KernelHaven User Documentation

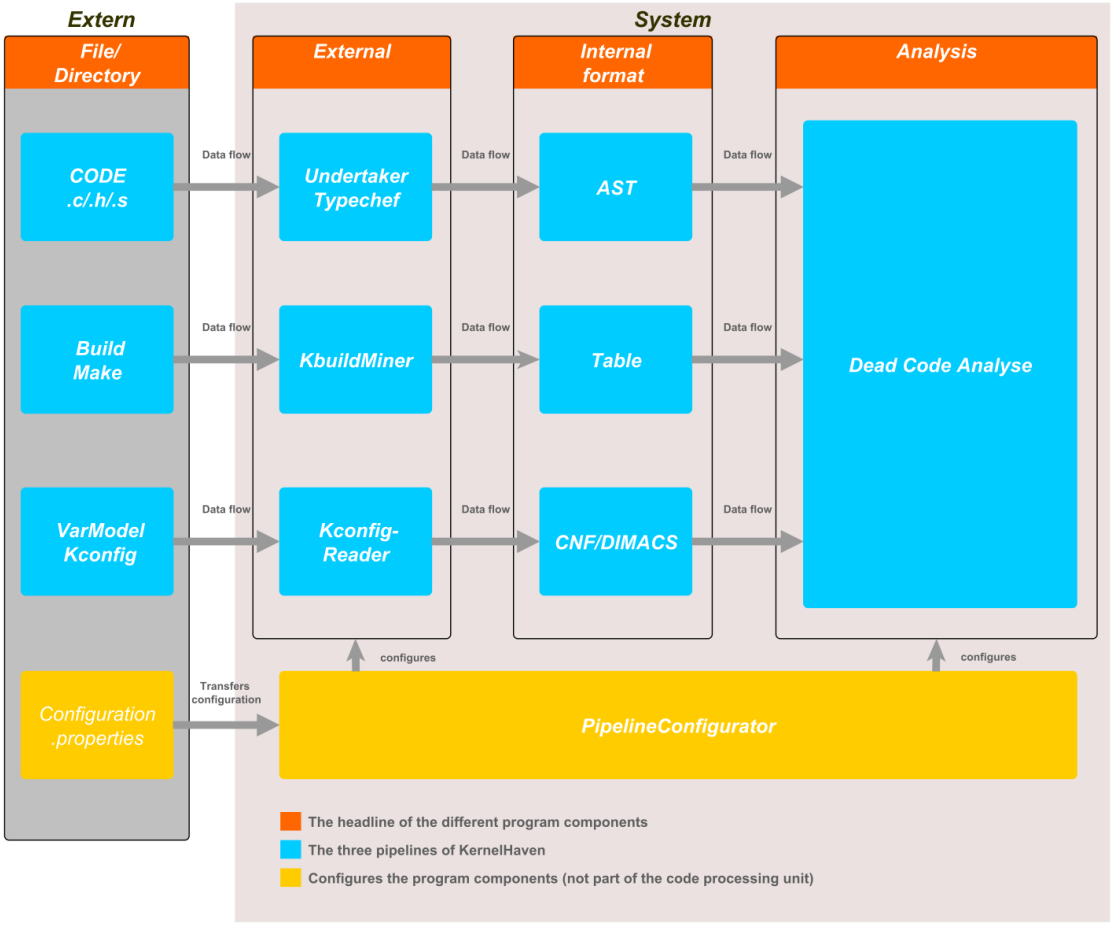
This document serves as a manual for users of KernelHaven. It will explain how to set up KernelHaven and execute an analysis. Refer to the developer documentation if you want to write new extractors or analyzes.

Contents

1 Concept

KernelHaven offers a generic infrastructure for performing different analyses on product lines. The architecture is shown in the figure below. The variability information from the product line is extracted via extractors. This is split into three pipelines: source code, build system and variability model. Each pipeline has an extractor that reads the necessary data from the product line, and a model in which the data is represented. Depending on the product line being analyzed, different extractors are used. This allows the infrastructure to adapt to different product lines, while still having a common data representation in the models.

KernelHaven has a plug-in system for easy replacement of extractors and analyses. Each plug-in is a Jar file placed in a central plugins folder. Based on the user configuration for an execution, different Java classes are instantiated at runtime, which serve as the extractors and analysis.



2 Setup and Execution

KernelHaven is written in Java, and thus does not depend on any specific operating system. The only requirement for executing KernelHaven is an installed Java VM (version 8 or greater). However, some extractors are not platform independent; they usually only run on Linux. Refer to the different plug-in descriptions for additional limitations and dependencies. KernelHaven and all extractors developed by us are tested to run at least on Ubuntu 16.04.

