autokernel

Jun 18, 2020

Getting started

1 Introduction 3

2 Quick start guide 7

3 Usage 11

4 Concepts 21

5 Configuration Syntax 25

6 Configuration Directives 33

7 License 47

8 Acknowledgments 49

Welcome to the official documentation of autokernel! If you are visiting for the fist time, we recommend that you read the *Introduction* page to get an overview of what autokernel has to offer.

Otherwise, the sidebar and the table of contents below should let you easily access your topic of interest. You can also use the search function in the top left corner.

CHAPTER 1

Introduction

Autokernel is primarily a kernel configuration management tool. This means its main purpose is to be used as a tool to generate a `.config` file from your particular set of kernel options. You can use it to document and build your configuration, and it will ensure that the configuration has no conflicts. To help you to write a good config, it comes with a set of helpful features which are outlined below.

To skip all of the chatter, head over to *Usage* to begin using autokernel or *Concepts* to learn about some important concepts in autokernel.

1.1 What problem does it solve?

» *Which kernel options do I need to enable to use this USB device?*

» *Why have I enabled this kernel option again?*

» *Which kernel options did I change compared to the default?*

» *Why is this option still not enabled even though i did explicitly set it?*

Does any of these question sound familiar? Then this tool might actually solve a problem or two for you. On the other hand, if you are now thinking what all this means, you will likely not gain any significant benefit from this tool. But obviously you are welcome to try it out!

1.2 Feature Overview

The main feature of autokernel is kernel configuration management. In practice this means you write an autokernel configuration and it generates a `.config` for your kernel version, but with additional features:

1.2.1 Conflict detection

Usually, enabling an option (for example `SECURITY_SELINUX`) but later disabling a direct or indirect dependency (like `AUDITING`) will lead to the first option being deselected again. This is annoying and misses the original intent. Autokernel will exit with a conflict error in this case and present the offending lines in the configuration. Once set, a symbol's value will be internally pinned and any assignments that would change it present an error. See *Pinning symbol values* for more info.

1.2.2 Satisfying dependencies

If you enable a symbol (for example `WLAN`), but forget to enable some of its dependencies (like `NETDEVICES`), autokernel will throw an error. But it can also help you to resolve these dependencies and present you with a list of options that need to be enabled to allow the assignment. See *Enabling arbitrary symbols* for more info.

1.2.3 Symbol validation

If you mistype an option, you will get an error when building the config. This also works when symbols get renamed or removed in a future kernel version (like `THUNDERBOLT` is now effectively `USB4` since 5.6). As invalid symbol usage is a hard error, you will notice them before your kernel is built.

1.2.4 Conditionals

Sometimes you want to support different machines or kernel versions, but as the kernel evolves, symbols might get added, renamed or removed. By allowing simple conditional expressions in the configuration, you can easily evolve your config while staying backwards compatible to previous versions, and have one configuration for multiple machines. See *Conditional Expressions* for more info.

1.2.5 Structure and Documentation

In the configuration you will be able to properly document your choices, which is often neglected when using menuconfig. If done correctly, you will never loose track of what has been changed and more importantly why it was changed. You can also structure parts of the configurations into individual modules, which is useful if you want to use the same configuration base on different machines.

1.2.6 Detecting options

Autokernel can automatically detect kernel configuration options for your system. This works by gathering system information from `/sys` and relating it to a configuration option database (LKDDb). For more information on how it works and how to use it see *Detecting kernel options*.

1.2.7 Build system

Autokernel can optionally be used as a full kernel build system. It sounds like a lot, but it is actually nothing more than executing `make` and the specified command to build your initramfs (optional) for you. Eventually, a second build pass is needed to integrate the initramfs into the kernel. Other than that, it supports mounting target directories, and keeping your installation directory clean by only keeping the last N builds. See *Building and installing the kernel* for more information.

1.2.8 Kernel hardening

Autokernel provides a preconfigured module for kernel hardening. Every choice is fully documented and explanined. See *Hardening the kernel* for more information.

But advantages never come without disadvantages. The obvious ones here are the additional effort of writing a proper configuration instead of simply using menuconfig, and also needing an additional tool for a task that shouldn't.

1.3 Alternative: Merging .config files

Some users might already be familiar with a similar workflow, in which you collect your changes to the default kernel configuration in one or more kconf files, which are then applied to a fresh kernel configuration with `./scripts/kconfig/merge_config.sh` from the kernel tree to create the final configuration.

While this method does work, it has some major downsides - like the total lack of error messages. If you mistype a config's name, nobody will tell you. You will notice it eventually, when you have started the new kernel and wonder why something is still not working. Other than that you might notice that even though you've typed everything correctly, an option might still be unchanged because it had missing dependencies. It can be a total pain to need 3 to 4 iterations of diffing config files just to ensure everything is finally as expected.

As autokernel uses kconfiglib to parse and process the Kconfig files exactly as the kernel would, it can directly check if options are assignable or would otherwise conflict, and report this as a warning or error to the user.

CHAPTER 2

Quick start guide

Welcome to the quick start guide! Here you will see a selection of commands to quickly get started with autokernel.

This is basically an extreme summary of the *Usage* section. It is recommended to read both the *Introduction* and the *Usage* section, as commands and their effects are explained in greater detail there.

2.1 Basic invokation

Hint: If you are **not** using gentoo or another source distribution, which has kernel sources available under `/usr/src/linux`, use `-K kernel_dir` to specify the kernel directory. Autokernel will not work otherwise.

2.2 Detecting kernel options

Detect configuration options for your system and report differences to the running kernel:

```
autokernel detect -c
```

The next command generates output as commonly seen in a .config file. This is useful if you want to use other tools with the detected options:

```
# Generates a kconf file for usage with other tools
autokernel detect -t kconf -o ".config.local"
```

You can also generate an autokernel module from the result:

```
# Generates a module named 'local'
autokernel detect -o "/etc/autokernel/modules.d/local.conf"
```

Warning: Be aware that even though this detection mechanism is nice to have, it is also far from perfect. The option database is automatically generated from kernel sources, and so you will have false positives and false negatives. You should work through the list of detected options and decide if you really want to enable them.

