BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

REFACTORING AND VERIFICATION OF THE CODE OF MKFS XFS

REFAKTORING A VERIFIKACE KÓDU MKFS XFS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. JAN ŤULÁK**
AUTOR PRÁCE

SUPERVISOR **prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2017

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems

Academic year 2016/2017

Master's Thesis Specification

For: **Ťulák Jan, Bc.**

Branch of study: Mathematical Methods in Information Technology

Title: **Refactoring and Verification of the Code of mkfs xfs**

Category: Software analysis and testing

Instructions for project work:

1. Get acquainted with the xfs journalling file system and with the code of mkfs xfs.
2. Study code analysis and verification techniques applicable on the code of mkfs xfs, including both light-weight approaches (e.g., searching for error patterns) as well as heavy-weight approaches (model checking).
3. Propose and implement a refactoring of the code of mkfs xfs with the aim of enhancing its maintainability and testability.
4. Propose a combination of light-weight and heavy-weight techniques suitable for analysis and verification of the refactored code of mkfs xfs and apply it on the code.
5. Discuss the obtained results and propose possible future improvements of your work.

Basic references:

- WiKi pages of project XFS, http://xfs.org/index.php/Main_Page.
- Křena, B., Vojnar, T.: Automated Formal Analysis and Verification: An Overview, In: International Journal of General Systems, 42(4):335-365, Taylor and Francis, 2013.
- Beyer, D., Erkan Keremoglu, M.: CPAchecker: A Tool for Configurable Software Verification, In: Proc. of CAV'11, LNCS 6806, Springer-Verlag, 2011.

Requirements for the semestral defense:

First two items plus at least some initial proposal of how to proceed with items 3 and 4.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Vojnar Tomáš, prof. Ing., Ph.D.**, DITS FIT BUT

Beginning of work: November 1, 2016

Date of delivery: May 24, 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

Petr Hanáček

Associate Professor and Head of Department

Abstrakt

Tato práce popisuje průběh refaktoringu programu mkfs.xfs za účelem zpřehlednění jeho kódu a vyčištění technického dluhu naakumulovaného za dvacet let existence tohoto programu, a následně jeho statickou analýzu. Použité nástroje (CppCheck, Coverity, Codacy, GCC, Clang) jsou srovnány z hlediska počtu i typu nalezených chyb.

Abstract

This work describes the processes of refactoring mkfs.xfs program for a purpose of refining its code and cleaning the technical debt accumulated over 20 years of the program's existence. The mkfs.xfs source code is then a subject to static analysis and the used tools (CppCheck, Coverity, Codacy, GCC, Clang) are compared in terms of the number and type of the found defects.

Klíčová slova

XFS, refaktoring, formální analýza, formální verifikace, Srovnání, Coverity, Codacy, GCC, Clang, CppCheck

Keywords

XFS, refactoring, formal analysis, formal verification, comparison, Coverity, Codacy, GCC, Clang, CppCheck

Citation

Jan Ťulák: Refactoring and Verification of the Code of mkfs xfs, diplomová práce, Brno, FIT VUT v Brně, 2017

Refactoring and Verification of the Code of mkfs xfs

Declaration

Hereby I declare that I wrote this work on my own and all used sources are stated and correctly noted as citations.

.....
Jan Ťulák
May 23, 2017

Acknowledgement

I want to thanks to my managers Eric Sandeen and Steven Whitehouse, my colleagues at Red Hat, to David Chinner, maintainer of XFS, and to everyone in XFS community.

Contents

1 Introduction

In software projects with long life, even an initially clean codebase can become messy and complicated. Moreso when we speak about open-source projects where the original creators left yers ago and new people of various capabilities and knowledge continue the development.

In such projects, new functionality is added to the existing code with minimal changes to the rest of the project. This may simplify the merging of these changes, as any responsible person can easily understand what the change does. But on the other side, in the long term, it turns the code into a disordered chaos.

The result is increasingly more difficult to maintain and test, and as a single functionality can be spread over many portions of the project, any change requires more and more attention and time.

xfsprogs, a package of tools for XFS filesystem, is such a project. While the filesystem itself is subject to careful scrutiny from the Linux kernel community, the tools like `mkfs.xfs`[1], `fsck.xfs`[2] and others are not so publicly exposed and get a lot less attention. From our experience with working on this project, it happens that only one or two persons other than the author of a patch may read it, and miss some subtle side effect the change has. Sometimes, the large set of tests XFS maintains captures this bug, sometimes it does not and it is noticed much later.

On this point, it is important to highlight that despite the naming convention, each `mkfs` tool is completely independent project and, for example, `mkfs.xfs` and `mkfs.ext4` do not share any code except system libraries.

Some parts of this code are more than 20 years old (see chapter 1 for a detailed history of XFS) and in need of intensive cleaning. The test suite (project xfstests) maintains hundreds of more or less complex tests, but these are limited in what they can detect as they usually work in this way: make a filesystem, then test that, so many errors in `mkfs.xfs` are difficult to capture or notice. XFS also uses an automatic static analysis from Coverity, which is useful, but the project has no good data on the reliability of this analysis.

With the approval of David Chinner, then the maintainer of XFS, we began the refactoring of mkfs.xfs, which was overdue. The goal of this work is to repay the technical debt accumulated over the years. That means not only fixing some long-known issues and cleaning particularly complex parts of the code, but also making structural changes to minimize the amount of code that must be added or changed during the regular development (adding and removing features of the filesystem). These changes should slow the build-up of the technical debt in the future.

After implementing these changes, this work should verify how effective the currently used tests and analysis are. Even if some testing and analysis methods can be used only on a part of the code, the results, when compared with other tools, still provides an estimate about the soundess and completeness of every used method.

[1] Formats a partition as XFS.

[2] Usually checks and repairs errors in an existing filesystem. But for XFS it only tells the user what other tools to use.

At the same time, this work can also be seen as a review of how well various analysis and verification methods perform on real and in-production code.

The refactoring was done in two parts. One set of changes was merged into upstream in June 2016 (xfsprogs 4.7[3]), the other set is, at the time of writing, still in development. The versions of xfsprogs at different stages of this work were:

- Before the beginning of refactoring – xfsprogs 4.6.
- After merging the first part – xfsprogs 4.7.
- Before merging the second part – xfsprogs 4.11 at the time of writing.
- After applying the second part – not yet merged, changes only in a local repository.

This work is structured as follows: First, information about XFS and mkfs.xfs are provided in the first chapter. In the following, third chapter, we look at the refactoring done and discuss the changes. After that, another chapter is dedicated to explaining formal analysis and verification, describing common techniques, and we point out notable tools, from which we select few to use in the fifth and sixth chapter, where the testing environment is described and results analysed.

[3]The releasing of xfsprogs is tightly coupled with releases of XFS kernel module, which is part of Linux. Thus, xfsprogs uses the same version number that the respective XFS kernel module and Linux has.

2 XFS filesystem

XFS is a journaling filesystem created by SGI in 1993. The new filesystem, intended as a powerful replacement of EFS with the expectation of growing amount of data in the future was first released in IRIX 5.3 in 1993 [30]. The Linux port began in 1999 and since 2002 XFS has been accessible in the mainline Linux Kernel [31, Chap. 1.2, 1.3].

XFS is actively developed for all its history since 1993, making it one of the oldest filesystems in active use on modern Linux machines [8, 40:25].

	ext3	ext4	XFS	NTFS
max fs size	1 EiB	16 TiB	16 EiB	256 TiB
max file size	8 TiB	16 TiB	8 EiB	256 TiB
max files	2^{32}	2^{32}	2^{64}	2^{32}
date resolution	1 s	1 ns	1 ns	100 ns

Table 2.1: Comparison of various filesystems and their limits. Sources: [29, 22, 21, 20, 16, 27].

Because of its capabilities, XFS is used by well known institutions like CERN and Fermilab [16] for storing large amounts of data. Unlike most other Linux filesystems, XFS is a 64bit filesystem, meaning it provides far greater limits for storage and file size[1], but its architecture also offer great scalability in terms of parallel I/O.

XFS as a whole is separated into three main projects: First, there is the XFS filesystem itself, in the form of a driver. Then a set of tools in a package called xfsprogs is tightly connected with XFS filesystem and contains programs useful or necessary for creation and manipulation of the filesystem. Among the tools included are `mkfs.xfs`, on which this thesis is focused, but also other tools: `xfs_io`, `xfs_growfs`, et cetera. And finally, there are xfstests, which is a test suite containing hundreds of shell scripts used for verifying the behaviour of entire XFS chain (from mkfs to the kernel code). This project is also partially used by other filesystems.

In addition to the xfstests test suite, xfsprogs also uses an automatic static analysis from Coverity [14]. However, to see detailed information and defects there, one must be approved by existing members of the project.

2.1 XFS Architecture overview

When a XFS partition is formated, up to three areas are created on the disk: Data section is always present. An optional real-time section is omitted by default. The log section must always exist, but can be placed on a different device. The data part is then split into multiple de facto independent regions called Allocation Groups, which handle space allocation and allows for higher parallelism, as most operations can be done on each Allocation Group independently on the others. The log section may be internal to one of the Allocation Groups.

[1] At least matured ones. Newer filesystems, like Btrfs, are also 64bit, but have not yet reached stability required in business sector.

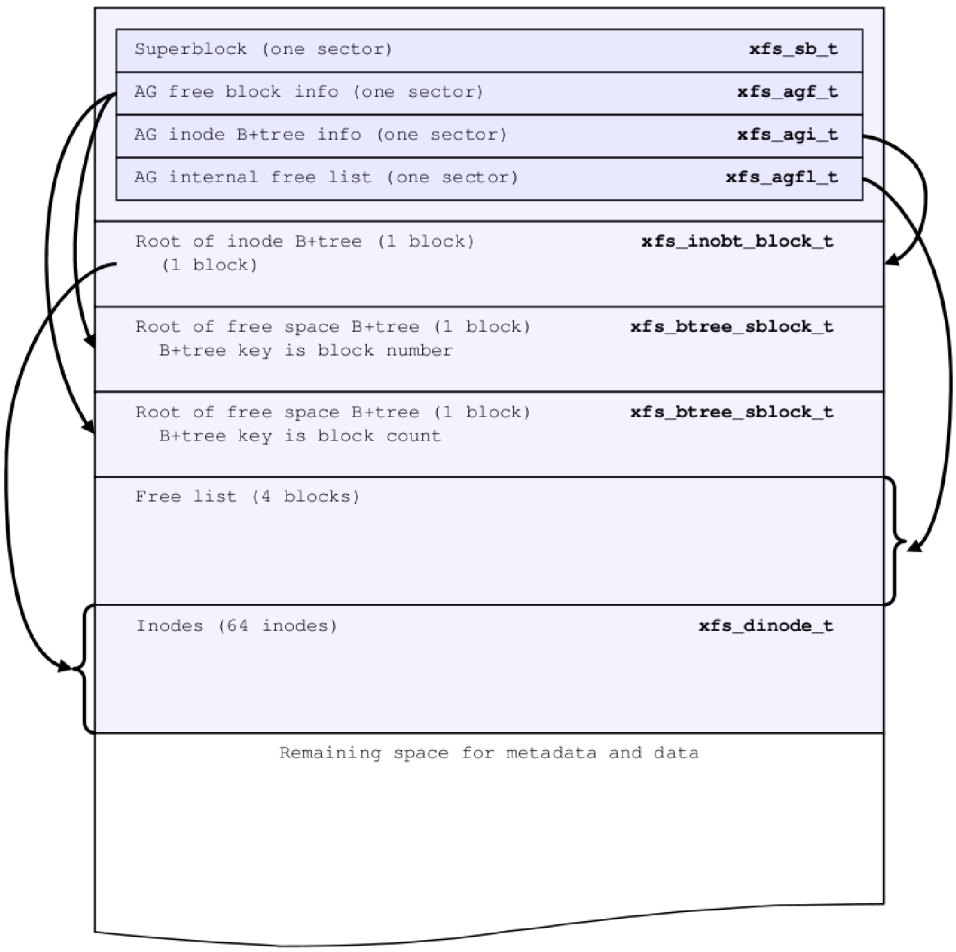

Figure 2.1: Primary AG immediately after mkfs [33, Ch. 3].

