Klever Documentation

ISP RAS

Jan 01, 2020

Contents

1 Contents **3**
1.1 Deployment 3
1.2 Tutorial 10
1.3 Developer Documentation 28
1.4 Glossary 35

Index **37**

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 Klever Python package you should run:

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

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

  ```
  $ pip install -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 it is also required to update Klever Python package itself (e.g. if you updated *$KLEVER_SRC*), then one additional command must be executed:

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

This additional command, however, should be skipped if Klever Python package was installed in *editable* mode (with -e flag).

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.

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:

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

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.

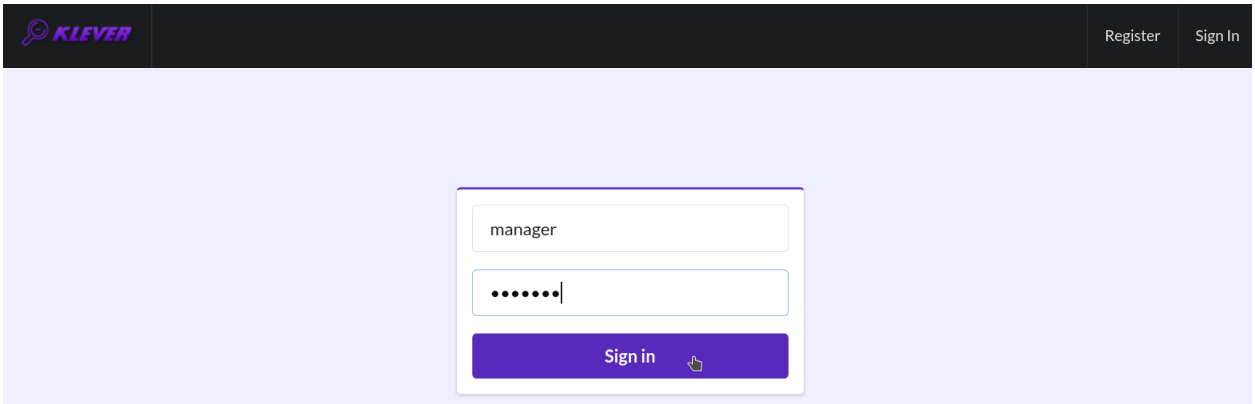


Fig. 1.1: Signing in

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).

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

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

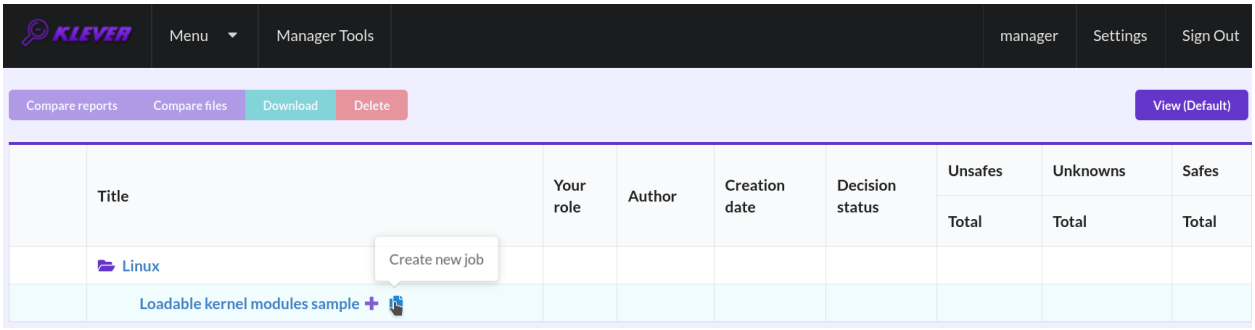


Fig. 1.2: Starting the creation of a new job

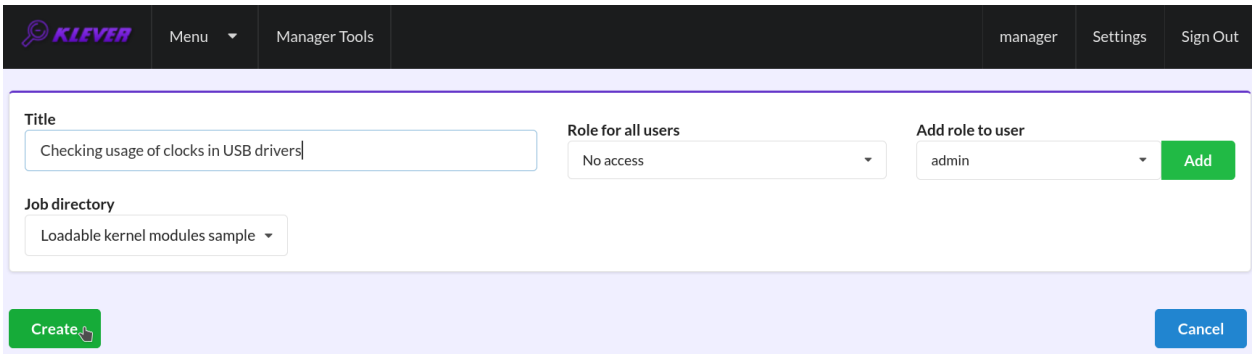


Fig. 1.3: The creation of the new job

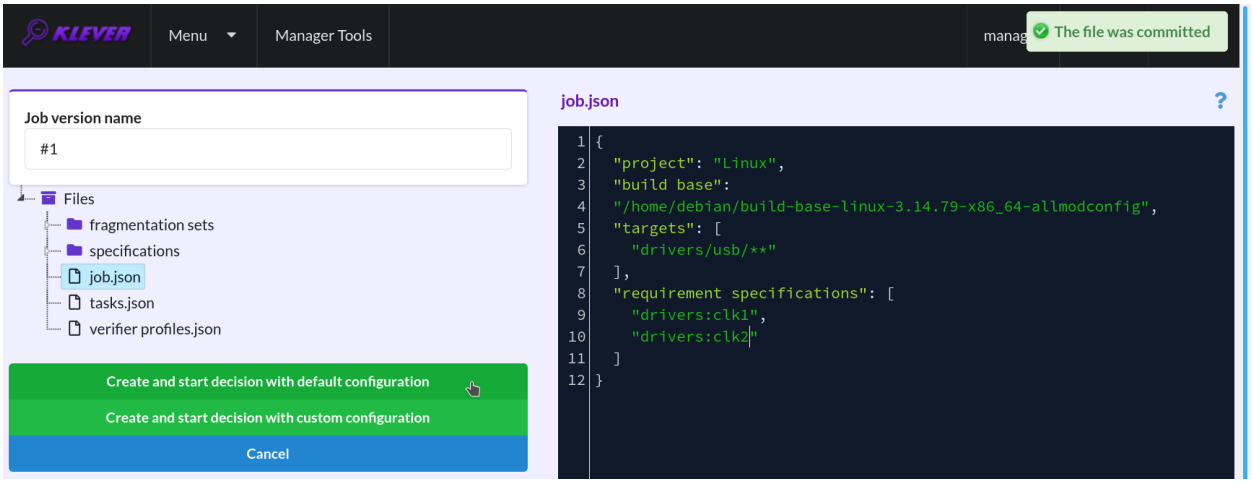


Fig. 1.4: Configuring the first job version and starting its decision

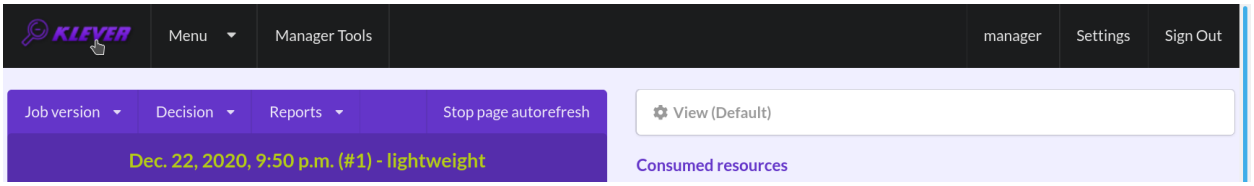


Fig. 1.5: Opening the job tree page

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
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).

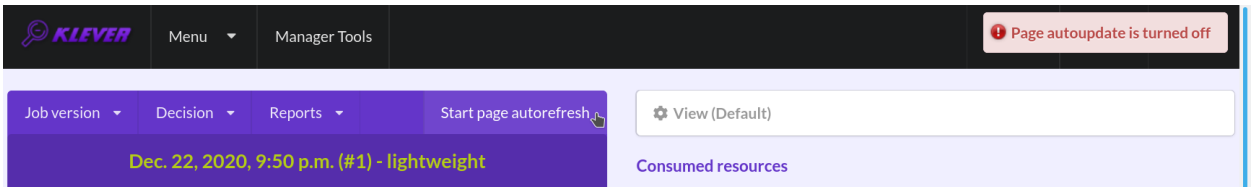


Fig. 1.6: Switching on the automatic refresh of the job version/decision page

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).

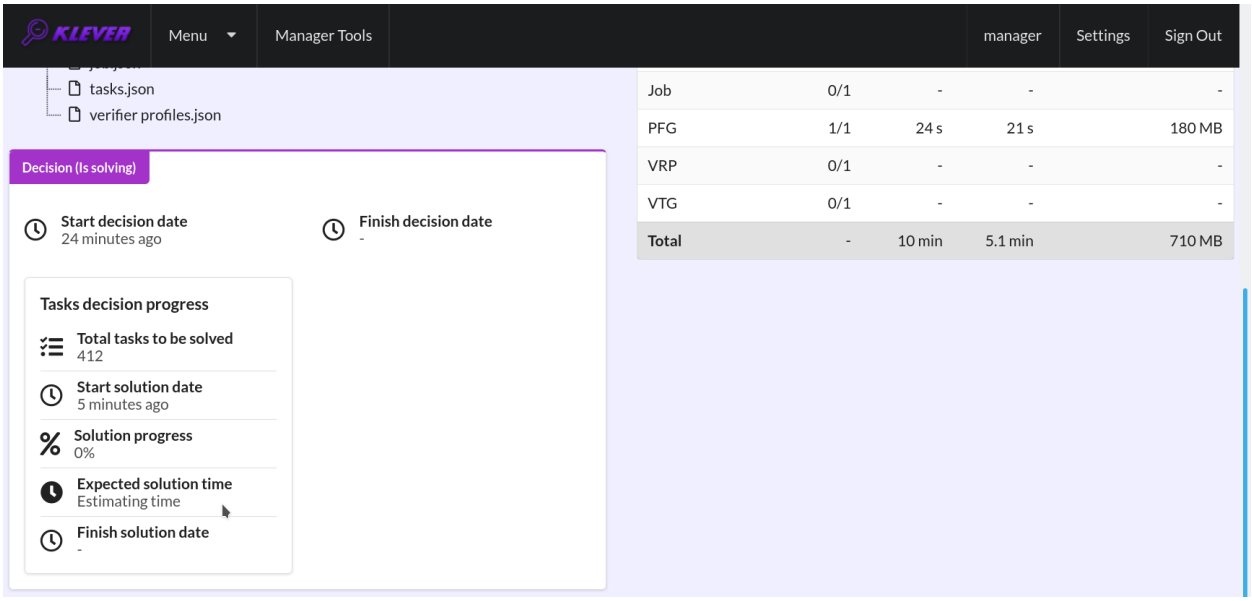


Fig. 1.7: The progress of the decision of the job version (estimating a remaining time)

