A Quantitative Analysis of Warnings in Linux
Draft: Tuesday 25th August, 2015 17:24

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]

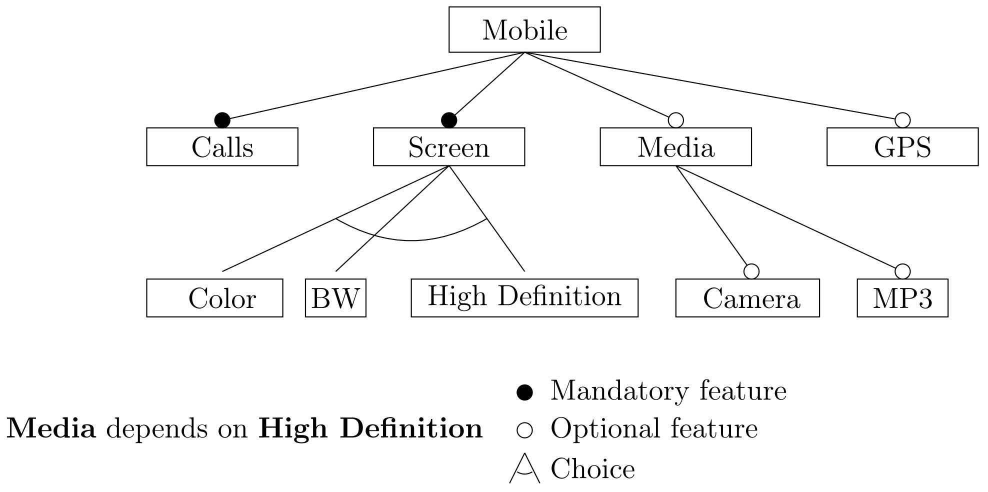


Figure 2.1: An example of a feature diagram of a phone

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]

[9] [10] [11]

[7]Found with the command `find . | grep Kconfig | wc`
[8]Write the Backus Naur Form of Kconfig in the Appendix
[9]describe that all models are typically wide and shallow, VarModSSD page 19
[10]write how Kconfig is a beast sometimes. Child nodes exclusding their parents
[11]How many are there of each of these. distirbution.

```
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

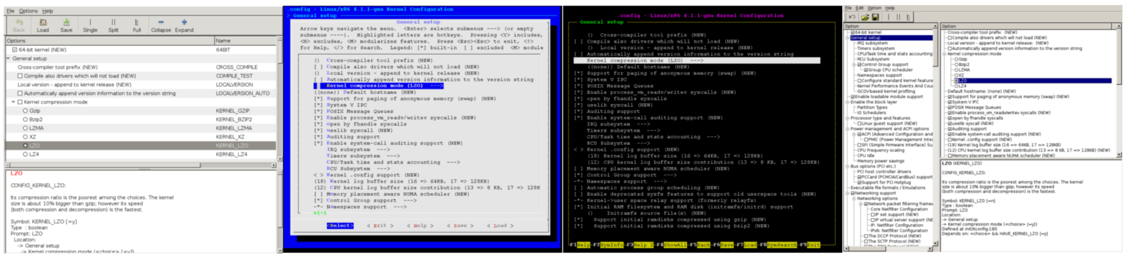
2.4.3 Configuring Linux

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

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

- `allyesconfig` (enabling as much as possible)
- `allnoconfig` (disabling as much as possible)
- `defconfig` (choosing the default values for everything)
- `randconfig` (choosing random values for everything).

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



```
Options                                                         Name
 ☑ 64-bit kernel (NEW)                                          64BIT
 ▾ General setup
    Cross-compiler tool prefix (NEW)                            CROSS_COMPILE
    ☐ Compile also drivers which will not load (NEW)            COMPILE_TEST
    Local version - append to kernel release (NEW)              LOCALVERSION
    ☐ Automatically append version information to the version string   LOCALVERSION_AUTO
  ▾ ☐ Kernel compression mode
      ○ Gzip                                                    KERNEL_GZIP
      ○ Bzip2                                                   KERNEL_BZIP2
      ○ LZMA                                                    KERNEL_LZMA
      ○ XZ                                                      KERNEL_XZ
      ● LZO                                                     KERNEL_LZO
      ○ LZ4                                                     KERNEL_LZ4

LZO

CONFIG_KERNEL_LZO:

Its compression ratio is the poorest among the choices. The kernel
size is about 10% bigger than gzip; however its speed
(both compression and decompression) is the fastest.

