A Quantitative Analysis of Variability Bugs in Linux

Elvis Flesborg
efle@itu.dk

September 1st 2015
IT University of Copenhagen

Contents

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 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, when different programs can be made from the same source code base.
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. [1] It is one of the highest degree of variability in the available open source software. 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 distibution. [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 hardness of maintaining the code. [2] [3]

3.2 Linux Kernel Development

Comment from the linux-next page or something

3.2.1 Mainline tree

The Linux kernel development cycle has approximately 2¾ months from one stable release to the next stable release. [krps] In the meantime, *RC* releases, also called *prepatch* releases, are

[1] 14,387 across all architectures, with an average of 9,984 per architecture
[2] Refer to Jean's project or something
[3] 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.

released once in a while. Mainly for kernel developers to test for new features.

3.2.2 linux-next tree

Then there is the *linux-next* tree. It is a *git* tree, which merges over 200 other *git* trees [4] , which all are 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.

3.2.3 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.

For this thesis project, both the *linux-next* tree and the latest stable version is used. As time of writing, the latest stable version is *4.1.1*.
[5] [6] [7] [8]

3.3 Inner Workings of Linux Kernel

This section will explain in coarse detail, the structure of the Linux 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 one both regarding lines of code (LOC) and size. A quick description of some of the subsystems will follow. For a visualization of the sizes of the subsystems see figure 3.1.
`Documentation` is as the name states, purely documentation about all different kinds of aspects of the Linux kernel.

Quote from Greg K. Hartman about they write tons of docs, but none are read

`arch` contains architecture specific source code. In this project, only the `arch/x86` architecture is compiled for, so only this subsubsystem will be in use.
Many of these architectures are very rarely used, but they are one of the reasons that Linux can run on so many devices. There are 30 architectures in the `arch` folder, and the sizes can be seen in figure 3.2.

- **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.

[4] See <src>/Next/Trees
[5] http://neuling.org/linux-next-size.html
[6] http://news.gmane.org/gmane.linux.kernel.next - the linux-next mailing list archive
[7] http://kisskb.ellerman.id.au/kisskb/matrix/ - this is some kind of build robot results page.
[8] 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

