A Quantitative Analysis of Warnings in Linux
Draft: Friday 28th August, 2015 19:41

Elvis Flesborg
efle@itu.dk

September 1st 2015
IT University of Copenhagen

Contents

Abstract

—Leave blank until the end—

Chapter 1

Introduction

Software projects with a high variability rate can be configured, to suit many needs with the same code base. Possibly the largest open source project, which also happens to be the project with the highest variability rate is the Linux kernel. It contains approximately 10,000 different configuration options in the feature model.

The basis for this paper is another paper: *42 Variability Bugs in the Linux Kernel: A Qualitative Analysis* [2], where bugs in the Linux kernel are qualitatively analyzed. In this report, warnings will be analysed quantitatively, and will function as a proxy for bugs.

The following contributions will be made: Analysis of distributions of warnings in the Linux kernel, comparisson of warnings in a stable version of the Linux kernel vs. an in-development version. Also an analysis of where the warnings are located.

Chapter 2

Background

2.1 Variability

Many software products are configurable in some way. Configurability creates the possibility of tailoring the software to suit different needs. For example different kinds of hardware, or different functionalities.

This is called *variability* in software, and a software product of this type is called a *Software Product Line* (SPL). Software products with different functionalities can be derived from the same source code base, and the code in its entirety is not a valid product [5, p. 1], it has to be configured.

Software with a high-degree of variability is usually refered to as *Variability-Intensive Systems* or *VISs*. Linux is a *VIS* with more than 14,000[1]different configuration options, called *features*. Other examples of *VISs* are BusyBox, Eclipse, Amazon Elastic Compute Service, and Drupal Content Management Framework [11, p. 1] to name a few.

All features in a software product line, and their options and relations are described in a *feature model*.

2.1.1 Feature Models

A feature model is a way of representing all the possible configurations - *the configuration space*. It contains all the features with their respective options and all the constraints and dependencies between the features.
A visualization of a feature model is called a feature diagram, there is an example in Figure 2.1. The example will be tiny compared to that of the Linux kernel. With many thousands of features, the feature model of the Linux kernel is too big to fit on a normal sized paper.
Figure 2.1 depicts a feature model of a phone configuration with 10 features, where some are mandatory: **Screen** and **Calls**, and some are optional: **GPS** and **Media**, which has 2 optional features **Camera** and **MP3** depending on it.
There is also some choice options **Color**, **BW**, and **High Definition** for the screen type, where only one of them may be enabled. **High Definition** is the default choice. A cross-tree constraint is also present, which states that **Media** with all of its children can only be enabled if a **High Definition** screen is enabled.
The feature diagram is inspired by [3], but there is no consensus on a unified notation for attributes in feature models [3]. And for there is for example no notation in the diagram, that

[1]14,387 across all architectures, with an average of 9,984 per architecture, and 10,335 for the `x86` architecture, which is used in this project.

High Definition is the default value for the choice clause.

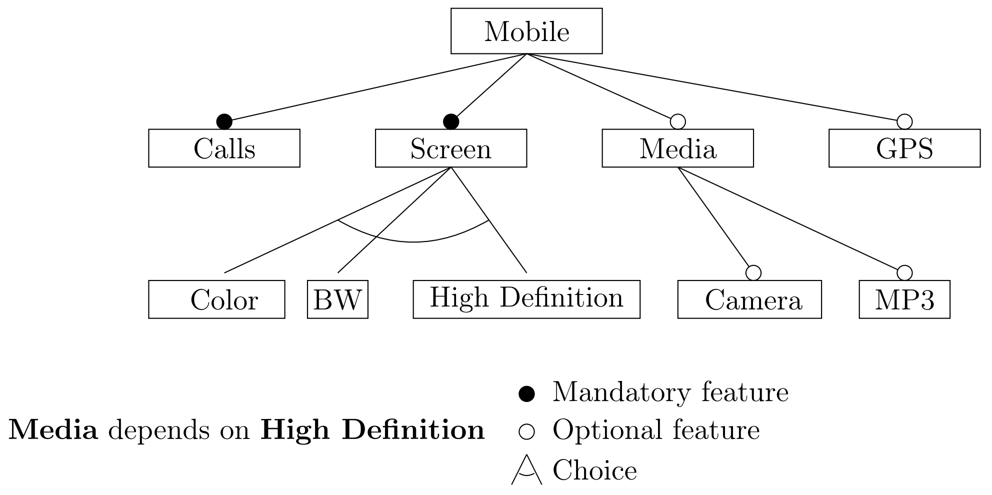


Figure 2.1: A feature diagram of a phone

The feature diagrams of *VIS*s are typically wide and shallow. The Linux kernel feature model hierarchy, for example, is only 8 layers deep [4, p. 17].

2.2 The Linux Kernel

The Linux kernel is written by many thousand of people all over the world, and has been ported to more than 20 architectures[2], which makes it very scalable, it is the operating system that supports the most hardware in the world [1, 7].
Its use cases range from small embedded devices like mobile phones and GPSs to supercomputers. In fact 98% of the top 500 supercomputers in the world run a Linux distribution [10].

It was first developed in 1991 by *Linus Torvalds*, and has since been growing. Today the code base is 19 million lines of code.