[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.

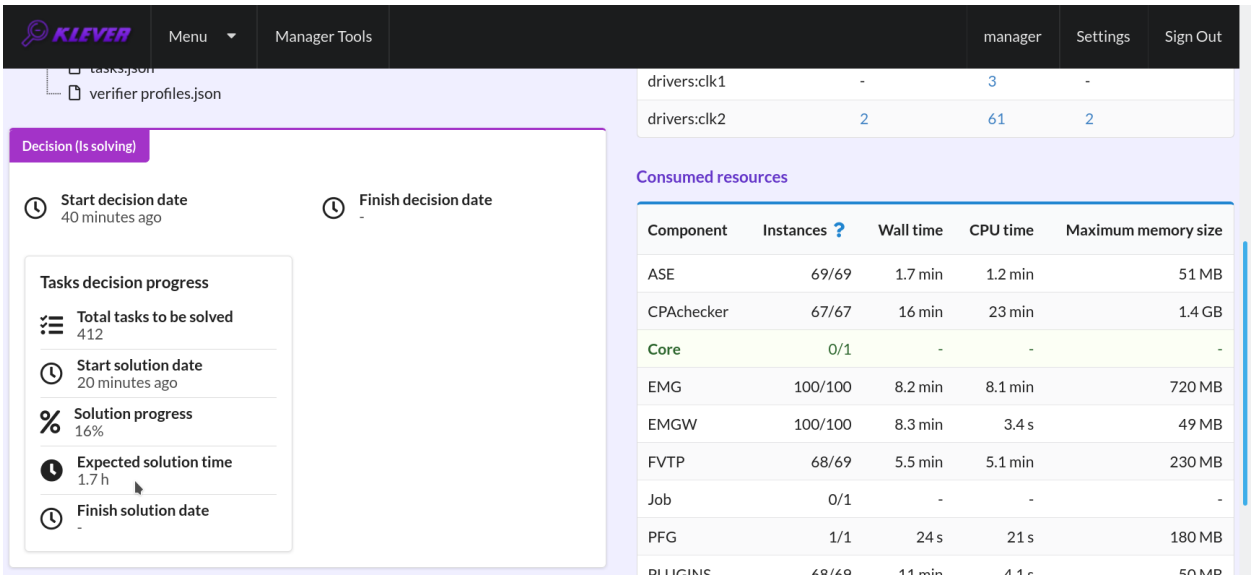


Fig. 1.8: The progress of the decision of the job version (the remaining time is estimated)

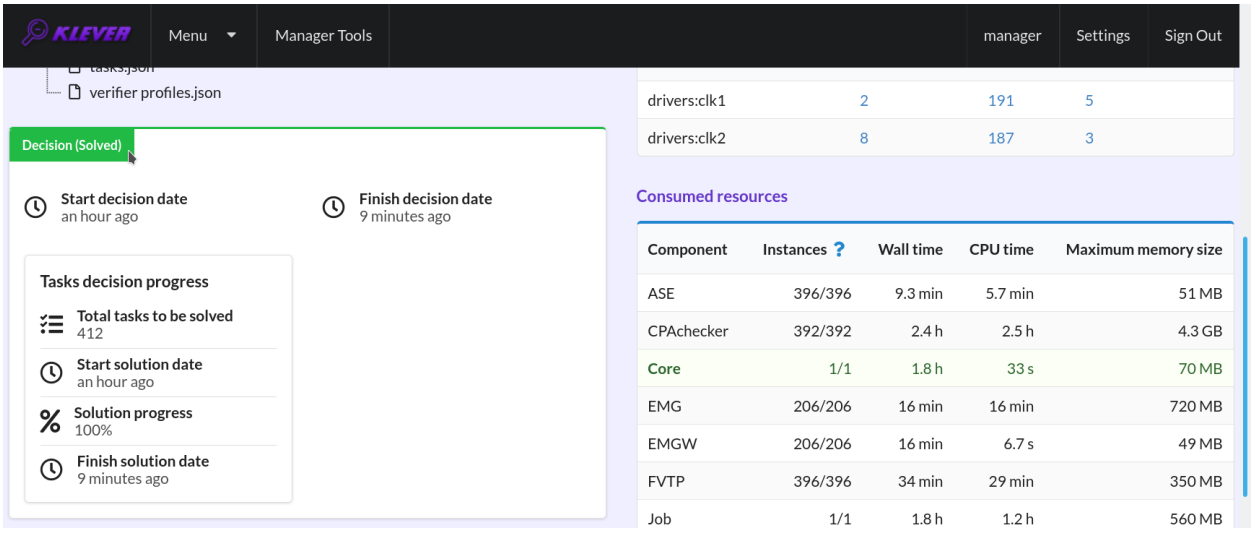


Fig. 1.9: The completed decision of the job version

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).

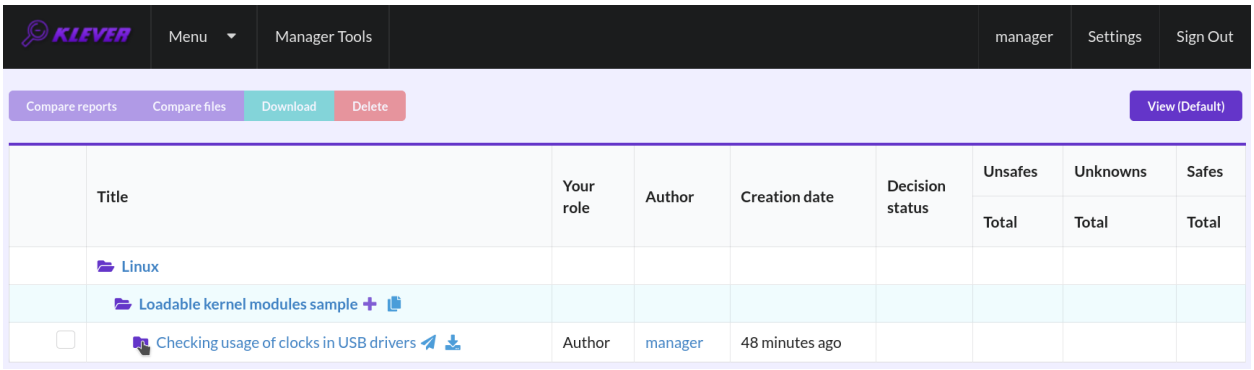


Fig. 1.10: Showing job versions

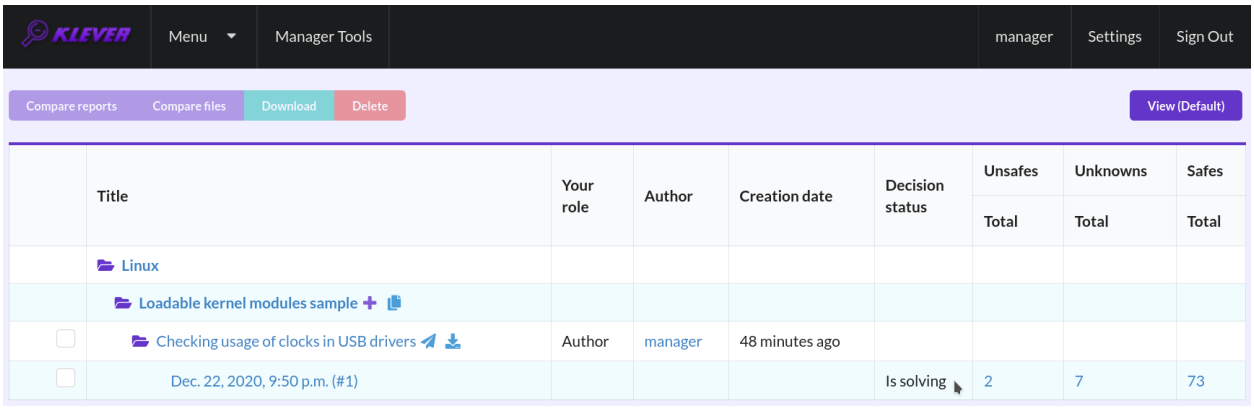


Fig. 1.11: The status of the decision of the job version

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 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.

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).

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

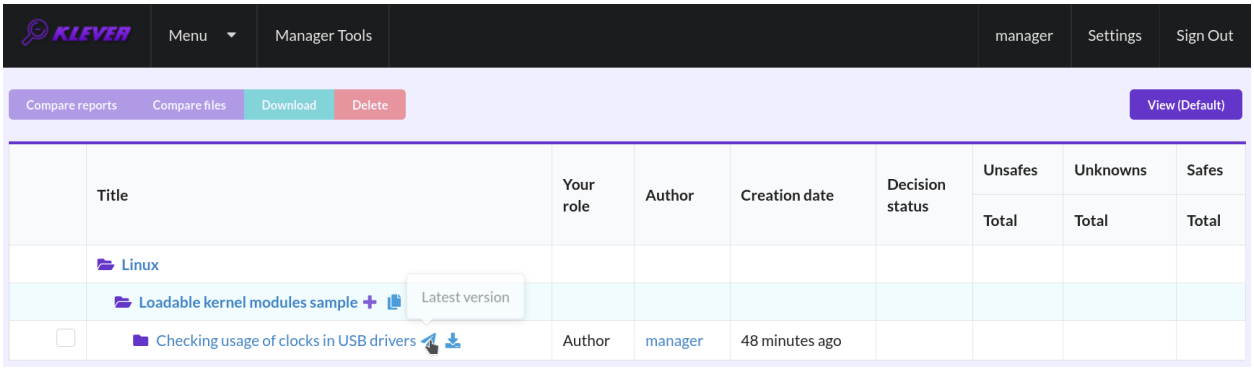


Fig. 1.12: Opening the page with the decision of the latest job version

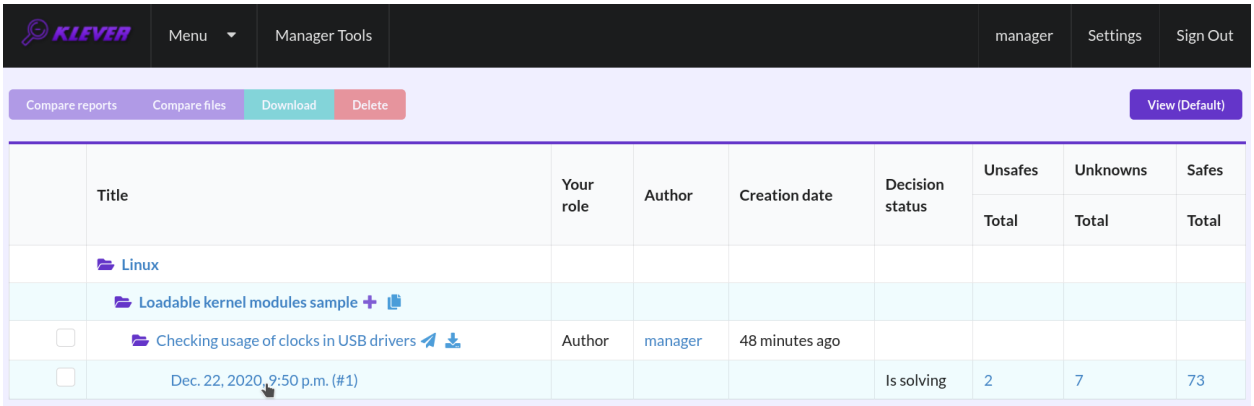


Fig. 1.13: Opening the page with the decision of the particular job version

