Klever Documentation

ISP RAS

Mar 31, 2021

Contents

Klever is a software verification framework that aims at automated checking of programs developed in the GNU C programming language against a variety of requirements using software model checkers. You can learn more about Klever at the project site.

CHAPTER 1

Contents

1.1 Deployment

Klever does not support standard deployment means because it consists of several components that may require complicating setup, e.g. configuring and running a web service with a database access, running system services that perform some preliminary actions with superuser rights, etc. Also, Klever will likely always require several specific addons that can not be deployed in a normal way. Please, be ready to spend quite much time if you follow this instruction first time.

1.1.1 Hardware Requirements

We recommend following hardware to run Klever:

- x86-64 CPU with 4 cores
- 16 GB of memory
- 100 GB of free disk space

We do not guarantee that Klever will operate well if you will use less powerful machines. Increasing specified hardware characteristics in 2-4 times can reduce total verification time very considerably. To generate *Klever Build Bases* for large programs, such as the Linux kernel, you need 3-5 times more free disk space.

1.1.2 Software Requirements

Klever deployment is designed to work on Debian 9, Ubuntu 18.04 and Fedora 32. You can try it for other versions of these distributions, as well as for their derivatives on your own risk.

To deploy Klever one has to clone its Git repository (a path to a directory where it is cloned is referred to as *$KLEVER_SRC*):

```
git clone --recursive https://forge.ispras.ru/git/klever.git
```

Note: Alternatively one can use https://github.com/ldv-klever/klever.git.

Then you need to install all required dependencies.

First of all it is necessary to install packages listed at the following files:

- Debian - `klever/deploys/conf/debian-packages.txt` from *$KLEVER_SRC*.
- Fedora - `klever/deploys/conf/fedora-packages.txt` from *$KLEVER_SRC*.

Then you need to install Python 3.7 or higher and a corresponding development package. If your distribution does not have them you can get them from:

- Debian - here.
- Fedora - here.

To install required Python packages we recommend to create a virtual environment using installed Python. For instance, you can run following commands within *$KLEVER_SRC*:

```
$ /usr/local/python3-klever/bin/python3 -m venv venv
$ source venv/bin/activate
```

To avoid some unpleasant issues during installation we recommend to upgrade PIP and associated packages:

```
$ pip install --upgrade pip wheel setuptools
```

Note: Later we assume that you are using the Klever Python virtual environment created in the way described above.

Then you need to install Python packages including the Klever one:

- For production use it is necessary to run the following command within *$KLEVER_SRC*:

  ```
  $ pip install -r requirements.txt .
  ```

  Later to upgrade the Klever Python package you should run:

  ```
  $ pip install --upgrade -r requirements.txt .
  ```

- If one is going to develop Klever one should install Klever Python package in the *editable* mode (with flag *-e*). To do it, run the following command within *$KLEVER_SRC*:

  ```
  $ pip install -r requirements.txt -e .
  ```

  In this case the Klever Python package will be updated automatically, but you may still need to upgrade its dependencies by running the following command:

  ```
  $ pip install --upgrade -r requirements.txt -e .
  ```

Note: Removing *-r requirements.txt* from the command will install latest versions of required packages. However, it is not guaranteed that they will work well with Klever.

Then one has to get *Klever Addons* and *Klever Build Bases*. Both of them should be described appropriately within *Deployment Configuration File*.

Note: You can omit getting *Klever Addons* if you will use default *Deployment Configuration File* since it contains URLs for all required *Klever Addons*.

1.1.3 Klever Addons

You can provide *Klever Addons* in various forms:

- Local files, directories, archives or Git repositories.
- Remote files, archives or Git repositories.

Deployment scripts will take care of their appropriate extracting. If *Klever Addons* are provided locally the best place for them is directory `addons` within *$KLEVER_SRC* (see *Structure of Klever Git Repository*).

Note: Git does not track `addons` from *$KLEVER_SRC*.

Klever Addons include the following:

- *CIF*.
- *Frama-C (CIL)*.
- *Consul*.
- One or more *Verification Backends*.
- *Optional Addons*.

CIF

One can download CIF binaries from here. These binaries are compatible with various Linux distributions since CIF is based on GCC that has few dependencies. Besides, one can clone CIF Git repository and build CIF from source using corresponding instructions.

Frama-C (CIL)

You can get Frama-C (CIL) binaries from here. As well, you can build it from this source (branch `18.0`) which has several specific patches relatively to the mainline.

Consul

One can download appropriate Consul binaries from here. We are successfully using version 0.9.2 but newer versions can be fine as well. It is possible to build Consul from source.

Verification Backends

You need at least one tool that will perform actual verification of your software. These tools are referred to as *Verification Backends*. As verification backends Klever supports CPAchecker well. Some other verification backends are supported experimentally and currently we do not recommend to use them. You can download binaries of CPAchecker from here. In addition, you can clone CPAchecker Git or Subversion repository and build other versions of CPAchecker from source referring corresponding instructions.

Optional Addons

If you are going to solve verification tasks using VerifierCloud, you should get an appropriate client. Most likely one can use the client from the *CPAchecker verification backend*.

Note: For using VerifierCloud you need appropriate credentials. But anyway it is an optional addon, one is able to use Klever without it.

1.1.4 Klever Build Bases

In addition to *Klever Addons* one should provide *Klever Build Bases* obtained for software to be verified. *Klever Build Bases* should be obtained using Clade. All *Klever Build Bases* should be provided as directories, archives or links to remote archives. The best place for *Klever Build Bases* is the directory `build bases` within *$KLEVER_SRC* (see *Structure of Klever Git Repository*).

Note: Git does not track `build bases` from *$KLEVER_SRC*.

Note: Content of *Klever Build Bases* is not modified during verification.

1.1.5 Deployment Configuration File

After getting *Klever Addons* and *Klever Build Bases* one needs to describe them within *Deployment Configuration File*. By default deployment scripts use `klever/deploys/conf/klever.json` from *$KLEVER_SRC*. We recommend to copy this file somewhere and adjust it appropriately.

There are 2 pairs within *Deployment Configuration File* with names *Klever Addons* and *Klever Build Bases*. The first one is a JSON object where each pair represents a name of a particular *Klever addon* and its description as a JSON object. There is the only exception. Within *Klever Addons* there is *Verification Backends* that serves for describing *Verification Backends*.

Each JSON object that describes a *Klever addon* should always have values for *version* and *path*:

- *Version* gives a very important knowledge for deployment scripts. Depending on values of this pair they behave appropriately. When entities are represented as files, directories or archives deployment scripts remember versions of installed/updated entities. So, later they update these entities just when their versions change. For Git repositories versions can be anything suitable for a Git checkout, e.g. appropriate Git branches, tags or commits. In this case deployment scripts checkout specified versions first. Also, they clone or clean up Git repositories before checkouting, so, all uncommited changes will be ignored. To bypass Git checkouting and clean up you can specify version *CURRENT*. In this case Git repositories are treated like directories.
- *Path* sets either a path relative to *$KLEVER_SRC* or an absolute path to entity (binaries, source files, configurations, etc.) or an entity URL.

For some *Klever Addons* it could be necessary to additionally specify *executable path* or/and *python path* within *path* if binaries or Python packages are not available directly from *path*. For *Verification Backends* there is also *name* with value *CPAchecker*. Keep this pair for all specified *Verification Backends*.

Besides, you can set *copy .git directory* and *allow use local Git repository* to *True*. In the former case deployment scripts will copy directory `.git` if one provides *Klever Addons* as Git repositories. In the latter case deployment scripts will use specified Git repositories for cleaning up and checkouting required versions straightforwardly without cloning them to temporary directories.

Warning: Setting *allow use local Git repository* to *True* will result in removing all your uncommited changes! Besides, ignore rules from, say, `.gitignore` will be ignored and corresponding files and directories will be removed!

Klever Build Bases is a JSON object where each pair represents a name of a particular *Build Base* and its description as a JSON object. Each such JSON object should always have some value for *path*: it should be either an absolute path to the directory that directly contains *Build Base*, or an absolute path to the archive with a *Build Base*, or a link to the remote archive with a *Build Base*. Particular structure of directories inside such archive doesn't matter: it is only required that there should be a single valid *Build Base* somewhere inside. In `job.json` you should specify the name of the *Build Base*.

Note: You can prepare multiple *deployment configuration files*, but be careful when using them to avoid unexpected results due to tricky intermixes.

Note: Actually there may be more *Klever Addons* or *Klever Build Bases* within corresponding locations. Deployment scripts will consider just described ones.

1.1.6 Structure of Klever Git Repository

After getting *Klever Addons* and *Klever Build Bases* the Klever Git repository can look as follows:

```
$KLEVER_SRC
├── addons
│   ├── cif-1517e57.tar.xz
│   ├── consul
│   ├── CPAchecker-1.6.1-svn ea117e2ecf-unix.tar.gz
│   ├── CPAchecker-35003.tar.xz
│   ├── toplevel.opt.tar.xz
│   └── ...
├── build bases
│   ├── linux-3.14.79.tar.xz
│   └── linux-4.2.6
│       ├── allmodconfig
│       └── defconfig
└── ...
```

1.1.7 Deployment Variants

There are several variants for deploying Klever:

Local Deployment

Warning: Do not deploy Klever at your workstation or valuable servers unless you are ready to lose some sensitive data or to have misbehaved software.

Warning: Currently deployment on Fedora makes the *httpd_t* SELinux domain permissive, which may negatively impact the security of your system.

To accomplish local deployment of Klever you need to choose an appropriate mode (one should select *development* only for development purposes, otherwise, please, choose *production*) and to run the following command within *$KLEVER_SRC*:

```
$ sudo venv/bin/klever-deploy-local --deployment-directory $KLEVER_DEPLOY_DIR␣
↪install production
```

Note: Absolute path to `klever-deploy-local` is necessary due to environment variables required for the Klever Python virtual environment are not passed to sudo commands most likely.

After successfull installation one is able to *update* Klever multiple times to install new or to update alredy installed *Klever Addons* and *Klever Build Bases*:

```
$ sudo venv/bin/klever-deploy-local --deployment-directory $KLEVER_DEPLOY_DIR␣
↪update production
```

If you need to update Klever Python package itself (e.g. this may be necessary after update of *$KLEVER_SRC*), then you should execute one additional command prior to the above one:

```
$ pip install --upgrade .
```

This additional command, however, should be skipped if Klever Python package was installed in the *editable* mode (with flag -e) unless you need to to upgrade Klever dependencies. In the latter case you should execute the following command prior updating Klever:

```
$ pip install --upgrade -e .
```

To *uninstall* Klever you need to run:

```
$ sudo venv/bin/klever-deploy-local --deployment-directory $KLEVER_DEPLOY_DIR␣
↪uninstall production
```

A normal sequence of actions for *Local Deployment* is the following: *install* → *update* → *update* → . . . → *update* → *uninstall*. In addition, there are several optional command-line arguments which you can find out by running:

```
$ klever-deploy-local --help
```

We strongly recommend to configure your file indexing service if you have it enabled so that it will ignore content of *$KLEVER_DEPLOY_DIR*. Otherwise, it can consume too much computational resources since Klever manipulates files very extensively during its operation. To do this, please, refer to an appropriate user documentation.

Troubleshooting

If something went wrong during installation, you need to uninstall Klever completely prior to following attempts to install it. In case of ambiguos issues in the development mode you should try to remove the virtual environment and to create it from scratch.

OpenStack Deployment

Note: Althouth we would like to support different OpenStack environments, we tested *OpenStack Deployment* just for the ISP RAS one.

Additional Software Requirements

To install additional packages required only by OpenStack deployment scripts you need to execute the following command:

```
$ pip install -r requirements-openstack.txt ".[openstack]"
```

Note: If in the previous step you installed Klever package with the *-e* argument, then you should use it here as well (i.e. execute *pip install -e ".[openstack]"*).

Supported Options

OpenStack Deployment supports 2 kinds of entities:

- *Klever Base Image* - with default settings this is a Debian 9 OpenStack image with installed Klever dependencies. Using *Klever Base Image* allows to substantially reduce a time for deploying other *Klever Instance*.
- *Klever Instance* - an OpenStack instance, either for development or production purposes. For development mode many debug options are activated by default.

Almost all deployment commands require you to specify path to the private SSH key and your OpenStack username:

```
$ klever-deploy-openstack --os-username $OS_USERNAME --ssh-rsa-private-key-
↪file $SSH_RSA_PRIVATE_KEY_FILE create instance
```

For brevity they are omitted from the following examples.

Also, in addition to command-line arguments mentioned above and below, there are several optional command-line arguments which you can find out by running:

```
$ klever-deploy-openstack --help
```

Klever Base Image

For *Klever Base Image* you can execute actions *show*, *create* and *remove*. The common workflow for *Klever Base Image* is *create* → *remove*, e.g.:

```
$ klever-deploy-openstack create image
```

Unless specified, name *Klever Base vN* (where N is 1 plus a maximum of 0 and vi) is used for new *Klever Base Image*. Besides, deployment scripts overwrites file `klever/deploys/conf/openstack-base-image.txt` with this name so that new instances will be based on the new *Klever Base Image*. To force other users to switch to the new *Klever Base Image* you need to commit changes of this file to the repository.

Klever Instance

For *Klever Instance* you can execute actions *show*, *create*, *update*, *ssh*, *remove*, *share* and *hide*. Basically you should perform actions with *Klever Instance* in the following order: *create* → *update* → *update* → … → *update* → *remove* exactly as for *Local Deployment*, e.g.:

```
$ klever-deploy-openstack create instance
```

By default Klever is deployed in production mode, but you can change this with the *–mode* command-line argument:

```
$ klever-deploy-openstack --mode development create instance
```

In addition, between creating and removing you can also *share/hide* for/from the outside world *Klever Instance* and open an SSH connection to it. By default name for *Klever Instance* is a concatenation of *$OS_USERNAME*, “klever”, and the mode used (development or production), e.g. *petrov-klever-development*.

Multiple Klever Instances

You can also create a specified number of OpenStack instances for performing various experiments by using the *–instances* command-line argument. In this mode you can only execute actions *show*, *create*, *update* and *remove*. The normal workflow for *Multiple Klever Instances* is the same as for *Klever Instance*, e.g.:

```
$ klever-deploy-openstack --instances $INSTANCES create instance
```

1.2 Tutorial

This tutorial describes a basic workflow of using Klever. We assume that you deploy Klever *locally* on Debian 9 in the production mode with default settings from the latest master. In addition, we assume that your username is **debian** and your home directory is **/home/debian**[1].

1.2.1 Preparing Build Bases

After a successful deployment of Klever you need to prepare a *build base* on the same machine where you deployed Klever. This tutorial treats just build bases for Linux kernel loadable modules since the publicly available version of Klever supports verification of other software in the experimental stage. You should not expect that Klever supports all versions and configurations of the Linux kernel well. There is a big list of things to do in this direction.

Below we consider as an example preparation of a build base for verification of Linux 3.14.79 modules (architecture *x86_64*, configuration *allmodconfig*, GCC 4.8.5). You can try to execute similar steps for other versions and configurations of the Linux kernel at your own risks. To build new versions of the Linux kernel you may need newer versions of GCC.

You can download the archive of the target build base prepared in advance from here. Let’s assume that you decompress this archive into directory **/home/debian/build-base-linux-3.14.79-x86_64-allmodconfig** so that there should be file *meta.json* directly at the top level in that directory.

To prepare the target build base from scratch you can follow the next steps:

[1] If this is not the case, you should adjust paths to build bases below respectively.

```
$ wget https://cdn.kernel.org/pub/linux/kernel/v3.x/linux-3.14.79.tar.xz
$ tar -xvf linux-3.14.79.tar.xz
$ cd linux-3.14.79/
$ make allmodconfig
$ clade -w ~/build-base-linux-3.14.79-x86_64-allmodconfig -p klever_linux_kernel make␣
↪-j8 modules
```

Then you will need to wait for quite a long period of time depending on the performance of your machine.

1.2.2 Signing in

Before performing all other actions described further in this tutorial you need to sign in to a Klever web interface:

1. Open page http://localhost:8998 in your web-browser[2].
2. Input **manager** as a username and a password and sign in (Fig. 1.1).

Then you will be automatically redirected to a *job tree* page presented in the following sections.

