

# A Quantitative Analysis of Variability Bugs in Linux

Elvis Flesborg  
`efle@itu.dk`

September 1st 2015  
IT University of Copenhagen

# Contents

<b>1</b>	<b>Abstract</b>	<b>1</b>
<b>2</b>	<b>Introduction</b>	<b>2</b>
<b>3</b>	<b>Background</b>	<b>3</b>
3.1	The Linux Kernel and Variability . . . . .	3
3.2	Linux Kernel Development . . . . .	3
3.2.1	Mainline tree . . . . .	4
3.2.2	Stable trees . . . . .	4
3.2.3	linux-next tree . . . . .	4
3.3	Inner Workings of Linux Kernel . . . . .	4
3.3.1	Subsystems . . . . .	4
3.3.2	Kconfig - The Language of the Feature Model . . . . .	5
3.3.3	Configuring of Linux . . . . .	7
3.3.4	Compiling and Catching errors . . . . .	8
<b>4</b>	<b>Methodology</b>	<b>12</b>
4.1	Research Questions . . . . .	12
4.2	Collecting data . . . . .	12
<b>5</b>	<b>Data</b>	<b>14</b>
<b>6</b>	<b>Results</b>	<b>15</b>
6.1	Analyzing the data . . . . .	15
6.2	Number of valid configurations . . . . .	15
6.3	Observations . . . . .	15
<b>7</b>	<b>Threats to Validity</b>	<b>16</b>
7.1	Generalization . . . . .	16
7.1.1	Architecture . . . . .	16
7.1.2	Firmware . . . . .	17
7.2	Generate'n'Filter . . . . .	17
7.3	elvisconfig . . . . .	17
7.4	RandomSAT . . . . .	17
<b>8</b>	<b>Related Work</b>	<b>18</b>
<b>9</b>	<b>Conclusion</b>	<b>19</b>

# Chapter 1

## Abstract

*—Leave blank until the end—*

## Chapter 2

# Introduction

*—Leave blank until the end—*

- TODO
- Explain about representative samples

# Chapter 3

## 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...) [pac]. Both is needed, to have a working operating system.

This report will only focus on the *Linux kernel*, and not the *GNU* bundle.

### 3.1 The Linux Kernel and 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 or *Software Product Lines* (SPL), when different programs can be made from the same source code base.<sup>1</sup>

The options that can be changed, are called *features*.

The Linux kernel is very configurable. In the configuration files (the Kconfig files), there is a total of 14,387 different features, that can be configured.<sup>2</sup> It is the open source software product with the highest degree of variability.

This high degree of variability has also had an impact on the use cases for the Linux kernel. Linux is used for anything from small embedded devices (like car GPSs, and cell phones) to supercomputers. In fact 98% of the top 500 supercomputers in the world run a Linux distribution [Sit15] .

This high variability rate makes maintaining the code somewhat harder to grasp, and this makes the code base more error prone. So what you get in scalability you pay for in complexity of maintaining the code.<sup>3 4</sup>

### 3.2 Linux Kernel Development

*Comment from the linux-next page or something*

---

<sup>1</sup>Check definition of SPL, and write it here

<sup>2</sup>14,387 across all architectures, with an average of 9,984 per architecture

<sup>3</sup>Refer to Jean's project or something

<sup>4</sup>Also refer to a paper that andrjew made about industrial something. It said something about how much more efficient the code was, when variability was there.

### 3.2.1 Mainline tree

The Linux kernel development cycle has approximately  $2^{3/4}$  months from one stable release to the next stable release [krps] . In the meantime, *RC* releases, also called *prepatch* releases, are released once in a while. Mainly for kernel developers to test for new features.

### 3.2.2 Stable trees

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.

### 3.2.3 linux-next tree

Then there is the *linux-next* tree. It is a *git* repository, which merges over 200 other *git* repositories<sup>5</sup>, 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. Therefore the *linux-next* tree always contain the newest commits.

This tree will get some bugs fixed sooner than if everyone contributed to the mainline and tested on that. Comparing to a stable version, one would suspect the *linux-next* tree to have more bugs, since it has not been tested, and it is the newest code available, which means more bugs have been inserted.

This leads to our first research question.

**RQ1:** Will the unstable linux-next repository generate more errors than the stable kernel.

For this thesis project, both the *linux-next* tree and the latest stable version is used. As time of data gathering, the latest stable version is 4.1.1.

