operations and data.

For example, consider the following code:

```
for (i=0 ; i<1000 ; ++i)
{   tab[i] = foo(i);
}
```

To check that the variable `i` never overflows the range of `tab`, a traditional approach would be to enumerate each possible value of `i`. One thousand checks would be required.

Using the static verification approach, the variable `i` is modelled by its domain variation. For instance, the model of `i` is that it belongs to the static interval `[0..999]`. (Depending on the complexity of the data, convex polyhedrons, integer lattices and more elaborate models are also used for this purpose).

By definition, an approximation leads to information loss. For instance, the information that `i` is incremented by one every cycle in the loop is lost. However, the important fact is that this information is not required to ensure that no range error will occur; it is only necessary to prove that the domain variation of `i` is smaller than the range of `tab`. Only one check is required to establish that — and hence the gain in efficiency compared to traditional approaches.

Static code verification has an exact solution. However, this exact solution is not practical, as it would require the enumeration of all possible test cases. As a result, approximation is required for a usable tool.

Exhaustiveness

Nothing is lost in terms of exhaustiveness. The reason is that Polyspace verification works by performing upper approximations. In other words, the computed variation domain of a program

variable is a superset of its actual variation domain. As a result, Polyspace verifies run-time error items that require checking.

Value of Polyspace Code Prover Verification

Polyspace verification can help you to:

- “Enhance Software Reliability” on page 5-3
- “Decrease Development Time” on page 5-3
- “Improve the Development Process” on page 5-4

Enhance Software Reliability

Polyspace software enhances the reliability of your C/C++ applications by proving code correctness and identifying run-time errors. Using advanced verification techniques, Polyspace software performs an exhaustive verification of your source code.

Because Polyspace software verifies all executions of your code, it can identify code that:

- Never has an error
- Always has an error
- Is unreachable
- Might have an error

With this information, you know how much of your code does not contain run-time errors, and you can improve the reliability of your code by fixing errors.

You can also improve the quality of your code by using Polyspace verification software to check that your code complies with established coding standards, such as the MISRA C™, MISRA™ C++ or JSF® C++ standards.¹

Decrease Development Time

Polyspace software reduces development time by automating the verification process and helping you to efficiently review verification results. You can use it at any point in the development process. However, using it during early coding phases allows you to find errors when it is less costly to fix them.

You use Polyspace software to verify source code before compile time. To verify the source code, you set up verification parameters in a project, run the verification, and review the results. This process takes significantly less time than using manual methods or using tools that require you to modify code or run test cases.

Color-coding of results helps you to quickly identify errors. You will spend less time debugging because you can see the exact location of an error in the source code. After you fix errors, you can easily run the verification again.

Polyspace verification software helps you to use your time effectively. Because you know the parts of your code that do not have errors, you can focus on the code with proven (red code) or potential errors (orange code).

¹ MISRA and MISRA C are registered trademarks of MISRA Ltd., held on behalf of the MISRA Consortium.

Reviewing code that might have errors (orange code) can be time-consuming, but Polyspace software helps you with the review process. You can use filters to focus on certain types of errors or you can allow the software to identify the code that you should review.

Improve the Development Process

Polyspace software makes it easy to share verification parameters and results, allowing the development team to work together to improve product reliability. Once verification parameters have been set up, developers can reuse them for other files in the same application.

Polyspace verification software supports code verification throughout the development process:

- An individual developer can find and fix run-time errors during the initial coding phase.
- Quality assurance engineers can check overall reliability of an application.
- Managers can monitor application reliability by generating reports from the verification results.

See Also

More About

- “Quick Start Guide for Polyspace Server and Access Products” on page 3-2
- “Run Polyspace Code Prover on Desktop” on page 2-2
- “Polyspace Products for Desktops and Servers”
- “Differences Between Polyspace Bug Finder and Polyspace Code Prover”

Polyspace Products and Software Development Workflows

In this section...

“Using Polyspace Products in Software Development” on page 5-5

“Coordinating Pre-Submit and Post-Submit Usage of Polyspace” on page 5-6

“Polyspace Products for Ada Code” on page 5-7

Polyspace products use static analysis to check code for run-time errors, coding standard violations, security vulnerabilities, and other issues:

- Polyspace Code Prover can cover all possible execution paths through a program and track data flow along these paths following certain mathematical rules. The exhaustive control and data flow analysis can complement dynamic testing and expose potential run-time errors that might not be otherwise found in regular robustness testing.
- Polyspace Bug Finder can scan a program for more obvious defects, security vulnerabilities, coding standard violations and other issues that potentially lead to run-time errors or unexpected results.

