Coccinelle

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\dev-tools\(linux-master) (Documentation) (dev-tools)coccinelle.rst`, line 5)

Unknown directive type "highlight".

```
.. highlight:: none
```

Coccinelle is a tool for pattern matching and text transformation that has many uses in kernel development, including the application of complex, tree-wide patches and detection of problematic programming patterns.

Getting Coccinelle

The semantic patches included in the kernel use features and options which are provided by Coccinelle version 1.0.0-rc11 and above. Using earlier versions will fail as the option names used by the Coccinelle files and coccicheck have been updated.

Coccinelle is available through the package manager of many distributions, e.g. :

- Debian
- Fedora
- Ubuntu
- OpenSUSE
- Arch Linux
- NetBSD
- FreeBSD

Some distribution packages are obsolete and it is recommended to use the latest version released from the Coccinelle homepage at http://coccinelle.lip6.fr/

Or from Github at:

https://github.com/coccinelle/coccinelle

Once you have it, run the following commands:

```
./autogen
./configure
make
```

as a regular user, and install it with:

```
sudo make install
```

More detailed installation instructions to build from source can be found at:

https://github.com/coccinelle/coccinelle/blob/master/install.txt

Supplemental documentation

For supplemental documentation refer to the wiki:

https://bottest.wiki.kernel.org/coccicheck

The wiki documentation always refers to the linux-next version of the script.

For Semantic Patch Language(SmPL) grammar documentation refer to:

http://coccinelle.lip6.fr/documentation.php

Using Coccinelle on the Linux kernel

A Coccinelle-specific target is defined in the top level Makefile. This target is named `coccicheck` and calls the `coccicheck` front-end in the `scripts` directory.

Four basic modes are defined: `patch`, `report`, `context`, and `org`. The mode to use is specified by setting the MODE variable with `MODE=<mode>`.

- `patch` proposes a fix, when possible.
- `report` generates a list in the following format: file:line:column-column: message
- `context` highlights lines of interest and their context in a diff-like style. Lines of interest are indicated with `-`.
- `org` generates a report in the Org mode format of Emacs.

Note that not all semantic patches implement all modes. For easy use of Coccinelle, the default mode is "report".

Two other modes provide some common combinations of these modes.

- `chain` tries the previous modes in the order above until one succeeds.
- `rep+ctxt` runs successively the report mode and the context mode. It should be used with the C option (described later) which checks the code on a file basis.

Examples

To make a report for every semantic patch, run the following command:

```
make coccicheck MODE=report
```

To produce patches, run:

```
make coccicheck MODE=patch
```

The coccicheck target applies every semantic patch available in the sub-directories of `scripts/coccinelle` to the entire Linux kernel.

For each semantic patch, a commit message is proposed. It gives a description of the problem being checked by the semantic patch, and includes a reference to Coccinelle.

As with any static code analyzer, Coccinelle produces false positives. Thus, reports must be carefully checked, and patches reviewed.

To enable verbose messages set the V= variable, for example:

```
make coccicheck MODE=report V=1
```

Coccinelle parallelization

By default, coccicheck tries to run as parallel as possible. To change the parallelism, set the J= variable. For example, to run across 4 CPUs:

```
make coccicheck MODE=report J=4
```

As of Coccinelle 1.0.2 Coccinelle uses Ocaml parmap for parallelization; if support for this is detected you will benefit from parmap parallelization.

When parmap is enabled coccicheck will enable dynamic load balancing by using `--chunksize 1` argument. This ensures we keep feeding threads with work one by one, so that we avoid the situation where most work gets done by only a few threads. With dynamic load balancing, if a thread finishes early we keep feeding it more work.

When parmap is enabled, if an error occurs in Coccinelle, this error value is propagated back, and the return value of the `make coccicheck` command captures this return value.

Using Coccinelle with a single semantic patch

The optional make variable COCCI can be used to check a single semantic patch. In that case, the variable must be initialized with the name of the semantic patch to apply.

For instance:

```
make coccicheck COCCI=<my_SP.cocci> MODE=patch
```

or:

```
make coccicheck COCCI=<my_SP.cocci> MODE=report
```

Controlling Which Files are Processed by Coccinelle

By default the entire kernel source tree is checked.

To apply Coccinelle to a specific directory, `M=` can be used. For example, to check drivers/net/wireless/ one may write:

```
make coccicheck M=drivers/net/wireless/
```

To apply Coccinelle on a file basis, instead of a directory basis, the C variable is used by the makefile to select which files to work with. This variable can be used to run scripts for the entire kernel, a specific directory, or for a single file.

For example, to check drivers/bluetooth/bfusb.c, the value 1 is passed to the C variable to check files that make considers need to be compiled.:

```
make C=1 CHECK=scripts/coccicheck drivers/bluetooth/bfusb.o
```

The value 2 is passed to the C variable to check files regardless of whether they need to be compiled or not.:

```
make C=2 CHECK=scripts/coccicheck drivers/bluetooth/bfusb.o
```

In these modes, which work on a file basis, there is no information about semantic patches displayed, and no commit message proposed.