6 7 8 9

## 3.3 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.

### 3.3.1 Subsystems

The directories in the root folder of the Linux kernel source code are called *subsystems*.

The **drivers/** subsystem is by far the largest subsystem. It contains all hardware drivers. It is also mostly contributed to by hardware vendors.<sup>10</sup> Since it is the largest subsystem, one could suspect it to contain the most errors. 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.

This leads to next research question:

**RQ2:** In what subsystems do the most errors occur. And in the largest subsystems, in what subsubsystem?

---

<sup>5</sup>See <src>/Next/Trees

<sup>6</sup><http://neuling.org/linux-next-size.html>

<sup>7</sup><http://news.gmane.org/gmane.linux.kernel.next> - the linux-next mailing list archive

<sup>8</sup><http://kisskb.ellerman.id.au/kisskb/matrix/> - this is some kind of build robot results page.

<sup>9</sup><http://git.kernel.org/cgit/linux/kernel/git/next/linux-next.git/tree/Next/Trees?id=HEAD> - here are all the git trees that are merged in linux-next

<sup>10</sup>Source? and is it true? -What would Greg Kroah Hartman say?

The `arch/` contains architecture specific source code. In this project, only the `arch/x86` architecture is compiled for, so only this subsystem will be in use.

Many of these architectures are very rarely used in day to day computing, but they are one of the reasons that Linux can run on so many devices. There are 30 architectures in the `arch` folder.

- **TODO**

- Find out how to find the exact number of possible configurations. It is in some article, I got by Andrzej. Plus maybe SharpSAT.
- Explain how to find the number of features by grepping.
- Talk about subsystems
- Maybe talk about the percentage of code in every subsystems. What is Linux kernel made of?
- talk about who contributes to the kernel. (intel, novell). Both in code and money?
- mention some more subsystems other than drivers and arch

### 3.3.2 Kconfig - The Language of the Feature Model

A **feature model** is a way of representing all the possible configurations - *the configuration space*. It contains all the features, with all the constraints and dependencies.

For a given configuration  $\kappa$ , that configuration must satisfy the feature model  $\psi_{FM}$ .

$$\kappa \in \psi_{FM} \quad (3.1)$$

The **Kconfig language** is the language of the feature model in Linux (also used for other projects like busybox, buildroot, and others).<sup>11</sup> The configuration files have the prefix `Kconfig`, and are scattered all over the Linux kernel source code tree, where they include each other.

The different data types are `boolean`, `tristate`, `string`, `hex`, and `integer`. See figure 3.1 for a toy Kconfig example showing some of the syntax.

```
1 config WIFI
2     tristate "WiFi support"
3     depends on NETWORK
4     selects WPA
5     default y
6     --- help ---
7         Select this if you want WiFi to work
8         If in doubt, choose y
```

Figure 3.1: A toy example of the Kconfig language.

A feature model can be visualized in a feature diagram. The example given here 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.

See figure ?? for the toy example of a feature diagram. See figure ?? for the corresponding Kconfig file. It depicts a feature model of a laptop configuration, where there are some mandatory features and some optional, and also a choice option for the screen size. A cross-tree dependency is also present.