Using Polyspace Products in Software Development

The Polyspace suite of products supports all phases of a software development process:

- *Prior to code submission:*

Developers can run the Polyspace desktop or IDE-focused products to check their code during development or right before submission to meet predefined quality goals.

The products can be integrated into IDEs such as Visual Studio Code, Visual Studio, or Eclipse, or run with scripts during compilation. The analysis results can be reviewed in the IDEs or in the graphical user interface of the desktop products.

Polyspace provides the following products for desktop usage. These products are meant to run on complete projects or smaller code modules (up to a single source file).

- **Polyspace Bug Finder** to check code for semantic errors that a compiler cannot detect (such as use of = instead of ==), concurrency issues, security vulnerabilities and other defects in C and C++ source code.
- **Polyspace Code Prover** to perform a much deeper check and prove absence of overflow, divide-by-zero, out-of-bounds array access and other run-time errors in C and C++ source code.
- *After code submission:*

The Polyspace server products can run automatically on newly committed code as a build step in a continuous integration process (using tools such as Jenkins). The analysis runs on a server and the results are uploaded to a web interface for collaborative review.

Polyspace provides these products for server usage:

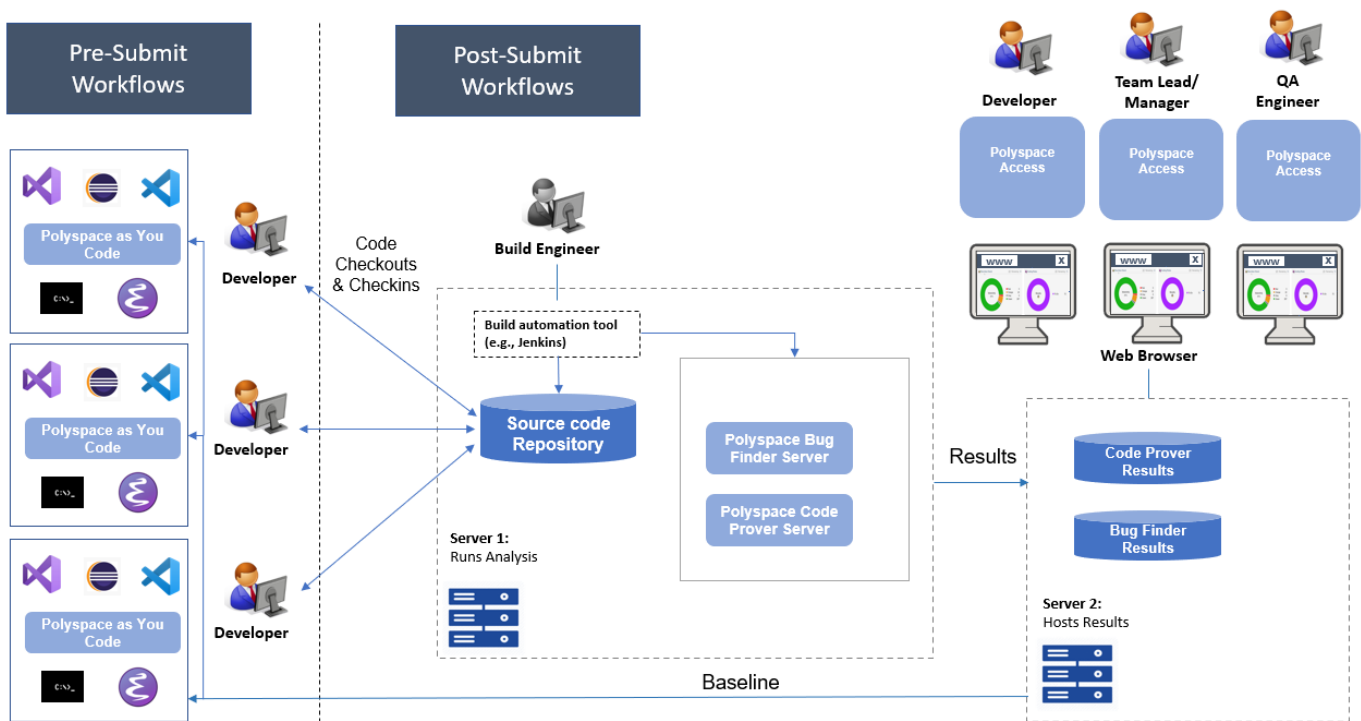
- **Polyspace Bug Finder Server** to run Bug Finder automatically on a server and upload the results to a web interface for review, and **Polyspace Access** to review the uploaded results.