2.3 Linux Kernel Development

The Linux kernel development model is unique in the way, that it is developed by many thousands of people all over the world. According to [7], approximately 75% of the contributions come from companies, and 25% come from individuals.
There is a hierarchy of people, who are head-maintainers of different parts of the kernel[3].

Stable Releases

The Linux kernel development cycle has approximately 2¾ months from one stable release to the next stable release [6]. In the meantime, weekly semi-stable versions are released.
When the top maintainers of the mainline tree think that enough bugs have been fixed, a new stable version is created, and the whole process is repeated.

[2]See the `README` file in the Linux kernel tree
[3]See the `MAINTAINERS` file in the Linux kernel tree

In-development Releases

During development, new code is added to the in-development version of Linux. This version is called *linux-next*, and has been the main in-development version since 2008[4].

The *linux-next* tree is a GIT repository, which merges over 200 other GIT repositories [8], which are all based on the *mainline* tree. The *linux-next* tree is merging these other trees and the merge conflicts are handled every day.

This project uses both the *linux-next* tree and the latest stable version. They will be referred to as the **in-development** version, and the **stable** version. As time of experiment execution, the latest stable version is *4.1.1*.

2.4 Inner Workings of Linux Kernel

This section will explain, the structure of the Linux kernel, from the directory structure, over configuration, to compilation of the kernel.

2.4.1 Subsystems

The directories in the root folder of the Linux kernel source code are called *subsystems*, and they contain code for different purposes. Some are large crucial subsystems, some are smaller, and some are infrastructure subsystems, which contain scripts and tools for various uses [2].

The `drivers/` subsystem is by far the largest subsystem (with 57% of the total lines of code). It contains all device drivers. It is also mostly contributed to by hardware vendors, who want their hardware to be able to run on Linux.

The `arch/` subsystem (18%) contains architecture specific source code. There are 29 folders in the `arch/` subsystem. One for each major architecture that is supported. To highlight a few, the `x86` architecture is used for most common personal computers, the `arm` architecture is used mostly for mobile devices, and the `powerpc` architecture has been used for video game consoles. In this project, only the `x86` architecture will be used.

The `fs/` subsystem (6%) is code regarding filesystems, the `net/` subsystem (5%) is about networking, and the `mm/` subsystem (.6%) is about memory management.
Then there are `sound/`(5%), `include/` (4%), `kernel/` (1%), `crypto/` (.4%), `security/` (.4%), and `block/` (.2%).

Other smaller subsystems are:
`virt/`, `ipc/`, `init/`, `usr/`, and `lib/`.

And infrastructure subsystems are:
`tools/`, `scripts/`, and `samples/`.

2.4.2 The KCONFIG language

KCONFIG is the language of the feature model in Linux (also used for other projects like BUSYBOX, BUILDROOT, COREBOOT, FREETZ and others) [4, p. 4].
The KCONFIG files have the prefix `Kconfig`, and are scattered all over the Linux kernel source code tree, where they include each other. There are 1195 KCONFIG files in total in the stable

[4]See the original post about it here: https://lkml.org/lkml/2008/2/11/512

Linux kernel[5]with 956 of them relevant for the `x86` architecture.

The corresponding KCONFIG code for the phone example in Figure 2.1 can be seen in Figure 2.2. It is a simple example, which lacks some of the possible data types, and other syntax available in KCONFIG.
For a description of the context free grammar of the KCONFIG language, see the Appendix 8.1.

The different data types in the Kconfig language used in Linux are `boolean` (35%)[6], `tristate` (61%), `string` (0.41%), `hex` (0.32%), and `integer` (3.7%). `tristate` is a datatype with three possible values: `y`,`n`, and `m`, where `m` will set the feature as a module instead of building it into the kernel. It can then be loaded into the kernel when installing and running the kernel.
This does not mean that `m` and `n` will be the same in this project.

```
config CALLS
    def_bool y
config SCREEN
    def_bool y

choice
    prompt "Choose screen type"
    default HD
    depends on SCREEN
    config COLOR
        bool "Color screen"
    config BW
        bool "Black and white screen"
    config HD
        bool "High Definition screen"
endchoice

config GPS
    bool "GPS location system"
config MEDIA
    bool "Media modules"
    depends on HD

if MEDIA
    config CAMERA
        bool "Camera support"
    config MP3
        bool "MP3 support"
endif
```

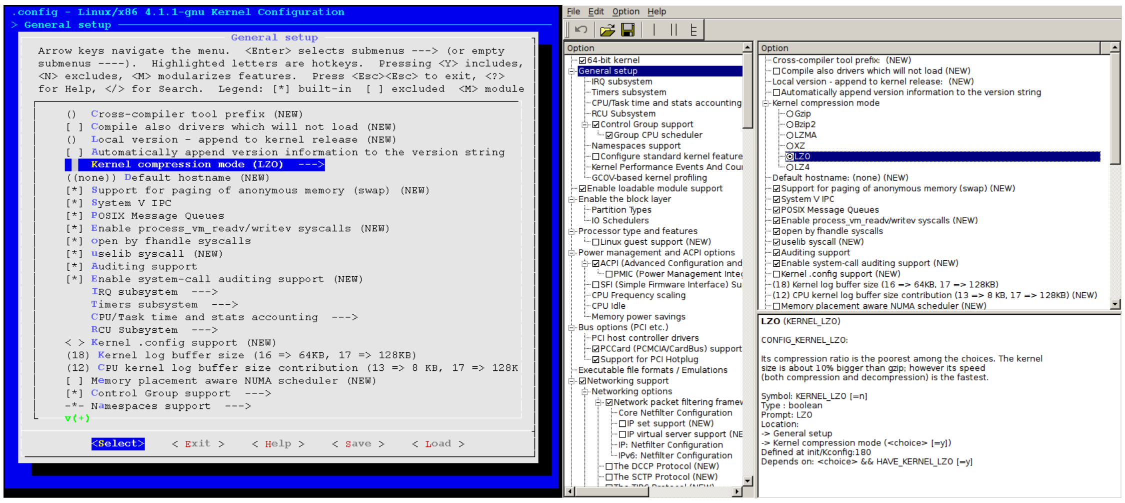
Figure 2.2: KCONFIG code for the phone example

