jamk

Machine Learning aided Linux Kernel Code Analysis

Heikki Krogerus

Master's thesis

May 2024

Information and Communication Technology

Artificial Intelligence and Data analytics

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Heikki Krogerus

Machine Learning aided Linux Kernel Code Analysis

Jyväskylä: JAMK University of Applied Sciences, May 2024, 74 pages.

Information and Communication Technology. Artificial Intelligence and Data analytics. Master's thesis.

Permission for web publication: Yes

Language of publication: English

Abstract

Several automated tests are used to find issues in every code change proposal for the Linux kernel. Despite automated testing, each change needs a review that is done by humans. Due to the small number of dedicated reviewers, a lot of issues have goon undetected.

The problem with automated testing has been that it relies on static analysis which does not detect issues very efficiently. The few reviewers are a result of not just anyone being able to review the new code. The reviewer does not only need to understand the architecture of Linux kernel itself, but also the hardware or technology that the proposed change was targeted for.

To see if modern deep learning techniques could be used to mitigate the problem, a set of steps, each progressing from the previous one, were designed. The focus was not only on the mitigation of the problem, but in the utilisation of deep learning to aid Linux kernel development in general. The first step in the plan was used as the Proof-of-Concept.

The goal of the first step in the plan was to construct a model that can detect rudimentary quality issues in the proposed code changes, but that could not be achieved due to the problems with the learning data. The code reviews from the past were used as the learning data. The assumption was that if a code change was revised, its overall quality improves. That assumption turned out not to be true.

Nevertheless, the deep learning model that was produced was first taught to understand Linux kernel source code in unsupervised fashion, and that the model could obviously learn.

By producing a Large Language Model that understands Linux kernel source code at the general level shows that deep learning can be used to aid Linux kernel development. The model can be fine-tuned to perform several different tasks. The failure to fine-tune the model to detect issues in the code changes were not caused by the model design or architecture. It was caused by the poor-quality learning data.

Keywords/tags (subjects)

Linux Kernel Development, C-Programming Language, Deep Learning, Large Language Models

Miscellaneous (Confidential information)

jamk

Kuvailulehti

Heikki Krogerus

Machine Learning aided Linux Kernel Code Analysis

Jyväskylä: Jyväskylän ammattikorkeakoulu. Toukokuu 2024, 74 sivua.

Tietojenkäsittely ja tietoliikenne. Artificial Intelligence and Data analytics. Opinnäytetyö YAMK.

Verkkojulkaisulupa myönnetty: kyllä

Julkaisun kieli: englanti

Tiivistelmä

Jokainen koodimuutos, jota ehdotetaan Linux-käyttöjärjestelmäytimeen, testataan monilla eri automaattisilla testeillä. Tästä huolimatta jokainen muutos on vaatinut ihmisten tekemän katselmoinnin. Johtuen rajallisesta määrästä ihmisiä, jotka pystyvät katselmoimaan Linux koodia, monia virheitä on jäänyt huomaamatta.

Automaattisten testien ongelmana on ollut niiden keskittyminen staattiseen analyysiin, jolla ongelmia ei pystytä löytämään tehokkaasti. Pieni määrä ihmisiä, jotka ovat katselmoineet Linux-käyttöjärjestelmäytimen koodia on seurausta tehtävän vaativuudesta. Linuxin koodin katselmointi ei vaadi ainoastaan kykyä ymmärtää Linux-käyttöjärjestelmäytimen rakennetta, vaan myös laitestoa tai teknologiaa johon koodimuutoksella vaikutettaan.

Jotta nähtäisiin kuinka moderneja syväoppimistekniikoita ja malleja voitaisiin hyödyntää tilanteen parantamiseksi luotiin joukko toisistaan periytyviä kehitysaskeleita. Tavoitteena ei ollut ainoastaan helpottaa ongelmaa koodin katselmoinnissa, vaan nähdä miten syväoppimista voidaan Linux kehityksessä hyödyntää yleisesti.

Ensimmäisenä tavoitteena oli luoda malli, jolla voitiin osoittaa syväoppimisen tarjoavan mahdollisuuksia Linux kehityksessä. Ensimmäisen tavoitteen mallin tuli kyetä löytämään perusongelmia ehdotetuista koodimuutoksista, mutta tähän mallia ei ollut mahdollista kouluttaa johtuen huonolaatuisesta opetusdatasta. Oletuksena oli ollut, että jos koodinkatselmointi johti uuteen versioon koodimuutoksesta, koodimuutoksen laatu paranee. Tämä oletus ei täysin vastannut odotuksia.

Tästä huolimatta, malli, joka kehitettiin tätä tehtävää varten, koulutettiin ensin ymmärtämään Linuxin lähdekoodia yleisellä tasolla, ja malli selkeästi kykeni.

Kouluttamalla malli, joka ymmärtää Linuxin lähdekoodia osoittaa, että syväoppimista voidaan hyödyntää Linuxin kehitystyössä. Sama malli voidaan jatkokouluttaa moniin eri tehtäviin. Epäonnistuminen kouluttaa malli löytämään virheitä koodimuutoksista ei johtunut mallin arkkitehtuurista, vaan opetusdatan heikosta laadusta.