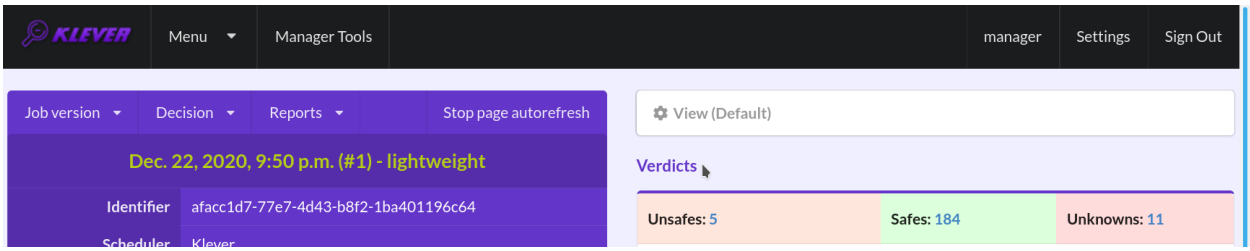


Fig. 1.14: Verdicts

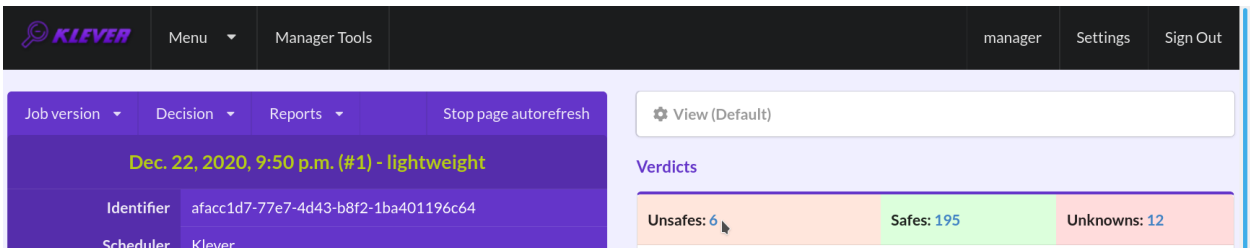


Fig. 1.15: The total number of unsafes reported thus far

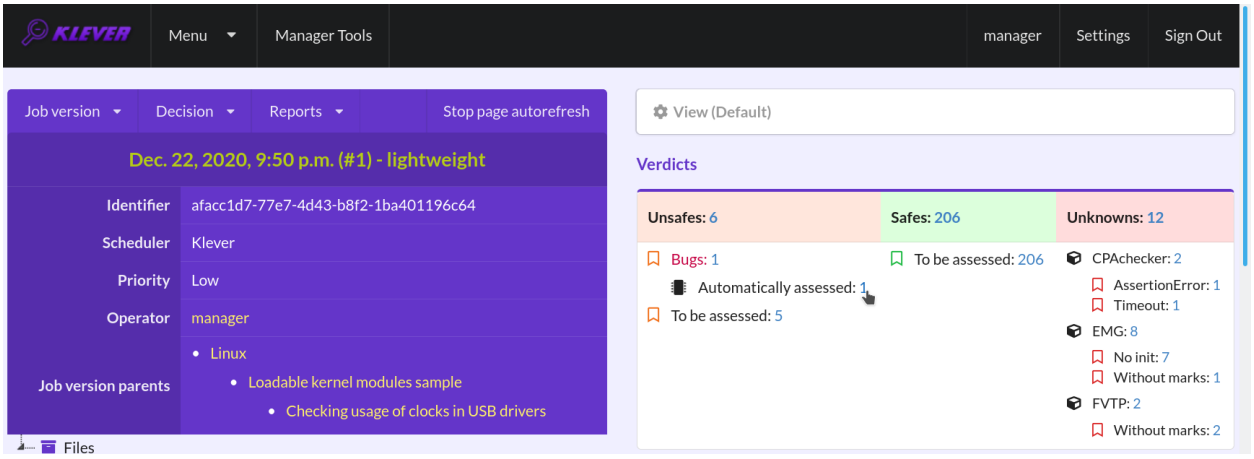


Fig. 1.16: The total number of automatically assessed unsafes

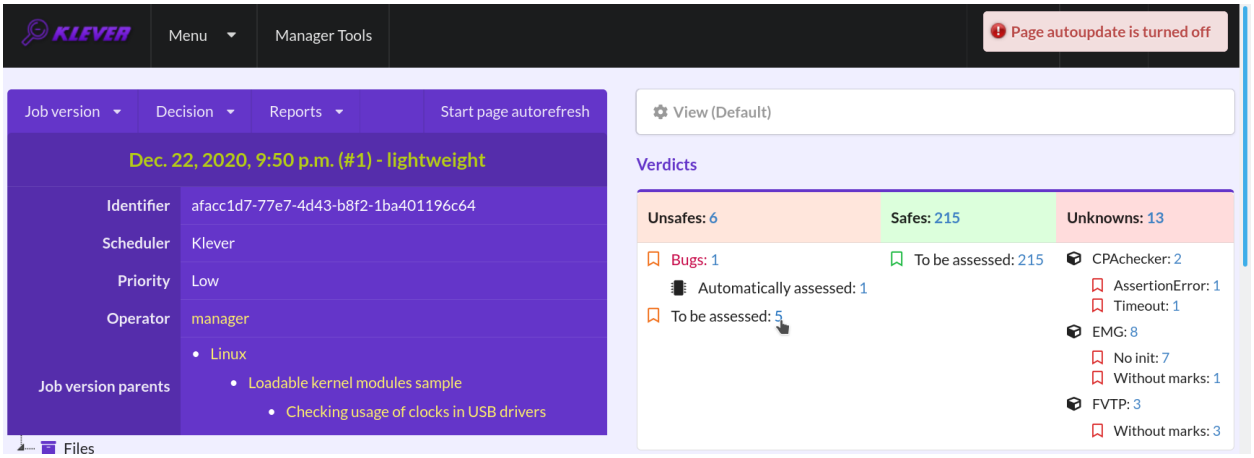


Fig. 1.17: The total number of unsafes without any assessment

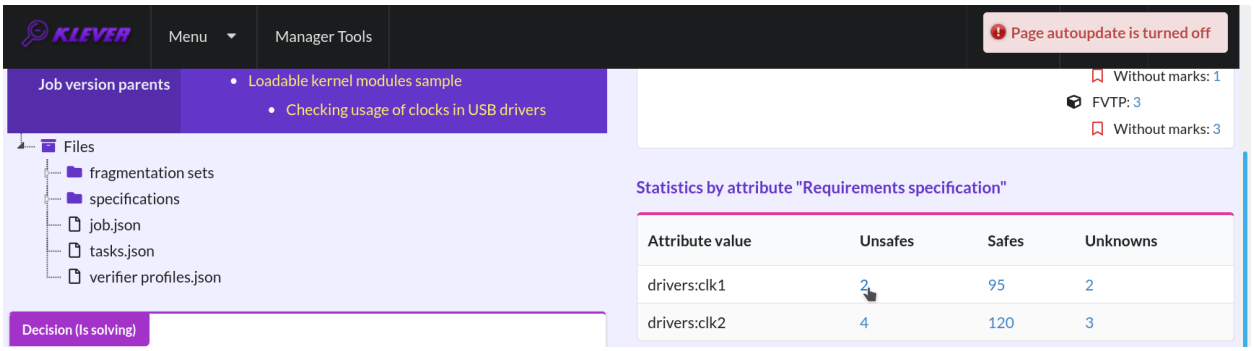
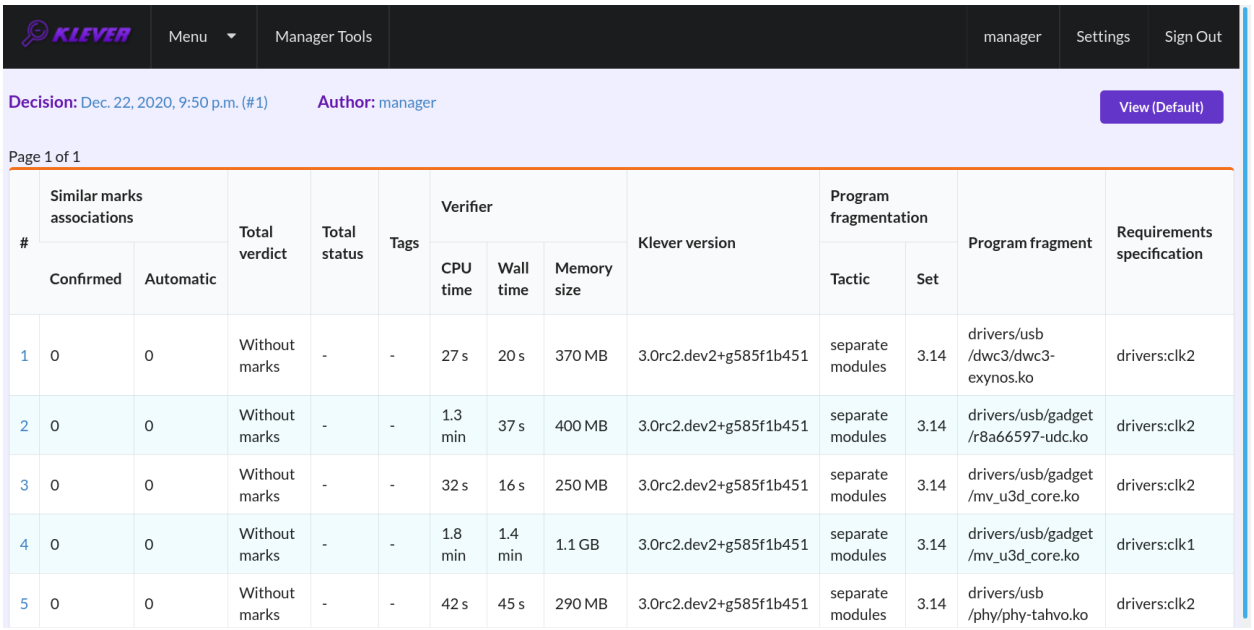


Fig. 1.18: The total number of unsafes corresponding to the particular requirements specification

analysis we recommend clicking on an unsafe index on the left to open a new page in a separate tab (Fig. 1.20). To 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.



#	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

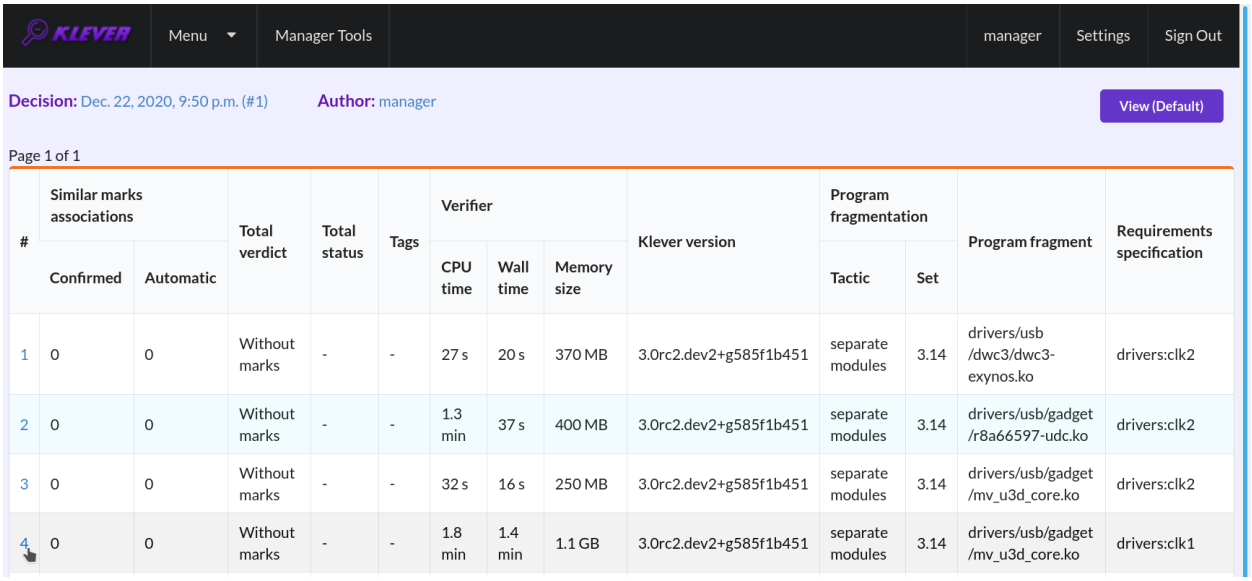
After clicking on links within the list of unsafes like in Fig. 1.20, you will see corresponding error traces. For instance, Fig. 1.22 demonstrates an error trace example for module *drivers/usb/gadget/mv_u3d_core.ko* and requirements specification *drivers:clk1*.