2.4.3 Configuring Linux

The Linux kernel comes with different ways of choosing features - these are called *configurators*. Some of them lets the user choose the features. There is a question based one: `config`, and some menu based ones: `menuconfig`, `xconfig`, `nconfig`, and `gconfig`.

Figure 2.3 shows what the graphical configurators `menuconfig` and `xconfig` look like.

[5]Found with the command `find . | grep Kconfig | wc`
[6]The percentages are of the 10,335 features available for the `x86` architecture



```
.config - Linux/x86 4.1.1-gnu Kernel Configuration
> General setup
                              General setup
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes,
  <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module

      ()   Cross-compiler tool prefix (NEW)
      [ ]  Compile also drivers which will not load (NEW)
      ()   Local version - append to kernel release (NEW)
      [ ]  Automatically append version information to the version string
           Kernel compression mode (LZO)  --->
      ((none)) Default hostname (NEW)
      [*]  Support for paging of anonymous memory (swap) (NEW)
      [*]  System V IPC
      [*]  POSIX Message Queues
      [*]  Enable process_vm_readv/writev syscalls (NEW)
      [*]  open by fhandle syscalls
      [*]  uselib syscall (NEW)
      [*]  Auditing support
      [*]  Enable system-call auditing support (NEW)
           IRQ subsystem  --->
           Timers subsystem  --->
           CPU/Task time and stats accounting  --->
           RCU Subsystem  --->
      < >  Kernel .config support (NEW)
      (18) Kernel log buffer size (16 => 64KB, 17 => 128KB)
      (12) CPU kernel log buffer size contribution (13 => 8 KB, 17 => 128K
      [ ]  Memory placement aware NUMA scheduler (NEW)
      [*]  Control Group support  --->
      -*-  Namespaces support  --->
      v(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

```
File  Edit  Option  Help

Option
 ☑64-bit kernel
 General setup
   IRQ subsystem
   Timers subsystem
   CPU/Task time and stats accounting
   RCU Subsystem
   ☑Control Group support
     ☑Group CPU scheduler
   Namespaces support
   ☐Configure standard kernel feature
   Kernel Performance Events And Cou
   GCOV-based kernel profiling
 ☑Enable loadable module support
 Enable the block layer
   Partition Types
   IO Schedulers
 Processor type and features
   ☐Linux guest support (NEW)
 Power management and ACPI options
   ☑ACPI (Advanced Configuration and
     ☐PMIC (Power Management Inte
   ☐SFI (Simple Firmware Interface) Su
   CPU Frequency scaling
   CPU Idle
   Memory power savings
 Bus options (PCI etc.)
   PCI host controller drivers
   ☑PCCard (PCMCIA/CardBus) support
   ☑Support for PCI Hotplug
 Executable file formats / Emulations
 ☑Networking support
   Networking options
     ☑Network packet filtering framev
       Core Netfilter Configuration
       ☐IP set support (NEW)
       ☐IP virtual server support (NE
       IP: Netfilter Configuration
       IPv6: Netfilter Configuration
     ☐The DCCP Protocol (NEW)
     ☐The SCTP Protocol (NEW)

Option
 Cross-compiler tool prefix:  (NEW)
 ☐Compile also drivers which will not load (NEW)
 Local version - append to kernel release:  (NEW)
 ☐Automatically append version information to the version string
 Kernel compression mode
   ○Gzip
   ○Bzip2
   ○LZMA
   ○XZ
   ◉LZO
   ○LZ4
 Default hostname: (none) (NEW)
 ☑Support for paging of anonymous memory (swap) (NEW)
 ☑System V IPC
 ☑POSIX Message Queues
 ☑Enable process_vm_readv/writev syscalls (NEW)
 ☑open by fhandle syscalls
 ☑uselib syscall (NEW)
 ☑Auditing support
 ☑Enable system-call auditing support (NEW)
 ☐Kernel .config support (NEW)
 (18) Kernel log buffer size (16 => 64KB, 17 => 128KB)
 (12) CPU kernel log buffer size contribution (13 => 8 KB, 17 => 128KB) (NEW)
 ☐Memory placement aware NUMA scheduler (NEW)

LZO (KERNEL_LZO)

CONFIG_KERNEL_LZO:

Its compression ratio is the poorest among the choices. The kernel
size is about 10% bigger than gzip; however its speed
(both compression and decompression) is the fastest.

