Software
Systems
Engineering

Stiftung Universität Hildesheim
2003

User-Documentation KernelHaven

Arbeitsgruppe Software Systems Engineering • Institute for Information Technology
Universität Hildesheim • Universitätsplatz 1 • D-31141 Hildesheim

Contents

List of Figures

List of Tables

1 Setup of an Execution Environment

KernelHaven was developed and tested on Ubuntu 16.04 LTS. Assuming a clean setup, some packages need to be installed to allow the execution of the kernelhaven-infrastructure and plugins.

The required installation commands for the installation of the dependencies are listed in Table 1.1.

Table 1.1: Installation of dependencies

Component	**Command for Installation of Dependencies**
`kernelhaven`	`$ apt-get install openjdk-8-jdk`
`kconfigreader`	`$ apt-get install build-essential libelf-dev bc`
`kbuildminer`	`-`
`typechefextractor`	`-`
`undertakerextractor`	`-`

The user only needs to install the packages that are required by the plugins that the user executes in his analysis.

2 Execution of an existing Analysis with KernelHaven

Extract the entire kernelhaven-binary release to a folder on your machine.

Open a terminal and change the path to the folder which contains `kernelhaven.jar`.

Assuming an existing properties file, the analysis can be started using:

```
$ java -jar kernelhaven.jar target_analysis.properties
```

The first parameter of the console input after `kernelhaven.jar` represents the location of the properties file (in this instance `target_analysis.properties`). The properties file defines all of the relevant parameters needed for the analysis. The content of such a properties file is explained in the next chapter. An overview of all parameters can be found in section 3.2.

Additionally you may choose to archive the tool and its configuration including all artifacts and results for this execution by adding the archive-flag:

```
$ java -jar kernelhaven.jar target_analysis.properties --archive
```

