A Quantitative Analysis of Variability Warnings in Linux
Draft: Monday 31st August, 2015 21:42

Elvis Flesborg
efle@itu.dk

September 1st 2015
IT University of Copenhagen

Contents

Abstract

The Linux kernel is the largest open source project to implement variability. It has more than 14,000 options in total that can be switched on and off. This can generate more variants of the Linux kernel, than there are atoms in the universe, and bugs can be harder to find with the extra dimension of variability.

In this project, a sample of these variants are produced and checked for compile warnings to get an insight in the distribution of warning types, and the location of the warnings. The experiment is run both on a stable version of the Linux kernel, and an in-development version, which are compared regarding types and location of the warnings.

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, comparison 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 (*variants*) 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 [16, 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.
The phone example is inspired by some examples from [19], which is an online toolbox for variability software that also has a repository of feature models and diagrams.

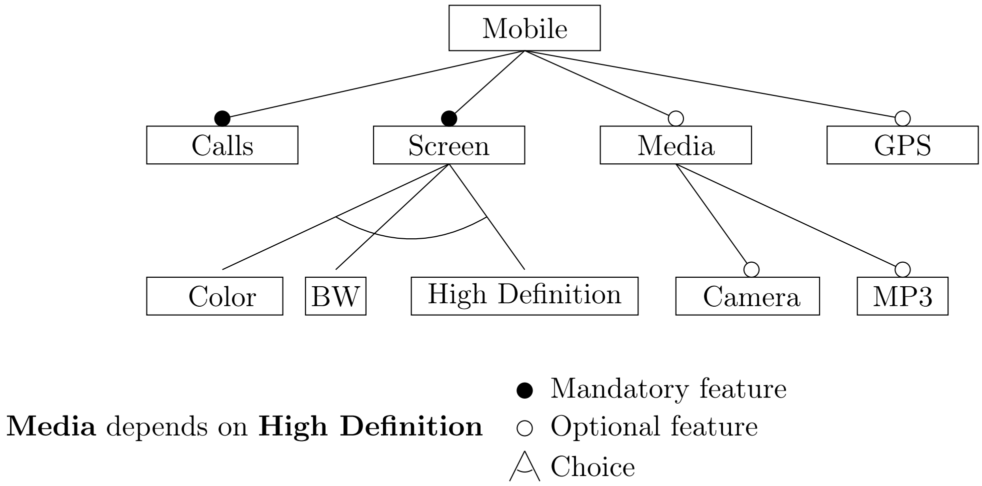


Figure 2.1: A feature diagram of a phone

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].

2.1.2 Bugs In Variability

Bugs in variability programs do not necessarily generate a bug in every variant of the program. A simple example is shown in Figure 2.2. There are only two feature in the feature model: `A`, and `B`, and thus there are four different configurations for this program; `{}`, `{A}`, `{B}`, and `{A,B}`. The feature model, denoted ψ_{FM}, for this example contains four configurations, κ_0, κ_1, κ_2, and κ_3, which can be denoted as conjunctions on logic form like this: $\kappa_0 = \neg A \wedge \neg B$, and $\kappa_1 = \neg A \wedge B$.

In the variant of the program with configuration κ_0, the compilation of the program will return an `uninitialized` warning, since the compiler never visits the two `#ifdef`-clauses. In all other variants of the program, there will be no warning, because `var` gets initialized.

```
int main(void) {
    int var;
    #ifdef CONFIG_A
    var = 1;
    #endif
    #ifdef CONFIG_B
    var = 2;
    #endif
return var+100;
}
```

Figure 2.2: An example of a program with a variability bug.

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

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

supports the most hardware in the world [1, 10].
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 [15].

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 [10], 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 2³⁄4 months from one stable release to the next stable release [9]. 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.

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[5]repository, which merges over 200 other GIT repositories [12], 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.

[3] See the `MAINTAINERS` file in the Linux kernel tree
[4] See the original post about it here: https://lkml.org/lkml/2008/2/11/512
[5] GIT is a version control system that is created by and for the Linux kernel project

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, the `mm/` subsystem (.6%) is about memory management, the `block/` subsystem (.2%) is drivers for data storage devices, and `include/` (4%) contains *header* files, which the kernel needs when compiling [8].

