A Quantitative Analysis of Warnings in Linux
Draft: Thursday 27th August, 2015 11:20

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 high variability rate can be configured to suit many needs with the same code base.
Possibly the largest open source code base which adapts the variability paradigm is the Linux kernel. It contains approximately 10,000 different configuration options.

The ground for this paper is somehow based on the paper *42 Variability Bugs in the Linux Kernel: A Qualitative Analysis* [2] , where bugs are qualitatively analyzed, not quantitatively. In this report, warnings will be used as a proxy for errors, and will be analysed quantitatively.

The following contributions will be made. Analysis of distributions of warnings in the Linux kernel, comparisson of warning distribution in a stable version of the Linux kernel and an unstable version. Also an analysis of which subsystems contain the most warnings.

Chapter 2

Background

A *Linux* operating system is often referring to a *GNU/Linux* operating system, where the Linux part of *GNU/Linux* is the Linux kernel, and the *GNU* part is a software bundle with utilities (eg. a shell, a compiler, etc...) [8]. Both is needed, to have a working operating system.
This report will only focus on the *Linux kernel*, and not the *GNU* bundle.

2.1 The Linux Kernel

The Linux kernel is written in by many thousand of people all over the world, and has been ported to more than 20 architectures, which makes it very scalable[1], it is the operating system that supports the most hardware in the world [1].
Its use case ranges from small embedded devices like mobile phones, GPSs to supercomputers. In fact 98% of the top 500 supercomputers in the world run a Linux distibution [**?**, 9].

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

2.2 Variability

Many software products are configurable in some way. This 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). When different software programs can be derived from the same source code base.
Before compilation, the source code will have to be configured, at a preprocessing step, which will choose (either automatically, or the user will choose) which parts of the code should be included, by enabling or disabling a set of *features*.
The code in its entirety is not a valid program, it must be preprocessed. [5, p. 1]

Software with a high-degree of variability is usually refered to as *Variability-Intensive Systems* or *VISs*. Linux is a *VIS* with more than 10,000 different features[2]. Other examples of *VISs* are *Eclipse*, *Amazon Elastic Compute Service*, and *Drupal Content Management Framework* [10, p. 1] to name a few.

[1] See the **README** file in the Linux kernel tree
[2] 14,387 across all architectures, with an average of 9,984 per architecture, and 10,335 for the `x86/` architecture

2.3 Linux Kernel Development

The Linux kernel development model is unique in many ways, since it has many thousand people world over, who contribute, but there is a very strict hierarchy where certain people have authority over a specific part of the kernel[3]

Stable Releases

The Linux kernel development cycle has approximately $2^3/_4$ months from one stable release to the next stable release [6]. In the meantime, *Release Candidates (RCs)*, are released approximately once every week.
Then, when the top maintainers of the mainline tree think that the kernel is stable enough, a new stable version is created, and the whole process is repeated.

In-development Releases

The *linux-next* tree is a *git* repository, which merges over 200 other *git* repositories [7], which are all based on the *mainline* tree. The *linux-next* tree is merging these other trees every day and the merge conflicts are handled. The *linux-next* tree always contain the newest commits and is the main testing version, and the latest in-development version[4].

For this thesis project, both the *linux-next* tree and the latest stable version is used. They will be referred to as the latest in-development version, and the latest stable version. As time of data gathering, the latest stable version is *4.1.1*.

2.4 Inner Workings of Linux Kernel

This section will explain in coarse detail, 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, and some are smaller and more for niche setups, and some are infrastructure subsystems [2], which contain scripts and tools for various uses.

The `drivers/` subsystem is by far the largest subsystem (with 57% of the lines of code). It contains all device drivers. It is also mostly contributed to by hardware vendors.
Since it is the largest subsystem, one could suspect it to contain the most warnings. And even when taking relative size into account, one could suspect this on the grounds of the majority of the code being written by hardware vendors.

The `arch/` subsystem (18%) contains architecture specific source code. There are 29 architectures in the `arch/` subsystem. To highlight a few, the `x86` architecture is used for most common personal computers, the `arm` architecture is used mostly for mobile devices.
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.