Avainsanat (asiasanat)

Linux Kernel-kehitys, C-Ohjelmointikieli, Syväoppiminen, Suuret Kielimallit

Muut tiedot (salassa pidettävät liitteet)

Contents

Figures

Acronyms

AI Artificial Intelligence

API Applications Programming Interface

AST Abstract Syntax Tree

BERT Bidirectional Encoder Representations from Transformers

CNN Convolutional Neural Networks

CSV Comma Separated File

DL Deep Learning

GDPR General Data Protection Regulation

GPT Generative Pre-Trained Transformer

LKML Linux Kernel Mailing List

LLM Large Language Model

LSTM Long Short-Term Memory

MFD Multi-Functional Device

ML Machine Learning

MLM Masked Language Model

NLP Natural Language Processing

NSP Next Sentence Prediction

PCI Peripheral Component Interconnect

PPML Privacy Preserving Machine Learning

ReLU Rectified Linear Unit

RNN Recurrent Neural Network

SLM Small Language Model

USB Universal Serial Bus

1 Introduction

1.1 Motivation and Research Problem

Linux kernel development relies heavily on code reviews that are done by Linux Kernel subsystem maintainers, and a handful of other designated reviewers. Most of these people are listed in a specific "MAINTAINERS" file, which is part of Linux kernel code repository (“List of maintainers”, 2024). The amount of people doing code reviews is relatively small when considering the size of the Kernel code base, and more importantly, the amount of new code that is sent for review every day, and that creates problems (Corbet, 2006; Nicholson, 2016).

The first problem is that most maintainers in Linux kernel, even when paid to do the maintenance and code review, can only do that as part of their job - the list of maintainers file shows if the person is a full-time or part-time maintainer. For the part time maintainers and reviewers, because of the limited amount of time they can use for the review, when faced with large amount of new code, may not be able to go over that code as thoroughly as it should. That sometimes leads into buggy code being introduced into the Kernel, which then has to be fixed afterwards, but in most cases the problem is that new features, or even fixes, are delayed. A large amount of new code or code changes can be a problem even for the full-time maintainer if that maintainer maintains several drivers or subsystems in Kernel.

Another problem caused by the fact that the amount of the people doing the code reviews is small, is that if a maintainer or reviewer stops doing the work for one reason or the other, finding replacements can be very difficult. In these cases, the orphan Kernel component is only reviewed at general level by other reviewers how do not know all the details about that specific component until a new maintainer is found.

On top of review that is done by humans, each code change - called a patch - is analysed by several automated testing systems, most notable being Intel’s Linux Kernel Performance/0-Day test service (Wu, 2020). These testing systems run static code analysers and execute various tests, such as compile test to see does the code compile without errors and warnings, to find issues in the code (“Linux* Kernel Performance”, 2024).

The automated tests are good, but unfortunately, they can not detect issues that would require deeper understanding of the code such as memory leaks, buffer overflows and race conditions. They

also can not detect more minor problems, such as code styling issues. To help with styling issues, the Linux kernel source code supplies a separate script called Checkpatch (“Checkpatch”, 2024) which the developers can execute before sending their patches for review.

On top of the public mainline Linux kernel development, most larger organisations review their code internally before sending it upstream by using either a similar process that the mainline Linux kernel is reviewed, where patches are sent to the public mailing lists to be reviewed (“A guide to the Kernel Development Process”, 2024), or by using tools especially designed for code review, such as “Gerrit Code Review” (2024). Inside Intel Corporation, Linux kernel development process follows the mainline process. Intel has internal mailing lists where the initial code reviews are done. Despite the process that is used in these organisations, the problem with the lack of skilled reviewers affects also these internal reviews efforts. The maintainers and other reviewers are usually employed by these organisations, so the time they can spare for the initial internal reviews may be even more limited. The upstream reviews come always first.

To improve the situation, the goal of this study is to get an understanding about what could be done with modern Deep Learning (DL) solutions. The study has four research questions.

1. Can the large language models perform preliminary Linux kernel code review?
2. What other Linux kernel development tasks can the models perform?
3. What is the level of code understanding each task requires?
4. Which DL models are the most suitable for each task?

The first question is answered with a practical implementation in this study, and the rest of the questions are answered based on research on existing papers. By answering these research questions, this study constructs a progressive plan where each step advances the code understanding of the models. The practical implementation is the very first step in this plan. The results of this study are first used in Intel Corporation’s internal Linux kernel development, and after that they will be offered for the entire public Linux kernel development community.

1.2 Methodology

The research methodology used in this thesis has two parts. The theoretical part follows a Systemization of Knowledge (SoK) approach that emphasises the analysis of the existing papers and bringing

of new insight on the research area (“Systemization of Knowledge (SoK) Papers”, 2024). There is a lot of literature and implementations about DL aided code improvements and code analysis, and those also include code review. This study will attempt to summarise them and consider how suitable they are for Linux kernel code review.

A lot of the work that has been done is not focused on code review, but instead for example on how source code can be generated. The existing models are also not usually targeted for individual downstream tasks, but instead they are trained to act as general-purpose base models that can then be fine-tuned for some final downstream tasks. That makes them robust and multi-purpose. It also means they must understand usually natural English language on top of the programming languages, because they are trained with the comments included, and they are trained to understand multiple programming languages instead of only one. All that may not be a problem. The thesis will try to weight the pros and cons of having a more general-purpose base model versus an already targeted base model.