2.3 Generating the kernel configuration

Generate a `.config` file from your autokernel configuration with `generate-config`:

```
# Generates a config file at the given location
autokernel generate-config -o .config
```

Alternatively, you can compare the theoretically generated config to another config by using `check`:

```
# Checks against the current kernel (/proc/config.gz)
autokernel check
# Check against the .config file of another kernel version
autokernel check -c -k some/kernel/dir some/kernel/dir/.config
```

2.4 Enabling arbitrary symbols

You can use the `satisfy` command to automatically show which dependencies are missing to enable an option:

```
autokernel satisfy -g WLAN
```

2.5 Querying symbol information

Query symbol information (menuconfig help text) with `info`:

```
autokernel info WLAN
```

2.6 Querying symbol reverse dependencies

Use the `revdeps` command to show all symbols that depend on the given symbol:

```
autokernel revdeps EXPERT
```

2.7 Hardening the kernel

Check out *Hardening the kernel* for a short description of the included hardening module.

2.8 Building and installing the kernel

To build and install the kernel according to your configuration use `build` and `install`:

```
autokernel build   # Just build targets
autokernel install # Just install targets
autokernel all     # Do both
```

CHAPTER 3

Usage

Welcome to the usage section! Below is a table of contents for this guide. Feel free to directly jump to the topic of your interest.

- *Installation*
- *Basic invokation*
- *Detecting kernel options*
  - *Comparing to the current kernel*
  - *Generating an autokernel module*
- *Generating the kernel configuration*
  - *Generating a .config file*
  - *Comparing to another config*
- *Writing configuration*
  - *Configuration primer*
    - *Configuration excerpt*
    - *Modules*
    - *Assigning symbols*
    - *Adding to the kernel command line*
    - *Best practices*
  - *Enabling arbitrary symbols*
  - *Querying symbol information*
  - *Querying symbol reverse dependencies*

- *Hardening the kernel*
- *Building and installing the kernel*
  - *Just the kernel*
  - *Using dracut to build an initramfs*
  - *Mounting directories and purging files*

3.1 Installation

Use can use pip to install autokernel, or run from source:

Using pip

```
pip install autokernel
autokernel --help
```

From source

```
git clone "https://github.com/oddlama/autokernel.git"
cd autokernel
pip install -r requirements.txt
./bin/autokernel.py --help
```

Setting up the basic configuration

If you intend to use autokernel for more than testing purposes, you should allow autokernel to setup a basic configuration in `/etc/autokernel`, which can then be edited. This command will never override any existing configuration.

```
# Populates /etc/autokernel with the default configuration, if no configuration␣
↪exists.
sudo autokernel setup
```

Note: If you don't setup the configuration directory, autokernel will fallback to a minimal internal configuration to allow for testing.

3.2 Basic invokation

All invocations of autokernel follow this scheme:

```
Usage: autokernel [opts...] <command> [command_opts...]
```

For additional information, refer to the help texts:

```
autokernel --help            # General help
autokernel <command> --help  # Command specific help
```

Configuration

Before proceeding, you might want to have a look at the default configuration in `/etc/autokernel` to familiarize yourself with the general format.

To explicitly specify a configuration, use the option `-C path/to/autokernel.conf`

Kernel location

By default, autokernel expects the kernel to reside in `/usr/src/linux`. If you want to specify another location, use the option `-K path/to/kernel`

Hint: If you are **not** using gentoo or another source distribution, use `-K kernel_dir` to specify the kernel directory. Autokernel will not work otherwise.

3.3 Detecting kernel options

Autokernel can automatically detect kernel configuration options for your system. This is done mainly by collecting bus and device information from the `/sys/bus` tree, which is exposed by the currently running kernel. It then relates this information to a configuration option database (LKDDb), selects the corresponding symbols and the necessary dependencies.

It might be beneficial to run detection while using a very generic and modular kernel, such as the kernel from Arch Linux. This increases the likelihood of having all necessary buses and features enabled to detect connected devices.

The problem is that we cannot detect USB devices, if the current kernel does not support that bus in the first place.

Hint: You can run autokernel directly on an Arch Linux live system.

3.3.1 Comparing to the current kernel

```
autokernel detect -c
```

This command detects option values and outputs a summary in which you can easily see the current value of the symbol and the suggested one. This gives a good overview over what would be changed.

Note: Be aware that autokernel never suggests to build modules, so you might see several `[m]` → `[y]` changes. You should build commonly used features into the kernel to cut down on load times and attack surface (if you manage to disable `MODULES`).

3.3.2 Generating an autokernel module

You can generate a module from the detected options, which can then be put into `/etc/autokernel/modules.d` and included in your configuration.

```
# Generates a module named 'local'
autokernel detect -o "/etc/autokernel/modules.d/local.conf"
```

Alternatively, autokernel can output kconf files (like `.config`) if you want to use other tools:

```
# Generates a kconf file for usage with other tools
autokernel detect -t kconf -o ".config.local"
```

Warning: Be aware that even though this detection mechanism is nice to have, it is also far from perfect. The option database is automatically generated from kernel sources, and so you will have false positives and false negatives. You should work through the list of detected options and decide if you really want to enable them.

3.4 Generating the kernel configuration

3.4.1 Generating a .config file

To generate a `.config` file, all you need to do is execute the following command:

```
# Generates .config directly in the kernel directory (see -K)
autokernel generate-config
# Generates a config file at the given location
autokernel generate-config -o test.config
```

3.4.2 Comparing to another config

If you instead want to see differences to another kconf file (.config), you can use the `check` command. This is especially useful to see what has changed between kernel versions.

```
# Checks against the current kernel (/proc/config.gz)
autokernel check
# Check against explicit file
autokernel check -c some/.config
# Check against the .config file of another kernel version
autokernel check -c -k some/kernel/dir some/kernel/dir/.config
```

3.5 Writing configuration

3.5.1 Configuration primer

You will most likely only need a few directives to write your kernel config. Apart from configuring kernel options, autokernel's configuration allows you to specify some settings for building the initramfs, and the general build and installation process. For a more in-depth explanation of autokernel's configuration, see the sections about *Configuration Syntax* and *Configuration Directives*.