[3] See the `MAINTAINERS` file in the root folder of the Linux kernel
[4] See the original post about it here: https://lkml.org/lkml/2008/2/11/512

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 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. 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. The feature model of the Linux kernel is too big to fit in a normal sized report.
Figure 2.1 depicts a feature model of a phone configuration, where there are some mandatory features (**Screen** and **Calls**) and some optional (**Media** and **GPS**), and also a choice option (**Color**, **BW**, and **High Definition**) for the screen type, where only one of them may be enabled. 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.
There is no consensus on a unified notation for attributes in feature models [3].

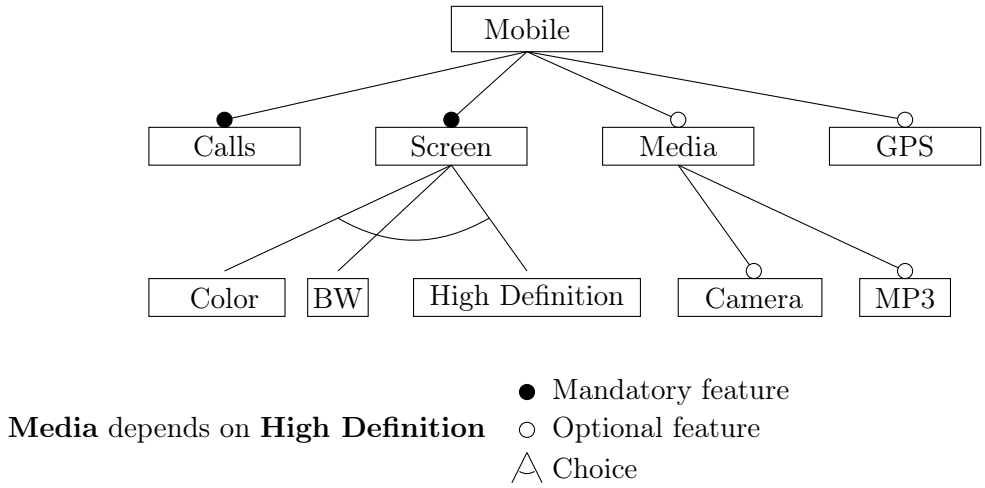


Figure 2.1: An example of a feature diagram of a phone

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].

The KCONFIG language 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 configuration 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 Linux kernel [5] with 956 of them relevant for the `x86/` architecture.

The corresponding KCONFIG code for the phone example is in Figure 2.2.

[5]Found with the command `find . | grep Kconfig | wc`

When a configuration file is created, it is saved as `.config`. In this file, all features are prefixed with `CONFIG_`, and this is the configuration of the Linux kernel. More on the configuration files in Section 2.4.3.
The different data types and the percentage of them in the `x86` architecture are `boolean` (35%), `tristate` (61%), `string` (0.41%), `hex` (0.32%), and `integer` (3.7%). For a description of the context free grammar of the KCONFIG language, see the Appendix 8.1.

```
1    config CALLS
2        def_bool y
3    config SCREEN
4        def_bool y
5
6    choice
7        prompt "Choose screen type"
8        default HD
9        depends on SCREEN
10       config COLOR
11           bool "Color screen"
12       config BW
13           bool "Black and white screen"
14       config HD
15           bool "High Definition screen"
16   endchoice
17
18   config GPS
19       bool "GPS location system"
20   config MEDIA
21       bool "Media modules"
22       depends on HD
23
24   if MEDIA
25       config CAMERA
26           bool "Camera support"
27       config MP3
28           bool "MP3 support"
29   endif
```

Figure 2.2: KCONFIG code for the phone example

2.4.3 Configuring Linux

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

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).

