

Oracle Solaris Studio 12.2 Discover and Uncover User's Guide

Copyright © 2010, 2011, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related software documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. UNIX is a registered trademark licensed through X/Open Company, Ltd.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Copyright © 2010, 2011, Oracle et/ou ses affiliés. Tous droits réservés.

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf disposition de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, breveter, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est concédé sous licence au Gouvernement des Etats-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l'utilise pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique :

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour ce type d'applications.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d'Advanced Micro Devices. Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. UNIX est une marque déposée concédée sous licence par X/Open Company, Ltd.

Contents

Preface	5
1 Introduction	9
Sun Memory Error Discovery Tool (Discover)	9
Code Coverage Tool (Uncover)	10
2 Sun Memory Error Discovery Tool (Discover)	11
Requirements for Using Discover	11
Binaries Must Be Prepared Correctly	11
Binaries That Use Preloading or Auditing Cannot Be Used	12
Binaries That Redefine Standard Memory Allocation Functions Can Be Used	12
Quick Start	13
Instrumenting a Prepared Binary	14
Caching Shared Libraries	14
Instrumenting Shared Libraries	14
Ignoring Libraries	15
Command Line Options	15
bit.rc Initialization Files	17
SUNW_DISCOVER_OPTIONS Environment Variable	17
SUNW_DISCOVER_FOLLOW_FORK_MODE Environment Variable	18
Running an Instrumented Binary	18
Analyzing Discover Reports	18
Analyzing the HTML Report	19
Analyzing the ASCII Report	26
Memory Access Errors and Warnings	29
Memory Access Errors	29
Memory Access Warnings	32

Interpreting Discover Error Messages	33
Partially Initialized Memory	33
Speculative Loads	34
Uninstrumented Code	34
Limitations When Using Discover	36
Only Annotated Code is Instrumented	36
Machine Instruction Might Differ From Source Code	36
Compiler Options Affect the Generated Code	36
System Libraries Can Affect the Errors Reported	37
Custom Memory Management Can Affect the Accuracy of the Data	37
Out of Bounds Errors for Static and Automatic Arrays Cannot Be Detected	37
3 Code Coverage Tool (Uncover)	39
Requirements for Using Uncover	39
Using Uncover	39
Instrumenting the Binary	40
Running the Instrumented Binary	40
Generating and Viewing the Coverage Report	40
Examples	41
Understanding the Coverage Report in the Performance Analyzer	42
The Functions Tab	42
The Source Tab	45
The Disassembly Tab	46
The Inst-Freq Tab	46
Understanding the ASCII Coverage Report	47
Understanding the HTML Coverage Report	51
Limitations When Using Uncover	53
Only Annotated Code Can Be Instrumented	53
Machine Instructions Might Differ From Source Code	54
Index	55

Preface

The *Oracle Solaris Studio 12.2 Discover and Uncover User's Guide* gives instructions on how to use the Sun Memory Error Discovery Tool (Discover) to find memory-related errors in binaries compiled on SPARC®-based platforms, and the Code Coverage Tool (Uncover) to measure code coverage of applications.

Note – This Oracle Solaris Studio release supports systems that use the SPARC and x86 families of processor architectures: UltraSPARC, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems for the version of the Solaris operating system you are running are available in the hardware compatibility lists at <http://www.sun.com/bigadmin/hcl>. These documents cite any implementation differences between the platform types.

In this document, these x86 related terms mean the following:

- “x86” refers to the larger family of 64-bit and 32-bit x86 compatible products.
- “x64” points out specific 64-bit information about AMD64 or EM64T systems.
- “32-bit x86” points out specific 32-bit information about x86 based systems.

For supported systems, see the hardware compatibility lists.

Who Should Use This Book

This book is designed for programmers who want to use Discover to find memory access errors in their programs, and Uncover to measure code coverage. Users of Discover and Uncover need a working knowledge of Fortran, C, or C++, and some understanding of the Oracle Solaris operating system (Solaris OS) or the Linux operating system, and UNIX commands.

Accessing Oracle Solaris Studio Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index page at <http://www.oracle.com/technetwork/server-storage/solarisstudio/documentation/index.html>.
- Online help for all components of the IDE is available through the Help menu, as well as through the F1 key, and through Help buttons on many windows and dialog boxes, in the IDE.
- Online help for the Performance Analyzer is available through the Help menu, as well as through the F1 key, and through Help buttons on many windows and dialog boxes, in the Performance Analyzer.
- Online help for dbxtool and DLight is available through the Help menus, as well as through the F1 key, and through Help buttons on many windows and dialog boxes in the tools.
- Online help for dbxtool is available through the Help menu in dbxtool.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table.

Type of Documentation	Format and Location of Accessible Version
Manuals and tutorials	HTML from the Oracle Solaris Studio 12.1 collection on http://docs.sun.com
<i>What's New in the Oracle Solaris Studio 12.2 Release</i> (Information that was included in the component Readme files in previous releases)	HTML from the Oracle Solaris Studio 12.2 collection on http://docs.sun.com
Man pages	In the installed product through the <code>man</code> command
Online help	HTML available through the Help menu, Help buttons, and F1 key in the IDE, Performance Analyzer, DLight, and dbxtool.
Release notes	HTML from the Oracle Solaris Studio 12.2 collection on http://docs.sun.com

Related Third-Party Web Site References

Third-party URLs are referenced in this document and provide additional, related information.

Note – Oracle is not responsible for the availability of third-party web sites mentioned in this document. Oracle does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Oracle will not be responsible or liable for any actual or alleged damage or loss caused or alleged to be caused by or in connection with use of or reliance on any such content, goods, or services that are available on or through such sites or resources.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P-1 Typographic Conventions

Typeface	Meaning	Example
AaBbCc123	The names of commands, files, and directories, and onscreen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% you have mail.</code>
AaBbCc123	What you type, contrasted with onscreen computer output	<code>machine_name% su</code> Password:
<i>aabbcc123</i>	Placeholder: replace with a real name or value	The command to remove a file is <i>rm filename</i> .
<i>AaBbCc123</i>	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the <i>User's Guide</i> . <i>A cache</i> is a copy that is stored locally. Do <i>not</i> save the file. Note: Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P-2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	\$
Bash shell, Korn shell, and Bourne shell for superuser	#
C shell	machine_name%
C shell for superuser	machine_name#

Documentation, Support, and Training

See the following web sites for additional resources:

- [Documentation \(http://docs.sun.com\)](http://docs.sun.com)
- [Support \(http://www.oracle.com/us/support/systems/index.html\)](http://www.oracle.com/us/support/systems/index.html)
- [Training \(http://education.oracle.com\)](http://education.oracle.com) – Click the Sun link in the left navigation bar.

Oracle Welcomes Your Comments

Oracle welcomes your comments and suggestions on the quality and usefulness of its documentation. If you find any errors or have any other suggestions for improvement, go to <http://docs.sun.com> and click Feedback. Indicate the title and part number of the documentation along with the chapter, section, and page number, if available. Please let us know if you want a reply.

[Oracle Technology Network \(http://www.oracle.com/technetwork/index.html\)](http://www.oracle.com/technetwork/index.html) offers a range of resources related to Oracle software:

- Discuss technical problems and solutions on the [Discussion Forums \(http://forums.oracle.com\)](http://forums.oracle.com).
- Get hands-on step-by-step tutorials with [Oracle By Example \(http://www.oracle.com/technology/obe/start/index.html\)](http://www.oracle.com/technology/obe/start/index.html).
- Download [Sample Code \(http://www.oracle.com/technology/sample_code/index.html\)](http://www.oracle.com/technology/sample_code/index.html).

Introduction

The *Oracle Solaris Studio 12.2 Discover and Uncover User's Guide* gives instructions on how to use the following tools:

- “Sun Memory Error Discovery Tool (Discover)” on page 9
- “Code Coverage Tool (Uncover)” on page 10

Sun Memory Error Discovery Tool (Discover)

The Sun Memory Error Discovery Tool (Discover) software is an advanced development tool for detecting memory access errors. Discover works on binaries compiled with the Sun Studio 12, Sun Studio 12 Update 1, Oracle Solaris Express 6/10, or Oracle Solaris Studio 12.2 compilers, or the GCC for Sun Systems compilers starting with version 4.2.0, on systems running the Solaris 10 5/08 operating system or a later Solaris 10 update.

Memory-related errors in programs are notoriously difficult to find. Discover allows you to find such errors easily by pointing out the exact place where the problem exists in the source code. For example, if your program allocates an array and does not initialize it, then tries to read from one of the array locations, the program will probably behave erratically. Discover can catch this problem when you run the program in the normal way.

Other errors detected by Discover include:

- Reading from and writing to unallocated memory
- Accessing memory beyond allocated array bounds
- Incorrect use of freed memory
- Freeing the wrong memory blocks
- Memory leaks

Since Discover catches and reports memory access errors dynamically during program execution, if a portion of user code is not executed at run time, errors in that portion are not reported.

Discover is simple to use. Any binary (even a fully optimized binary) that has been prepared by the compiler can be instrumented with a single command, then run in the normal way. During the run, Discover produces a report of the memory anomalies, which you can view as a text file, or as HTML in a web browser.

Code Coverage Tool (Uncover)

Uncover is a simple and easy to use command-line tool for measuring code coverage of applications. Code coverage is an important part of software testing. It gives you information on which areas of your code are exercised in testing and which are not, enabling you to improve your test suites to test more of your code. The coverage information reported by Uncover can be at a function, statement, basic block, or instruction level.

Uncover provides a unique feature called `uncoverage`, which allows you to quickly find major functional areas that are not being tested. Other advantages of Uncover code coverage over other types of instrumentation are:

- The slowdown relative to uninstrumented code is fairly small.
- Since Uncover operates on binaries, it can work with any optimized binary.
- Measurements can be done by instrumenting the shipping binary. The application does not have to be built differently for coverage testing.
- Uncover provides a simple procedure for instrumenting the binary, running tests, and displaying the results.
- Uncover is multithread safe and multiprocess safe.

Sun Memory Error Discovery Tool (Discover)

The Sun Memory Error Discovery Tool (Discover) software is an advanced development tool for detecting memory access errors.

This chapter includes information about the following:

- “Requirements for Using Discover” on page 11
- “Quick Start” on page 13
- “Instrumenting a Prepared Binary” on page 14
- “Running an Instrumented Binary” on page 18
- “Analyzing Discover Reports” on page 18
- “Memory Access Errors and Warnings” on page 29
- “Interpreting Discover Error Messages” on page 33
- “Limitations When Using Discover” on page 36

Requirements for Using Discover

Binaries Must Be Prepared Correctly

Discover works on binaries compiled with the Sun Studio 12, Sun Studio 12 Update 1, or Oracle Solaris Studio 12.2 compilers, or the GCC for Sun Systems compilers starting with version 4.2.0, on a SPARC-based or x86-based system running the Solaris 10 5/08 operating system or a later Solaris 10 update.