- **Polyspace Code Prover Server** to run Code Prover automatically on a server and upload the results to a web interface for review, and **Polyspace Access** to review the uploaded results.

Typically, Polyspace Bug Finder Server (or Polyspace Code Prover Server) runs on a few build servers and checks newly committed code as part of software build and testing. Each reviewer (developer, quality assurance engineer or development manager) has a Polyspace Access license to review the uploaded analysis results.

In addition, if developers have access to **Polyspace Access** for web review of post-submission results, they can also install **Polyspace as You Code** in their IDEs for pre-submission analysis. When installed as an IDE extension, Polyspace as You Code performs a file-scope Bug Finder-like analysis and provides near-instant feedback to developers while coding.

This diagram shows *one possible deployment* of Polyspace products:



When you use both the desktop and server products, your pre-submission workflow can transition smoothly to the post-submission workflow.

Coordinating Pre-Submit and Post-Submit Usage of Polyspace

When you run more than one Polyspace products at separate stages in your software development workflow, the later runs can benefit from the earlier usage, and vice versa. In particular:

- Developers using Polyspace as You Code in their IDEs can easily fix defects and coding standard violations that can be found and resolved within a single file. A later Polyspace Bug Finder Server analysis after code submission no longer shows these issues.

- The results of a Polyspace Bug Finder Server analysis can act as a baseline for Polyspace as You Code runs. Developers using the latest Polyspace Bug Finder Server result as baseline for their IDE runs can focus only on issues that result from their code changes.

Polyspace Products for Ada Code

Polyspace provides these products for verifying Ada code:

- **Polyspace Client™ for Ada** to check Ada code for run-time errors on a desktop.
- **Polyspace Server for Ada** to check Ada code for run-time errors on a server.

You can either use the desktop product to run the analysis on your desktop, or a combination of the desktop and server products to run the analysis on a server. The analysis results are downloaded to your desktop for review.

If you have a Polyspace Code Prover Access license and have set up the web interface of Polyspace Code Prover Access, you can upload each individual Ada result from the Ada desktop products to the web interface for collaborative review.

See also <https://www.mathworks.com/products/polyspace-ada.html>.

See Also

Related Examples

- “Install Polyspace Desktop Products”
- “Install Polyspace Server and Access Products”
- “Differences Between Polyspace Bug Finder and Polyspace Code Prover” on page 5-8

Differences Between Polyspace Bug Finder and Polyspace Code Prover

Polyspace Bug Finder and Polyspace Code Prover detect run-time errors through static analysis. Though the products have a similar user interface and the mathematics underlying the analysis can sometimes be the same, the goals of the two products are different.

Bug Finder (or Polyspace as You Code, which performs a single-file analysis similar to Bug Finder) quickly analyzes your code and detects many types of defects. Code Prover checks *every* operation in your code for a set of possible run-time errors and tries to prove the absence of the error for all execution paths². For instance, for *every* division in your code, a Code Prover analysis tries to prove that the denominator cannot be zero. Bug Finder does not perform such exhaustive verification. For instance, Bug Finder also checks for a division by zero error, but it might not find all operations that can cause the error.

The two products involve differences in setup, analysis and results review, because of this difference in objectives. In the following sections, we highlight the primary differences between a Bug Finder and a Code Prover analysis (also known as verification). Depending on your requirements, you can incorporate one or both kinds of analyses at appropriate points in your software development life cycle.

How Bug Finder and Code Prover Complement Each Other

- “Overview” on page 5-8
- “Faster Analysis with Bug Finder” on page 5-9
- “More Exhaustive Verification with Code Prover” on page 5-9
- “More Specific Defect Types with Bug Finder” on page 5-10
- “Easier Setup Process with Bug Finder” on page 5-10
- “Fewer Runs for Clean Code with Bug Finder” on page 5-11
- “Results in Real Time with Bug Finder” on page 5-11
- “More Rigorous Data and Control Flow Analysis with Code Prover” on page 5-11
- “Few False Positives with Bug Finder” on page 5-12
- “Zero False Negatives with Code Prover” on page 5-12
- “Coding Rule Support in Bug Finder” on page 5-13

Overview

Use both Bug Finder and Code Prover regularly in your development process. The products provide a unique set of capabilities and complement each other. For possible ways to use the products together, see “Workflow Using Both Polyspace Bug Finder and Polyspace Code Prover”.

