A Quantitative Analysis of Warnings in Linux
Draft: Monday 24th August, 2015 20:16

Elvis Flesborg
`efle@itu.dk`

September 1st 2015
IT University of Copenhagen

Contents

Abstract

—Leave blank until the end—

Chapter 1

Introduction

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

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

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

Chapter 2

Background

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

2.1 The Linux Kernel

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

It was first developed in 1991 by *Linus Torvalds*, and has since been growing. Today the code base is 19 million lines of code[2].

2.2 Variability

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

This is called *variability* in software and a software product of this type is called a *Software Product Line* (SPL). When different software programs can be derived from the same source code base.
Before compilation, the source code will have to be configured, at a preprocessing step, which will choose (either automatically, or the user will choose) which parts of the code should be included, by enabling or disabling a set of *features*.
The code in its entirety is not a valid program, it must be preprocessed. [5, p. 1]

Software with a high-degree of variability is usually refered to as *Variability-Intensive Systems* or *VISs*. Linux is a *VIS* with more than 10,000 different features[3]. Other examples of *VISs* are *Eclipse*, *Amazon Elastic Compute Service*, and *Drupal Content Management Framework* [10, p. 1] to name a few.

[1]See the **README** file in the Linux kernel tree
[2] `grep -r '.*' * | wc` in the kernel root folder
[3]14,387 across all architectures, with an average of 9,984 per architecture, and 10,335 for the `x86/` architecture

2.3 Linux Kernel Development

The Linux kernel development model is unique in many ways, since it has many thousand people world over, who contribute, but there is a very strict hierarchy where certain people have authority over a specific part of the kernel[4]

Comment from the Linux team or something...

Stable Releases

The Linux kernel development cycle has approximately 2¾ months from one stable release to the next stable release [6]. In the meantime, *Release Candidates (RCs)*, are released approximately once every week.
Then, when the top maintainers of the mainline tree think that the kernel is stable enough, a new stable version is created, and the whole process is repeated.

In-development Releases

The *linux-next* tree is a *git* repository, which merges over 200 other *git* repositories [7], which are all based on the *mainline* tree. The *linux-next* tree is merging these other trees every day and the merge conflicts are handled. The *linux-next* tree always contain the newest commits and is the main testing version, and the latest in-development version[5].

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

2.4 Inner Workings of Linux Kernel

This section will explain in coarse detail, the structure of the Linux kernel. From the directory structure, over configuration, to compilation of the kernel.

2.4.1 Subsystems

The directories in the root folder of the Linux kernel source code are called *subsystems*.
The `drivers/` subsystem is by far the largest subsystem. It contains all hardware drivers. It is also mostly contributed to by hardware vendors. [6] 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.

The `arch/` subsystem contains architecture specific source code. There are 29 architectures in the `arch/` folder, which is why Linux is the operating system, which supports the most architectures in the world. [1] .

The `mm/` subsystem is for memory management, `security` is libraries regarding security, and `kernel` is the where the kernel specific code is. Other subsystems are `sound/`, `net/`, and `lib/`

[4]See the `MAINTAINERS` file in the root folder of the Linux kernel
[5]See the original post about it here: https://lkml.org/lkml/2008/2/11/512
[6]Source? and is it true? -What would Greg Kroah Hartman say?

2.4.2 Feature Models

A feature model is a way of representing all the possible configurations - *the configuration space.* It contains all the features with their respective options and all the constraints and dependencies. A visualization of a feature model is called a feature diagram, there is an example in Figure 2.1. The example will be tiny compared to that of the Linux kernel. The feature model of the Linux kernel is too big to fit in a normal sized report.
In Figure 2.1 there is a toy example of a feature diagram. It depicts a feature model of a phone configuration, where there are some mandatory features (**Screen** and **Calls**) and some optional (**Media** and **GPS**), and also a choice option (**Color**, **BW**, and **High Definition**) for the screen type, where only one of them may be enabled. A cross-tree dependency is also present, which states that **Media** with all of its children can only be enabled if a **High Definition** screen is enabled.
There is no consensus on a unified notation for attributes in feature models [3]