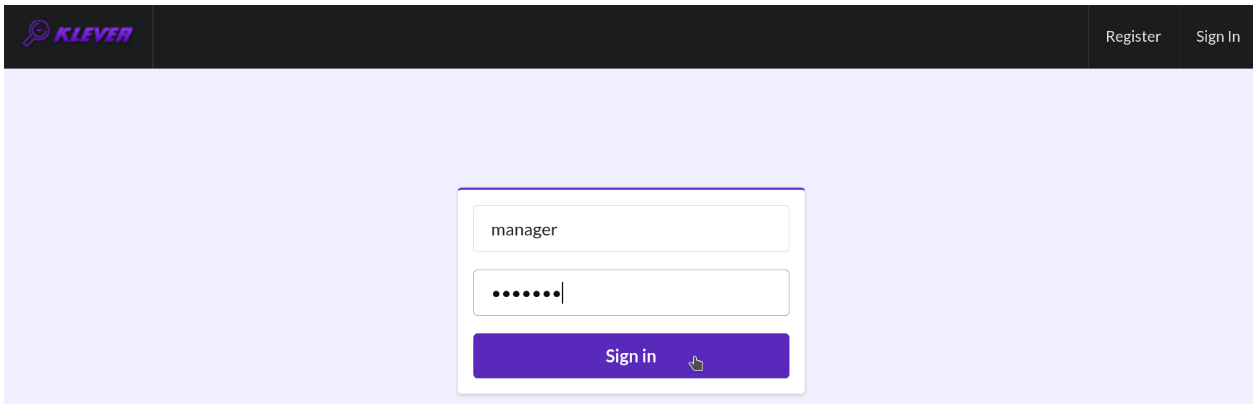


Fig. 1.1: Signing in

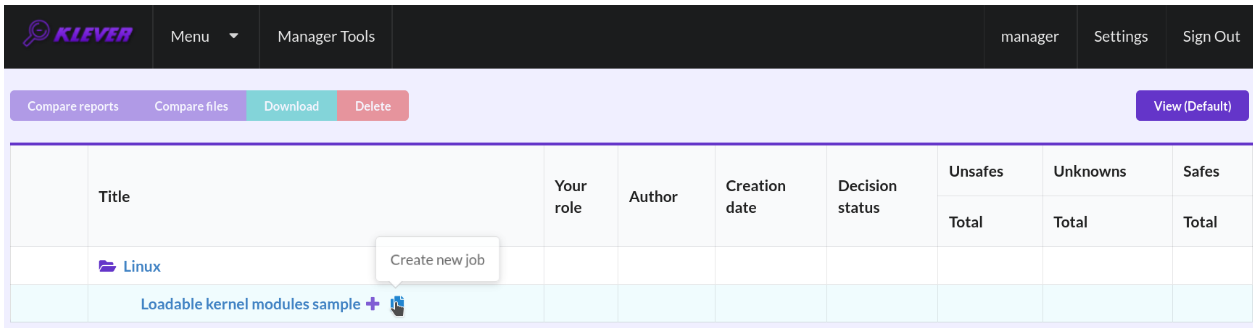
1.2.3 Starting Verification

As an example we consider checking usage of clocks in USB drivers. To start up verification you need to do as follows:

1. Start the creation of a new *job* (Fig. 1.2).
2. Specify an appropriate title and create the new job (Fig. 1.3).
3. To configure a first *job version* you need to specify (Fig. 1.4):
   - The path to the prepared build base that is **/home/debian/build-base-linux-3.14.79-x86_64-allmodconfig**.
   - Targets, e.g. USB drivers, i.e. all modules from directory **drivers/usb** in our example.
   - Requirement specifications to be checked, e.g. **drivers:clk1** and **drivers:clk2** in our example (you can see a complete list of supported requirement specifications at the end of this section).
4. Press *Ctrl-S* when being at the editor window to save changes.
5. Start a *decision of the job version* (Fig. 1.4).

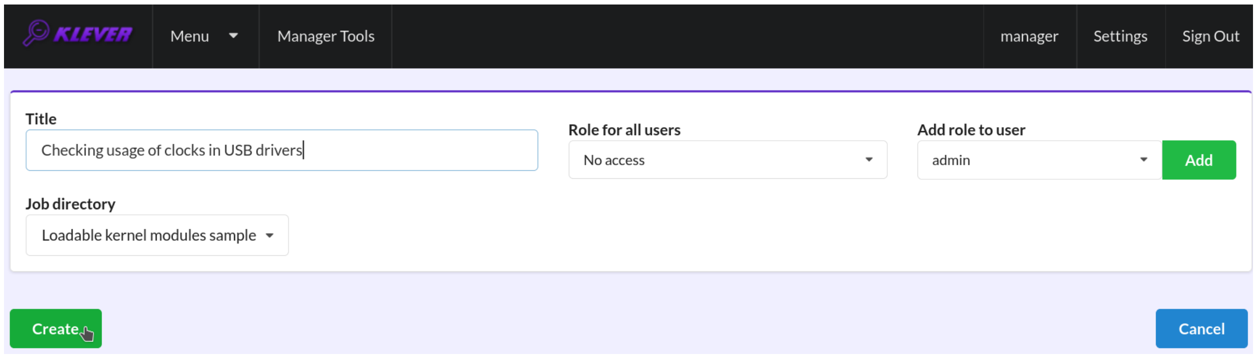
[2] You can open the Klever web interface from other machines as well, but you need to set up appropriate access for that.

After that Klever automatically redirects you to a *job version/decision page* that is described in detail in the following sections.



	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							

Fig. 1.2: Starting the creation of a new job



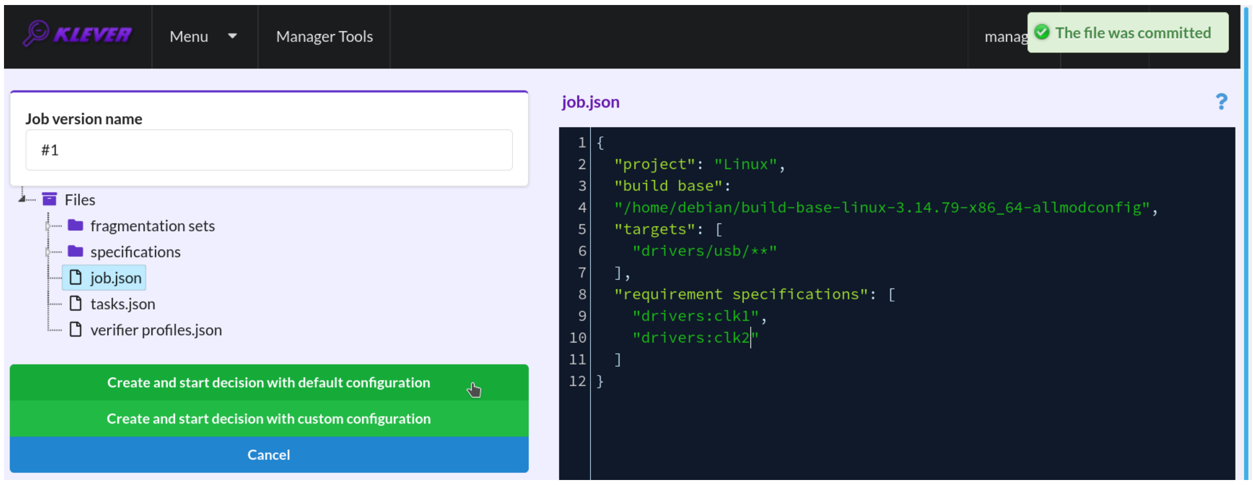
Title: Checking usage of clocks in USB drivers
Role for all users: No access
Add role to user: admin
Job directory: Loadable kernel modules sample

Fig. 1.3: The creation of the new job

Later you can create new jobs by opening the job tree page, e.g. through clicking on the Klever logo (Fig. 1.5), and by executing steps above. You can create new jobs even when some job version is being decided, but job versions are decided one by one by default.

Below there are requirement specifications that you can choose for verification of Linux loadable kernel modules (we do not recommend to check requirement specifications which identifiers are italicised since they produce either many false alarms or there are just a few violations of these requirements at all):

1. alloc:irq
2. alloc:spinlock
3. alloc:usb lock
4. arch:asm:dma-mapping
5. arch:mm:ioremap
6. *block:blk-core:queue*
7. *block:blk-core:request*
8. *block:genhd*
9. *concurrency safety*
10. drivers:base:class
11. drivers:usb:core:usb:coherent



```
{
  "project": "Linux",
  "build base":
  "/home/debian/build-base-linux-3.14.79-x86_64-allmodconfig",
  "targets": [
    "drivers/usb/**"
  ],
  "requirement specifications": [
    "drivers:clk1",
    "drivers:clk2"
  ]
}
```

Fig. 1.4: Configuring the first job version and starting its decision

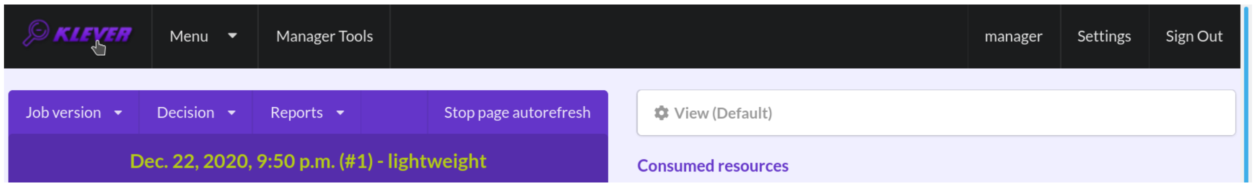


Fig. 1.5: Opening the job tree page

12. drivers:usb:core:usb:dev
13. drivers:usb:core:driver
14. drivers:usb:core:urb
15. drivers:usb:gadget:udc-core
16. drivers:clk1
17. drivers:clk2
18. fs:sysfs:group
19. kernel:locking:mutex
20. kernel:locking:rwlock
21. kernel:locking:spinlock
22. kernel:module
23. *kernel:rcu:update:lock bh*
24. *kernel:rcu:update:lock shed*
25. kernel:rcu:update:lock
26. *kernel:rcu:srcu*
27. *kernel:sched:completion*
28. *lib:find_next_bit*
29. *lib:idr*

30. memory safety
31. net:core:dev
32. *net:core:rtnetlink*
33. *net:core:sock*

In case of verification of the Linux kernel rather than vanilla 3.14.79, you may need to specify one extra parameter **specifications set**, when configuring the job version (Fig. 1.4), with a value from the following list:

1. 2.6.33
2. 4.6.7
3. 4.15
4. 4.17
5. 5.5

These specification sets correspond to vanilla versions of the Linux kernel. You should select such a specifications set that matches your custom version of the Linux kernel better through trial and error.

1.2.4 Decision Progress

At the beginning of the decision of the job version Klever indexes each new build base. This can take rather much time before it starts to generate and to decide first *tasks*[3] for large build bases. In about 15 minutes you can refresh the page and see some tasks and their decisions there. Please, note that the automatic refresh of the job version/decision page stops after 5 minutes, so you either need to refresh it through web browser means or request Klever to switch it on back (Fig. 1.6).

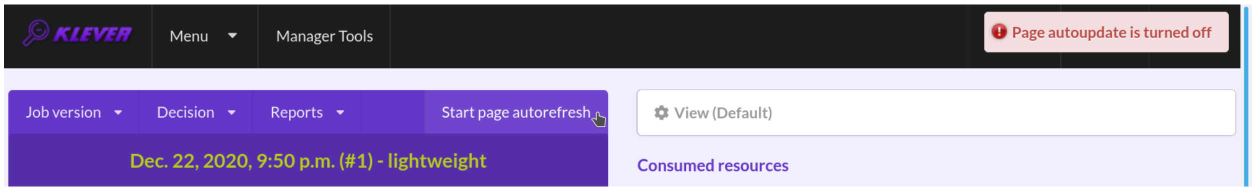


Fig. 1.6: Switching on the automatic refresh of the job version/decision page

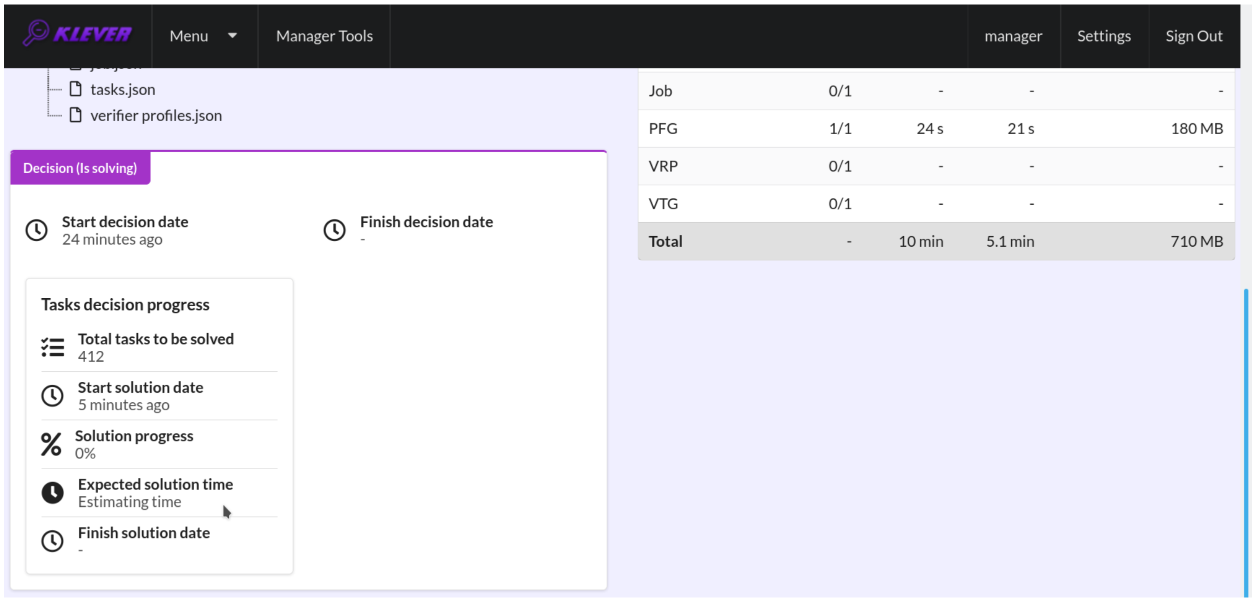
Before the job version is eventually decided Klever estimates and provides a *decision progress* (Fig. 1.7 and Fig. 1.8). You should keep in mind that Klever collects statistics for 10% of tasks before it starts predicting an approximate remaining time for their decision. After that, it recalculates it on the base of new, accumulated statistics. In our example it takes 1 day and 2 hours to decide the job version completely (Fig. 1.9).

At the job tree page you can see all versions of particular jobs (Fig. 1.10) and their *decision statutes* (Fig. 1.11). Besides, you can open the page with details of the decision of the latest job version (Fig. 1.12) or the page describing the decision of the particular job version (Fig. 1.13).

1.2.5 Analyzing Verification Results

Klever can fail to generate and to decide tasks. In this case it provides users with *unknown* verdicts, otherwise there are *safe* or *unsafe* verdicts (Fig. 1.14). You already saw the example with summaries of these verdicts at the job tree page (Fig. 1.10 and Fig. 1.11). In this tutorial we do not consider other verdicts rather than unsafes that are either

[3] For the considered example each task is a pair of a Linux loadable kernel module and a requirements specification. There are 3355 modules under verification and 2 requirement specifications to be checked, so there are 6710 tasks in total.



tasks.json
verifier profiles.json

Decision (Is solving)

Start decision date
24 minutes ago

Finish decision date
-

Tasks decision progress

Total tasks to be solved
412

Start solution date
5 minutes ago

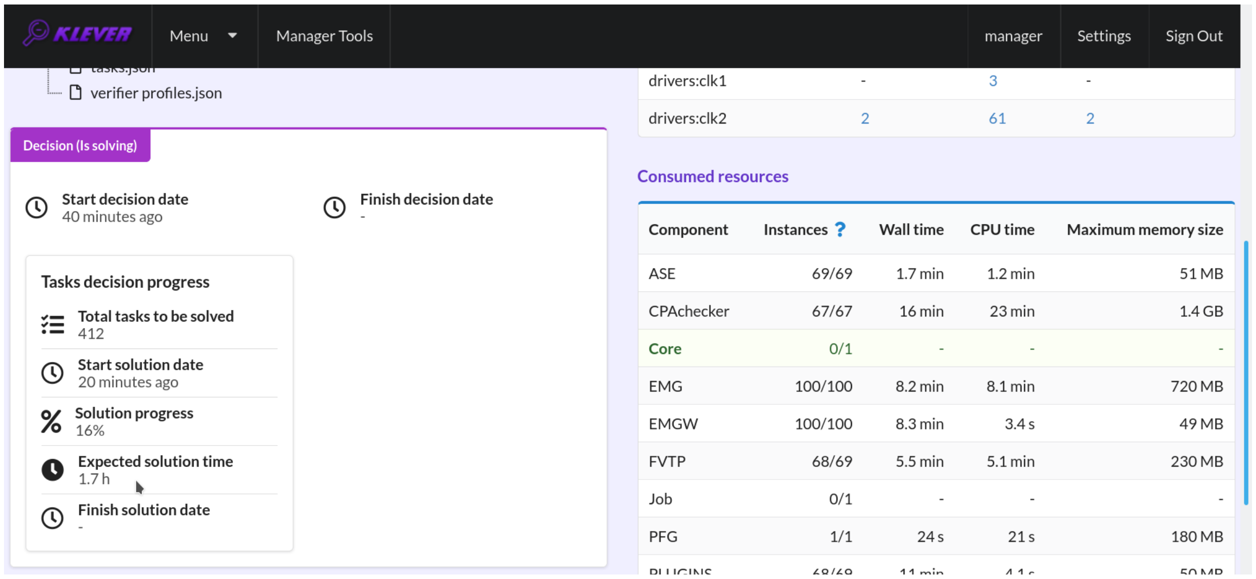
Solution progress
0%

Expected solution time
Estimating time

Finish solution date
-

Job	0/1	-	-	-
PFG	1/1	24 s	21 s	180 MB
VRP	0/1	-	-	-
VTG	0/1	-	-	-
Total	-	10 min	5.1 min	710 MB

Fig. 1.7: The progress of the decision of the job version (estimating a remaining time)



verifier profiles.json

Decision (Is solving)