In the first part, the thesis will also look at the review process that is currently done by humans; the data, which is the Linux kernel source code and patches - code snippets introducing changes to the existing source code; and look at how all those can be split into different components. After this the thesis will look at how each of those components can be utilised in a Large Language Model (LLM).

The second part of the thesis is a Proof-of-Concept (PoC). This part will explain how to train the initial model that is taken into use first inside Intel to help Intel’s internal the Kernel Code review, and what the model does as well as how it is used. This thesis will also formulate a plan how to continue the work. The initial model will only look at the source code from function level, but later entire source code file, or even entire subsystem consisting of multiple files, should be possible to handle using the results of this study.

2 Linux Kernel Development Process

2.1 About Linux Kernel

Linux kernel is a general-purpose open-source operating system kernel that was originally developed and still maintained by Linus Torvalds (Bovet & Cesati, 2006). Linux is the leading server operating system, and Linux is also the most common operating system in smartphones because of Android and

other Linux based mobile phone operating systems (“Operating System Market Share Worldwide”, 2024).

Linux kernel is monolithic operating system kernel, which means the entire operating system is in kernel space. Basically, all supported subsystems (such as file systems, IPC, power management, etc.), and almost all the device drivers are all made part of the *mainline* (“Active kernel releases”, 2024) Linux kernel code base. The Linux kernel code maintained by Linus Torvalds together with all the other Linux kernel developers in the Linux kernel code repository (see section 3.3).

2.2 Linux Kernel Code Review

The code review of Linux kernel is conducted on subsystem specific mailing lists and the general Linux kernel mailing list (“A guide to the Kernel Development Process”, 2024).

The review process starts when a developer submits a patch to the mailing list, either mailing list dedicated to the subsystem or component (for example device driver) that the patch affects, the general Linux Kernel Mailing List, or both. The patches are therefore normal emails that anybody following those mailing lists can respond.

After the patch is submitted by the developer to the mailing lists, the maintainer and the other reviewers will comment the patch. If there are no issues, the other people who do not maintain the component that is being modified, may reply separately to the patch to let the maintainer know they approve the change. In this case the maintainer will pick the patch from mailing list and apply it to is code repository. In case issues are found in the patch, the people doing the review will reply to the patch email and point out the problematic parts. The developer will then revise the patch correcting the problems and resend the patch. The life cycle of the patch can be seen in Figure 1.

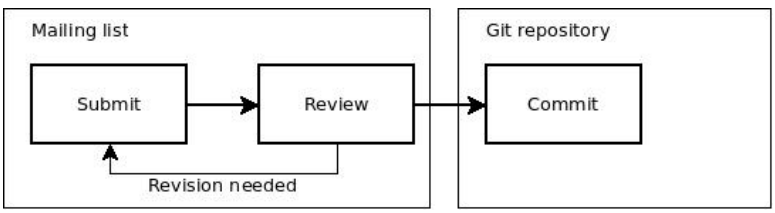

Figure 1: Simplified Patch Lifecycle

The process also expects the patches to be always versioned by having the version number in the subject line, and a changelog (“Submitting patches: the essential guide to getting your code into the kernel”, 2024). That information can be used when collecting learning data, but that is not done in

this study. The reason for that is because a common mistake in patches is that the version number is left out, or it is otherwise wrong. The version of a patch can be acquired more safely by simply looking at the chronological order where the patches were submitted.

2.3 Linux Kernel Releases

Every subsystem, and some of the larger drivers, has its own git repository which is maintained by a dedicated maintainer, or maintainers. All the new code is always first collected by the subsystem (and driver) maintainers to these git repositories. From those repositories Linus Torvalds then merges all the changes collected to them during a period called “merge window”. Merge window is a two-week period that starts the moment a new Linux kernel version is released. During the merge window the maintainers do not accept new code to their repositories. After the first two weeks has finished, the merge window is closed, and the first release candidate ***rc1*** of the next Linux Kernel version is released. Now only fixes are accepted into the Linus Torvalds’ kernel. The first release candidate ***rc1*** is followed by another release candidate every week, until ***rc7*** is released (in some rare cases ***rc8***). After ***rc7***, the next full version of Linux kernel is then released. The Linux kernel release cycle is therefore 9-10 weeks long (“A guide to the Kernel Development Process”, 2024).

This research will focus on Linus Torvalds “mainline” Linux kernel. Though, the maintainer git repositories have relevant code during the kernel development release cycle, all that code will in any case be merged when the next merge window opens.

3 Linux Kernel as Data

3.1 Linux Kernel Source Code

The Linux kernel is written in the C programming language (Kernighan & Ritchie, 1988). C is powerful standardised general purpose programming language (International Organization for Standardization, 2023). Although, it is necessary to have strong understanding of C programming language and Linux kernel programming, this thesis will not go over C programming in detail. The thesis will only look at the basic structure of C code to give an idea about how it can be utilised in this research in section 3.2. The thesis is especially looking into how C code can be partitioned into components that resemble natural language sentences.

As of Linux kernel v6.7 there are approximately 60 thousand files and 33 million lines of C code. The latest numbers can be calculated by running the script in appendix 1.

3.2 C Programming Language Structure

3.2.1 About C Source Code

The C code consists of tokens which form statements. Statements can be viewed as the actual commands that are given to the computer. All the executable code in C is contained in a block called a function. Functions are subroutines that are executed only when they are called. A program execution starts when a specific *main()* function is called. All the C code is defined in C code files and C header files.