When using a Sun Studio or Oracle Solaris Studio compiler, you must compile with optimizations by using the `-O` option or the `-xO[n]` option. When using a GCC compiler, no particular optimization level is required.

Discover issues an error and does not instrument a binary if it does not meet these requirements. However, you can instrument a binary that does not meet these requirements and run it to detect a limited number of errors by using the `-l` option (see [“Instrumentation Options” on page 16](#)).

A binary compiled as described includes information called annotations to help Discover instrument it correctly. The addition of this small amount of information does not affect the performance of the binary or its runtime memory usage.

Using the `-g` option to generate debug information when compiling the binary allows Discover to display source code and line number information while reporting errors and warnings, and to produce more accurate results. If your binary is not compiled with the `-g` option, Discover displays only the program counters of the corresponding machine level instructions. Also, compiling with the `-g` option helps Discover produce more accurate reports (see [“Interpreting Discover Error Messages” on page 33](#)).

Binaries That Use Preloading or Auditing Cannot Be Used

Because Discover uses some special features of the runtime linker, you cannot use it with binaries that use preloading or auditing.

If a program requires the setting of the `LD_PRELOAD` environment variable, it probably won't work correctly with Discover, because Discover needs to interpose on certain system functions, and it cannot do so if the function has been preloaded.

Similarly, if a program uses runtime auditing (either the binary was linked with the `-p` option or the `-P` option, or it requires the `LD_AUDIT` environment variable to be set), this auditing will conflict with Discover's use of auditing. If the binary was linked with auditing, Discover fails at instrumentation time. If you set the `LD_AUDIT` environment variable at runtime, the results are undefined.

Binaries That Redefine Standard Memory Allocation Functions Can Be Used

Discover supports binaries that redefine the standard memory allocation functions: `malloc()`, `calloc()`, `memalign()`, `valloc()`, and `free()`.

Quick Start

The following is an example of preparing a program, instrumenting it with Discover, and then running it and producing a report on the detected memory access errors. This example uses a simple program that accesses uninitialized data.

```
% cat test_UMR.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // UMR: accessing uninitialized data
    int *p = (int*) malloc(sizeof(int));
    printf("**p = %d\n", *p);
    free(p);
}

% cc -g -O2 test_UMR.c
% a.out
*p = 131464
% discover -w - a.out
% a.out
ERROR (UMR): accessing uninitialized data from address 0x50010 (4 bytes) at:
    main() + 0x54 <test_UMR.c:7>
        4:  {
        5:      // UMR: accessing uninitialized data
        6:      int *p = (int*) malloc(sizeof(int));
        7:=>   printf(**p = %d\n", *p);
        8:      free(p);
        9:  }
    _start() + 0x108
block at 0x50010 (4 bytes long) was allocated at:
    malloc() + 0x220
    main() + 0x1c <test_UMR.c:6>
        3:  int main()
        4:  {
        5:      // UMR: accessing uninitialized data
        6:=>   int *p = (int*) malloc(sizeof(int));
        7:      printf(**p = %d\n", *p);
        8:      free(p);
        9:  }
    _start() + 0x108
*p = 327704
***** Discover Memory Report *****
No allocated memory left on program exit.
DISCOVER SUMMARY:
    unique errors   : 1 (1 total, 0 filtered)
    unique warnings : 0 (0 total, 0 filtered)
FILTERS: type=ML:func=_rt_boot:depth=*; type=*:func__f90_esfw:depth=*;
type=ML:func=__f90_ssfw:depth=*
```

The Discover output shows where the uninitialized memory was used and where it was allocated, along with summary of results.

Instrumenting a Prepared Binary

Once you have prepared the target binary, the next step is to instrument it. Instrumentation adds code in strategic places so that Discover can keep track of memory operations while the binary is running.

You instrument a binary using the `discover` command. For example, the following command instruments the binary `a.out` and overwrites the input `a.out` with the instrumented `a.out`:

```
discover a.out
```

When you run the instrumented binary, Discover monitors the program's use of memory. During the run, Discover writes a report detailing any memory access errors to an HTML file (in this case, by default, `a.out.html`) that you can view in your web browser. You can use the `-w` option when you instrument the binary to request that the report be written to an ASCII file or to `stderr`.

When Discover instruments a binary, if it finds any code that it cannot instrument because it is not annotated, it displays a warning like the following:

```
discover: (warning): a.out: 80% of code instrumented (16 out of 20 functions)
```

Non-annotated code could come from assembly language code linked into the binary, or from modules compiled with compilers or on operating systems older than those listed in [“Binaries Must Be Prepared Correctly” on page 11](#).

Caching Shared Libraries

When Discover instruments a binary, it adds code to it that works with the runtime linker to instrument dependent shared libraries when they are loaded at runtime. The instrumented libraries are stored in a cache where they can be reused if the original has not changed since it was last instrumented. By default, the cache directory is `$HOME/SUNW_Bit_Cache`. You can change the directory with the `-D` option.

Instrumenting Shared Libraries

Discover produces the most accurate results if the entire program, including all shared libraries, is instrumented. By default, when you instrument the main executable, Discover inserts code such that when you run the program, Discover automatically instruments all shared libraries as they are opened, whether they are statically linked in the program or dynamically opened by `dlopen()`.

All shared libraries used by the program should be prepared as described in [“Binaries Must Be Prepared Correctly” on page 11](#). By default, if the runtime linker encounters an unprepared library, a fatal error occurs. You can, however, tell Discover to ignore one or more libraries.

Ignoring Libraries

Some libraries might not be possible to prepare, or they might not be instrumentable for some other reason. To provide for this case, with some loss of accuracy, you can tell Discover to ignore these libraries with the `-s`, `-T`, or `-N` option (see “[Instrumentation Options](#)” on page 16, or with specifications in `bit.rc` files (see “[bit.rc Initialization Files](#)” on page 17

If a library cannot be instrumented and is not designated as ignorable, then either Discover fails at instrumentation time or your program fails at runtime with an error message.

By default, Discover uses specifications in the system `bit.rc` file to set certain system and compiler-supplied libraries as ignorable because they are not prepared. The effect on accuracy is minimal because Discover knows the memory characteristics of the most commonly used libraries.

Command Line Options

You can use the following options with the `discover` command to instrument a binary.

Output Options

- `-o file` Write the instrumented binary to *file*. By default, the instrumented binary overwrites the input binary.
- `-w text_file` Write Discover's report on the binary to *text_file*. The file is created when you run the instrumented binary. If *text_file* is a relative pathname, the file is placed relative to the working directory where you run the instrumented binary. To make the filename unique for each time you run the binary, add the string `%p` to the filename to ask the Discover runtime to include the process id. For example, the option `-w report.%p.txt` generates a report file with the filename `report.process_id.txt`. If you include `%p` in the filename more than once, only the first instance is replaced with the process id.

If you do not specify this option or the `-H` option, the report is written in HTML format to *output_file.html*, where *output_file* is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

You can specify both this option and the `-H` option to write the report in both text and HTML formats.

- `-H html_file` Write Discover's report on the binary in HTML format to *html_file*. This file is created when you run the instrumented binary. If *html_file* is a relative pathname, it is placed relative to the working directory where you run the instrumented binary. To make the filename unique for each time you run the

binary, add the string %p to the filename to ask the Discover runtime to include the process id. For example, the option `-H report.%p.html` generates a report file with the filename `report.process_id.html`. If you include %p in the filename more than once, only the first instance is replaced with the process id.

If you do not specify this option or the `-w` option, the report is written in HTML format to `output_file.html`, where `output_file` is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

You can specify both this option and the `-w` option to write the report in both text and HTML formats.

- `-e n` Show only *n* memory errors in the report (default is show all errors).
- `-E n` Show only *n* memory leaks in the report (default is 100).
- `-f` Show offsets in the report (default is to hide them).
- `-m` Show mangled names in the report (default is to show de-mangled names).
- `-S n` Show only *n* stack frames in the report (default is 8).

Instrumentation Options

- `-l` Run Discover in light mode. This option provides faster execution of your program and the program does not have to be specially prepared as described in [“Binaries Must Be Prepared Correctly” on page 11](#), but the number of errors detected is limited.
- `-i` Instrument for data race detection using the Thread Analyzer. When you use this option, only data race detection is done at runtime; no other memory checking is done. The instrumented binary must be run with the `collect` command to generate an experiment that you can view in the Performance Analyzer (see the [Oracle Solaris Studio 12.2: Thread Analyzer User's Guide](#)).
- `-s` Issue a warning, but do not flag an error, if an attempt is made to instrument an uninstrumentable binary.
- `-T` Instrument the named binary only. Do not instrument any dependent shared libraries at runtime.
- `-N library` Do not instrument any dependent shared library matching the prefix *library*. If the initial characters of a library name match *library*, the library is ignored. If *library* begins with a `/`, matching is done on the full absolute pathname of the library. Otherwise, matching is done on the basename of the library.
- `-K` Do not read the `bit.rc` initialization files (see [“bit.rc Initialization Files” on page 17](#)).

Caching Options

- D *cache_directory* Use *cache_directory* as the root directory for storing cached instrumented binaries. By default, the cache directory is `$HOME/SUNW_Bit_Cache`.
- k Force reinstrumentation of any libraries found in the cache.

Other Options

- h or -? Help. Print a short usage message and exit.
- v Verbose. Print a log of what Discover is doing. Repeat the option for more information.
- V Print Discover version information and exit.

bit.rc Initialization Files

Discover initializes its state by reading a series of `bit.rc` files at startup. A system file, `Oracle_Solaris_Studio_installation_directory/prod/lib/postopt/bit.rc`, provides default values for certain variables. Discover reads this file first, followed by `$HOME/.bit.rc` if it exists, and `current_directory/.bit.rc` if it exists.

The `bit.rc` files contain commands to set, append to, and remove from certain variables. When Discover reads a set command, it discards the previous value, if any, of the variable. When it reads an append command, it appends the argument (after a colon separator) to the existing value of the variable. When it reads a remove command, it removes the argument and its colon separator from the existing value of the variable.

The variables set in the `bit.rc` files include the list of libraries to ignore when instrumenting, and lists of functions or function prefixes to ignore when computing the percentage of nonannotated (not prepared) code in a binary.

For more information, refer to the comments in the header of the system `bit.rc` file.

SUNW_DISCOVER_OPTIONS Environment Variable

You can change the runtime behavior of an instrumented binary by setting the `SUNW_DISCOVER_OPTIONS` environment variable to a list of the command-line options `-b`, `-e`, `-E`, `-f`, `-H`, `-l`, `-L`, `-m`, `-S`, and `-w`. For example, if you want to change the number of errors reported to 50 and limit the stack depth in the report to 3, you would set the environment variable to `-e 50 -s 3`.