As each Allocation Group is a de facto standalone region, each contains a superblock as well as block and inode allocation structures, and the only global information maintained by the first (primary) AG is free space and total inode counts across the whole filesystem as can be seen on Figure 2.1.

The XFS real-time section is dedicated for files with a real-time attribute bit set, and operations with these files should have predictable latencies [32]. The log section is used for metadata journaling, to recover from situations like power failure on the next mount [33, 13].

When created on striped RAID[2], XFS can be informed about the underlying storage geometry and align all allocations and size to the stripe unit to maximize speed.

As a native Unix filesystem, XFS uses *inodes* as a data structure to save information about files and directories. The first of the three parts of an inode (see Figure 2.2), core, contains the basic information, describing what the inode represents. Some example of the data in this field is the id of the user and group owning this inode, size and modification time.

[2]Striped RAID is e.g. RAID 0, where logically sequential data are split into a number of physical blocks and written on multiple disks interleaved. For a RAID 0 on two drives it means that odd blocks are located on one drive and even blocks on the other.

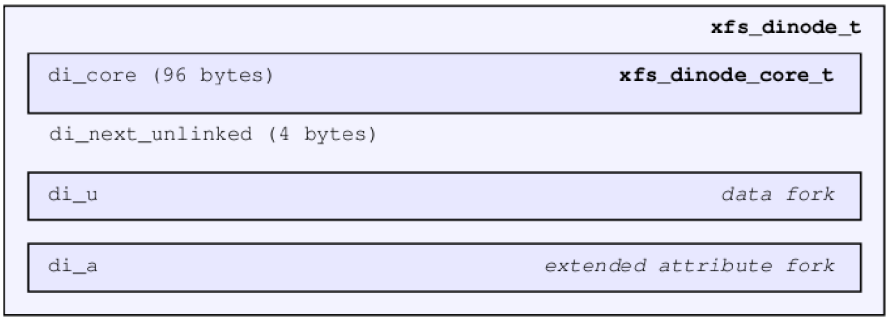

Figure 2.2: On-disk inode [33, Ch. 4].

The second part, `di_u`, or data fork, is for data for any specific type the inode can be; a directory, a symbolic link, a regular file, etc. For a directory, it will contain the entries in the directory. And the third member of each inode, `di_a`, is reserved for extended attributes (abbreviated `xattr`), which are used for example by SELinux.

2.2 mkfs.xfs

This short chapter describes in a greater detail the mkfs.xfs program itself, located in file `mkfs/mkfs_xfs.c`, from user point of view. For information about its implementation, see Section 3.2. This tool creates a new XFS filesystem with given properties. It is, as is usual for core Unix utilities, a non-interactive program which accepts multiple arguments when called (the basic synopsis is shown in Listing 2.1) and prints out the properties of the newly created filesystem if successful, or prints an error and usage help when an error occurs.

Listing 2.1: Synopsis of mkfs.xfs utility [12].

```
mkfs.xfs [ -b block_size ] [ -d data_section_options ] [ -f ]
         [ -i inode_options ] [ -l log_section_options ] [ -n naming_options ]
         [ -p protofile ] [ -q ] [ -r real-time_section_options ]
         [ -s sector_size ] [ -L label ] [ -N ] [ -K ] device
```

An example of such usage is `mkfs.xfs -f /dev/sda1`. This simple example creates a XFS filesystem on device `/dev/sda1` even if a filesystem already existed there – thus the `-f` (as force) flag. For the whole description of mkfs.xfs usage it is better to refer to mkfs.xfs manual page. What is important to note here is that parsing the input arguments and computing inner values based on these inputs makes most of the circa 3,500[3] lines of code.

2.3 xfstests

The project named *xfstests*, or also FSQA, is a framework and a collection of test suites written in Bash. Most of the tests run some filesystem utilities and either validates whether some part of the FS ecosystem behaves correctly, or tries to replicate a specific known issue to prevent regressions.

The tests are grouped into multiple categories according to the tested filesystem:

[3] Before the merge of the last part of my changes. With these changes, mkfs has over 4,000 lines.

`btrfs, cifs, ext4, f2fs, generic, ocfs2, overlay, shared, udf, xfs`

A grouping orthogonal to these categories assigns each test into one or more groups that allows for finer tuning of which tests should be run[4]

These tests do not analyse any program (either its source code, or the compiled binary), but only the results of running these programs, i.e. printed messages, filesystem behaviors, etc. Because xfstests do not employ any formal technique and focus on completely different means of testing, it is not compared with other tools in this work.

[4] An example of the groups: `mkfs, quick, all, dangerous, auto, quota, attr, symlink, ....`

3 Refactoring of mkfs.xfs

The primary goal of the changes described in this work is to rewrite a complex and chaotic code for parsing user input with a table that holds values like minimum/maximum, default values, conflicts and others. Thus, instead of ad-hoc conditions and operations, there will be just one global structure, well documented and easily readable and extendable. This structure should hold also the user-entered values and limit code and variables duplication as much as possible.

During development, we had to repeatedly solve conflicts with changes from other developers that got merged into xfsprogs while we were still working on our changes. That led me to cut the work into multiple parts. That way, others could benefit from changes that were already done and we would not have to maintain so many patches at any given time. There are two main patchsets which are accompanied by several small and enclosed changes that could be easily submitted independently.

The naming convention used in this work is the same as what is internally used in xfsprogs:

Option Can be referred as a section. The highest-level argument of mkfs, starting with a dash. E.g. `-b` or `-d` . Most options have a mandatory argument consisting of suboptions.

Suboption Can be also referred as some section's option. Consists of one or more items in a format `name=value` separated by a comma, but no space. The value can be optional, e.g. when the suboption is a boolean flag.

Look on the example in the Listing 3.1. The command has two options and two suboptions. The options are `-f`, which does not have any argument and serves as a *force* flag, so mkfs does overwrite any existing filesystem on the target device.

The second option is `-d`, which has arguments for setting up non-default values for data section. There are two used suboptions of this option: `file` is only a flag, which tells mkfs that the target device is not a block device, but a regular file (and thus mkfs should not use direct IO, or compute blocksize differently). `size` has its own argument and denotes the size of the data section, 10 GB in this case. Because nothing else is specified, the size of other sections is computed automatically.

Listing 3.1: An example of mkfs.xfs invocation.

```
mkfs.xfs -f -d file,size=10G /foo/bar
```

3.1 Development processes

At first we will briefly describe the development processes and tools used for xfsprogs, which are similar as tools and processes used for Linux Kernel development.

Most of the communication is happening on a mailing list[1] while IRC chat is used for some less important and more day to day issues. The code is hosted in a Git repository, but only selected maintainers have a write access.

[1] Specifically linux-xfs@vger.kernel.org.

Any commit an author wants to get merged into the code must be submitted as a patch to the mailing list. There the patch awaits a review – that is, some other developer must check the changes and append his or her signature to this patch. Once the patch is reviewed and if there are no objections, the maintainer will merge it in a batch with other changes (for xfsprogs, this usually occurs about twice a month).

However, there are many unwritten rules and customs, that are not apparent at first and a new developer finds about them usually only when she or he breaks such a rule.

An example of such an unwritten rule is the exact coding style and the use of a code style checking script `checkpatch.pl` which originated in Kernel community and is part of Linux Kernel source. Such rules have their place, and helps to keep a consistent style throughout xfsprogs, but the fact that they are not documented causes unnecessary issues and delays.

3.2 Initial codebase

Almost all the important code we were changing is located in `mkfs/xfs_mkfs.c` file. The code before the first patchset was merged can be accessed in the project's Git repository as a version 4.6. Git revision hash for this version is 09033e35. In this revision, the parsing of user input works as follows.

In the `main(int argc, char **argv)` function is a loop using a standard `getopt` from `unistd.h` to detect an option like `-d` for data section or `-l` for log section. For options that have arguments, a nested loop uses custom functions to parse specific suboptions and their values.

As an example, here is the beginning of aforementioned loops, as it is in the code, and some issues with this code.

Listing 3.2: Part of option-parsing loop from mkfs.xfs with additional comments.

```
while ((c = getopt(argc, argv, "b:d:i:l:L:m:n:KNp:qr:s:CfV")) != EOF) {
        switch (c) {
        case 'C':
        case 'f':
                force_overwrite = 1;
                break;
        case 'b':
                p = optarg;
                /*
                 * This nested loop will parse the argument of -b, which is
                 * a list of suboptions separated by a comma, but not space.
                 */
                while (*p != '\0') {
                        char    *value;

                        /*
                         * The getsubopt() function removes the first suboption
                         * from the 'p' variable and returns a number
                         * representing the specific suboption, while
                         * saving its value (if any) to 'value'.
                         */
                        switch (getsubopt(&p, (constpp)bopts, &value)) {

                        /*
```

```
        * an example of how one suboption is parsed.
        */
       case B_LOG:
               if (!value || *value == '\0')
                       reqval('b', bopts, B_LOG);
               if (blflag)
                       respec('b', bopts, B_LOG);
               if (bsflag)
                       conflict('b', bopts, B_SIZE,
                                        B_LOG);
               blocklog = atoi(value);
               if (blocklog <= 0)
                       illegal(value, "b log");
               blocksize = 1 << blocklog;
               blflag = 1;
               break;
```

While the `-f` option is simple, in case of `-b log=XX`[2] we can see how the parsing can get complex. The code tests if the value is not empty and if it is, it raises an error. Then it tests whether this specific option was already used, because repeated specification of the same option is prohibited[3] mkfs.xfs allows to specify the block size both as an explicit size in bytes or in a logarithmic scale, but only one of these options can be used at a time. So the code must also check if the other variant was used and compute both values.

Some options have a test directly within this assignment for conflicting options, others simply set up the value and test the conflicts later, after the `getopt` loop. Other options use both of these methods, depending on what the author of each change considered a better solution, how it was required to achieve a given functionality or how this part of code was changed over time.

We can see that most of the work happening in this part of code is rather generic – all options are checked for respecification, whether a required value is present, or possibly whether the value is in a certain range of valid values.

However, none of these universal tasks is automated – every single option must reimplement the same tests. Some options have almost all logic in it's `case` statement, where it is at leas in one place. But options with more complex dependencies and conflicts have only part of their logic there and the rest of it is in a section of code following the main loop, in ad-hoc tests and computations.

To further complicate situation, some parts of `mkfs.xfs` are more than 20 years old and the coding style and the general approach to specific things changed since then, but the old code did not. If such old code needs a change, there is always a risk that the editing programmer assumes a different behaviour similar to the one that newer options have, but that assumption is incorrect.

Also the six variables specific for `-b log` are not explicitly tied together and because almost

[2] Here and in other places, the block option is used as an example, because this option has only two suboptions, so it can be shown in full if needed.