Hint: The default configuration that is generated when using `autokernel setup` is a great starting point to write your own configuration. If you have already changed it, you can view the original files on github or in the autokernel module directory under `autokernel/contrib/etc`.

The most important directives are outlined in the following and by this example:

Configuration excerpt

```
module base {
    # Begin with the kernel defconfig
    merge "{KERNEL_DIR}/arch/{ARCH}/configs/{UNAME_ARCH}_defconfig";

    # Enable expert options
    set EXPERT y;
    # Enable the use of modules
    set MODULES y;
}

module net {
    # Enable basic networking support.
    set NET y;
    # Enable IP support.
    set INET y;
    # Enable ipv6
    set IPV6 y;
    # IPv6 through IPv4 tunnel
    set IPV6_SIT y;

    # Enable wireguard tunnel
    if $kernel_version >= 5.6 {
        set WIREGUARD y;
    }
}

# The main module
kernel {
    # Begin with a proper base config
    use base;

    # The hardening module is provided in /etc/autokernel/modules.d,
    # if you have used `autokernel setup`.
    use hardening;
    # You can detect configuration options for your local system
    # by using `autokernel detect` and store them in /etc/autokernel/modules.d/local.
↪conf
    use local;

    # Proceed to make your changes.
    use net;
}
```

Modules

Kernel configuration is done in module blocks. Modules provide encapsulation for options that belong together and help to keep the config organized. The main module is the `kernel { ... }` block. You need to `use` (include) modules in this block to include them in your config. Module can also include other modules, cyclic or recursive includes are impossible by design.

Assigning symbols

To write your configuration, you need to assign values to kernel symbols. This must be done inside a module. Here is an example which shows the most common usage patterns.

```
module test {
    set USB y;     # Enable usb support
    set USB;       # Shorthand syntax for y
    set USB "y";   # All parameters may be quoted

    set KVM m;     # Build KVM as module
    # Example of setting a non-tristate option.
    set DEFAULT_MMAP_MIN_ADDR 65536;
    set DEFAULT_MMAP_MIN_ADDR "65536"; # or with quotes

    # Set a string symbol
    set DEFAULT_HOSTNAME refrigerator;   # OK
    set DEFAULT_HOSTNAME "refrigerator"; # Also OK

    # Inline condition example
    set WIREGUARD if $kernel_version >= 5.6;

    # Conditions work with usual expression syntax
    # and you can examine symbols
    if X86 and not X86_64 {
        set DEFAULT_HOSTNAME "linux_x86";
    else if (X86_64) {
        set DEFAULT_HOSTNAME "linux_x86_64";
    } else if $arch == "mips" {
        set DEFAULT_HOSTNAME "linux_mips";
    } else {
        set DEFAULT_HOSTNAME "linux_other";
    }
}
```

Adding to the kernel command line

By using a statement like

```
add_cmdline "rng_core.default_quality=512";
```

you can directly append options to the builtin commandline.

Note: This will cause `CMDLINE_BOOL` to be enabled and `CMDLINE` to be set to the resulting string.

Best practices

Here are some general best practices for writing autokernel configurations:

- Always start by merging a `defconfig` file, to use the equivalent of `make defconfig` as the base.
- Use modules to organize your configuration.
- Document your choices with comments.
- Use conditionals to write generic modules so they can be used for multiple kernel versions and maybe even across machines.

3.5.2 Enabling arbitrary symbols

Sometimes you want to enable a symbol, but don't know which dependencies you have to enable first. Use the `satisfy` command to let autokernel find a valid configuration for you. By default the output will be based on the kernel configuration managed by autokernel. If you want to use a clean default config, use `satisfy -g`.

```
autokernel satisfy -g WLAN
```

Will output the following on kernel version 5.6.1:

```
# Generated by autokernel on 2020-04-13 13:37:42 UTC
# vim: set ft=ruby ts=4 sw=4 sts=-1 noet:
# required by config_netdevices
# required by config_wlan
module config_net {
        set NET y;
}

# required by config_wlan
module config_netdevices {
        use config_net;
        set NETDEVICES y;
}

# required by rename_me
module config_wlan {
        use config_netdevices;
        use config_net;
        set WLAN y;
}

module rename_me {
        use config_wlan;
}
```

Hint: To preserve the dependency structure and avoid duplication, autokernel will output one module per encountered option. You can and probably should extract only the relevant symbols assignments.

Note: Even though modules are used, autokernel guarantees to set dependencies before dependents. You can therefore simply extract all set statemtents and write them one after another for the same result.

3.5.3 Querying symbol information

In case you have forgotten the meaning of a kernel symbol, you can use the `info` command to show the attached help text as you would encounter it in `make menuconfig`.

```
autokernel info WLAN
```

3.5.4 Querying symbol reverse dependencies

You can use the `revdeps` command to show all symbols that somehow depend on the given symbol.

```
autokernel revdeps EXPERT
```

3.6 Hardening the kernel

Autokernel provides a preconfigured module for kernel hardening, which is installed to `/etc/autokernel/modules.d/hardening.conf` if you used `autokernel setup`. Otherwise you will find it on github or in the autokernel module directory under `autokernel/contrib/etc/modules_d/hardening.conf`.

The hardening module is compatible with any kernel version >= 4.0. Every choice is also fully documented and explanined. Feel free to adjust it to your needs.

If the file is included, you can enable it like this:

```
kernel {
    # Use hardening early in your config. Errors will then be caused by
    # the offending assignment instead of the assignments in the hardening
    # module, which makes error messages easier to read.
    use hardening;

    # ...
}
```

3.7 Building and installing the kernel

Autokernel can be used to build the kernel and to install the resulting files. The respective commands are `build` and `install`, but they can be combined by using the `all` command.

```
autokernel all # Build the kernel and install it
```

Hint: You can use *build hooks* and *install hooks* to add additional functionality before and after execution of the respective phase.

Warning: Be especially careful with file and directory permissions for hook scripts! Autokernel will do a sanity check to ensure that no other user can inject commands by editing the autokernel configuration, but in the end it is your responsibility.