This table provides an overview of how the products complement each other. For details, see the sections below.

² For each operation in your code, Code Prover considers all execution paths leading to the operation that do not have a previous error. If an execution path contains an error prior to the operation, Code Prover does not consider it. See “Code Prover Analysis Following Red and Orange Checks”.

Feature	Bug Finder	Code Prover
Number of checkers	300+ checkers for defects	30 checks for critical run-time errors and 4 checks on global variable usage
Depth of analysis	Fast. For instance: <ul style="list-style-type: none"> • Faster analysis. • Easier set up and review. • Fewer runs for clean code. • Results in real time. 	Exhaustive. For instance: <ul style="list-style-type: none"> • All operations of a type checked for certain critical errors. • More rigorous data and control flow analysis.
Reporting criteria	Probable defects	Proven findings
Bug finding criteria	Few false positives	Zero false negatives

Faster Analysis with Bug Finder

How much faster the Bug Finder analysis is depends on the size of the application. The Bug Finder analysis time increases linearly with the size of the application. The Code Prover verification time increases at a rate faster than linear.

One possible workflow is to run Code Prover to analyze modules or libraries for robustness against certain errors and run Bug Finder at integration stage. Bug Finder analysis on large code bases can be completed in a much shorter time, and also find integration defects such as **Declaration mismatch** and **Data race**.

More Exhaustive Verification with Code Prover

Code Prover tries to prove the absence of:

- **Division by Zero** error on *every* division or modulus operation
- **Out of Bounds Array Index** error on *every* array access
- **Non-initialized Variable** error on *every* variable read
- **Overflow** error on *every* operation that can overflow

and so on.

For each operation:

- If Code Prover can prove the absence of the error for all execution paths, it highlights the operation in green.
- If Code Prover can prove the presence of a definite error for all execution paths, it highlights the operation in red.
- If Code Prover cannot prove the absence of an error or presence of a definite error, it highlights the operation in orange. This small percentage of orange checks indicate operations that you must review carefully, through visual inspection or testing. The orange checks often indicate hidden vulnerabilities. For instance, the operation might be safe in the current context but fail when reused in another context.

You can use information provided in the Polyspace user interface to diagnose the checks. See “More Rigorous Data and Control Flow Analysis with Code Prover”. You can also provide

contextual information to reduce unproven code even further, for instance, constrain input ranges externally.

Bug Finder does not aim for exhaustive analysis. It tries to detect as many bugs as possible and reduce false positives. For critical software components, running a bug finding tool is not sufficient because despite fixing all defects found in the analysis, you can still have errors during code execution (false negatives). After running Code Prover on your code and addressing the issues found, you can expect the quality of your code to be much higher. See “Zero False Negatives with Code Prover”.

More Specific Defect Types with Bug Finder

Code Prover checks for types of run-time errors where it is possible to mathematically prove the absence of the error. In addition to detecting errors whose absence can be mathematically proven, Bug Finder also detects other defects.

For instance, the statement `if (a=b)` is semantically correct according to the C language standard, but often indicates an unintended assignment. Bug Finder detects such unintended operations. Although Code Prover does not detect such unintended operations, it can detect if an unintended operation causes other run-time errors.

Examples of defects detected by Bug Finder but not by Code Prover include good practice defects, resource management defects, some programming defects, security defects, and defects in C++ object oriented design.

For more information, see:

- “Defects”: List of defects that Bug Finder can detect.
- “Run-Time Checks”: List of run-time errors that Code Prover can detect.

Easier Setup Process with Bug Finder

Even if your code builds successfully in your compilation toolchain, it can fail in the compilation phase of a Code Prover verification. The strict compilation in Code Prover is related to its ability to prove the absence of certain run-time errors.

- Code Prover strictly follows the ANSI® C99 Standard, unless you explicitly use analysis options that emulate your compiler.

To allow deviations from the ANSI C99 Standard, you must use the “Target and Compiler” options. If you create a Polyspace project from your build system, the options are automatically set.