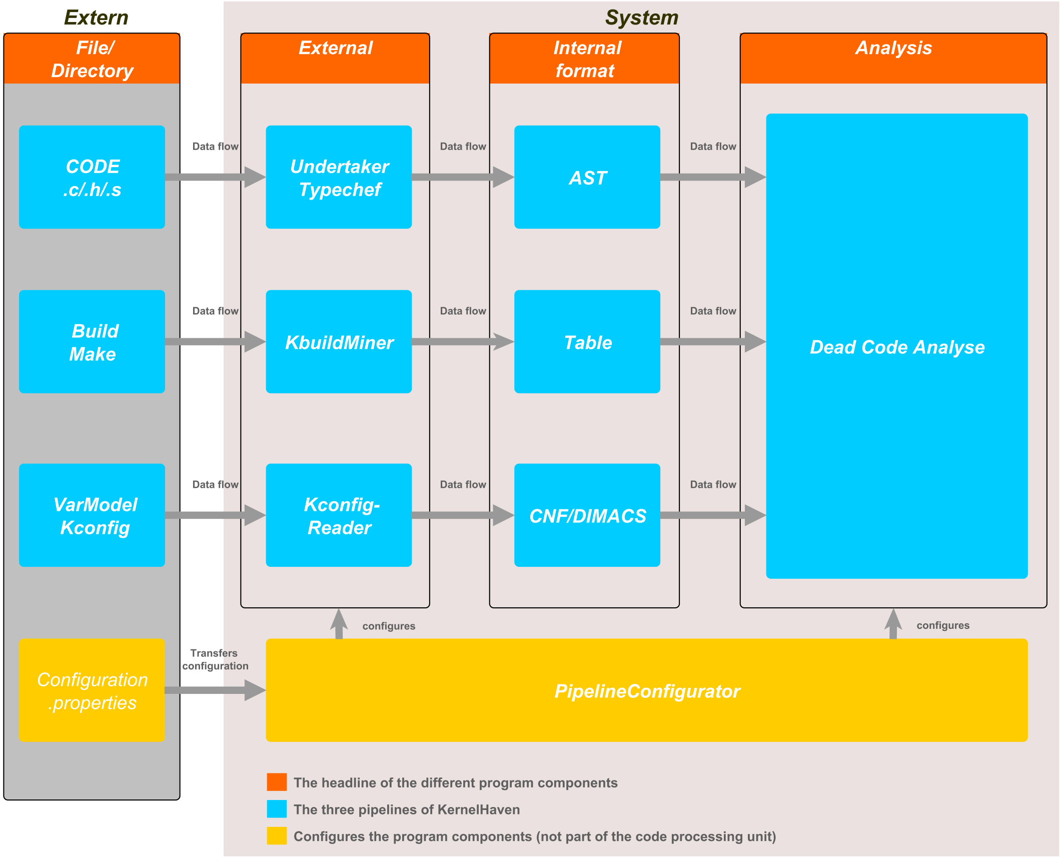
3 Definition of an Analysis

3.1 Understanding the Concept of KernelHaven

KernelHaven offers a generic infrastructure for perfoming different analyses on product lines. The infrastructure is shown in an overview in Figure 3.1. It can support different product lines as it allows the use of plugins. Plugins can be analysis-components or extractors. The extractors can be used in the existing pipelines processing Code, BuildModels or VariabilityModels.

The targets of the analysis as well as other parameters required for its execution can be configured through a properties file. The setup of such a file is explained in the following section.

Figure 3.1: Architecture of KernelHaven



3.2 Configuration through a Properties-File

The Properties-File follows the generic pattern of a java Properties-file. The properties described on the following pages are also included in the file `config_template.properties` that is included in every release of KernelHaven. The collection of parameters only includes parameters that are integrated into the KernelHaven infrastructure. Any plugin might require more parameters for its execution. Please consult the documentation of the according plugin for more information (See chapter 4).

Some parameters include a `default:` value while others contain a `example:` value. The reason for this is that not all parameters have default values and while default-values also provide a good example of a valid value, an explicit example was needed for those parameters.

The description following `mandatory:` reflects whether a parameter is optional or not. "yes" means that the parameter is required in every configuration. "no" without any additional information means that the parameter is optional in every configuration. There are some parameters with additional information for `mandatory` that explains under which circumstances the parameter becomes mandatory.

All parameters that have `default`-values are optional.

For boolean properties: Anything but "true" is read as "false"; the "true" is not case sensitive. If a boolean setting is missing and no default value is specified, then it is read as false.

Table 3.1: General Parameters

Parameter	**Description**
`resource_dir` example: /some/folder/tmp mandatory: yes	The path where extractors can store their resources. The extractors create sub-folders in this called the same as their fully qualified class names (to prevent conflicts). This has to be always set to a valid directory with write and read access.
`output_dir` example: /some/folder/output mandatory: yes	The path where the output files of the analysis will be stored. This has to be always set to a valid directory with write access.
`plugins_dir` example: /some/folder/plugins mandatory: yes	The directory where plugins should be placed. All *.jar files in that directory will be loaded into the JVM. This means that 3rd party libraries which are required by plugins can also be included through this directory. This has to be always set to a valid directory with read access.
`cache_dir` example: /some/folder/dir mandatory: no	This is the directory where the providers will write and read their cache. If `cache.read` or `cache.write` of a provider is set to true, then this has to be set to a valid directory with write and read access.
`log.dir` example: /some/folder/log mandatory: no	This is the path where log files will be written. Needs to be defined if `log.file` is true. If required this has to be set to a valid directory with write access.
`log.console` default: true mandatory: no	If set to true all logging sequences will be pushed to console, excluding huge third party outputs which will be trimmed to 500 characters. If a detailed output is needed refer to logging.file for a complete output.
`log.file` default: false	If set to true all logging sequences will be pushed to a log file in the directory defined in `log.dir` setting.
`log.error` default: true	If set to false error log messages will be excluded from logging.
`log.warning` default: true	If set to false warning log messages will be excluded from logging.
`log.info` default: true	If set to false info log messages will be excluded from logging.
`log.debug` default: false	If set to false debug log messages will be excluded from logging.
`archive` default: false	Archives the the tool and its configuration including all artifacts and results. Alternative to setting the −−`archive` flag on terminal.
`archive.dir` example: /some/dir	Directory to write the archive of the current configuration and results to.