Start decision date
40 minutes ago

Finish decision date
-

Tasks decision progress

Total tasks to be solved
412

Start solution date
20 minutes ago

Solution progress
16%

Expected solution time
1.7 h

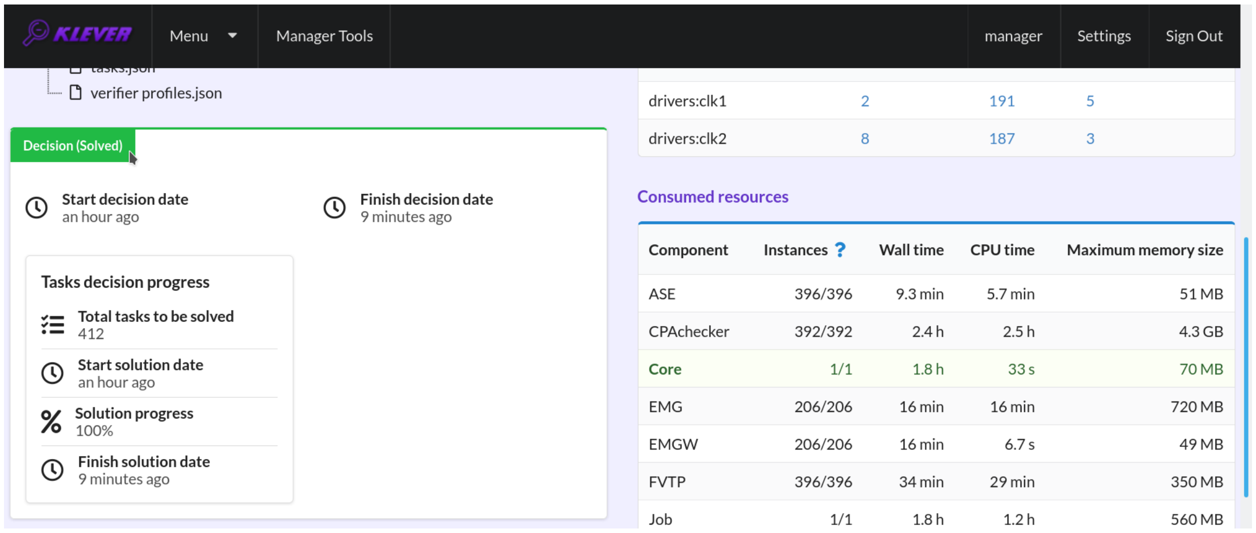
Finish solution date
-

drivers:clk1	-	3	-
drivers:clk2	2	61	2

Consumed resources

Component	Instances ?	Wall time	CPU time	Maximum memory size
ASE	69/69	1.7 min	1.2 min	51 MB
CPAchecker	67/67	16 min	23 min	1.4 GB
Core	0/1	-	-	-
EMG	100/100	8.2 min	8.1 min	720 MB
EMGW	100/100	8.3 min	3.4 s	49 MB
FVTP	68/69	5.5 min	5.1 min	230 MB
Job	0/1	-	-	-
PFG	1/1	24 s	21 s	180 MB
PLUGINS	68/69	11 min	4.1 s	50 MB

Fig. 1.8: The progress of the decision of the job version (the remaining time is estimated)



Start decision date
an hour ago

Finish decision date
9 minutes ago

Tasks decision progress

Total tasks to be solved
412

Start solution date
an hour ago

Solution progress
100%

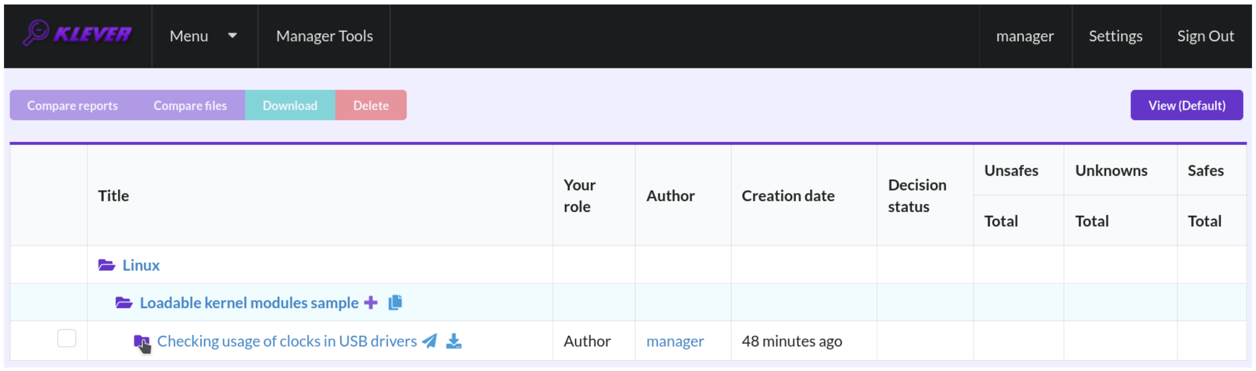
Finish solution date
9 minutes ago

drivers:clk1	2	191	5
drivers:clk2	8	187	3

Consumed resources

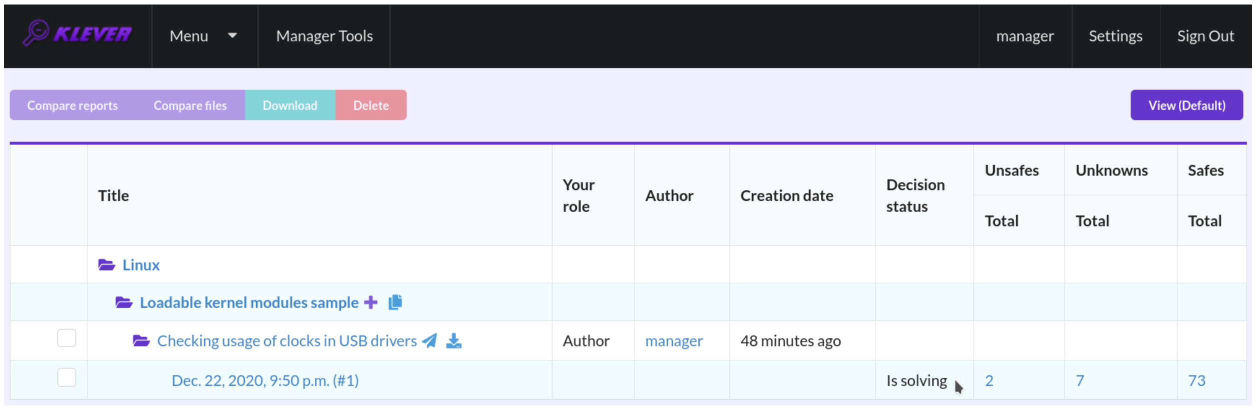
Component	Instances ?	Wall time	CPU time	Maximum memory size
ASE	396/396	9.3 min	5.7 min	51 MB
CPAchecker	392/392	2.4 h	2.5 h	4.3 GB
Core	1/1	1.8 h	33 s	70 MB
EMG	206/206	16 min	16 min	720 MB
EMGW	206/206	16 min	6.7 s	49 MB
FVTP	396/396	34 min	29 min	350 MB
Job	1/1	1.8 h	1.2 h	560 MB

Fig. 1.9: The completed decision of the job version



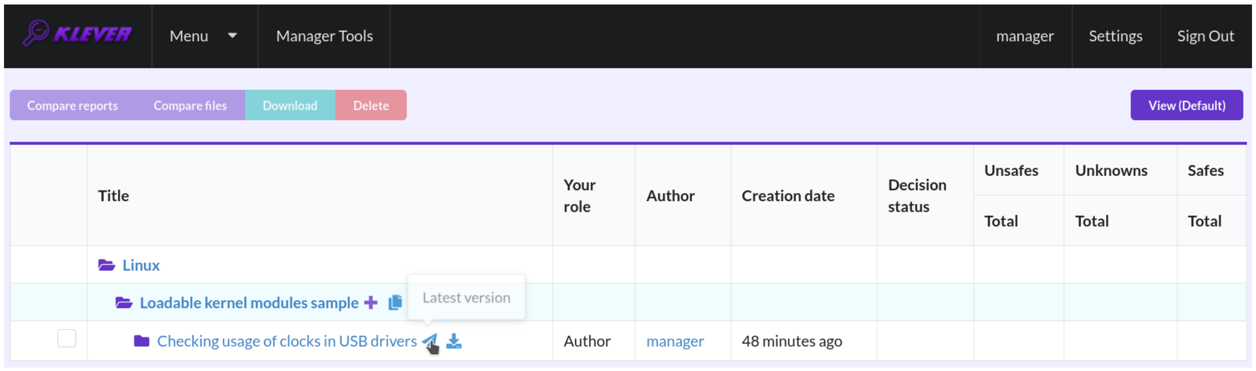
	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							
	Checking usage of clocks in USB drivers	Author	manager	48 minutes ago				

Fig. 1.10: Showing job versions



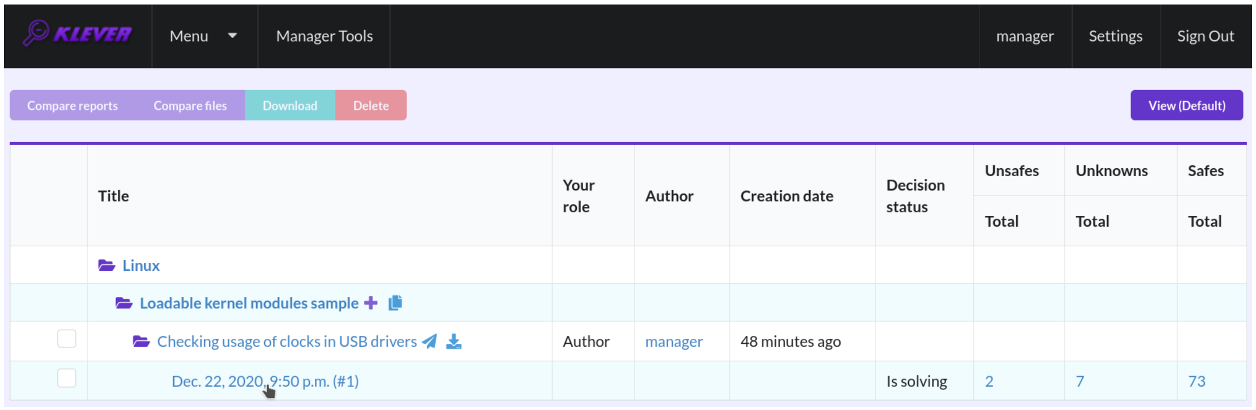
	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							
	Checking usage of clocks in USB drivers	Author	manager	48 minutes ago				
	Dec. 22, 2020, 9:50 p.m. (#1)				Is solving	2	7	73

Fig. 1.11: The status of the decision of the job version



	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							
	Checking usage of clocks in USB drivers	Author	manager	48 minutes ago				

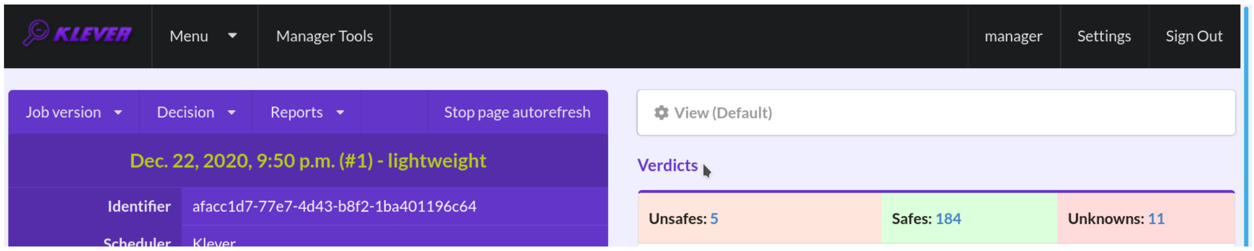
Fig. 1.12: Opening the page with the decision of the latest job version



	Title	Your role	Author	Creation date	Decision status	Unsafes	Unknowns	Safes
						Total	Total	Total
	Linux							
	Loadable kernel modules sample							
	Checking usage of clocks in USB drivers	Author	manager	48 minutes ago				
	Dec. 22, 2020, 9:50 p.m. (#1)				Is solving	2	7	73

Fig. 1.13: Opening the page with the decision of the particular job version

violations of checked requirements or false alarms (Fig. 1.15). Klever reports unsafes if so during the decision of the job version and you can assess them both during the decision and after its completion.

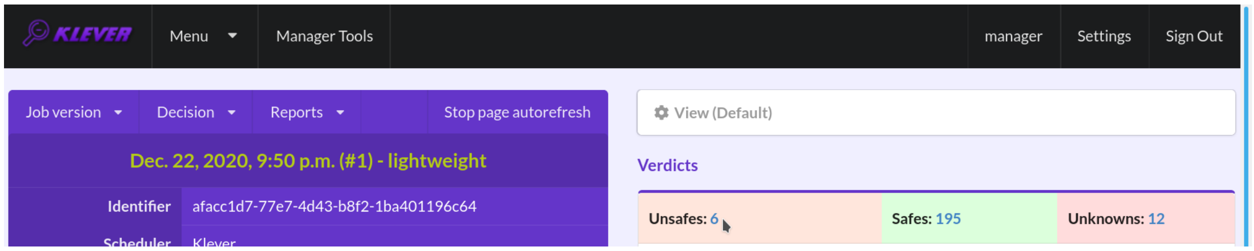


Dec. 22, 2020, 9:50 p.m. (#1) - lightweight

Identifier	afacc1d7-77e7-4d43-b8f2-1ba401196c64
Scheduler	Klever

Unsafes: 5	Safes: 184	Unknowns: 11

Fig. 1.14: Verdicts



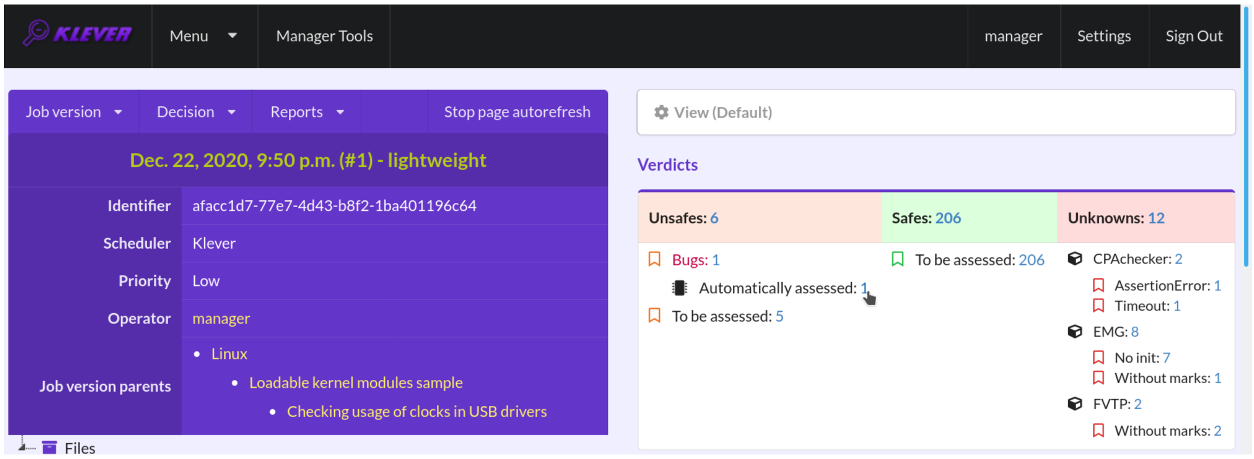
Dec. 22, 2020, 9:50 p.m. (#1) - lightweight

Identifier	afacc1d7-77e7-4d43-b8f2-1ba401196c64
Scheduler	Klever

Unsafes: 6	Safes: 195	Unknowns: 12

Fig. 1.15: The total number of unsafes reported thus far

During assessment of unsafes experts can create marks that can match other unsafes with similar error traces (we consider marks and error traces in detail within the next section). For instance, there is a preset mark for a sample job that matches one of the reported unsafes (Fig. 1.16). Automatic assessment can reduce efforts for analysis of verification results considerably, e.g. when verifying several versions or configurations of the same software. But experts should analyze such automatically assessed unsafes since the same mark can match unsafes with error traces that look very similar but correspond to different faults. Unsafes without marks need assessment as well (Fig. 1.17). When checking several requirement specifications in the same job, one is able to analyze unsafes just for a particular requirements specification (Fig. 1.18).



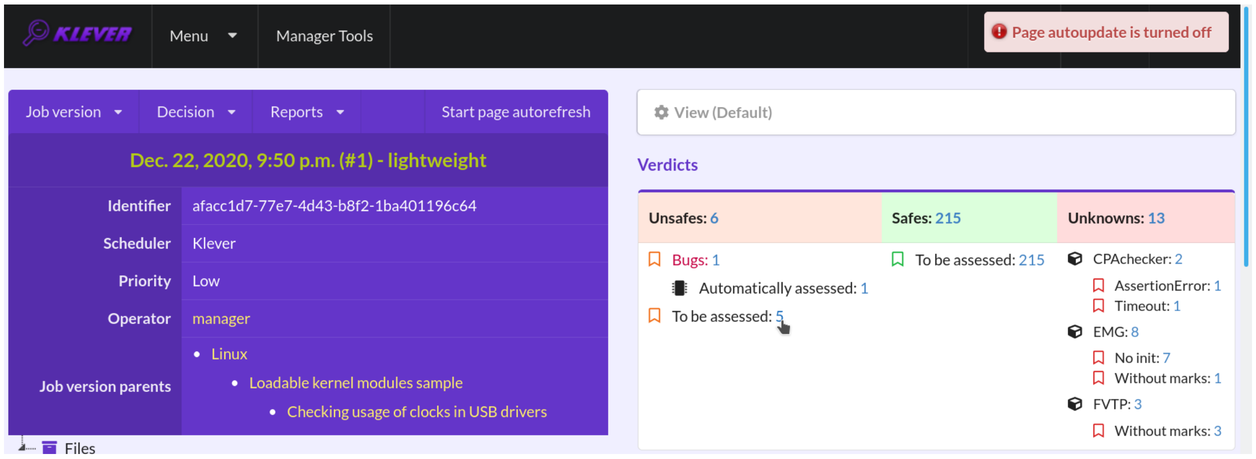
Dec. 22, 2020, 9:50 p.m. (#1) - lightweight

Identifier	afacc1d7-77e7-4d43-b8f2-1ba401196c64
Scheduler	Klever
Priority	Low
Operator	manager
Job version parents	• Linux • Loadable kernel modules sample • Checking usage of clocks in USB drivers

Unsafes: 6	Safes: 206	Unknowns: 12
Bugs: 1 Automatically assessed: 1 To be assessed: 5	To be assessed: 206	CPAchecker: 2 AssertionError: 1 Timeout: 1 EMG: 8 No init: 7 Without marks: 1 FVTP: 2 Without marks: 2

Fig. 1.16: The total number of automatically assessed unsafes

After clicking on the links in Fig. 1.15-Fig. 1.18 you will be redirected to pages with lists of corresponding unsafes (e.g. Fig. 1.19) except for if there is the only element in this list an error trace will be shown immediately. For further analysis we recommend clicking on an unsafe index on the left to open a new page in a separate tab (Fig. 1.20). To