A C code file is compiled into an object file that contains the code in form that the computer understands - binary code. C header files are included into the object during a specific stage of compilation called preprocessing, so the header files are not compiled themselves.

On top of the C code itself, the C files and headers contain comments and preprocessor macro definitions and calls. The comments are removed during compilation, and the preprocessor macros are replaced with the actual C code that they define during the preprocessing stage of the code compilation.

The preprocessing is usually executed separately with a discrete C preprocessor program. The C compiler itself is usually only responsible of performing lexical analysis, which tokenises the code, followed by the parsing stage, which produces a grammatical representation of the code called Abstract Syntax Tree (AST).

3.2.2 Tokens

Tokens in programming languages refer to the *lexical tokens*. Lexical tokens are words or punctuation marks with a specific meaning in the programming language. All the words and punctuation marks are *lexical tokens* in C. In C the exact meaning of a token is defined by its type. The tokens are categorised into six different token types in C (Kernighan & Ritchie, 1988, p. 191).

- Identifiers
- Keywords

- Constants
- Strings
- Operators
- Special Symbols

A statement (3.2.3) in C is a sequence of tokens that are constructed according to the grammatical rules of the C programming language. Therefore, the type of the token may be very important information to have. To include the token type into the C code before future processing, for example before constructing the *probabilistic tokens* for an LLM, the type needs to be embedded into the code somehow. The easiest way to do this is to place the token type before the token itself.

If we take the following C statement as an example `int i = 0;` the token types embedded into it would look like this:

```
<keyword>int
<identifier>i
<operator>=
<constant>0
<puctuation>;
```

The token type can be determined by any software the is designed to perform lexical analysis on C code, so for example the C compilers such as **gcc** (“GNU project C and C++ compiler”, 2024) or **Clang** (“Clang: a C language family frontend for LLVM”, 2024). There are also dedicated lexical analysers.

3.2.3 Statements

Statements are the smallest executable components in C code, and therefore the smallest units that were used in this research. A statement in C can be compared to a sentence in a natural language.

A statement alone does not contain any information about its position, or even purpose in the function that it belongs to. So, for example, a statement that declares a variable *i* of type *int*, and assigns value to it:

```c
int i = 0;
```

It is not possible to say is the declaration in the beginning of the function or it the middle. C23 standard does permit declarations in the middle of the function (International Organization for Standardization, 2023), but Linux kernel prefers the C99 and C90 style, so variables must be declared in the beginning of the function.

By looking at an individual C statement, it is only possible to determine if that statement is grammatically correct, but as compilers and other semantic analysers (such as *sparse*) already detect grammatical mistakes, there is not much point in teaching a model to only do that.

Using statements requires the use of other techniques that give hints about the position and purpose inside the function. One of those techniques is the *Next Sentence Prediction* (Devlin et al., 2018).

Other things to consider with the statements is that statements are hierarchical in C programming language. A statement can consist of multiple child statements, and those child statements may have child statements of their own. A statement that groups multiple statements is called a *Compound Statement*, but they are often referred to as *code blocks* (Kernighan & Ritchie, 1988). The child statement in C compound statements are wrapped in braces (i.e. *{* and textit}) as shown in listing 1.

```
if ( expression ) { child_statement1; child_statement2; ... }
```

Listing 1: Compound Statement in C.

The compound statements complicate the handling of the statements a little bit, because they always need to be split into the child statements separately. A decision must be also made about how to express the compound statement itself after the split. The component statements in C often do have a declarative part that usually contains the expression of the statement, like in the above example: *if (expression) {*. That part can usually be extracted relatively easily from the entire statement, for example by checking where the first child statement begins, but there are exceptions. For example the ***do*** *statement* ***while*** *(expression);* iterative statements (Kernighan & Ritchie, 1988).

Splitting the compound statements is nevertheless possible, but it should be pointed out, that unless separately expressed somehow, the information about the level where the statement is in the hierarchy is lost. The level is important with the stylistic requirements of Linux kernel code, where the level is expressed with tabulator indentation. This is irrelevant in C grammar, but not for the Linux kernel coding style. An example is show in listing 2.

```
/* Wrong. We are now one level in the hierarchy. Tabulator is missing. */
if (i > 0) {
i = 0;
}

/* Correct! The level is now expressed correctly. */
if (i > 0) {
   i = 0;
}
```

Listing 2: Corrected indentation.

A final note about the statements when considering how they could be used with the LLMs is about the preprocessor macros. The C preprocessor macros in their simplest form is just an alias for a constant or a string. In more complex case they define statements and even compound statements. C preprocessor macros can also hide grammatical aspects of C, for example a preprocessor macro may or may not include the expression statement terminating ***;*** character as shown in listing 3.

```
#define EXAMPLE_MACRO1 int i = 0;
#define EXAMPLE_MACRO2 int i = 0
```

Listing 3: Macros may or may not have the ***;*** character at the end of statements.

This makes detecting the statements difficult unless the code is pre-processed with a C preprocessor first. Otherwise, there is simply no clear way to determine where the expression ends. Preprocessing the code can be done to avoid this issue, but that will remove all the stylistic information that Linux kernel has a lot of rules for (“A guide to the Kernel Development Process”, 2024).

This issue can fully be avoided only with semantic analyser that can first pre-process the code to find the expressions and other statements in the code, but without loosing any of the stylistic information.