Table 3.2: Analysis Parameters

Parameter	Description
`analysis.class` example: some.package.ClassName mandatory: yes	The fully qualified class name of the analysis that should be run.

Table 3.3: Common Extractor Parameters

Parameter	Description
`source_tree` example: /some/source/tree mandatory: depends on extractors	The path to the source tree that should be analyzed.
`arch` example: x86 mandatory: depends on extractors	The architecture of the Linux kernel that should be analyzed.

Table 3.4: Parameters for Code Model

Parameter	Description
`code.provider.timeout` default: 0	The maximum time the provider waits for the results of the extractor until an exception is thrown in milliseconds, 0 means no timeout used.
`code.provider.cache.write` default: false mandatory: no	If set to true, then the code model provider will write its results to the cache directory.
`code.provider.cache.read` default: false mandatory: no	If set to true, then the code model provider is allowed to read the cache instead of starting the extractor.
`code.extractor.class` example: some.package.ClassName mandatory: no, but may be required by analysis	The fully qualified class name of the extractor for the code model.
`code.extractor.files` example: file1.c, dir/file2.c, dir/subdir/ default: empty	Defines which files the code extractor should run on. Comma separated list of paths relative to the source tree. If directories are listed, then they are searched recursively for files that match the `code.extractor.file_regex` pattern. Leave empty to specify the complete source tree.
`code.extractor.file_regex` example: .*\.(h|c|S) default: .*\.c	A Java regular expression defining which files are considered to be source files for parsing.
`code.extractor.threads` default: 1	The number of threads the code extractor should use. This many files are parsed in parallel.

Table 3.5: Parameters for Build Model

Parameter	**Description**
`build.provider.timeout` default: 0 mandatory: no	The maximum time the provider waits for the results of the extractor until an exception is thrown in milliseconds. 0 means no timeout is used.
`build.provider.cache.write` default: false mandatory: no	If set to true, then the build model provider will write its result to the cache directory.
`build.provider.cache.read` default: false mandatory: no	If set to true, then the build model provider is allowed to read the cache instead of starting the extractor.
`build.extractor.class` example: some.package.ClassName mandatory: no, but may be required by analysis	The fully qualified class name of the extractor for the build model.

Table 3.6: Parameters for Variability Model

Parameter	**Description**
`variability.provider.timeout` default: 0 mandatory: no	The maximum time the provider waits for the results of the extractor until an exception is thrown in milliseconds. 0 means no timeout is used.
`variability.provider.cache.write` default: false mandatory: no	If set to true, then the variability model provider will write its result to the cache directory.
`variability.provider.cache.read` default: false mandatory: no	If set to true, then the variability model provider is allowed to read the cache instead of starting the extractor.
`variability.extractor.class` example: some.package.ClassName mandatory: no, but may be required by analysis	The fully qualified class name of the extractor for the variability model.
`variability.extractor.find_locations` default: false	If set to true, the extractor will store source locations for each variable. Those locations represent occurrences of the variable in the files that kconfigreader used for generating the VariabilityMode.

4 KernelHaven Plugins

4.1 DeadCodeAnalysis (Part of DefaultAnalyses)

Type: Analyis-Plugin

Class: `de.uni_hildesheim.sse.kernel_haven.default_analyses.DeadCodeAnalysis`

License: KernelHaven-License

Prerequisites:

Needs the three extractors for the variability model, build model and code model. Depends on CnfUtils.

Capabilities:

This is a simple implementation to detect dead code blocks. It considers file presence conditions and ifdef blocks.

Additional Parameters:

None.

4.2 Missing Analysis (Part of DefaultAnalyses)

Type: Analyis-Plugin

Class: `de.uni_hildesheim.sse.kernel_haven.dummy_analysis.MissingAnalysis`

License: KernelHaven-License

Prerequisites:

Needs the three extractors for the variability model, build model and code model.

Capabilities:

This analysis uses the variability model and returns a file of all variables which are defined in the variability model but not used in the code model or build model, or a file of all variables which are used in the code model or build model but not defined in the variability model.