[3] Respecification is forbidden for this reason: consider, what happens if a user uses this combination of options: `-b size=4k -d size=1000b -b size=512`, where the `b` suffix in a number denotes a block. At first, blocksize is set to one value, a size of data section is computed based on this value and then the blocksize is changed. Thus, any following use of blocksize will have a value different than what was used for the first computation. This could be countered by computing all values after all options are parsed, yet it would still be ambiguous and might behave differently than the user expected. Forbidding it is a cleaner and safer approach.

every option has a similar mix of multiple variables, it is difficult to keep all the important ones in a mental image of the code and always use the correct one. Many of these variables are unnecessary or reduntant, so in some cases, the values are copied from one to another and if a change is put into a wrong place, a specific condition may cause the changed value to be overwritten later on with the old one, etc.

It is easy to see what could go bad in this: When changing one option, it was possible to forget to change the other one. If a test was done after `getopt`, any other option that would modify a value[4], which is used for a computation in another option, could overwrite the value and cause a conflict without a notice.

Any new option required a careful reading through the existing code and possibly the placement new checks in multiple places. Thus it was difficult to know when any value is checked and safe for use in further computations.

HERE

The consequence of these issues is that `mkfs.xfs` did a bad job of validating user input from the command line. Even if an issue was detected and the specific error fixed, the minimal code reuse meant that other options could still be susceptible to the same or similar issue.

3.3 First patchset

As is shown Section 3.2, the situation was not ideal and the state of the code led to many known issues. David Chinner, then maintainer of XFS, presented a set of patches as an RFC[5] in November 2013 [7] in an attempt to raise a discussion. However, nobody joined him and David Chinner himself did not continue in pressing this matter for few years. Here is an excerpt from his RFC:

> This is still a work in progress, but is complete enough to get feedback on the general structure. The problem being solved here is that mkfs does a terrible job of input validation from the command line, has huge amounts of repeated code in the sub options processing loops and has many, many unnecessary variable for tracking simply things like whether a parameter was specified.
>
> This patchset introduces a parameter table structure that is used to define the parameters and their constraints. Things like minimum and maximum valid values, default values, conflicting options, etc are all contained within the table, so all the „policy“ is found in a single place.
>
> ...
>
> The flow on effect of this is that we can remove the many, many individual variables and start passing the option structures to functions rather than avoiding using functions because passing so many variables is messy and nasty. IOWs, it lays the groundwork for factoring xfs_mkfs.c into something more than a bunch of spagetti...

[4] See `-i size=x` and `-i log=y`. In the code, both options modify the same variables and differs only in accepted values. `size` expects a number of bytes while `log` expects a base two logarithm value.

[5] Request for comment - signalling, that the presented patches are not meant to be merged, but the author wants to hear other people's thoughts about these changes.

When we joined the XFS team and began with the refactoring in 2015 [36], we picked up this patchset and brought it up to date with the codebase that in some parts changed substantially in the preceeding two years. Once the patches were applicable for the current code, we began fixing functional issues and adding further changes.

This lasted until May 2016, when this patchset was merged into the upstream repository [37, 9]. These changes implemented the core parts from the desired state. The implementation of the basic table made the `mkfs_xfs.c` file more readable, even if it was possible to remove only basic checks. It also brought a much more strict input validation, so few of the existing tests in xfstests had to be updated and a new test was created, with the goal to watch only for input validation, whether `mkfs.xfs` correctly accepts or refuses any given combination of options and values.

Size and commits of this patchset are described in Listing 3.3. It is 19 patches that are grouped by the initial author, in this case David Chinner and Jan Ťulák.

Listing 3.3: Git statistics for the first patchset [37]. Note: Git attributes changes only to the first author of each commit.

```
Dave Chinner (15):
  xfsprogs: use common code for multi-disk detection
  mkfs: sanitise ftype parameter values.
  mkfs: Sanitise the superblock feature macros
  mkfs: validate all input values
  mkfs: factor boolean option parsing
  mkfs: validate logarithmic parameters sanely
  mkfs: structify input parameter passing
  mkfs: getbool is redundant
  mkfs: use getnum_checked for all ranged parameters
  mkfs: add respecification detection to generic parsing
  mkfs: table based parsing for converted parameters
  mkfs: merge getnum
  mkfs: encode conflicts into parsing table
  mkfs: add string options to generic parsing
  mkfs: don't treat files as though they are block devices

Jan Tulak (4):
  mkfs: move spinodes crc check
  mkfs: unit conversions are case insensitive
  mkfs: add optional 'reason' for illegal_option
  mkfs: conflicting values with disabled crc should fail

 include/Makefile         |    5 +-
 include/xfs_multidisk.h  |   73 ++
 libxfs/init.c            |    6 +
 libxfs/linux.c           |   11 +-
 man/man8/mkfs.xfs.8      |   45 +-
 mkfs/Makefile            |    2 +-
 mkfs/maxtrres.c          |    2 +-
 mkfs/proto.c             |   58 +-
 mkfs/xfs_mkfs.c          | 1983 ++++++++++++++++++++++++++++++++-----------------
 mkfs/xfs_mkfs.h          |   89 ---
 repair/xfs_repair.c      |   44 +-
 11 files changed, 1417 insertions(+), 901 deletions(-)
 create mode 100644 include/xfs_multidisk.h
 delete mode 100644 mkfs/xfs_mkfs.h
```

3.3.1 Timeline and progress

- *November 2013* – Dave Chinner submits his RFC.
- *May 2015* – We are beginning the work on this patchset.
- *June 2015* – The first published version. It contains only minor changes except updating and fixing the most serious errors. We are getting the first feedback.
- *March 2016* – Another version submitted, this time with more custom changes.
- *April 2016* – Further big changes. Some patches are reverted to older versions, while a new patch is added.
- *May 2016* – Changes are made only in specific patches, no new version of the whole set is submitted.
- *June 2016* – The patchset is accepted and merged into the repository.

3.3.2 Description of important changes

The key part of this patchset is the creation of `opt_params` table, shown on Listing 3.4. It is a structure that is holds all the important values for a specific option in one place, easily accessible and consistent across the whole file.

Listing 3.4: Definition of the table.

```
struct opt_params {
        const char      name;
        const char      *subopts[MAX_SUBOPTS];

        struct subopt_param {
                int             index;
                bool            seen;
                bool            str_seen;
                bool            convert;
                bool            is_power_2;
                int             conflicts[MAX_CONFLICTS];
                long long       minval;
                long long       maxval;
                long long       defaultval;
        }               subopt_params[MAX_SUBOPTS];
};
```

The meaning of the specific fields is this:

name *MANDATORY* Name is a single char, e.g., for '-d file', name is 'd'.

subopts *MANDATORY* Subopts is a list of strings naming suboptions. In the example above, it would contain „file“. The last entry of this list must be NULL.

subopt_params *MANDATORY* This is a list of structs tied with subopts. For each entry in subopts, a corresponding entry must be defined.

The `subopt_param` has the following members. The displayed descriptions are part of the code:

index *MANDATORY* This number, starting from zero, denotes which item in `subopt_params` it is. The index must be the same as is the order in subopts list, so we can access the right item both in `subopt_params` and subopts.

seen *INTERNAL* Do not set this flag when defining a `subopt`. It is used to remember that this `subopt` was already seen, for example for conflicts detection.

str_seen *INTERNAL* Do not set. It is used internally for respecification, when some options must be parsed twice - at first as a string, then later as a number.

convert *OPTIONAL* A flag signalling whether the user-given value can use suffixes. If you want to allow the use of user-friendly values like 13k, 42G, set it to true.

is_power_2 *OPTIONAL* An optional flag for `subopts` where the given value must be a power of two.

conflicts *MANDATORY* If your subopt is in a conflict with some other option, specify it. Accepts the .index values of the conflicting subopts and the last member of this list must be `LAST_CONFLICT`.

minval, maxval *OPTIONAL* These options are used for automatic range checking and they have to be always used together in a pair. If you do not want to limit the max value, use something like `UINT_MAX`. If no value is given, then you must either supply your own validation, or refuse any value in the '`case X_SOMETHING`' block. If you forget to define the min and max value, but call a standard function for validating user's value, it will cause an error message notifying you about this issue.

(Said in another way, you can not have `minval` and `maxval` both equal to zero. But if one value is different: `minval=0` and `maxval=1`, then it is OK.)

defaultval *MANDATORY* The value used if user specifies the subopt, but no value. If the subopt accepts some values (`-d file=[1|0]`), then this sets what is used with simple specifying the subopt (-d file). A special `SUBOPT_NEEDS_VAL` can be used to require a user-given value in any case.

It was later revealed that the name of this field is confusing and can be mistaken for a default value in the sense of „user did not specify anything.“ As this led to an incorrect configuration for an option (albeit semi-internal one, used only by developers for testing purposes), we proposed a name change to `flagval`. This change is a part of the next set.

`opt_params` is instantiated for every option category, e.g. Listing 3.5 shows instantiation for `-b`.

Listing 3.5: Instantiation of the table for block options.

```
struct opt_params bopts = {
        .name = 'b',
        .subopts = {
#define B_LOG           0
                "log",
#define B_SIZE          1
                "size",
                NULL
        },
        .subopt_params = {
                { .index = B_LOG,
                  .conflicts = { B_SIZE,
                                 LAST_CONFLICT },
                  .minval = XFS_MIN_BLOCKSIZE_LOG,
```

```
                .maxval = XFS_MAX_BLOCKSIZE_LOG,
                .defaultval = SUBOPT_NEEDS_VAL,
            },
            { .index = B_SIZE,
              .convert = true,
              .is_power_2 = true,
              .conflicts = { B_LOG,
                             LAST_CONFLICT },
              .minval = XFS_MIN_BLOCKSIZE,
              .maxval = XFS_MAX_BLOCKSIZE,
              .defaultval = SUBOPT_NEEDS_VAL,
            },
        },
};
```

With this structure, many functions had to be completely rewritten or added, but the result was that the option parsing loop could be greatly simplified. For comparison, here is the nested loop from 3.2 code example after this patchset was applied. You can see that the section for `B_LOG` is now much cleaner (no branching, only few assignments) and the generic logic was moved away into a function shared with other options as can be seen in Listing 3.6.

Listing 3.6: Part of option-parsing loop from mkfs.xfs after the first patch set.

```
case 'b':
        p = optarg;
        while (*p != '\0') {
                char    **subopts = (char **)bopts.subopts;
                char    *value;

                switch (getsubopt(&p, (constpp)subopts,
                                  &value)) {
                case B_LOG:
                        blocklog = getnum(value, &bopts, B_LOG);
                        blocksize = 1 << blocklog;
                        blflag = 1;
                        break;
```

Another important issue fixed in this set was the behaviour difference when mkfs.xfs is run to create a filesystem on a block device vs. in a file on another filesystem.

The issue was that if the target was a file, but `-d file` is not specified, mkfs behaved as if the target is a block device. That meant, however, that if the underlying block device had e.g. sector size 512B, on which a filesystem with sector size 4kB existed, then, when creating a new filesystem in a file, mkfs used the (incorrect) 512B size of the physical device and ignored the value used in the intermediate layers.

This was mitigated by automatic detection of whether the target is a regular file or a block device, and by changing the flow of the program on various places where the difference between file and device was important.

However, there were still many issues that were not addressed. The conflicting options were only enumerated, without any additional information, and thus the field was usable only for always conflicting options, like `-b log|size` – it did not help with conditional conflicts. For example, checksums for metadata, enabled with `-m crc`, works only on newer version of metadata format: `-m crc -i attr=1` is conflicting, but `-i attr=2` is not. Such tests still had to be done as before. Also, it was possible to specify conflicts only between suboptions of a single option.