Symbol: KERNEL_LZO [=y]
Type  : boolean
Prompt: LZO
  Location:
    -> General setup
      -> Kernel compression mode (<choice> [=y])
```

```
.config - Linux/x86 4.1.1-gnu Kernel Configuration
> General setup
                              General setup
  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty
  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y> includes,
  <N> excludes, <M> modularizes features.  Press <Esc><Esc> to exit, <?>
  for Help, </> for Search.  Legend: [*] built-in  [ ] excluded  <M> module

      ()  Cross-compiler tool prefix (NEW)
      [ ] Compile also drivers which will not load (NEW)
      ()  Local version - append to kernel release (NEW)
      [ ] Automatically append version information to the version string
          Kernel compression mode (LZO)  --->
      ((none)) Default hostname (NEW)
      [*] Support for paging of anonymous memory (swap) (NEW)
      [*] System V IPC
      [*] POSIX Message Queues
      [*] Enable process_vm_readv/writev syscalls (NEW)
      [*] open by fhandle syscalls
      [*] uselib syscall (NEW)
      [*] Auditing support
      [*] Enable system-call auditing support (NEW)
          IRQ subsystem  --->
          Timers subsystem  --->
          CPU/Task time and stats accounting  --->
          RCU Subsystem  --->
      < > Kernel .config support (NEW)
      (18) Kernel log buffer size (16 => 64KB, 17 => 128KB)
      (12) CPU kernel log buffer size contribution (13 => 8 KB, 17 => 128K
      [ ] Memory placement aware NUMA scheduler (NEW)
      [*] Control Group support  --->
      -*- Namespaces support  --->
      v(+)

        <Select>    < Exit >    < Help >    < Save >    < Load >
```

```
                .config - Linux/x86 4.1.1-gnu Kernel Configuration
 General setup
     ()  Cross-compiler tool prefix (NEW)
 [ ] Compile also drivers which will not load (NEW)
     ()  Local version - append to kernel release (NEW)
 [ ] Automatically append version information to the version string
     Kernel compression mode (LZO)  --->
     ((none)) Default hostname (NEW)
 [*] Support for paging of anonymous memory (swap) (NEW)
 [*] System V IPC
 [*] POSIX Message Queues
 [*] Enable process_vm_readv/writev syscalls (NEW)
 [*] open by fhandle syscalls
 [*] uselib syscall (NEW)
 [*] Auditing support
 [*] Enable system-call auditing support (NEW)
     IRQ subsystem  --->
     Timers subsystem  --->
     CPU/Task time and stats accounting  --->
     RCU Subsystem  --->
 < > Kernel .config support (NEW)
     (18) Kernel log buffer size (16 => 64KB, 17 => 128KB)
     (12) CPU kernel log buffer size contribution (13 => 8 KB, 17 => 128KB)
 [ ] Memory placement aware NUMA scheduler (NEW)
 [*] Control Group support  --->
 -*- Namespaces support  --->
 [ ] Automatic process group scheduling (NEW)
 [ ] Enable deprecated sysfs features to support old userspace tools (NEW)
 -*- Kernel->user space relay support (formerly relayfs)
 [*] Initial RAM filesystem and RAM disk (initramfs/initrd) support
     ()    Initramfs source file(s) (NEW)
 [*]   Support initial ramdisks compressed using gzip (NEW)
 [*]   Support initial ramdisks compressed using bzip2 (NEW)