Dec. 22, 2020, 9:50 p.m. (#1) - lightweight

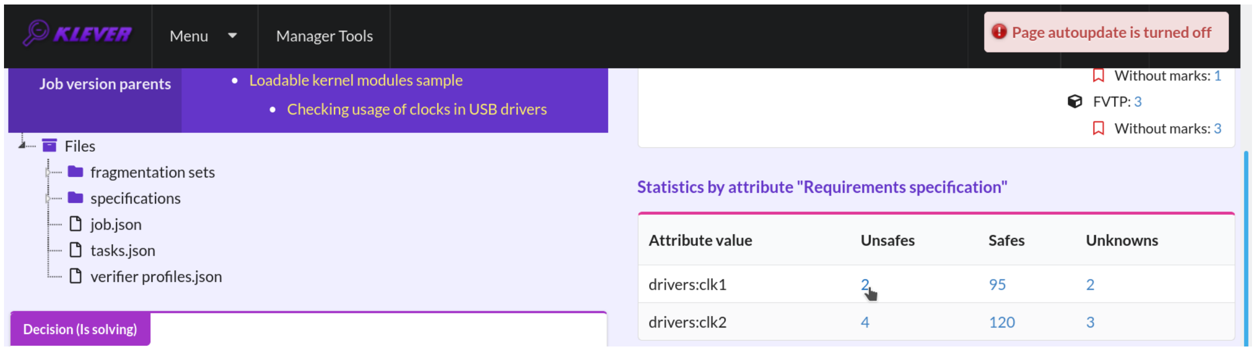
Identifier	afacc1d7-77e7-4d43-b8f2-1ba401196c64
Scheduler	Klever
Priority	Low
Operator	manager
Job version parents	• Linux • Loadable kernel modules sample • Checking usage of clocks in USB drivers

Files

Verdicts

Unsafes: 6	Safes: 215	Unknowns: 13
Bugs: 1 Automatically assessed: 1 To be assessed: 5	To be assessed: 215	CPAchecker: 2 AssertionError: 1 Timeout: 1 EMG: 8 No init: 7 Without marks: 1 FVTP: 3 Without marks: 3

Fig. 1.17: The total number of unsafes without any assessment



Job version parents	• Loadable kernel modules sample • Checking usage of clocks in USB drivers

Without marks: 1
FVTP: 3
Without marks: 3

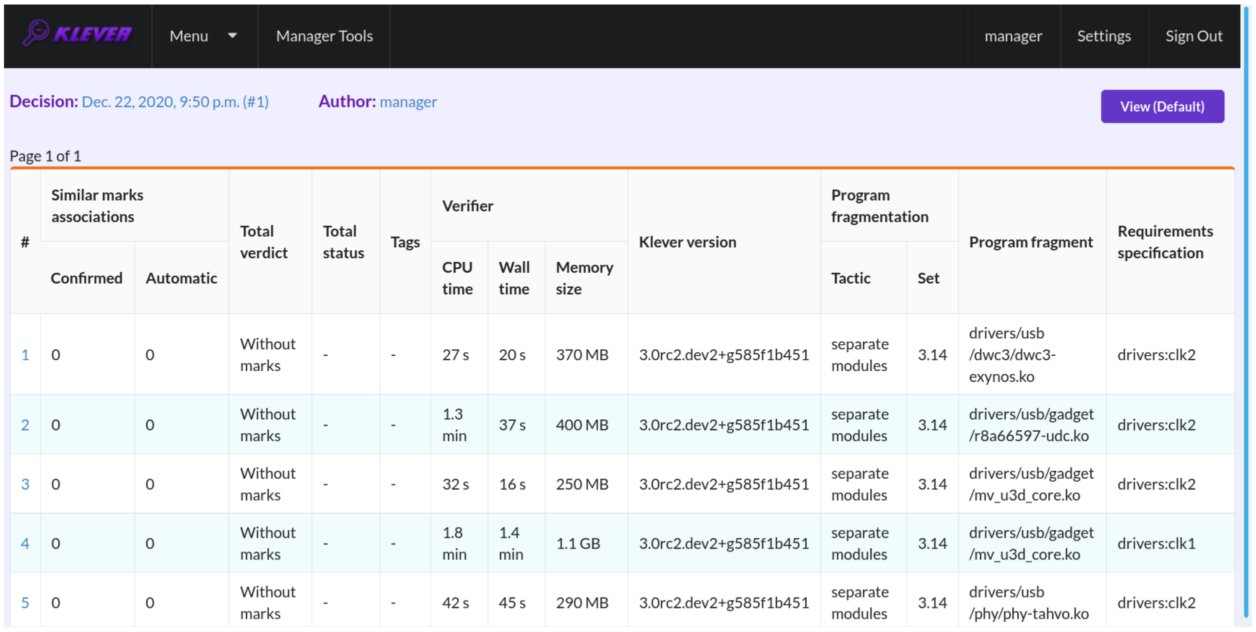
Files
- fragmentation sets
- specifications
- job.json
- tasks.json
- verifier profiles.json

Statistics by attribute "Requirements specification"

Attribute value	Unsafes	Safes	Unknowns
drivers:clk1	2	95	2
drivers:clk2	4	120	3

Fig. 1.18: The total number of unsafes corresponding to the particular requirements specification

return back to the job version/decision page you can click on the title of the job decision on the top left (Fig. 1.21). This can be done at any page with such the link.



Decision: Dec. 22, 2020, 9:50 p.m. (#1) **Author:** manager

Page 1 of 1

#	Similar marks associations		Total verdict	Total status	Tags	Verifier			Klever version	Program fragmentation		Program fragment	Requirements specification
	Confirmed	Automatic				CPU time	Wall time	Memory size		Tactic	Set		
1	0	0	Without marks	-	-	27 s	20 s	370 MB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb /dwc3/dwc3-exynos.ko	drivers:clk2
2	0	0	Without marks	-	-	1.3 min	37 s	400 MB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb/gadget /r8a66597-udc.ko	drivers:clk2
3	0	0	Without marks	-	-	32 s	16 s	250 MB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb/gadget /mv_u3d_core.ko	drivers:clk2
4	0	0	Without marks	-	-	1.8 min	1.4 min	1.1 GB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb/gadget /mv_u3d_core.ko	drivers:clk1
5	0	0	Without marks	-	-	42 s	45 s	290 MB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb /phy/phy-tahvo.ko	drivers:clk2

Fig. 1.19: The list of unsafes without any assessment

1.2.6 Analyzing Error Traces

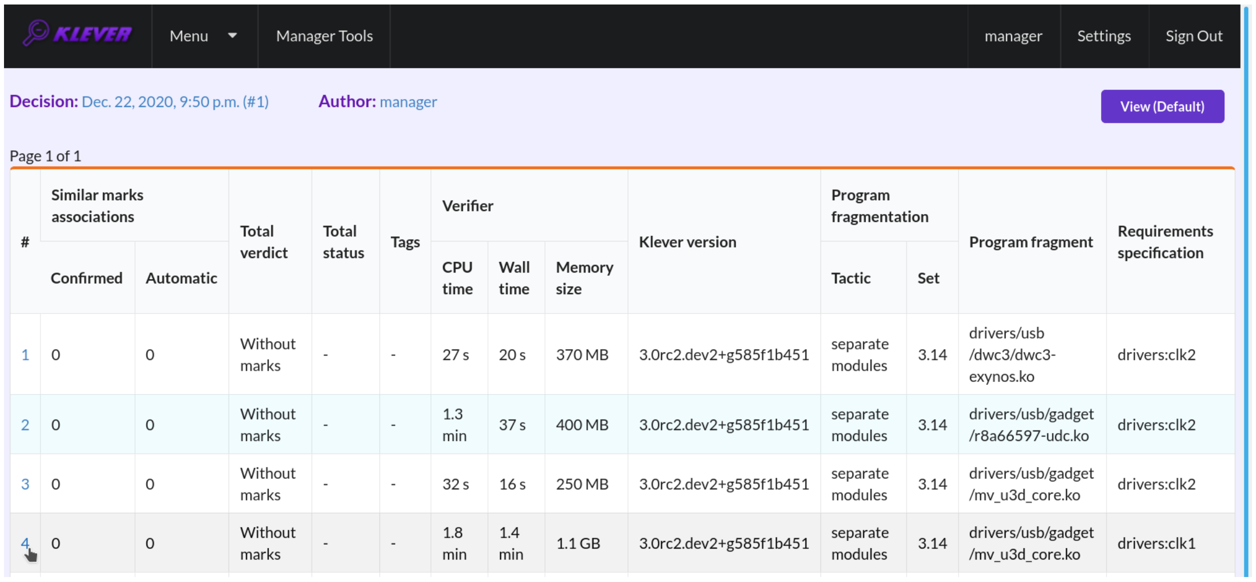
After clicking on links within the list of unsafes like in Fig. 1.20, you will see corresponding error traces. For instance, Fig. 1.22 demonstrates an error trace example for module *drivers/usb/gadget/mv_u3d_core.ko* and requirements specification *drivers:clk1*.

An *error trace* is a sequence of declarations and statements in a source code of a module under verification and an *environment model* generated by Klever. Besides, within that sequence there are *assumptions* specifying conditions that a software model checker considers to be true. Declarations, statements and assumptions represent a path starting from an entry point and ending at a violation of one of checked requirements. The entry point analogue for userspace programs is the function *main* while for Linux loadable kernel modules entry points are generated by Klever as a part of environment models. Requirement violations do not always correspond to places where detected faults should be fixed. For instance, the developer can omit a check for a return value of a function that can fail. As a result various issues, such as leaks or null pointer dereferences, can be revealed somewhere later.

Numbers in the left column correspond to line numbers in source files and models. Source files and models are displayed to the right of error traces. Fig. 1.22 does not contain anything at the right part of the window since there should be the environment model containing the generated *main* function but by default models are not demonstrated for users in the web interface. If you click on a line number corresponding to an original source file, you will see this source file as in Fig. 1.23.

You can click on eyes and on rectangles to show hidden parts of the error trace (Fig. 1.24-Fig. 1.25). Then you can hide them back if they are out of your interest. The difference between eyes and rectangles is that functions with eyes have either notes (Fig. 1.26) or warnings (Fig. 1.27) at some point of their execution, perhaps, within called functions. *Notes* describe important actions in models. *Warnings* represent places where Klever detects violations of checked requirements.

You can see that before calling module initialization and exit functions as well as module callbacks there is additional stuff in the error trace. These are parts of the environment model necessary to initialize models, to invoke module

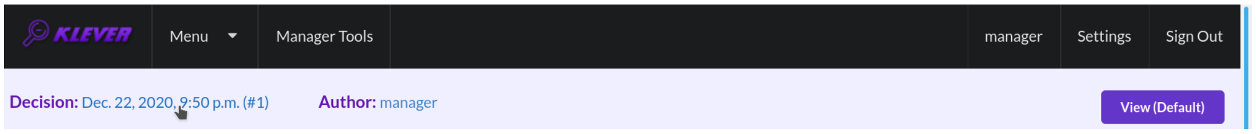


Decision: Dec. 22, 2020, 9:50 p.m. (#1) **Author:** manager

Page 1 of 1

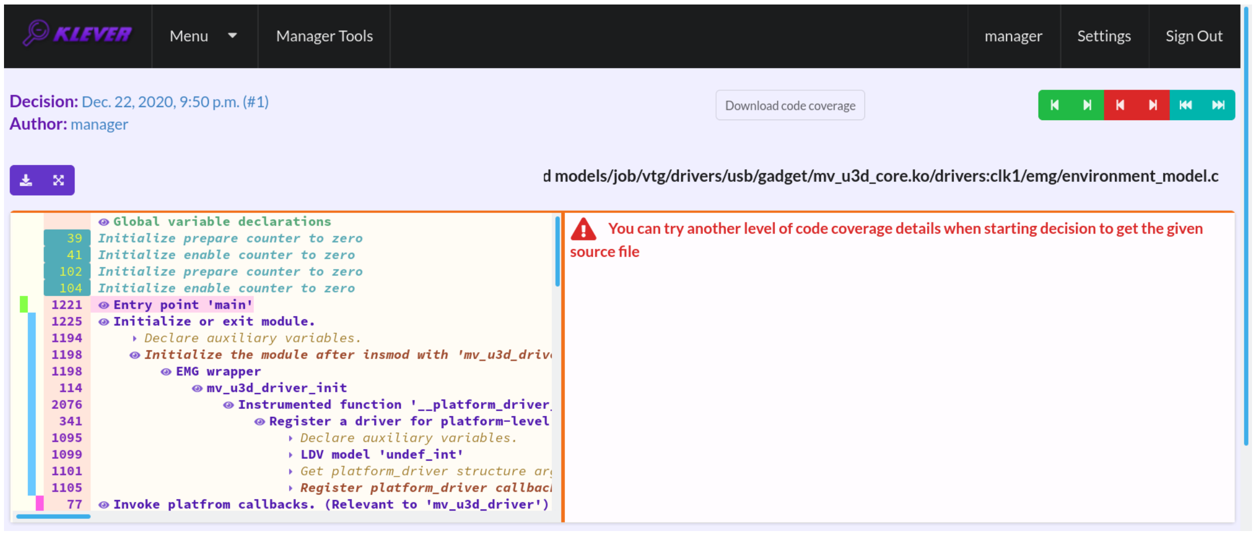
#	Similar marks associations		Total verdict	Total status	Tags	Verifier			Klever version	Program fragmentation		Program fragment	Requirements specification
	Confirmed	Automatic				CPU time	Wall time	Memory size		Tactic	Set		
1	0	0	Without marks	-	-	27 s	20 s	370 MB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb/dwc3/dwc3-exynos.ko	drivers:clk2
2	0	0	Without marks	-	-	1.3 min	37 s	400 MB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb/gadget/r8a66597-udc.ko	drivers:clk2
3	0	0	Without marks	-	-	32 s	16 s	250 MB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb/gadget/mv_u3d_core.ko	drivers:clk2
4	0	0	Without marks	-	-	1.8 min	1.4 min	1.1 GB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb/gadget/mv_u3d_core.ko	drivers:clk1

Fig. 1.20: Opening the error trace corresponding to the unsafe without any assessment



Decision: Dec. 22, 2020, 9:50 p.m. (#1) **Author:** manager

Fig. 1.21: Moving back to the job version/decision page



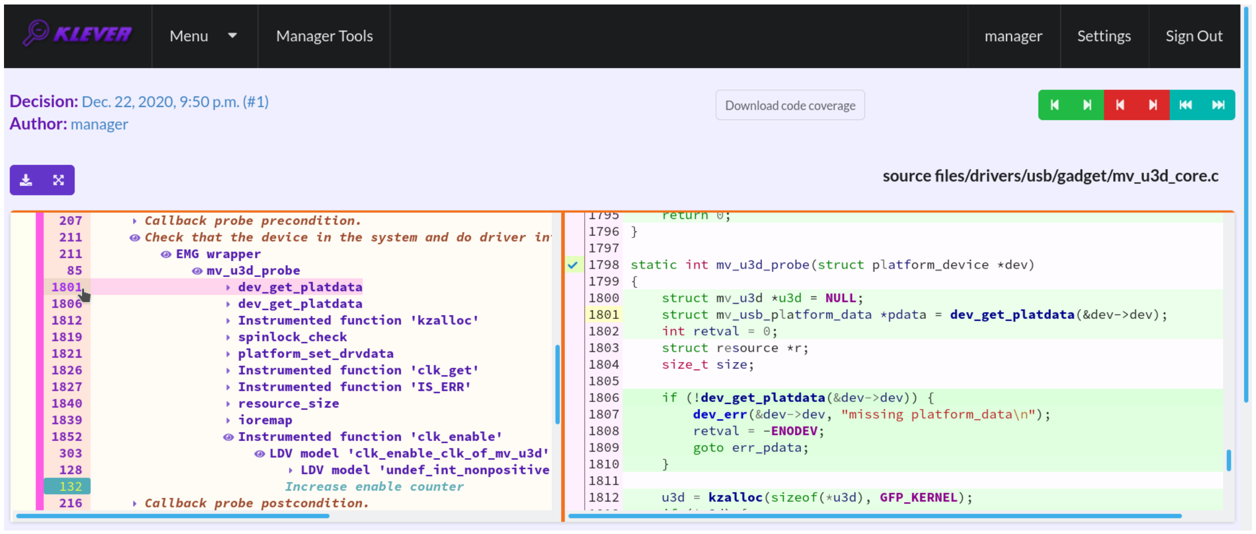
Decision: Dec. 22, 2020, 9:50 p.m. (#1)
Author: manager

d models/job/vtg/drivers/usb/gadget/mv_u3d_core.ko/drivers:clk1/emg/environment_model.c

```
Global variable declarations
Initialize prepare counter to zero
Initialize enable counter to zero
Initialize prepare counter to zero
Initialize enable counter to zero
Entry point 'main'
Initialize or exit module.
    Declare auxiliary variables.
    Initialize the module after insmod with 'mv_u3d_driv
        EMG wrapper
            mv_u3d_driver_init
                Instrumented function '__platform_driver_
                    Register a driver for platform-level
                        Declare auxiliary variables.
                        LDV model 'undef_int'
                        Get platform_driver structure ar
                        Register platform_driver callbac
Invoke platfrom callbacks. (Relevant to 'mv_u3d_driver')
```

You can try another level of code coverage details when starting decision to get the given source file

Fig. 1.22: The error trace for module drivers/usb/gadget/mv_u3d_core.ko and requirements specification drivers:clk1



Decision: Dec. 22, 2020, 9:50 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/mv_u3d_core.c

```
 207     Callback probe precondition.
 211     Check that the device in the system and do driver in
 211         EMG wrapper
  85             mv_u3d_probe
1801                 dev_get_platdata
1806                 dev_get_platdata
1812                 Instrumented function 'kzalloc'
1819                 spinlock_check
1821                 platform_set_drvdata
1826                 Instrumented function 'clk_get'
1827                 Instrumented function 'IS_ERR'
1840                 resource_size
1839                 ioremap
1852                 Instrumented function 'clk_enable'
 303                     LDV model 'clk_enable_clk_of_mv_u3d'
 128                         LDV model 'undef_int_nonpositive
 132                         Increase enable counter
 216     Callback probe postcondition.
```