3.4 Second patchset

Once this change was merged and provided a stable point so we did not have to keep so much code in our own local repository up to date with upstream, we began to work on the second set of changes. We submitted an RFC of these changes in December 2016 [38]. Such a big and complex change is something that most of the developers postpone, so it is usually reviewed only by the maintainer when nobody else starts it. In this case, however, the maintainer changed in late December – Eric Sandeen took this position instead of David Chinner.

Size of this patchset in the first RFC is 22 patches and the patches can be seen in Listing 3.7.

Listing 3.7: Git statistics for the second patchset [38].

```
Jan Tulak (22):
  mkfs: remove intermediate getstr followed by getnum
  mkfs: merge tables for opts parsing into one table
  mkfs: extend opt_params with a value field
  mkfs: change conflicts array into a table capable of cross-option
    addressing
  mkfs: add a check for conflicting values
  mkfs: add cross-section conflict checks
  mkfs: Move opts related #define to one place
  mkfs: move conflicts into the table
  mkfs: change conflict checks to utilize the new conflict structure
  mkfs: change when to mark an option as seen
  mkfs: add test_default_value into conflict struct
  mkfs: expand conflicts declarations to named declaration
  mkfs: remove zeroed items from conflicts declaration
  mkfs: rename defaultval to flagval in opts
  mkfs: replace SUBOPT_NEEDS_VAL for a flag
  mkfs: Change all value fields in opt structures into unions
  mkfs: use old variables as pointers to the new opts struct values
  mkfs: prevent sector/blocksize to be specified as a number of blocks
  mkfs: subopt flags should be saved as bool
  mkfs: move uuid empty string test to getstr()
  mkfs: remove duplicit checks
  mkfs: prevent multiple specifications of a single option

 mkfs/xfs_mkfs.c | 2952 +++++++++++++++++++++++++++++++++--------------------
 1 file changed, 1864 insertions(+), 1088 deletions(-)
```

Together, these two issues caused that despite our urging, there was not much reaction until March. In March we submitted another version, this time intentionally not as an RFC. We also mentioned to few people that this is part of our thesis.

The review of the second set revealed many disputable points and it become certain that these patches will need further changes. The second part of our changes is focused mostly on conflict detection and allows for almost all checks to be removed from the code as ad-hoc solutions, as the new structures and functions take care of them automatically. Any programmer making a change only must correctly specify values in a `struct opt_params`, write in a list of conflicting options, and the validation of the new option is guaranteed to work correctly and seamlessly.

To make the process faster, we decided to split this patchset into multiple smaller ones, which can get fixed, reviewed and merged faster. The first group focused on extending the options table not only for encoding validity range and basic conflicts, but also for user input.

Most notable changes are functions `parse|get|set_conf_val` – a set of functions to ma-

nipulate the user input values. This is a key difference from the RFC. There the values were manipulated as pointers to the table, but other developers raised objections. Most notably Dave Chinner, who rebutted our worries about the use of setters and getters in a reply to our e-mail where we suggested them as another option [41, 6]

Compare code examples 3.8 and 3.9. The first example with aliases keeps a lot of seemingly unconnected variables in the code where the programmer does not know where exactly the pointer leads to. And even if he finds the first assignment, it is possible to mistakenly override the target address. In the second example, with setters and getters, it is apparent at first glance, where the value is to be written or read. And it is impossible to mistakenly alter the target address. The disadvantage of using setters is that it is no longer possible to do in-situ increments/decrements (e.g. `i++;`), however this is only a minor issue.

Listing 3.8: Pointer aliases in RFC of the second set.

```
long long *agcount = &opts[OPT_D].subopt_params[D_AGCOUNT].value;

// ... lines skipped
*agcount = foo(bar);

// ... lines skipped
if (*agcount < SOME_CONSTANT)
        // do something
```

Listing 3.9: Setters and getters in later version of the second set.

```
set_conf_val(OPT_D, D_AGCOUNT, foo(bar));

// ... lines skipped
if (get_conf_val(OPT_D, D_AGCOUNT) < SOME_CONSTANT)
        // do something
```

Furthermore, this approach allows for a verification of all values saved into the table for the whole run of the program. However, after attempting to implement this feature, we found out this is not possible to add at this moment. While we know valid bounds for user input values, some of these values are then recomputed and can get out of these bounds, while still being valid. An example of this is `L_SUNIT`[6], which is specified as a number of 512-byte blocks. However, it is later multiplied by the 512, at which moment it can get out of the valid bounds specified for input.

The proposed but not yet implemented solution is to utilize the existing infrastructure and create a new pseudo option for the table, which would not be visible to the end user, but would hold all the internal variables for which the bound range (or any other condition, like being power of 2) can be applied. This topic was briefly discussed in replies to one of the patches in this set [28], because one of the other developers, Luis R. Rodriguez, has a work in progress that requires such infrastructure to be implemented.

Because even this smaller set was not fully accepted by the end of April, and addressing the issues other developers raised required too much time, we further focused on the next part of this work and did not use the second patchset in formal analysis.

Size of the smaller set can be seen in Listing 3.10.

[6] `L_SUNIT` specifies the alignment of log writes.

Listing 3.10: Git statistics for the first part of the second set after its breaking into smaller parts [40].

```
Jan Tulak (12):
  mkfs: Save raw user input field to the opts struct
  mkfs: rename defaultval to flagval in opts
  mkfs: remove intermediate getstr followed by getnum
  mkfs: merge tables for opts parsing into one table
  mkfs: extend opt_params with a value field
  mkfs: create get/set functions for opts table
  mkfs: save user input values into opts
  mkfs: replace variables with opts table: -b,d,s options
  mkfs: replace variables with opts table: -i options
  mkfs: replace variables with opts table: -l options
  mkfs: replace variables with opts table: -n options
  mkfs: replace variables with opts table: -r options

 mkfs/xfs_mkfs.c | 2457 ++++++++++++++++++++++++++++++++++----------------------
 1 file changed, 1420 insertions(+), 1037 deletions(-)
```

3.4.1 Timeline and progress

- *June 2016* – The first patchset is accepted and merged into the repository, beginning of the work on the second set.
- *August 2016* – First draft of changes of the second set [39].
- *December 2016* – RFC of the full second set. This gained just a little attention.
- *March 2017* – Second set without the RFC. This version attracted enough attention to be useful, and provided valuable feedback.
- *March 2017* – Vault conference in Boston. We met some other developers personally and debated some of the changes. This helped to raise some attention towards our patches.
- *April 2017* – resubmitting only part of the second set with requested changes and setters/getters as an addition. This generated a lot of feedback.

3.5 Summary

Part of the changes was successfully merged into the project in time. However, some other patches gained the necessary attention too late and all the found issues could not be fixed or changed before the deadline for this work. These changes will get merged eventually.

The difficulties were analysed in an attempt to avoid these delays in the future. Our hypothesis is that the set as a whole was too big and complex, an effect which was multiplied by unintentionally not adhering to unwritten standards. A proposed process change for further iterations is to send few smaller patches more often and wait with other changes depending on those submitted until they are accepted.

The higher activity on the last patchset, which was just a subset of the second big set, seems to confirm this hypothesis, however, more iterations are needed.

4 Formal Analysis and Verification

As the role of computers in human society is growing in ever faster pace, the consequences of any error are growing too. Consider, for example, the speed with which smartphones seized our pockets. They certainly bring many benefits, but as we become dependent on the smartphones, any malfunction or error in them can affect our life. From not having access to an important information to a direct danger, such as in the case of motorists stranded by their mobile navigation in the middle of the wilderness [24].

Or consider the recent advances in the area of autonomous vehicles. Where an error in smartphones can only deprive us of information, an error in a self-driving car can cause it to swerve into a wall with dire consequences for the passengers.

Common testing techniques, despite advances in this field, are still mostly reactive and can detect only known errors, for which a test was written, and can not provide a guarantee of correctness. That is, they can tell that „none of these specified errors happened," but can not tell whether the system is really free of errors with respect to a specification.

Formal methods, with roots in mathematical areas like theorem proving, on the other side frequently have the power to verify correctness. But unlike the common testing techniques, and despite an interest in the industry, they are not yet widely used. A notable exception to this is *static analysis*, which, in some of its weaker forms, is becoming a part of integrated development environments (IDE) like Xcode or Eclipse [4].

One reason for the small adoption of formal methods is their complexity. They either require advanced user knowledge, like human-driven deductions in *theorem proving*, require excessive modelling of the environment for the system like *model checking*, or are simply unable to cope with the size of the code and the size of its state space.

By the term *formal analysis*, we describe methods for answering questions other than whether the tested system is free of errors with respect to some specification. That is, it includes questions like whether the program is guaranteed to always terminate if a buffer size is bound and so on.

Formal verification then denotes methods capable of proving that the given system is error free with respect to a correctness specification. *Completeness* of a method is a property guaranteeing that it will not raise a false alarm, while if a method that is *sound* terminates and tells that there are no errors, the system is indeed correct.

In the following parts of this work, we will first discuss some of the formal techniques (the rest of this chapter) and then also their usefulness on a real, production codebase in the chapter Used Techniques and procedures. We will look not only at their result, but also at the cost of using them, both in time and expertise necessary for their correct use.

4.1 Static Analysis

A rather broad, but commonly used definition of *static analysis* is that it is an analysis that collects some information about the behaviour of a system without actually executing it under

its original semantics [17, Chap. 2.2]. It can manage very large systems in a reasonable time and is highly automated, but can suffer false positives and is generally weaker than other methods (it is difficult to express some problems for *static analysis*). This category includes methods:

Abstract interpretation, in which an abstract overrepresentation of the statements of the program is evaluated in an abstract machine for all possible inputs at once and we exchange completeness for speed or even the possibility to analyse the system.

Data flow analysis tracks how a given set of properties propagates through the program without directly executing it.

Error patterns then denote the most common method used in various lightweight implementations already present in various IDEs, in Lint and Cppcheck software, and others. As the name itself explains, these methods attempt to detect commonly occurring patterns that programmers make, but which lead to an error. A simple example may be a missing `break` statement, or missing boundary checks before accessing an array.

Let us now look more in detail at each of these methods and at their implementations, but bear in mind that in many cases, the tools we will see are not clearly distinguished and can be placed into more than one category. Thus, the tools are categorised according to the most important principles in their implementation.

4.1.1 Error patterns

Tools using it: Lint [17] (and it's followers), cppcheck[1]

Error patterns detection is a rather wide array of different techniques and methods with a common goal: To detect more or less frequent types of errors. The great advantage of this class of methods is that they usually does not require any deeper knowledge, are fully automated and can be very fast. Their disadvantages are that they are limited to a very specific kind of errors[2] and suffer from false positives.

An example of the approaches used in this class of static analysis is a detection of matching pairs of functions. For example, any `open()` call should be later followed with one `close()` in every possible path. Or a state machine can be used to detect missing delimiter between statements or a missing `break` in a `switch` [18].

4.1.1.1 Lint

Lint was originally created in the 70's for the early C language [17, Chap. 2.2] and since then, multiple new tools for various languages were inspired by it: splint, cpplint, JSLint, Pylint, etc. It searches for patterns that are likely to be bugs, to be non-portable, or to be wasteful [26]. Many of the follower implementations are open-source.

4.1.1.2 CppCheck

[1] We are not sure about this and did not found it cited anywhere, but from its code, it looks like they search for error patterns.

[2] Basically, every error pattern needs its own filter and only some kinds of error patterns are generic enough to be shipped within the tool. Consider a support for usage of a library. It makes sense to watch for patterns in usage of `stdlib.h`, but how it should know patterns in a custom library? And even if the specific library is publicly available, including everything is impossible. Many tools allow for providing of custom definitions of these patterns, but from a personal experience, they are rarely used.