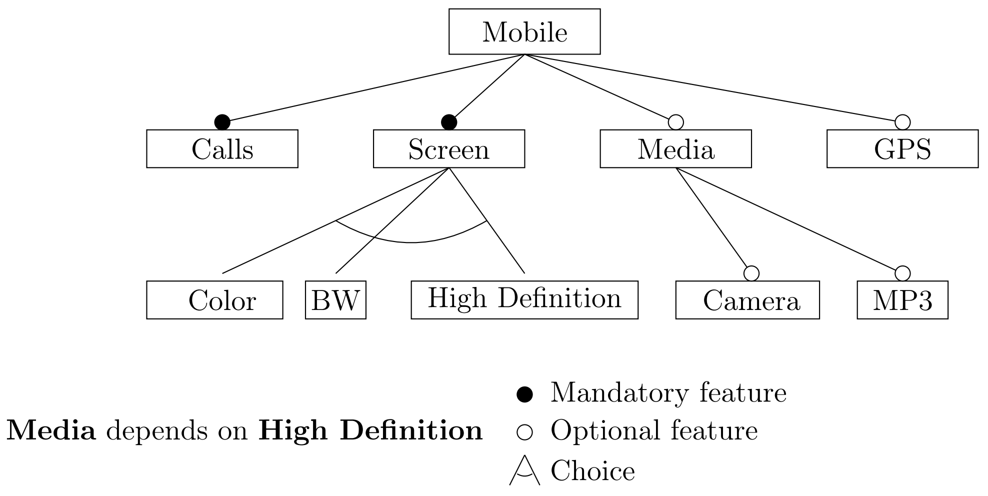


Figure 2.1: An example of a feature diagram of a phone

The Kconfig language is the language of the feature model in Linux (also used for other projects like BusyBox, BuildRoot, CoreBoot, Freetz and others) [4, p. 4] . The configuration files have the prefix `Kconfig` , and are scattered all over the Linux kernel source code tree, where they include each other. There are 1195 Kconfig files in total in the Linux kernel [7] with 956 of them relevant for the `x86/` architecture.

When a configuration is created, it is saved to a file called `.config`. In this file, all features are prefixed with `CONFIG_` .
The different data types and the percentage of them in the `x86` architecture are `boolean` (35%), `tristate` (61%), `string` (0.41%), `hex` (0.32%), and `integer` (3.7%). For a description of the context free grammar of the Kconfig language, see the appendices. [8]

The features in Kconfig can be split into four different types, *User features*, *Implementation features*, *Derived features*, and *Capability features* [4, p. 8] .

User features are features that the user chooses a value for in the configurator. In other words, a feature, that has a prompt option in the Kconfig language.

[7]Found with the command `find . | grep Kconfig | wc`
[8]Write the Backus Naur Form of Kconfig in the Appendix

```
mainmenu MOBILE
    config CALLS
        bool "Needed to make calls"
    config GPS
        bool "GPS location system"
    config MEDIA
        bool "Media modules"
        default y
    config SCREEN
        bool "Screen module"
        default y

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

    if HD
        config CAMERA
            bool "Camera support"
        config MP3
            bool "MP3 support"
    endif
endmenu
```

Figure 2.2: Kconfig code for the phone example

These features may not directly inflict the compiler's choices, but may only open up for other features in the configurator.

Implementation features are referenced in the source code, and will directly influence what code the preprocessor will select for compilation.

Derived features are invisible to the user, and are set by other features using the `select FOO` syntax.

Capability features in Linux are features, that simulate a reverse dependency. These features are also not randomly chosen by `randconfig` [9] . Their names often start with `CONFIG_HAVE_` or `CONFIG_ARCH_HAVE_` , and only have *type* option set (often to `bool`).
[10] [11] [12]

2.4.3 Configuring 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 an automated config: `allyesconfig` (enabling as much as possible), `allnoconfig` (disabling as much as possible), `defconfig` (choosing the default values for everything), and `randconfig` (choosing random values for everything).