Additional Parameters:

Parameter	**Description**
`analysis.missing.type` default: D	This parameter is for choosing the missing analysis. The parameter D is for the 'defined but not used' analysis. The parameter U is for the 'used but not defined' analysis. This is not case sensitive.

4.3 KBuildMinerExtractor

Type: BuildModel-Extractor

Class: `de.uni_hildesheim.sse.kernel_haven.kbuildminer.KbuildMinerExtractorFactory`

License: GPL-3.0

KernelHaven-License would be possible with following restrictions:

The extractor contains kbuildminer.jar which is under GPL-3.0. We do not link against kbuildminer, so technically we are not infected by GPL. However a release under a license other than GPL-3.0 would require the removal of the contained kbuildminer.jar.

Prerequisites:

None.

Capabilities:

This extractor finds presence conditions of source files defined in Kbuild.

Additional Parameters:

Parameter	Description
`build.extractor.top_folders` example: kernel,drivers,arch/x86 mandatory: no	List of top folders to analyze in the product line. If not supplied, then a default set for Linux is generated from the `arch` setting.

4.4 KConfigReaderExtractor

Type: VariabilityModel-Extractor

Class: `de.uni_hildesheim.sse.kernel_haven.kconfigreader.KconfigReaderExtractorFactory`

License: GPL-3.0

KernelHaven-License would be possible with following restrictions:

The extractor contains kconfigreader.jar which is under GPL-3.0. We do not link against kconfigreader, so technically we are not infected by GPL. However a release under a license other than GPL-3.0 would require the removal of the contained kconfigreader.jar.

Prerequisites:

This extractor can only run on a Linux operating system. It also requires make and gcc (`$ apt-get install build-essential libelf-dev bc`).

Capabilities:

This extractor reads the Kconfig model. To do that, it has to modify the Linux source tree by calling `make allyesconfig prepare` on it. Be aware that this overrides any previously present `.config` file in the Linux source tree.

Additional Parameters:

None.

4.5 UndertakerExtractor

Type: CodeModel-Extractor

Class: `de.uni_hildesheim.sse.kernel_haven.undertaker.UndertakerExtractorFactory`

License:

KernelHaven-License would be possible with following restrictions:

The extractor contains undertaker which is under GPL-3.0. We do not link against undertaker, so technically we are not infected by GPL. However a release under a license other than GPL-3.0 would require the removal of the contained undertaker.

Prerequisites:

This extractor can only run on a Linux operating system.

Capabilities:

This extractor finds `#ifdef` blocks in source files.

`ExpressionFormatException` is a common exception thrown by this extractor. This is the expected behaviour because ifdef-Blocks often contain non-boolean expressions while we only work with boolean expressions. This needs to be handled in any analysis using this extractor.

Additional Parameters:

Parameter	**Description**
`code.extractor.hang_timeout` default: 20000	Undertaker has a bug where it hangs forever on some few files of the Linux kernel. This setting defines a timeout in milliseconds until the undertaker executable is forcibly terminated.

5 Examples

5.1 DummyAnalysis

The execution environment in this example is a machine running Ubuntu 16.04 with all of the packages listed in chapter 1 installed.