This runs every semantic patch in scripts/coccinelle by default. The COCCI variable may additionally be used to only apply a single semantic patch as shown in the previous section.

The "report" mode is the default. You can select another one with the MODE variable explained above.

Debugging Coccinelle SmPL patches

Using coccicheck is best as it provides in the spatch command line include options matching the options used when we compile the kernel. You can learn what these options are by using V=1; you could then manually run Coccinelle with debug options added.

Alternatively you can debug running Coccinelle against SmPL patches by asking for stderr to be redirected to stderr. By default stderr is redirected to /dev/null; if you'd like to capture stderr you can specify the `DEBUG_FILE="file.txt"` option to coccicheck. For instance:

```
rm -f cocci.err
make coccicheck COCCI=scripts/coccinelle/free/kfree.cocci MODE=report DEBUG_FILE=cocci.err
cat cocci.err
```

You can use SPFLAGS to add debugging flags; for instance you may want to add both --profile --show-trying to SPFLAGS when debugging. For example you may want to use:

```
rm -f err.log
```

```
export COCCI=scripts/coccinelle/misc/irqf_oneshot.cocci
make coccicheck DEBUG_FILE="err.log" MODE=report SPFLAGS="--profile --show-trying" M=./drivers/mfd
```

err.log will now have the profiling information, while stdout will provide some progress information as Coccinelle moves forward with work.

NOTE:

DEBUG_FILE support is only supported when using coccinelle >= 1.0.2.

Currently, DEBUG_FILE support is only available to check folders, and not single files. This is because checking a single file requires spatch to be called twice leading to DEBUG_FILE being set both times to the same value, giving rise to an error.

.cocciconfig support

Coccinelle supports reading .cocciconfig for default Coccinelle options that should be used every time spatch is spawned. The order of precedence for variables for .cocciconfig is as follows:

- Your current user's home directory is processed first
- Your directory from which spatch is called is processed next
- The directory provided with the --dir option is processed last, if used

Since coccicheck runs through make, it naturally runs from the kernel proper dir; as such the second rule above would be implied for picking up a .cocciconfig when using `make coccicheck`.

`make coccicheck` also supports using M= targets. If you do not supply any M= target, it is assumed you want to target the entire kernel. The kernel coccicheck script has:

```
if [ "$KBUILD_EXTMOD" = "" ] ; then
    OPTIONS="--dir $srctree $COCCIINCLUDE"
else
    OPTIONS="--dir $KBUILD_EXTMOD $COCCIINCLUDE"
fi
```

KBUILD_EXTMOD is set when an explicit target with M= is used. For both cases the spatch --dir argument is used, as such third rule applies when whether M= is used or not, and when M= is used the target directory can have its own .cocciconfig file. When M= is not passed as an argument to coccicheck the target directory is the same as the directory from where spatch was called.

If not using the kernel's coccicheck target, keep the above precedence order logic of .cocciconfig reading. If using the kernel's coccicheck target, override any of the kernel's .coccicheck's settings using SPFLAGS.

We help Coccinelle when used against Linux with a set of sensible default options for Linux with our own Linux .cocciconfig. This hints to coccinelle that git can be used for `git grep` queries over coccigrep. A timeout of 200 seconds should suffice for now.

The options picked up by coccinelle when reading a .cocciconfig do not appear as arguments to spatch processes running on your system. To confirm what options will be used by Coccinelle run:

```
spatch --print-options-only
```

You can override with your own preferred index option by using SPFLAGS. Take note that when there are conflicting options Coccinelle takes precedence for the last options passed. Using .cocciconfig is possible to use idutils, however given the order of precedence followed by Coccinelle, since the kernel now carries its own .cocciconfig, you will need to use SPFLAGS to use idutils if desired. See below section "Additional flags" for more details on how to use idutils.

Additional flags

Additional flags can be passed to spatch through the SPFLAGS variable. This works as Coccinelle respects the last flags given to it when options are in conflict.

```
make SPFLAGS=--use-glimpse coccicheck
```

Coccinelle supports idutils as well but requires coccinelle >= 1.0.6. When no ID file is specified coccinelle assumes your ID database file is in the file .id-utils.index on the top level of the kernel. Coccinelle carries a script scripts/idutils_index.sh which creates the database with:

```
mkid -i C --output .id-utils.index
```

If you have another database filename you can also just symlink with this name.

```
make SPFLAGS=--use-idutils coccicheck
```

Alternatively you can specify the database filename explicitly, for instance:

```
make SPFLAGS="--use-idutils /full-path/to/ID" coccicheck
```

See `spatch --help` to learn more about spatch options.

Note that the `--use-glimpse` and `--use-idutils` options require external tools for indexing the code. None of them is thus active by default. However, by indexing the code with one of these tools, and according to the cocci file used, spatch could proceed the entire code base more quickly.