3.7.1 Just the kernel

This is an example that shows how the configuration can be used to:

- disable initramfs generation
- install the kernel to `/boot`
- install modules to the default location

```
initramfs {
    enable false;
}

install {
    target_dir "/boot";
    target_initramfs false;
    target_config false;
    # ...
}
```

3.7.2 Using dracut to build an initramfs

Here is an example which builds an initramfs with dracut and integrates the result back into the kernel. This means you still only need to install the kernel.

Hint: When using builtin initramfs, setting any of the `INITRAMFS_COMPRESSION_*` options will still compress it on integration.

```
kernel {
    # Optional: Use LZ4 as compression algorithm for built-in initramfs
    set RD_LZ4 y if BLK_DEV_INITRD;
    set INITRAMFS_COMPRESSION_LZ4 y if INITRAMFS_SOURCE;
}

initramfs {
    enable true;
    builtin true;

    # Adjust this to your needs
    build_command "dracut"
        "--conf"          "/dev/null" # Disables external configuration
        "--confdir"       "/dev/null" # Disables external configuration
        "--kmoddir"       "{MODULES_PREFIX}/lib/modules/{KERNEL_VERSION}"
        "--kver"          "{KERNEL_VERSION}"
        "--no-compress"   # Only if the initramfs is to be integrated into the kernel
        "--no-hostonly"
        "--ro-mnt"
        "--add"           "bash crypt crypt-gpg"
        "--force"         # Overwrite existing files
        "{INITRAMFS_OUTPUT}";
}
```

3.7.3 Mounting directories and purging files

If you have an fstab entry for a directory used in the target directory, you can have autokernel mount the directory on install. See *mount* for more information.

If you want to purge old builds from the target directory, you can use the *keep_old* directive.

```
install {
    # ...

    # Mount /boot when installing and unmount afterwards
    mount "/boot";

    # Keeps the last two builds and removes the rest from the
    # target directory
    keep_old 2;
}
```

CHAPTER 4

Concepts

This page is going to elaborate on some important concepts in autokernel.

4.1 Modules

Modules are blocks in the autokernel configuration which are used to write the actual kernel configuration. A module can *set* symbol values, *merge* external kconf files, *assert* expressions, *use* (include) other modules and *add command-line* strings. They are intended to provide a level of encapsulation for groups of symbols.

Example module

```
module example {
    # Asserts that the configured kernel is at least on version 4.0
    assert $kernel_version >= 4.0: "this kernel is too old!";

    # Merge in the x86_64 defconfig
    merge "{KERNEL_DIR}/arch/x86/configs/x86_64_defconfig";

    # Sets DEFAULT_MMAP_MIN_ADDR if X86 is set
    if X86 {
        set DEFAULT_MMAP_MIN_ADDR 65536;
    }

    # Include another module
    use some_dependency;
}

module some_dependency {
    # ...
}
```

4.2 Pinning symbol values

The first important concept is pinning. As soon as a symbol's value is changed or observed, it will be pinned, meaning the value is then fixed. In the beginning, all symbols will start with their default values, as specified by the kernel's Kconfig.

Successive assigments to these symbols will become hard errors, if they would change the pinned value. This allows modules to use logic based on symbol values, without imposing implicit ordering constraints, or surprise pitfalls down the road. Wrong ordering will lead to errors instead of silently breaking previous assumptions.

Note: Conflicts are always errors. This ensures that the same conditions always has the same outcome, no matter where it stands in the configuration.

Pinning Examples

```
# Sets and pins NET to [y] (cause: explicit assignment)
set NET y;

# Pins USB to its current value (cause: evaluation in condition)
if USB {
    set EXAMPLE y;
}

# Does not pin BT, because no statement depends on the condition
if BT { }

# Does nothing if IWLWIFI is already pinned. Otherwise assigns
# *without* pinning. Useful to set new defaults for values
# but still allowing explicit changes.
try set IWLWIFI y;
```

Conflict Example

```
1  # If NET is enabled, also enable TUN. This pins NET.
2  if NET {
3      set TUN y;
4  }
5
6  # Assume NET was [y]. In that case NET is pinned to [y] in line 3.
7  # This would break the assumption in line 3, as a re-evaluation of
8  # the condition would have a different result.
9  set NET n; # error: confilict
```

4.3 Implicit vs. explicit changes

There are explicit and implict assignments of symbol values. All direct assignments via `set` are explicit. An implicit assignment occurrs, when an explicit assignment triggers a change in a symbols that depends on the assigned symbol.

Note: Explicit changes will pin the value of a symbols, while implicit changes do not.

Implicit assignments also occurr when using the *merge* statement. They can also be forced by using *try set* instead of just `set`. This should only be used in special occasions, like when you want to set a new default value for a symbol while still allowing the user to override it.

Correct usage of `try set`

It's a common pattern to use `try set` directly followed by a conditional on the same symbol. This way you can ensure a module works with either setting, but add a default in case the user didn't care:

```
# By default disable DEVMEM
try set DEVMEM n;

# If the user has still enabled it, at least enable STRICT mode
if DEVMEM {
    set STRICT_DEVMEM y;
}
```

Warning: Do not use `try set` to resolve conflicts! A conflict always means that there is something wrong with your configuration or ordering. Only use `try set` to set new defaults.

Explicit assignments

```
# Explicitly sets NET to n
set NET n;

# Explicitly sets symbols mentioned in the given kconf file
merge "{KERNEL_DIR}/arch/x86/configs/x86_64_defconfig";
```

Implicit assignments

```
# Implicitly sets NET to n
try set NET n;

# Implicitly assigns a lot of other options
# (all that indirectly depend on MODULES)
set MODULES n;
```

CHAPTER 5

Configuration Syntax

First of all, here is a short example for a configuration that is builds upon on the `x86_64_defconfig`, disables `MODULES` and enables `WIREGUARD` if the kernel version is greater than 5.6:

Listing 1: Short configuration example

```
module base {
    # Begin with the x86_64 defconfig
    merge "{KERNEL_DIR}/arch/x86/configs/x86_64_defconfig";
    # Disable modules
    set MODULES n;
}

kernel {
    use base;

    # Enable wireguard on new kernels
    set WIREGUARD y if $kernel_version >= 5.6;
}
```

5.1 General Syntax

Whitespace

The configuration format is not sensitive to whitespace. Sometimes, a whitespace character is needed to separate tokens, but other than that whitespace is ignored.