Symbol: KERNEL_LZO [=n]
Type : boolean
Prompt: LZO
Location:
-> General setup
-> Kernel compression mode (<choice> [=y])
Defined at init/Kconfig:180
Depends on: <choice> && HAVE_KERNEL_LZO [=y]
```

Figure 2.3: `menuconfig` and `xconfig`

Other configurators will never prompt the user for anything, but create a configuration automatically:

- `allyesconfig` (enabling as much as possible)
- `allnoconfig` (disabling as much as possible)
- `tinyconfig` (same as `allnoconfig` but with higher compression rate, to fit on smaller devices.
- `defconfig` (choosing the default values for everything)
- `randconfig` (choosing random values for everything).

When a configuration file is generated, it is saved as `.config`. In this file, all features are prefixed with `CONFIG_`, and this is the configuration of the Linux kernel.
Running the configurator `allnoconfig` on the KCONFIG code for the phone example will disable all the features in the feature model that can be disabled. Figure 2.4(a) shows the `.config` file that will be generated when running `allnoconfig`.

The `CALLS` and `SCREEN` are mandatory, and will therefore be enabled. Out of the three possibilities in the choice clause, the `HD` feature has been enabled, since it is the default choice. Everything else is disabled.

```
CONFIG_CALLS=y
CONFIG_SCREEN=y
# CONFIG_COLOR is not set
# CONFIG_BW is not set
CONFIG_HD=y
# CONFIG_GPS is not set
# CONFIG_MEDIA is not set
```

(a) allnoconfig

```
CONFIG_CALLS=y
CONFIG_SCREEN=y
# CONFIG_COLOR is not set
# CONFIG_BW is not set
CONFIG_HD=y
CONFIG_GPS=y
CONFIG_MEDIA=y
CONFIG_CAMERA=y
CONFIG_MP3=y
```

(b) allyesconfig

Figure 2.4

Figure 2.4(b) shows the `.config` file that is generated by the configurator `allyesconfig`. Every feature has been enabled, except `COLOR` and `BW`, which must be disabled because only one feature within the choice clause can be enabled.

Running `allnoconfig` on the Linux kernel, will enable 228 features and `allyesconfig` will enable 6976 features.

2.4.4 Compiling and Catching Warnings

When the Linux kernel has been configured, it is ready to be compiled to create the executable kernel. The compiler will then only include the code parts, which are indicated by the configuration file. This is written in the source code as `#ifdef CONFIG_FOO`.
When writing code for variability software, a programmer has to be aware of the `#ifdef` clauses, and that different configurations can exist, which will only enable some of the code. A programmer can fail to check all possible configurations when writing the code, and accidentally make coding mistakes.

When compiling the Linux kernel, the command `make all` is run, and this will start the compilation. If any warnings or errors happen, they will be posted in the standard output in the terminal.

In a warning output, there is information about a possible coding mistake that might break the code. There are 100s of different warning types, that can be enabled. 33 of the these can be enabled by giving the `-Wall` flag to the compiler GCC [13]. This will enable warnings *"about contructions that some users consider questionable"* [13].

2.4.5 GCC Warnings

All experiments are run with the GCC-flag `-Wall`, which is a group of 33 warnings. In many cases, though, the Linux kernel enables even more warning flags by itself, so it is not expected to only get warnings from the `-Wall` group.

The warnings will be categorized into these types of warnings:

1. **Wrong value warnings**

   These warnings will have a chance of breaking the code by returning a wrong value or doing wrong logic.

2. **Code pollution warnings**

   These warnings will not break the code, or return wrong values, merely be unused code. This plays a role in fitting the Linux kernel on devices with space limitations (such as embedded devices).

   It can confuse other programmers, who think the code is being used.

3. **Bad code practice warnings**

   These warnings are deemed unsevere. This does not mean, that they can not be severe in other projects, it is usually a heads up to the programmer, that some bad practices have been used.

   This can also contribute to confusing programmers, who might misunderstand the logic.

4. **Irrelevant warnings**

   These are unsevere warning types.

Here follows a list of the different warning types that was found during the experiment. They are ordered alphabetically, and when code snippets from the Linux kernel are present, they are simpilified.

array-bounds

This warning is given, when GCC is certain that a subscript to an array is always out of bounds.
This will be categorized as a *wrong value* warning.

cpp

This shows `#warning` directives written in the code. This is a warning message that the coder can pass to the compilee. Often they will not result in an error, but inform about specifics.
This is categorized as *irrelevant*.

deprecated-declarations

This will show a warning when a function is used, which a programmer has declared deprecated. This typically means that a function is run, which is old and has been replaced - or has not yet been replaced - by another function.
This is categorized as a *wrong value* warning.

discarded-array-qualifiers

error=

This is a prefix, which can be put in front of all warning types. It will treat the warning as an error, because the developers do not want a specific warning to happen without the compilation stopping.
This is an error, and will be categorized as *irrelevant*.

frame-larger-than=

implicit-function-declaration

This is given, when a a function has not been declared, but is being used.
This is categorized as *bad code practice*.

incompatible-pointer-types

This shows when there is converted between pointers, which have incompatible types.
This is categorized as a *wrong value* warning.

int-conversion

This warns about incompatible conversions between integers and pointers.
This is categorized as *wrong value*.

int-to-pointer-cast

This warns about an integer being cast to a pointer with a different size.
This will be categorized as *wrong value*.

logical-not-parentheses

This warning occurs when the left hand side of an expression contains a *logical not* (a `!` in C). An example is `if (!ret == template[i].fail)` from the file `crypto/testmgr.c`.
The statement can be interpreted in these 2 ways, which do not necessarily mean the same in all cases:

- `! (ret == template[i].fail)`
- `(!ret) == template[i].fail`

This is categorized as *wrong value*.

maybe-uninitialized

This warning shows when there is an uncertainty about a variable being uninitialized. In the case in Figure 2.5 there is a `switch-case` where the variable `sgn` is not initialized in all of the cases.
If GCC cannot see for sure that the variable is initialized, it will return this warning.