An open-source tool for C/C++ languages. It can only a very simple *control flow analysis*, where it expects that all statements can be always either true or false and thus all statements should be always reachable [19].

4.1.2 Data flow analysis

Tools using it: Coverity [17], CodeSonar [17], TruePath [17], FindBugs [17]

These methods, in industrial tools frequently combined with *error patterns*, track how so-called *data flow facts* propagate within a *control flow graph* between its nodes. These nodes represent *basic blocks* in the original code, each block having only one entry point and one exit point[3], which simplifies the analysis.

The analysis can be either *forward*, where the state at the exit of one basic block is used as the input of the following block and which starts at the beginning of the program. Or *backwards analysis*, where the algorithm starts in an end state and attempts to find a path to a start state. This second approach can be useful to determine whether a particular end configuration is reachable.

4.1.2.1 Coverity

A proprietary tool (and company) providing a free service for open-source projects (`http://scan.coverity.com`). Uses restricted formal verification, but getting to specific details is hard or impossible. Supports languages Java, C/C++, C#, JavaScript, Ruby, and Python.

4.1.2.2 FindBugs

FindBugs is an open-source code analyser for Java language with plugins for many Java IDEs.

4.1.3 Abstract interpretation

Tools using it: Astrée [23, 17], PolySpace [17]

Abstract interpretation shares a blurry border with *model checking* and it may be difficult to determine where a specific method is. The basic way in which abstract interpretation works is to run a symbolic execution of the program. On every statement, it transforms specific values into an abstract context and *widen* (over-approximate) or *narrow* to refine the result after widening.

The widening and narrowing is usually implemented by using a pair of monotone functions: *Abstraction* denoted α and *concretization* denoted γ forms a Galois connection.

These methods can be sound, but not every abstract interpretation is, as they can range from a simple syntax analyser to full model checking.

4.1.3.1 Astrée

A tool for analysing applications written in C language. It is a proprietary tool used for safety-critical applications, for example by Airbus [17]. It provides a sound static analysis. False positives are considered a reasonable price for the soundness [23].

[3] One entry point means that in no case can any instruction inside of the block other than the first one be a target of a jump. One exit point means that if there is a branching, only the last instruction of a block can cause it and jump to multiple different target instructions.

4.1.3.2 PolySpace

A proprietary tool for C, C++, and Ada languages.

4.2 Model Checking

Tools using it: RuleBase [17], Incisive Verifier [17], Magellan [17], JasperGold Formal Property Verification App [17], Questa Formal Verification [17],CPAchecker [17], Wolverine [17], CBMC [17], LLBMC [17]

Model checking is an algorithmic means of checking whether the given system is correct with regards to any given property through systematic exploring of the state space of this system. The properties are usually specified in some temporal logic like LTL, CTL, CTL* or μ-calculus.

The advantages of *model checking* are that it can be fully automated, is rather universal, does not require a deep knowledge for usage and if it finds an error, it can generate a path leading to the case, which is useful for repairs.

However, it also has two big disadvantages. It requires a model of the environment for the system and suffers *state space explosion*. The number of reachable space grows exponentially – consider a 32bit variable, which has 2^{32} possible values, which equals states. That means that n of such variables equals $2^{32\times n}$ states. The result is that any attempt in a practical use of *model checking* has to cope with this state space explosion.

The methods used for this include *symmetry reduction* in cases where it is not important which specific entity (if there are more entities of the same type in a specific state). See Figure 4.1 for an example of symmetry in the well known dining philosophers problem.

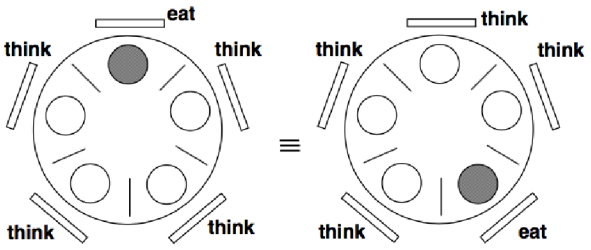

Figure 4.1: Symmetries and the dining philosophers [17].

Other solutions can be to use only one of many possible paths for the ordering of concurrent actions that are independent of each other and compress the size of states by using pointers to the previous state for values that did not change. Or the tool can evaluate the properties at the same time when a new state is generated, and stop immediately once it is clear that this prefix cannot be accepted by the automata denoting correctness specification.

4.2.0.1 CPAChecker

An open-source tool and a framework for an analysis of programs in C language. It is based on the idea of *configurable program analysis* [5], which uses user configuration to perform a reachability analysis.

4.3 Theorem Proving

Tools using it: VCC [17], ESC/Java2 [17], VS3 [17]

Theorem proving is similar to mathematical deduction, where we get a proof from an initial set of axioms. It also shares advantages and disadvantages with its purely mathematical counterpart. On the one side, it is really universal, but on the other side, it can not provide a *counterexample* (a path to an error), but just says yes/no, and is only semiautomatic. The tools can correctly apply inference rules, but their choice is up to the user. Thus, an insufficiently skilled user may not be able to prove that the system is correct even if there are no errors in the system.

5 Used Techniques and procedures

In this chapter, we discuss which techniques and models of formal analysis and verification are useful for the code of `mkfs.xfs`. Let us at first define important constraints that are limiting or directing our choice.

We are analysing a single-threaded application. This greatly reduces the state space and means that we also can use methods that do not allow for concurrency. On the other side, given that the program accepts user input, some variables have an infinite number of potential values and any method based on state space checks must cope with this fact.

A comparison that could be interesting for further research is to experiment with tools that use neural networks and deep learning as an integral part of their algorithms if such tools exist. Examples of the use of those technologies are a heuristic to drive the selection of inference rules in theorem proving, or spotting error patterns [1] However, these approaches are even more complex and experimental than the traditional approaches and would probably deserve a standalone work on their own.

List of tools that were used in this work (in no particular order): Coverity, CppCheck and the analysis in GCC and Clang compilers.

5.1 Testing Environment

The tools were run in a Docker container[2] based on Fedora Linux 25. The use of containers ensures a clean and identical environment for every tool and every run. Image with the compilers and CppCheck are published in Docker Hub and all the recipes to build and use them are provided with this work. Image with Coverity could not be published because it contains confidential information[3].

Every image has its own starting script for easier manipulation – `run.sh`. This script creates a container from the image and mounts the directory with xfsprogs (or any other directory which is passed to it). Then, if necessary, it can pass few options to the script started in the container.

`run-test.sh` is the script started by Docker after creating the container. This script copies xfsprogs from the mounted directory to another one, so it does not change the original repository in any way. In this copied directory the script then starts whatever tool it is prepared for.

Users can, if they wish to do so, enter an interactive shell in the container instead of starting the tool. Also, it is possible to skip the copying or to run `make clean`. For an automated run described in Section 5.2, neither of this is necessary, but these options are useful for manual experiments.

[1] Some attempts in modelling a code are hinted in *On the Naturalness of Software* by Abram Hindle: http://dl.acm.org/citation.cfm?id=2902362.

[2] Simply stated, a container is an image that has been started, similar to the difference between a running virtual machine and its on-disk virtual HDD image. Unlike virtualization, containers are only processes isolated from the rest of the system using kernel capabilities, like *cgroups* and *chroot*. Docker is a specific implementation [15] of containers.

[3] Jan Ťulák had an access to Red Hat Coverity license server as a Red Hat employee. However, the server information and some tools Red Hat provided with Coverity are considered confidential.

Coverity, GCC and Clang are run using `csbuild`, a tool to plug static analyzers into the build process [2]. Because csbuild attempts to use all supported analyzers it finds, the images for each tool we are testing are modified to contain only the single specific tool we need, but no other.

5.1.1 CppCheck

Docker image: `jtulak/cppcheck` [35]

CppCheck is also used in *Codacy*, an automated code review application with Github integration. Results from Codacy are included for comparison.

Because CppCheck does not need preprocessed code, it was reasonable to use it for every commit in our changes.

When running this tool, default configuration was used, and all types of messages were enabled. No custom rules were used and the invocation of CppCheck on whole `xfsprogs/mkfs/` directory was:

```
cppcheck --enable=all mkfs/
```

5.1.2 Coverity

Coverity was used both manually in a Docker container, and automatically using the public Coverity service for open source projects which is part of the standard xfsprogs development process, to compare the results between those two instances.

Furthermore, in the local analysis, four levels of analysis aggressiveness were tested: low (default settings), medium, high and custom (all). The different results are compared in Section 6.4, where were look at a summary of what these levels enabled. With a higher level, Coverity makes more aggressive assumptions during the analysis, which means more defects being reported and longer analysis time [34].

Coverity manual claims false positive rate for all checkers that are not parse warnings increases approximately by 50% for medium and by 70% for high. It does not have an effect on parse warnings checkers.

All levels have enabled everything that a previous, lower level has, plus some additional checks. The medium level uses low level plus enables some other checks, including parse warnings, infinite loops, some resource leaks checks, etc. The high level then adds e.g. integer overflow detections and more checks for infinite loops. A special, level enabled by `--all` flag for `cov-analysis`, then enables almost all checkers, with only a few minor exceptions. For the full (rather long) list of what checkers are enabled on every level, consult the `cov-analysis` manual [34].

The Figure 5.1 describes the steps happening in Coverity container. Where other tools are only a single step, taking source code on one side and producing an output on the other side, Coverity has three standalone steps. First, it builds the source code using its own parser and produces an analyzable data.

The second step, `cov-analyze` application, does the analysis itself. Here it is possible to set up the aggressiveness level and where the license is required. The last step then takes the raw output from the analyser and converts it into one of the selected formats. The scripts in the container generate all three variants. Due to the size of the HTML output, it is not included in the digital attachment to this work.

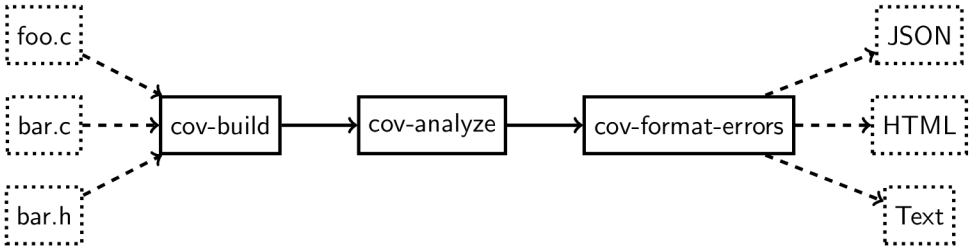

Figure 5.1: Process steps for coverity analysis.

5.1.3 GCC

The *GNU project C and C++ compiler* is used for compiling xfsprogs and it has some static analysis capabilities because it must understand the code to compile it. The only difference in its use from standard configuration is to use the most strict reporting.

5.1.4 Clang

Clang is another C/C++ compiler. It was founded by Apple and LLVM communities in 2007 as an alternative to GCC, which did not work well for Apple's needs. It is designed to be highly compatible with GCC for C-based languages (C, C++, ObjC), but does not have the desire to replace it completely [25].

It is not used by xfsprogs but can compile the code as well, so we can compare it with GCC and other analysis tools. Even if it has its analyzing part as a standalone application [10], the easiest way to use it and to make it the most comparable with GCC was to rename clang binary to GCC and let xfsprogs behave as if it was GCC, rather than modify Autotools configuration. According to Clang's `scan-build` description, the tool replaces certain environmental variables to achieve the same result. However, from the nature of tool-specific containers used for the tests, binary replacement avoids uncertainty about whether the environmental variables were correctly taken into the account in this specific Autotools configuration without the usual consequences of manipulating with system files without the knowledge of a package manager.