Comments

The comment character is ‘#’. It can be appended to any line and will comment everything until the end of that line.

Booleans

Boolean options recognize the following arguments:

Boolean value	Recognized aliases
`false`	`false`, `0`, `no`, `n`, `off`
`true`	`true`, `1`, `yes`, `y`, `on`

Strings

You can optionally quote strings with `"double"` or `'single'` quotes. There is no semantic difference between the two quoting styles. Quoting ist mostly optional, but some option parameters require quoting (will be specified). In quoted strings, you can use the following escape sequences:

Escape sequence	Meaning
`\\`	Single backslash \
`\"`	`"`
`\'`	`'`
`\n`	Newline
`\r`	Carriage Return
`\t`	Tab
`\x1b`	2-digit hex escapes
`\033`	Octal escapes
`\u2665`	4-digit unicode hex escapes
`\U0001f608`	8-digit unicode hex escapes
`\N{Dark Shade}`	Unicode characters by name

Blocks

Blocks are enclosed in brackets, like `blockname { }`. Empty blocks may be omitted.

Statements

Statements are terminated with a semicolon `;`. Different blocks allow different statements.

Formal specification (EBNF)

For an EBNF like description of the config syntax, refer to the config.lark file inside the autokernel python module directory.

Syntax highlighting

You can use ruby syntax highlighting, which gives quite good results (at least in vim).

5.2 Common variables

In several places you will be able to refer to variables. Mostly in strings and pathnames. All statements will explicitly state which variables can be used in the description of the directive.

The common variables are:

Name	Description
`{KERNEL_DIR}`	The current kernel directory path
`{KERNEL_VERSION}`	The current kernel version
`{ARCH}`	The host architecture as the kernel sees it
`{UNAME_ARCH}`	The host architecture as `uname -m` reports it

Note: The internal kernel architecture differs from `uname -m`. For example it will be `x86` for both `x86` and `x86_64` systems.

5.3 Conditional Expressions

Autokernel supports conditions for all statements in *module* blocks. They will only be executed if all attached conditions are met.

5.3.1 Overview

Conditions can be used in both traditional form with optional `else if` and `else` blocks, or as python like trailing conditions. The block form can of course be nested, and styles can be mixed freely.

Block form

```
# Traditional if clause with optional blocks
if <expression> {
    set A y;
} else if <expression> {
    set A n;
} else {
    # Nested block
    if <expression> {
        set B n;
    }

    set C n;
}
```

Inline short form

```
set A y if <expression>;

# is the same as
if <expression> {
    set A y;
}
```

Note: Trailing conditions are currently attached to the whole statement and cannot use an `else` token to specify an alternate value.

5.3.2 Expressions

Expressions are written as they are in most other programming languages:

Expression syntax

Expression	Meaning
`A or B`, `A || B`	(A B)
`A and B`, `A && B`	(A B)
`A or B and C`	(A (B C))
`not A`, `!A`	¬A
`A`	Shorthand for `A != 'n'`
`A <cmp> B`	Comparison. See *Comparisons*

Operator precedence

1. `()`: expression grouping
2. `A <cmp> B`: any explicit comparison
3. `not`: inversion operator
4. `and`: and clauses
5. `or`: or clauses

5.3.3 Comparisons

All expressions boil down to comparisons.

Comparison syntax

Expression	Meaning
`A == B`, `A is B`	A is equal to B
`A != B`, `A is not B`	A is not equal to B
`A <= B`	A is less than or equal to B
`A < B`	A is less than B
`A >= B`	A is greater than or equal to B
`A > B`	A is greater than B

Chaining

All comparison operators can be chained. This means you can write `4.0 <= $kernel_version < 5.0`, or even `A != B != C != D`. There is no difference between chaining and writing the expanded form.

Note: Comparisons in chained form will always compare actual values and *never* intermediate truth values. `A != B != C` is guaranteed to be the same as `A != B and B != C`.

Type inference

The result of a comparison depends on the inferred type, as for example strings comparisons are different to integer comparisons. The rules are simple:

1. Literals have no type and will inherit the type from the rest of the expression.
2. Kernel symbols and special variables have fixed types.
3. If no type can be inferred, string comparison will be used (e.g. when comparing two literals).
4. Variables of different types cannot be mixed.

Special variables

There are several special variables which you can use in comparison expressions. They must be used in unquoted form and will allow you to depend on runtime information.

Variable	Type	Description
`$kernel_version`	semver	Expands to the semver of the specified kernel
`$uname_arch`	string	The uname as reported by `uname -m`
`$arch`	string	The architecture as seen by the kernel internally
`$false`	tristate	Always `'n'`
`$true`	tristate	Always `'y'`
`$env[VAR]`	string	Environment variable `VAR`, throws an error if unset
`$env[VAR:<quoted_str>]`	string	Environment vairable `VAR` or the given default

Comparison types

These are the existing comparison types:

Type	Description
`string`	Lexicographical comparison
`int`	Integer comparison, base 10
`hex`	Integer comparison, base 16, requires `0x` prefix
`tristate`	Same as for string, but arguments are restricted to `n`, `m` or `y`
`semver`	Semantic versioning comparison

Note: The format for semver versions is `major[.minor[.patch[-ignored]]]`. Missing parts are treated as `0`, which makes `4` equal to `4.0.0`.