F1Help F2SymInfo F3Help 2 F4ShowAll F5Back F6Save F7Load F8SymSearch F9Exit
```

```
Option
 ☑64-bit kernel
 General setup
   IRQ subsystem
   Timers subsystem
   CPU/Task time and stats accounting
   RCU Subsystem
   ☑Control Group support
     ☑Group CPU scheduler
   Namespaces support
   ☐Configure standard kernel feature
   Kernel Performance Events And Cou
   GCOV-based kernel profiling
 ☑Enable loadable module support
 Enable the block layer
   Partition Types
   IO Schedulers
 Processor type and features
   ☐Linux guest support (NEW)
 Power management and ACPI options
   ☑ACPI (Advanced Configuration and
     ☐PMIC (Power Management Inte
   ☐SFI (Simple Firmware Interface) Su
   CPU Frequency scaling
   CPU Idle
   Memory power savings
 Bus options (PCI etc.)
   PCI host controller drivers
   ☑PCCard (PCMCIA/CardBus) support
   ☑Support for PCI Hotplug
 Executable file formats / Emulations
 ☑Networking support
   Networking options
     ☑Network packet filtering framew
       Core Netfilter Configuration
       ☐IP set support (NEW)
       ☐IP virtual server support (NE
       IP: Netfilter Configuration
       IPv6: Netfilter Configuration
     ☐The DCCP Protocol (NEW)
     ☐The SCTP Protocol (NEW)

Option
 Cross-compiler tool prefix:  (NEW)
 ☐Compile also drivers which will not load (NEW)
 Local version - append to kernel release:  (NEW)
 ☐Automatically append version information to the version string
 Kernel compression mode
   ○Gzip
   ○Bzip2
   ○LZMA
   ○XZ
   ●LZO
   ○LZ4
 Default hostname: (none) (NEW)
 ☑Support for paging of anonymous memory (swap) (NEW)
 ☑System V IPC
 ☑POSIX Message Queues
 ☑Enable process_vm_readv/writev syscalls (NEW)
 ☑open by fhandle syscalls
 ☑uselib syscall (NEW)
 ☑Auditing support
 ☑Enable system-call auditing support (NEW)
 ☐Kernel .config support (NEW)
 (18) Kernel log buffer size (16 => 64KB, 17 => 128KB)
 (12) CPU kernel log buffer size contribution (13 => 8 KB, 17 => 128KB) (NEW)
 ☐Memory placement aware NUMA scheduler (NEW)

LZO (KERNEL_LZO)

CONFIG_KERNEL_LZO:

Its compression ratio is the poorest among the choices. The kernel
size is about 10% bigger than gzip; however its speed
(both compression and decompression) is the fastest.

Symbol: KERNEL_LZO [=n]
Type : boolean
Prompt: LZO
Location:
-> General setup
-> Kernel compression mode (<choice> [=y])
Defined at init/Kconfig:180
Depends on: <choice> && HAVE_KERNEL_LZO [=y]
```

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

2.4.4 Compiling and Catching Warnings

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

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

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.
12

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.

[12]example of an error message

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 7*

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)` [13] .
The warning refers to the `!ret` , which should have been inside parentheses to be correct, like so `(!ret)` .

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.

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.* [14]

return-type

switch-bool

uninitialized

[13]From the file `crypto/testmgr.c`
[14]source: Claus... find other source.