3.2.4 Functions

A function in C programming language is a named sequence of statements that other parts of the program can call by using the name of the function (Kernighan & Ritchie, 1988). A small program consist of only one function, usually the *main()* function. A function can be looked at as a similar complete unit of text such as a paragraph or a section in a natural language corpus.

Just like with paragraphs and sections in texts, the problem with functions is that a function can contain anything from a single statement to thousands of statements. This forces a decision to be made when considering how functions can be utilised as input for the LLMs. If a function is larger

than the maximum input size of the LLM, the function must be either split into smaller parts, or not handled at all.

Splitting a function will create a situation that is like the problem that exists individual statements are used; the context is lost. If a function is split in half, the second part does not know what the first part contains and vise versa. This issue must be the same when splitting a text such as a paragraph into two, but what is different is the hierarchical information of the statements, which could cause additional confusion for the LLMs. The input may for example end at level that is lower than what it started with, as seen in listing 4.

```
--> First part
int main(coid)
{
   int i;

   for (i = 0; i < 10; i++) {
<-- First part

--> Second part (starts from level 3 and ends at level 1)
       printf("%d\n", i);
   }
}
<-- Second part
```

Listing 4: The problem with the split functions.

This may or may not be a problem, but the split may happen even in the middle of an expression statement. In that case the model can not determine the full content of the first (or last) statement.

To avoid this kind of problems, the splitting can not be done by just cutting fixed sized chunks from the function. Ideally the split would always end in complete statement, so the tool splitting the function would need to be able to identify start and stop of statements in the code.

Only including functions that fit to the limitations of the model avoids any problems that the splitting would cause, but it is not a realistic option for the final task that the model will be intended to perform. If the purpose of the model is to find issues in a function, then it must be able to look at any function, regardless of how big or small it is. This option may still be used when teaching pre-trained base models, but it creates another problem. If the largest functions are excluded, a lot of training data is lost. Using only the small functions may not be enough.

3.2.5 Source Files

A complete C source file is compiled into an object containing the machine language instructions of the architecture used. A source file in Linux kernel usually contains a complete device driver, subsystem drivers such as device class or device bus drivers that create the devices for the device drivers, or in some cases parts of one of those things - some drivers are split into multiple files.

Source files can be looked at very similarly as functions. The same issues apply to them. Some C source files in Linux kernel are smaller than the maximum input size of the model being used, but most files are larger. But if a source file were to be split into even sized segments, the segments would not only cut statements in half, but also the functions.

3.3 Linux Kernel Code Repository

The Linux kernel source code is maintained by Linus Torvalds and others Linux Kernel maintainers in Git repositories (“The Linux Kernel Archives”, 2024). Each of these git repositories has the entire source code tree of Linux kernel and the history of all the changes. Each of those changes, called a commit, originated from a patch that was first send to the mailing list (“Submitting patches: the essential guide to getting your code into the kernel”, 2024).

As explained in section 2.3, the Linux kernel source code in the code repository is constantly evolving. If the source is used as learning data, a decision must be made about which release version should be used. The release may impact a lot of things. On top of the source code itself, also the guidelines evolve in the Linux kernel, so the rules regarding things like the preferred coding style may differ between two release versions.

The Linux kernel version needs to be therefore considered when training an LLM that is designed to understand the Linux kernel source code. The model should be expected to notice the new changes in the coding style. By default, the model should consider them as faults before it has been taught to consider them as the valid style. The model will therefore need to be taught the new guidelines one way or the other. But this may create a problem when training a model with the task of finding issues in the code style.

It is unlikely the kernel source code alone is enough to teach any model to find problems in the source code. The source code can be used to give the model a general level understanding of the

```
commit fac20b9e738523fc884ee3ea5be360a321cd8bad
Author: XXXXX XXXXX <XXXXX@redhat.com>
Date:   Thu Jan 30 21:50:35 2020 +0000

   rxrpc: Fix use-after-free in rxrpc_put_local()

   Fix rxrpc_put_local() to not access local->debug_id after calling
   atomic_dec_return() as, unless that returned n==0, we no longer have the
   right to access the object.

   Fixes: 06d9532fa6b3 ("rxrpc: Fix read-after-free in rxrpc_queue_local()")
   Signed-off-by: XXXXX XXXXX <XXXXX@redhat.com>
```

Listing 5: An example commit with a "Fixes" tag.

Linux kernel source code, but to teach the model to find issues in the code, the model needs to be given examples of bad and good code changes. Those examples must be searched either from the historical commits, or other code patches that have been submitted to the Linux Kernel mailing lists 3.4 over the time. Those changes therefore are not targeted for the latest Linux kernel releases, so they can not be expected to follow the guidelines regarding coding style, or anything else, that are defined for the latest kernel releases.

The Linux kernel code repository does not have many commits that introduce things like styling issues that could be used as example during the teaching of a model, as code style issues can often be eliminated during the normal review process (“A guide to the Kernel Development Process”, 2024). It is more likely that a commit introduces more serious issues, such as memory leak or buffer overflow. When the code style guidelines change, the existing code may be fixed accordingly with separate commits as necessary, but in many cases the new style is only used with the new code changes. There may therefore be different styles being used in the Linux Kernel source code at any given time. That should also be considered.