Then there are `sound/`(5%), `kernel/` (1%), `crypto/` (.4%), `security/` (.4%), `lib/` (.6%), and `block/` (.2%).

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

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 Linux kernel[6]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 2.3. 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 9.1.

The different data types in the Kconfig language used in Linux are `boolean` (35%)[7], `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.

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

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

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

```
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 2.3: KCONFIG code for the phone example

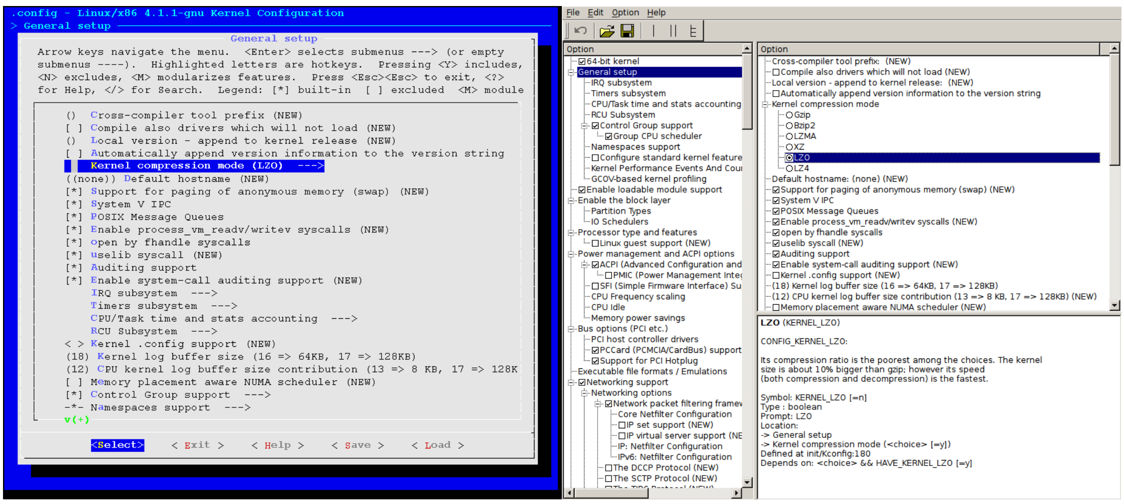


Figure 2.4: `menuconfig` and `xconfig`

- `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 2.5(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 2.5

Figure 2.5(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 [20]. This will enable warnings *"about contructions that some users consider questionable"* [20].

2.4.5 GCC Warnings

All experiments are run with the GCC-flag `-Wall`. In many cases, though, the Linux kernel enables even more warning flags by itself, and many of the warnings in the results are not 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 data, 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 data* 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 data* warning.

frame-larger-than=NUMBER

This is a warning that the stack frame is larger than NUMBER.
This is categorized as *irrelevant*.

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*.

int-to-pointer-cast

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

maybe-uninitialized

This warning is related to the *uninitialized* warning. This shows when there is an uncertainty about a variable being uninitialized. In the case in Figure 2.6 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.

```c
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 2.6: A real example of maybe-uninitialized function

This is categorized as *wrong data*.

overflow

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

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.

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 data* 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 2.7, 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 2.7: 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 some two months old in-development versions 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 to get generated 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% `booleans` 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).

This means that there might be a generalization problem, and a clever way of generating configurations will have to be found. See Section 3.2.

3.1 Experimental Setup

The experimental setup consists of a loop, which does three things. *1:* a configuration is generated, *2:* the Linux kernel is then compiled with this configuration, and *3:* the output warnings are categorized and collected.
Notice that the Linux kernels are not installed and executed in the experiment.