5.2 Results Processing

The outputs of these tools have different syntax and verbosity, but we had to find a way, how to compare them, both between the tools and across revisions, despite some of the tools finding a lot of issues. A set of scripts to help both with automating the tests and with analysing was created. The Figure 5.2 shows how these scripts are connected with what happens in the containers.

First, there is tool `parse.py`, which can automatically run all the tools across specified revisions. It takes care of changing the revisions, starting every docker container again and finally, it organises the outputs in a logical way: in a specified directory, it creates a subdirectory for every revision (using the revision's short hash as the directory name) and each such directory then contains log files with outputs from each tool.

The output files are not modified in the first step, but to simplify their parsing, it is useful to preprocess some of those files (namely from GCC and / Clang) with script `format-outputs.sh`,

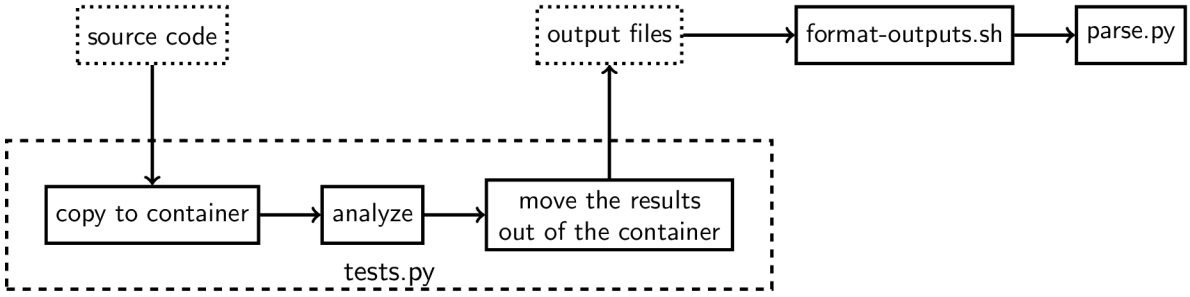

Figure 5.2: Process steps for testing.

to remove color formatting escape sequences and unnecessary compiler outputs. Such data may be useful for some further analysis, but for the next step, it would only make the parsing more complex.

In the last step, script `parse.py`, when supplied with the output directory, translates the different syntaxes into a single inner representation, which can be then used to simply compute deltas between different revisions.

The algorithms to complete these deltas has one known issue: if there are multiple issues with the same message (e.g. because variable with name `foo` was declared, but not used, in multiple functions) and later some of these issues are fixed, the number of issues is correct, but the indicated lines may be incorrect. This is because the script must cope with changing code; an issue on line X in one revision can be on line Y in another one, and that would require employing much more complex algorithms that would use information from git and understand which lines moved where. Thus, in such cases, the behaviour selecting specific instances of the same kind of issue is undefined.

Finally, while it is possible to find differences between revisions within the results of a single tool, albeit with the small instability in case of multiple similar entries mentioned above, doing this between tools on a single revision proved a much more complex task. Every tool describes the same issue with different words, so to be able to automatically compute any differences, such an algorithm would have to understand the issue in all details. Thus, cross-tool differences are not computed automatically, but manually for the cases where it is reasonable given to the number of issues that must be analysed. Standard Unix tools like `diff` and `grep` were used in these cases.

6 Results

In this chapter, the results of every tool are compared and analysed. If not stated otherwise, the number of issues is for mkfs-specific files, i.e., for files in `mkfs/` directory. Every tool has its own section in which its performance is analysed across of multiple revisions of mkfs in detail.

Table 6.1 offers an overview of how many issues did every tool find on various revisions. These revisions are stored in the project's git repository and they are identified by their hash, so this table also shows which revision follows which. Selected revisions (annotated with *total issues*) shows the number of outstanding issues on this specific point of development.

For the remaining revisions, the table shows new/fixed issues: $+x$ denotes the number of new issues found, $-x$ denotes the number of issues fixed between this and the previously tested revision. For example, revision `a887c950` has a value $+1/-3$ for Coverity. That means that, according to Coverity, one new issue appeared in this revision, while 3 others were fixed. A zero, in this case, means no change. A — dash means that the tool was not used in this specific revision. The numbers of issues were gained with every tool set up to the strictest analysis.

It is apparent from the table, even on the first glance, that the performance of the tools varies widely. Not only in absolute numbers of reported issues (of which some are false positives), but also in detecting specific issues. For example, revision `a9dad670` fixed 54 issues according to GCC, but according to Coverity, it caused 4 new and did not fix anything.

Below, we have both a simple statistical analysis of what each tool found and a more detailed look at some specific issues and revisions, especially where the tools have seriously different results.

6.1 CppCheck

CppCheck (and Codacy, which is using it) found fewer issues than other tools. When the kind of issues found is analysed, it becomes apparent that this tool is greatly limited. Still, this tool is the easiest to use and it is open source, which can make it a useful entry point for projects that do not use any other form of analysis.

The only two revisions we mention here are v4.6.0 and 4.7.0, before and after our patches. The differences between other revisions are negligible.

In version 4.6.0, 5 issues were found in mkfs-specific files, and 460 issues in whole xfsprogs. From these, 100 issues were not stylistic.

The issues found in mkfs are:

1. `mkfs/xfs_mkfs.c:1067`: *Checking if unsigned variable 'blocksize' is less than zero.*
2. `mkfs/xfs_mkfs.c:1698`: *Checking if unsigned variable 'sectorsize' is less than zero.*
3. `mkfs/xfs_mkfs.c:1225`: *Checking if unsigned variable 'sectorsize' is less than zero.*
4. `mkfs/xfs_mkfs.c:2487`: *Condition '0' is always false*
5. `mkfs/xfs_mkfs.c:2733`: *The scope of the variable 'bucket' can be reduced.*

Commit	CppCheck	Codacy	Coverity	GCC	Clang
Total issues					
(tag: v4.11.0-rc1) 07a3e793	1	3	119	30	34
(tag: v4.7.0) d7e1f5f1	1	4	119	30	28
Changes					
[Last of our set] 2aca16d6	0	0	0	0	0
aa3034d4	0	0	+2/−2	0	0
6de2e6c0	0	0	+5	0	0
ddc3b2da	0	0	0	0	0
06ac92fd	0	0	+12/−3	+1	+1
27ae3a59	0	0	+1	0	0
3ec1956a	0	0	+1/−3	0	0
6c855628	0	0	+2/−2	−2	−2
627e74fd	−3	−2	0	+2	+2
9090e187	0	0	+2	0	0
1974d3f1	0	0	0	−1	0
56e4d368	0	0	−3	0	0
a9dad670	0	0	+4	−54	−54
147e0f31	0	0	+3	0	0
c81c8460	0	0	0	−50	−50
a887c950	0	0	+1/−3	+13	+12
5f1a2100	0	0	0	0	0
ff21c709	0	0	0	+1	+1
[First of our set] 4a32b9e9	−1	−1	0	0	0
Total issues					
[Before our set] 6aa32b47	5	6	111	121	117
(4.6.0) 09033e35	5	6	111	121	117

Table 6.1: An overview of issues found by the tested tools on specific revisions in mkfs-only files.

All these issues were present for multiple years. Precise dating is difficult, however, because e.g. issue 1 is blamed to a commit 16 years old. But at that time, the variable was signed. Thus, the issue appeared some time later, when the specific variable was turned to unsigned, but not every use was fully converted.

Neither of these issues is of any seriousness. Every found check of an unsigned variable being less than zero is, in fact, a less-or-equal check. Listing 6.1 shows the specific code for both cases of offending sectorsize. Thus, `unsigned_variable <= 0` may be misleading, but functionally is equivalent to `unsigned_variable == 0`. And the condition '0' being always false is a value intentionally passed to a macro.

Other tools do not report this case of unsigned variable comparison, likely because the lower-than symbol, in this case, does not have any effect.

Listing 6.1: Condition in which unsigned sectorsize is tested to be less than zero.

```
if (sectorsize <= 0 || !ispow2(sectorsize))
        // do something
```

The patches we wrote removed most of the offending code, so only 1 issue was found in mkfs-specific files in version 4.7.0. In that version in the whole xfsprogs, 440 issues were found, from which 100 issues were not style issues.

The issue found in mkfs is:
`mkfs/xfs_mkfs.c:2918`: *The scope of the variable 'bucket' can be reduced.*

When compared with Codacy, CppCheck reports one issue twice: Checking if an unsigned variable is less than zero. This happens in two places, but Codacy ignores the second occurrence. On the other side, CppCheck did not find any issue in `mkfs/proto.c` file, where Codacy did.

These differences might be caused by a different configuration of CppCheck, because we used the default configuration, but do not know what changes Codacy did.

Despite this, the results of those two tools are very similar when compared to others, so we use only CppCheck in further comparison with other tools. CppCheck is selected because we have greater control over it, unlike cloud service Codacy, and on this sample produced no false positives, although the usefulness of some reports is arguable.

The low number of issues found can be attributed to the detailed review of all patches submitted to the project. If most issues are usually spotted during the development of the patches and fixed before they are merged into the code, it leaves space for only more complex and not so obvious issues, which CppCheck analysis is not capable of finding and a stronger tool is necessary.

6.2 Codacy

Codacy shows only per-commit and total issues for a branch. That is, a developer can view whether a specific commit fixed or caused an issue, and can see what are the issues for the top of the repository, but checking the complete state at a particular point in the history requires creating a new branch, which is uncomfortable, but manageable for a private[1] repository. In a repository with many contributors, it can be confusing.

Codacy provides some rating on the project's page [1], which considers xfsprogs as a quality project (A-grade), but the weight of this rating is unclear and rather informal. Also, it is not clear without checking every issue, what metrics Codacy uses to assess the type, whether it is style, error or security issue.

In mkfs-specific files in version 4.6.0, 6 issues were found. Whole xfsprogs had 839 issues, from which 809 were code style issue and 30 were potential errors.

Codacy found most of the same issues as CppCheck with few exceptions. It reported these two issues:

1. `mkfs/proto.c:49`: *The function 'setup_proto' is never used.*
2. `mkfs/proto.c:601`: *The function 'parse_proto' is never used.*

However, these functions are used in `mkfs/xfs_mkfs.c` file, thus they are false positives. Also, CppCheck found two places on which `sectorize` is checked to be less than zero, but Codacy reports it only once. Curiously, the same issue appears in two places not far away, and in both cases, it is in this exact condition as can be seen in Listing 6.1 in Section 6.1, just inside of

[1] In the sense of being the only user, not in terms of visibility.

different blocks, but still in the same function and path to both places is possible. Why Codacy reports only one of those issues is unclear.

In version 4.7.0 in mkfs-specific files, 4 issues were found. Whole xfsprogs had 749 issues, from which 719 were code style issue and 30 were potential errors.

The four issues found in this version are similar to what can be seen for the 4.6.0:

1. `mkfs/xfs_mkfs.c:2918`: *The scope of the variable bucket can be reduced.*
2. `mkfs/maxtrres.c:31`: *The function max_trans_res is never used.*
3. `mkfs/proto.c:49`: *The function setup_proto is never used.*
4. `mkfs/proto.c:601`: *The function parse_proto is never used.*

Also in this case, the supposedly unused function `max_trans_res` is in fact used in another file.

In total, Codacy results are similar to CppCheck itself, but with more of false positives. The only advantage it offers is automated integration with GitHub. With a correct set-up, it can ensure that every push into the repository is tested.

6.3 GCC and Clang

Despite being developed independently, GCC and Clang are very similar in what they found, with GCC finding a few more issues. In this section, we describe some of the notable differences between those two tools and compare them to others where it is reasonable.