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

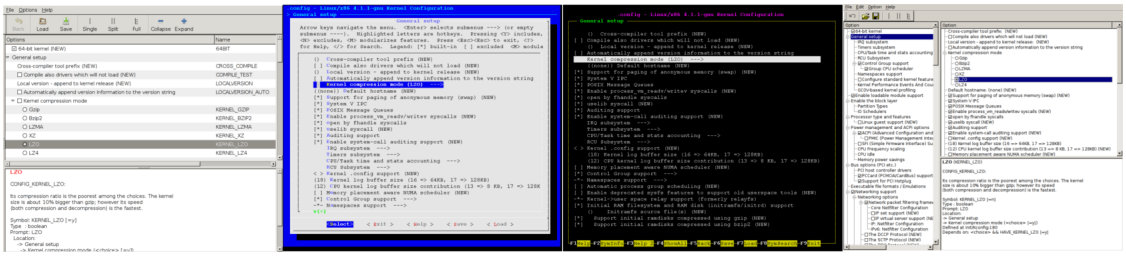


Figure 2.3: `gconfig`, `menuconfig`, `nconfig`, and `xconfig`

Running `make allnoconfig` on the phone example KCONFIG code in Figure 2.2, will try to disable all the features in the feature model. See Figure 2.4(a) for the `.config` file that will be generated.

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

```
1 CONFIG_CALLS=y
2 CONFIG_SCREEN=y
3 # CONFIG_COLOR is not set
4 # CONFIG_BW is not set
5 CONFIG_HD=y
6 # CONFIG_GPS is not set
7 # CONFIG_MEDIA is not set
```

(a) allnoconfig

```
1 CONFIG_CALLS=y
2 CONFIG_SCREEN=y
3 # CONFIG_COLOR is not set
4 # CONFIG_BW is not set
5 CONFIG_HD=y
6 CONFIG_GPS=y
7 CONFIG_MEDIA=y
8 CONFIG_CAMERA=y
9 CONFIG_MP3=y
```

(b) allyesconfig

Figure 2.4

Figure 2.4(b) shows the `.config` that is generated by the configurator `allyesconfig`. Every feature has been enabled, except `COLOR` and `BW`, which must be disabled within the choice clause for the default `HD` feature to be enabled.

2.4.4 Compiling and Catching Warnings

In a warning, there is information about a possible coding mistake that might break the code. There are about 30 different warning types that are enabled by default [12] Some of them are more severe than others, that will be commented on in the next section about the GCC warnings.

The warnings have the output format `[-Wwarningtype]` , and this is output whenever there is a warning. This makes the warnings very easy to quantify automatically.

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, so it is not expected to only get warnings from the `-Wall` group.

The warnings will be categorized into these types of warnings:

1. **Severe warnings**

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

2. **Code pollution warnings**

   These warnings will not break the code, or return wrong values, but will play a role in fitting the Linux kernel on a device with space limitations (such as embedded devices).

3. **Unsevere warnings**

   These warnings are deemed unsevere. This does not mean, that they can not be severe in other projects, but at least regarding correct values, and space limitations,

Here follows a list of the warnings 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

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

cpp

will show `#warning` directives written in the code. This is a warning message that the coder can pass to the compiler. Often they will not result in an error.
This is categorized as *unsevere*.

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 by another function.
This is categorized as *severe*.

discarded-array-qualifiers

error=

This warning is a prefix, which can be put in front of all warning types. It means that the warning will cause an error. Sometimes developers do not want a specific warning to happen without the compilation stopping. Then this flag can be enabled to make sure it will stop.
This is an error, and will *not* be categorized.

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 *unsevere*.

incompatible-pointer-types

This shows when there is converted between pointers, which have incompatible types.
This is categorized as *unsevere*.

int-conversion

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

int-to-pointer-cast

See the section about pointer-to-int on page 8

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

The warning refers to the `!ret`, which should have been inside parentheses to be correct, like so `(!ret)`.
This is categorized as *severe*.

maybe-uninitialized

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

```
1  static int __add_delayed_refs()
2  {
3      int sgn;
4
5      switch (node->action) {
6      case BTRFS_ADD_DELAYED_REF:
7          sgn = 1;
8          break;
9      case BTRFS_DROP_DELAYED_REF:
10         sgn = -1;
11         break;
12     default:
13         BUG_ON(1);
14     }
15     *total_refs += (node->ref_mod * sgn);
16 }
```