KernelHaven does not require any specific directory structure. All paths that are relevant for an execution can be configured. However, an installation of KernelHaven typically looks like this:

```
kernel_haven/
├── cache/
│   └── ...
├── log/
│   └── ...
├── output/
│   └── ...
├── plugins/
│   ├── cnfutils.jar
│   ├── kbuildminerextractor.jar
│   ├── kconfigreaderextractor.jar
│   ├── undeadanalyzer.jar
│   └── undertakerextractor.jar
├── res/
│   └── ...
├── kernel_haven.jar
└── dead_code.properties
```

The following list describes the usage of the different files and folders:

- The `kernel_haven.jar` file is an executable Jar file and contains the main infrastructure.
- The `dead_code.properties` file contains the configuration of a KernelHaven execution.
- The `cache/` folder will contain the cache files generated by the extractors. The user will usually not look into this directory (except maybe for manually clearing the cache).
- The `log/` folder will contain the log files generated by the infrastructure.
- The `output/` folder will contain the output files created by the analysis.
- The `plugins/` folder contains Jar files with plugins, that will be loaded by the infrastructure.
- The `res/` folder will contain resource files needed by the extractors. The extractors manage the contents of this themselves; the user does not have to look into this folder.

An example configuration file (`dead_code.properties`) for executing a dead code analysis on a Linux Kernel could look like this:

```
# Linux Source Tree
source_tree = /path/to/linux_source
arch = x86

# Analysis
analysis.class = net.ssehub.kernel_haven.default_analyses.DeadCodeAnalysis

# Code Extractor
code.provider.cache.read = true
code.provider.cache.write = true
code.extractor.class = net.ssehub.kernel_haven.undertaker.UndertakerExtractor
code.extractor.threads = 4

# Build Extractor
build.provider.cache.read = true
build.provider.cache.write = true
build.extractor.class = net.ssehub.kernel_haven.kbuildminer.KbuildMinerExtractor

# Variability Extractor
variability.provider.cache.read = true
variability.provider.cache.write = true
variability.extractor.class = net.ssehub.kernel_haven.kconfigreader.KconfigReaderExtractor

# Logging
log.console = true
log.file = true

# Directories
archive.dir = .
cache_dir = cache/
log.dir = log/
output_dir = output/
plugins_dir = plugins/
resource_dir = res/
```

This is a standard Java properties file. For a full list of possible configuration options, refer to the `config_template.properties` file in KernelHaven. Note that some extractors also have their own configuration options, in addition to the ones from the infrastructure. Relative paths are interpreted relative to the working directory from which KernelHaven is started; this can be different from the location of the `kernel_haven.jar` file.

To execute the infrastructure with the given configuration, start `kernel_haven.jar` and pass the configuration file as the only parameter:

```
java –jar kernel_haven.jar dead_code.properties
```

KernelHaven also supports archiving of an execution. If it is enabled, all relevant files are combined into a Zip archive after the execution finished. To enable archiving, either set the configuration option `archive` to `true` in the properties file, or start KernelHaven with the `--archive` parameter:

```
java –jar kernel_haven.jar --archive dead_code.properties
```

Note that it is possible to pass additional parameters to the Java call. For example, it is possible to set the maximum Java heap size, which may become necessary when dealing with a lot of data:

```
java –Xmx100g –jar kernel_haven.jar dead_code.properties
```

3 Extractors

The following sections introduce some commonly used extractor plug-ins for KernelHaven. They mostly focus on extracting data from the Linux Kernel and similar product lines.

3.1 KconfigReaderExtractor

A variability model extractor for Kconfig product lines based on the KconfigReader tool.

3.1.1 Capabilities

This extractor reads the Kconfig model of the Linux Kernel. 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.

3.1.2 Usage

Set `variability.extractor.class` to `net.ssehub.kernel_haven.kconfigreader.KconfigReaderExtractor` in the KernelHaven properties.

3.1.3 Dependencies

- Only runs on a Linux operating system.
- Requires a C compiler and `make` to be installed. On Ubuntu, just install the `build-essential` package via: `sudo apt install build-essential`

3.1.4 Configuration

In addition to the default ones, this extractor has the following configuration options in the KernelHaven properties:

- `variability.extractor.find_locations`: 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 variability model. Default is false.

3.2 KbuildMinerExtractor

A build model extractor for Kconfig product lines based on the KbuildMiner tool.