In the Linux kernel, it is possible to find a problematic commits from the code repository after a fix for it has been added. In the Linux kernel code repository commits that are labelled as *fixes* will contain a special *tag* named “Fixes” (“Submitting patches: the essential guide to getting your code into the kernel”, 2024) as shown in listing 5. The *Fixes* tag contains the commit identifier and subject of the commit that introduced an issue. It should be noted, though, that a commit labelled as a *fix* may be fixing multiple issues and can therefore have multiple *Fixes* tags.

It is possible to build a dataset that contains examples of bad changes by finding all the commits that contain the *tag* “Fixes”. The dataset can have the examples of the commits that introduced the issues as well as the commits that fix them. The problem with the *Fixes* tag is that it is used with serious issues as well as more minor issues, for example code style issues.

3.4 Mailing Lists

The normal review that is done by the Linux kernel maintainers often detects the problems, preventing those problems from ever entering the Linux kernel Source Code Repository. The patch that had the problems is still useful for teaching purposes, to show a model an example of a bad patch. If the patch with the problem is revised, and the problems in it are corrected, then it is likely that the patch will still make it into the mainline Linux kernel Source tree. Only the first version (or first versions) is rejected. That makes it possible to compare the final version to the earlier versions of the patch. All the versions of the patch are needed, and those can be found from the mailing list archives.

On top of the source code repositories, the Linux kernel Organisation hosts and archives also the Linux kernel mailing lists (“List archives on lore.kernel.org”, 2024). The mailing lists are archived in git repositories just like the sources code. In the mailing list archives, each mail that was send to the mailing list is a separate commit in the git repository. There is only a single file in each repository that then ends up being modified in each commit. The file is named *m*.

Because the same file is being modified in the archive, showing any given commit is not useful as such, because it will always show the changes that are made to the previous email in diff format. Luckily the *git* tool allows dumping of the full content of a given file by supplying the file name after colon as shown in listing 6. Since there is only one file that is being modified, this is no problem.

```
git show HEAD:m
```

Listing 6: The full content of the latest email on the list.

3.5 Ethics, Privacy and Legislation related to the use of this data

3.5.1 Guidelines

European Union has defined General Data Protection Regulation (GDPR) and European Commission provides Ethics guidelines for trustworthy AI (“Ethics guidelines for trustworthy AI”, 2019). Those work

as and excellent guidelines in any AI project and must be always followed also in any implementation of this research.

On top of the regulatory guidelines, since the data is code in this project, the license of the code must also be considered. Linux kernel is licensed under GPLv2 license (“GNU General Public License, version 2”, 1991) which puts a lot of restrictions on how the code can be used, and for what purposes.

The Ethics guidelines for trustworthy AI define seven demands for AI related projects. All of them must be considered in the implementations based on this research, but privacy is perhaps the most important. The following items are looked at in more detail: privacy, loss of jobs, and transparency.

3.5.2 Privacy

As all the data in Linux kernel is produced by humans, privacy is the biggest part in the ethical considerations of the project. The question that should be always asked especially when dealing privacy related data is, what data is really needed for the project? In this project, all information related to personal information such as names email addresses etc. - information that is part of each commit message for example - is not being analysed or used in any way and should therefore be always filtered out completely. This step must be done as the very first step when preprocessing the data.

Malicious code injections into Linux kernel is a major issue. Malicious code that is deliberately prepared does not always have the same details that a "normal" faulty code has, and it can be difficult to detect because of that. If the project one day evolves to a point where also malicious code needs to be detected, mere code analysis may not be enough. The implementation will at that point have to look at also the behaviour of the people involved, both the authors of the code as well as the reviewers. The only way this could be achieved would be if all the pre-processed data and any models that were produced out of it were encrypted. That is possible by utilising Privacy Preserving Machine Learning (PPML) techniques such as Homomorphic Encryption (Lee et al., 2021).

3.5.3 Taking Work from Humans

Since one of the main targets of this work is to release human resources that currently must be tied to the code review, it brings an obvious question to mind - Are we going to take somebody’s job?

The goal of the project is primarily to answer to a resource shortage. As explained in section 1.1, the people involved in the Linux kernel code review are mostly doing the review work voluntarily as an extra task that takes time away from the actual work those people are meant to do. It is therefore safe to say, nobody will lose their job because of this project.

3.5.4 Transparency

Information about how and for what purpose the data is used should be always made available to every party involved in any AI project, at least when the data is completely human produced like in this case. This is clearly dictated for example in GDPR. In the case of Linux kernel, which is an open-source project, this is perhaps even more important than usual.

Everybody doing the code review in Linux kernel must not only be informed about the project and the possibility that their behaviour will be monitored, but they must also give their approval for that. If somebody refuses, the subsystem they work on cannot be considered in the project. This aspect is very important in this project. If the Linux kernel community does not approve the project, any recommendations or other output that would be generated would quite likely be ignored.

4 Research

4.1 General

The research was started by looking at the models that the popular ML/DL libraries *Tensorflow* (**tensorflow**), and later *Hugging Face* (“Hugging Face”, 2024), have implementations for. The same models were also referred to in almost all the most resent articles related to LLM published on the popular ML/DL specific websites such as “Towards Data Science”, 2024. The primary database for publications that were used in the study was the *arXiv e-print archive* (“arXiv.org e-print archive”, 2024).