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



#	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

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

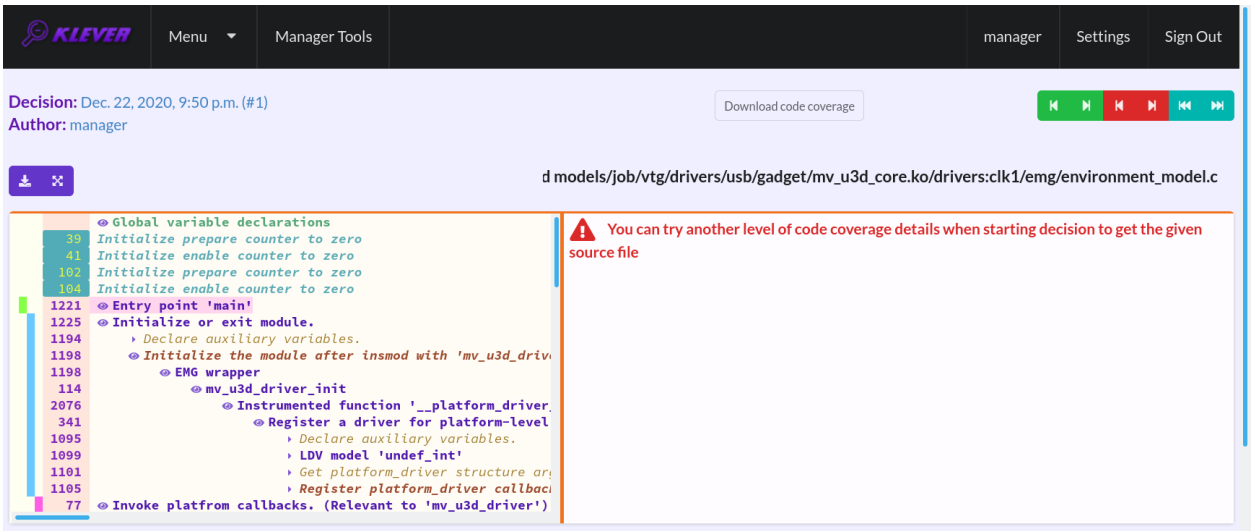


Fig. 1.22: The error trace for module drivers/usb/gadget/mv_u3d_core.ko and requirements specification drivers:clk1

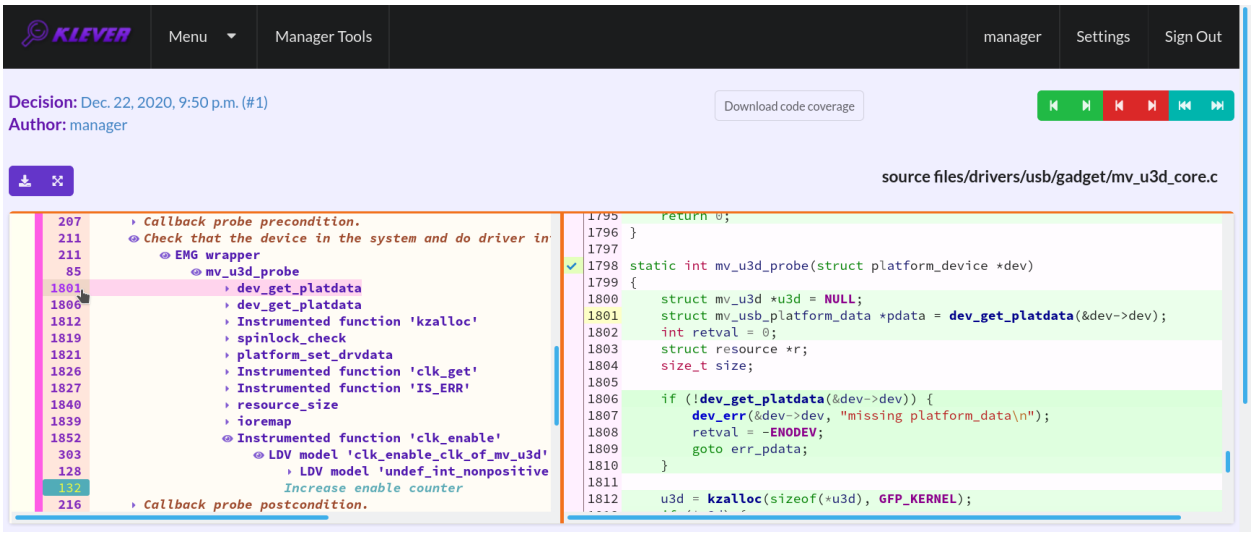


Fig. 1.23: Showing the line in the original source file corresponding to the error trace statement

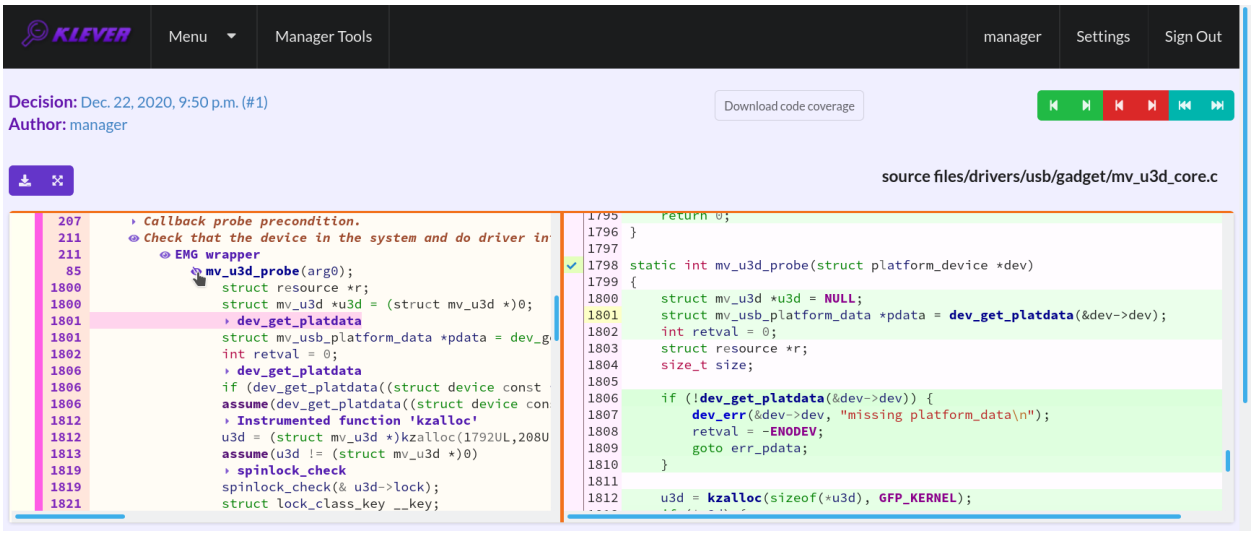


Fig. 1.24: Showing hidden declarations, statements and assumptions for functions with notes or warnings

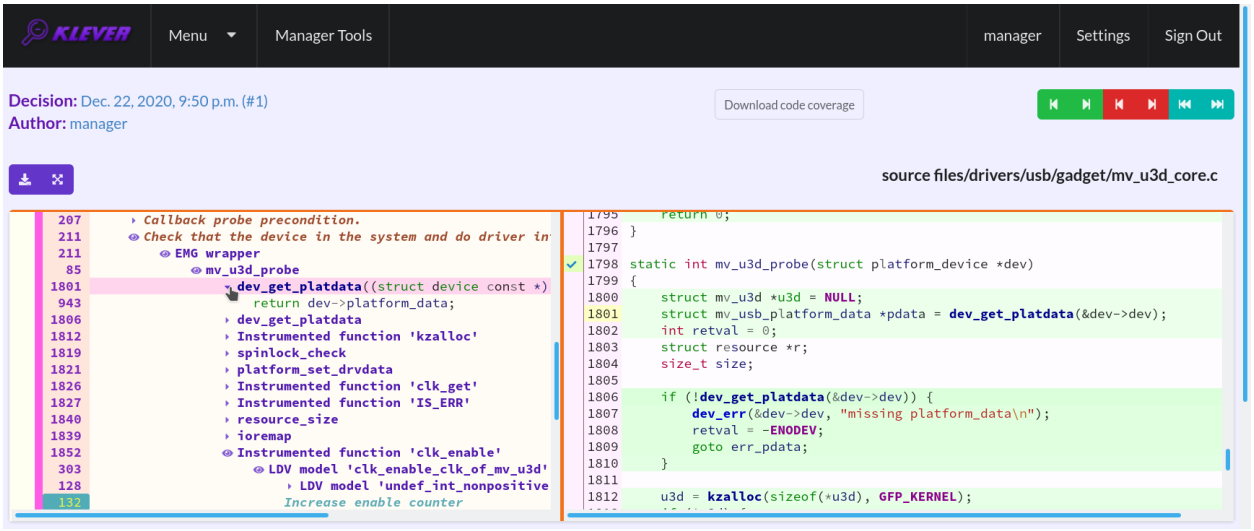


Fig. 1.25: Showing hidden declarations, statements and assumptions for functions without notes or warnings

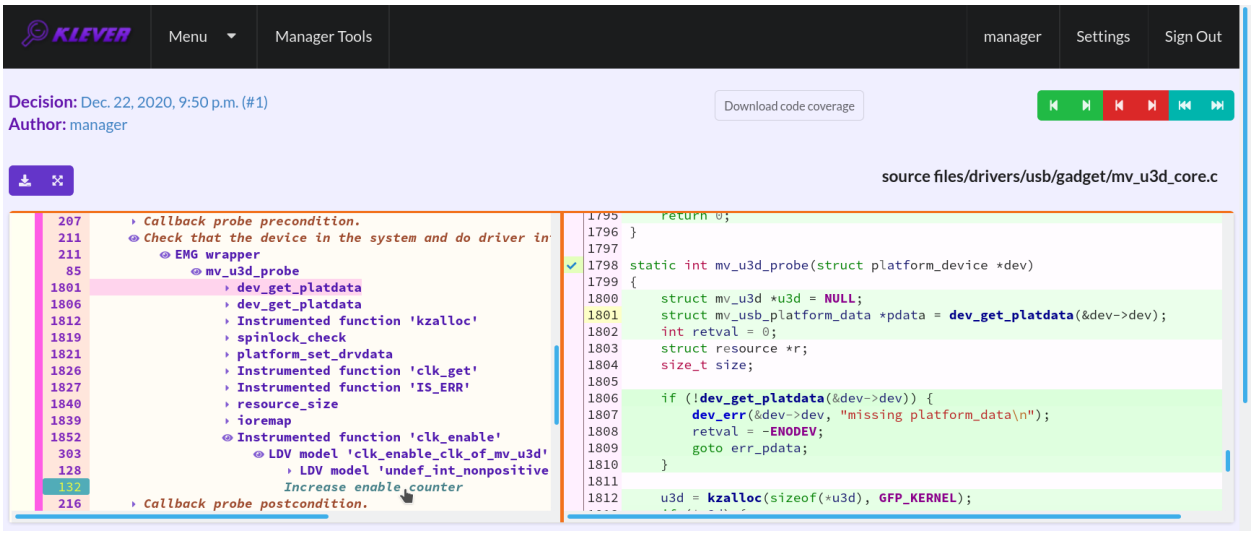


Fig. 1.26: The error trace note

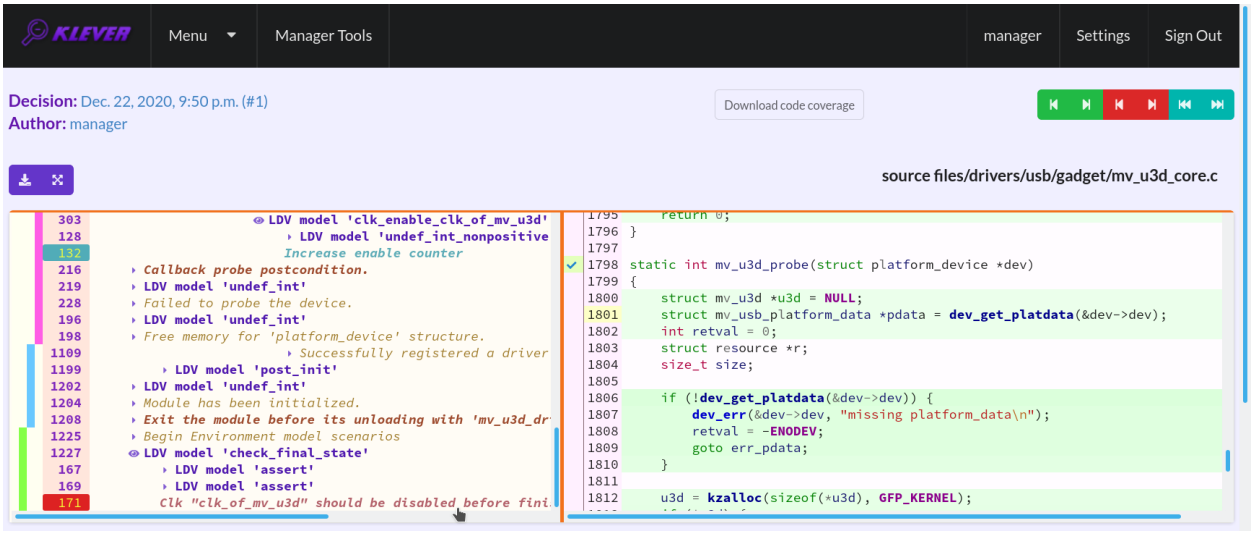


Fig. 1.27: The error trace warning