6.3.1 Version 4.6.0

Table 6.2 compares GCC and Clang in this revision, and we specifically look at the differences between these two tools and Codacy. Other issues are not listed here due to their amount, but the reader can find them on an attached optical disc, or replicate them using the tools attached to this work.

Tool	`mkfs/xfs_mkfs.c`	`mkfs/proto.c`	Whole xfsprogs
GCC	121	2	2013
Clang	113	4	2597

Table 6.2: Comparison of the number of issues reported by GCC and Clang in version 4.6.0.

As is shown in Section 6.2, Codacy found two issues in the `proto.c` file too. Curiously, Codacy found two unused functions, while GCC found these issues:

1. `mkfs/proto.c:270`: *comparison between signed and unsigned integer expressions*
2. `mkfs/proto.c:332`: *unused parameter 'mp'*

These issues are not in the two functions found by Codacy. However, they are inside of functions called from the ones marked as unused by Codacy. It is possible that they were not reported because of this, but given that the mentioned Codacy issues are false positives and that Codacy did not found many other issues, it is likely that Codacy simply did not notice them, while GCC did.

In addition to the two issues found by GCC in `mkfs/proto.c`, Clang found two other issues (both of the same kind):

1. `mkfs/proto.c:130`: *missing field 'tr_logcount' initializer*
2. `mkfs/proto.c:631`: *missing field 'tr_logcount' initializer*

These two new issues, complaining about a missing field, are probably false positives because, on these lines, a structure with all members zeroed is created, as can be seen in Listing 6.2.

Listing 6.2: One of the two lines on which Clang reports a missing field in structure initialization.
```
struct xfs_trans_res    tres = {0};
```

The difference between GCC and Clang in `mkfs_xfs.c` file is 8 issues in absolute numbers, and the real difference is not much bigger; GCC reports more cases of a comparison between signed and unsigned integer than Clang does. Clang, on the other hand, reports few cases of this issue:

`mkfs/xfs_mkfs.c:2906`: *cast from 'char *' to 'xfs_alloc_rec_t *' (aka 'struct xfs_alloc_re c *') increases required alignment from 1 to 4*

Other than that, they report the same issues.

6.3.2 Revision a887c950

GCC detected 13 new issues in `mkfs/xfs_mkfs.c`. These 13 issues are only of two kinds:

1. `mkfs/xfs_mkfs.c:1487`: *comparison is always false due to limited range of data type*
2. `mkfs/xfs_mkfs.c` multiple occurences: *passing argument 2 of 'illegal' discards 'const' qualifier from pointer target type*

Clang detected all the 12 occurrences of the second issue but missed the first comparison issue. The offending line for the missed issue is shown in Listing 6.3. A closer look on this line reveals that there is an explicit type casting. The new type is a signed integer. However, the variable `logagno` is declared with type `xfs_agnumber_t` and this type is declared as an alias to `__uint32_t` in file `libxfs/xfs_types.h`.

According to the standard of C language, the type casting has a precedence over comparison [3, A.2.1], so Clang, when evaluating this line, sees an integer. That is technically correct, but in a wider context, it is also clear that the possible values are still limited by the original type, and so this comparison will be always false. Which is what GCC noticed and also correctly reported. No other tool reported this issue.

Listing 6.3: Line on which GCC found the comparison issue.
```
if ((__int64_t)logagno < 0)
        // do something
```

The other issues are similar to what Clang found but appears to suffer the instability in parsing mentioned in Section 5.2. All these issues are caused by passing a constant string to a function which does not has the `const` keyword for an argument: `illegal(value, ,,b log‘‘);`. The file `xfs_mkfs.c` contains 44 such calls, but only some of them were added in this revision. Thus, we correctly detected these issues, but some of the line numbers we see had this issue before. If we would want to see exactly which lines were added, the easiest way is to look to look at changes in this specific commit.

6.3.3 Version 4.7.0

Clang found 3 more issues in this revision than GCC did in file `mkfs/proto.c`, as is shown in the comparison in Table 6.3. Two of them already appeared in the description of revision 4.6 in Section 6.3.1 and were not fixed, the third issue is of the same kind and was introduced by some other patch other than which are part of this work.

Tool	`mkfs/xfs_mkfs.c`	`mkfs/proto.c`	Whole xfsprogs
GCC	28	2	2013
Clang	23	5	2511

Table 6.3: Comparison of the number of issues reported by GCC and Clang in version 4.7.0.

Almost all issues found in `mkfs/xfs_mkfs.c` are about a comparison, with only two exceptions:

1. `mkfs/xfs_mkfs.c:728`: *unused parameter 'lsectsz'*
2. `mkfs/xfs_mkfs.c:1896`: *passing argument 2 of 'unknown' discards 'const' qualifier from pointer target type*

Most issues in Clang are also about a comparison, with multiple versions of wording, because where GCC uses only one universal message, Clang uses a template into which it substitutes specific types. This makes the analysis of the results more challenging but does not have any effect on the results. In this revision, Clang does not report any new kind of issues against version 4.6.0.

6.4 Coverity

Coverity, when it is run manually in the docker container, found a comparable number of issues as GCC and Clang in mkfs-specific files. On the other hand, the online version did not find any issues in mkfs on a recent (4.11) version [14], compared to 111 found on a manual run with the highest level of aggressiveness. What effect the aggressiveness level has can be seen in Table 6.4. The numbers in this table are for mkfs-specific files. The tested revision is 07a3e793 (v4.11.0-rc1). From the results, it is probable that the service is using configuration similar to the low aggressiveness. Especially when numbers for the whole xfsprogs are compared.

	online	low	medium	high	custom
Issues reported	0	0	36	97	111

Table 6.4: Comparison of issues found by Coverity online service and Coverity run locally with different levels of aggressiveness.

In most of this section, we focus on results gained from the custom (highest) level. The online service is briefly described and some interesting statistics comparing xfsprogs with other open-source projects (in an aggregated manner) are provided only in Section 6.4.1.

The reasons for selecting the custom level aggressiveness as the level on which our analysis focusses is that for other tools we used the strictest settings available, thus, analysing relaxed approach on any other level is not comparable to other tools.

Unlike CppCheck or the compilers, Coverity can also show examples of the data flow for which a defect can appear. E.g. tracking multiple conditions and noting if true or false branch was

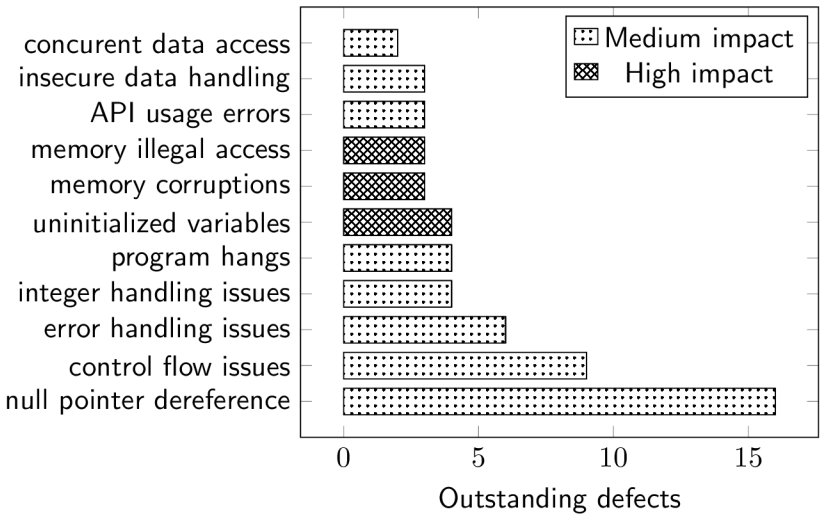

Figure 6.1: Outstanding defects per category for whole xfsprogs.

taken. This is especially useful when the case is not clear and the path includes a longer chain of conditions over a larger part of the code. Both the online service and the locally-run application provides this information. However, it is not particularly useful in our comparison of what issues were found.

6.4.1 Online Service

The online service available at https://scan.coverity.com/projects/xfsprogs provides various statistics in addition to reported issues. The online service found no issues in mkfs. In whole xfsprogs, it reported 71 issues, which is similar to what the manual execution on the low level found (77 issues). The service also provides a view on specific categories, as can be seen in Figure 6.1

This service is used since 2013 and since xfsprogs was first analysed, a total number of 273 issues was found. Of these, 176 were fixed and 26 dismissed as intentional or false positives. The average defect density according to this analysis is currently 0.52 issues per 1,000 lines, with 135,302 lines analysed.

When xfsprogs was analysed for the first time by this service, 139 issues were found, but many were fixed shortly afterwards and the number of outstanding defects is not changing much, averaging between 60 and 70 issues at any specific revision. When compared with other open-source projects of similar size[2] analysed by this service, xfsprogs oscillates around the average value, which is 0.5.

It is also useful to note that the defects are not distributed equally in the whole xfsprogs. When we look at the defects rate per component, seen in Table 6.5, we can see that the tools used by developers for e.g. debugging, or by advanced users (`xfs_copy`, `xfs_logprint`) have a higher rate than the tools intended for a general use, like `mkfs`. The high rate in `libxcmd` and `libhandle` is caused by the small size of these two components. Both has only a single issue, but as `libhandle` has under 500 lines, the average per 1000 lines makes it look worse.

[2] 100,000 to 400,000 lines of code.

	libxfs	libxlog	xfs_repair	xfs_db	xfs_copy	xfs_fsr	xfs_io
Lines of code	43,770	1,165	22,620	18,508	1,043	1,354	6,970
Defect density	0.80	0.86	0.44	0.49	2.88	0.74	0.00
	xfs_logprint	xfs_quota	mkfs_xfs	xfs_growfs	libhandle	libxcmd	other
Lines of code	2,616	4,037	3,540	408	493	1,338	27,539
Defect density	1.15	0.50	0.00	0.00	2.03	1.38	0.15

Table 6.5: Defect density per component as reported by Coverity online service.

6.4.2 Local analysis

This section analyses the results from the local execution of Coverity on the highest level of aggressiveness.

In mkfs-specific files in version 4.6.0, 111 issues were found, while whole xfsprogs had 3309 issues. From the issues reported for mkfs, 60 is a complaint about dereferencing a pointer that might be null in `printf` or `fprintf` call.

An example of a line with such a warning is in Listing 6.4. The issue lies with Gettext[3]. The `_` macro is translated as a `dcgettext` call and it is the result of this function that Coverity can't verify. And because there is no check of the return value for null before it is passed to `printf`, Coverity makes an aggressive assumption and raise a warning.

Listing 6.4: `xfs_mkfs.c:1713`: Line which is reportedly dereferencing a potentially null pointer with Gettext

```
printf(_("%s␣version␣%s\n"), progname, VERSION);
```

These issues can probably be considered false positives, or at least intentional. A brief search in Gettext implementation suggests that if Gettext cannot allocate memory for a translated string, it simply returns the original one. For example, see file `/gettext-runtime/intl/dcigettext.c`, line 391 in Gettext source code [11].

Only 10 of the dereferencing issues are related to something else than `dcgettext` and might be useful. An example of such issue is Listing 6.5. In this case, memory is allocated with a `malloc` call. The returned value is not tested, so it is possible that null is passed to the `read` call.

Listing 6.5: `proto.c:66`: Line which is reportedly dereferencing a potentially null pointer - no malloc check.

```
buf = malloc(size + 1);
if (read(fd, buf, size) < size) {
        // do something...
```

These two issues also illustrate the difference between the aggressiveness level. The Gettext-related issue is reported only on high or custom level, but the malloc issue is reported also on medium level.