SmPL patch specific options

SmPL patches can have their own requirements for options passed to Coccinelle. SmPL patch-specific options can be provided by providing them at the top of the SmPL patch, for instance:

```
// Options: --no-includes --include-headers
```

SmPL patch Coccinelle requirements

As Coccinelle features get added some more advanced SmPL patches may require newer versions of Coccinelle. If an SmPL patch requires a minimum version of Coccinelle, this can be specified as follows, as an example if requiring at least Coccinelle >= 1.0.5:

```
// Requires: 1.0.5
```

Proposing new semantic patches

New semantic patches can be proposed and submitted by kernel developers. For sake of clarity, they should be organized in the sub-directories of `scripts/coccinelle/`.

Detailed description of the `report` mode

`report` generates a list in the following format:

```
file:line:column-column: message
```

Example

Running:

```
make coccicheck MODE=report COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@r depends on !context && !patch && (org || report)@
expression x;
position p;
@@

  ERR_PTR@p(PTR_ERR(x))

@script:python depends on report@
p << r.p;
x << r.x;
@@

msg="ERR_CAST can be used with %s" % (x)
coccilib.report.print_report(p[0], msg)
</smpl>
```

This SmPL excerpt generates entries on the standard output, as illustrated below:

```
/home/user/linux/crypto/ctr.c:188:9-16: ERR_CAST can be used with alg
/home/user/linux/crypto/authenc.c:619:9-16: ERR_CAST can be used with auth
/home/user/linux/crypto/xts.c:227:9-16: ERR_CAST can be used with alg
```

Detailed description of the `patch` mode

When the `patch` mode is available, it proposes a fix for each problem identified.

Example

Running:

```
make coccicheck MODE=patch COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@ depends on !context && patch && !org && !report @
expression x;
@@

- ERR_PTR(PTR_ERR(x))
+ ERR_CAST(x)
</smpl>
```

This SmPL excerpt generates patch hunks on the standard output, as illustrated below:

```
diff -u -p a/crypto/ctr.c b/crypto/ctr.c
--- a/crypto/ctr.c 2010-05-26 10:49:38.000000000 +0200
+++ b/crypto/ctr.c 2010-06-03 23:44:49.000000000 +0200
@@ -185,7 +185,7 @@ static struct crypto_instance *crypto_ct
    alg = crypto_attr_alg(tb[1], CRYPTO_ALG_TYPE_CIPHER,
                              CRYPTO_ALG_TYPE_MASK);
    if (IS_ERR(alg))
-           return ERR_PTR(PTR_ERR(alg));
+           return ERR_CAST(alg);

    /* Block size must be >= 4 bytes. */
    err = -EINVAL;
```

Detailed description of the `context` mode

`context` highlights lines of interest and their context in a diff-like style.

> **NOTE**: The diff-like output generated is NOT an applicable patch. The intent of the `context` mode is to highlight the important lines (annotated with minus, `-`) and gives some surrounding context lines around. This output can be used with the diff mode of Emacs to review the code.

Example

Running:

```
make coccicheck MODE=context COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@ depends on context && !patch && !org && !report@
expression x;
@@

* ERR_PTR(PTR_ERR(x))
</smpl>
```

This SmPL excerpt generates diff hunks on the standard output, as illustrated below:

```
diff -u -p /home/user/linux/crypto/ctr.c /tmp/nothing
--- /home/user/linux/crypto/ctr.c   2010-05-26 10:49:38.000000000 +0200
+++ /tmp/nothing
@@ -185,7 +185,6 @@ static struct crypto_instance *crypto_ct
    alg = crypto_attr_alg(tb[1], CRYPTO_ALG_TYPE_CIPHER,
                              CRYPTO_ALG_TYPE_MASK);
    if (IS_ERR(alg))
-           return ERR_PTR(PTR_ERR(alg));

    /* Block size must be >= 4 bytes. */
    err = -EINVAL;
```

Detailed description of the `org` mode

`org` generates a report in the Org mode format of Emacs.

Example

Running:

```
make coccicheck MODE=org COCCI=scripts/coccinelle/api/err_cast.cocci
```

will execute the following part of the SmPL script:

```
<smpl>
@r depends on !context && !patch && (org || report)@
expression x;
position p;
@@

  ERR_PTR@p(PTR_ERR(x))

@script:python depends on org@
p << r.p;
x << r.x;
@@

msg="ERR_CAST can be used with %s" % (x)
msg_safe=msg.replace("[","@(").replace("]",")")
coccilib.org.print_todo(p[0], msg_safe)
</smpl>
```

This SmPL excerpt generates Org entries on the standard output, as illustrated below:

```
* TODO [[view:/home/user/linux/crypto/ctr.c::face=ovl-face1::linb=188::colb=9::cole=16][ERR_CAST can be used with
* TODO [[view:/home/user/linux/crypto/authenc.c::face=ovl-face1::linb=619::colb=9::cole=16][ERR_CAST can be used
* TODO [[view:/home/user/linux/crypto/xts.c::face=ovl-face1::linb=227::colb=9::cole=16][ERR_CAST can be used with
```