- Code Prover does not allow linking errors that common compilers can permit.

Though your compiler permits linking errors such as mismatch in function signature between compilation units, to avoid unexpected behavior at run time, you must fix the errors.

For more information, see “Troubleshoot Compilation and Linking Errors”.

Bug Finder is less strict about certain compilation errors. Linking errors, such as mismatch in function signature between different compilation units, can stop a Code Prover verification but not a Bug Finder analysis. Therefore, you can run a Bug Finder analysis with less setup effort. In Bug Finder, linking errors are often reported as a defect after the analysis is complete.

Fewer Runs for Clean Code with Bug Finder

To guarantee absence of certain run-time errors, Code Prover follows strict rules once it detects a run-time error in an operation. Once a run-time error occurs, the state of your program is ill-defined and Code Prover cannot prove the absence of errors in subsequent code. Therefore:

- If Code Prover proves a definite error and displays a red check, it does not verify the remaining code in the same block.

Exceptions include checks such as **Overflow**, where the analysis continues with the result of overflow either truncated or wrapped around.

- If Code Prover suspects the presence of an error and displays an orange check, it eliminates the path containing the error from consideration. For instance, if Code Prover detects a **Division by Zero** error in the operation $1/x$, in the subsequent operation on x in that block, x cannot be zero.
- If Code Prover detects that a code block is unreachable and displays a gray check, it does not detect errors in that block.

For more information, see “Code Prover Analysis Following Red and Orange Checks”.

Therefore, once you fix red and gray checks and rerun verification, you can find more issues. You need to run verification several times and fix issues each time for completely clean code. The situation is similar to dynamic testing. In dynamic testing, once you fix a failure at a certain point in the code, you can uncover a new failure in subsequent code.

Bug Finder does not stop the entire analysis in a block after it finds a defect in that block. Even with Bug Finder, you might have to run analysis several times to obtain completely clean code. However, the number of runs required is fewer than Code Prover.

Results in Real Time with Bug Finder

Bug Finder shows some analysis results while the analysis is still running. You do not have to wait until the end of the analysis to review the results.

Code Prover shows results only after the end of the verification. Once Bug Finder finds a defect, it can display the defect. Code Prover has to prove the absence of errors on all execution paths. Therefore, it cannot display results during analysis.

More Rigorous Data and Control Flow Analysis with Code Prover

For each operation in your code, Code Prover provides:

- Tooltips showing the range of values of each variable in the operation.

For a pointer, the tooltips show the variable that the pointer points to, along with the variable values.

- Graphical representation of the function call sequence that leads to the operation.

By using this range information and call graph, you can easily navigate the function call hierarchy and understand how a variable acquires values that lead to an error. For instance, for an **Out of Bounds Array Index** error, you can find where the index variable is first assigned values that lead to the error.

When reviewing a result in Bug Finder, you also have supporting information to understand the root cause of a defect. For instance, you have a traceback from where Bug Finder found a defect to its

root cause. However, in Code Prover, you have more complete information, because the information helps you understand all execution paths in your code.

```

167 static void Square_Root_conv(double alpha, float* beta_pt)
168 /* Perform arithmetic conversion of alpha to beta */
169 {
170     *beta_pt = (float)((1.5 + cos(alpha)) / 5.0);
171 }
172
173
174 stati
175 {
176     d
177     f
178     f
179
180     Square_Root_conv(alpha, &beta);
181
182     gamma = (float)sqrt(beta - 0.75); /* always sqrt(negative number) */
183 }

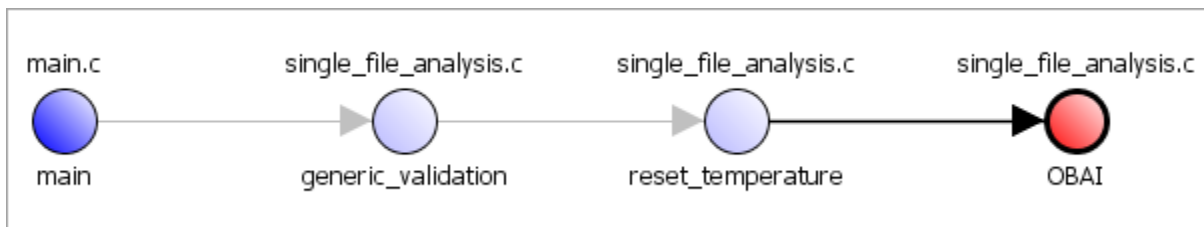
```

Dereference of parameter 'beta_pt' (pointer to float 32, size: 32 bits):
 Pointer is not null.
 Points to 4 bytes at offset 0 in buffer of 4 bytes, so is within bounds (if memory is allocated).
 Pointer may point to variable or field of variable:
 'beta', local to function 'Square_Root'.