```
static int __add_delayed_refs()
{
    int sgn;

    switch (node->action) {
    case BTRFS_ADD_DELAYED_REF:
        sgn = 1;
        break;
    case BTRFS_DROP_DELAYED_REF:
        sgn = -1;
        break;
    default:
        BUG_ON(1);
    }
    *total_refs += (node->ref_mod * sgn);
}
```

Figure 2.5: A real example of maybe-uninitialized function

This is categorized as *wrong value*.

overflow

This is when an integer is truncated into an unsigned type. This can lead to wrong data, and is categorized as a *wrong value* warning.

pointer-to-int-cast

This warns about a pointer being cast to an integer with a different size.
This will be categorized as *wrong value*.

return-type

This will be shown when there is a non-void function, which has no return statement.
This is categorized as a *bad code practice* warning.

switch-bool

This warning is shown when a `boolean` is used in a `switch` statement. When dealing with a `boolean`, an `if` statement should suffice, and this warning might indicate that the wrong variable is used.
This is categorized as *wrong value*.

uninitialized

This warns about uninitialized variables. The variable has been declared, but has not yet been given a value.
This is categorized as a *wrong value* warning.

unused-function

This is a warning about a function, which has been declared and initialized, but has never been called.
This is categorized as *code pollution*.

In the example in Figure 2.6, the function `bq27x00_powersupply_unregister` will only be called if the feature `CONFIG_BATTERY_BQ27X00_I2C` is enabled, and is therefore an unused function.

```
static void bq27x00_powersupply_unregister(struct bq27x00_device_info *di) {
    poll_interval = 0;
    cancel_delayed_work_sync(&di->work);
    power_supply_unregister(di->bat);
    mutex_destroy(&di->lock);
}

#ifdef CONFIG_BATTERY_BQ27X00_I2C

static int bq27x00_battery_remove(struct i2c_client *client) {
    struct bq27x00_device_info *di = i2c_get_clientdata(client);
    bq27x00_powersupply_unregister(di);
    return 0;
}