Valid expression examples

Comparison expression	Type
`SOME_STRING == abc`	string
`SOME_STRING == "abc"`	string
`SOME_INT < 1`	int
`SOME_INT < "1"`	int
`SOME_HEX == 0x1`	hex
`SOME_TRISTATE == 'n'`	tristate
`SOME_TRISTATE == 'm'`	tristate
`SOME_TRISTATE == 'y'`	tristate
`12345 != 12`	string
`$env[CC] == "gcc"`	string
`$kernel_version >= 5.6`	semver
`SOME_TRISTATE`	tristate (implicit bool conversion)
`SOME_STRING`	string (implicit bool conversion)
`$env[HOSTNAME]`	string (implicit bool conversion)

Invalid expression examples

Comparison expression	Type and reason for invalidity
`SOME_STRING <= "abc"`	string, invalid operator for string
`SOME_STRING < 1`	string, invalid operator for string
`SOME_HEX > 1`	hex, invalid prefix
`SOME_INT == SOME_HEX`	cannot mix types
`$kernel_version >= SOME_INT`	cannot mix types
`SOME_HEX`	hex, invalid implicit bool conversion

Implicit bool conversion

All kernel symbols, all tristate special variables and `$env[...]` variables can implicitly be converted to bool. This means you can use these short forms:

```
# Same as X86 != n (X86 is a tristate symbol)
if X86 { ... }

# Same as DEFAULT_HOSTNAME != "" (DEFAULT_HOSTNAME is a string symbol)
if DEFAULT_HOSTNAME { ... }

# Same as $false != n
if $false { ... }

# Same as $env[TEST] != ""
if $env[TEST] { ... }
```

5.3.4 Short-circuiting (early-out)

All expressions support short-circuiting. The main reason is for this is that it allows conditional pinning. Consider the symbol `USB4`, which was first introduced in kernel version `5.6`. Just writing the conditional block

```
if USB4 {
    # ...
}
```

would fail on all kernels older than `5.6`, since the symbol `USB4` does not exist and therefore will raise an error in the expression. But if you change the statement to

```
if $kernel_version >= 5.6 and USB4 {
    # ...
}
```

the short circuiting of the expression will prevent the `USB4` part from being evaluated when the kernel version constraint is not met. This allows to maintain compatibility to several kernel versions.

CHAPTER 6

Configuration Directives

This is a documentation of all configuration directives.

6.1 global

The following block directives may appear in the unnamed global scope:

Block	Description
module	See *module*.
kernel	See *kernel*.
initramfs	See *initramfs*.
build	See *build*.
install	See *install*.

Additionally the following statements may be used:

6.1.1 include_module

`include_module <path>`

Arguments:

`path`: The path to the `.conf` file

Include module definitions from a single file. Path can be absolute, or relative to the current working directory when executing the script. Using absolute paths is recommended, but relative paths can be beneficial when testing different configurations. All module files should end with the `.conf` extension.

Example:

```
include_module "/usr/share/autokernel/modules.d/security.conf";
```

6.1.2 include_module_dir

`include_module_dir <path>`

Arguments:

`path`: The path to the directory with `.conf` files

Include module definitions from all `.conf` files in the given folder. Path can be absolute or relative.

Example:

```
include_module_dir "/etc/autokernel/modules.d";
```

6.2 module

`module <name> { ... }`

Arguments:

`name`	The module name

Defines a new module. Definition order is not important. Modules can be included in other modules to provide a level of encapsulation for different tasks. See *Concepts → Modules* and *Concepts → Pinning symbol values* for more information.

Example:

```
module example {
    use example_dep;

    set EXAMPLE y;
}
```

6.2.1 if

`module :: if <expr> { ... } [else if <expr> { ... }]... [else <expr> { ... }]`

Arguments:

`expr`	Expressions

Guards statements with the given expressions.

Example:

```
module example {
    if X86 {
        # X86 is set
    } else if $env[CC] == "gcc" {
        # env var CC is "gcc"
    }

    if $env[HOSTNAME:""] {
        # env var HOSTNAME is set
    } else {
```

(continues on next page)

(continued from previous page)

```
        # env var HOSTNAME is empty or unset
    }
}
```

6.2.2 use

module :: use <modules>... [if <cexpr>]

Arguments:

`modules`	A list of modules to include
`cexpr`	Attached *condition*

Include one or multiple a modules at this point. Referenced modules do not need to be defined before usage, as definition order is not important.

If a module has already been included before, it will be skipped. Modules will be included in the order they are encountered in use statements. Due to skipping, cyclic and duplicate inclusions are impossible. This statement may occurr multiple times.

Example:

```
module example {
    use foo;
    use other_example module_three;
}
```

6.2.3 set

module :: [try] set <symbol> [value] [if <cexpr>]

Arguments:

`symbol`	Kernel symbol name, the `CONFIG_` prefix is optional but discouraged.
`value`	The new value for the symbol (or `y` by default)
`cexpr`	Attached *condition*

Sets the value of a symbol. Omitting the value will default to setting the symbol to `y`. Prefixing symbol names with `CONFIG_` is allowed, but considered bad style.

Valid values for tristate symbols are `y` (yes), `m` (as module) and `n` (no). Symbols are always assigned by string, but restrictions for type conversion apply (e.g. integer symbols will only take valid integers).

Variables can be set to environment variables by using the same syntax as described in *Special variables*.

If the statement is prefixed with `try`, it will only be executed if the value is not already pinned, and the assignment will also not cause the value to be pinned. Useful to set a new default value for a symbol but still allowing the user to change it.

Repeated assignments of the same symbol are valid, as long as the same value is assigned each time, or the assignment uses the `try set`. Conflicts will cause hard errors.

Example:

```
module example {
    # Enable WIREGUARD if kernel version is at least 5.6
    set WIREGUARD y if $kernel_version >= 5.6;
    # Build KVM as module
    set KVM m;

    # Set a hex symbol
    set MAGIC_SYSRQ_DEFAULT_ENABLE 0x1;
    # Set an integer symbol
    set DEFAULT_MMAP_MIN_ADDR 65536;

    # Set a string symbol
    set DEFAULT_HOSTNAME "my_host";
    # Set to an environment variable, throws an error if unset
    set DEFAULT_HOSTNAME $env[HOSTNAME];
    # Set to an environment variable, or uses the default value if unset
    set DEFAULT_HOSTNAME $env[HOSTNAME:"(none)"];

    # Try to set MODULES, if it isn't pinned already
    try set MODULES n;
}
```

6.2.4 merge

`module :: merge <path> [if <cexpr>]`

Arguments:

`path`	The path to the kconf file
`cexpr`	Attached *condition*

Variables:

Allowed in `path`. See *Common variables*.

Merges an external kernel configuration file. This can be a whole .config file or just a collection of random symbol assignments (as it is the case for the defconfig files). All merged values will count as implicit changes (no pinning). They will trigger conflicts if a variable is already pinned.