See Figure 2.3 for a few examples of what the graphical configuration tools look like.

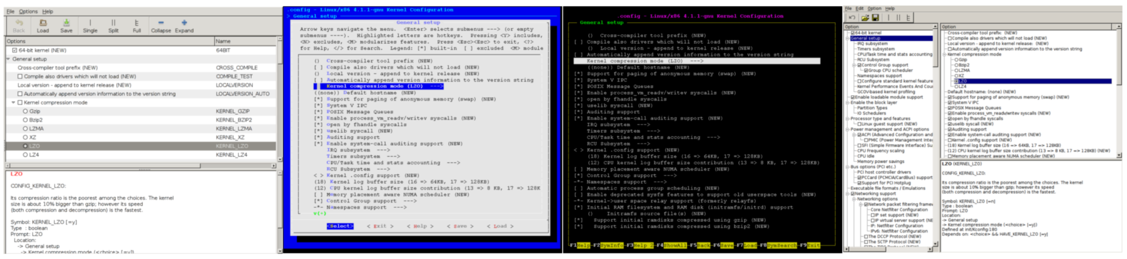


Figure 2.3: gconfig, menuconfig, nconfig, and xconfig

2.4.4 Experimental Setup

The compilation is done mainly by *GCC - The GNU Compiler Collection* [13].
During compilation, `GCC` will output warnings when there are any [14] .
In a warning, there is information about a possible coding mistake that might break the code. There are about 30 different warning types that are enabled by default [11] Some of them are more severe than others.
The warnings have the output format `[-Wwarningtype]` , and this is output whenever there is a warning. This makes the warnings very easy to quantify automatically.

[9] Should I give an example here of the capability one?
[10] describe that all models are typically wide and shallow, VarModSSD page 19
[11] write how Kconfig is a beast sometimes. Child nodes exclusding their parents
[12] How many are there of each of these. distirbution.
[13] `https://gcc.gnu.org`
[14] There are on average 12 warning messages per compilation

If an error occurs during compilation, `GCC` will stop, and give an error message. The error messages though, are not as fulfilling as the warning messages. Often an erroneous subprocess will return an errorcode to `GCC`, which will then stop.
All that is shown is the errorcode, and the directory in which it occured.
[15]

An error need not be a bug in the Linux kernel code. It can be a missing program on the machine that is compiling the code (the *host machine*). These will be called *host errors*.

2.4.5 GCC Warnings

Here follows a list of the warnings that was found during the compilations. They are ordered alphabetically.

array-bounds is given, when GCC is certain that a subscript to an array is always out of bounds.

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

deprecated-declarations will show warnings when functions, variables, or types have been declared deprecated be a programmer.

discarded-array-qualifiers

error= is a prefix, which means that the warning will cause an error. Sometimes developers do not want a specific warning to happen without the compilation stopping. Then they can enable this flag in the makefile.
There is also a `no-error=`, which nullifies the `error=` flag.

frame-larger-than=

implicit-function-declaration

incompatible-pointer-types

int-conversion

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

logical-not-parentheses occurs when the left hand side of an expression contains a *logical not* (a `!` in C). An example is `if (!ret == template[i].fail)` [16] .
The warning refers to the `!ret` , which should have been inside parentheses to be correct, like so `(!ret)` .

[15] example of an error message
[16] From the file `crypto/testmgr.c`

maybe-uninitialized is when there is an uncertainty about a variable being uninitialized. In the case in Figure 2.4 there is a `switch-case` where `sgn` is not initialized in all of the cases. If *GCC* cannot see for sure that the variable is initialized, even if it would have been initialized in all of the cases, it will return this warning.

