

Dialyzer

Copyright © 2006-2025 Ericsson AB. All Rights Reserved.
Dialyzer 4.3.1.1
December 10, 2025

Copyright © 2006-2025 Ericsson AB. All Rights Reserved.
Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.apache.org/licenses/LICENSE-2.0 Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License. Ericsson AB. All Rights Reserved..

December 10, 2025

1 Dialyzer User's Guide

1.1 Dialyzer

1.1.1 Introduction

Scope

Dialyzer is a static analysis tool that identifies software discrepancies, such as definite type errors, code that has become dead or unreachable because of programming error, and unnecessary tests, in single Erlang modules or entire (sets of) applications.

Dialyzer can be called from the command line, from Erlang, and from a GUI.

Prerequisites

It is assumed that the reader is familiar with the Erlang programming language.

1.1.2 The Persistent Lookup Table

Dialyzer stores the result of an analysis in a Persistent Lookup Table (PLT). The PLT can then be used as a starting point for later analyses. It is recommended to build a PLT with the Erlang/OTP applications that you are using, but also to include your own applications that you are using frequently.

The PLT is built using option `--build_plt` to Dialyzer. The following command builds the recommended minimal PLT for Erlang/OTP:

```
dialyzer --build_plt --apps erts kernel stdlib mnesia
```

Dialyzer looks if there is an environment variable called `DIALYZER_PLT` and places the PLT at this location. If no such variable is set, Dialyzer places the PLT at `$HOME/.dialyzer_plt`. The placement can also be specified using the options `--plt` or `--output_plt`.

Information can be added to an existing PLT using option `--add_to_plt`. If you also want to include the Erlang compiler in the PLT and place it in a new PLT, then use the following command:

```
dialyzer --add_to_plt --apps compiler --output_plt my.plt
```

Then you can add your favorite application my_app to the new PLT:

```
dialyzer --add_to_plt --plt my.plt -r my_app/ebin
```

But you realize that it is unnecessary to have the Erlang compiler in this one:

```
dialyzer --remove_from_plt --plt my.plt --apps compiler
```

Later, when you have fixed a bug in your application my_app, you want to update the PLT so that it becomes fresh the next time you run Dialyzer. In this case, run the following command:

```
dialyzer --check_plt --plt my.plt
```

Dialyzer then reanalyzes the changed files and the files that depend on these files. Notice that this consistency check is performed automatically the next time you run Dialyzer with this PLT. Option `--check_plt` is only for doing so without doing any other analysis.

To get information about a PLT, use the following option:

```
dialyzer --plt_info
```

To specify which PLT, use option `--plt`.

To get the output printed to a file, use option `--output_file`.

Notice that when manipulating the PLT, no warnings are emitted. To turn on warnings during (re)analysis of the PLT, use option `--get_warnings`.

1.1.3 Using Dialyzer from the Command Line

Dialyzer has a command-line version for automated use. See `dialyzer(3)`.

1.1.4 Using Dialyzer from Erlang

Dialyzer can also be used directly from Erlang. See `dialyzer(3)`.

1.1.5 Using Dialyzer from the GUI

Choosing the Applications or Modules

The **File** window displays a listing of the current directory. Click your way to the directories/modules you want to add or type the correct path in the entry.

Mark the directories/modules you want to analyze for discrepancies and click **Add**. You can either add the `.beam` and `.erl` files directly, or add directories that contain these kind of files. Notice that you are only allowed to add the type of files that can be analyzed in the current mode of operation (see below), and that you cannot mix `.beam` and `.erl` files.

Analysis Modes

Dialyzer has two analysis modes: "Byte Code" and "Source Code". They are controlled by the buttons in the top-middle part of the main window, under **Analysis Options**.

Controlling the Discrepancies Reported by Dialyzer

Under the **Warnings** pull-down menu, there are buttons that control which discrepancies are reported to the user in the **Warnings** window. By clicking these buttons, you can enable/disable a whole class of warnings. Information about the classes of warnings is found on the "Warnings" item under the **Help** menu (in the rightmost top corner).

If modules are compiled with inlining, spurious warnings can be emitted. In the **Options** menu you can choose to ignore inline-compiled modules when analyzing byte code. When starting from source code, this is not a problem because inlining is explicitly turned off by Dialyzer. The option causes Dialyzer to suppress all warnings from inline-compiled modules, as there is currently no way for Dialyzer to find what parts of the code have been produced by inlining.