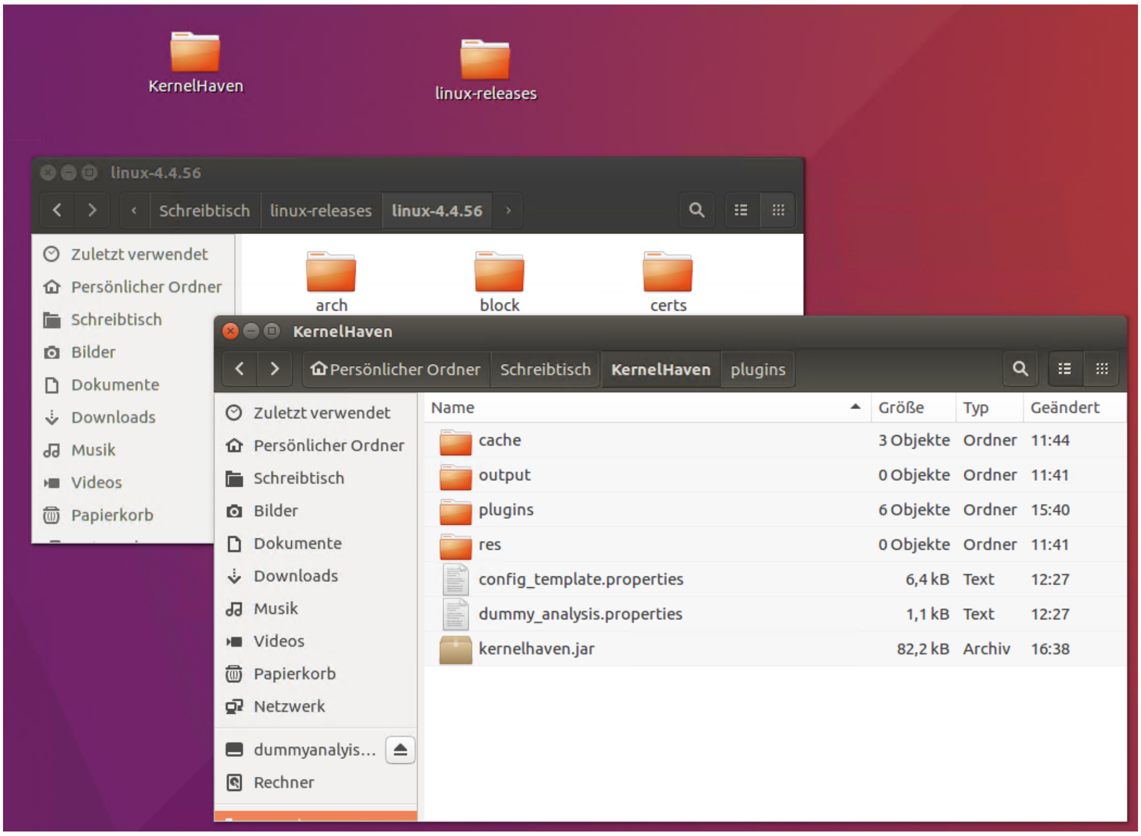
First, the entire contents of the binary-release of KernelHaven need to be extracted. In our case, we choose to extract the contents of the zip-archive (binary-release) to a directory our desktop. You may however choose any other directory with read/write access to reproduce this on your own machine.

Now `kernelhaven.jar`, `dummy_analysis.properties`, `config_template.properties` and the folder `plugins` with all of the plugins contained in the release are present in the directory that we just created.

Additionally, we create the folders "cache", "output" and "res" inside of the same directory that already contains `kernelhaven.jar`.

On the desktop we have a folder `linux-releases` containing the linux kernel linux-4.4.56 as shown in Figure 5.1. You can use any other version of the linux kernel but will have to adjust the parameter `source_tree` in `dummy_analysis.properties` accordingly.

Figure 5.1: Folder Strucure



Name	Größe	Typ	Geändert
cache	3 Objekte	Ordner	11:44
output	0 Objekte	Ordner	11:41
plugins	6 Objekte	Ordner	15:40
res	0 Objekte	Ordner	11:41
config_template.properties	6,4 kB	Text	12:27
dummy_analysis.properties	1,1 kB	Text	12:27
kernelhaven.jar	82,2 kB	Archiv	16:38

We will use the configuration `dummy_analysis.properties` that is contained in every release. This configuration can be used to confirm that the KernelHaven-infrastucture is working. `dummy_analysis.properties` includes every relevant folder setting as relative path. If you choose to setup your folders in a different way, you might need to use absolute paths instead for those parameters.

Now we open the console and change to the directory in which `kernelhaven.jar` is contained.

```
$ cd ~/Desktop/KernelHaven/
```

Because `kernelhaven.jar` is in the same the directory as `dummy_analysis.properties` we can execute KernelHaven by only passing a relative path as parameter.

```
$ java -jar kernelhaven.jar dummy_analysis.properties
```

After running the command, the console shows the output of the KernelHaven-Execution.