- **TODO**

---

<sup>11</sup>Cite the Variability Modeling in the systems software domain paper

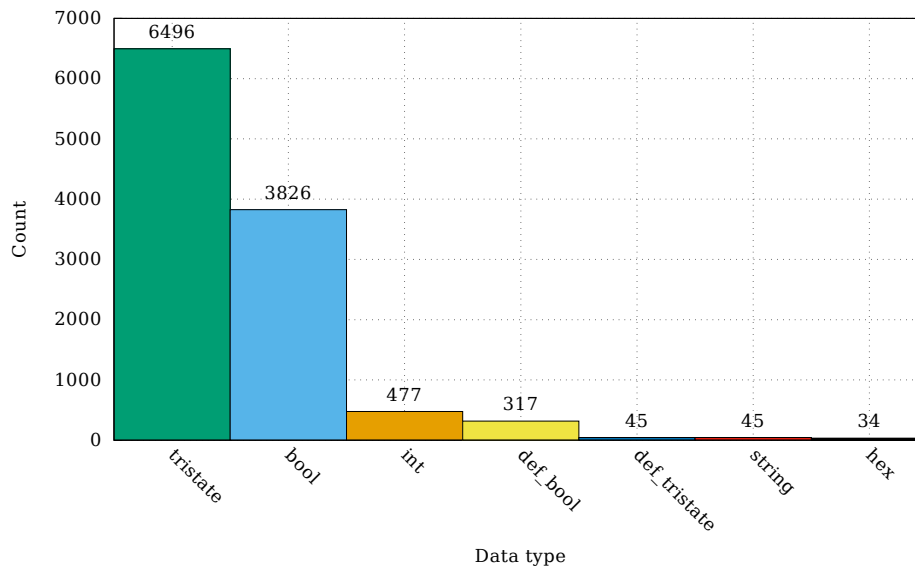


Figure 3.2: The distribution of different datatypes in the feature model of the x86 architecture

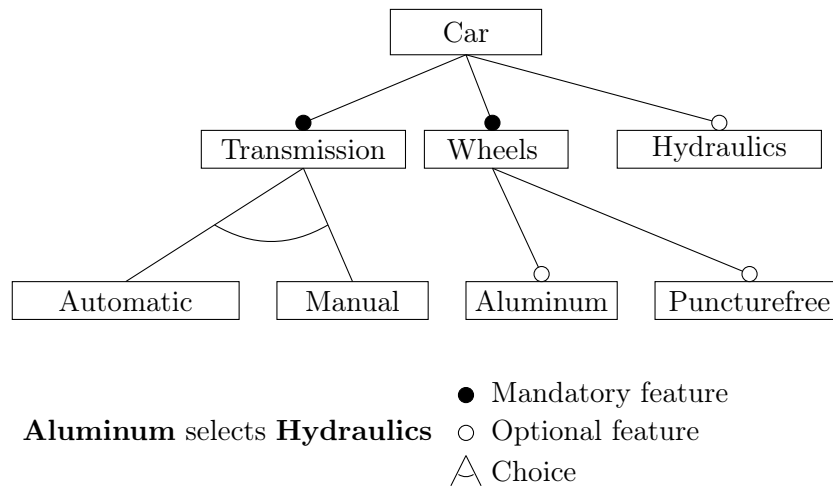


Figure 3.3: An example of a feature diagram of a car



- Explain what the Linux kernel tree is.
- Find another example than a car
- Find a more simple Kconfig code example
- Explain the Kconfig language a bit
- Reference to Thorsten's projects
- Refer to the /Documentation/kbuild/kconfig-language.txt
- explain that CONFIG\_MMU and MMU is the same
- Should I have the Backus Naur Form of the Kconfig language here?

### 3.3.3 Configuring of Linux

The Linux kernel comes with different ways of creating your own configuration file. There is a question based one (**config**), and some menu based ones (**menuconfig**, **xconfig**, **nconfig**, **gconfig**), but also some that will never prompt the user for anything, but create some automated config (**allyesconfig**, **allnoconfig**, **defconfig**, and **randconfig**).

See figure 3.4 for a few examples of how the graphical configuration tools look.

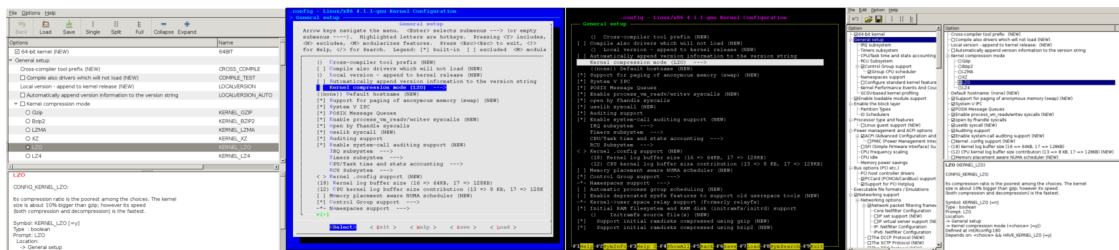


Figure 3.4: gconfig, menuconfig, nconfig, and xconfig

That last one **randconfig** is interesting, as it will go through the Kconfig feature model files, and pick random values for all of the features. This should be perfect for the goal for this project. Although time will tell that it is indeed biased, and will not create configurations that are of representative character.

See figure 3.5 for a toy example of a Kconfig feature model. It is a very small example with only two features, but it will easily explain some limitations of using **randconfig**.

```

1 config A
2     bool
3 config B
4     bool
5     depends on A

```

Figure 3.5: A toy Kconfig feature model

The example in figure 3.6 shows that the three possible configurations (**{ }**, **{A}**, and **{A,B}**) do not have the same chance of getting created by **randconfig**.