3.2.1 Capabilities

This extractor finds conditional compilation settings in the Kbuild files (`Kbuild*`, `Makefile*`) of the Linux Kernel.

3.2.2 Usage

Set `build.extractor.class` to `net.ssehub.kernel_haven.kbuildminer.KbuildMinerExtractor` in the KernelHaven properties.

3.2.3 Dependencies

None.

3.2.4 Configuration

In addition to the default ones, this extractor has the following configuration options in the KernelHaven properties:

- `build.extracator.top_folders`: Comma-separated list of top-folders to analyze in the product line. By default, this is generated for the Linux Kernel from the `arch` setting as follows: `arch/<arch>,block,crypto,drivers,firmware,fs,init,ipc,kernel,lib,mm,net,security,sound`

3.3 UndertakerExtractor

A code model extractor for `#ifdef`s based on the Undertaker tool.

3.3.1 Capabilities

This extractor finds `#ifdef` (and similar) blocks in source files (`*.c`, `*.h`, `*.S`) and extracts the hierarchical condition structure.

3.3.2 Usage

Set `code.extractor.class` to `net.ssehub.kernel_haven.undertaker.UndertakerExtractor` in the KernelHaven properties.

3.3.3 Dependencies

- Only runs on a Linux operating system.

3.3.4 Configuration

In addition to the default ones, this extractor has the following configuration options in the KernelHaven properties:

- `code.extractor.hang_timeout`: 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. The default value is 20000.

3.4 TypeChefExtractor

A code model extractor for C code based on the TypeChef tool.

3.4.1 Capabilities

This extractor extracts a variability-aware abstract syntax tree (AST) from C code files (`*.c`) and considers preprocessor macros.

3.4.2 Usage

Set `code.extractor.class` to `net.ssehub.kernel_haven.typechef.TypechefExtractor` in the KernelHaven properties.

3.4.3 Dependencies

- The CnfUtils plugin.

3.4.4 Configuration

TODO

4 Analyzes

The following sections introduce some commonly used analysis plug-ins for KernelHaven. They mostly focus on the Linux Kernel and similar product lines.

4.1 UnDeadAnalyzer

Analysis for finding dead code and unused variability variables.

4.1.1 Capabilities

Contains two analyzes:

1. Dead code block detection: Detects code blocks in source files that can never be selected.
2. Missing variability variables detection: Detects Kconfig (`CONFIG_`) variables, which are either defined but not used or used but not defined.

4.1.2 Usage

Set `analysis.class` to `net.ssehub.kernel_haven.default_analyses.DeadCodeAnalysis` or `net.ssehub.kernel_haven.default_analyses.MissingAnalysis` in the KernelHaven properties.

4.1.3 Dependencies

- The CnfUtils plugin.

4.1.4 Configuration

In addition to the default ones, this extractor has the following configuration options in the KernelHaven properties:

- `analysis.missing.type`: Sets the type of missing analysis. `D` for “defined but not used”, `U` for “used but not defined”. Not case sensitive. Default is `D`.

4.2 FeatureEffectAnalysis

Analysis for calculating feature effects.

4.2.1 Capabilities

Contains two analyzes:

1. PcFinder: Finds all presence conditions for variability variables in the source code.
2. FeatureEffectFinder: Calculates feature effects for variability variables.

4.2.2 Usage

Set `analysis.class` to `net.ssehub.kernel_haven.feature_effects.PcFinder` or `net.ssehub.kernel_haven.feature_effects.FeatureEffectFinder` in the KernelHaven properties.

4.2.3 Dependencies

None.

4.2.4 Configuration

In addition to the default ones, this extractor has the following configuration options in the KernelHaven properties:

- `analysis.relevant_variables`: A Java regular expression to decide which variables are relevant variability variables. Default is `.*` (match all). Example: `CONFIG_.*` (all variables that start with `CONFIG_`).

5 Utilities

The following sections introduce some utility plug-ins for KernelHaven. These are neither extractors nor analyzes, but provide useful functionality for them. Some other plug-ins depend on these utilities.

5.1 CnfUtils

Utilities for converting Boolean formulas to CNF (conjunctive normal form) and solving them via SAT-solvers.

5.1.1 Usage

Place `cnfuitls.jar` in the plugins folder to make it available for other plug-ins.

5.2 IOUtils

Utilities for reading and writing CSV and Excel (`.xls`) files.

5.2.1 Usage

Place `ioutils.jar` in the plugins folder to make it available for other plug-ins.