SUNW_DISCOVER_FOLLOW_FORK_MODE Environment Variable

By default, if a binary you have instrumented with Discover forks while you are running it, Discover continues to collect memory access error data from the parent process. If you want Discover to follow the fork and collect memory access data from the child process, set the `SUNW_DISCOVER_FOLLOW_FORK_MODE` environment variable.

Running an Instrumented Binary

After you have instrumented your binary with Discover, you run it the same way you would ordinarily. Typically, if a particular combination of input causes your program to behave strangely, you would instrument it with Discover and run it with the same input to investigate potential memory problems. While the instrumented program is running, Discover writes information about any memory problems it finds to the specified output files in the selected formats (text, HTML, or both). For information on interpreting the reports, see [“Analyzing Discover Reports” on page 18](#).

Because of the overhead of the instrumentation, your program runs significantly slower after you instrument it. Depending on the frequency of memory access, it might run as much as 50 times slower.

Analyzing Discover Reports

The Discover report provides you with information to effectively pinpoint and fix the problems in your source code.

By default, the report is written in HTML format to `output_file.html`, where `output_file` is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

When you instrument your binary, you can use the `-H` option to request that the HTML output be written to a specified file, or the `-w` option to request that it be written to a text file (see [“Command Line Options” on page 15](#)).

After your binary is instrumented, you can change the settings of the `-H` and `-w` options for the report in the [“SUNW_DISCOVER_OPTIONS Environment Variable” on page 17](#) if, for example, you want to write the report to a different file for a subsequent run of the program.

Analyzing the HTML Report

The HTML report format allows interactive analysis of your program. The data in HTML format can easily be shared between developers using email or placement on a web page. Combined with JavaScript interactive features, it provides a convenient way to navigate through the Discover messages.

The Errors tab (see [“Using the Errors Tab” on page 19](#)), Warnings tab (see [“Using the Warnings Tab” on page 22](#), and Memory Leaks tab (see [“Using the Memory Leaks Tab” on page 23](#)) let you navigate through error messages, warning messages, and the memory leak report, respectively.

The control panel on the left (see [“Using the Control Panel” on page 25](#)) lets you change the contents of the tab that is currently displayed on the right.

Using the Errors Tab

When you first open an HTML report in your browser, the Errors tab is selected and displays the list of memory access errors that occurred during execution of your instrumented binary.

The screenshot displays the Oracle Solaris Studio 12.2 Discover and Uncover interface. At the top, there are three tabs: "Errors", "Warnings", and "Memory Leaks". The "Errors" tab is active, showing two error messages: "UMR: accessing uninitialized data from address 0x50010 (4 bytes)" and "UAW: writing to unallocated memory at address 0x50018 (4 bytes)".

On the left side, there are three panels:

- Stack Trace:** Contains "Expand all" and "Collapse all" buttons.
- Source Code:** Contains "Expand All" and "Collapse All" buttons.
- Show Errors:** A list of error types with checkboxes:
 - ABR
 - ABW
 - BFM
 - BRP
 - CGB
 - DFM
 - FMR
 - FMW
 - FRP
 - IPR
 - IMW
 - IPR
 - SBR
 - SBW
 - UAR
 - UAW
 - UMR

At the bottom left, there is a **Summary** panel showing:

- Errors: 2
- Warnings: 1
- Leaked: 4 Bytes

At the very bottom left, there is a small copyright notice: "Copyright © 2011, 2010, Oracle and/or its affiliates. All rights reserved."

When you click on an error, the stack trace at the time of the error is displayed:

The screenshot displays the Sun Memory Error Discovery Tool (Discover) interface. The main window is titled "Errors" and shows two error types: UMR (Uninitialized Memory Read) and UAW (Unallocated Memory Write). The UMR error is expanded, showing the source code for the function `main() + 0xcc` in `test_UMR.c`. The error message indicates that 4 bytes of uninitialized data were accessed from address `0x50010`. The UAW error is also visible, indicating that 4 bytes of unallocated memory were written to at address `0x50018`.

On the left side, there are three panels: "Stack Trace" with "Expand all" and "Collapse all" buttons; "Source Code" with "Expand All" and "Collapse All" buttons; and "Show Errors" with a list of error types and their checkboxes. The "Show Errors" panel includes checkboxes for ABR, ABW, BFM, BRP, CGB, DFM, FMR, FMW, FRP, IMR, IMW, IR, SBR, SBW, UAR, UAW, and UMR. The "UAW" checkbox is checked.

At the bottom left, there is a "Summary" panel showing the following statistics:

- Errors: 2
- Warnings: 1
- Leaked: 4 Bytes

Copyright © 2009, 2010, Oracle and/or its affiliates. All rights reserved.

If you compiled your code with the `-g` option, you can see the source code for each function in the stack trace by clicking the function:

The screenshot displays the Oracle Solaris Studio 12.2 Discover and Uncover interface. On the left, there are three panels: 'Stack Trace' with 'Expand all' and 'Collapse all' buttons; 'Source Code' with 'Expand All' and 'Collapse All' buttons; and 'Show Errors' with a grid of checkboxes for various error types (ABR, ABW, BFM, BRP, CGB, DFM, FMR, FMV, FRP, IMR, IPW, IPR, SBR, SBW, UAR, UAW, UMR). The 'UAW' checkbox is checked. Below these panels is a 'Summary' box showing 'Errors: 2', 'Warnings: 1', and 'Leaked: 4 Bytes'. At the bottom left, there is a copyright notice: 'Copyright © 2009, 2010. Oracle and/or its affiliates. All rights reserved.'

The main area shows three tabs: 'Errors', 'Warnings', and 'Memory Leaks'. The 'Errors' tab is active, displaying a message: 'UMR: accessing uninitialized data from address 0x50010 (4 bytes)'. Below this, a stack trace is shown for the function `main() + 0xcc (line -9) in "test_UMR.c"`. The stack trace indicates that the data was allocated at (4 bytes) in `main() + 0x1c (line -8) in "test_UMR.c"`. The source code for `main()` is displayed, with line 8 highlighted: `p = (int*) malloc(sizeof(int));`. The stack trace also shows the function `_start() + 0x108`.

The 'Warnings' tab is active, displaying a message: 'UAW: writing to unallocated memory at address 0x50018 (4 bytes)'. The 'Memory Leaks' tab is inactive.

Using the Warnings Tab

The Warnings tab displays all of the warning messages for possible access errors. When you click on a warning, the stack trace at the time of the warning is displayed. If you compiled your code with the `-g` option, you can see the source code for each function in the stack trace by clicking the function.

The screenshot displays the Sun Memory Error Discovery Tool (Discover) interface. On the left, there are three panels: 'Stack Trace' with 'Expand all' and 'Collapse all' buttons; 'Source Code' with 'Expand All' and 'Collapse All' buttons; and 'Show Warnings' with a grid of checkboxes for various warning types (AZS, NAW, UFR, USR, NAR, SMP, UPW, USV). The 'AZS' checkbox is checked. Below these panels is a 'Summary' box showing 'Errors: 2', 'Warnings: 1', and 'Leaked: 4 Bytes'. At the bottom left, there is a copyright notice: 'Copyright © 2009, 2010, Oracle and/or its affiliates. All rights reserved.'

The main area shows three tabs: 'Errors', 'Warnings', and 'Memory Leaks'. The 'Warnings' tab is active, displaying a warning titled 'AZS: allocating zero size memory block'. The warning details are as follows:

- Location: `main() + 0x1a4 (line -11) in "test_UMR.c"`
- Code snippet:


```

8:   p = (int*) malloc(sizeof(int));
9:   printf("**p = %d\n",*p);
10:  p[2]=x;
11:  p = (int*)malloc(x);
12:  }
      
```
- Address: `_start() + 0x108`

Using the Memory Leaks Tab

The Memory Leaks tab displays the total number of blocks remaining allocated at the end of the program's run at the top, with the blocks listed below.



When you click on a block, the stack trace for the block is displayed. If you compiled your code with the -g option, you can see the source code for each function in the stack trace by clicking the function.

The screenshot displays the Sun Memory Error Discovery Tool (Discover) interface. The top navigation bar includes tabs for 'Errors', 'Warnings', and 'Memory Leaks'. The left sidebar contains a 'Stack Trace' section with 'Expand all' and 'Collapse all' buttons, and a 'Source Code' section with 'Expand All' and 'Collapse All' buttons. At the bottom of the sidebar is a 'Summary' section showing 'Errors: 2', 'Warnings: 1', and 'Leaked: 4 Bytes'. The main content area shows a memory leak report for a block at location 0x1c, allocated on the heap with a total size of 4 bytes. Below this, the source code for the function `main() + 0x1c (line -8) in "test_UMR.c"` is displayed. The code includes a comment indicating a warning: `// UMR: accessing uninitialized data`. The line `p = (int*) malloc(sizeof(int));` is highlighted in orange. The code also shows a `printf` statement, an array declaration `z[2] = x;`, and another `malloc` call. The stack trace indicates the leak occurred at `_start() + 0x108`.

Using the Control Panel

To see the stack traces for all of the errors, warnings, and memory leaks, click **Expand All** in the Stack Traces section of the control panel. To see the source code for all of the functions, click **Expand All** in the Source Code section of the control panel.

To hide the stack traces or source code for all of the errors, warnings, and memory leaks, click the corresponding **Collapse All**.

The Show Errors section of the control panel is displayed when the Errors tab is selected and lets you control which types of errors are displayed. By default, the checkboxes for all of the detected errors are checked. To hide a type of error, click its checkbox to remove the checkmark.

The Show Warnings section of the control panel is displayed when the Warnings tab is selected and lets you control which types of warnings are displayed. By default, the checkboxes for all of the detected warnings are checked. To hide a type of warning, click its checkbox to remove the checkmark.

A summary of the report listing the total numbers of errors and warnings, and the amount of leaked memory, is displayed at the bottom of the control panel.