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

  while (n) {

    switch (node->action) {
    case BTRFS_ADD_DELAYED_EXTENT:
    case BTRFS_UPDATE_DELAYED_HEAD:
      WARN_ON(1);
      continue;
    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 2.4: A real example of maybe-uninitialized function

overflow

pointer-to-int-cast This warning would normally be sign of bad code. It is usually when a pointer is cast to a `long` and then later cast back into a function call. This is widely spread in Linux to simulate the effects of *Object Oriented Programming.* [17]

return-type

switch-bool

uninitialized

unused-function is a warning about a function that has never been called. This represents dead code, or code pollution.
In the example in Figure 2.5, the function `bq27x00_powersupply_unregister` will not be called if the feature `CONFIG_BATTERY_BQ27X00_I2C` is not enabled, and is therefore an unused function.

unused-variable is basically the same as `unused-function`, but only with a variable instead of a function. There will be no example of this.

unused-label /iffalse . declare each warning either dangerous or not-dangerous. /fi

[17]source: Claus... find other source.

```c
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);
    mutex_lock(&battery_mutex);
    idr_remove(&battery_id, di->id);
    mutex_unlock(&battery_mutex);
    return 0;
}

#endif
```

Figure 2.5: A real example of unused function - from the file `drivers/power/bq27x00_battery.c`

Chapter 3

Methodology

Objective: The objective is to make a quantitative analysis of warnings in all of the Linux kernel by checking randomly generated Linux kernels. The warnings functions as a proxy for errors. These are the research questions:

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

RQ2: Where do most warnings occur?

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

Subject: 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 an in-development version of the Linux kernel.

Methodology: The methodology will be in three parts. First part is finding a way of generating random configurations. Second part is compiling the Linux kernel using these configurations while obtaining information about the compilation. Third part is analyzing the data.

3.1 Generating Random Configurations

The configurator `randconfig` is used in this project. It is used as a proxy for totally representative configurations. The truth is that `randconfig` is not representative, but is biased towards features higher up in the feature model tree.

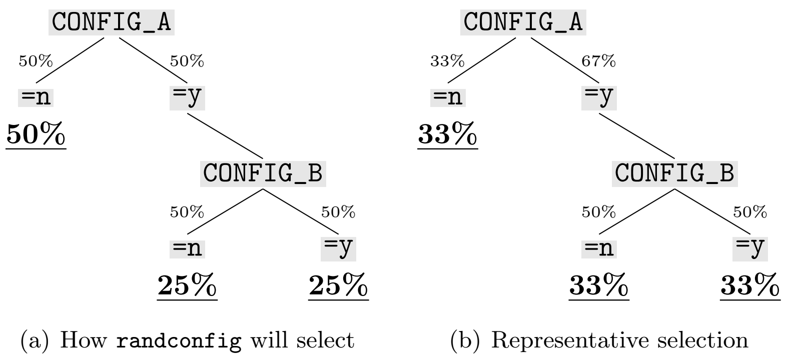
Figure 3.1 shows a toy example of a KCONFIG feature model written in the KCONFIG language. It is a very small example with only two features, but it will easily explain some limitations of using `randconfig`.

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`.
Figure 3.2(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.

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

Figure 3.1: A toy KCONFIG feature model

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



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

Figure 3.2

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. more about representativeness in the section 3.1.1.

Even though **`randconfig`** generates valid configurations, there were still some features that showed to be creating bogus errors.

For instance, there is a feature in the feature model that is called `CONFIG_KERNEL_LZO` , which enables kernel compression with the *LZO library*.
If the *LZO library* is not installed on the host machine, then the compilation will fail.

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* package however was not. [1]

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.

[1]be sure of that the warnings are ONLY for the linux kernel and not *host warnings*

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` [2]

See chapter 5 about Threats to Validity on page 19 for a more thorough description of the specific features.

3.1.1 The Hunt for Representativeness

To make the sample of configurations representative, three different ways are proposed, and researched if feasible. They are *changing randconfig*, *permuting* KCONFIG, and *generate and filter*.