Assignment to dereference of parameter 'beta_pt' (float 32): [0.1 .. 0.5]

Press 'F2' for focus

Data Flow Analysis in Code Prover



Control Flow Analysis in Code Prover

Few False Positives with Bug Finder

Bug Finder aims for few false positives, that is, results that you are not likely to fix. By default, you are shown only the defects that are likely to be most meaningful for you.

Bug Finder also assigns an attribute called impact to the defect types based on the criticality of the defect and the rate of false positives. You can choose to analyze your code only for high-impact defects. You can also enable or disable a defect that you do not want to review³.

Zero False Negatives with Code Prover

Code Prover aims for an exhaustive analysis. The software checks every operation that can trigger specific types of error. If a code operation is green, it means that the operation cannot cause those run-time errors that the software checked for⁴. In this way, the software aims for zero false negatives.

³ You can also disable certain Code Prover defects related to non-initialization.

If the software cannot prove the absence of an error, it highlights the suspect operation in red or orange and requires you to review the operation.

Coding Rule Support in Bug Finder

Bug Finder supports checking for compliance with external coding standards, such as:

- AUTOSAR C++14. See “AUTOSAR C++14 Rules”.
- MISRA C:2012. See “MISRA C:2012 Directives and Rules”.
- MISRA C++:2008. See “MISRA C++:2008 Rules”.
- CERT C. See “CERT C Rules and Recommendations”.
- CERT C++. See “CERT C++ Rules”.

For the complete list of coding standards support, see “Polyspace Support for Coding Standards”.

See Also

More About

- “Workflow Using Both Polyspace Bug Finder and Polyspace Code Prover” on page 5-14

4 The Code Prover result holds only if you execute your code under the same conditions that you supplied to Code Prover through the analysis options.

Workflow Using Both Polyspace Bug Finder and Polyspace Code Prover

If you have both Bug Finder and Code Prover, based on the above differences, you can deploy the two products appropriately in your software development workflow. For instance:

- All developers in your organization can run Bug Finder on newly developed code. For maintaining standards across your organization, you can deploy a common configuration that looks only for specific defect types.

Code Prover can be deployed as part of your unit testing suite.

- You can run Code Prover only on critical components of your project, while running Bug Finder on the entire project.
- You can run Code Prover on modules of code at the unit testing level, and run Bug Finder when integrating the modules.

You can run Code Prover before unit testing. Code Prover exhaustively checks your code and tries to prove the presence or absence of errors. Instead of writing unit tests for your entire code, you can then write tests only for unproven code. Using Code Prover before unit testing reduces your testing efforts drastically.

Depending on the nature of your software development workflow and available resources, there are many other ways you can incorporate the two kinds of analysis. You can run both products on your desktop during development or as part of automated testing on a remote server. Note that it is easier to interpret and fix bugs closer to development. You benefit from using both products if you deploy them early and often in your development process.

There are two important considerations if you are running both Bug Finder and Code Prover on the same code.

- Starting in R2022a, Polyspace Bug Finder is the recommended tool for checking compliance to external coding standards such as AUTOSAR C++14 or MISRA C++:2008. Check for violation of these coding standards when you use Bug Finder on your code. You might have used Polyspace Code Prover to check for this purpose. Migrate your workflow to use Bug Finder. See “Migrate Code Prover Workflows for Checking Coding Standards and Code Metrics to Bug Finder”.
- You can use the same project for both Bug Finder and Code Prover analysis. The following set of options are common between Bug Finder and Code Prover:
 - “Target and Compiler”
 - “Macros”
 - “Environment Settings”
 - “Inputs and Stubbing”
 - “Multitasking”
 - “Coding Standards & Code Metrics”
 - “Reporting”, except Bug Finder and Code Prover report (-report-template)

You might have to change more of the default options when you run the Code Prover verification because Code Prover is stricter about compilation and linking errors.