Running the Analysis

Once you have chosen the modules or directories you want to analyze, click the **Run** button to start the analysis. If you for some reason want to stop the analysis while it is running, click the **Stop** button.

The information from the analysis is displayed in the **Log** window and the **Warnings** window.

Include Directories and Macro Definitions

When analyzing from source, you might have to supply Dialyzer with a list of include directories and macro definitions (as you can do with the `erlc` flags `-I` and `-D`). This can be done either by starting Dialyzer with these flags from the command line as in:

```
dialyzer -I my_includes -DDEBUG -Dvsn=42 -I one_more_dir
```

or by adding these explicitly using submenu **Manage Macro Definitions** or **Manage Include Directories** in the **Options** menu.

Saving the Information on the Log and Warnings Windows

The **File** menu includes options to save the contents of the **Log** window and the **Warnings** window. Simply choose the options and enter the file to save the contents in.

There are also buttons to clear the contents of each window.

Inspecting the Inferred Types of the Analyzed Functions

Dialyzer stores the information of the analyzed functions in a Persistent Lookup Table (PLT), see section The Persistent Lookup Table.

After an analysis, you can inspect this information. In the **PLT** menu you can choose to either search the PLT or inspect the contents of the whole PLT. The information is presented in EDoc format.

1.1.6 Feedback and Bug Reports

We very much welcome user feedback - even wishlists! If you notice anything weird, especially if Dialyzer reports any discrepancy that is a false positive, please send an error report describing the symptoms and how to reproduce them.

2 Reference Manual

dialyzer

Erlang module

Dialyzer is a static analysis tool that identifies software discrepancies, such as definite type errors, code that has become dead or unreachable because of programming error, and unnecessary tests, in single Erlang modules or entire (sets of) applications.

Dialyzer starts its analysis from either debug-compiled BEAM bytecode or from Erlang source code. The file and line number of a discrepancy is reported along with an indication of what the discrepancy is about. Dialyzer bases its analysis on the concept of success typings, which allows for sound warnings (no false positives).

Using Dialyzer from the Command Line

Dialyzer has a command-line version for automated use. This section provides a brief description of the options. The same information can be obtained by writing the following in a shell:

```
dialyzer --help
```

For more details about the operation of Dialyzer, see section Using Dialyzer from the GUI in the User's Guide.

Exit status of the command-line version:

`0`

No problems were found during the analysis and no warnings were emitted.

`1`

Problems were found during the analysis.

`2`

No problems were found during the analysis, but warnings were emitted.

Usage:

```
dialyzer [--add_to_plt] [--apps applications] [--build_plt]
         [--check_plt] [-Ddefine]* [-Dname] [--dump_callgraph file]
         [files_or_dirs] [--fullpath] [--get_warnings] [--gui] [--help]
         [-I include_dir]* [--no_check_plt] [--no_indentation] [--no_native]
         [--no_native_cache] [-o outfile] [--output_plt file] [-pa dir]*
         [--plt plt] [--plt_info] [--plts plt*] [--quiet] [-r dirs]
         [--raw] [--remove_from_plt] [--shell] [--src] [--statistics]
         [--verbose] [--version] [-Wwarn]*
```

Note:

* denotes that multiple occurrences of the option are possible.

Options:

`--add_to_plt`

The PLT is extended to also include the files specified with `-c` and `-r`. Use `--plt` to specify which PLT to start from, and `--output_plt` to specify where to put the PLT. Notice that the analysis possibly can include files from the PLT if they depend on the new files. This option only works for BEAM files.

`--apps applications`

This option is typically used when building or modifying a PLT as in:

```
dialyzer --build_plt --apps erts kernel stdlib mnesia ...
```

to refer conveniently to library applications corresponding to the Erlang/OTP installation. However, this option is general and can also be used during analysis to refer to Erlang/OTP applications. File or directory names can also be included, as in:

```
dialyzer --apps inets ssl ./ebin ../other_lib/ebin/my_module.beam
```

`--build_plt`

The analysis starts from an empty PLT and creates a new one from the files specified with `-c` and `-r`. This option only works for BEAM files. To override the default PLT location, use `--plt` or `--output_plt`.