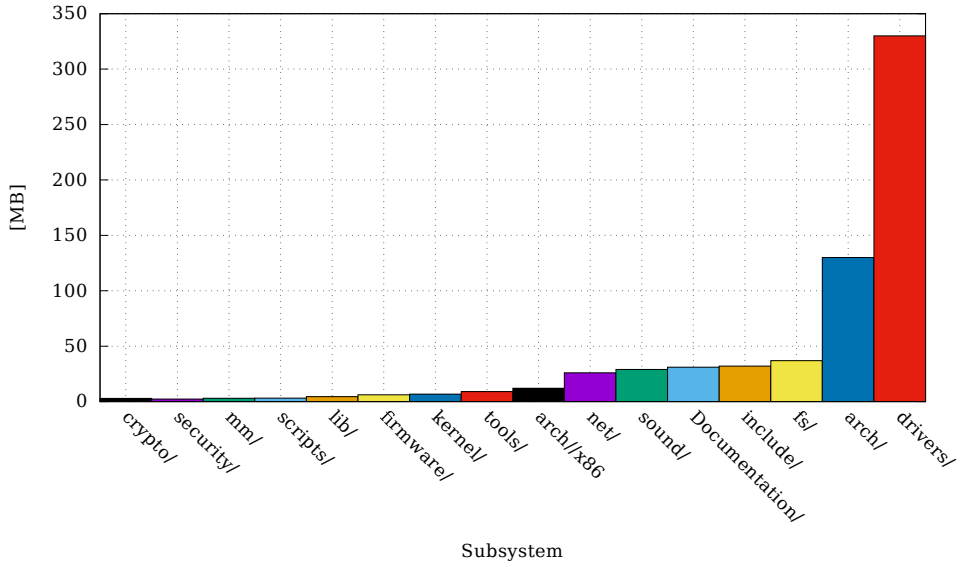

Figure 3.1: Visualization of the sizes of the subsystems

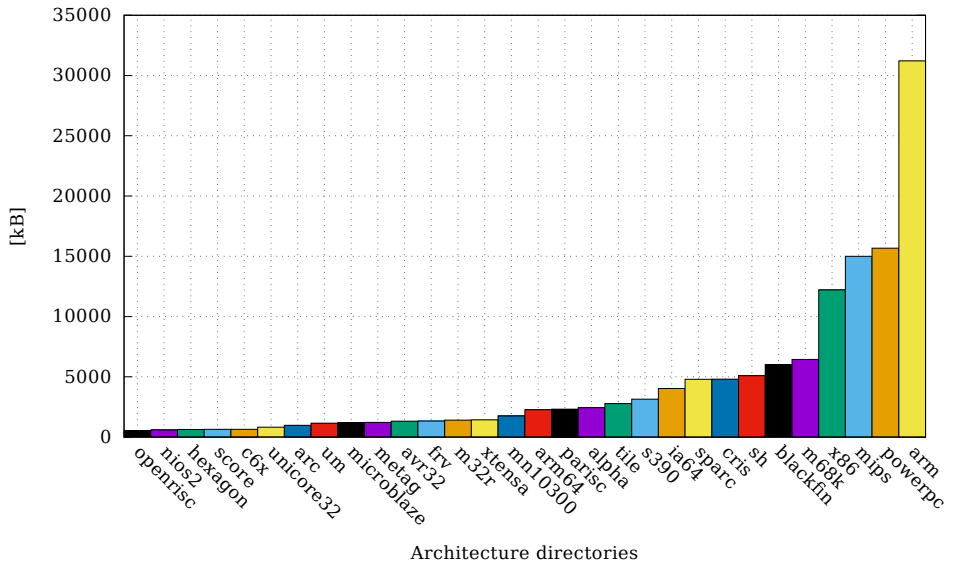

Figure 3.2: The sizes of the different architecture diretories

- 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?
- talk about subsystems

3.4 Kconfig - The Feature Model Language

The feature model is a way of representing all the possible configurations. It contains dependencies and constraints for the features, and if a chosen configuration fulfills the constraints, then the configuration is valid.
For a given configuration κ, it must satisfy the feature model ψ_{FM}.

$$\kappa \in \psi_{FM} \tag{3.1}$$

The Kconfig language is the language of the feature model in Linux (also used for other projects like busybox, buildroot, and others). 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 3.3 and figure 3.4 for some real example of the Kconfig syntax. Figure 3.3 shows an example of a `choice` option, where only one of multiple features can be enabled.

```
config IKCONFIG
    tristate "Kernel .config support"
    select BUILD_BIN2C
config IKCONFIG_PROC
    bool "Enable access to .config through /proc/config.gz"
    depends on IKCONFIG && PROC_FS
config LOG_BUF_SHIFT
    int "Kernel log buffer size (16 => 64KB, 17 => 128KB)"
    range 12 21
    default 17
    depends on PRINTK
config INITRAMFS_SOURCE
    string "Initramfs source file(s)"
    default ""
config PAGE_OFFSET
    hex
    default 0xB0000000 if VMSPLIT_3G_OPT
    default 0x80000000 if VMSPLIT_2G
    default 0x78000000 if VMSPLIT_2G_OPT
    default 0x40000000 if VMSPLIT_1G
    default 0xC0000000
    depends on X86_32
config HIGHMEM
    def_bool y
    depends on X86_32 && (HIGHMEM64G || HIGHMEM4G)
```

Figure 3.3: A real Kconfig example showing the basic data types

This can be visualized in feature diagram. 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 sizes report.
See figure 3.6 for the toy example of a feature diagram. It depicts a feature model of a car configuration, where there are some mandatory features and some optional, and also a choice option for the transmission. A cross-tree dependency is also present.

```
choice
    prompt "Kernel compression mode"
    default KERNEL_GZIP
    depends on HAVE_KERNEL_GZIP || HAVE_KERNEL_BZIP2 || HAVE_KERNEL_LZMA ||
    HAVE_KERNEL_XZ || HAVE_KERNEL_LZO || HAVE_KERNEL_LZ4
config KERNEL_GZIP
    bool "Gzip"
    depends on HAVE_KERNEL_GZIP
config KERNEL_BZIP2
    bool "Bzip2"
    depends on HAVE_KERNEL_BZIP2
config KERNEL_LZMA
    bool "LZMA"
    depends on HAVE_KERNEL_LZMA
config KERNEL_XZ
    bool "XZ"
    depends on HAVE_KERNEL_XZ
config KERNEL_LZO
    bool "LZO"
    depends on HAVE_KERNEL_LZO
config KERNEL_LZ4
    bool "LZ4"
    depends on HAVE_KERNEL_LZ4
endchoice
```