For the creation to be representative, all the three possible configurations should have equal chance of being created (33%). See figure 3.7 for a visualization of how the selection should be to be representative.

This has apparently not been implemented in the Linux kernel tools, possibly because representativeness does not matter for what it is used for. It is merely used as a simple fuzz testing tool with the help of entropy.<sup>12</sup>

<sup>12</sup>source

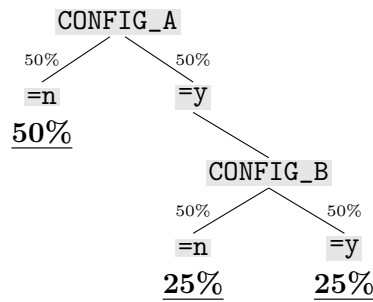


Figure 3.6: A toy example showing the way randconfig selection of features

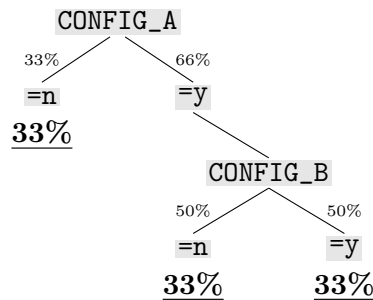


Figure 3.7: A toy example showing a correct selection of features

To make this representative, one would have to make **randconfig** aware of the whole dependency tree or the whole configuration space before randomly selecting, to skew the possibilities. This seems infeasible since the configuration space is so large.

Some different ways of generating a representative sample was considered. In the end, **randconfig** was chosen because it showed to be too cumbersome to create a method that actually was representative.

The methods that was considered will be explained later in this report.<sup>13</sup> They included the aforementioned **randconfig** tool being rewritten, a script that scrambled the Kconfig files, and a generate and filter method.

- **TODO**
- .config file
- show results from running randconfig a 1000 times.
- Talk about how there is 7 layers of dependencies. And how I found out.
- Hmm . Are there only 7 layers when counting ifs? what about 'depends on's
- mention that it is also called a preprocessor.

### 3.3.4 Compiling and Catching errors

When a configuration has been created, the Linux kernel is compiled using that configuration file by running the command **make all**. This looks in a **Makefile**, which specifies what files to compile. The compilation of the files is done mainly by *gcc - The GNU Compiler Collection*.

During compilation, **gcc** will output warnings if there are any. These warnings warn about bad code, which might result in a compilation error, or which might result in dead code, or they

---

<sup>13</sup>refer to it when you have written it

might just warn about bad code practice.

These output messages hold the information that will be used in this project. In a warning, there is information about a possible coding mistake that might break the code. There are about 30 different warning types [wt] Some of these are not so interesting as others for this project.

If there is an error, the compilation will stop immediately (sometimes it will try to keep on going, but in this project, it is told to stop after one error).

Generally there will be distinguished between four different types of output. These are **Linux errors**, **host errors**, **relevant warnings**, and **irrelevant warnings**.

The different types of output will be described here.

**The Linux errors** are reproducible errors that will occur on all machines when a certain configuration is being built.<sup>14</sup>

**The host errors** are a result of the building system (a host system) not having some specific dependencies installed.

For instance, there is a feature in the feature model that is called `CONFIG_KERNEL_LZO`, which dictates that the kernel should be compressed by the *lzop* compression library instead of the default *gzip* compression library.

If a configuration has the `CONFIG_KERNEL_LZO` enabled, then to build the kernel, the *lzop* library will have to be installed on the build system. If it is not, then the compilation will stop with an error.

There are six different ways to compress the Linux kernel (*gzip*, *bzip2*, *lzma*, *xz*, *lzo*, and *lz4*), which leads to that every sixth compilation will stop with the same error.

See figure 3.4 for an overview of how this specific choice looks in the different graphical configurators..

There exist many other examples like this in the Linux kernel where an uninstalled program will result in an unbuilt kernel. These **host errors** would have to be minimized, since they are not actual kernel errors, but local errors.

On the build system for this project, the *lzo* package was in the Linux distribution's repository, and therefore easy to install. The *lz4* packager however was not.