`--check_plt`

Check the PLT for consistency and rebuild it if it is not up-to-date.

`-Dname` (or `-Dname=value`)

When analyzing from source, pass the define to Dialyzer. (**)

`--dump_callgraph file`

Dump the call graph into the specified file whose format is determined by the filename extension. Supported extensions are: `raw`, `dot`, and `ps`. If something else is used as filename extension, default format `.raw` is used.

`files_or_dirs` (for backward compatibility also as `-c files_or_dirs`)

Use Dialyzer from the command line to detect defects in the specified files or directories containing `.erl` or `.beam` files, depending on the type of the analysis.

`--fullpath`

Display the full path names of files for which warnings are emitted.

`--get_warnings`

Make Dialyzer emit warnings even when manipulating the PLT. Warnings are only emitted for files that are analyzed.

`--gui`

Use the GUI.

`--help` (or `-h`)

Print this message and exit.

`-I include_dir`

When analyzing from source, pass the `include_dir` to Dialyzer. (**)

`--no_check_plt`

Skip the PLT check when running Dialyzer. This is useful when working with installed PLTs that never change.

`--no_indentation`

Do not insert line breaks in types, contracts, and Erlang Code when formatting warnings.

`--no_native` (or `-nn`)

Bypass the native code compilation of some key files that Dialyzer heuristically performs when dialyzing many files. This avoids the compilation time, but can result in (much) longer analysis time.

`--no_native_cache`

By default, Dialyzer caches the results of native compilation in directory `$XDG_CACHE_HOME/erlang/dialyzer_hipe_cache`. `XDG_CACHE_HOME` defaults to `$HOME/.cache`. Use this option to disable caching.

`-o outfile` (or `--output outfile`)

When using Dialyzer from the command line, send the analysis results to the specified outfile rather than to `stdout`.

`--output_plt file`

Store the PLT at the specified file after building it.

`-pa dir`

Include `dir` in the path for Erlang. This is useful when analyzing files that have `-include_lib()` directives.

`--plt plt`

Use the specified PLT as the initial PLT. If the PLT was built during setup, the files are checked for consistency.

`--plt_info`

Make Dialyzer print information about the PLT and then quit. The PLT can be specified with `--plt(s)`.

`--plts plt*`

Merge the specified PLTs to create the initial PLT. This requires that the PLTs are disjoint (that is, do not have any module appearing in more than one PLT). The PLTs are created in the usual way:

```
dialyzer --build_plt --output_plt plt_1 files_to_include
...
dialyzer --build_plt --output_plt plt_n files_to_include
```

They can then be used in either of the following ways:

```
dialyzer files_to_analyze --plts plt_1 ... plt_n
```

or

```
dialyzer --plts plt_1 ... plt_n -- files_to_analyze
```

Notice the `--` delimiter in the second case.

`--quiet` (or `-q`)

Make Dialyzer a bit more quiet.

`-r dirs`

Same as `files_or_dirs`, but the specified directories are searched recursively for subdirectories containing `.erl` or `.beam` files in them, depending on the type of analysis.

`--raw`

When using Dialyzer from the command line, output the raw analysis results (Erlang terms) instead of the formatted result. The raw format is easier to post-process (for example, to filter warnings or to output HTML pages).

`--remove_from_plt`

The information from the files specified with `-c` and `-r` is removed from the PLT. Notice that this can cause a reanalysis of the remaining dependent files.

`--shell`

Do not disable the Erlang shell while running the GUI.

`--src`

Override the default, which is to analyze BEAM files, and analyze starting from Erlang source code instead.

`--statistics`

Print information about the progress of execution (analysis phases, time spent in each, and size of the relative input).

`--verbose`

Make Dialyzer a bit more verbose.

`--version` (or `-v`)

Print the Dialyzer version and some more information and exit.

`-Wwarn`

A family of options that selectively turn on/off warnings. (For help on the names of warnings, use `dialyzer -Whelp`.) Notice that the options can also be specified in the file with a `-dialyzer()` attribute. For details, see section Requesting or Suppressing Warnings in Source Files.

Note:

** options `-D` and `-I` work both from the command line and in the Dialyzer GUI; the syntax of defines and includes is the same as that used by erlc(1).

Warning options:

`-Werror_handling` (***)

Include warnings for functions that only return by an exception.