Figure 2.5: A real example of maybe-uninitialized function

overflow

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

pointer-to-int-cast

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

return-type

This will be shown when there is a non-void function, which has no return statement.
NOT CATEGORIZED

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 *unsevere*.

uninitialized

This warns about uninitialized variables. The variable has been declared, but has not yet been given a value.
This is categorized as a *severe* 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 not be called if the feature `CONFIG_BATTERY_BQ27X00_I2C` is not enabled, and is therefore an unused function.

```
1  static void bq27x00_powersupply_unregister(struct bq27x00_device_info *di)
2  {
3      poll_interval = 0;
4      cancel_delayed_work_sync(&di->work);
5      power_supply_unregister(di->bat);
6      mutex_destroy(&di->lock);
7  }
8
9  #ifdef CONFIG_BATTERY_BQ27X00_I2C
10
11 static int bq27x00_battery_remove(struct i2c_client *client)
12 {
13     struct bq27x00_device_info *di = i2c_get_clientdata(client);
14     bq27x00_powersupply_unregister(di);
15     mutex_lock(&battery_mutex);
16     idr_remove(&battery_id, di->id);
17     mutex_unlock(&battery_mutex);
18     return 0;
19 }
20
21 #endif
```

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

unused-variable

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.
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 signifant 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.

Since there are more than 10,000 different features in the feature model of the Linux kernel, there will be approximately $2^{10,000}$ possible configurations[1], which 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 2 bit computers).

3.1 The Hunt for Representativeness

To make the sample of configurations representative, five different methods are proposed and discussed.

1. **Using randconfig**

   Using the built-in `randconfig`

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.

[1] Not counting cross-tree contraints, but also saying everything is a `boolean` and not `tristate`, or `string`.

This would require good knowledge of programming in the GNU C language and also an algorithmic way of solving this without the time complexity escalating.

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.

   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[2].

   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 [11]. This formula can be used to check for propositional satisfiability.

Unfortunately, these four methods were all unobtainable, and `randconfig` is used as a proxy for getting a representative sample.

3.2 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.

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. Every single configuration should have equal likelihood of being chosen.

3.2.1 Compiling and Collecting Data

When a configuration has been created, 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 GNU Compiler Collection*.
GCC will output warnings and error messages, which is saved for analysis.

The compilations has mainly been done 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. A fairly regular laptop with 4 cores and 8 GB of RAM does that in just over 8 minutes on average.

Every time a compilation is running, *GCC* will output warning and error messages in the *standard error* output. This output is then scraped through for categorization.

[2] It also is aware about `choice` clauses

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

When a compilation is done, the output is scraped and categorized, and then uploaded to a database, for easy querying during and after the project.

3.3 Analyzing Data

This report is a quantitative analysis of warnings in all of the Linux kernel. The analysis of the warnings is purely quantitative, and this report will not contain any in-depth analysis of some of the warnings. The different warning types will be categorized according to severity.
in comparisson to [2] , which is a qualitative analysis of bugs in Linux.

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.

4.1 Stable Linux Version

In Figure 4.1 is shown the distribution of warnings in the experiment runs with the stable Linux kernel. The distinct warning types are only counted once per experiment run.
The ones that causes errors are marked with red background.

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, which is natural since many different experiments are bound to create some of the same warnings.
It is the same code base, so...

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.

With these results, we can answer **RQ1:**

Observation 1: The most common warnings in the *severe* categorization is: *maybe-uninitialized*, *logical-not-parentheses*, and *uninitialized*.

Observation 2: The warnings *unused-function* and *unused-variable* are among the top 4 of all warnings.

4.2 Subsystems With Warnings

Figure 4.2 shows the distribution of subsystems with warnings in all of the compilations on the stable Linux version.
In many compilation runs, there were multiple warnings, so the percentages will not add up to 100%.
The greyed out rows are for subsystems that are not a major part of the kernel functionality. This is inspired by [2].