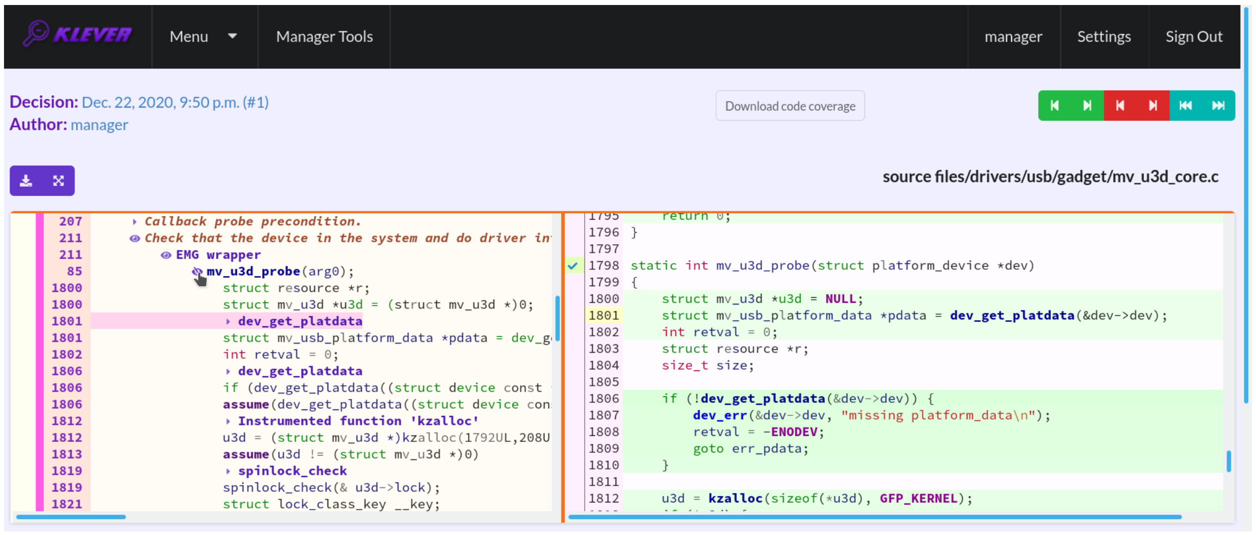
```
        return 0;
}

static int mv_u3d_probe(struct platform_device *dev)
{
        struct mv_u3d *u3d = NULL;
        struct mv_usb_platform_data *pdata = dev_get_platdata(&dev->dev);
        int retval = 0;
        struct resource *r;
        size_t size;

        if (!dev_get_platdata(&dev->dev)) {
                dev_err(&dev->dev, "missing platform_data\n");
                retval = -ENODEV;
                goto err_pdata;
        }

        u3d = kzalloc(sizeof(*u3d), GFP_KERNEL);
```

Fig. 1.23: Showing the line in the original source file corresponding to the error trace statement



Decision: Dec. 22, 2020, 9:50 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/mv_u3d_core.c

```
 207     Callback probe precondition.
 211     Check that the device in the system and do driver in
 211         EMG wrapper
  85             mv_u3d_probe(arg0);
1800                 struct resource *r;
1800                 struct mv_u3d *u3d = (struct mv_u3d *)0;
1801                 dev_get_platdata
1801                 struct mv_usb_platform_data *pdata = dev_g
1802                 int retval = 0;
1806                 dev_get_platdata
1806                 if (dev_get_platdata((struct device const
1806                 assume(dev_get_platdata((struct device con
1812                 Instrumented function 'kzalloc'
1812                 u3d = (struct mv_u3d *)kzalloc(1792UL,208U
1813                 assume(u3d != (struct mv_u3d *)0)
1819                 spinlock_check
1819                 spinlock_check(& u3d->lock);
1821                 struct lock_class_key __key;
```

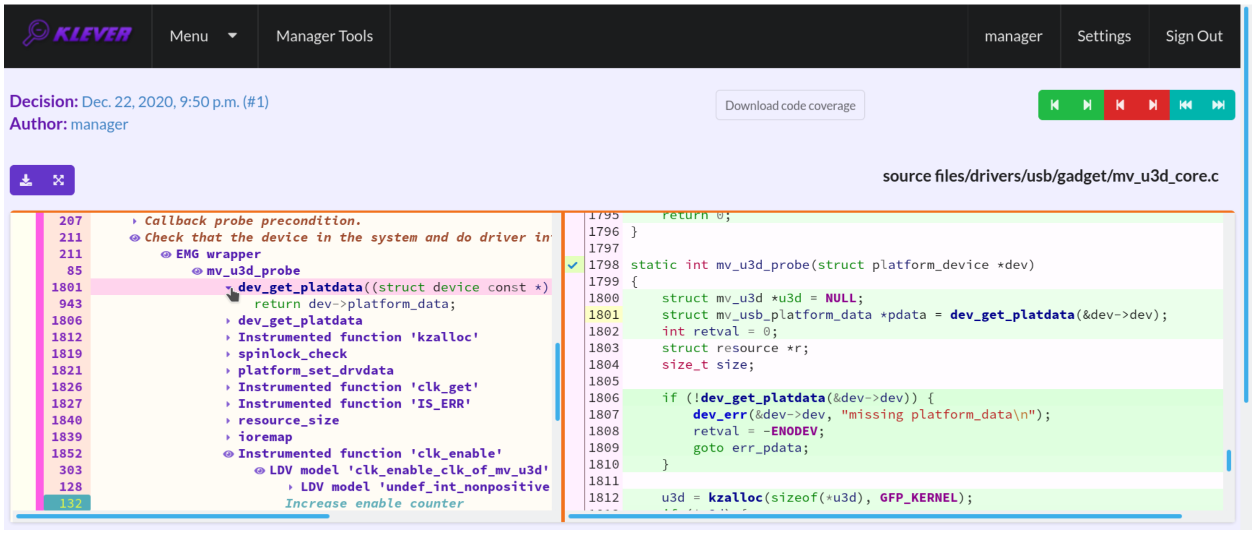
```
        return 0;
}

static int mv_u3d_probe(struct platform_device *dev)
{
        struct mv_u3d *u3d = NULL;
        struct mv_usb_platform_data *pdata = dev_get_platdata(&dev->dev);
        int retval = 0;
        struct resource *r;
        size_t size;

        if (!dev_get_platdata(&dev->dev)) {
                dev_err(&dev->dev, "missing platform_data\n");
                retval = -ENODEV;
                goto err_pdata;
        }

        u3d = kzalloc(sizeof(*u3d), GFP_KERNEL);
```

Fig. 1.24: Showing hidden declarations, statements and assumptions for functions with notes or warnings



Decision: Dec. 22, 2020, 9:50 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/mv_u3d_core.c

```
207    Callback probe precondition.
211    Check that the device in the system and do driver in
211      EMG wrapper
85         mv_u3d_probe
1801         dev_get_platdata((struct device const *)
943            return dev->platform_data;
1806         dev_get_platdata
1812         Instrumented function 'kzalloc'
1819         spinlock_check
1821         platform_set_drvdata
1826         Instrumented function 'clk_get'
1827         Instrumented function 'IS_ERR'
1840         resource_size
1839         ioremap
1852         Instrumented function 'clk_enable'
303            LDV model 'clk_enable_clk_of_mv_u3d'
128              LDV model 'undef_int_nonpositive
132                Increase enable counter
```

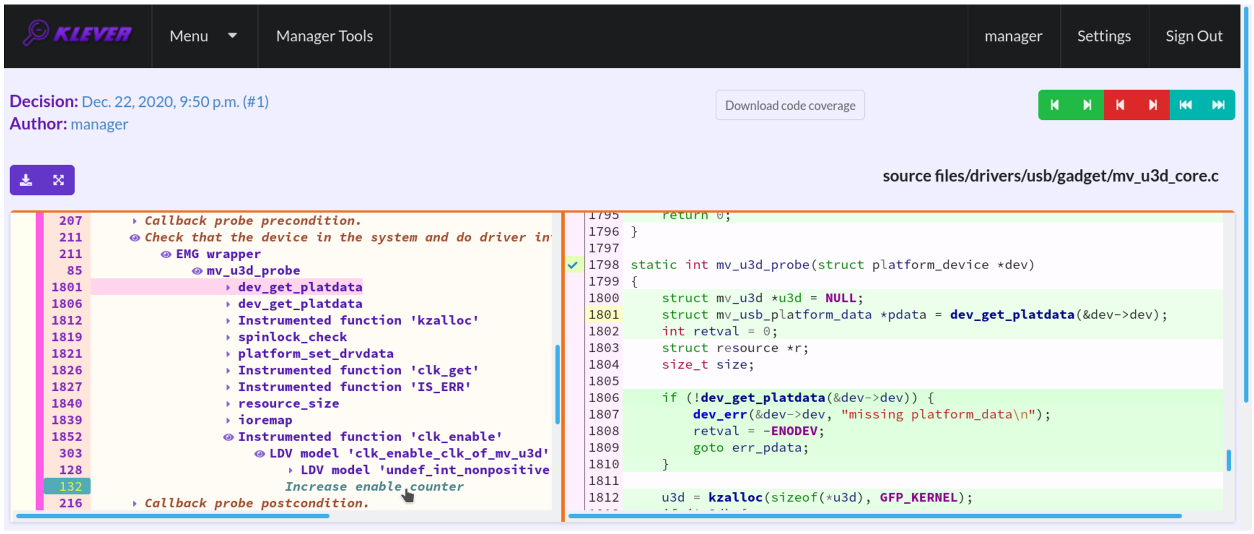
```
        return 0;
}

static int mv_u3d_probe(struct platform_device *dev)
{
    struct mv_u3d *u3d = NULL;
    struct mv_usb_platform_data *pdata = dev_get_platdata(&dev->dev);
    int retval = 0;
    struct resource *r;
    size_t size;

    if (!dev_get_platdata(&dev->dev)) {
        dev_err(&dev->dev, "missing platform_data\n");
        retval = -ENODEV;
        goto err_pdata;
    }

    u3d = kzalloc(sizeof(*u3d), GFP_KERNEL);
```

Fig. 1.25: Showing hidden declarations, statements and assumptions for functions without notes or warnings



Decision: Dec. 22, 2020, 9:50 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/mv_u3d_core.c

```
207    Callback probe precondition.
211    Check that the device in the system and do driver in
211      EMG wrapper
85         mv_u3d_probe
1801         dev_get_platdata
1806         dev_get_platdata
1812         Instrumented function 'kzalloc'
1819         spinlock_check
1821         platform_set_drvdata
1826         Instrumented function 'clk_get'
1827         Instrumented function 'IS_ERR'
1840         resource_size
1839         ioremap
1852         Instrumented function 'clk_enable'
303            LDV model 'clk_enable_clk_of_mv_u3d'
128              LDV model 'undef_int_nonpositive
132                Increase enable counter
216    Callback probe postcondition.
```

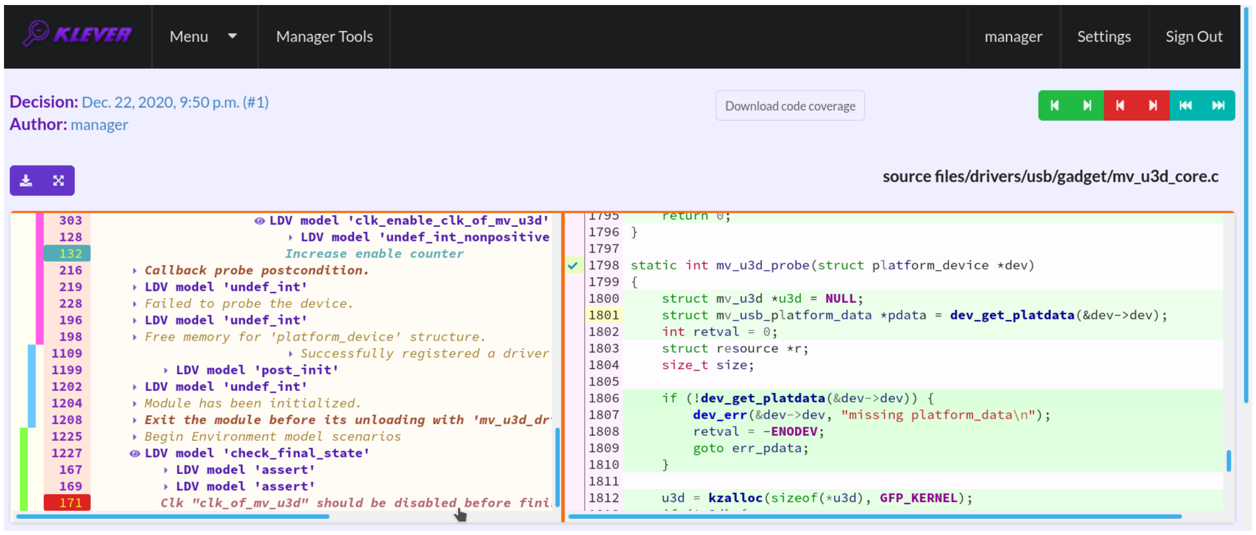
```
        return 0;
}

static int mv_u3d_probe(struct platform_device *dev)
{
    struct mv_u3d *u3d = NULL;
    struct mv_usb_platform_data *pdata = dev_get_platdata(&dev->dev);
    int retval = 0;
    struct resource *r;
    size_t size;

    if (!dev_get_platdata(&dev->dev)) {
        dev_err(&dev->dev, "missing platform_data\n");
        retval = -ENODEV;
        goto err_pdata;
    }

    u3d = kzalloc(sizeof(*u3d), GFP_KERNEL);
```

Fig. 1.26: The error trace note



Decision: Dec. 22, 2020, 9:50 p.m. (#1)
Author: manager

source files/drivers/usb/gadget/mv_u3d_core.c

```
 303              LDV model 'clk_enable_clk_of_mv_u3d'
 128                  LDV model 'undef_int_nonpositive
 132                  Increase enable counter
 216     Callback probe postcondition.
 219     LDV model 'undef_int'
 228     Failed to probe the device.
 196     LDV model 'undef_int'
 198     Free memory for 'platform_device' structure.
1109                  Successfully registered a driver
1199         LDV model 'post_init'
1202     LDV model 'undef_int'
1204     Module has been initialized.
1208     Exit the module before its unloading with 'mv_u3d_dr
1225     Begin Environment model scenarios
1227     LDV model 'check_final_state'
 167         LDV model 'assert'
 169         LDV model 'assert'
 171       Clk "clk_of_mv_u3d" should be disabled before fini
```

```
        return 0;
}

static int mv_u3d_probe(struct platform_device *dev)
{
        struct mv_u3d *u3d = NULL;
        struct mv_usb_platform_data *pdata = dev_get_platdata(&dev->dev);
        int retval = 0;
        struct resource *r;
        size_t size;

        if (!dev_get_platdata(&dev->dev)) {
            dev_err(&dev->dev, "missing platform_data\n");
            retval = -ENODEV;
            goto err_pdata;
        }

        u3d = kzalloc(sizeof(*u3d), GFP_KERNEL);
```

Fig. 1.27: The error trace warning

interfaces in the way the environment does and to check the final state. This tutorial does not consider models in detail, but you should keep in mind that Klever can detect faults not only directly in the source code under verification but also when checking something after execution of corresponding functions. For instance, this is the case for the considered error trace (Fig. 1.27).

1.2.7 Creating Marks

The analyzed unsafe corresponds to the fault that was fixed in commit 374a1020d21b to the Linux kernel. To finalize assessment you need to create a new *mark* (Fig. 1.28-Fig. 1.29):

1. Specify a verdict (**Bug** in our example).
2. Specify a status (**Fixed**).
3. Provide a description.
4. Save the mark.

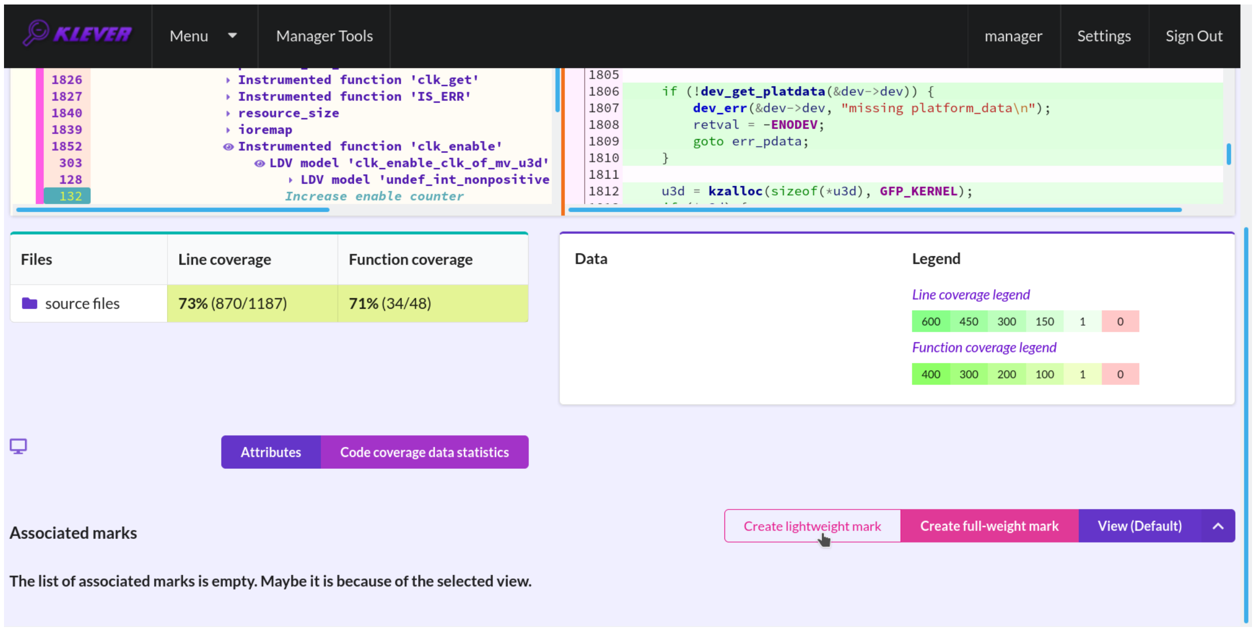
After that you will be automatically redirected to the page demonstrating changes in total verdicts (Fig. 1.30). In our example there is the only change that corresponds to the analyzed unsafe and the new mark. But in a general case there may be many changes since the same mark can match several unsafes, and you may need to investigate these changes.