The following is a list of mandatory features, which are added to decrease the amount of host errors. In an early period of the project, it was found out that these specific features with specific values would stop the compilation every time.

It decreases the configuration space of this project, but otherwise would just give a lot of false positives.

All these lines are added to the configurations, which are generated in this project.

- `CONFIG_STANDALONE=y`
- `CONFIG_FW_LOADER=n`
- `CONFIG_SECCOMP=y`
- `CONFIG_PREVENT_FIRMWARE_BUILD=y`
- `CONFIG_*LZ4*=n`<sup>15</sup>

---

<sup>14</sup>Refer to the list of errors - some link somewhere

<sup>15</sup>This means that all features having to do with *lz4* were disabled

See chapter 7 about Threats to Validity on page 16 for a more thorough description of the specific features.

**The relevant warnings** are warnings that may cause an error at some point. The *uninitialized variable* warning is such a warning. If a variable is uninitialized, and ends up being called, it may cause an error, and stop the compilation. See figure 3.8 for a code example.

The relevant warnings are:

- Uninitialized variable
- Maybe uninitialized
- Overflow
- Deprecated declarations <sup>16</sup>

```
1 int main(void) {  
2     int n;  
3     #ifdef A  
4         n = 10;  
5     #endif  
6     printf("%d", n);  
7     return 0;  
8 }
```

Figure 3.8: A toy program showing an uninitialized variable.

If `CONFIG_A` is not enabled in the program in figure 3.8 it will return the warning `'n' is used uninitialized in this function [-Wuninitialized]` and the output, when the program is run, will be `'0Hello world'`, which is wrong, as the program was meant to output `'Hello World'`.

**The irrelevant warnings** are warnings that will not stop the compilation, and will not make the program output wrong things. Warnings like *unused function*, *unused variable*, *unused label*, and *unused value*, will all end up as dead code, which is bad for fitting a kernel on some hardware with limited space. Apart from that it is nothing else than code pollution. See figure 3.9 for a code example of an unused variable.

The irrelevant warnings are:

- Unused function
- Unused variable
- Unused label
- Unused value <sup>17</sup>

```
1 int main(void) {  
2     int n = 50;  
3     #ifdef A  
4         printf("%d", a);  
5     #endif  
6     return 0;  
7 }
```

Figure 3.9: A program resulting in an unused variable warning.

This will return the warning `'unused variable 'n' [-Wunused-variable]`.

---

<sup>16</sup>Complete this list

<sup>17</sup>Complete this list

- TODO

# Chapter 4

## Methodology

The objective is to make a quantitative analysis of errors and warnings in the Linux kernel based on as many random configurations as are possible to generate within the time span of this project.

There will be generated random configurations, and these will be used to compile two different versions of the Linux kernel, a stable Linux kernel version *4.1.1*, and a Linux kernel from the *linux-next* repository.

### 4.1 Research Questions

It is likely that some errors or warnings are more severe than others, and some types of errors get fixed more than others. This leads to the next question.

**RQ2:** What warnings are mostly generated.

And this can also be different between the stable and unstable versions. Hence the next question.

**RQ3:** What bugs seem to be fixed the most when going from unstable to stable.

As mentioned <sup>1</sup>, the subsystems of the kernel are of very different character. <sup>2</sup> And therefore the bug types must differ from subsystem to subsystem.

**RQ5:** What percentage of the possible configurations for the Kernel are valid.

**RQ6:** Are there any specific features, that when enabled, will produce more warnings than others.

### 4.2 Collecting data

Every time a randomly configured compilation is running, *gcc* will output warning and error messages in the *standard error* output. This output is then scraped through to categorize it. An output line will contain a *bug type*, a *filename*, *line number*, and a *message* describing the warning in english.

The collection of data 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

---

<sup>1</sup>Where is this mentioned about the subsystems? refer to it

<sup>2</sup>mention somewhere earlier, that drivers are different from security or 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.

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.

See the figures 4.1, 4.2, and 4.3 for the database tables.