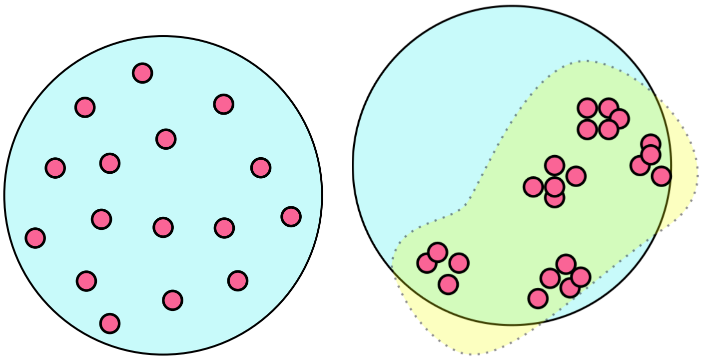
The experiment is mainly carried out on a computer at the IT University of Copenhagen. The computer has a $32 \times 2.8MHz$ cpu and $128GB$ of RAM, and the average time to run an experiment loop is around 1 minute and 35 seconds. Also a conventional laptop with a $4 \times 2.5MHz$ cpu and 4 GB of RAM has been used. The experiment runs for little over a month.

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

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

3.2 Part 1: The Hunt For Representativeness

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 5.2.1 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 [18].

   This formula can be used with a SAT solver, which uniformly selects a random value for all features, which also satisfies the propositional formula, and thereby always returns a valid random configuration.

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.

3.3 Part 2: Compiling and Collecting Data

When a configuration file has been generated, two different Linux kernel versions are compiled using that configuration file, a stable Linux kernel, and an in-development version. The in-development versions are suspected to be more prone to contain warnings, but the warnings are also suspected to be fixed quicker than the bugs in the stable version. To even out any short-lifed warnings, multiple in-development versions are used; nine different versions from nine following days.

The kernel is compiled 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. This data about the experiment run is saved: The original error messages, the `.config` file, the Linux version, the GCC version.
This way the experiment will be reproducible.

3.4 Part 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 comparison to [2], which is a qualitative analysis of bugs in Linux.
The analysis is in the next chapter.

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 400,000 warnings were returned with 3,800,000 filenames.

The stable Linux version is analyzed at first, to answer the first two research questions. At last, these results are hold against the results from the in-development version.

4.1 Stable Linux Warnings

In Figure 4.1 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: *maybe-uninitialized* and *uninitialized*, which are two related warnings.

Observation 2: Two out of three code pollution warnings *unused-function* and *unused-variable* are both in the top 3 of most common warnings.

Observation 3: There are only found 15 different warning types. Only 8 of these are from the `-Wall` group.

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

Warning	Percentage	Category
unused-function	59.%	Code pollution
maybe-uninitialized	45.%	Wrong data
unused-variable	29.%	Code pollution
cpp	24.%	Irrelevant
uninitialized	19.%	Wrong data
ERROR	17.%	Irrelevant
pointer-to-int-cast	17.%	Wrong data
frame-larger-than=	14.%	Irrelevant
array-bounds	11.%	Wrong data
return-type	7.7%	Bad Code Practice
int-to-pointer-cast	7.6%	Wrong data
overflow	6.5%	Wrong data
unused-label	5.4%	Code Pollution
deprecated-declarations	5.4%	Wrong data
implicit-function-declaration	5.6%	Bad code practice

Figure 4.1: Distribution of warnings in the stable kernel

Conclusion 1: The warning types that appear the most, are warnings regarding code pollution, and next come warning types related to variables being uninitialized.

4.2 Stable Linux Subsystems

Figure 4.2 shows the distribution of subsystems with warnings in all of the experiment runs on the stable Linux version. In many experiment runs, there were multiple warnings in the same subsystems. These are only counted once.
The gray rows are subsystems that are not a major part of the kernel functionality, as mentioned in Section 2.4.1. These are the smaller subsystems, and the infrastructure subsystems.

With these results, the following observations can be made:

Observation 3: The subsystem, which appear the most in the warnings, is the `drivers/` subsystem, followed by the `include/` subsystem.

Observation 4: There are zero warnings in the `security/` subsystem.

With these observations **RQ2** can be answered.

Conclusion 2: The subsystems with drivers, and header files have the most warnings, and the security subsystem together with the most smaller, and infrastructure subsystems had zero or near zero warnings.

4.3 In-Development Version vs. Stable Version

In Figure 4.3 is a comparison of the warnings between the stable version, and the in-development version. Only the 6 warnings out of 15 that actually change percentage are shown. All non-shown warnings had less than one percent difference between the two versions.
Observation 5: There are more warnings in the in-development version of Linux, than the stable version.