Analyzing the ASCII Report

The ASCII (text) format of the Discover report is suitable for processing by scripts or when you don't have access to a web browser. The following is an example of an ASCII report.

```
$ a.out
```

```
ERROR 1 (UAW): writing to unallocated memory at address 0x50088 (4 bytes) at:
```

```
main() + 0x2a0 <ui.c:20>
17:   t = malloc(32);
18:   printf("hello\n");
19:   for (int i=0; i<100;i++)
20:=>   t[32] = 234; // UAW
21:   printf("%d\n", t[2]); //UMR
22:   foo();
23:   bar();
_start() + 0x108
```

```
ERROR 2 (UMR): accessing uninitialized data from address 0x50010 (4 bytes) at:
```

```
main() + 0x16c <ui.c:21>$
18:   printf("hello\n");
19:   for (int i=0; i<100;i++)
20:   t[32] = 234; // UAW
21:=> printf("%d\n", t[2]); //UMR
22:   foo();
23:   bar();
24:   }
```

```
_start() + 0x108
was allocated at (32 bytes):
```

```
main() + 0x24 <ui.c:17>
14:   x = (int*)malloc(size); // AZS warning
15:   }
16:   int main() {
17:=>   t = malloc(32);
18:   printf("hello\n");
19:   for (int i=0; i<100;i++)
20:   t[32] = 234; // UAW
```

```
_start() + 0x108
```

```
0
```

```
WARNING 1 (AZS): allocating zero size memory block at:
```

```

foo() + 0xf4 <ui.c:14>
11: void foo() {
12:   x = malloc(128);
13:   free(x);
14:=> x = (int*)malloc(size); // AZS warning
15: }
16: int main() {
17:   t = malloc(32);
main() + 0x18c <ui.c:22>
19:   for (int i=0; i<100;i++)
20:     t[32] = 234; // UAW
21:   printf("%d\n", t[2]); //UMR
22:=> foo();
23:   bar();
24: }
_start() + 0x108

```

***** Discover Memory Report *****

1 block at 1 location left allocated on heap with a total size of 128 bytes

```

1 block with total size of 128 bytes
bar() + 0x24 <ui.c:9>
6:       7: void bar() {
8:   int *y;
9:=> y = malloc(128); // Memory leak
10: }
11: void foo() {
12:   x = malloc(128);
main() + 0x194 <ui.c:23>
20:   t[32] = 234; // UAW
21:   printf("%d\n", t[2]); //UMR
22:   foo();
23:=> bar();
24: }
_start() + 0x108

```

ERROR 1: repeats 100 times

DISCOVER SUMMARY:

```

unique errors   : 2 (101 total, 0 filtered)
unique warnings : 1 (1 total, 0 filtered)

```

The report consists of error and warning messages followed by a summary.

The error message starts with the word ERROR and contains a three-letter code, an id number, and an error description (writing to unallocated memory in the example). Other details include the memory address that was accessed and the number or bytes read or written. Following the description is a stack trace at the time of the error that pinpoints the location of the error in the process life cycle.

If the program was compiled with the `-g` option, the stack trace includes the source file name and line number. If the source file is accessible, the source code in the vicinity of the error is printed. The target source line in each frame is indicated by the `=>` symbol.

When the same kind of error at the same memory location with the same number of bytes repeats, the complete message including the stack trace is printed only once. Subsequent occurrences of the error are counted and a repetition count, as shown in the following example, is listed at the end of the report for each identical error that occurs multiple times.

```
ERROR 1: repeats 100 times
```

If the address of the faulty memory access is on the heap, then information on the corresponding heap block is printed after the stack trace. The information includes the block starting address and size, and a stack trace at the time the block was allocated. If the block was freed, a stack trace of the deallocation point is also included.

Warning messages are printed in the same format as error messages except that they start with the word `WARNING`. In general, these messages alert you to conditions that do not affect application correctness, but provide useful information that you can use to improve the program. For example, allocating memory of zero size is not harmful, but if it happens too often, it can potentially degrade performance.

The memory leak report contains information about memory blocks allocated on the heap but not released at program exit. The following is an example of a memory leak report.

```
$ DISCOVER_MEMORY_LEAKS=1 ./a.out
...
***** Discover Memory Report *****

2 blocks left allocated on heap with total size of 44 bytes
  block at 0x50008 (40 bytes long) was allocated at:
    malloc() + 0x168 [libdiscover.so:0xea54]
    f() + 0x1c [a.out:0x3001c]
    <discover_example.c:9>:
      8:      {
      9:=>    int *a = (int *)malloc( n * sizeof(int) );
     10:      int i, j, k;
    main() + 0x1c [a.out:0x304a8]
    <discover_example.c:33>:
     32:      /* Print first N=10 Fibonacci numbers */
     33:=>    a = f(N);
     34:      printf("First %d Fibonacci numbers:\n", N);
    _start() + 0x5c [a.out:0x105a8]
...

```

The first line following the header summarizes the number of heap blocks left allocated on the heap and their total size. The reported size is from the developer's perspective, that is, it does not include the bookkeeping overhead of the memory allocator.

After the memory leak summary, detailed information is printed on each unfreed heap block with a stack trace of its allocation point. The stack trace report is similar to the one described for error and warning messages.

The Discover report is concluded with an overall summary. It reports the number of unique warnings and errors and in parentheses, the total numbers of errors and warnings, including repeated ones. For example:

```
DISCOVER SUMMARY:
    unique errors   : 3 (3 total)
    unique warnings : 1 (5 total)
```

Memory Access Errors and Warnings

Discover detects and reports many memory access errors, as well as warning you about accesses that might be errors.

Memory Access Errors

Discover detects the following memory access errors:

- ABR: beyond array bounds read
- ABW: beyond array bounds write
- BFM: freeing the wrong memory block
- BRP: bad realloc address parameter
- CGB: corrupted array guard block
- DFM: double freeing memory
- FMR: freed memory read
- FMW: freed memory write
- FRP: freed realloc parameter
- IMR: invalid memory read
- IMW: invalid memory write
- PIR: partially initialized read
- SBR: beyond stack frame bounds read
- SBW: beyond stack frame bounds write
- UAR: unallocated memory read
- UAW: unallocated memory write
- UMR: uninitialized memory read

The following sections list some simple sample programs that will produce some of these errors.

ABR

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // ABR: reading memory beyond array bounds at address 0x%1x (%d byte%s)"
```

```
    int *a = (int*) malloc(sizeof(int[5]));
    printf("a[5] = %d\n",a[5]);
}
```

ABW

```
#include <stdlib.h>
int main()
{
    // ABW:  writing to memory beyond array bounds
    int *a = (int*) malloc(sizeof(int[5]));
    a[5] = 5;
}
```

BFM

```
#include <stdlib.h>
int main()
{
    // BFM:  freeing wrong memory block
    int *p = (int*) malloc(sizeof(int));
    free(p+1);
}
```

BRP

```
#include <stdlib.h>
int main()
{
    // BRP is "bad address parameter for realloc 0x%lx"
    int *p = (int*) realloc(0,sizeof(int));
    int *q = (int*) realloc(p+20,sizeof(int[2]));
}
```

DFM

```
#include <stdlib.h>
int main()
{
    // DFM is "double freeing memory"
    int *p = (int*) malloc(sizeof(int));
    free(p);
    free(p);
}
```

FMR

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    // FMR is "reading from freed memory at address 0x%lx (%d byte%s)"
    int *p = (int*) malloc(sizeof(int));
}
```

```

    free(p);
    printf("p = 0x%h\n",p);
}

```

FMW

```

#include <stdlib.h>
int main()
{
    // FMW is "writing to freed memory at address 0x%lx (%d byte%s)"
    int *p = (int*) malloc(sizeof(int));
    free(p);
    *p = 1;
}

```

FRP

```

#include <stdlib.h>
int main()
{
    // FRP: freed pointer passed to realloc
    int *p = (int*) malloc(sizeof(int));
    free(0);
    int *q = (int*) realloc(p, sizeof(int[2]));
}

```

IMR

```

#include <stdlib.h>
int main()
{
    // IMR: read from invalid memory address
    int *p = 0;
    int i = *p;    // generates Signal 11...
}

```

IMW

```

#include <stdlib.h>
int main()
{
    // IMW: write to invalide memory address
    int *p = 0;
    *p = 1;    // generates Signal 11...
}

```

PIR

```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    // PIR: accessing partially initialized data
    int *p = (int*) malloc(sizeof(int));
}

```

```
    *((char*)p) = 'c';  
    printf("*(p = %d\n", *(p+1));  
}
```

UAR

```
#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
    // UAR is "reading from unallocated memory"  
    int *p = (int*) malloc(sizeof(int));  
    printf("*(p+1) = %d\n", *(p+1));  
}
```

UAW

```
#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
    // UAW is "writing to unallocated memory"  
    int *p = (int*) malloc(sizeof(int));  
    *(p+1) = 1;  
}
```

UMR

```
#include <stdio.h>  
#include <stdlib.h>  
int main()  
{  
    // UMR is "accessing uninitialized data from address 0x%1x (A%d byte%s)"  
    int *p = (int*) malloc(sizeof(int));  
    printf("*(p = %d\n", *p);  
}
```

Memory Access Warnings

Discover reports the following memory access warnings:

- AZS: allocating zero size
- NAR: non-annotated read
- NAW: non-annotated write
- SMR: speculative memory read
- SMW: speculative memory write
- UFR: unknown stack frame read
- UFW: unknown stack frame write
- USR: unknown status while reading
- USW: unknown status while writing

The following section lists a simple example program that will produce an AZS warning.

AZS

```
#include <stdlib.h>
int main()
{
    // AZS: allocating zero size memory block
    int *p = malloc();
}
```

Interpreting Discover Error Messages

In some cases, Discover can report an error that is not actually an error. Such cases are called false positives. Discover analyzes code at instrumentation time to reduce the occurrence of false positives compared to similar tools, but there are cases where they still occur. The following sections provide a few tips that might help you to identify and possibly avoid false positives in Discover reports.

Partially Initialized Memory

Bit fields in C and C++ allow you to create compact data types. For example:

```
struct my_struct {
    unsigned int valid : 1;
    char        c;
};
```

In the example, the structure member `my_struct.valid` takes only one bit in memory. However, on SPARC platforms, the CPU can modify memory only in bytes, so the whole byte containing `struct.valid` must be loaded in order to access or modify the structure member. Moreover, sometimes the compiler might find it more efficient to load several bytes (for example, a machine word of four bytes) at once. When Discover detects such a load, without additional information it assumes that all four bytes are used. And if, for example, the field `my_struct.valid` was initialized, but the field `my_struct.c` was not, and the machine word containing both fields was loaded, Discover would flag a partially initialized memory read (PIR).

Another source of false positives is initialization of a bit field. To write a part of a byte, the compiler must first generate code that loads the byte. If the byte was not written prior to a read, the result is an uninitialized memory read error (UMR).

To avoid false positives for bit fields, use the `-g` option or the `-g0` option when compiling. These options provide extra debugging information to Discover to help it identify bit field loads and

initialization, which will eliminate most false positives. If you cannot compile with the `-g` option for some reason, then initialize structures with a function such as `memset()`. For example:

```
...
struct my_struct s;
/* Initialize structure prio to use */
memset(&sm 0, sizeof(struct my_struct));
...
```

Speculative Loads

Sometimes the compiler generates a load from a known memory address under conditions where the result of the load is not valid on all program paths. This situation often occurs on SPARC platforms because such a load instruction can be placed in the delay slot of a branch instruction. For example, here is a C code fragment:

```
int i'
if (foo(&i) != 0) { /* foo returns nonzero if it has initialized i */
    printf("5d\n", i);
}
```

From this code, the compiler could generate code equivalent to:

```
int i;
int t1, t2'
t1 = foo(&i);
t2 = i; /* value in i is loaded */
if (t1 != 0) {
    printf("%d\n", t2);
}
```

Assume that in the example, the function `foo()` returns `0` and does not initialize `i`. The load from `i` is still generated, though not used. But the load will be seen by Discover, which will report a load of an uninitialized variable (UMR).