Configurations		
Name	Type	primary key
hash	char(64)	primary key
exit_status	int(1)	
conf_errs	text	
linux_version	varchar(100)	primary key
original	longtext	

Figure 4.1: The Configurations table

Bugs		
Name	Type	primary key
hash	char(64)	primary key
type	varchar(50)	
linux_version	varchar(100)	
config_hash	char(64)	
subsystem	varchar(30)	
original	longtext	

Figure 4.2: The Bugs table referring to the Configurations table

Files		
Name	Type	primary key
id	int(11)	primary key
path	varchar(50)	
line	varchar(15)	
bug_id	char(64)	

Figure 4.3: The Files table referring to the Bugs table

- Tell why we need to compile
- Tell about coccinelle, and why we dont use that
- Categorizing the warnings/Errors

## Chapter 5

# Data

- TODO
-



# Chapter 6

## Results

6.1 Analyzing the data

6.2 Number of valid configurations

6.3 Observations

- TODO
-

# Chapter 7

## Threats to Validity

*Blahblah about Threats to validity...*

### 7.1 Generalization

One major threat to validity is in the representativeness of the random configurations. Since the configurations are created using the `make randconfig` method, some configurations are more likely to be generated than others.

This means, that when something is generally said about all of the Linux Kernel, instead something is said about mostly a subset of Linuxes. See figure 7.1 for a visualization of subsets which are both representative, and not representative.

*Show an image of true representativeness*

Figure 7.1: A representative subset

1

### Permutation of Kconfig files

Another idea was to take all the spread out Kconfig files, and concatenate them all into one big text file, which would then be modified. If the order of the features were randomized, then the coins would not be flipped in the same order every time, and this could have an effect on the dependent features.

### Generate and Filter

The third idea was to manually generate a configuration file, and then check - somehow - if the config file was valid, and then use it if it was valid. By not looking at dependencies, but only considering *choice* options and enabling *mandatory features*, a configuration file was created by randomly flipping a coin per each feature.

This seemed doable until after generating 100,000 configurations, not one valid configuration was found. It seems like many of the  $2^{11,000}$  configurations are invalid.

#### 7.1.1 Architecture

To create a true representative sample space, all the different architectures should be compiled for. There exist 31 different architectures, but the most common one for laptop and server use is the *x86* architecture.

---

<sup>1</sup>Here needs to be explained about changing randconfig first

This architecture is the only one, that are compiled for in this project, as getting the cross compilers for all the different architectures would be very cumbersome.

This potentially leaves out 97%<sup>2</sup> of the possible configurations to check.

### 7.1.2 Firmware

Some places in the kernel, there are drivers, which rely on some external proprietary binary blob, before they can be built. These binaries are not in the kernel, but would have to be downloaded specifically and put in certain folders in the kernel tree.

These are throwing errors, which are local errors, and therefore invalid. The task of getting all the drivers into the kernel tree was too great, so instead, those configurations are simply skipped. This gives another bias in the sample, which now does not contain any configurations with these features on.

The feature `CONFIG_STANDALONE` for instance, has been enabled in all the configurations in this project. If this feature is disabled, it allows for firmware drivers to be compiled in the kernel. But these firmware drivers are not open source, and must be placed in the kernel directories manually for the compilation to work.

Since it would be too cumbersome to find all the proprietary drivers, this option of enabling the `CONFIG_STANDALONE` feature has been chosen, and will commit to a threat to validity in respect to representativeness.

Also every feature that had some relation to the *z4c* library has been disabled. This would require the host<sup>3</sup> to have this library installed. This library was not in the Linux distribution's repository, and it would have been too time consuming to have it installed.

Therefore this also marks as a threat to validity.

- randconfig bias
- Internal vs. External validity
- The mandatory features?

## 7.2 Generate'n'Filter

- Uniform distribution
- Too low percentage
- Get a lower bound on the percentage
- sharpSAT?

## 7.3 elvisconfig

## 7.4 RandomSAT

- TODO
- Maybe the permutation script can be used elsewhere (Thorsten seemed interested)

---

<sup>2</sup>is this number true. Should I calculate?

<sup>3</sup>explain what a host and target are at some point further up

## Chapter 8

### Related Work

- TODO
- 42 bugs
- Variability in ...
- Paper from Iago
- Fengguang Wu and Intel

## Chapter 9

# Conclusion

— *Leave empty until the end*—

# Bibliography

- [krps] Linux kernel release prediction site. <http://phb-crystal-ball.org>.
- [pac] GNU packages. <http://www.gnu.org/software/software.html>.
- [Sit15] Top 500 Supercomputer Sites. <http://top500.org/statistics/list>, June 2015.
- [wt] GCC warning types. <https://gcc.gnu.org/onlinedocs/gcc-5.1.0/gcc/Warning-Options.html#index-Wall-292>.