```
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

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.

```
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`

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

Chapter 3

Methodology

Objective: This report aims to make a quantitative analysis of warnings in all of the Linux kernel by checking randomly generated Linux kernels for warnings.
This includes addressing the following research questions:

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

RQ2: Where do most warnings occur?

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

Subject: To respond to these questions there will be generated random configurations, and these will be used to compile two different versions of the Linux kernel, the latest stable Linux kernel version, and a two months old in-development version of the Linux kernel.
The warning messages will be categorized and collected, and be subject for analysis.

Methodology: The methodology will be in three parts. First part is finding a way of generating random configurations in a representative way. Second part is collecting any warnings that a compilation might return. Third part is analyzing the data and answering the research questions.

Since there are more than 10,000 different features in the feature model of the Linux kernel, there will be approximately $2^{10,000}$ possible configurations[1], which is more than the estimated number of atoms in the universe. So getting a list of all the possible configurations is not possible (at least not with 2 bit computers).

3.1 The Hunt for Representativeness

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

1. **Using randconfig**

   Using the built-in `randconfig`

2. **Changing randconfig**

   This method will rewrite the code for `randconfig` or create a new configurator (eg. `reprandconfig`). This configurator must not have the bias for features in the upper levels of the dependency tree, as `randconfig` has.

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

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

3. **Permuting KCONFIG**

   This proposal will desugarize the KCONFIG files to be in one level only[2], and then randomly scramble the position of the feature clauses in the files.

   The hope is that the `randconfig` script loads the KCONFIG files from top to bottom, and will then load in the features at random each time.

4. **Generate and filter**

   In this method, a configuration file is generated with a script, which does not know the relation betweeen all the features. It knows all the feature names, and the possible values for the features[3].

   It goes through the list and randomly selects values for all the features.

   Then all invalid configurations are filtered away.

5. **RandomSAT**

   The Linux kernel feature model can be extracted from the KCONFIG files, to get a propositional formula [11]. This formula can be used to check for propositional satisfiability.

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

3.2 Experimental Setup

The experimental setup consists of a loop, where a configuration is generated, the Linux kernel is then compiling with this configuration, and the output warnings are categorized and collected.

To say something about all of the Linux kernel, the generated configurations for the experiment should be a representative sample of all the possible configurations. Every single configuration should have equal likelihood of being chosen.

3.2.1 Compiling and Collecting Data

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

The compilations has mainly been done on a computer at the IT University of Copenhagen. The computer has $32 \times 2.8 MHz$ and $128 GB$ of RAM, and the average time to compile a kernel and upload the data was around 1 minute and 35 seconds. A fairly regular laptop with 4 cores and 8 GB of RAM does that in just over 8 minutes on average.

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

[2]For example by replacing `if FOO ... endif` clauses by `depends on` options in the features clauses.
[3]It also is aware about `choice` clauses

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

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

3.3 Analyzing Data

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

Chapter 4

Results

A total of 42,060 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

4.1 Subsystems With Warnings

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

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

Chapter 5

Threats to Validity

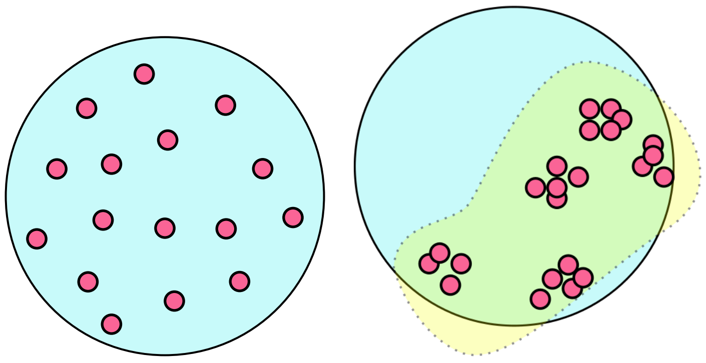
5.1 External Validity

5.1.1 Generalization

All the selected configurations are not a representative subset of all possible configurations, and therefore does not fully represent all of the Linux kernel. The method by which the configurations are selected, does not select configurations between the whole configuration space equally.
There is a bias towards selecting configurations that will not visit the deepest layers of the dependency tree.
In Figure 5.1 a blue area represents the whole configuration space, and red dots inside a blue area represent a subset of all the configurations, which is in a given sample.

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

Figure 5.1(b) shows a sample, which is not representatively distributed. All the dots tend to cluster together around certain areas of the configuration space, and if one was to tell what the configuration space looked like by only looking at the sample, it would look like the yellow area.



(a) A representative sample (b) An unrepresentative sample

Figure 5.1: A representative subset

5.2 Internal Validity

5.2.1 9 Different In-Development Versions

There was used multiple different in-development versions of the Linux kernel to minimize the skewing of warnings. One could argue that 9 different versions was not enough to get rid of over-representation of certain warnings.

5.2.2 More Features in the In-Development version

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.

5.2.3 Gcc versions

Roughly a third of the compilations were done with GCC version 5.1.0 and two thirds with GCC version 4.9.2. This should not matter on what warnings are returned. There is only one new warning that is enabled by the `-Wall` flag in version 5.1.0 [1] , and none of this type was found.

5.2.4 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.
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.
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`

[1] `-Wc++14compat`
[2] explain what a host and target are at some point further up

- `CONFIG_PREVENT_FIRMWARE_BUILD=y`
- `CONFIG_*LZ4*=n` [3]

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

[3] This means that all features having to do with *lz4* were disabled

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

[12] 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