Discover uses dataflow analysis to identify such cases whenever possible, but sometimes they are impossible to detect.

You can reduce the occurrence of these types of false positives by compiling with a lower optimization level.

Uninstrumented Code

Sometimes it is not possible for Discover to instrument 100% of your program. Perhaps some of your code comes from an assembly language source file or a third-party library that cannot be

recompiled and so cannot be instrumented. Discover has no knowledge of the memory blocks the non-instrumented code is accessing and modifying. Assume for example that a function from a third-party shared library initializes a block of memory that is later read by the main (instrumented) program. Since Discover does not know that the memory has been initialized by the library, the subsequent read generates an uninitialized memory error (UMR).

To provide a solution for such cases, the Discover API includes the following functions:

```
void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
void __ped_memory_copy(unsigned long src, unsigned long dst, long size, unsigned long pc);
```

You can call the API functions from your program to inform Discover of specific events such as a write to a memory area (`__ped_memory_write()`) or a read from a memory area (`__ped_memory_read()`). In both cases, the starting address of the memory area is passed in the `addr` parameter and its size is passed in the `size` parameter. Set the `pc` parameter to `0`.

Use the `__ped_memory_copy` function to inform Discover of memory that is being copied from one location to another. The starting address of the source memory is passed in the `src` parameter, the starting address of the destination area is passed in the `dst` parameter, and the size is passed in the `size` parameter. Set the `pc` parameter to `0`.

To use the API, declare these functions in your program as weak. For example, include the following code fragment in your source code.

```
#ifdef __cplusplus
extern "C" {
#endif

extern void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_copy(unsigned long src, unsigned long dst, long size, unsigned long pc);

#pragma weak __ped_memory_write
#pragma weak __ped_memory_read
#pragma weak __ped_memory_copy

#ifdef __cplusplus
}
#endif
```

The API functions are defined in the internal Discover library, which is linked with your program at instrumentation time. However, when your program is not instrumented, this library is not linked and thus all calls to the API functions will result in application hang-up. So you must disable these functions when you are not running your program under Discover. Alternatively, you can create a dynamic library with empty definitions of the API functions and link it with your program. In this case, when you run your program without Discover, your library will be used, but when you run it under Discover, the real API functions will be called automatically.

Limitations When Using Discover

Only Annotated Code is Instrumented

Discover can instrument only code that has been prepared as described in [“Binaries Must Be Prepared Correctly” on page 11](#). Non-annotated code might come from assembly language code linked into the binary, or from modules compiled with older compilers or operating systems than those listed in that section.

Specifically excluded from preparation are assembly language modules and functions that contain `asm` statements or `.il` templates.

Machine Instruction Might Differ From Source Code

Discover operates on machine code. The tool detects errors on machine instructions such as loads and stores, and correlates the errors with the source code. Some source code statements do not have associated machine instructions, so it may appear that Discover did not detect an obvious user error. For example, consider the following C code fragment:

```
int *p = (int *)malloc(sizeof(int));
int i;

i = *p; /* compiler may not generate code for this statement */
printf("Hello World!\n");

return;
```

Reading a value stored at the address pointed to by `p` is a potential user error since the memory was not initialized. However, an optimizing compiler will detect that the variable `i` is not used, so the code for the statement reading from memory and assigning to `i` will not be generated. In this case, Discover will not report uninitialized memory usage (UMR).

Compiler Options Affect the Generated Code

Compiler-generated code is not always as you expect it to be. Because the code the compiler generates varies depending on the compiler options you use, including the `-On` optimization options, the errors reported by Discover might also vary. For example, errors reported in code generated at the `-O1` optimization level could disappear for code generated at the `-O4` optimization level.

System Libraries Can Affect the Errors Reported

System libraries are preinstalled with the operating system and cannot be recompiled for instrumentation. Discover provides support for the common function from the standard C library (`libc.so`); that is, Discover knows what memory is accessed or modified by these functions. However, if your application uses other system libraries, you might see false positives in the Discover report. If false positives are reported, you can call the Discover API from your code to eliminate them.

Custom Memory Management Can Affect the Accuracy of the Data

Discover can track heap memory when it is allocated by standard programming language mechanisms like `malloc()`, `calloc()`, `free()`, `operator new()`, and `operator delete()`.

If your application uses a custom memory management system working on top of the standard functions (for example, pool allocation management implemented with `malloc()`), then Discover works, but is not guaranteed to correctly report leaks or access to freed memory.

Discover does not support the following memory allocators:

- Custom heap allocators that use `brk(2)` or `sbrk(2)` system calls directly
- Standard heap management function linked statically into a binary
- Memory allocated from the user code using `mmap(2)` and `shmget(2)` system calls

The `sigaltstack(2)` function is not supported.

Out of Bounds Errors for Static and Automatic Arrays Cannot Be Detected

Because of the algorithms that Discover uses to detect array bounds, it is not possible to detect out of bounds access errors for static and automatic (local) arrays. Errors can be detected only for dynamically allocated arrays.

Code Coverage Tool (Uncover)

- “Requirements for Using Uncover” on page 39
- “Using Uncover” on page 39
- “Understanding the Coverage Report in the Performance Analyzer” on page 42
- “Understanding the ASCII Coverage Report” on page 47
- “Understanding the HTML Coverage Report” on page 51

Requirements for Using Uncover

Uncover works on binaries compiled with the Sun Studio 12 Update 1, Oracle Solaris Studio Express 6/10, or Oracle Solaris Studio 12.2 compilers, or the GCC for Sun Systems 4.2.0 or later compilers, with at least the `-O1` optimization option, on a SPARC-based or x86-based system running the Solaris 10 5/08 operating system or a later Solaris 10 update.

A binary compiled as described includes information that Uncover uses to reliably disassemble the binary to instrument it for coverage data collection.

Using the `-g` option to generate debug information when compiling the binary allows Uncover to use source code level coverage information. If your binary is not compiled with the `-g` option, Uncover uses only program counter (PC) based coverage information.

Using Uncover

Generating coverage information using Uncover is a three-step process:

1. Instrumenting the binary
2. Running the instrumented binary
3. Generating and viewing coverage reports

Instrumenting the Binary

The input binary can be an executable or a shared library. You must instrument each binary you want to analyze separately.

You instrument the binary with the `uncover` command. For example, the following command instruments the binary `a.out` and overwrites the input `a.out` with the instrumented `a.out`. It also creates a directory with the suffix `.uc` (`a.out.uc` in this case) in which the coverage data will be collected. A copy of the input binary is saved in this directory.

```
uncover a.out
```

You can use the `-d directory` option to tell Uncover to create the coverage data directory in *directory*. This option is useful when you are collecting coverage data for multiple binaries, so that all of the coverage data directories are created in the same directory. Also, if you run different instances of the same instrumented binary from different locations, using this option ensures that the coverage data from all of these runs is accumulated in the same coverage data directory.

If you do not use the `-d` option, the coverage data directory is created in the current run directory.

If you run the `uncover` command on a input binary that is already instrumented, Uncover issues an error message telling you that the binary cannot be instrumented because it is already instrumented, and that you can run it to generate coverage data.

Running the Instrumented Binary

After you have instrumented your binary, you can run it normally. Every time you run the instrumented binary, code coverage data is collected in the coverage data directory with the `.uc` suffix that Uncover created during the instrumentation. Since Uncover data collection is multi-thread safe and multi-process safe, there is no restriction on the number of simultaneous runs or threads in the process. The coverage data is accumulated over all of the runs and threads.

Generating and Viewing the Coverage Report

To generate a coverage report, run the `uncover` command on the coverage data directory. For example:

```
uncover a.out.uc
```

This command generates an Oracle Solaris Studio Performance Analyzer experiment directory called `binary_name.er` from the coverage data in the `a.out.uc` directory, starts the

Performance Analyzer GUI, and displays the experiment. If you have an `.er.rc` file (see the *Oracle Solaris Studio 12.2 Performance Analyzer* manual) in the current directory or your home directory, it might affect the way the Analyzer displays the experiment.

You can also use `uncover` command options to generate the report as HTML and view it in your web browser, or as ASCII to view in a terminal window:

<code>-e on off</code>	Generate experiment directory for the coverage report and display the experiment in the Performance Analyzer GUI. On by default.
<code>-H <i>html_directory</i></code>	Save the coverage data as HTML in the specified directory and automatically display it in your web browser. Off by default.
<code>-h or -?</code>	Help.
<code>-n</code>	Generate coverage reports but do not start viewers like the Performance Analyzer or web browser.
<code>-t <i>ascii_file</i></code>	Generate an ASCII coverage report in the specified file. Off by default.
<code>-V</code>	Print Uncover version and exit.
<code>-v</code>	Verbose. Print a log of what Uncover is doing.

Only one output format is enabled, so if you specify multiple output options, Discover uses the last option in the command.

Examples

`uncover a.out`

This command instruments the binary `a.out`, overwrites the input `a.out`, creates an `a.out.uc` coverage data directory in the current directory, and saves a copy of the input `a.out` in the `a.out.uc` directory. If `a.out` is already instrumented, a warning message is displayed and no instrumentation is done.

`uncover -d coverage a.out`

This command does everything that the first example does, except it creates the `a.out.uc` coverage directory in the directory `coverage`.

`uncover a.out.uc`

This command uses the data in the `a.out.uc` coverage directory to create a code coverage experiment (`a.out.er`) in your working directory, and starts the Performance Analyzer GUI to display the experiment.

`uncover -H a.out.html a.out.uc`

This command uses the data in the `a.out.uc` coverage directory to create an HTML code coverage report in the directory `a.out.html` and displays the report in your web browser.

```
uncover -t a.out.txt a.out.uc
```

This command uses the data in the `a.out.uc` coverage directory to create an ASCII code coverage report in the file `a.out.txt`.

Understanding the Coverage Report in the Performance Analyzer

By default, when you run the `uncover` command on the coverage directory, the coverage report is opened as an experiment in the Oracle Solaris Studio Performance Analyzer. The Analyzer uses the Functions, Source, Disassembly, and Inst-Freq tabs to display the coverage data.

The Functions Tab

When you open the coverage report in the Analyzer, the Functions tab is selected. The tab displays columns listing the Uncoverage, Function Count, Instr Exec, Block Covered %, and Instr Covered % counters for each function. You can make any column the sort key for the data by clicking on the column header. Clicking the arrow on the column header reverses the sort order.