After creating the mark you can see the first manually assessed unsafe (Fig. 1.31). Besides, as it was already noted, you should investigate automatically assessed unsafes by analyzing corresponding error traces and marks and by (un)confirming their associations (Fig. 1.32-Fig. 1.33).

False alarms can happen due to different reasons. There are corresponding *tags* for most common of them. You can find a complete tree of tags at *Menu* → *Marks* → *Tags* (Fig. 1.34).

Each tag has a description that is shown when covering a tag name (Fig. 1.35).

You can choose appropriate tags during creation of marks from the dropdown list (Fig. 1.36). This list can be filtered out by entering parts of tag names (Fig. 1.37).



```
1826    › Instrumented function 'clk_get'
1827    › Instrumented function 'IS_ERR'
1840    › resource_size
1839    › ioremap
1852    ⊖ Instrumented function 'clk_enable'
 303        ⊖ LDV model 'clk_enable_clk_of_mv_u3d'
 128            › LDV model 'undef_int_nonpositive
 132              Increase enable counter
```

```
	if (!dev_get_platdata(&dev->dev)) {
		dev_err(&dev->dev, "missing platform_data\n");
		retval = -ENODEV;
		goto err_pdata;
	}

	u3d = kzalloc(sizeof(*u3d), GFP_KERNEL);
```

Files	Line coverage	Function coverage
source files	**73%** (870/1187)	**71%** (34/48)

Data **Legend**

Line coverage legend

600	450	300	150	1	0

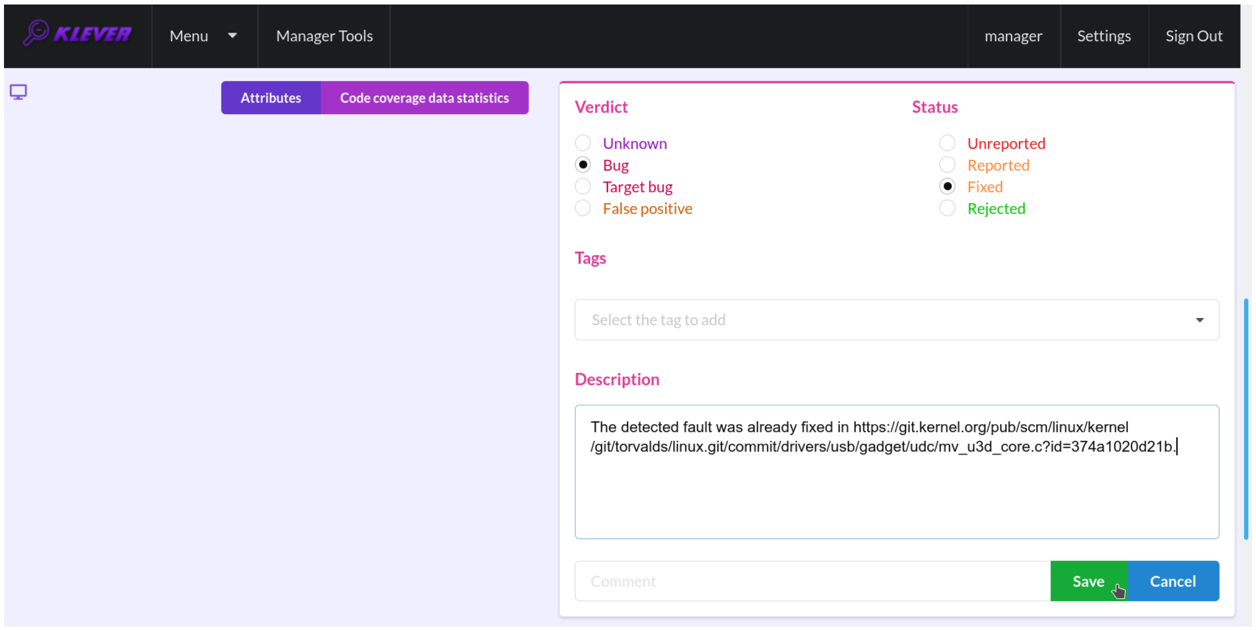
Function coverage legend

400	300	200	100	1	0

Associated marks

The list of associated marks is empty. Maybe it is because of the selected view.

Fig. 1.28: Starting the creation of a new lightweight mark



Verdict

- ○ Unknown
- ● Bug
- ○ Target bug
- ○ False positive

Status

- ○ Unreported
- ○ Reported
- ● Fixed
- ○ Rejected

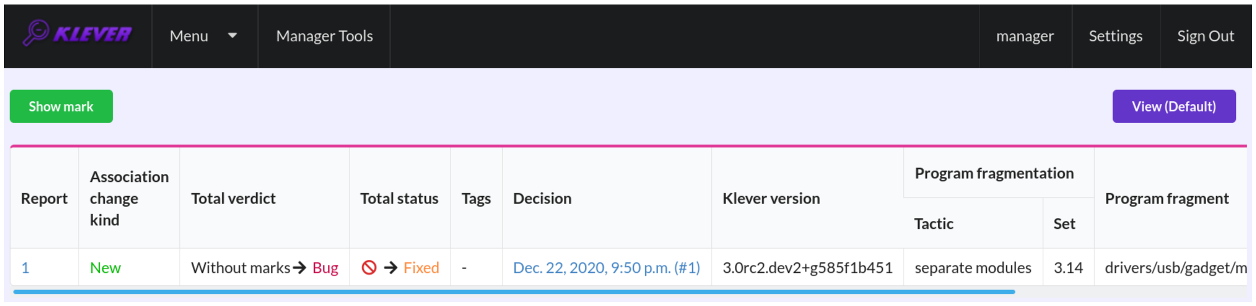
Tags

Select the tag to add

Description

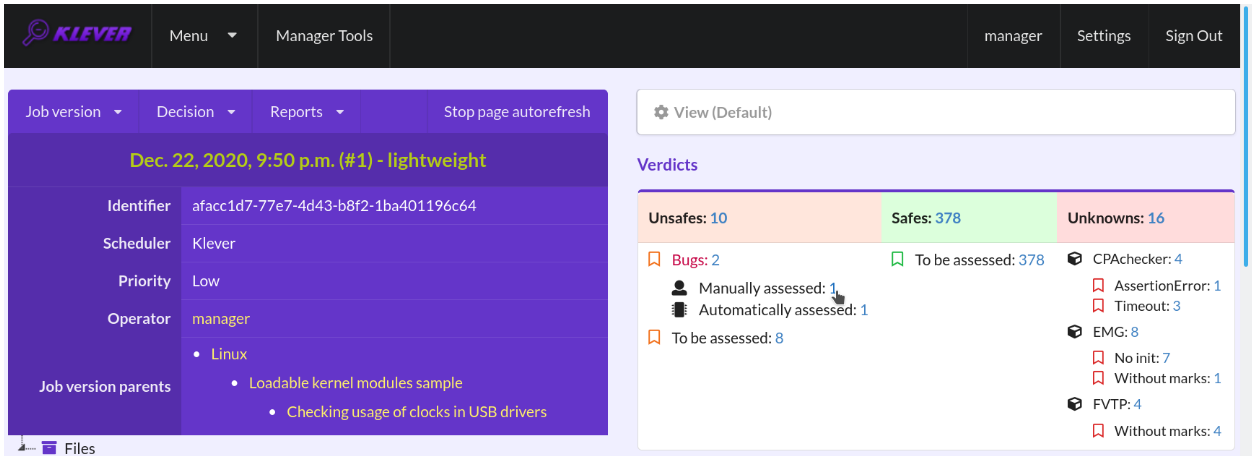
The detected fault was already fixed in https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/drivers/usb/gadget/udc/mv_u3d_core.c?id=374a1020d21b

Fig. 1.29: The creation of the new lightweight mark



Report	Association change kind	Total verdict	Total status	Tags	Decision	Klever version	Program fragmentation		Program fragment
							Tactic	Set	
1	New	Without marks → Bug	⃠ → Fixed	-	Dec. 22, 2020, 9:50 p.m. (#1)	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb/gadget/m

Fig. 1.30: Changes in total verdicts



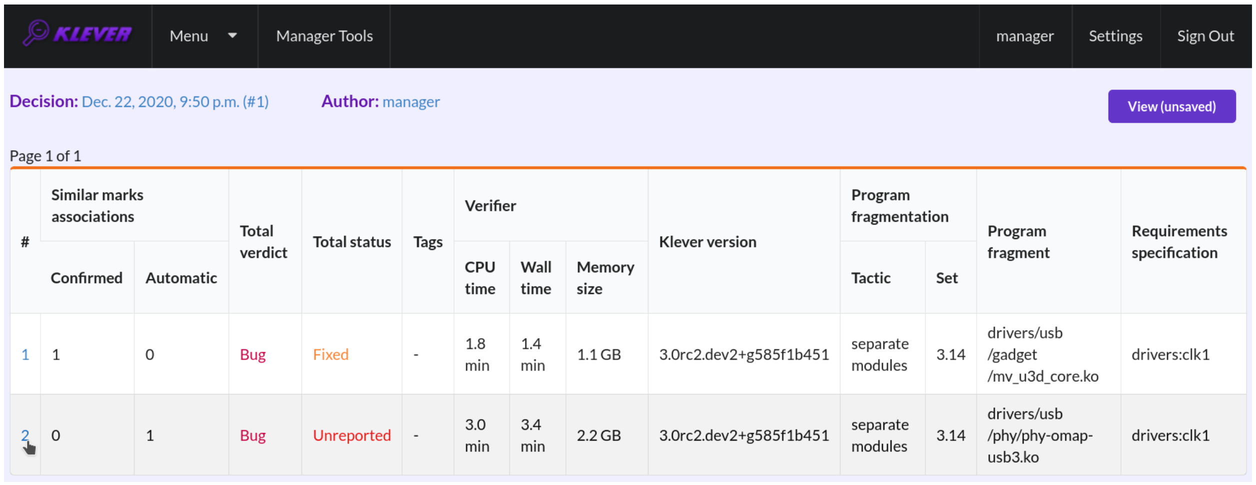
Dec. 22, 2020, 9:50 p.m. (#1) - lightweight

Identifier	afacc1d7-77e7-4d43-b8f2-1ba401196c64
Scheduler	Klever
Priority	Low
Operator	manager
Job version parents	• Linux • Loadable kernel modules sample • Checking usage of clocks in USB drivers

Verdicts

Unsafes: 10	Safes: 378	Unknowns: 16
Bugs: 2 Manually assessed: 1 Automatically assessed: 1 To be assessed: 8	To be assessed: 378	CPAchecker: 4 AssertionError: 1 Timeout: 3 EMG: 8 No init: 7 Without marks: 1 FVTP: 4 Without marks: 4

Fig. 1.31: The total number of manually assessed unsafes

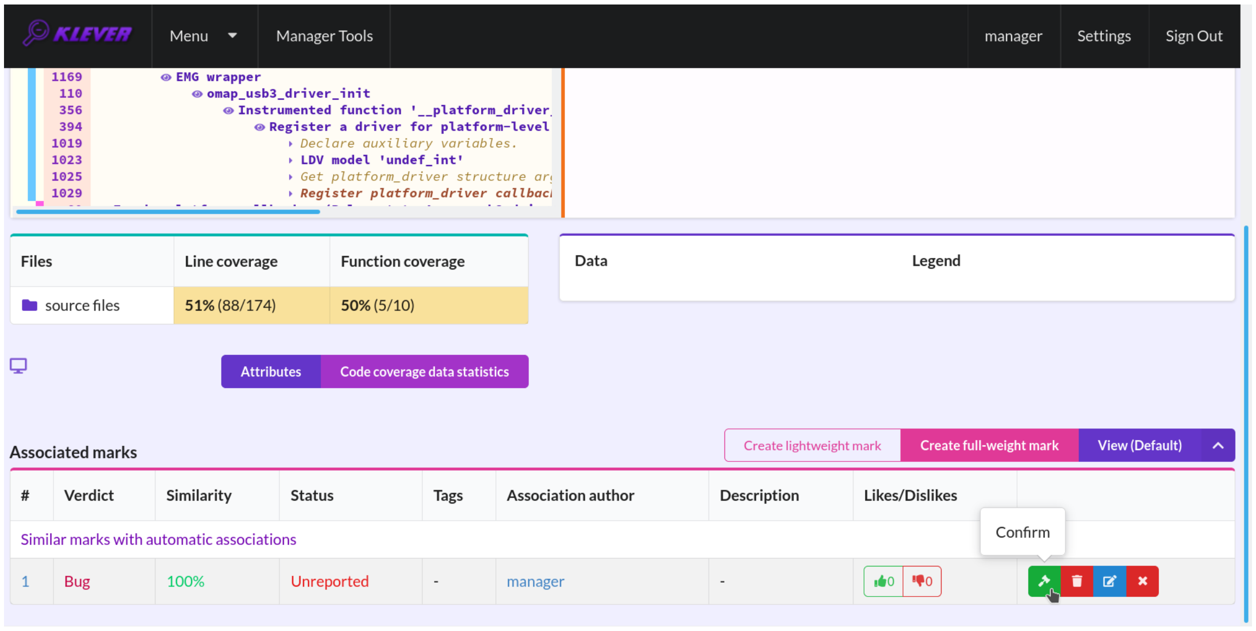


Decision: Dec. 22, 2020, 9:50 p.m. (#1) Author: manager

Page 1 of 1

#	Similar marks associations		Total verdict	Total status	Tags	Verifier			Klever version	Program fragmentation		Program fragment	Requirements specification
	Confirmed	Automatic				CPU time	Wall time	Memory size		Tactic	Set		
1	1	0	Bug	Fixed	-	1.8 min	1.4 min	1.1 GB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb /gadget /mv_u3d_core.ko	drivers:clk1
2	0	1	Bug	Unreported	-	3.0 min	3.4 min	2.2 GB	3.0rc2.dev2+g585f1b451	separate modules	3.14	drivers/usb /phy/phy-omap-usb3.ko	drivers:clk1

Fig. 1.32: Opening the error trace of the unsafe with automatic assessment



```
1169     EMG wrapper
110        omap_usb3_driver_init
356          Instrumented function '__platform_driver_
394            Register a driver for platform-level
1019             Declare auxiliary variables.
1023             LDV model 'undef_int'
1025             Get platform_driver structure ar
1029             Register platform_driver callbac
```

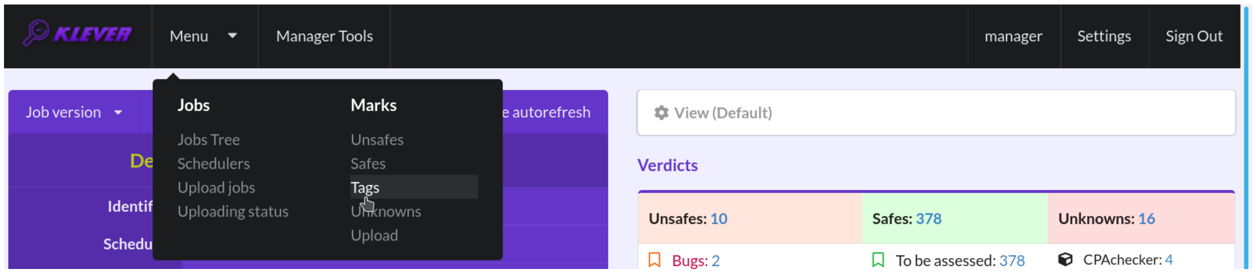
Files	Line coverage	Function coverage
source files	51% (88/174)	50% (5/10)

Data	Legend

Associated marks

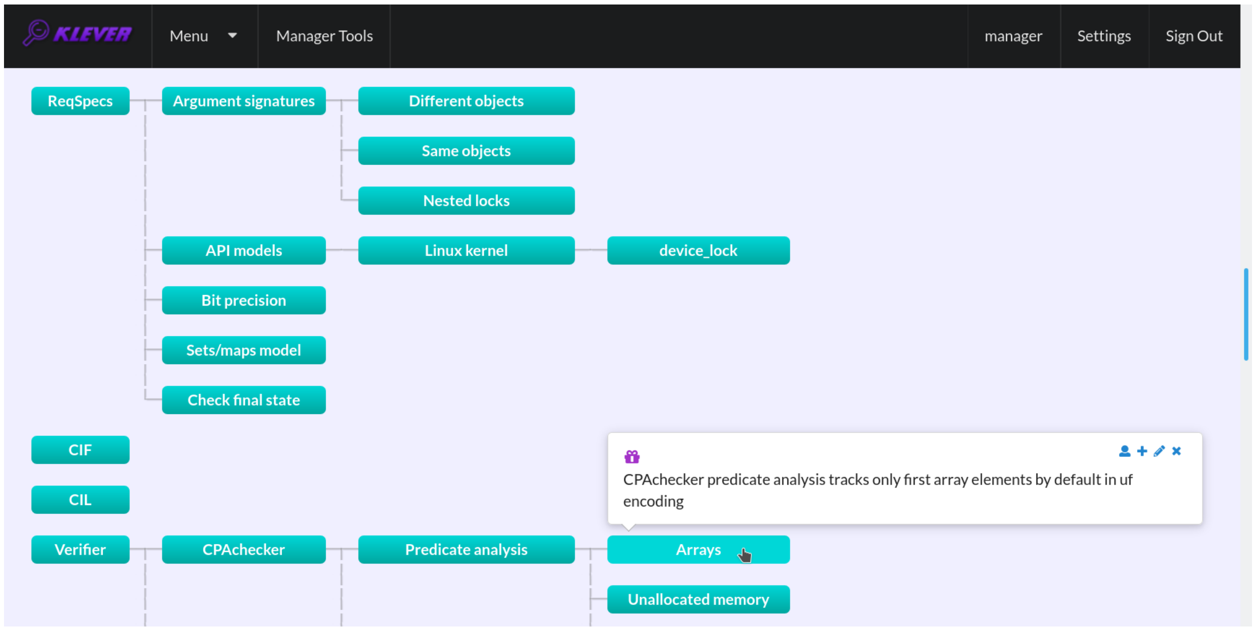
#	Verdict	Similarity	Status	Tags	Association author	Description	Likes/Dislikes	
Similar marks with automatic associations								
1	Bug	100%	Unreported	-	manager	-	0 0	