stuff in the error trace. These are parts of the environment model necessary to initialize models, to invoke module 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).

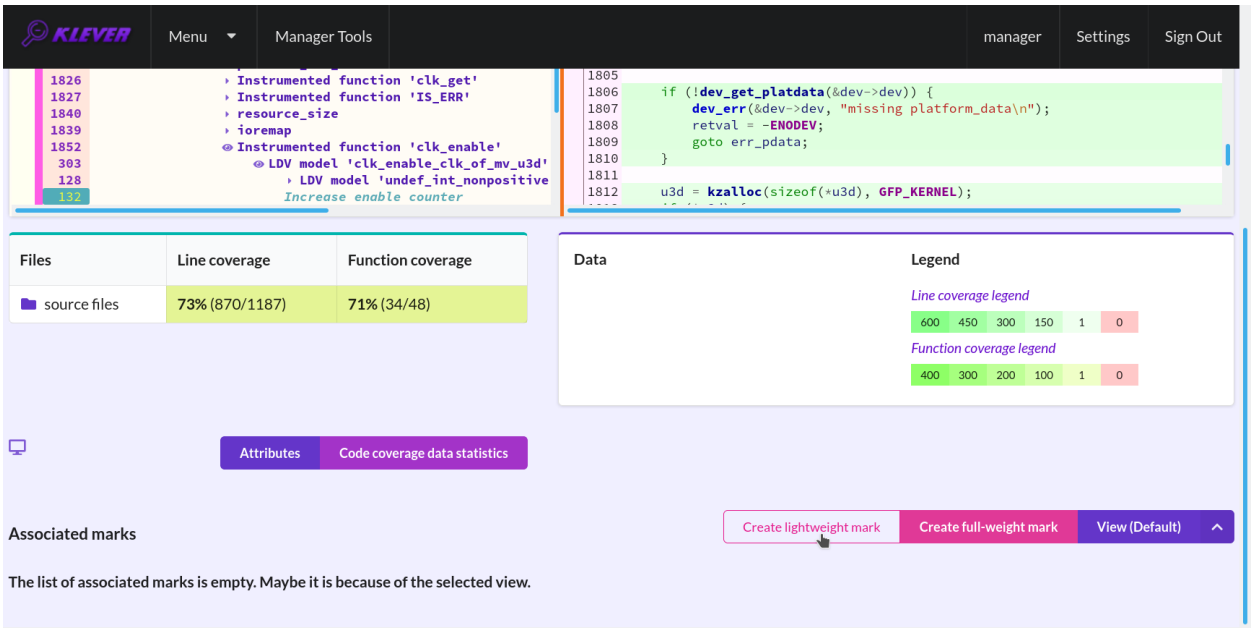


Fig. 1.28: Starting the creation of a new lightweight mark

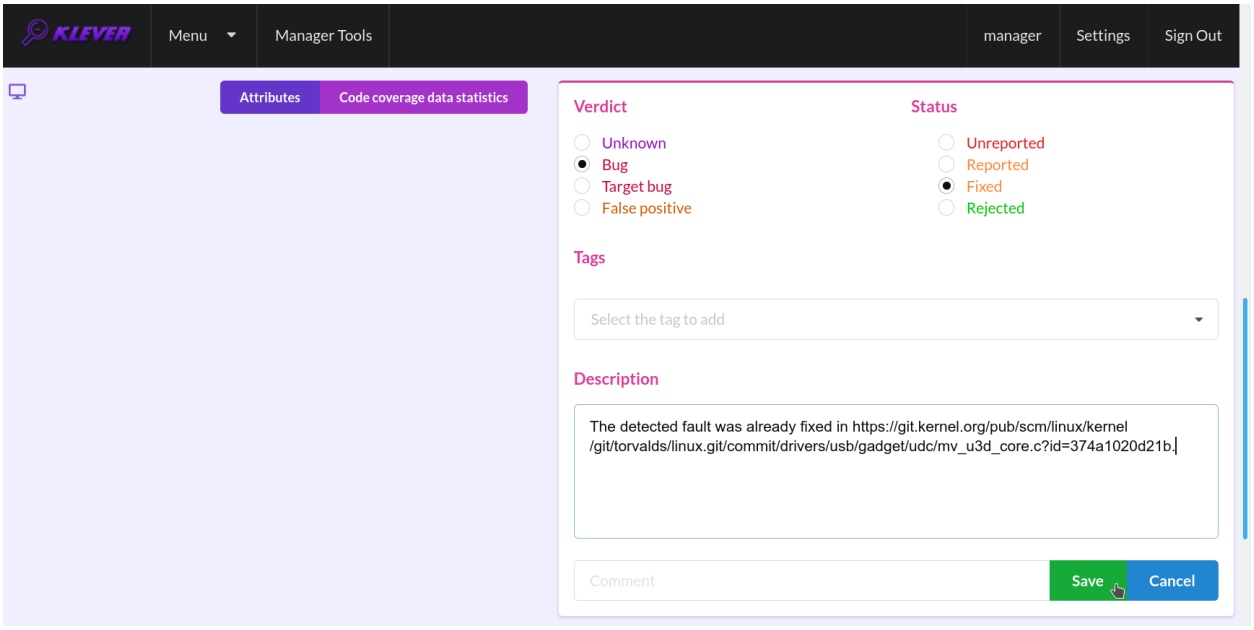


Fig. 1.29: The creation of the new lightweight mark

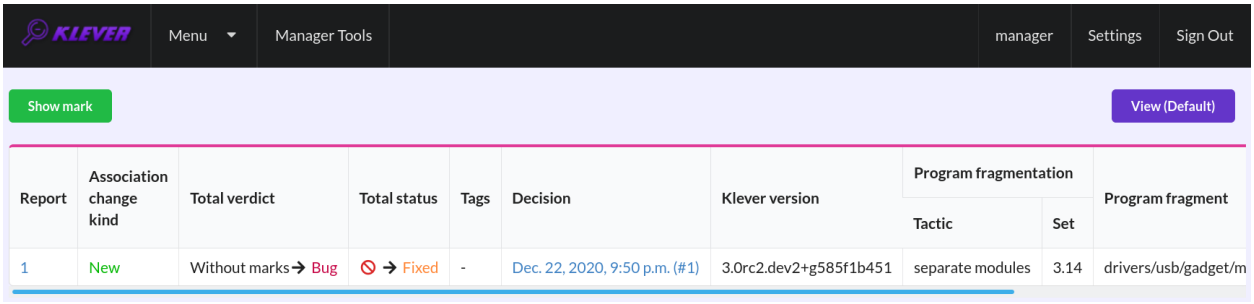


Fig. 1.30: Changes in total verdicts

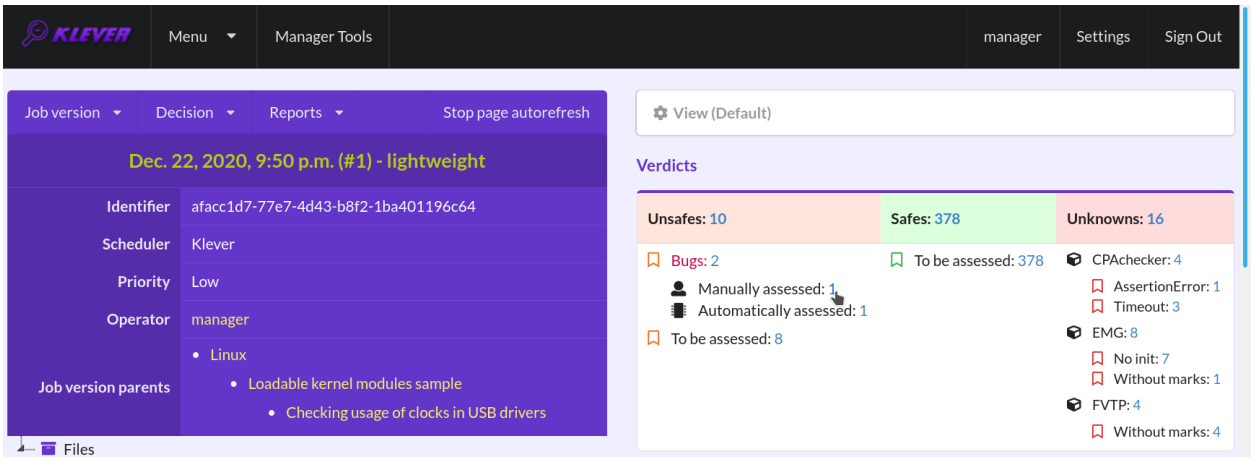


Fig. 1.31: The total number of manually assessed unsafes

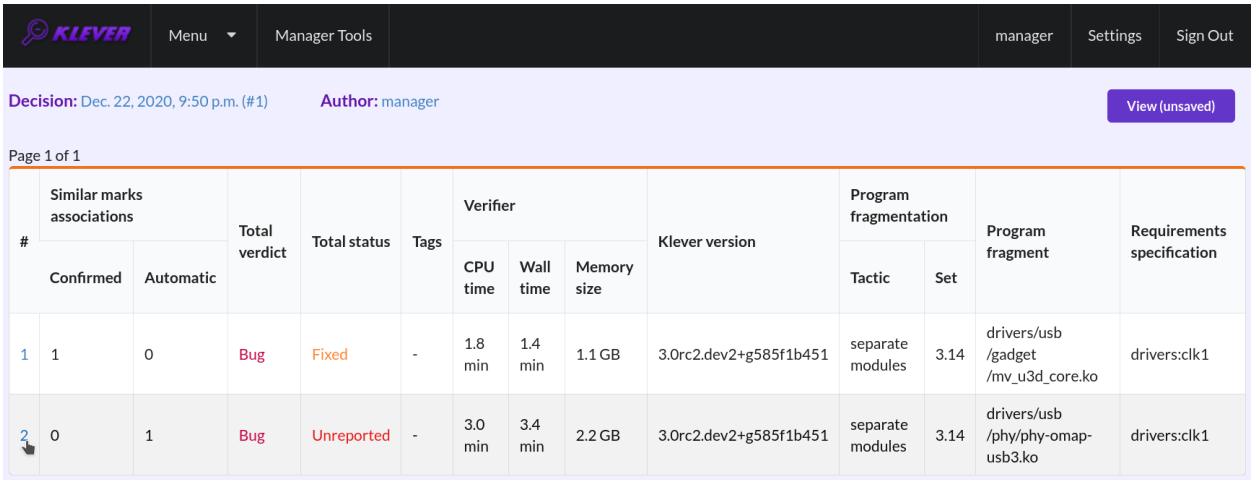


Fig. 1.32: Opening the error trace of the unsafe with automatic assessment

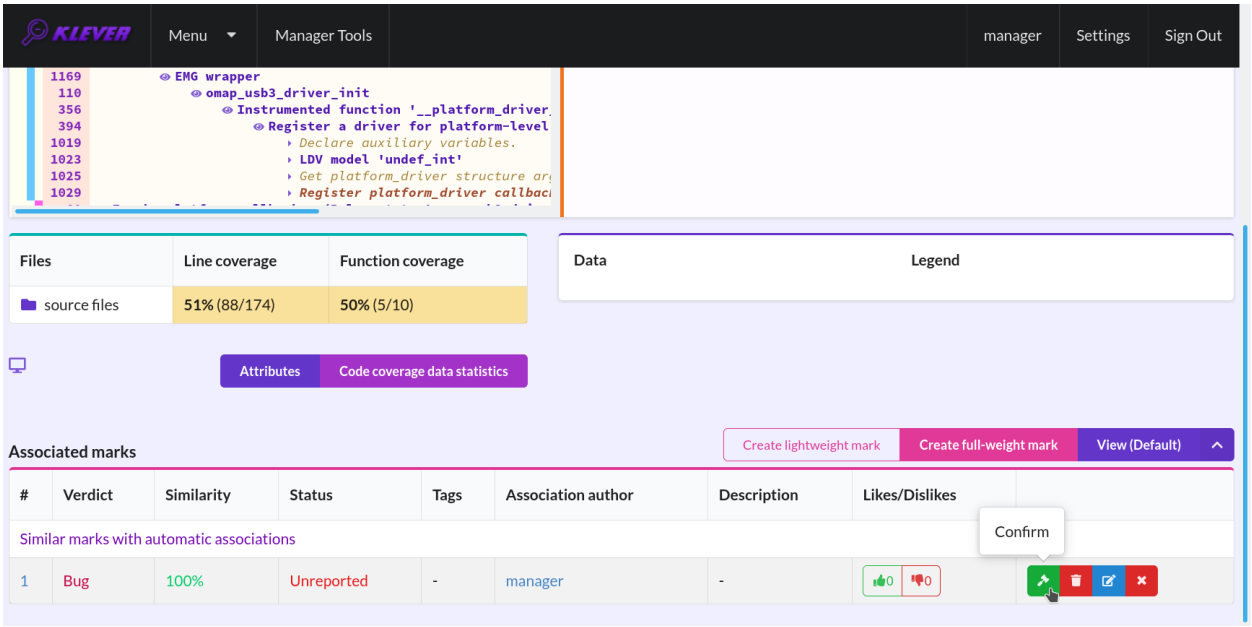


Fig. 1.33: Confirming the automatic association

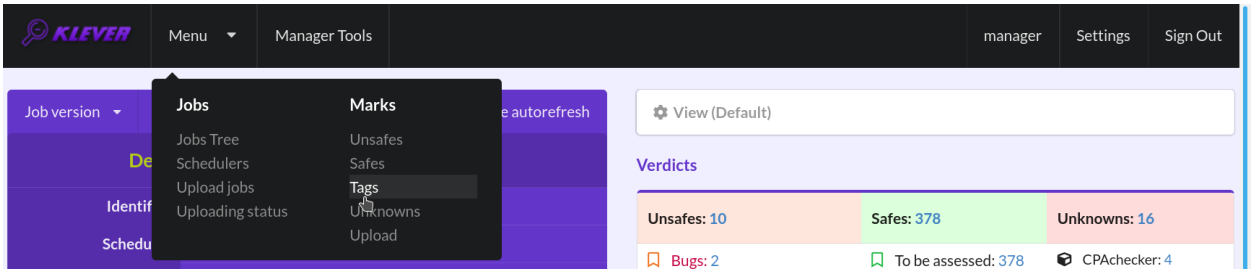


Fig. 1.34: Opening the tags page

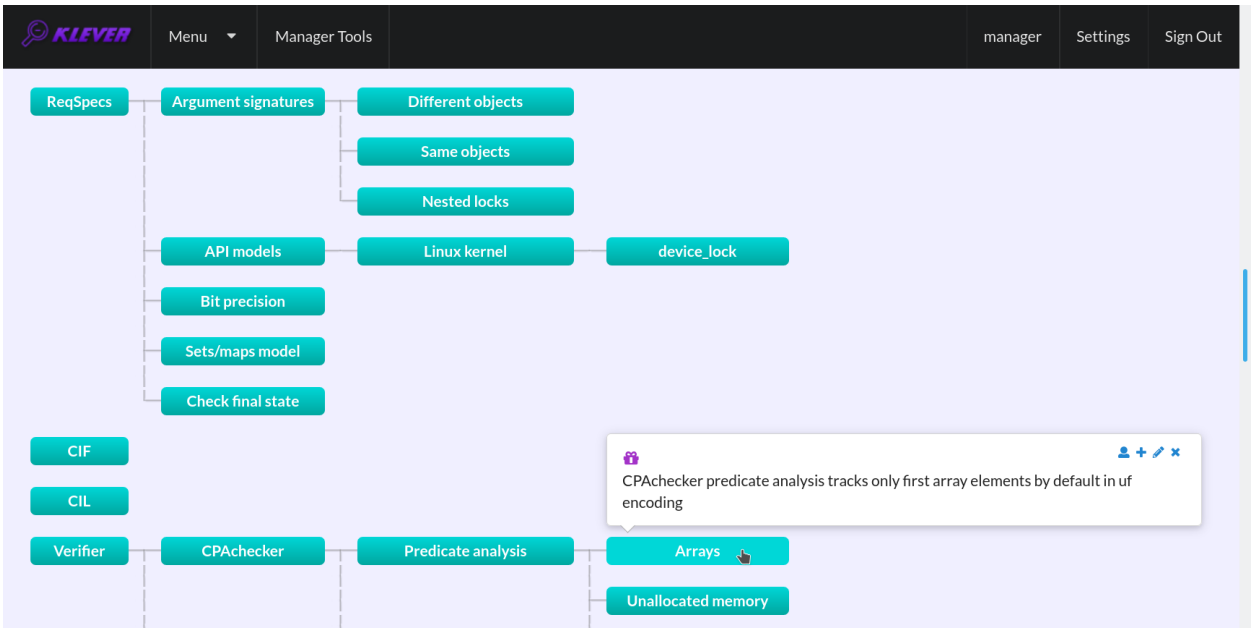


Fig. 1.35: Showing tag description

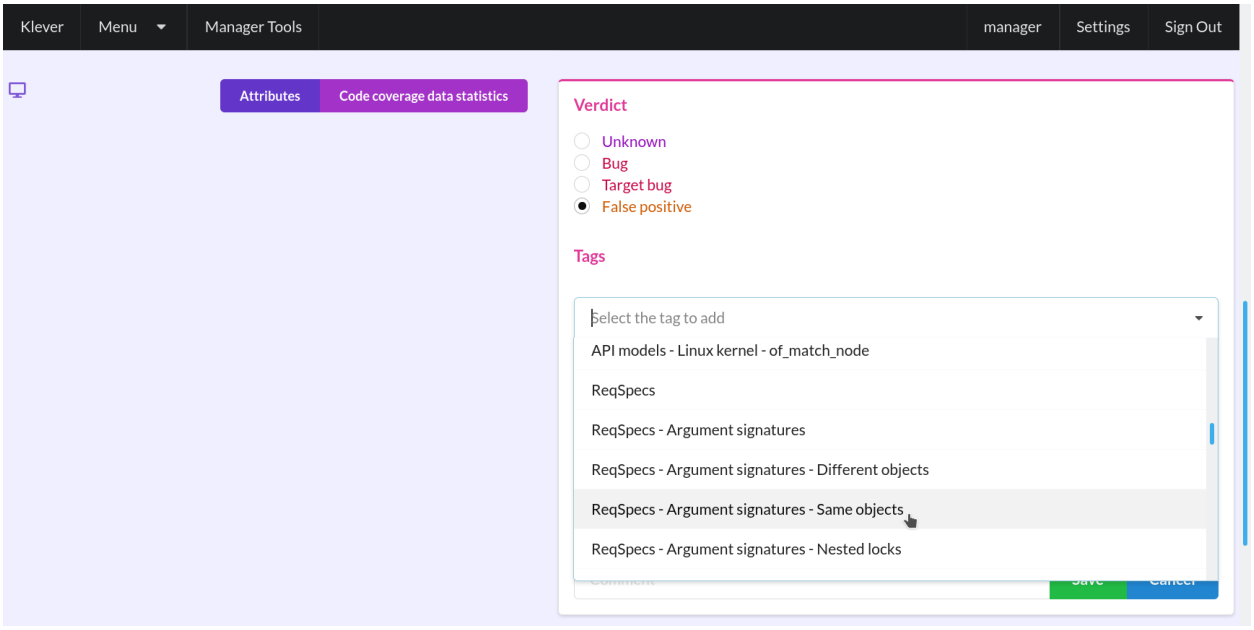


Fig. 1.36: Choosing tag from the dropdown list