To change `randconfig` to be representative, one will have to make it aware of the whole dependency tree before randomly selecting features. This seems infeasible since the configuration space is so large. It will not fit in memory.

The next proposal is to *permute the* KCONFIG files. First by desugaring the KCONFIG files to be in one *level* only [3] , and then randomly scramble the feature clauses in the files. If the `randconfig` script then loads the KCONFIG files from the top to the bottom, it will 'flip the coin' starting a random place each time.
This will not work, as `randconfig` does not flip the coin when reading the KCONFIG files. Rather, it reads the KCONFIG files, and then 'flips the coin' on the dependency tree.

The last way of generating a representative sample of configurations is a method which is dubbed *generate and filter*.
`.config` files are generated, and the invalid ones are filtered away.

In the generation process, some mandatory features are given fixed values [4] . All other values are randomly generated.
Unfortunately this method never did find a valid configuration even after generating 100,000 configurations.

Since none of the three proposed methods works, `randconfig` has become the preferable choice.

3.2 Compiling and Collecting Data

When a configuration has been created, the Linux kernel is compiled using that configuration file by running the command `make all` . This command runs *GNU Make*, which is instructed how to compile the kernel with *GCC - The GNU Compiler Collection*.
GCC will output warnings and error messages, which is saved for analysis.

The compilations has mainly been done on a computer at the IT University of Copenhagen. The computer has $32 \times 2.8MHz$ and $128GB$ of RAM, and the average time to compile a kernel and

[2] This means that all features having to do with *lz4* were disabled
[3] For example by replacing `if FOO` clauses by `depends on` options in the features clauses.
[4] These are, among few others, features from the `allnoconfig`

upload the data was around 1 minute and 35 seconds. A fairly regular laptop with 4 cores and 8 GB of RAM does that in just over 8 minutes on average.

Every time a compilation is running, *GCC* will output warning and error messages in the *standard error* output. This output is then scraped through for categorization.
An output line will contain a *bug type*, a *filename*, *line number*, and a *message* describing the warning in english.

When a compilation is done, the output is scraped and categorized, and then uploaded to a database, for easy querying during and after the project.
See the Figures 3.3, 3.4, and 3.5 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 3.3: 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 3.4: 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 3.5: The Files table referring to the Bugs table

3.3 Analyzing Data

This report is a quantitative analysis in comparisson to [2] , which is a qualitative analysis of bugs in Linux.
In this paper, warnings are analysed, not errors, seeing as the format of the error messages prohibits to automatically make sense of categorization.

3.4 Original Objective

The original objective of this report was to categorize errors and not warnings, and make a quantitative analysis on those. The majority of the error messages do not have an error type classified, but just a filename, and a message.

Chapter 4

Results

A total of 42,060 configurations were compiled. Half of them from the unstable Linux version, and the other half a stable version of Linux.

4.0.1 Stable

In table 4.1 is shown the distribution of warnings in the stable Linux kernel. The warnings that will cause code pollution are typed in grey and the more severe ones, that might cause a compilation error is typed in black.
17% of the compilations stopped with an error. These errors were often host errors, 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.

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

Figure 4.1: Distribution of warnings in the stable kernel

4.0.2 Subsystems With Errors

Every warning with a subsystem is distributed as in Figure 4.2. Many warnings contain multiple subsystems, so the percentages will not equal 100%.

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

Figure 4.2: Distribution of all subsystems with warnings

4.1 Analyzing the data

4.2 Number of valid configurations

There was a hope, that something could be said about the percentage of valid configurations out of all possible combinations.
Some mandatory features were fixed, and an experiment was run, but after 100,000 configurations, none of them were valid. This can only be used to say something about the upper bound of the percentage.

4.3 Observations

4.3.1 Configuration Warnings

When creating a random configuration, 25% [1] of the times, it will output a warning about unmet dependencies. One might suspect that configurations, which yielded a warning would result in compilation errors.
As can be seen in Figure 4.5, fewer of the stable kernels that had configuration warnings also had an error in the compilation. But the rise in erroneous compilations from configurations without config warnings to configurations with config warnings is very little, and is claimed irrelevant.