#endif
```

Figure 2.6: A real example of an unused function - from the file `drivers/power/bq27x00_battery.c`

unused-variable

This is the same as `unused-function`, but only with a variable instead of a function. There will be no example of this.
It is also categorized as *code pollution*.

unused-label

This is the same as the `unused-function` and `unused-variable` warnings with a label instead.
This is categorized as *code pollution*.

Chapter 3

Methodology

Objective: This report aims to make a quantitative analysis of warnings in all of the Linux kernel by checking randomly generated Linux kernels for warnings and then generalize.
This includes addressing the following research questions:

RQ1: What warnings are the most common in the stable Linux kernel?

RQ2: Where do most warnings occur?

RQ3: Are there any significant differences between an in-development version of Linux and a stable version?

Subject: To respond to these questions there will be generated random configurations, and these will be used to compile two different versions of the Linux kernel, the latest stable Linux kernel version, and a two months old in-development version of the Linux kernel.
The warning messages will be categorized and collected, and be subject for analysis.

Methodology: The methodology will be in three parts. First part is finding a way of generating random configurations in a representative way. Second part is collecting any warnings that a compilation might return. Third part is analyzing the data and answering the research questions.

For the configurations to be representative every configuration must be equally likely as others in the configuration space. But since there are more than 10,000 different features in the feature model of the Linux kernel, there will be approximately $2^{10,000}$ $(= 10^{3080})$ possible configurations in the configuration space, assuming the feature model is 100% `boolean`s and there are no cross-constraints. This is more than the estimated number of atoms in the universe, so getting a list of all the possible configurations is not possible (at least not with today's computers).

3.1 Experimental Setup

The experimental setup consists of a loop, where a configuration is generated, the Linux kernel is then compiling with this configuration, and the output warnings are categorized and collected. Notice that the Linux kernels are not installed and executed.

The experiment has mainly been carried out on a computer at the IT University of Copenhagen. The computer has $32 \times 2.8MHz$ and $128GB$ of RAM, and the average time to compile a kernel and upload the data was around 1 minute and 35 seconds. Also a conventional laptop with $4 \times 2.5MHz$ and 8 GB of RAM has been used.

To say something about all of the Linux kernel, the generated configurations for the experiment should be a representative sample of all the possible configurations.

3.2 Part 1: Choosing the Configurator

BLABLA about representativeness

In Figure 3.1 a blue area represents the whole configuration space, and red dots inside a blue area represent a subset of all the configurations, which is in a given sample.

Figure 3.1(a) shows a sample, which is represesentatively distributed over the configuration space.

Figure 3.1(b) shows a sample, which is not representatively distributed. All the dots tend to cluster together around certain areas of the configuration space, and if one was to tell what the configuration space looked like by only looking at the sample, it would look like the yellow area.

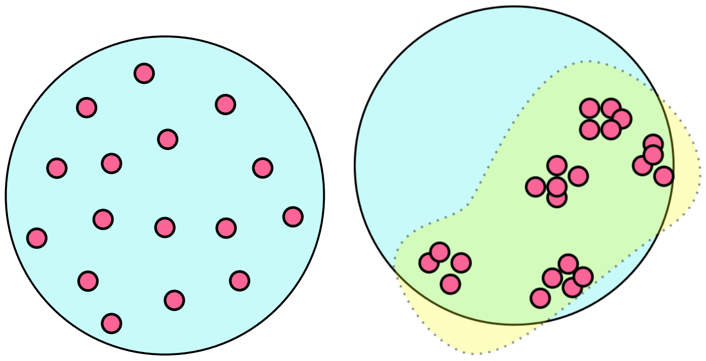

(a) A representative sample (b) An unrepresentative sample

Figure 3.1: Showing both representative (a) and unrepresentative (b) samples

Five different methods are proposed and discussed to generate random configurations.

1. **Using randconfig**

   Using the built-in `randconfig` configurator.

   This comes out of the box with Linux, and always generates valid configurations, and does so in a few seconds. A caveat is that it does not generate the configurations representatively. Read section 5.1.2 about how `randconfig` is not representative.

2. **Changing randconfig**

   This method will rewrite the code for `randconfig` or create a new configurator (eg. `reprandconfig`). This configurator must not have the bias for features in the upper levels of the dependency tree, as `randconfig` has.

   This would require good knowledge of programming in the C language and also an algorithmic way of estimating the number of features in all the subtrees.

3. **Permuting Kconfig**

   This proposal will desugarize the Kconfig files to be in one level only, by replacing `if FOO ... endif` clauses by `depends on` options in the features clauses, and then randomly scramble the position of the feature clauses in the files. After this, `randconfig` is run.

The hope is that the `randconfig` script loads the KCONFIG files from top to bottom, and will then load in the features at random each time.

4. **Generate and filter**

   In this method, a configuration file is generated with a script, which does not know the relation betweeen all the features. It knows all the feature names, and the possible values for the features, and is aware of the choice clauses.

   It goes through the list and randomly selects values for all the features. Then all invalid configurations are filtered away.

5. **RandomSAT**

   The Linux kernel feature model can be extracted from the KCONFIG files, to get a propositional formula [12]. This formula can be used to check for propositional satisfiability.

Out of these five methods, `randconfig` is selected as a proxy for getting a representative sample, on the grounds that the other methods either do not work as intended, or depends on too much time or expertise to be realistic in this project.

3.2.1 Compiling and Collecting Data

When a configuration has been generated, the Linux kernel is compiled using that configuration file by running the command `make all`. This command runs GNU MAKE, which is instructed how to compile the kernel with GCC.
The warning and error outputs from GCC will be saved for analysis.

A warning output will contain a *bug type*, a *filename*, *line number*, and a *message* describing the warning in english.

— blabla about how to categorize —

3.3 Analyzing Data