`-Wno_behaviours`

Suppress warnings about behavior callbacks that drift from the published recommended interfaces.

`-Wno_contracts`

Suppress warnings about invalid contracts.

`-Wno_fail_call`

Suppress warnings for failing calls.

`-Wno_fun_app`

Suppress warnings for fun applications that will fail.

`-Wno_improper_lists`

Suppress warnings for construction of improper lists.

`-Wno_match`

Suppress warnings for patterns that are unused or cannot match.

`-Wno_missing_calls`

Suppress warnings about calls to missing functions.

`-Wno_opaque`

Suppress warnings for violations of opacity of data types.

`-Wno_return`

Suppress warnings for functions that will never return a value.

`-Wno_undefined_callbacks`

Suppress warnings about behaviors that have no `-callback` attributes for their callbacks.

`-Wno_unused`

Suppress warnings for unused functions.

`-Wrace_conditions` (***)

Include warnings for possible race conditions. Notice that the analysis that finds data races performs intra-procedural data flow analysis and can sometimes explode in time. Enable it at your own risk.

`-Wunderspecs` (***)

Warn about underspecified functions (the specification is strictly more allowing than the success typing).

`-Wunknown` (***)

Let warnings about unknown functions and types affect the exit status of the command-line version. The default is to ignore warnings about unknown functions and types when setting the exit status. When using Dialyzer from Erlang, warnings about unknown functions and types are returned; the default is not to return these warnings.

`-Wunmatched_returns` (***)

Include warnings for function calls that ignore a structured return value or do not match against one of many possible return values. However, no warnings are included if the possible return values are a union of atoms or a union of numbers.

The following options are also available, but their use is not recommended (they are mostly for Dialyzer developers and internal debugging):

`-Woverspecs` (***)

Warn about overspecified functions (the specification is strictly less allowing than the success typing).

`-Wspecdiffs` (***)

Warn when the specification is different than the success typing.

Note:

*** denotes options that turn on warnings rather than turning them off.

Using Dialyzer from Erlang

Dialyzer can be used directly from Erlang. Both the GUI and the command-line versions are also available. The options are similar to the ones given from the command line, see section Using Dialyzer from the Command Line.

Requesting or Suppressing Warnings in Source Files

Attribute `-dialyzer()` can be used for turning off warnings in a module by specifying functions or warning options. For example, to turn off all warnings for the function `f/0`, include the following line:

```
-dialyzer({nowarn_function, f/0}).
```

To turn off warnings for improper lists, add the following line to the source file:

```
-dialyzer(no_improper_lists).
```

Attribute `-dialyzer()` is allowed after function declarations. Lists of warning options or functions are allowed:

```
-dialyzer([{nowarn_function, [f/0]}, no_improper_lists]).
```

Warning options can be restricted to functions:

```
-dialyzer({no_improper_lists, g/0}).
```

```
-dialyzer({[no_return, no_match], [g/0, h/0]}).
```

For help on the warning options, use `dialyzer -Whelp`. The options are also enumerated, see function `gui/1` below (`WarnOpts`).

Note:

Warning option `-Wrace_conditions` has no effect when set in source files.

Attribute `-dialyzer()` can also be used for turning on warnings. For example, if a module has been fixed regarding unmatched returns, adding the following line can help in assuring that no new unmatched return warnings are introduced:

```
-dialyzer(unmatched_returns).
```

Exports

`format_warning(Msg) -> string()`

Types:

`Msg = {Tag, Id, msg()}`

See `run/1`.

Get a string from warnings as returned by `run/1`.

`format_warning(Msg, Options) -> string()`

Types:

`Msg = {Tag, Id, msg()}`

See `run/1`.

`Options = [{indent_opt, boolean()}]`

Get a string from warnings as returned by `run/1`.

If `indent_opt` is set to `true` (default), line breaks are inserted in types, contracts, and Erlang code to improve readability.

`gui() -> ok | {error, Msg}`
`gui(OptList) -> ok | {error, Msg}`

Types:

`OptList`

See below.