[1]get the correct percentage

ss	errors	loc [M]	err/loc
usr	2471	845	290 %
samples	2471	7,500	32%
include	2421	692,000	0.35 %
drivers	1405	10,800,000	0.013 %
lib	99	1,050,00	0.094 %
init	1	5,730	0.017 %
ipc	1	8,860	0.011 %
net	88	890,000	0.0099 %
crypto	8	83,900	0.0095 %
arch/x86	26	343,000	0.0076 %
scripts	5	90,700	0.0055 %
sound	26	876,000	0.0030 %
mm	3	108,000	0.0028 %
fs	31	1,170,000	0.0026 %
kernel	6	244,000	0.0025 %
arch	26	3,440,000	0.00076 %
firmware	0	129,000	0.00046 %
virt	0	10,500	0 %
tools	1	218,000	0 %
block	0	38,100	0 %
security	0	75,500	0 %

Figure 4.3

Subsystem	**Percent**
samples	68%
usr	68%
include	67%
drivers/	39%
lib	2.7%
net	2.4%
fs/	0.86%
sound	0.72%
arch/	0.71%
crypto/	0.22%
kernel	0.17%
scripts	0.13%
mm/	0.083%
init	0.028%
ipc	0.028%
security	0.0%
block/	0.0%

Figure 4.4: Percentage of subsystems that were present in errorneous compilations

	Stable	**Unstable**
No config warnings	17%	36%
Config warnings	19%	40%

Figure 4.5: Percentage of the configs with and without warnings, which resulted in an erroneous compilation

/iffalse . write how many bugs/config on average there were. . mention that security has none. /fi

Chapter 5

Threats to Validity

Blahblah about Threats to validity...

5.1 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 5.1 for a visualization of subsets which are both representative, and not representative.

Show an image of true representativeness

Figure 5.1: A representative subset

Three ways was thought of, that would make the configurations more generalizable. If successful with any of the methods, they could have been committed to the Linux kernel. Unfortunately, none of them were successful.

Changing the code of `randconfig`

The first idea was to change the code of `randconfig`, so it would know what the possibilities of each outcome should be. See Figure **??** for a visualization.
This was deemed impossible due to the very large feature model. The *Abstract Syntax Tree (AST)* would simply not fit in memory of any computer. [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 and exposed to syntactic desugaring. 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.

It turned out, that `randconfig` does not regard the order of the kconfig files. The KCONFIG files are loaded, and `randconfig` then utilizes the AST and randomly selects from the root of that, which gives the same result as when KCONFIG is not permuted.

[1]Give short example

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, but after generating 100,000 configurations, not one valid configuration was found.
It was considered running *SharpSAT* (or *#SAT*) to count the number of valid configurations, but it was deemed out of scope of this project. [2]

5.1.1 Features in Stable vs. Unstable

In the unstable Linux version, there are 20 more features than in the stable one. This means that there is a possibility that errors might happen in this new code.
Configurations are created in the unstable version, and when the stable version runs the same configuration file, it will just disregard these unknown features.

If it was done the other way around, the unstable version would always set the new features to the default value, and they would therefore not be randomized.

5.1.2 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 [3] , and none of this type was found.

5.1.3 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% [4] of the possible configurations to check.

5.1.4 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.

[2] Link to sharpSAT thingy
[3] `-Wc++14compat`
[4] is this number true. Should I calculate?

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 [5] 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?

5.2 Generate'n'Filter

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

5.3 elvisconfig

5.4 RandomSAT

[5] explain what a host and target are at some point further up

Chapter 6

Related Work

Chapter 7

Conclusion

— *Leave empty until the end*—

Bibliography

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

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

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

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

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

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

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

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

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

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

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

Chapter 8

Appendices

8.1 Kconfig language

8.2 Data