Subsystem	Percentage
drivers/	64.%
include/	40.%
crypto/	17.%
fs/	14.%
samples/	12.%
net/	10.%
arch/	9.2%
arch/x86/	9.2%
lib/	9.1%
mm/	7.9%
kernel/	5.9%
sound/	3.8%
scripts/	1.6%
usr/	.076%
block/	.75%
security/	.0%

(a) Percentage of experiment runs

Subsystem	Warn/kLOC
samples/	270.
scripts/	58.
crypto/	42.
usr/	30.
mm/	25.
lib/	18.
include/	13.
arch/x86/	8.8
kernel/	2.5
net/	2.4
fs/	2.3
block/	1.5
drivers/	1.2
arch/	0.90
sound/	0.36
security/	0.0

(b) Warnings per 1000 lines of code in the subsystem

Figure 4.2: Distribution of all subsystems within the warnings from the stable Linux

Observation 6: There is only one type of warning that occur more in the stable version, that is the `frame-larger-than=` warning.

With these observations **RQ3** can be answered.

Conclusion 3: There are more warnings and errors in the in-development version.

Warning	%
unused-function	59.%
unused-variable	29.%
ERROR	17.%
frame-larger-than=	14.%
int-to-pointer-cast	7.6%
implicit-function-declaration	5.6%

(a) Stable version

Warning	%
unused-function	62.%
unused-variable	51.%
ERROR	38.%
int-to-pointer-cast	25.%
implicit-function-declaration	23.%
frame-larger-than=	7.8%

(b) In-development version

Figure 4.3: Showing the warning types that has changed more than 1%

Subsystem	Percentage
drivers/	64.%
include/	40.%
crypto/	17.%
fs/	14.%
samples/	12.%
net/	10.%
arch/	9.2%
arch/x86/	9.2%
lib/	9.1%
mm/	7.9%
kernel/	5.9%
sound/	3.8%
scripts/	1.6%
block/	.75%
usr/	.076%
security/	.0%

(a) Stable version

Subsystem	Percentage
drivers/	62.%
include/	42.%
scripts/	25.%
crypto/	16.%
arch/	14.%
arch/x86/	14.%
fs/	13.%
mm/	13.%
net/	10.%
samples/	9.7%
lib/	9.0%
kernel/	3.0%
sound/	1.5%
block/	.27%
usr/	.12%
security/	.0%

(b) In-development version

Figure 4.4: Comparison of the subsystems

Chapter 5

Threats to Validity

5.1 External Validity

5.1.1 Only One Architecture

The experiment is only run on the `x86` architecture. There are more than 20 different architectures supported. For the results to say anything about all of the other architectures, a cross-compiler for each architecture will have to be installed on the building system.
This is possible through various scripts, and by installing a new version of GCC per new architecture[1].

5.2 Internal Validity

5.2.1 The Built-In `randconfig` Configurator

The built-in configurator `randconfig` is not representative, but is biased towards features higher up in the feature model tree.
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`.

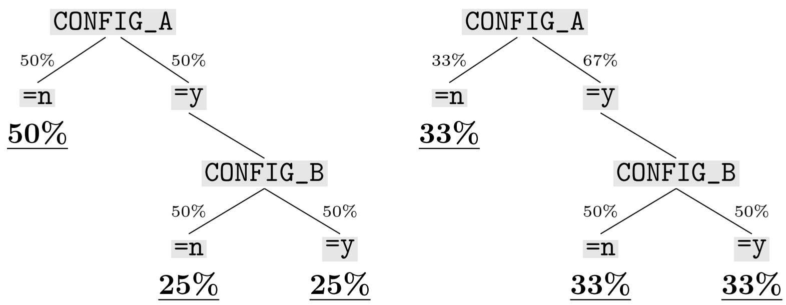
```
config A
    bool
config B
    bool
    depends on A
```

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.

[1] Can be found here: https://www.kernel.org/pub/tools/crosstool/files/bin/x86_64/4.9.0/

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.)