The screenshot displays the Oracle Solaris Studio Performance Analyzer interface. The main window shows a table of functions with columns for Uncoverage, Function Count, Instr Exec, Block Covered %, and Instr Covered %. The 'main' function is highlighted at the top of the list.

Uncoverage	Function Count	Instr Exec	Block Covered %	Instr Covered %	Name
22 424	212 276 980	\$ 030 976 791	3 370	3 158	<Total>
2 076	0	24	5	14	main
1 596	0	0	0	0	forkchild
1 144	0	0	0	0	pagethresh
1 116	0	42	33	47	endcases
1 000	0	0	0	0	forkcopy
980	0	0	0	0	iofile
748	0	0	0	0	do_vforkexec
732	0	0	0	0	callso
712	0	1 603	27	13	commandline
708	0	0	0	0	do_forkexec
648	0	0	0	0	callso
644	0	0	0	0	sigprof
644	0	0	0	0	sigprofh
556	0	0	0	0	do_chdir
548	0	0	0	0	correlate
492	0	0	0	0	do_popen
472	0	62	25	26	atpwtch_print
384	0	0	0	0	so_cpuline
384	0	0	0	0	so_cpuline
348	0	0	0	0	itimer_realprof
336	0	0	0	0	ldso
328	0	2	33	18	get_ocpus
304	0	0	0	0	hrv
300	0	0	0	0	do_sysatem
300	0	23	33	30	sigtime
300	0	0	0	0	so_burncpu
300	0	0	0	0	so_burncpu
284	0	156	37	33	get_clock_rate
276	0	0	0	0	masksignals
268	0	23	33	34	gpf
268	0	0 054	40	59	atpwtch_stop
256	0	0	0	0	sigprof_handler
256	0	0	0	0	sigprof_sigaction
248	0	18	14	18	init_alarm_actr

The Summary panel on the right shows details for the selected object 'main':

- Name: main
- PC Address: 0:0x00005050
- Size: 624
- Source File: jcc/Aten/Uncover/symprof/symprof.c
- Object File: .out.er/archives/a.out_8F8U5q3Zd
- Load Object: <a.out>
- Mangled Name:
- Aliases:

Metrics for Selected Object:

	Exclusive	Inclusive
Uncoverage:	2076 (0.26%)	2076 (0.26%)
Function Count:	0 (0.0%)	0 (0.0%)
Instr Exec:	24 (0.00%)	24 (0.00%)
Block Covered %:	5 (0.15%)	5 (0.15%)
Instr Covered %:	14 (0.44%)	14 (0.44%)

The Uncoverage Counter

The Uncoverage metric is a very powerful feature of Uncover. If you use this column as the sort key, in decreasing order, the top functions in the display are the functions that offer the greatest potential to increase coverage. In the example, the `main()` function is at the top of the list because it has the largest number in the Uncoverage column. (The `sigprof()` and `sigprofh()` functions all have the same number, so they are listed in alphabetical order.)

The Uncoverage number for the `main()` function is number of bytes of code that could potentially be covered if a test is added to the suite that causes the function to be called. The amount that coverage would actually increase varies according to the structure of the function. If there are no branches in the function, and all the functions it calls are also straight line functions, then coverage will indeed increase by the stated number of bytes. But in general, the coverage increase is less than the potential, perhaps much less.

The uncovered functions with non-zero values in the Uncoverage column are called root uncovered functions, meaning that they are all called by covered functions. Functions that are called only by non-root uncovered functions do not have their own uncoverage numbers. It is presumed that these functions will be either covered, or revealed as uncovered, in subsequent runs, as the test suite is improved to cover the high-potential uncovered functions.

The coverage numbers are non-exclusive.

The Function Count Counter

The Function Count is the number of times a function was called, but in the context of coverage analysis, the actual count is irrelevant. All that matters is whether the count is zero or non-zero. If the count is zero, the function is not covered. If the count is non-zero, the function is covered. If any instruction in the function is executed, the function is considered to be covered.

You can detect non-top-level uncovered functions in this column. If the Function Count for a function is zero and the Uncoverage number is also zero, the function is not a top-level covered function.

The Instr Exec Counter

The Instr Exec counter displays the dynamic instruction count. In the Functions tab, it shows the total number of instructions executed for each function. This counter also appears in the Source tab (see [“The Source Tab” on page 45](#)) and the Disassembly tab (see [“The Disassembly Tab” on page 46](#)).

The Block Covered % Counter

For each function, the Block Covered % counter displays the percentage of basic blocks in the function that are covered. This number gives you an idea of how well the function is covered. Disregard this number in the <Total> row; it is the sum of percentages in the column and is meaningless.

The Instr Covered % Counter

For each function, the Instr Covered % counter displays the percentage of instructions in the function that are covered. This number also gives you an idea of how well the function is covered. Disregard this number in the <Total> row; it is the sum of percentages in the column and is meaningless.

The Source Tab

If you compiled your binary with the `-g` option, the Source tab displays the source code of your program. Because Uncover instruments your program at the binary level, and you have compiled the program with optimization, the coverage information in this tab can be puzzling to interpret.

The Instr Exec counter in the Source tab shows the total number of instructions executed for each source line, which is essentially the statement level code coverage information. A non-zero value implies that the statement is covered; a zero value means that the statement is not covered. Variable declarations and comments have no Instr Exec counts.

The screenshot shows the Oracle Solaris Studio Performance Analyzer interface. The main window displays the Source tab for a C program. The coverage report is as follows:

Uncoverage	Function Count	Instr Exec	Block Covered %	Instr Covered %
2076	0	0	0	14

The source code is displayed on the right side of the window, showing a function named `main` with various declarations and assignments. The Instr Exec counter is highlighted in yellow for the line `main(int argc, char **ar`.

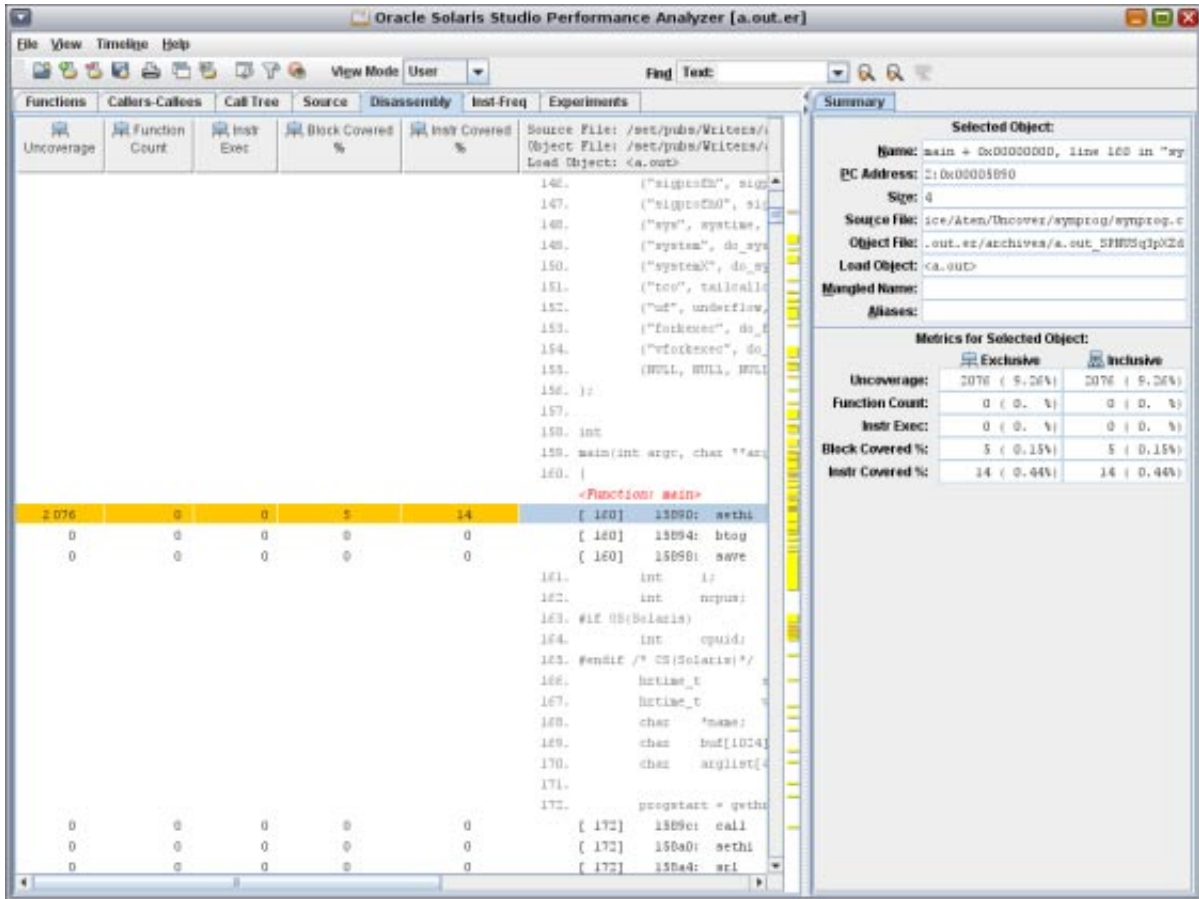
The Summary panel on the right shows the Selected Object: `line 120 in "myprog.c"`. The Metrics for Selected Object are:

	Exclusive	Inclusive
Uncoverage:	2076 (9.26%)	2076 (9.26%)
Function Count:	0 (0. %)	0 (0. %)
Instr Exec:	0 (0. %)	0 (0. %)
Block Covered %:	0 (0.15%)	0 (0.15%)
Instr Covered %:	14 (0.44%)	14 (0.44%)

The Disassembly Tab

If you select a line in the Source tab, and then select the Disassembly tab, the Analyzer tries to find the selected line in the binary and display its disassembly.

The Instr Exec counter in this tab shows the number of times each instruction was executed.



The Inst-Freq Tab

The Inst-Freq tab displays the overall coverage summary.

The screenshot shows the Oracle Solaris Studio Performance Analyzer interface. The main window displays a table of functions with columns for Functions, Callers/Calloes, Call Tree, Source, Disassembly, Inst Freq, and Experiments. The right-hand panel shows a Summary for the Selected Object, including Name, PC Address, Size, Source File, Object File, Load Object, and a table of Metrics for Selected Object.