Figure 3.4: A real Kconfig example showing a choice clause with 5 choices

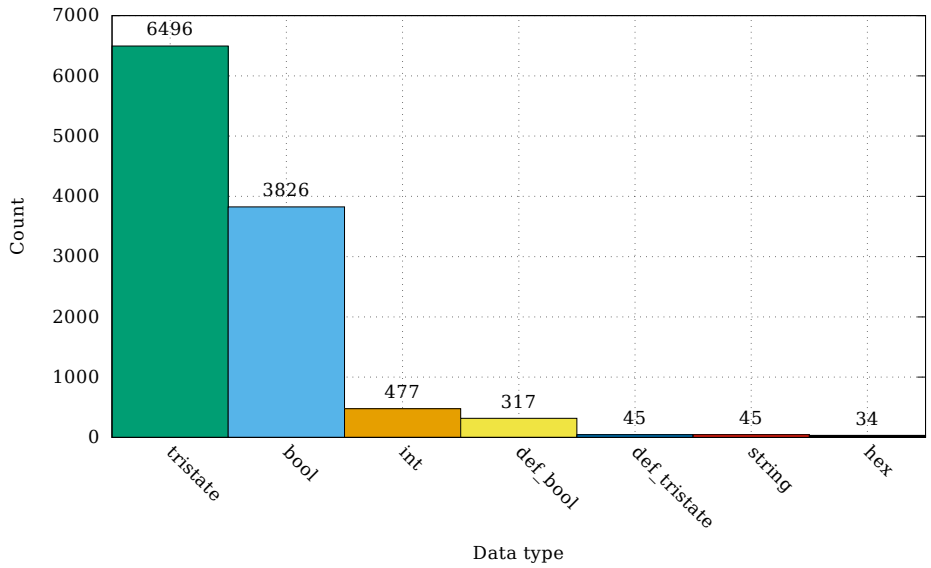

Figure 3.5: The distribution of different datatypes in the feature model

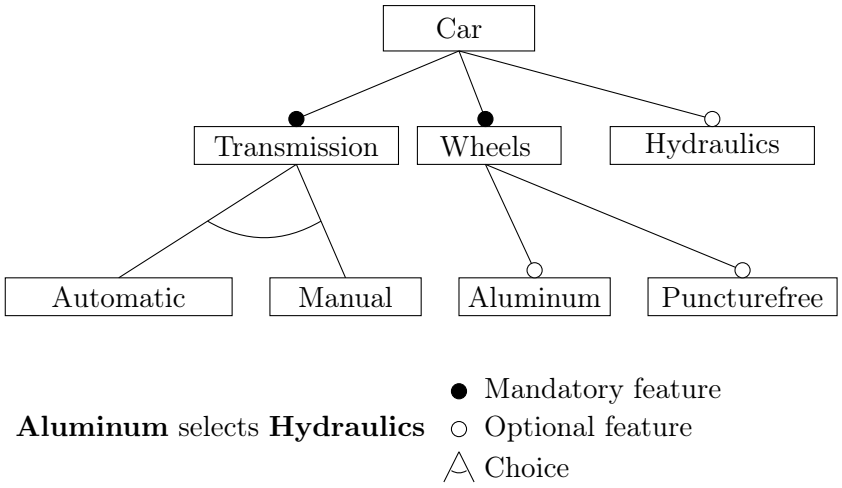

Figure 3.6: An example of a feature diagram of a car

- **TODO**
- Explain what the Linux kernel tree is.
- Find another example than a car
- Find a more simple Kconfig code example
- It creates a '.config' file.
- Explain the Kconfig language a bit
- Examples of constraints in the language
- 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?

3.5 randconfig

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*).

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 3.7 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 *randconfig*.

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

Figure 3.7: A toy Kconfig feature model

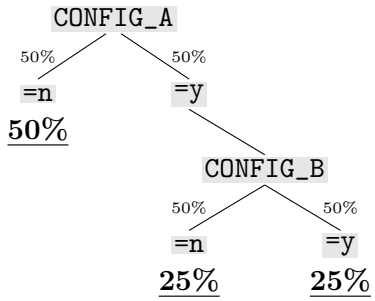

Figure 3.8: A toy example showing the way randconfig selection of features

In the example in figure 3.8 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 3.9 for a visualization of how the selection should be to be representative.