5.2.2 Multiple 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.
In this experiment, nine different versions of the in-development version was used, which is not a lot. Therefore there may be an overrepresentation of some warnings in the in-development version results.

5.2.3 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.

5.2.4 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.

The warnings `discarded-array-qualifiers`, `incompatible-pointer-types`, `int-conversion`, `logical-not-parentheses`, and `switch-bool` showed only on the experiments with the newer version of GCC. These warnings were discarded from the results.

There is a possibility though, that the newer version is better at finding certain warnings that are in the results. This will not be clear to see in the data.

5.2.5 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

Future Work

More Than Linux

This experiment only runs on the linux kernel, but there are variability in many other software products. A similar experiment could be run on other products to see it the results correlate. The paper [2] has an attached online database of the variability bugs found in the paper. These bugs are from both the Linux kernel and also BusyBox[1].

Other Compilers

There has been some disputes between the Linux community and the GCC creators [17]. The Linux is often not too happy with how GCC handles warnings, and in the last couple of years, there has been an initiative to deprecate GCC as the standard compiler for Linux, and instead use LLVM Clang[2], which is faster, and has good static analysis [11].

Analyze Warning-Prone Features

This experiment can be extended to look at what features, or combination of features that result in warnings. When the same warning appears in the same file, on different configurations, it must necessarily be the same combination of a few features that generated the warning. Finding the features that all these configurations have in common will give a lead to what feature combination is creating the warning.
It will be possible to narrow down which features might be generating the bug.

Analyze errors

In this project, only warnings could be automatically quantified, because of the nature of the errors that occured was due to the build system's missing dependencies, and also due to the experiment sample not being large enough to find many errors.

[1] `http://vbdb.itu.dk/`
[2] `http://clang.llvm.org/`

Chapter 7

Related Work

Variability bugs

The paper *42 Variability Bugs in the Linux Kernel...* [2] 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.

Intel and the Kbuild-Robot

INTEL is a large hardware vendor, which is one of the top ten contributors to the Linux kernel [10]. They have started a bugfix initiative called `Kbuild-robot`, which every day runs part 1, and part 2 of this experiment on the in-development version of Linux. As a part 3 and 4, some of the kernels are run, and when an error is found, the author of the change is notified with an e-mail. The mailing lists are publicly available at [**?**, 6, 7].

What is gathered of information about the `kbuild-robot` project, only the mailing lists are publicly available, and the experiment run data is thrown away soon after.
According to [13, 14], 30,000 kernels are compiled and run each day. That is about the same amount of experiment runs this project does in a month.
An effort was made to get their experiment data available for analysis, but no luck.

Chapter 8

Conclusion

The Linux kernel has been analyzed, and there are 14 different warnings that appear in the Linux kernel. The majority of these are regarding code pollution, and uninitialized variables.

The `drivers/` subsystem has the most warnings, but there are warnings in the most subsystems with exception of some of the smaller subsystems and infrastructure subsystems, and also no warnings were found in the `security/` subsystem.

In the in-development version of Linux, there are generally more warnings (and errors) than in the stable version. Half of the warnings appear about the same amount of the experiment runs, while the other half appear almost double as many times.

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] Intel, *kbuild-all mailing list*. https://lists.01.org/mailman/listinfo/kbuild-all.

[7] ——, *kbuild mailing list*. https://lists.01.org/mailman/listinfo/kbuild.

[8] D. Johnson, *The linux kernel: The source code*. http://www.linux.org/threads/the-linux-kernel-the-source-code.4204/.

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

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

[11] M. Larabel, *Clang'ing kernels*. http://www.phoronix.com/scan.php?page=news_item&px=MTE4MTk.

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

[13] LWN.net, *Testing - kernel summit 2012*. http://lwn.net/Articles/514278/.

[14] ——, *Testing - kernel summit 2013*. http://lwn.net/Articles/571991/.

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

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

[17] L. Torvalds, *Usenet chat about gcc*. http://yarchive.net/comp/linux/gcc.html.

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

[19] ——, *Software product lines online tools*. http://gsd.uwaterloo.ca:8088/SPLOT/.

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

Chapter 9

Appendices

9.1 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