Functions	Callers/Calloes	Call Tree	Source	Disassembly	Inst Freq	Experiments
call	47335672 (7.0)				0	0
all	42798155 (7.1)				0	0
ldf	36319634 (6.0)				1600003	
ctx	33465905 (5.5)				0	154223558
lds	31163801 (4.2)				0	2
exl	304369191 (5.0)				0	0
ctx	23601254 (3.5)				0	1600001
lddf	200172473 (3.3)				0	31
save	18427685 (3.1)				0	0
nop	155566976 (2.8)				0	155566976
stw	9401070 (1.4)				0	1
sub	6850017 (1.1)				0	0
lx	66791300 (1.1)				0	0
subcc	64791182 (1.1)				0	0
ldw	4601268 (0.8)				0	6
fabdd	2329027 (0.4)				0	0
fdtas	2200756 (0.4)				0	0
fatod	1600031 (0.3)				0	0
fmuls	1200723 (0.2)				0	0
scdf	6242027 (0.1)				0	0
jmp1	6002127 (0.1)				0	0
restore	6002111 (0.1)				0	6002111
fitod	6000027 (0.1)				0	0
mov	6000003 (0.1)				0	0
fatosa	6000000 (0.1)				0	0
umul	6000000 (0.1)				0	0
fitos	6000000 (0.1)				0	0
andb	6000000 (0.1)				0	0
fdtas	1292164 (0.0)				0	0
fdtod	1290226 (0.0)				0	0
fmuld	1290190 (0.0)				0	0
ldsb	588 (0.0)				0	0
orcc	57 (0.0)				0	0
fdjwd	49 (0.0)				0	0
sth	16 (0.0)				0	0
fcapped	0 (0.0)				0	0
thi	0 (0.0)				0	0
lpt	6 (0.0)				0	0
axa	3 (0.0)				0	0
fdhos	2 (0.0)				0	0
mulr	1 (0.0)				0	0
xxx	1 (0.0)				0	0

Summary

Selected Object:

Name: main + 0x0000000, line 100 in "xy"
PC Address: 210x0005500
Size: 4
Source File: /usr/atom/Uncover/ymprprog/ymprprog.o
Object File: .out.gz/archives/a.out_59M5q3p2d
Load Object: <a.out>
Bridged Name:
Aliases:

Metrics for Selected Object:

	Exclusive	Inclusive
Uncoverage:	2076 (9.26%)	2076 (9.26%)
Function Count:	0 (0. %)	0 (0. %)
Inst Exec:	0 (0. %)	0 (0. %)
Block Covered %:	5 (0.15%)	5 (0.15%)
Inst Covered %:	14 (0.44%)	14 (0.44%)

Understanding the ASCII Coverage Report

If you specify the `-t` option when you generate the coverage report from the coverage data directory, Uncover writes a coverage report to the specified ASCII (text file).

```
UNCOVER Code Coverage
Total Functions: 95
Covered Functions: 58
Function Coverage: 61.1%
Total Basic Blocks: 568
Covered Basic Blocks: 258
Basic Block Coverage: 45.4%
Total Basic Block Executions: 564,812,760
Average Executions per Basic Block: 994,388.66
Total Instructions: 6,201
Covered Instructions: 3,006
Instruction Coverage: 48.5%
```

Total Instruction Executions: 4,760,934,518
 Average Executions per Instruction: 767,768.83
 Number of times this program was executed: unavailable
 Functions sorted by metric: Exclusive Uncoverage

Excl. Uncoverage	Excl. Function Count	Excl. Block Covered %	Excl. Instr Covered %	Name
13404	6004876	5464	5384	<Total>
1036	0	0	0	main
980	0	0	0	iofile
748	0	0	0	do_vforkexec
732	0	0	0	callso
708	0	0	0	do_forkexec
648	0	0	0	callsx
644	0	0	0	sigprof
644	0	0	0	sigprofh
556	0	0	0	do_chdir
548	0	0	0	correlate
492	0	0	0	do_popen
404	0	0	0	pagethrash
384	0	0	0	so_cputime
384	0	0	0	sx_cputime
348	0	0	0	itimer_realprof
336	0	0	0	ldso
304	0	0	0	hrv
300	0	0	0	do_system
300	0	0	0	do_burncpu
300	0	0	0	sx_burncpu
288	0	0	0	forkcopy
276	0	0	0	masksignals
256	0	0	0	sigprof_handler
256	0	0	0	sigprof_sigaction
216	0	0	0	do_exec
196	0	0	0	iotest
176	0	0	0	closeso
156	0	0	0	gethrustime
144	0	0	0	forkchild
144	0	0	0	gethrpxtime
136	0	0	0	whrlog
112	0	0	0	masksig
92	0	0	0	closesx
84	0	0	0	reapchildren
36	0	0	0	reapchild
32	0	0	0	doabort
8	0	0	0	csig_handler
0	1	66	72	acct_init
0	1	100	100	bounce
0	63	100	96	bounce_a
0	60	100	100	bounce-b
0	16	71	58	check_sigmask
0	1	83	77	commandline
0	1	100	98	cputime
0	1	100	98	dousleep
0	1	100	100	endcases
0	1	100	95	ext_inline_code
0	1	100	96	ext_macro_code
0	1	100	99	fitos

0	2	81	80	get_clock_rate
0	1	100	100	get_ncpus
0	1	100	100	gpf
0	1	100	100	gpf_a
0	1	100	100	gpf_b
0	10	100	93	gpf_work
0	1	100	97	icputime
0	1	100	96	inc_body
0	1	100	96	inc_brace
0	1	100	95	inc_entry
0	1	100	95	inc_exit
0	1	100	96	inc_func
0	1	100	94	inc_middle
0	1	57	72	init_micro_acct
0	1	50	43	initksig
0	1	100	95	inline_code
0	1	100	95	macro_code
0	1	100	98	muldiv
0	6000000	100	100	my_irand
0	1	100	98	naptime
0	19	50	83	prdelta
0	21	100	100	prhrdelta
0	21	100	100	prhrvdelta
0	1	100	100	prtime
0	552	100	98	real_recurse
0	1	100	100	recurse
0	1	100	100	recursedeeep
0	1	100	95	s_inline_code
0	1	100	100	sigtime
0	1	100	95	sigtime_handler
0	19	100	100	snaptod
0	1	100	100	so_init
0	2	66	75	stpwtch_alloc
0	1	100	100	stpwtch_calibrate
0	2	75	66	stpwtch_print
0	2002	100	100	stpwtch_start
0	2000	90	91	stpwtch_stop
0	1	100	100	sx_init
0	1	100	99	systeme
0	3	100	95	tailcall_a
0	3	100	95	tailcall_b
0	3	100	95	tailcall_c
0	1	100	100	tailcallopt
0	1	100	97	underflow
0	21	75	71	whrvlog
0	19	100	100	wlog

Instruction frequency data from experiment a.out.er

Instruction frequencies of /export/home1/synprog/a.out.uc

Instruction	Executed	()
TOTAL	4760934518	(100.0)
float ops	2383657378	(50.1)
float ld st	1149983523	(24.2)
load store	1542440573	(32.4)
load	882693735	(18.5)
store	659746838	(13.9)

Instruction	Executed ()	Annulled	In Delay Slot
TOTAL	4760934518 (100.0)		
add	713013787 (15.0)	16	1501335
subcc	558774858 (11.7)	0	6002
br	558769261 (11.7)	0	0
stf	432500661 (9.1)	726	36299281
ldf	408226488 (8.6)	40	103000396
faddd	391230847 (8.2)	0	0
fdtos	366200726 (7.7)	0	0
fstod	360200000 (7.6)	0	0
lddf	288250336 (6.1)	500	282200229
stw	138028738 (2.9)	26002	25974065
lduw	118004305 (2.5)	71	94000270
ldx	68212446 (1.4)	0	2000
stx	68211370 (1.4)	7	23532716
fitod	36026002 (0.8)	0	0
sethi	36002986 (0.8)	0	228
fdtoi	30000001 (0.6)	0	0
fdivd	26000088 (0.5)	0	0
call	22250348 (0.5)	0	0
srl	21505246 (0.5)	0	21
stdf	21006038 (0.4)	0	0
or	19464766 (0.4)	0	10981277
fmuls	6004907 (0.3)	0	0
jmpl	6004853 (0.1)	0	0
save	6004852 (0.1)	0	0
restore	6002294 (0.1)	0	6004852
sub	6000019 (0.1)	0	0
xor	6000000 (0.1)	0	0
fitos	6000000 (0.1)	0	0
fstoi	6000000 (0.1)	0	0
and	6000000 (0.1)	0	0
andn	6000000 (0.1)	0	0
sll	3505225 (0.1)	0	0
nop	3505219 (0.1)	0	3505219
fxtod	7763 (0.0)	0	0
bpr	6000 (0.0)	0	0
fcmped	4837 (0.0)	0	0
fbr	4837 (0.0)	0	0
fmuld	2850 (0.0)	0	0
orcc	383 (0.0)	0	0
sra	241 (0.0)	0	0
ldsb	160 (0.0)	0	0
mulx	87 (0.0)	0	0
stb	31 (0.0)	0	0
mov	21 (0.0)	0	0
fdtox	15 (0.0)	0	0

=====

Understanding the HTML Coverage Report

The HTML report is similar to the report displayed in the Performance Analyzer.

```
HTML data from experiment(s):
a.out.exe

Functions sorted by metric: Exclusive Uncoverage
```

Excl. Uncoverage	Excl. Function Count	Excl. Instr Exec	Excl. Block Covered %	Excl. Instr Covered %	Name
20340	6004793	4398402003	3424	5430	<Total>
1680	0	0	0	0	[trimmed] iofile <small>src Caller-callee</small>
1316	0	0	0	0	[trimmed] do_forbuser <small>src Caller-callee</small>
1300	0	0	0	0	[trimmed] do_vforbuser <small>src Caller-callee</small>
1056	0	0	0	0	[trimmed] callas <small>src Caller-callee</small>
956	0	0	0	0	[trimmed] do_sopen <small>src Caller-callee</small>
940	0	0	0	0	[trimmed] sigprok <small>src Caller-callee</small>
940	0	0	0	0	[trimmed] sigprokH <small>src Caller-callee</small>
932	0	0	0	0	[trimmed] curvalate <small>src Caller-callee</small>
832	0	0	0	0	[trimmed] do_chdir <small>src Caller-callee</small>
800	0	0	0	0	[trimmed] openatash <small>src Caller-callee</small>
694	0	0	0	0	[trimmed] as_creatas <small>src Caller-callee</small>
694	0	0	0	0	[trimmed] as_creatas <small>src Caller-callee</small>
612	0	0	0	0	[trimmed] do_sysvms <small>src Caller-callee</small>
596	0	0	0	0	[trimmed] itimer_readdir <small>src Caller-callee</small>
572	0	0	0	0	[trimmed] ldev <small>src Caller-callee</small>
540	0	0	0	0	[trimmed] openatash2 <small>src Caller-callee</small>
532	0	0	0	0	[trimmed] lrv <small>src Caller-callee</small>
520	0	0	0	0	[trimmed] forbcomr <small>src Caller-callee</small>
528	0	0	0	0	[trimmed] as_burncom <small>src Caller-callee</small>
528	0	0	0	0	[trimmed] as_burncom <small>src Caller-callee</small>
512	0	0	0	0	sigprok_sigaction
496	0	0	0	0	sigprok_handler
484	0	0	0	0	do_err
440	0	0	0	0	iofile
312	0	0	0	0	wholeg
280	0	0	0	0	closeo
244	0	0	0	0	forchild
220	0	0	0	0	gethrptime
212	0	0	0	0	gethrustime
144	0	0	0	0	waitsig
140	0	0	0	0	closeo
92	0	0	0	0	reapchildren
90	0	0	0	0	doabort
88	0	0	0	0	reapchild
20	0	0	0	0	sig_handler
0	1	58	46	73	acct_init
0	1	131	100	100	[trimmed] bounce <small>src Caller-callee</small>
0	21	25600457	100	93	[trimmed] bounce_a <small>src Caller-callee</small>
0	20	260	100	100	[trimmed] bounce_b <small>src Caller-callee</small>
0	1	79	33	33	collo
0	16	563	71	40	check_sigmask
0	1	8928	88	73	crossedline
0	1	24800423	100	98	[trimmed] creatias <small>src Caller-callee</small>
0	1	23800744	100	99	[trimmed] duplino <small>src Caller-callee</small>
0	1	159	100	100	[trimmed] endomas <small>src Caller-callee</small>
0	1	8000022	100	98	[trimmed] set_inline_code <small>src Caller-callee</small>
0	1	9800020	100	97	[trimmed] set_macro_code <small>src Caller-callee</small>
0	1	143011931	100	98	[trimmed] fsize <small>src Caller-callee</small>
0	2	10470	76	67	get_clock_rate