Fig. 1.33: Confirming the automatic association



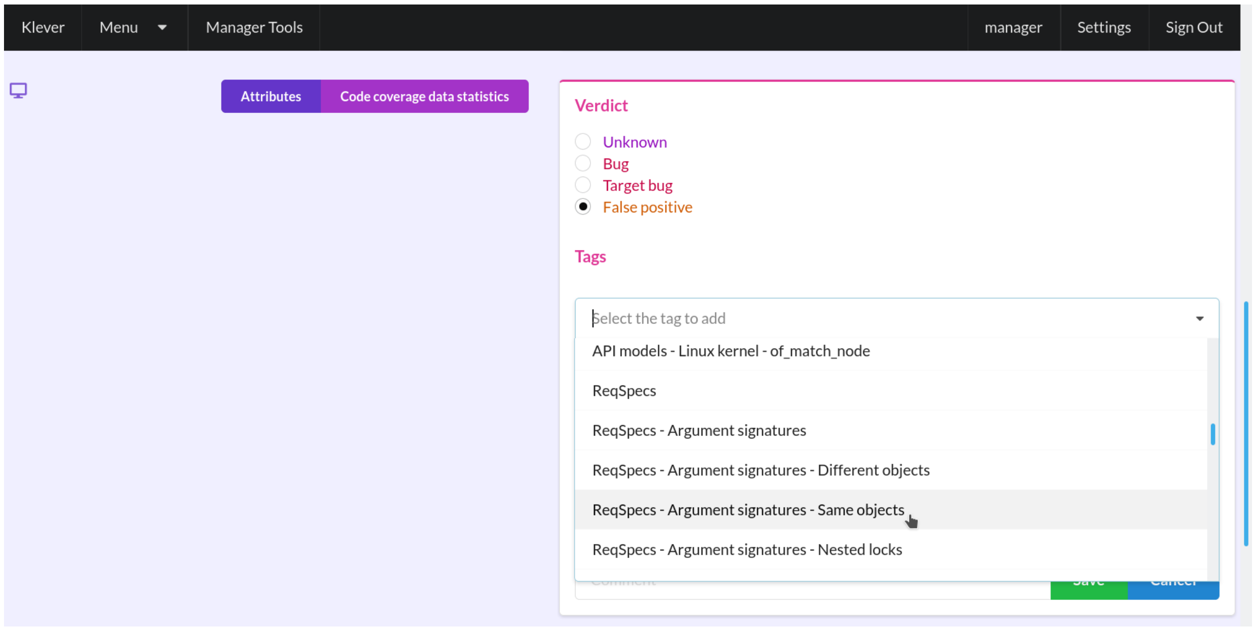
Unsafes: 10	Safes: 378	Unknowns: 16
Bugs: 2	To be assessed: 378	CPAchecker: 4

Fig. 1.34: Opening the tags page



CPAchecker predicate analysis tracks only first array elements by default in uf encoding

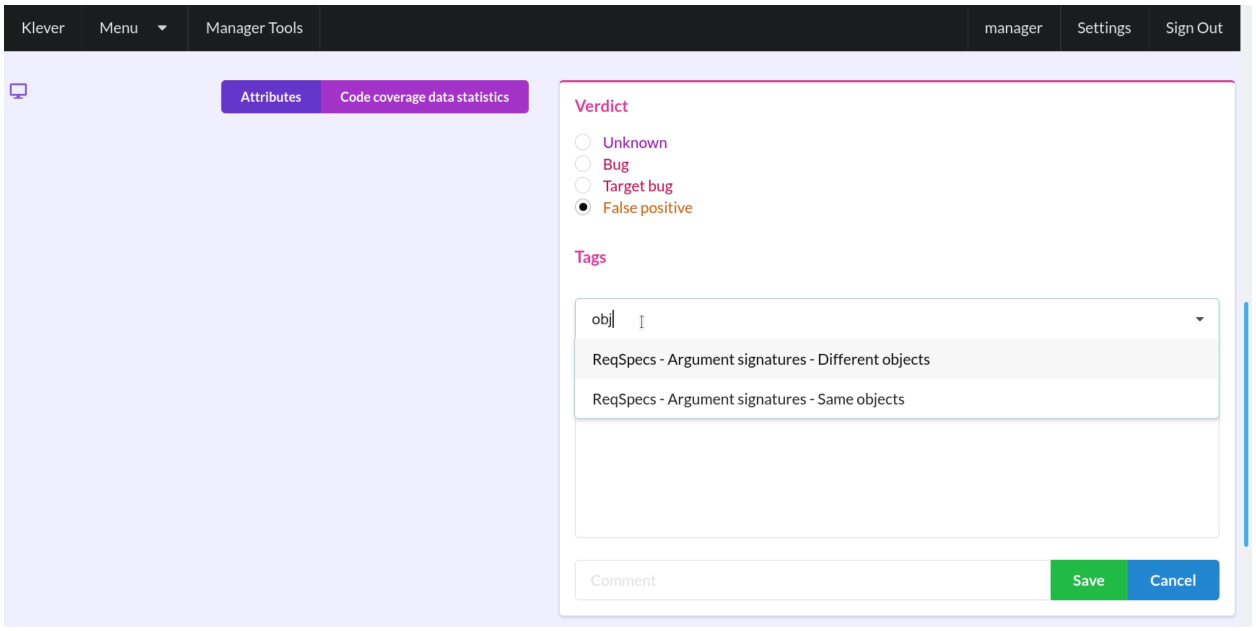
Fig. 1.35: Showing tag description



Verdict: Unknown, Bug, Target bug, [✓ False positive]

API models - Linux kernel - of_match_node
ReqSpecs
ReqSpecs - Argument signatures
ReqSpecs - Argument signatures - Different objects
ReqSpecs - Argument signatures - Same objects
ReqSpecs - Argument signatures - Nested locks

Fig. 1.36: Choosing tag from the dropdown list



Verdict

Unknown
Bug
Target bug
● False positive

Tags

obj

ReqSpecs - Argument signatures - Different objects

ReqSpecs - Argument signatures - Same objects

Comment

Fig. 1.37: Entering tag name part

1.2.8 What's Next?

We assume that you can be non-satisfied fully with a quality of obtained verification results. Perhaps, you even could not obtain them at all. This is expected since Klever is an open source software developed in the Academy and we support verification of Linux kernel loadable modules for evaluation purposes primarily. Besides, this tutorial misses many tricky activities like development of specifications and support for verification of additional software. We are ready to discuss different issues and even to fix some crucial bugs, but we do not have the manpower to make any considerable improvements for you for free.

1.3 CLI

Klever supports a command-line interface for starting solution of verification jobs, for getting progress of their solution, etc. One can use CLI to automate usage of Klever, e.g. within CI. You should note that CLI is not intended for generation of *Klever Build Bases* and expert assessment of verification results.

This section describes several most important commands and the common workflow. We used Python 3.7 to describe commands, but you can levearage any appropriate language.

1.3.1 Credentials

All commands require credentials for execution. For default *Local Deployment* they look like:

```
credentials = ('--host', 'localhost:8998', '--username', 'manager', '--password',
↪'manager')
```

1.3.2 Starting Solution of Verification Jobs

You can start solution of a verification job based on any preset verification job. For this you should find out a corresponding identifier, **preset_job_id**, e.g. using Web UI. For instance, Linux loadable kernel modules sample has identifier "c1529fbf-a7db-4507-829e-55f846044309". Then you should run something like:

```
ret = subprocess.check_output(('klever-start-preset-solution', preset_job_id,␣
↪*credentials)).decode('utf8').rstrip()
job_id = ret[ret.find(': ') + 2:]
```

After this **job_id** will keep an identifier of the created verification job (strictly speaking, it will be an identifier of a first version of the created verification job).

There are several command-line arguments that you can want to use: `--rundata` and `--replacement`.

`--rundata` `<job solution configuration file>`
If you need some non-standard settings for solution of the verification job, e.g. you have a rather powerful machine and you want to use more parallel workers to generate verification tasks to speed up the complete process, you can provide a specific job solution configuration file. We recommend to develop an appropriate solution configuration using Web UI first and then you can download this file at the verification job page (e.g. *Decision* → *Download configuration*).

`--replacement` `<JSON string or JSON file>`
If you need to add some extra files in addition to files of the preset verification job or you want to replace some of them, you can describe corresponding changes using this command-line option. For instance, you can provide a specific *Klever build base* and refer to it in **job.json**. In this case the value for this option may look like:

```
'{"job.json": "job.json", "loadable kernel modules sample.tar.gz": "loadable␣
↪kernel modules sample.tar.gz"}'
```

File **job.json** and archive **loadable kernel modules sample.tar.gz** should be placed into the current working directory.

1.3.3 Waiting for Solution of Verification Job

Most likely you will need to wait for solution of the verification job whatever it will be sucessfull or not. For this purpose you can execute something like:

```
while True:
  time.sleep(5)
  subprocess.check_call(('klever-download-progress', '-o', 'progress.json', job_id,␣
↪*credentials))

  with open('progress.json') as fp:
    progress = json.load(fp)

  if int(progress['status']) > 2:
    break
```

1.3.4 Obtaining Verification Results

You can download verification results by using such the command:

```
subprocess.check_call(('klever-download-results', '-o', 'results.json', job_id,␣
↪*credentials))
```

Then you can inspect file **results.json** somehow. Though, as it was noted, most likely you will need to analyze these results manually via Web UI.

1.4 Development of Requirement Specifications

To check requirements with Klever it is necessary to develop *requirement specifications*. This part of the user documentation describes how to do that. It will help to fix both existing requirement specifications and to develop new ones. At the moment this section touches just rules of correct usage of specific APIs while some things may be the same for other requirements.

In ideal development of any requirements specification should include the following steps:

1. Analysis and description of checked requirements.
2. Development of the requirements specification itself.
3. Testing of the requirements specification.

If you will meet some issues on any step, you should repeat the process partially or completely to eliminate them. Following subsections consider these steps in detail. As an example we consider a requirements specification devoted to correct usage of a module reference counter API in the Linux kernel.

1.4.1 Analysis and Description of Checked Requirements

At this step one should clearly determine requirements to be checked. For instance, for rules of correct usage of specific APIs it is necessary to describe related elements of APIs and situations when APIs are used wrongly. Perhaps, various versions and configurations of target programs can provide APIs differently while considered correctness rules may be the same or almost the same. If you would like to support these versions/configurations, you should also describe corresponding differences of APIs.

There are different sources that can help you to formulate requirements and to study APIs. For instance, for the Linux kernel they are as follows:

- Documentation delivered together with the source code the Linux kernel (directory `Documentation`) as well as the source code of the Linux kernel itself.
- Books, papers and blog posts devoted to development of the Linux kernel and its loadable modules such as device drivers.
- Mailing lists, including Linux Kernel Mailing List.
- The history of development in Git.

Using the latter source you can bugs fixed in target programs. These bugs can correspond to common weaknesses of C programs like buffer overflows as well as they can implicitly refer to specific requirements, in particular rules of correct usage of specific APIs.

Technically it is possible to check very different requirements within the same specification, we do not recommend to do this due to some limitations of software model checkers (*verification tools*). Nevertheless, you can formulate and check requirements related to close API elements together.

Let's consider rules of correct usage of the module reference counter API in the Linux kernel. For brevity we will not consider some elements of this API.

Linux loadable kernel modules can be unloaded just when there is no more processes using them. To notify the Linux kernel that module is necessary one should call *`try_module_get()`*.

bool **`try_module_get`**(struct module **module*)
    Try to increment the module reference count.

Parameters

- **`module`** – The pointer to the target module. Often this the given module.

Returns *True* in case when the module reference counter was increased successfully and *False* otherwise.

To give the module back one should call *`module_put()`*.

void **`module_put`** (struct module **module*)

Decrement the module reference count.

Parameters

- **`module`** – The pointer to the target module.

There are static inline stubs of these functions when module unloading is disabled via a special configuration of the Linux kernel (**CONFIG_MODULE_UNLOAD** is unset). One can consider them as well, though, strictly speaking, in this case there is no requirements for their usage.

Correctness rules can be formulated as follows:

1. One should not decrement non-incremented module reference counters. Otherwise the kernel can unload modules in use that can result to different issues.
2. Module reference counters should be decremented to their initial values before finishing operation. If this will not be the case one will not be able to unload modules ever.

1.4.2 Development of Requirements Specification

Development of each requirements specification includes the following steps:

1. Developing a model of an API.
2. Binding the model with original API elements.
3. Description of the new requirements specification.

We recommend to develop new requirement specifications on the basis of existing ones to avoid various tricky issues and to speed up the whole process considerably. Also, we recommend you to deploy Klever in the development mode. In this case you will get much more debug information that can help you to identify various issues. Moreover, you will not even need to update your Klever installation. Though Web UI supports rich means for creating, editing and other operations with verification job files including specifications, we recommend you to develop requirement specifications directly within *$KLEVER_SRC* by means of some IDE. To further reduce manual efforts using such the workflow, you can temporarily modify necessary preset verification jobs, e.g. to specify requirement specifications and program fragments of interest within `job.json`. Do not forget to not commit these temporary changes to the repository!

Developing Model

First of all you should develop a model of a considered API and specify preconditions of API usage within that model. Klever suggests to use the C programming language for this purpose while one can use some library functions having a special semantics for software model checkers, e.g. for modeling nondeterministic behavior, for using sets and maps, etc.

The model includes a *model state* that is represented as a set of global variables usually. Besides, it includes *model functions* that change the model state and check for preconditions according to semantics of the modelled API.

Ideally the model behavior should correspond to behavior of the corresponding implementation. However in practice it is rather difficult to achieve this due to complexity of the implementation and restrictions of verification tools. You can extend the implementation behavior in the model. For example, if a function can return one of several error codes in the form of the corresponding negative integers, the model can return any non-positive number in case of errors. It is not recommended to narrow the implementation behavior in the model (e.g. always return 0 though the implementation can return values other than 0) as it can result in some paths will not be considered and the overall verification quality will decrease.

In the example below there is the model state represented by global variable **ldv_module_refcounter** initialized by 1. This variable is changed within model functions **ldv_try_module_get()** and **ldv_module_put()** according to the semantics of the corresponding API.

The model makes 2 checks by means of **ldv_assert()**. The first one is within **ldv_module_put()**. It is intended to find out cases when modules decrement the reference counter without incrementing it first. The second check is within **ldv_check_final_state()** invoked by the *environment model* after modules are unloaded. It tracks that modules should decrement the reference counter to its initial value before finishing their operation.

```c
/* Definition of struct module. */
#include <linux/module.h>
/* Definition of ldv_assert() that calls __VERIFIER_error() when its argument is not␣
↪true. */
#include <ldv/verifier/common.h>
/* Definition of ldv_undef_int() invoking __VERIFIER_nondet_int(). */
#include <ldv/verifier/nondet.h>

/* NOTE Initialize module reference counter at the beginning */
static int ldv_module_refcounter = 1;

int ldv_try_module_get(struct module *module)
{
    /* NOTE Nondeterministically increment module reference counter */
    if (ldv_undef_int() == 1) {
        /* NOTE Increment module reference counter */
        ldv_module_refcounter++;
        /* NOTE Successfully incremented module reference counter */
        return 1;
    }
    else
        /* NOTE Could not increment module reference counter */
        return 0;
}

void ldv_module_put(struct module *module)
{
    /* ASSERT One should not decrement non-incremented module reference counters */
    ldv_assert(ldv_module_refcounter > 1);
    /* NOTE Decrement module reference counter */
    ldv_module_refcounter--;
}

void ldv_check_final_state(void)
{
    /* ASSERT Module reference counter should be decremented to its initial value␣
↪before finishing operation */
    ldv_assert(ldv_module_refcounter == 1);
}
```

It is worth noting that model functions do not refer their parameter **module**, i.e. they consider all modules the same. This can result to both false alarms and missed bugs. Nevertheless, often it does have sense to do such tricks to avoid too complicated models for verification, e.g. accurate tracking of dynamically created objects of interest using lists. Another important thing is modelling of nondeterminism in **ldv_try_module_get()** by invoking **ldv_undef_int()**. Thanks to it a software model checker will cover paths when **try_module_get()** can successfully increment the module reference counter and when this is not the case.

In the example above you can see comments starting with words **NOTE** and **ASSERT**. These comments are so called *model comments*. They emphasize expressions and statements that make some important actions, e.g. changing the model state. Later these comments will be used during visualization and expert assessment of verification results. You should place model comments just before corresponding expressions and statements. Each model comment has to occupy the only line.

The given API model is placed into a separate C file that will be considered alongside the source code of verified modules. A bit later we will discuss how to name this file and where to place it.

Binding Model with Original API Elements

To activate the API model you should bind model functions to points of use of original API elements. For this purpose we use an aspect-oriented extension for the C programming language. Below there is a content of an aspect file for the considered example. It replaces calls to functions *try_module_get()* and *module_put()* with calls to corresponding model functions **ldv_try_module_get()** and **ldv_module_put()**.

```
before: file ("$this") {
/* Definition of struct module. */
#include <linux/module.h>

extern int ldv_try_module_get(struct module *module);
extern void ldv_module_put(struct module *module);
}

around: call(bool try_module_get(struct module *module))
{
    return ldv_try_module_get(module);
}

around: call(void module_put(struct module *module))
{
    ldv_module_put(module);
}
```

It is not hard to accomplish this aspect file with bingins for static inline stubs of these functions.

Description of New Requirements Specification

Bases of requirement specifications are located in JSON files corresponding to projects, e.g. `Linux.json`, within directory *$KLEVER_SRC*`/presets/jobs/specifications`. Also, there is corresponding directory `specifications` in all verification jobs. Each requirements specification can contain one or more C source files with API models. We suggest to place these files according to the hierarchy of files and directories with implementation of the corresponding API elements. For example, you can place the C source file from the example above into *$KLEVER_SRC*`/presets/jobs/specifications/linux/kernel/module.c` as the module reference counter API is implemented in file `kernel/module.c` of the Linux kernel.