Warning: Because of the implicit nature, the merge statement should only be used to include default values, and not to externalize parts of the config.

Example:

```
module example {
    # Merge the x86_64 defconfig file
    merge "{KERNEL_DIR}/arch/x86/configs/x86_64_defconfig";
}
```

6.2.5 assert

`module :: assert <aexpr> [: <quoted_message>] [if <cexpr>]`

Arguments:

aexpr	Expression to assert
quoted_message	An error message to display in case the assertion fails
cexpr	Attached *condition*

Asserts that a given expression evaluates to true, otherwise causes an error and optionally prints the given error message.

Example:

```
module example {
    # Assert that WIREGUARD is enabled if the kernel version is at least␣
↪5.6
    assert $kernel_version >= 5.6 and WIREGUARD :
        "Refusing to compile a 5.6 kernel without wireguard";
}
```

6.2.6 add_cmdline

module :: add_cmdline <quoted_args>... [if <cexpr>]

Arguments:

quoted_args	A list of strings to append to CMDLINE
cexpr	Attached *condition*

Adds the given parameters to the kernel commandline. Quotation is required. This will automatically set the CMDLINE symbol to the resulting string and enable the builtin commandline via CMDLINE_BOOL.

Example:

```
module example {
    # Adds the two strings to the builtin command line.
    add_cmdline "page_alloc.shuffle=1" "second_param";
}
```

6.3 kernel

kernel { ... }

A block for kernel related options. Multiple appearances of this block will be merged. The kernel block is also a *module* block. It represents the main module which is included by autokernel.

Example:

```
kernel {
    use hardening;
    use my_module;
}
```

6.4 initramfs

initramfs { ... }

A block for initramfs related options. Multiple appearances of this block will be merged.

Example:

```
initramfs {
    enabled true;
    builtin true;
}
```

6.4.1 enabled

initramfs :: enabled <bool>

Arguments:

`bool`	A *boolean value*

Default: `false`

Enables or disables building an initramfs. When using autokernel to build the kernel.

Example:

```
# Enable the initramfs
enabled true;
```

6.4.2 builtin

initramfs :: builtin <bool>

Arguments:

`bool`	A *boolean value*

Default: `false`

This will determine if the initramfs will be integrated into the kernel. It will cause an automatic second kernel build pass, to first allow the initramfs to include any modules for the newly built kernel, and secondly to pack the initramfs into the kernel. The second build will not require any rebuilds of previously compiled components, and should thus be quick.

Example:

```
# Use a builtin initramfs
builtin true;
```

6.4.3 build_command

initramfs :: build_command <exe> [<args>...]

Arguments:

`exe`	The command to execute
`args`	parameters to the command

Default: `None`

Variables:

Allowed in `exe` and `args`.

- Any of the *Common variables*
- `{MODULES_PREFIX}`

  A directory which contains all installed modules. This means the subdirectory `{MODULES_PREFIX}/lib/modules` exists and can be used by the initramfs generator to include compiled modules for the new kernel.

- `{INITRAMFS_OUTPUT}`

  The desired output file for the initramfs. If your generator doesn't support this, you can specify an alternate location with *build_output*.

Specifies the command used to build the initramfs. The resulting initramfs should directly be placed at `{INITRAMFS_OUTPUT}`. If your generator does not support this, you can fallback to the *build_output* statement to specify where the finished initramfs will be.

Note: Each string in `<args>` is a separate argument to the command, and arguments will never be interpreted or split on spaces. If you need more logic here, please execute a wrapper script to do so.

This statement is required, if the initramfs is enabled.

Example:

Listing 1: Building an initramfs with dracut

```
# You can use a command like this to build an initramfs with dracut
build_command "dracut"
    "--conf"          "/dev/null" # Disables external configuration
    "--confdir"       "/dev/null" # Disables external configuration
    "--kmoddir"       "{MODULES_PREFIX}/lib/modules/{KERNEL_VERSION}"
    "--kver"          "{KERNEL_VERSION}"
    "--no-compress"   # Only if the initramfs is to be integrated into␣
↪the kernel
    "--no-hostonly"
    "--ro-mnt"
    "--add"           "bash crypt crypt-gpg"
    "--force"         # Overwrite existing files
    "{INITRAMFS_OUTPUT}";
```

Listing 2: Building an initramfs with genkernel

```
# You can use a command like this to build an initramfs with genkernel
build_command "genkernel"
    "--module-prefix=${MODULES_PREFIX}"
    "--cachedir=/tmp/genkernel/cache"
    "--tmpdir=/tmp/genkernel"
    "--logfile=/tmp/genkernel/genkernel.log"
    "--kerneldir={KERNEL_DIR}"
    "--no-install"
    "--no-mountboot"
    "--no-compress-initramfs"
    "--no-ramdisk-modules"
```

(continues on next page)

(continued from previous page)

```
    "--luks"
    "--gpg"
    "initramfs";
build_output "/tmp/genkernel/initramfs-{UNAME_ARCH}-{KERNEL_VERSION}";
```

6.4.4 build_output

`initramfs :: build_output <path>`

Arguments:

`path`	The path where the finished initramfs will be

Default: `None`

Variables:

Same as for *build_command*.

Optional. Specifies where the output from the initramfs build command will be. You do not need to specify this, if your generator placed the initramfs at location given via `{INITRAMFS_OUTPUT}`.

6.5 build

`build { ... }`

A block for build related options. Multiple appearances of this block will be merged.

Example:

```
build {
    umask 0077;
}
```

6.5.1 umask

`build :: umask <value>`

Arguments:

`value`	Octal umask value to use

Default: `0077`

Specifies the umask used while building the kernel and the initramfs.

Note: If you are tempted to set this to 022 (allow read for others), you should probably rethink your build process. This can expose valuable information about your kernel to other users and renders some hardening methods useless.

Example:

```
build {
    # Set umask to 0027.
    umask 0027;
}
```

6.5.2 hooks

`build :: hooks { ... }`

Default: `None`

See *hooks* for more information. Specifies hooks for the build phase.

Example:

```
build {
    hooks {
        pre "echo" "pre-build";
    }
}
```

6.6 install

`install { ... }`

A block for options related to target installation. Multiple appearances of this block will be merged.