This thesis began with the assumption that traditional Recurrent Neural Network (RNN) based models, especially Long Short-Term Memory (LSTM) (Hochreiter & Schmidhuber, 1997) based models, are still being used in LLM research. It very soon became obvious that after the introduction of the Transformer Architecture (Vaswani et al., 2017), the focus of the LLM related research, and also *code understanding* related research, had almost completely moved to them. This thesis does go over some of the work that was done before the introduction of the Transformer Architecture. On top of RNNs, Convolutional Neural Networks (CNN) have also been used previously in code and Linux Kernel un-

derstanding applications. For example, PatchNet (Hoang et al., 2019) is an CNN based solution to find patches that should be applied to the stable Linux kernel trees. It is still in use.

Nevertheless, the focus of the thesis is in the use of Transformer-based models. The following sections will first go over the high-level architecture of the Transformer model, followed by an overview of the Transformer-based models that were studied and utilised in this work. After that, the thesis will construct a progressive path with a set of goals, each evolving from the previous one. The steps are described as a set of use-cases. To goal of this research is to supply an analysis about how machine learning can be utilised in the Linux kernel development.

4.2 Transformer Based Language Models

4.2.1 Transformer

Transformer (Vaswani et al., 2017) is an architecture where self-attention weights are used in parallel instead of in sequence. That makes them faster to teach when compared to the traditional Recurrent Neural Networks, where also the attention weights, if used, are usually calculated sequentially instead of in parallel. If the weights are processed in parallel, parallel processors such as GPUs can be utilised. After they were introduced, the transformers have surpassed RNNs in language processing, but lately they have been reported to replace also architectures such as CNNs (Radhakrishnan, 2021).

At the core of the transformer architecture is the attention mechanism, which allows every token in the input sequence to interact with every other token in the sequence to find out which tokens it should pay the most attention to. By computing the attention mechanism in parallel, the transforms do not need any sequential components. Like the name of the paper says: *Attention Is All You Need* (Vaswani et al., 2017).

Unlike RNN where the size of the model is usually determined by the number of recurrent units, the transformers tend to be quite large due to their architecture. The components in transformers are made of self-attention layers, feed-forward layers and positional-encodings, which are all stacked in both the encoder and the decoder. The number of parameters in a transformer can therefore be considerable high.

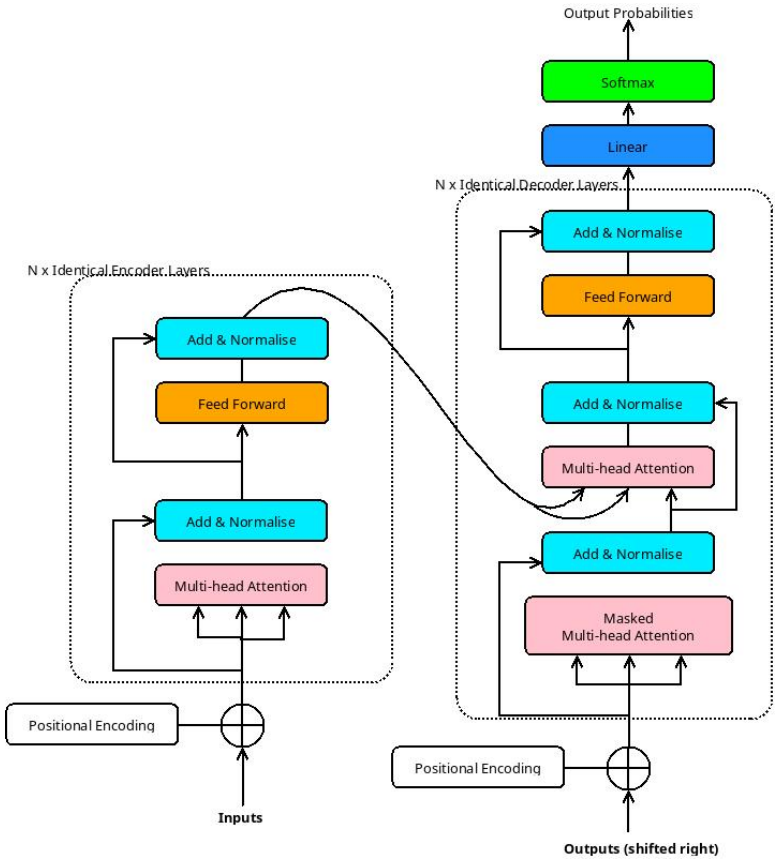

Figure 2: The Transformer Model.

The transformer architecture is at the core of popular LLMs such as Bidirectional Encoder Representations from Transformers (BERT) (Devlin et al., 2018) and Generative Pre-Trained Transformer (GPT) (Radford et al., 2018). The original Transformer model is illustrated in figure 2.

4.2.2 BERT and Masked Language Models

BERT is a common Transformer-based encoder-only model (Devlin et al., 2018). BERT models are trained by looking sequence of tokens bidirectionally (or without direction) instead of from left-to-right or from right-to-left.

Transformer is a sequence-to-sequence model that has both encoder and decoder - encoder that reads the text, and decoder that produces a prediction for a task - BERT only implements the encoder part. This makes BERT useful primarily for task such as categorising, but it can of course be used as the encoder of a model that has also the decoder part.