Additional files such as aspect files should be placed in the same way as C source files but using appropriate extensions, e.g. *$KLEVER_SRC*`/presets/jobs/specifications/linux/kernel/module.aspect`. You should not specify aspect files within the base since they are found automatically.

As a rule identifiers of requirement specifications are chosen according to relative paths of C source files with main API models. For example, for the considered example it is **kernel:module**. Requirement specification bases represent these identifiers in the tree form.

1.4.3 Testing of Requirements Specification

We recommended to carry out different types of testing to check syntactic and semantic correctness of requirement specifications during their development and maintenance:

1. Developing a set of rather simple test programs, e.g. external Linux loadable kernel modules, using the modelled API incorrectly and correctly. The verification tool should report Unsafes and Safes respectively unless you will develop such the test programs that do not fit your models.
2. Validating whether known violations of checked requirements can be found. Ideally the verification tool should detect violations before their fixes and it should not report them after that. In practice, the verification tool can find other bugs or report false alarms, e.g. due to inaccurate environment models.
3. Checking target programs against requirement specifications. For example, you can check all loadable kernel modules of one or several versions or configurations of the Linux kernel or consider some relevant subset of them, e.g. USB device drivers when developing appropriate requirement specifications. In ideal, a few false alarms should be caused by incorrectness or incompleteness of requirement specifications.

For item 1 you should consider existing test cases and their descriptions in the following places:

- *$KLEVER_SRC*`/klever/cli/descs/linux/testing/requirement specifications/tests/linux/kernel/module`
- *$KLEVER_SRC*`/klever/cli/descs/linux/testing/requirement specifications/desc.json`
- *$KLEVER_SRC*`/presets/jobs/linux/testing/requirement specifications`

For item 2 you should consider existing test cases and their descriptions in the following places:

- *$KLEVER_SRC*`klever/cli/descs/linux/validation/2014 stable branch bugs/desc.json`
- *$KLEVER_SRC*`presets/jobs/linux/validation/2014 stable branch bugs`

In addition, you should refer *How to generate build bases for testing Klever* to obtain build bases necessary for testing and validation.

Requirement specifications can be incorrect and/or incomplete. In this case test and validation results will not correspond to expected ones. It is necessary to fix and improve the requirements specification while you will have appropriate resources. Also, you should take into account that non-ideal results can be caused by other factors, for example:

- Incorrectness and/or incompleteness of *environment models*.
- Inaccurate algorithms of the verification tool.
- Generic restrictions of approaches to development of requirement specifications, e.g. when using counters rather than accurate representations of objects.

1.4.4 Using Argument Signatures to Distinguish Objects

As it was specified above, it may be too hard for the verification tool to accurately distinguish different objects like modules and mutexes since this can involve complicated data structures. From the other side treating all objects the same, e.g. by using integer counters when modeling operations on them, can result in a large number of false alarms as well as missed bugs. For instance, if a Linux loadable kernel module acquires two different mutexes sequentially, the verification tool will detect that the same mutex can be acquired twice that will be reported as an error.

To distinguish objects we suggest using so-called *argument signatures* — identifiers of objects which are calculated syntactically on the basis of the expressions passed as corresponding actual parameters. Generally speaking different objects can have identical argument signatures. Thus, it is impossible to distinguish them in this way. Ditto the same object can have different argument signatures, e.g. when using aliases. Nevertheless, our observation shows that in most cases the offered approach allows to distinguish objects rather precisely.

Requirement specifications with argument signatures differ from requirement specifications which were considered earlier. You need to specify different model variables, model functions and preconditions for each calculated argument signature. For the example considered above it is necessary to replace:

```
/* NOTE Initialize module reference counter at the beginning */
static int ldv_module_refcounter = 1;

int ldv_try_module_get(struct module *module)
{
    /* NOTE Nondeterministically increment module reference counter */
    if (ldv_undef_int() == 1) {
        /* NOTE Increment module reference counter */
        ldv_module_refcounter++;
        /* NOTE Successfully incremented module reference counter */
        return 1;
    }
    else
        /* NOTE Could not increment module reference counter */
        return 0;
}
```

with:

```
// for arg_sign in arg_signs
/* NOTE Initialize module reference counter{{ arg_sign.text }} at the beginning */
static int ldv_module_refcounter{{ arg_sign.id }} = 1;

int ldv_try_module_get{{ arg_sign.id }}(struct module *module)
{
    /* NOTE Nondeterministically increment module reference counter{{ arg_sign.text }}
↪ */
    if (ldv_undef_int() == 1) {
        /* NOTE Increment module reference counter{{ arg_sign.text }} */
        ldv_module_refcounter{{ arg_sign.id }}++;
        /* NOTE Successfully incremented module reference counter{{ arg_sign.text }}␣
↪*/
        return 1;
    }
    else
        /* NOTE Could not increment module reference counter{{ arg_sign.text }} */
        return 0;
}
// endfor
```

In bindings of model functions with original API elements it is necessary to specify for what function arguments it i necessary to calculate argument signatures. For instance, it is necessary to replace:

```
around: call(bool try_module_get(struct module *module))
{
    return ldv_try_module_get(module);
}
```

with:

```
around: call(bool try_module_get(struct module *module))
{
    return ldv_try_module_get_$arg_sign1(module);
}
```

Models and bindings that use argument signatures should be described differently within requirement specification bases. It is recommended to study how to do this on the base of existing examples, say, **kernel:locking:mutex**.

You can find more details about the considered approach in [N13].

1.5 Developer Documentation

1.5.1 How to Write This Documentation

This documentation is created using Sphinx from reStructuredText source files. To improve existing documentation or to develop the new one you need to read at least the following chapters of the Sphinx documentation:

1. Defining document structure.
2. Adding content.
3. Running the build.
4. reStructuredText Primer.
5. Sphinx Markup Constructs.
6. Sphinx Domains (you can omit language specific domains).

Please, follow these advises:

1. Do not think that other developers and especially users are so smart as you are.
2. Clarify ambiguous things and describe all the details without missing anything.
3. Avoid and fix misprints.
4. Write each sentence on a separate line.
5. Do not use blank lines except it is required.
6. Write a new line at the end of each source file.
7. Break sentences longer than 120 symbols to several lines if possible.

To develop documentation it is recommended to use some visual editor.

Warning: Please do not reinvent the wheel! If you are a newbie then examine carefully the existing documentation and create the new one on that basis. Just if you are a guru then you can suggest to improve the existing documentation.

1.5.2 Using Git Repository

Klever source code resides in the Git repository. There is plenty of very good documentation about Git usage. This section describes just rules specific for the given project.

Update

1. Periodically synchronize your local repository with the main development repository (it is available just internally at ISP RAS):

```
branch $ git fetch origin
branch $ git remote prune origin
```

Note: This is especially required when you are going to create a new branch or to merge some branch to the master branch.

2. Pull changes if so:

```
branch $ git pull --rebase origin branch
```

Warning: Forget about pulling without rebasing!

3. Resolve conflicts if so.

Fixing Bugs and Implementing New Features

1. One must create a new branch to fix each individual bug or implement a new feature:

```
master $ git checkout -b fix-conf
```

Warning: Do not intermix fixes and implementation of completely different bugs and features into one branch. Otherwise other developers will need to wait or to make some tricky things like cherry-picking and merging of non-master branches. Eventually this can lead to very unpleasant consequences, e.g. the master branch can be broken because of one will merge there a branch based on another non working branch.

2. Push all new branches to the main development repository. As well re-push them at least one time a day if you make some commits:

```
fix-conf $ git push origin fix-conf
```

3. Merge the master branch into your new branches if you need some recent bug fixes or features:

```
fix-conf $ git merge master
```

Note: Do not forget to update the master branch from the main development repository.

Note: Do not merge remote-tracking branches.

4. Ask senior developers to review and to merge branches to the master branch when corresponding bugs/features are fixed/implemented.

5. Delete merged branches:

```
master $ git branch -d fix-conf
```

1.5.3 Releases

Generally we follow the same rules as for development of the Linux kernel.

Each several months a new release will be issued, e.g. 0.1, 0.2, 1.0.

Just after this a merge window of several weeks will be opened. During the merge window features implemented after a previous merge window or during the given one will be merged to master.

After the merge window just bug fixes can be merged to the master branch. During this period we can issue several release candidates, e.g. 1.0-rc1, 1.0-rc2.

In addition, after issuing a new release we can decide to support a stable branch. This branch will start from a commit corresponding to the given release. It can contain just bug fixes relevant to an existing functionality and not to a new one which is supported within a corresponding merge window.

1.5.4 Updating List of Required Python Packages

To update the list of required Python packages first you need to install Klever package from scratch in the newly created virtual environment without using the old *requirements.txt* file. Run the following commands within *$KLEVER_SRC*:

```
$ python3 -m venv venv
$ source venv/bin/activate
$ pip install -e .
```

This will install latest versions of required packages. After confirming that Klever works as expected, you should run the following command within *$KLEVER_SRC*:

```
$ python -m pip freeze > requirements.txt
```

Updated list of requirements will be saved and should be committed to the repository afterwards.

1.5.5 How to generate build bases for testing Klever

Most likely you can get actual, prepared in advance build bases for testing Klever from *ldvuser@ldvdev:/var/lib/klever/workspace/Branches-and-Tags-Processing/build-bases.tar.gz* (this works just within the ISP RAS local network).

To generate build bases for testing Klever you need to perform following preliminary steps:

1. Install Klever locally for development purposes according to the user documentation (see *Deployment*).
2. Create a dedicated directory for sources and build bases and move to it. Note that there should be quite much free space. We recommend at least 100 GB. In addition, it would be best of all if you will name this directory “build bases” and create it within the root of the Klever Git repository (this directory is not tracked by the repository).
3. Clone a Linux kernel stable Git repository to *linux-stable* (scripts prepare build bases for different versions of the Linux kernel for which the Git repository serves best of all), e.g.:

   ```
   $ git clone https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/␣
   ↪linux-stable
   ```

   You can use alternative sources of the Git repository, if the above one is not working well and fast enough:

   1. https://kernel.googlesource.com/pub/scm/linux/kernel/git/stable/linux-stable
   2. https://github.com/gregkh/linux

4. Read notes regarding the compiler after the end of this list.
5. Run the following command to find out available descriptions of build bases for testing Klever:

```
$ klever-build -l
```

6. Select appropriate build bases descriptions and run the command like below:

```
$ klever-build "linux/testing/requirement specifications" "linux/testing/common␣
↪models"
```

7. Wait for a while. Prepared build bases will be available within directory "build bases". Note that there will be additional identifiers, e.g. "build bases/linux/testing/6e6e1c". These identifiers are already specified within corresponding preset verification jobs.
8. You can install prepared build bases using deployment scripts, but it is boring. If you did not follow an advice regarding the name and the place of the dedicated directory from item 2, you can create a symbolic link with name "build bases" that points to the dedicated directory within the root of the Klever Git repository.

Providing an appropriate compiler

Most of build bases for testing Klever could be built using GCC 4.8 on Debian or Ubuntu. Otherwise there is an explicit division of build bases descriptions, e.g.:

- linux/testing/environment model specifications/gcc48
- linux/testing/environment model specifications/gcc63

(the former requires GCC 4.8 while the latter needs GCC 6.3 at least).

That's why you may need to get GCC 4.8 and make it available through PATH. Users of some other Linux distributions, e.g. openSUSE 15.1, can leverage the default compiler for building all build bases for testing Klever.

The simplest way to get GCC 4.8 on Ubuntu is to execute the following commands:

```
$ sudo apt update
$ sudo apt install gcc-4.8
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-7 70
$ sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-4.8 48
$ sudo update-alternatives --config gcc
```

(after executing the last command you need to select GCC 4.8; do not forget to make v.v. after preparing build bases!)

1.5.6 Generating Bare CPAchecker Benchmarks

Development of Klever and development of CPAchecker are not strongly coupled. Thus, verification tasks that are used for testing/validation of Klever including different versions and configurations of CPAchecker as back-ends may be useful to track regressions of new versions of CPAchecker. This should considerably simplify updating CPAchecker within Klever (this process usually involves a lot of various activities both in Klever and in CPAchecker; these activities can take enormous time to be completed that complicates and postpones updates considerably). In addition, this is yet another test suite for CPAchecker. In contrast to other test suites this one likely corresponds to the most industry close use cases.

One can (re-)generate bare CPAchecker benchmarks almost automatically. To do this it is recommended to follow next steps:

1. Clone https://gitlab.com/sosy-lab/software/ldv-klever-benchmarks.git or git@gitlab.com:sosy-lab/software/ldv-klever-benchmarks.git once.
2. After some changes within Klever specifications, configurations and test cases you need to solve appropriate verification jobs. To avoid some non-determinism it is better to use the same machine, e.g. LDV Dev, to do this. Though particular verification jobs to be solved depend on changes made, in ideal, it is much easier to consider all verification jobs at once to avoid any tricky interdependencies (even slight improvements or fixes of some specifications may result in dramatic and unexpected changes in some verification results).
3. Download archives with verifier input files for each solved verification jobs to the root directory of the cloned repository.
4. Run "python3 make-benchs.py" there.
5. Estimate changes in benchmarks and verification tasks (there is not any formal guidance). If you agree with these changes, then you need to commit them and to push to the remote. After that one may expect that new commits to trunk of the CPAchecker repository will be checked for regressions against an updated test suite.

1.5.7 Using PyCharm IDE

To use PyCharm IDE for developing Klever follow the following steps.

Installation

1. Download PyCharm Community from https://www.jetbrains.com/pycharm/download/ (below all settings are given for version 2018.8.8, you have to adapt them for your version by yourself).
2. Follow installation instructions provided at that site.

Setting Project

At the "Welcome to PyCharm" window:

1. Specify your preferences.
2. *Open*.
3. Specify the absolute path to directory *$KLEVER_SRC*.
4. *OK*.

Configuring the Python Interpreter

1. *File* → *Settings* → *Project: Klever* → *Project Interpreter* → *Settings* → *Show all...*.
2. Select the Python interpreter from the Klever Python virtual environment.
3. *OK*.
4. Select the added Python interpreter from the list and press `Enter`.
5. Input *Python 3.7 (klever)* in field *name*.
6. *OK*.
7. For the rest projects select *Python 3.7 (klever)* in field *Project Interpreter*.

Setting Run/Debug Configuration

Common run/debug configurations are included into the Klever project. Common configurations with names starting with **$** should be copied to configurations with names without **$** and adjusted in accordance with instructions below. If you want to adjust configurations with names that not starting with **$** you also have to copy them before.

1. *Run* → *Edit Configurations. . . .*

Klever Bridge Run/Debug Configuration

Note: This is available just for PyCharm Professional.

- Specify *0.0.0.0* in field *Host* if you want to share your Klever Bridge to the local network.
- Specify your preferred port in field *Port*.

Note: To make your Klever Bridge accessible from the local network you might need to set up your firewall accordingly.

Klever Core Run/Debug Configuration

This run/debug configuration is only useful if you are going to debug Klever Core.

- Extend existing value of environment variable `PATH` so that CIF (`cif` or `compiler`), Aspectator (`aspectator`) and CIL (`toplever.opt`) binaries could be found (edit value of field *Environment variables*).
- Specify the absolute path to the working directory in field *Working directory*.

  Note: Place Klever Core working directory somewhere outside the main development repository.

  Note: Klever Core will search for its configuration file `core.json` in the specified working directory. Thus, the best workflow to debug Klever Core is to set its working directory to the one created previously when it was run without debugging. Besides, you can provide this file by passing its name as a first parameter to the script.

Documentation Run/Debug Configuration

Specify another representation of documenation in field *Command* if you need it.

Testing

Klever Bridge Testing

Note: This is available just for PyCharm Professional.

1. *Tools* → *Run manage.py Task. . .* :

```
manage.py@bridge > test
```

Note: To start tests from console:

```
$ cd bridge
$ python3 manage.py test
```

Note: Another way to start tests from console:

```
$ python3 path/to/klever/bridge/manage.py test bridge users jobs reports marks service
```

Note: The test database is created and deleted automatically. If the user will interrupt tests the test database will preserved and the user will be asked for its deletion for following testing. The user should be allowed to create databases (using command-line option *–keedb* does not help).

Note: PyCharm has reach abilities to analyse tests and their results.

Additional documentation

A lot of usefull documentation for developing Django projects as well as for general using of the PyCharm IDE is available at the official site.

1.5.8 Extended Violation Witness Format

TODO: Translate from Russian.

1.5.9 Error Trace Format

TODO: Translate from Russian.

1.5.10 Code Coverage Format

TODO: Translate from Russian.

1.6 Glossary

Environment model Environment models emulate interactions of target programs or *program fragments* like Linux kernel loadable modules with their environment like libraries, user inputs, interruptions and so on. Ideally they should cover only those interaction scenarios that are possible during real executions, but usually this is not the case, so false alarms and missing bugs take place. Klever generates each environment model on the basis of specifications and it is represented as a number of additional C source files (*models*) bound with original ones through instrumentation.

$KLEVER_SRC A path to a root directory of a Klever source tree.

$KLEVER_DEPLOY_DIR A path to a directory where Klever should be deployed. Although this directory can be one of standard ones like `/usr/local/bin` or `/bin`, it is recommended to use some specific one.

$SSH_RSA_PRIVATE_KEY_FILE A path to a file with SSH RSA private key. It is not recommended to use your sensitive keys. Instead either create and use a specific one or use keys that are accepted in your groups to enable an access to other group members.

$OS_USERNAME Username used to login to OpenStack.

$INSTANCES A number of OpenStack instances to be deployed.

Bibliography

[N13] Novikov E.M. Building Programming Interface Specifications in the Open System of Componentwise Verification of the Linux Kernel. Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), volume 24, pp. 293-316. 2013. https://doi.org/10.15514/ISPRAS-2013-24-13. (In Russian)

Index