Example:

```
install {
    # Disable initramfs installation
    target_initramfs false;
}
```

6.6.1 umask

`install :: umask <value>`

Arguments:

`value`	Octal umask value to use

Default: `0077`

Specifies the umask used while installing files.

Example:

```
install {
    # Set umask to 0027.
    umask 0027;
}
```

6.6.2 assert_mounted

install :: assert_mounted <path>

Arguments:

`path`	The directory to assert is mounted

Asserts that the given directory is a mountpoint. Otherwise, autokernel will abort installation.

Example:

```
install {
    # Abort installation if /boot is not mounted
    assert_mounted "/boot";
}
```

6.6.3 mount

install :: mount <path>

Arguments:

`path`	The directory to mount

Temporarily mounts the given directory. Will be unmounted after installation, in case it had to be mounted. Requires an fstab entry for the directory. Autokernel will abort if the directory could not be mounted. If you use this, an additional *assert_mounted* entry is unnecessary.

Example:

```
install {
    # Mount /boot before installation
    mount "/boot";
}
```

6.6.4 modules_prefix

install :: modules_prefix <path>

Arguments:

`path`	The prefix path for `make modules_install`

Default: `/`

Variables:

Allowed in `path`. See *Common variables*.

The prefix path for `make modules_install`. This must an absolute path. Installation can be disabled by setting this to a false *boolean value*.

Example:

```
install {
    # Install into '/' (default)
    modules_prefix "/";

    # Disable installing modules
    modules_prefix false;
}
```

6.6.5 target_dir

install :: target_dir <path>

Arguments:

path	The target directory when installing files

Default: `/boot`

Variables:

Allowed in `path`. See *Common variables*.

The target installation directory. All other `target_*` statements will be relative to this directory. Must be an absolute path.

Example:

```
install {
    # Proper target directory for an efi partition mounted in /boot/efi
    target_dir "/boot/efi/EFI";
}
```

6.6.6 target_kernel

install :: target_kernel <path>

Arguments:

path	The kernel target path

Default: `bzImage-{KERNEL_VERSION}`

Variables:

Allowed in `path`. See *Common variables*.

The target path for the kernel image. This is relative to *target_dir*, but may also be an absolute path if desired. Installation can be disabled by setting this to a false *boolean value*.

Example:

```
install {
    # Don't include version number and use .efi suffix
    target_kernel "bzImage.efi";
    # Disable installing the kernel image
    target_kernel false;
}
```

6.6.7 target_config

install :: target_config <path>

Arguments:

`path`	The config target path

Default: `config-{KERNEL_VERSION}`

Variables:

Allowed in `path`. See *Common variables*.

The target path for a backup of the generated config. This is relative to *target_dir*, but may also be an absolute path if desired. Installation can be disabled by setting this to a false *boolean value*.

Example:

```
install {
    # Disable installing the config
    target_config false;
}
```

6.6.8 target_initramfs

install :: target_initramfs <path>

Arguments:

`path`	The initramfs target path

Default: `initramfs-{KERNEL_VERSION}.cpio`

Variables:

Allowed in `path`. See *Common variables*.

The target path for the initramfs image. This is relative to *target_dir*, but may also be an absolute path if desired. Installation can be disabled by setting this to a false *boolean value*. This option only has an effect if the initramfs is enabled.

Example:

```
install {
    # Disable installing the initramfs image
    target_initramfs false;
}
```

6.6.9 keep_old

install :: keep_old <number>

Arguments:

`number`	Number of old builds to keep

Default: `-1` (disable purging)

Automatic purging of old files. Determines the amount of old installed files to keep. Only has an effect on `target_dir` and `target_*` if `{KERNEL_VERSION}` is used in the path. A negative value like `-1` disables purging completely, which is the default.

Warning: Purging is done immediately after installing a file. The `{KERNEL_VERSION}` token will be replaced in all paths with a semver wildcard. All matching paths older than the given amount of builds will be removed.

Example:

```
install {
    # Keep previous two builds, purge the rest
    keep_old 2;
}
```

6.6.10 hooks

`install :: hooks { ... }`

Default: `None`

See *hooks* for more information. Specifies hooks for the install phase.

Example:

```
install {
    hooks {
        pre "echo" "pre-install";
    }
}
```

6.7 hooks

`hooks { ... }`

A block for hooks. Multiple appearances of this block will be merged. Specifies pre and post hooks for the phase in which the block is included.

Example:

```
hooks {
    pre  "echo" "pre-hook";
    post "echo" "post-hook";
}
```

6.7.1 pre

`hooks :: pre <exe> [<args>...]`

Arguments:

`exe`	The command to execute
`args`	parameters to the command

Default: `None`

Variables:

Allowed in `exe` and `args`. See *Common variables*.

Optional. Defines a pre hook. If the hook returns an unsuccessful exit code, autokernel will abort.

Note: Each string in `<args>` is a separate argument to the command, and arguments will never be interpreted or split on spaces. If you need more logic here, please execute a wrapper script to do so.

Example:

```
hooks {
    pre "echo" "pre-hook";
}
```

6.7.2 post

`hooks :: post <exe> [<args>...]`

Arguments:

`exe`	The command to execute
`args`	parameters to the command

Default: `None`

Variables:

Allowed in `exe` and `args`. See *Common variables*.

Optional. Defines a post hook. If the hook returns an unsuccessful exit code, autokernel will abort.

Note: Each string in `<args>` is a separate argument to the command, and arguments will never be interpreted or split on spaces. If you need more logic here, please execute a wrapper script to do so.

Example:

```
hooks {
    post "echo" "post-hook";
}
```

CHAPTER 7

License

CHAPTER 8

Acknowledgments

I would like to especially thank the following projects and people behind them:

- kconfiglib for the awesome python library to load and process Kconfig files, whithout which this project would have been impossible.
- sympy for the sophisitcated symbolic logic solver
- lark for the great parsing library
- LKDDb for providing the awesome Linux Kernel Driver Database (which is used for option detection)
- KSSP for the great list of kernel hardening options
- CLIP OS for their well documented and well chosen kernel options
- kconfig-hardened-check for the collection of options from several kernel hardening resources