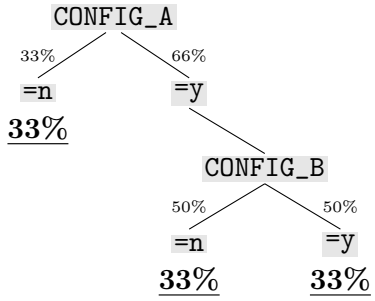

Figure 3.9: A toy example showing a correct selection of features

This has apparently not been implemented in the Linux kernel tools, because representativeness does not matter for what it is used for. It is merely used as a simple testing tool.
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. [9] They included the aforementioned *randconfig* tool being rewritten, a script that scrambled the Kconfig files, and a generate and filter method.

- **TODO**
- Show some code of randconfig not doing it representatively
- show results from running randconfig a 1000 times.
- Talk about how there is 7 layers of dependencies. And how I found out.
- Ĥmm . Are there only 7 layers when counting ifs? what about 'depends on's
- show some scrots of menuconfig/allnoconfig/gconfig
- mention that it is also called a preprocessor.

[9] refer to it when you have written it

3.6 gcc bug types

When a configuration has been created, the Linux kernel is compiled using that configuration file. This is done mainly by *gcc - The GNU Compiler Collection*, which then outputs warning and error messages. If there is an error, the compilation will also stop immediately (sometimes it will try to keep on going, but in this project, it is told to stop after one error).
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.

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

The relevant errors are reproducible errors that will occur on all machines when a certain configuration is being built. [10]

The irrelevant errors are a result of the building 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* 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*), [11] which leads to that every sixth compilation will stop with the same error.

There exist many other examples like this in the *Linux kernel* where an uninstalled program will result in an unbuilt kernel. These *irrelevant errors* would have to be minimized, since they are not actual kernel errors, but local errors. [12]
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 list is a *build system mandatory features* [13] list, which 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` [14]

See chapter 7 about Threats to Validity on page 17 for a more thorough description of the specific features.

[10] Refer to the list of errors - some link somewhere
[11] refer to the picture of gconfig and friends
[12] Give list of all the features I have disabled
[13] maybe find another word for this?
[14] This means that all features having to do with *lz4* were disabled

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 3.10 for a code example.
The relevant warnings are:

- Uninitialized variable
- Maybe uninitialized
- Overflow
- Deprecated declarations [15]

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

Figure 3.10: A toy program showing an uninitialized variable.

The program in figure 3.10 will return the warning '`'a' is used uninitialized in this function [-Wuninitialized]`' and the output, when the program is run, will be '`0Hello world`'.

The irrelevant warnings These are mainly warnings that represent dead code. 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 3.11 for a code example of an unused variable.
The irrelevant warnings are:

- Unused function
- Unused variable
- Unused label
- Unused value [16]

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

Figure 3.11: A program resulting in an unused variable warning.

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

- **TODO**
- Give real examples of unused function, and uninitialed... MAYBE

[15] Complete this list
[16] Complete this list

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 Linux kernels. One stable Linux kernel, and one from the linux-next repository.

4.1 Research Questions

The point of the stable versions are that they have been tested more, and generally are stable. duh! This should mean that fewer errors and warnings would occur when compiling stable versions.
When the unstable version is getting tested, many errors get fixed. An interesting question would be to see how many errors exist before all the testing happen. So how many errors are there before stabilization vs after. This leads to the first research question.

RQ1: Will the unstable linux-next repository generate more errors than the stable kernel, and how many more.

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 [1], the subsystems of the kernel are of very different character. [2] And therefore the bug types must differ from subsystem to subsystem.

RQ4: In what subsystems do the most errors occur. And in the largest subsystems, in what subsubsystem?

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

[1] Where is this mentioned about the subsystems? refer to it
[2] mention somewhere earlier, that drivers are different from security or kernel

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. It has 32 cores and 128 GB of RAM, and the mean 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.

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

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 subset of configurations, where generalization is representative, and figure 7.2 for the opposite.

Show an image of true representativeness

Figure 7.1: A representative subset

Show an image of true representativeness

Figure 7.2: A representative subset

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.

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.
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% [1] of the possible configurations to check.

[1] is this number true. Should I calculate?

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 diabled. This would require the host [2] 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)

[2] 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.