When compared to GCC or Clang, Coverity finds different issues than the other tools. While GCC reports a lot of comparison between signed and unsigned integers or discarding `const`

[3]Gettext is a tool/library for translation of programs. It generates a list of marked string from a program. These string can be translated and packaged with the compiled program. When the program is run, Gettext selects the correct language based on system configuration.

qualifier, Coverity finds a lot of potential null pointer dereferences and numeric types overflows when 32bit and 64bit arithmetics are mixed.

6.5 Summary

As we have seen, the results of the tools vary widely, both in types of issues the tools report and in their amount. The mediocre results of CppCheck and Codacy can be probably attributed to the fact that we tested it on a production code which already passed a certain quality assurance and thus, the kinds of issues these tools are best capable of finding were already fixed.

Coverity and both compilers were capable of finding less obvious defects, but the price for a too high sensitivity was a lot of reported issues with only a minimal, if any, effect, like the comparisons between signed and unsigned integers from GCC.

In any case, this work shows that on a relatively error-free code, there is only a minimum of defects that would be reported by multiple tools. This makes it apparent that it is useful to use as many diverse methods in the analysis as possible.

The most helpful tool from the tested ones was Coverity, not least because of its ability to show the flow of the program in which the defect can appear. However, the free analysis for open source projects is limited to more obvious defects and for non-open source projects, or for a detailed analysis, it requires a paid license.

This analysis was done after the first patches were written and merged, but found minor issues only, which speaks well for the quality of mkfs, at least in terms of correctness with respect to the language standard. Whether the code is well structured or not, or if it does what is expected from it on a higher level, is not possible to assess with these tools.

Our patches reduced the number of issues found by all tools except Coverity. Coverity saw a small growth on higher levels of aggressiveness[4]. On the other hand, GCC, Clang and CppCheck saw a fall to roughly about one quarter. Codacy, after subtracting false positives, shows the same trend, although the only tool from the tested ones, which was used during the development of these patches, is the online Coverity service.

[4] On the low level, Coverity found no defect both before and after the patches.

7 Conclusion

As we have found, the accumulation of technical debt in long-living projects can impair the understanding of the code and grow as a snowball, with each change requiring more ad-hoc adjustments and edits than the previous one. When this happens, it is important to devote some effort to clean the code, even if it can take a long time, because otherwise, the situation will only grow worse.

As Chapter 3 shows, we have begun this work and successfully merged the first set of changes. Because the development process continues at a slower pace than we expected, not all of the desired changes were merged before this work was published. This does not change our plan for merging them. Rather, we only have to find better processes that will limit long delays and speed up the merging of the changes into the project. Some of these possible changes were discussed in Section 3.5.

We also compared the online Coverity service xfsprogs is using with few other tools, to see how effective the current reviews and tests are. The results in the Chapter 6 speak rather well. All the found issues in mkfs.xfs were only of a minor importance, even though different tools found different kinds of issues. With higher sensitivity, the tools were reporting more of those issues, but with diminishing importance and growing amount of false positives. Also, we found out that our patches lowered the number of issues found in mkfs by most of the tools.

While it might be useful to incorporate the other tools and let the tools use more aggressive assumptions, it is uncertain if the return of investment into fixing a large amount of minor and stylistic issues is positive, or if the effort is better spent on keeping the XFS filesystem competitive with other modern and much younger filesystems.

Bibliography

[1] *Codacy: jtulak/xfsprogs results*. https://www.codacy.com/app/jtulak/xfsprogs-dev/dashboard. [Online; visited 29. Apr. 2017].

[2] *CSBUILD MAN PAGE*. https://www.mankier.com/1/csbuild. [Online; visited 1. May 2017].

[3] C language standard – committee draft, iso 9899:tc3. 2007.

[4] Apple. *Debugging with Xcode*. https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/debugging_with_xcode/chapters/static_analyzer.html#//apple_ref/doc/uid/TP40015022-CH11-DontLinkElementID_16, 2016. [Online, visited at 01. Jan. 2017].

[5] D. Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. *Proc. of CAV'11*, 2011.

[6] Dave Chinner. *Re: [RFC PATCH 1/2] mkfs: unify numeric types of main variables in main()*. https://www.spinics.net/lists/linux-xfs/msg05750.html, 2017. [Online, visited at 9. Apr. 2017].

[7] David Chinner. *[RFC, PATCH 00/15] mkfs: sanitise input parameters*. http://oss.sgi.com/archives/xfs/2013-11/msg00833.html, 2013. [Online, visited at 01. Jan. 2017].

[8] David Chinner. *Linux Filesystems: Where did they come from?* https://www.youtube.com/watch?v=DxZzSifuV4Q&index=2&list=WL, 2014. [Video of a presentation; visited 1. May 2017].

[9] David Chinner. *ANNOUNCE] xfsprogs: master branch updated to c5d584c*. http://oss.sgi.com/archives/xfs/2016-05/msg00230.html, 2016. [Online, visited at 01. Jan. 2017].

[10] Clang community. *Clang Static Analyzer*. https://clang-analyzer.llvm.org/. [Online; visited 12. May. 2017].

[11] GNU community. *Gettext Git repository*. http://git.savannah.gnu.org/cgit/gettext.git. [Online; visited 12. May. 2017].

[12] XFS community. *mkfs.xfs manual page*. XFS man page. [Current version at 01. Jan. 2017].

[13] XFS community. *XFS manual page*. XFS man page. [Current version at 01. Jan. 2017].

[14] Coverity. *Coverity Scan: xfsprogs*. https://scan.coverity.com/projects/xfsprogs. [Online; visited 01. Jan. 2017].

[15] Docker. *What are containers*. https://www.docker.com/what-container. [Online; visited 29. Apr. 2017].

[16] Christopher Hellwig. *XFS: The big storage file system for Linux*. http://oss.sgi.com/projects/xfs/papers/hellwig.pdf. [Online; visited 01. Jan. 2017].

[17] Bohuslav Křena and Tomáš Vojnar. Automated Formal Analysis and Verification: An Overview. *International Journal of General Systems*, 42(4):335–365, 2013.

[18] Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. *ESEC-FSE'05, September 5–9*, 2005.

[19] Daniel Marjamäki. *Cppcheck Design*. http://sourceforge.net/projects/cppcheck/files/Articles/cppcheck-design-2010.pdf/download, 2010. [Online, visited at 01. Jan. 2017].

[20] Microsoft. *FILETIME structure*. https://msdn.microsoft.com/en-us/library/windows/desktop/ms724284(v=vs.85).aspx. [Online; visited 01. Jan. 2017].

[21] Microsoft. *How NTFS Works*. https://technet.microsoft.com/en-us/library/cc781134(v=ws.10).aspx, 2013. [Online; visited 01. Jan. 2017].

[22] Microsoft. *NTFS Overview*. https://technet.microsoft.com/en-us/library/dn466522(v=ws.11).aspx, 2016. [Online; visited 01. Jan. 2017].

[23] Antoine Miné, Laurent Mauborgne, Xavier Rival, Jerome Feret, Patrick Cousot, and et al. Taking static analysis to the next level: Proving the absence of run-time errors and data races with astrée. *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[24] Steven Musil. *Australia police discourage use of Apple maps app after rescues*. https://www.cnet.com/news/australia-police-discourage-use-of-apple-maps-app-after-rescues/, 2012. [Online, visited at 01. Jan. 2017].

[25] Steve Naroff. *New LLVM C Front-end*. http://llvm.org/devmtg/2007-05/09-Naroff-CFE.pdf. [Online; visited 12. May. 2017].

[26] Jochen Pohl. *Lint manual page*. http://www.unix.com/man-page/FreeBSD/1/lint. [Current version at 01. Jan. 2017].

[27] Inc. Red Hat. *What are the file and file system size limitations for Red Hat Enterprise Linux?* https://access.redhat.com/solutions/1532, [Online; visited 19. May 2017].

[28] Luis R. Rodriguez. *Re: [PATCH 08/12] mkfs: replace variables with opts table: -b,d,s options*. https://www.spinics.net/lists/linux-xfs/msg06273.html, 2017. [Online, visited at 26. Apr. 2017].

[29] Eric Sandeen. *What is the maximum number of inodes in Linux filesystems?* https://www.quora.com/What-is-the-maximum-number-of-inodes-in-Linux-filesystems-I-found-suggestion-that-for-E 2014. [Online; visited 01. Jan. 2017].

[30] SGI. *Original XFS Documentation*. http://oss.sgi.com/projects/xfs/design_docs/xfsdocs93_pdf/. [Online; visited 1. May 2017].

[31] SGI. *XFS Documentation, ch. 1: XFS Background*. http://xfs.org/docs/xfsdocs-xml-dev/XFS_User_Guide/tmp/en-US/html/xfs-background.html. [Online; visited 01. Jan. 2017].

[32] SGI. *XFS Documentation, ch. 4.9: mkfs - Realtime*. http://xfs.org/docs/xfsdocs-xml-dev/XFS_User_Guide/tmp/en-US/html/ch04s09.html. [Online; visited 01. Jan. 2017].

[33] SGI. *XFS Filesystem Structure*. http://xfs.org/docs/xfsdocs-xml-dev/XFS_Filesystem_Structure//tmp/en-US/html/index.html. [Online; visited 01. Jan. 2017].

[34] Inc. Synopsys. *Coverity 8.7.1 Command and Ant Task Reference*. Bundled with Coverity installation., 2017.

[35] Jan Ťulák. *Docker Hub: jtulak/cppcheck*. https://hub.docker.com/r/jtulak/cppcheck/. [Docker image].

[36] Jan Ťulák. *[RFC, PATCH 00/17] mkfs: sanitise input parameters*. http://oss.sgi.com/archives/xfs/2015-06/msg00309.html, 2015. [Online, visited at 01. Jan. 2017].

[37] Jan Ťulák. *[PATCH 00/19 v2] mkfs cleaning*. http://oss.sgi.com/archives/xfs/2016-04/msg00542.html, 2016. [Online, visited at 01. Jan. 2017].

[38] Jan Ťulák. *[RFC PATCH 00/22] mkfs.xfs: Make stronger conflict checks*. https://www.spinics.net/lists/linux-xfs/msg02728.html, 2016. [Online, visited at 01. Jan. 2017].

[39] Jan Ťulák. *[RFC PATCH 0/8] mkfs: centralised conflict detection*. https://www.spinics.net/lists/xfs/msg41336.html, 2016. [Online, visited at 02. Aug. 2016].

[40] Jan Ťulák. *[PATCH 00/12] mkfs: save user input into opts table*. https://www.spinics.net/lists/linux-xfs/msg06158.html, 2017. [Online, visited at 23. Apr. 2017].

[41] Jan Ťulák. *Re: [RFC PATCH 1/2] mkfs: unify numeric types of main variables in main()*. https://www.spinics.net/lists/linux-xfs/msg05696.html, 2017. [Online, visited at 7. Apr. 2017].

Appendices

The attached optical medium has this content:

- `docker/` – The directory with the containers and used scripts.
- `tex/` – Latex source code, and data used in this work in `results` subdirectory.
- `xfsprogs-dev/` – The directory with xfsprogs git repository. Includes also the changes that were not yet merged in `conflicts` branch.

The `docker/` directory contains subdirectories for every container, plus these scripts, which are described in Chapter 5:

- `prepare.sh` – A script that will download sources for the docker containers. Not necessary when the directories already exists.
- `tests.py` – A script that runs all the tests.
- `format-outputs.sh` – A script to preprocess the output of the used tools for later analysis.
- `parse.py` – A simple analysis of the outputs, capable of printing differences between revisions.
- `README.md` – An example of how to use these scripts.