The warnings will be quantitatively analyzed. The analysis of the warnings is quantitative, and this project will not contain in-depth analysis of any of the warnings in comparisson to [2], which is a qualitative analysis of bugs in Linux.
The different warning types will be categorized according to section 2.4.5.

Chapter 4

Results

A total of 42,060 experiments were run. Half of them from the in-development Linux version, and the other half of the latest stable version of Linux.

A total of 506,000 bugs were found and 4,600,000 files.

4.1 Stable Linux Version

In Figure 4.3 is shown the distribution of warnings in the experiments run with the stable Linux kernel. The distinct warning types are only counted once per experiment run.
The warning categories are colored like this: Wrong data, Code pollution, Bad code practice, and irrelevant.

A total of 245,000 warnings were collected from these experiments, with the highest amount of warnings for a single experiment being 111. Many of the warnings found, are the same exact warnings, happening in the same files over and over. This is natural, since many different experiments are bound to create some of the same warnings, as it all comes from the same code base.

17% of the compilations stopped with an error. These errors were mostly errors, which were specific for the build machine because of missing libraries or programs, and will not be looked into. The compilations that had errors were made to stop after the first error was found to not have data pollution caused by an avalanche effect.

The following observations have been made:

Observation 1: The most common warnings in the *Wrong data* category are: x, x, and x.

Observation 2: The most common warnings in the *Code pollution* category are: x, x, and x.

Observation 3: The most common warnings in the *Bad code practice* category are: x, x, and x.

With these observations, **RQ1** can be answered.

Conclusion 1:

4.2 Subsystems

Figure 4.2 shows the distribution of subsystems with warnings in all of the compilations on the stable Linux version.

Warning	Percentage	Category
unused-function	59.%	Code pollution
maybe-uninitialized	45.%	Wrong data
logical-not-parentheses	30.%	Wrong data
unused-variable	29.%	Code pollution
cpp	24.%	Irrelevant
uninitialized	19.%	Wrong data
ERROR	17.%	Irrelevant
pointer-to-int-cast	17.%	Wrong data
discarded-array-qualifiers	17.%	xxx
switch-bool	15.%	Wrong data
frame-larger-than=	14.%	xxx
array-bounds	11.%	Wrong value
return-type	7.7%	Bad Code Practice
int-to-pointer-cast	7.6%	Wrong data
overflow	6.5%	Wrong value
unused-label	5.4%	Code Pollution
deprecated-declarations	5.4%	Wrong data
int-conversion	2.7%	Wrong data
implicit-function-declaration	5.6%	Bad code practice
incompatible-pointer-types	0.74%	Wrong data

Figure 4.1: Distribution of warnings in the stable kernel

In many compilation runs, there were multiple warnings, in the same subsystems. they are only counted once.
The green and blue rows are subsystems that are not a major part of the kernel functionality, as mentioned in Section 2.4.1.

With these results, the following observations can be made:

Observation 3: In 93% of the experiments, the `drivers/` subsystem was present in a warning.

Observation 4: There are near zero warnings in the `security/` subsystem. This can both be an indication of a small subsystem, but rather, that it is an important subsystem, which a lot of work are put into.

4.3 In-Development Version vs. Stable Version

The results show some differences, and the following observations can be made.
— table with comparisson —

Subsystem	Percentage
drivers/	93.%
include/	55.%
fs/	35.%
arch/	35.%
arch/x86/	35.%
kernel/	24.%
net/	18.%
crypto/	17.%
sound/	16.%
mm/	14.%
usr/	12.%
samples/	12.%
lib/	11.%
block/	8.5%
scripts/	1.8%
security/	0.091%
ipc/	0.0048%
init/	0.0048%
tools/	0.0048%
virt/	0.0%

Figure 4.2: Distribution of all subsystems within the warnings

Warning	Stable	In-Dev	Category
unused-function	59.%	62.%	Code pollution
maybe-uninitialized	45.%	44.%	Wrong data
logical-not-parentheses	30.%	29.%	Wrong data
unused-variable	29.%	51.%	Code pollution
cpp	24.%	22.%	Irrelevant
uninitialized	19.%	18.%	Wrong data
ERROR	17.%	38.%	Irrelevant
pointer-to-int-cast	17.%	17.%	Wrong data
discarded-array-qualifiers	17.%	17.%	xxx
switch-bool	15.%	15.%	Wrong data
frame-larger-than=	14.%	7.8%	xxx
array-bounds	11.%	10.%	Wrong value
return-type	7.7%	7.8%	Bad Code Practice
int-to-pointer-cast	7.6%	25.%	Wrong data
overflow	6.5%	6.0%	Wrong value
unused-label	5.4%	4.7%	Code Pollution
deprecated-declarations	5.4%	5.2%	Wrong data
int-conversion	2.7%	4.2%	Wrong data
implicit-function-declaration	5.6%	23.%	Bad code practice
incompatible-pointer-types	0.74%	3.2%	Wrong data

Figure 4.3: Distribution of warnings in the stable kernel

Chapter 5

Threats to Validity

5.1 External Validity

5.1.1 Generalization

All the selected configurations are not a representative subset of all possible configurations, and therefore does not fully represent all of the Linux kernel. The method by which the configurations are selected, does not select configurations between the whole configuration space equally.
There is a bias towards selecting configurations that will not visit the deepest layers of the dependency tree.

5.1.2 The Built-In `randconfig` Configurator

Unfortunately, the built-in `randconfig` is not representative, but is biased towards features higher up in the feature model tree. See Section 5 about Threats to Validity for more information about how `randconfig` is biased.
So, for the sample to be representative, another way to generate configurations must be found.

Figure 5.1 shows a toy example of a KCONFIG feature model written in the KCONFIG language. It is a very small example with only two features, but it will easily explain some limitations of using `randconfig`.

There are two features (`A` and `B`) in the example, which can be enabled or disabled, and feature `B` depends on `A` being enabled. This leaves three possible outcomes.
One where both is enabled, one where only `A` is enabled, and one where none of them are enabled. The outcome where only `B` is enabled is an invalid configuration since `B` depends on `A`.

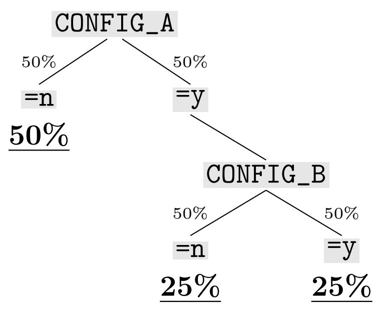
```
config A
    bool
config B
    bool
    depends on A
```

Figure 5.1: A toy KCONFIG feature model

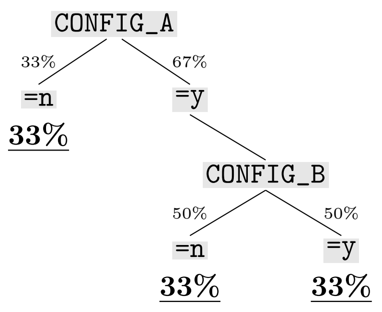