If you click the function name link or the `trimmed` link for a function, the disassembly data for that function is displayed.

```

current filename for subsequent output: a.out.html/FILE_3F.dsp
Current archive: a.bit_INCP/a.bit_funcs/a.bit_x.a.bit_BCM../a.bit_ICP_0000
Current Sort Metric: Exclusive Coverage ( a.bit_INCP )
Source File: ioajv.c
Object File: a.out.as/archivas/a.out_LJUN3p5hm02
Load Object: a.out.as/archivas/a.out_LJUN3p5hm02

  Sect.      Sect.      Sect.      Sect.      Sect.
  Coverage  Function  Instr  Block  Instr
  Count      Count      Row    Covered # Covered #

1. /* Copyright 03/13/08 Sun Microsystems, Inc. All Rights Reserved */
2.
3. #include <stdio.h>
4. #include <stdlib.h>
5. #include <errno.h>
6. #include <sys/types.h>
7. #include <sys/stat.h>
8. #include <fcntl.h>
9. #include <unistd.h>
10. #include "stopwatch.h"
11.
12. /* constants defining various tests */
13. #define BUFSIZE      16384
14. #define NFILES      1024
15.
16. /*=====*/
17. /* ioFile - do some file io operations */
18. int
19. ioFile()
20. |
21. |<Function: ioFile>
22. | [ ?] 14e40: movl $0,-640,%sp
23. | 440 0 0 0 0 0 istest + 0x00000000
24. | 0 0 0 0 0 0 [ ?] 14e44: movl $0,%eax
25. | 0 0 0 0 0 0 [ ?] 14e48: movl $0,%eax
26. | 0 0 0 0 0 0 [ ?] 14e4c: add $0,%eax
27. | 0 0 0 0 0 0 subtotals for skipped section ...
28. |
29. | 24. char buf;
30. | 25. hrtime_t start;
31. | 26. hrtime_t vstart;
32. | 27. char *fname = "/usr/tmp/ajvprogXXXXX";
33. | 28. int ret;
34. | 29.
35. | 30. start = gethrtime();
36. | 31. vstart = gethrvtime();
37. | 32.
38. | 33. /* Log the event */
39. | 34. wlog("start of ioFile == %d", NULL);
40. | 35.
41. | 36. ret = write(buf, NULL);
42. | 37.
43. | 38. subtotals for skipped section ...
44. | 39.
45. | 40. buf, NULL);
46. | 41.
47. | 42. /* now reopen the file, and read it */
48. | 43. start = gethrtime();
49. | 44. vstart = gethrvtime();

```

If you click the Caller - callee link for a function, the Caller-Callee data is displayed.

Function Names: <Total>

current filenames for subsequent output: a.out.html/calls

Functions sorted by metric: Exclusive Uncoverage

Callers and callees sorted by metric: Attributed Uncoverage

Attr. Uncoverage	Attr. Function Count	Attr. Instr Exec	Attr. Block Covered %	Attr. Instr Covered %	Name
20240	6004793	6199402002	5424	5430	*<Total>
1690	0	0	0	0	acfile
1316	0	0	0	0	do_forbuser
1200	0	0	0	0	do_cfarkosac
1036	0	0	0	0	callc
956	0	0	0	0	do_sosm
940	0	0	0	0	siomocf
940	0	0	0	0	siomocf
932	0	0	0	0	corralate
832	0	0	0	0	do_dslir
800	0	0	0	0	seowthreash
684	0	0	0	0	as_spectac
684	0	0	0	0	as_spectac
612	0	0	0	0	do_syster
596	0	0	0	0	itisc_rsalncf
572	0	0	0	0	lida
560	0	0	0	0	seabimole
552	0	0	0	0	hac
528	0	0	0	0	forbeser
528	0	0	0	0	as_burney
520	0	0	0	0	as_burney
512	0	0	0	0	siomocf_sioaction
496	0	0	0	0	siomocf_bandler
484	0	0	0	0	do_smc
440	0	0	0	0	actact
312	0	0	0	0	whrlow
280	0	0	0	0	classe
244	0	0	0	0	forchid
220	0	0	0	0	seathracine
212	0	0	0	0	seathracine
184	0	0	0	0	seabash
140	0	0	0	0	classer
92	0	0	0	0	reatchildren
80	0	0	0	0	doabact
88	0	0	0	0	reatchid
20	0	0	0	0	reac_bandler
0	1	58	66	73	act_act
0	1	131	100	100	hosmca
0	21	256000457	100	93	hosmca_a
0	20	260	100	100	hosmca_b
0	1	79	33	33	callc
0	16	562	71	40	check_siomocf
0	1	826	89	72	comenoline
0	1	240000623	100	96	sealims
0	1	238000744	100	98	dosalms
0	1	159	100	100	siomocf

Limitations When Using Uncover

Only Annotated Code Can Be Instrumented

Uncover can instrument only code that has been prepared as described in “Requirements for Using Uncover” on page 39. Non-annotated code might come from assembly language code linked into the binary, or from modules compiled with older compilers or operating systems than those listed in that section.

Specifically excluded from preparation are assembly language modules and functions that contain `asm` statements or `.il` templates.

Machine Instructions Might Differ From Source Code

Uncover operates on machine code. It finds coverage of machine instructions and then correlates this coverage with source code. Some source code statements do not have associated machine instructions, so Uncover might appear to not report coverage for such statements. For example, consider the following code fragment:

```
#define A 100
#define B 200
...
    if (A>B) {
        ...
    }
```

You might expect Uncover to report a non-zero execution count for the `if` statement, but the compiler is likely to remove this code, so Uncover will not see it during instrumentation. So no coverage will be reported for these instructions.

Index

A

accessible documentation, 6

B

binaries

instrumented with Discover

changing the runtime behavior of, 17

running, 18

writing to a specific file, 15

instrumented with Uncover, running, 40

instrumenting for Discover, 14–18

instrumenting for Uncover, 40

preparing for Discover, 11–12

that cannot be used by Discover, 12

bit.rc initialization files, 17

telling Discover not to read, 16

D

Discover

API, 35

following fork, 18

forcing reinstrumentation of cached libraries, 17

ignoring shared libraries, 15, 16

instrumenting the named binary only, 16

issuing a warning if an attempt is made to

instrument an uninstrumentable binary, 16

limitations, 36–37

memory access error examples, 29

Discover (*Continued*)

memory access errors, 29–32

memory access warnings, 32–33

options

-D, 14, 17

-E, 16

-e, 16

-f, 16

-H, 15, 18

-h, 17

-i, 16

-K, 16

-k, 17

-l, 16

-m, 16

-N, 15, 16

-o, 15

-S, 16

-s, 16

-T, 15, 16

-V, 17

-v, 17

-w, 14, 15, 18

overview, 9–10

requirements for using, 11–12

running in light mode, 16

specifying cache directory, 17

specifying verbose mode, 17

Discover reports

ASCII, 26–29

error messages, 27

heap blocks left allocated, 28

Discover reports, ASCII (*Continued*)

- memory leaks, 28
 - stack trace, 27, 28
 - summary, 29
 - unfreed heap blocks, 28
 - warning messages, 28
 - writing, 15
 - error messages, interpreting, 33
 - false positives, 33
 - avoiding, 33
 - caused by partially initialized memory, 33–34
 - caused by speculative loads, 34
 - caused by uninstrumented code, 34
 - HTML, 19–26
 - control panel, 25–26
 - controlling types of errors displayed, 26
 - controlling types of warnings displayed, 26
 - Errors tab, 19–21
 - Memory Leaks tab, 23–24
 - number of blocks remaining allocated, 23
 - showing all stack traces, 25
 - showing source code, 21, 22, 24
 - showing source code for all functions, 25
 - showing stack trace, 20, 22, 24
 - Warnings tab, 22
 - writing, 15
 - limiting number of memory errors reported, 16
 - limiting number of memory leaks reported, 16
 - limiting number of stack frames shown in, 16
 - showing mangled names in, 16
 - showing offsets in, 16
- documentation, accessing, 6
- documentation index, 6

I

- instrumenting a binary
 - for data race detection with Discover, 16
 - for Discover, 14–18
 - for uncover, 40

N

- non-annotated code
 - how Discover treats, 14
 - sources of, 14

R

- requirements
 - Discover, 11–12
 - Uncover, 39

S

- shared libraries
 - caching by Discover, 14
 - instrumenting with Discover, 14
 - telling Discover to ignore, 15, 16
- SUNW_DISCOVER_FOLLOW_FORK_MODE environment variable, 18
- SUNW_DISCOVER_OPTIONS environment variable, 17, 18

U

- Uncover
 - command examples, 41–42
 - coverage report, generating, 40–41
 - limitations, 53–54
 - options
 - d, 40
 - e, 41
 - H, 41
 - h, 41
 - n, 41
 - t, 41
 - V, 41
 - v, 41
 - overview, 10
 - requirements for using, 39
 - running in verbose mode, 41
- Uncover ASCII coverage report, 47–50
 - generating, 41

Uncover coverage report for the Performance

Analyzer, 42–46

Disassembly tab, 46

Functions tab, 42–44

Block Covered % counter, 44

Function Count counter, 44

Instr Covered % counter, 44

Instr Exec counter, 44

Uncoverage counter, 43–44

generating, 41

Inst-Freq tab, 46

Source tab, 45

Uncover HTML coverage report, 51–52

saving, 41