Warning	Percentage	Category
unused-function	59.%	Code pollution
maybe-uninitialized	45.%	Severe
logical-not-parentheses	30.%	Severe
unused-variable	29.%	Code pollution
cpp	24.%	Unsevere
uninitialized	19.%	Severe
ERROR	17.%	-
pointer-to-int-cast	17.%	Severe
discarded-array-qualifiers	17.%	xxx
switch-bool	15.%	Severe
frame-larger-than=	14.%	xxx
array-bounds	11.%	Severe
return-type	7.7%	xxx
int-to-pointer-cast	7.6%	Severe
overflow	6.5%	Severe
unused-label	5.4%	Code Pollution
deprecated-declarations	5.4%	Severe
error=implicit-function-declaration	4.6%	-
int-conversion	2.7%	Unsevere
implicit-function-declaration	1.0%	Unsevere
incompatible-pointer-types	0.74%	Unsevere
error=implicit-int	0.029%	-

Figure 4.1: Distribution of warnings in the stable kernel

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

With these results, the following observations can be made:
Observation 3: In 93% of the experiments, the `drivers/` subsystem was present in a warning. This suits the information about `drivers/` being the largest subsystem, and also that this subsystem is in particular contributed to by hardware vendors[1].

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 —

[1] Source on this.

Chapter 5

Threats to Validity

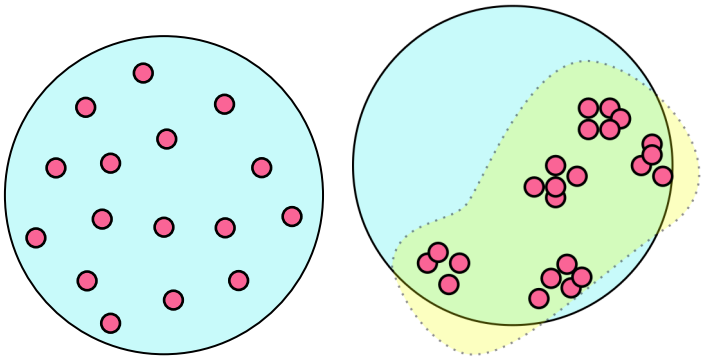
5.1 External Validity

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.
In Figure 5.1 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 5.1(a) shows a sample, which is represesentatively distributed over the configuration space.

Figure 5.1(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 5.1: Showing both representative (a) and unrepresentative (b) samples

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 5.2 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`.

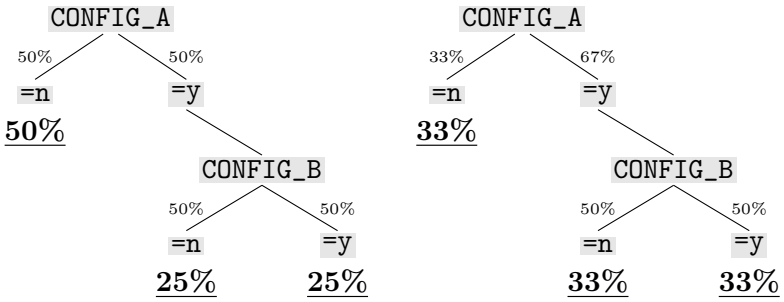
There are two features (`A` and `B`) in the example, which can be enabled or disabled, and feature `B` depends on `A` being enabled. This leaves three possible outcomes.
One where both is enabled, one where only `A` is enabled, and one where none of them are enabled.
The outcome where only `B` is enabled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

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

Figure 5.2: A toy KCONFIG feature model

Figure 5.3(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 5.3(b) for a visualization of how the selection should be to be representative.



(a) How `randconfig` will select (b) Representative selection

Figure 5.3

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 3.1.)

5.2 Internal Validity

5.2.1 9 Different In-Development Versions

There is used multiple different in-development versions of the Linux kernel 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 a quantitative angle. The bugs are manually analysed, and discussed.
This report is somewhat a continuation of the research in that paper.

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