Figure 5.2(a) shows how `randconfig` will decide whether the features are enabled or disabled. It always goes from the top of the tree and down. So feature `A` will always be decided for at first. And since it is a `boolean` there will be a fifty fifty chance.

Then it proceeds further down in the dependency tree, and decides for feature `B`, which also has a fifty fifty chance.

For the creation to be representative, all the three possible configurations should have equal chance of being created (33%). See Figure 5.2(b) for a visualization of how the selection should be to be representative.



(a) How `randconfig` will select



(b) Representative selection

Figure 5.2

Representativeness is not of high prioritization for the Linux kernel developers, the `randconfig` function is merely used as a simple fuzz testing tool.
This ultimately means that `randconfig` is not representative in its configuration creation. (See more about representativeness in the section 3.2.)

5.2 Internal Validity

5.2.1 9 Different In-Development Versions

Multiple different in-development versions of the Linux kernel are used to minimize the skewing of warnings. If only one version of the in-development version is used, it gives a very one-sighted view of the in-development versions, where certain bugs may be over-represented. The more different versions of the in-development Linux should be used, to get a more uniform results.

5.2.2 More Features in the In-Development version

In the in-development Linux version, there are 20 more features than in the stable version. The configurations are created in the in-development version, and are copied over to the stable version. This will result in some unknown features being set on the stable version, but there is no code corresponding to the features, so no harm is done.
If they are created in the stable version, instead, they will never be given random values, but always default values, which will skew the results.

5.2.3 Gcc versions

Roughly a third of the compilations were done with GCC version 5.1.0 and two thirds with GCC version 4.9.2. This should not matter on what warnings are returned. There is only one new warning that is enabled by the `-Wall` flag in version 5.1.0 (`-Wc++14compat`), and none of this type was found.

There is a possibility though, that GCC has been improved to be better at finding certain warnings, and therefore the newer version will find more warnings that older version.

5.2.4 Firmware

When building certain firmware drivers in Linux, external proprietary drivers are needed, before they can be built. This firmware is not in the kernelcode, but must be downloaded from the hardware vendors homepages.

There are libraries on the internet which contain these firmware drivers, but in this report, they will not be included. These drivers are in a sense not a part of the open source Linux kernel, and are out of scope with this report.

To disable configurations, which would require these proprietary firmware drivers, the following features were given a fixed value in the configurations:

- `CONFIG_STANDALONE=y`
- `CONFIG_FW_LOADER=n`
- `CONFIG_PREVENT_FIRMWARE_BUILD=y`

Also every feature that had some relation to the Z4C library has been disabled, since it was not installed on the build system.
So all features, which were related to the Z4C library were also given a fixed value.

- `CONFIG_*LZ4*=n`

Furthermore this feature is fixed, since it is also dependent on a library not installed on the build system.

- `CONFIG_SECCOMP=y`

Chapter 6

Related Work

Variability bugs The paper *42 Variability Bugs in the Linux Kernel...* [**?**] is looking at bugs in the linux kernel from a qualitative angle rather than quantitative. The bugs are manually analysed, and discussed.
This report is somewhat a continuation of the research in that paper, trying to quantify the warnings part of the bugs.

Chapter 7

Conclusion

— Leave empty until the end—

Bibliography

[1] L. 22th Birthday Is Commemorated. http://www.cmswire.com/cms/information-management/linux-22th-birthday-is-commemorated-subtly-by-creator-022244.php, August 2013.

[2] I. Abal, C. Brabrand, and A. Wasowski, *42 variability bugs in the linux kernel: A qualitative analysis*. http://www.itu.dk/people/brabrand/42-bugs.pdf.

[3] D. Benavides, S. Segura, and A. Ruiz-Cortés, *Automated analysis of feature models 20 years later: A literature review*, University of Seville, (2010).

[4] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki, *Variability modeling in the systems software domain, version 2*, University of Waterloo, (2013).

[5] C. Brabrand, M. Riberio, T. Toledo, J. Winther, and P. Borba, *Intraprocedural dataflow analysis for software product lines*.

[6] L. kernel release prediction site. http://phb-crystal-ball.org.

[7] G. Kroah-Hartman, *Google tech talks 2008*. https://www.youtube.com/watch?v=L2SED6sewRw.

[8] linux next, *merge trees*. http://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git/tree/Next/Trees.

[9] G. packages. http://www.gnu.org/software/software.html.

[10] T. . S. Sites. http://top500.org/statistics/list, June 2015.

[11] A. B. Sánchez, S. Segura, J. A. Parejo, and A. Ruiz-Cortés, *Variability testing in the wild: The drupal case study*, xxx, (2015).

[12] W. University, *Linux variability analysis tools*. http://gsd.uwaterloo.ca/node/313.

[13] G. warning types. https://gcc.gnu.org/onlinedocs/gcc-5.1.0/gcc/Warning-Options.html#index-Wall-292.

Chapter 8

Appendices

8.1 KCONFIG language

Following is the KCONFIG language context free grammar in the *Backus Naur Form* inspired by the LUA documentation[1].

```
stat ::= 'config ' id '\n' options

id ::= characters { characters }

characters ::= [A-Za-z_0-9]\*

options ::= mandatory optional

types ::= 'bool' | 'boolean' | 'tristate' | 'int' | 'string' | 'hex'
deftypes ::= ( 'def_bool' | 'def_tristate' ) expr

mandatory ::= types [ '"' characters '"' ] [ expr ] |

optional ::= 'depends on ' expr
             'select ' expr
             'help\n' characters

expr ::= id [ '=' characters ]
         expr '&&' expr |
         expr '||' expr |
         '(' expr ')'
         'if ' expr
         'menu' text
         'y'
         'n'

text ::= characters | symbols
         text text

symbols ::= '/-(),.'
```

[1]http://www.lua.org/manual/5.1/manual.html

8.2 Data