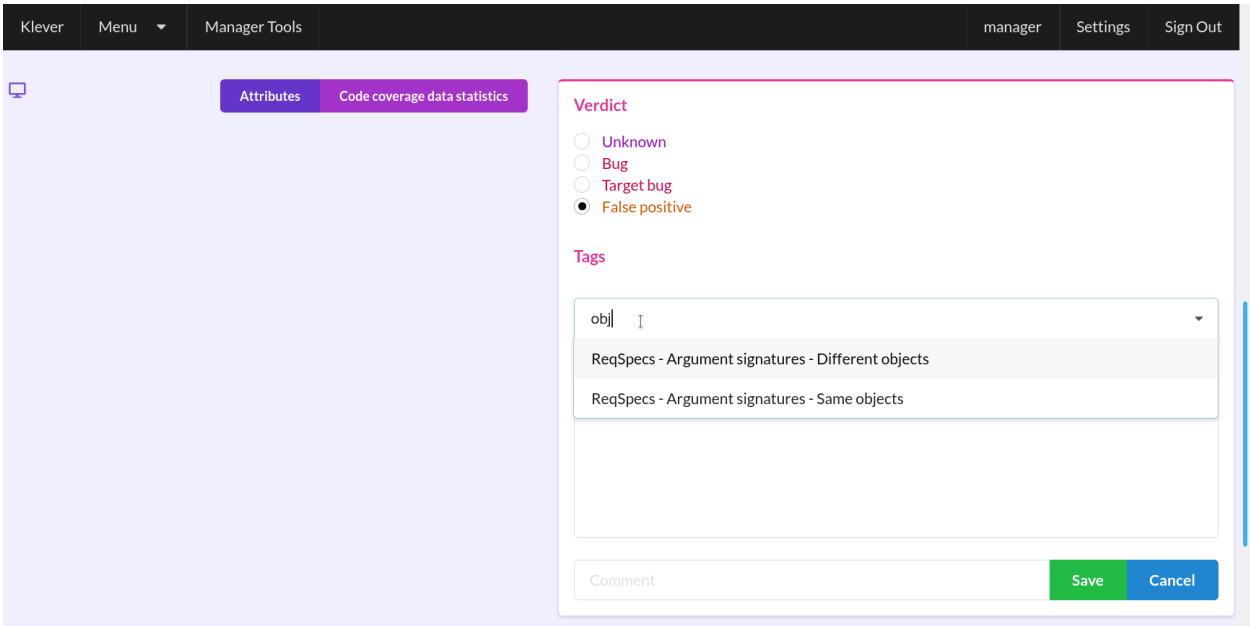


Fig. 1.37: Entering tag name part

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 Developer Documentation

1.3.1 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.

1.3.2 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
```

1.3.3 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.

1.3.4 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.

1.3.5 Deployment for Development Purposes

To deploy Klever for development purposes in addition to using mode *development* (see *Local Deployment*) one needs to specify command-line option *–allow-symbolic-links*.

1.3.6 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 for Development Purposes*).
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. Make CIF executables to be available through the PATH environment variable, e.g.:

   ```
   $ export PATH=$KLEVER_DEPLOY_DIR/klever-addons/CIF/bin/:$PATH
   ```

   where the KLEVER_DEPLOY_DIR environment variable is explained at *$KLEVER_DEPLOY_DIR*.

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

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

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

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

8. 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.

9. 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 --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!)

1.3.7 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.

1.3.8 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.

1.3.9 Extended Violation Witness Format

TODO: Translate from Russian.

1.3.10 Error Trace Format

TODO: Translate from Russian.

1.3.11 Code Coverage Format

TODO: Translate from Russian.

1.4 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.

Index

Symbols

$INSTANCES, **35**
$KLEVER_DEPLOY_DIR, **35**
$KLEVER_SRC, **35**
$OS_USERNAME, **35**
$SSH_RSA_PRIVATE_KEY_FILE, **35**

E

Environment model, **35**
environment variable
    PATH, 34

P

PATH, 34