Dialyzer GUI version.

```
OptList  :: [Option]
Option   :: {files,          [Filename :: string()]}
          | {files_rec,      [DirName :: string()]}
          | {defines,        [{Macro :: atom(), Value :: term()}]}
          | {from,           src_code | byte_code}
                               %% Defaults to byte_code
          | {init_plt,       FileName :: string()}
                               %% If changed from default
          | {plts,           [FileName :: string()]}
                               %% If changed from default
          | {include_dirs,   [DirName :: string()]}
          | {output_file,    FileName :: string()}
          | {output_plt,     FileName :: string()}
          | {check_plt,      boolean()},
          | {analysis_type,  'succ_typings' |
                             'plt_add' |
                             'plt_build' |
                             'plt_check' |
                             'plt_remove'}
          | {warnings,       [WarnOpts]}
          | {get_warnings,   boolean()}
          | {native,         boolean()}
                               %% Defaults to false when invoked from Erlang
          | {native_cache,   boolean()}

WarnOpts :: error_handling
          | no_behaviours
          | no_contracts
          | no_fail_call
          | no_fun_app
          | no_improper_lists
          | no_match
   | no_missing_calls
          | no_opaque
   | no_return
          | no_undefined_callbacks
          | no_unused
          | race_conditions
          | underspecs
          | unknown
          | unmatched_returns
          | overspecs
          | specdiffs
```

`plt_info(string()) -> {'ok', [{atom(), any()}]} | {'error', atom()}`

Returns information about the specified PLT.

`run(OptList) -> Warnings`

Types:

`OptList`

See `gui/0,1`.

`Warnings`

See below.

Dialyzer command-line version.

```
Warnings :: [{Tag, Id, Msg}]
Tag      :: 'warn_behaviour'
          | 'warn_bin_construction'
          | 'warn_callgraph'
          | 'warn_contract_not_equal'
          | 'warn_contract_range'
          | 'warn_contract_subtype'
          | 'warn_contract_supertype'
          | 'warn_contract_syntax'
          | 'warn_contract_types'
          | 'warn_failing_call'
          | 'warn_fun_app'
          | 'warn_matching'
          | 'warn_non_proper_list'
          | 'warn_not_called'
          | 'warn_opaque'
          | 'warn_race_condition'
          | 'warn_return_no_exit'
          | 'warn_return_only_exit'
          | 'warn_umatched_return'
          | 'warn_undefined_callbacks'
          | 'warn_unknown'
Id  = {File :: string(), Line :: integer()}
Msg = msg() -- Undefined
```

typer

Command

TypEr shows type information for Erlang modules to the user. Additionally, it can annotate the code of files with such type information.

Using TypEr from the Command Line

TypEr is used from the command-line. This section provides a brief description of the options. The same information can be obtained by writing the following in a shell:

```
typer --help
```

Usage:

```
typer [--help] [--version] [--plt PLT] [--edoc]
      [--show | --show-exported | --annotate | --annotate-inc-files]
      [-Ddefine]* [-I include_dir]* [-pa dir]* [-pz dir]*
      [-T application]* [-r] file*
```

Note:

* denotes that multiple occurrences of the option are possible.

Options:

`-r`

Search directories recursively for .erl files below them.

`--show`

Print type specifications for all functions on stdout. (This is the default behaviour; this option is not really needed.)

`--show-exported` (or `show_exported`)

Same as `--show`, but print specifications for exported functions only. Specs are displayed sorted alphabetically on the function's name.

`--annotate`

Annotate the specified files with type specifications.

`--annotate-inc-files`

Same as `--annotate` but annotates all `-include()` files as well as all .erl files. (Use this option with caution - it has not been tested much).

`--edoc`

Print type information as Edoc `@spec` comments, not as type specs.

`--plt`

Use the specified dialyzer PLT file rather than the default one.

`-T file*`

The specified file(s) already contain type specifications and these are to be trusted in order to print specs for the rest of the files. (Multiple files or dirs, separated by spaces, can be specified.)

`-Dname` (or `-Dname=value`)

Pass the defined name(s) to TypEr. (**)

`-I`

Pass the include_dir to TypEr. (**)

`-pa dir`

Include `dir` in the path for Erlang. This is useful when analyzing files that have `-include_lib()` directives or use parse transforms.

`-pz dir`

Include `dir` in the path for Erlang. This is useful when analyzing files that have `-include_lib()` directives or use parse transforms.

`--version` (or `-v`)

Print the TypEr version and some more information and exit.

Note:

** options `-D` and `-I` work both from the command line and in the TypEr GUI; the syntax of defines and includes is the same as that used by erlc(1).