BERT is based on two separate training strategies, Masked Language Model (MLM) and Next Sentence Prediction (NSP). In Masked Language Modelling, 15% of tokens are replaced with a special *[MASK]*

token. The model then tries to predict the original tokens that are replaced. The concept of MLM is illustrated in listing 7.

```
Banana is a [MASK] —> fruit.
```

Listing 7: The basic idea of MLM.

With NSP, the model was trained by supplying it with two sentences as input. The model was then thought whether the second sentence was subsequent sentence in the original document. After the introduction of the BERT, it was future developed into an entire family of model. One of the most popular models that evolved from BERT is called *RoBERTa: A Robustly Optimized BERT Pre-Training Approach* (Y. Liu et al., 2019). RoBERTta removes NSP completely, relying only on MLM strategy.

Because BERT is encoder only model, it can not be used for generative tasks such as summarisation, but it is very suitable for categorisation. It could work well for tasks such as patch type detection.

4.2.3 T5

Text-to-Text Transfer Transformer, or T5, is a Transformer based model that includes both the encoder and decoder (Raffel et al., 2020). Text-to-text approach means simply getting an input text and training the model to produce target text from it. T5 is designed to perform multiple text-to-text tasks as shown in figure 3. The original model was trained to perform task such as summarisation and translation (from English to Germany).

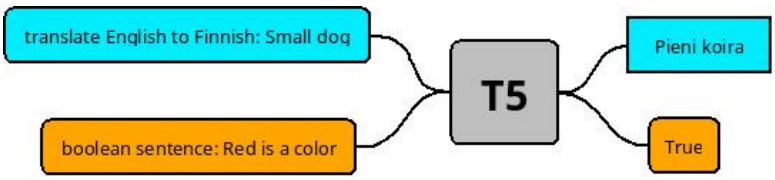

Figure 3: T5 model is designed to perform multiple tasks.

T5 is first trained to a baseline model by using somewhat similar masked language modelling as BERT. T5 baseline model is trained by inputting a sentence that has randomly chosen tokens replaced with unique masking tokens that are referred to as *sentinel tokens* (Raffel et al., 2020, p. 13). The model is then trained to produce a target output that shows the original tokens that were masked with the sentinel tokens, and the tokens that were not masked in the input sentence now replaced with separate unique sentinel tokens (see figure 4). This is referred to as the *Unsupervised Denoising Objective* (Raffel et al., 2020, p. 19).

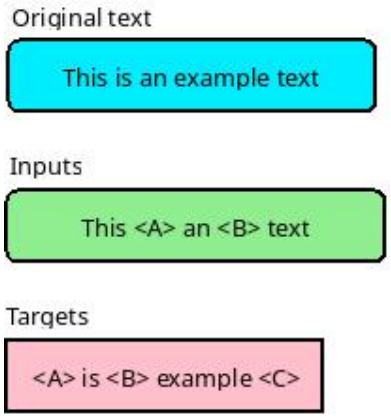

Figure 4: Sentinel tokens.

The goal of the baseline model training is to give the model a generalised knowledge that can then be utilised in downstream tasks.

Because T5 is designed to perform multiple tasks, it seemed ideal for this research. The same model could potentially be used for code quality checks as well as retuning review in form of text (in English). The same model could also even categorise the patch into fix or feature (as an example).

A decision was made very early in this research to use T5 as the primary base model architecture. Because of that, the thesis will go over a few more technical details regarding it.

The T5 encoder is made from several *blocks* that consist of self-attention layer followed by feed-forward network. The decoder is similar except it includes a standard attention mechanism for the encoder output. Otherwise T5 is like the standard Transformer model, where the input sequence of tokens is mapped to a sequence of embeddings which are then forwarded to the encoder. The number of *blocks* determine the size of the model.

The original T5 baseline model (T5base) had 12 encoder blocks and 12 decoder blocks, resulting in approximately 220 million parameters. For regularisation, dropout with probability of 0.1 was used.

4.2.4 BART

BART (Lewis et al., 2019) is another commonly used sequence to sequence model. This research will attempt to use it for comparison. The accuracy and performance of T5 could ideally be compared to BART.

BART is made of bidirectional encoder like BERT (Devlin et al., 2018) and left-to-right decoder like GPT (Radford et al., 2018). BART is effective in text generation, but it is also used for comprehension tasks.

BART is pre-trained by randomly shuffling the order of sentences in the training data text, and then by replacing sections of the text with a single mask token (Lewis et al., 2019, p. 1). The objective of the pre-training is therefore to corrupt the input text and optimizing a reconstruction loss.

When compared to T5, besides the different pre-training objectives, BART uses a little bit different activation function than T5. BART uses a modified ReLU and absolute position embeddings and T5 uses relative position embedding. Nevertheless, both models are sequence-to-sequence models that are suitable for very similar tasks.

The original base BART model was trained with 6 encoder and 6 decoder layers, and a larger version of the model was trained with 12 encoder and 12 decoder layers (Lewis et al., 2019, p. 2).

4.2.5 Language Model Pre-training and Fine-tuning

Pre-training in general refers to the use of a model that was initially trained to perform one task that is like some new target tasks. Pre-training in general is therefore a way to transfer learning. With LLMs the pre-trained models are not necessarily trained to perform any specific task. Instead, they are trained to have a general level of understanding of some specific corpus, or an individual text (Raffel et al., 2020, p. 1). The pre-trained model can then be used to train other models that are fine-tuned for some specific downstream tasks.

Although in the original T5 paper "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer (Raffel et al., 2020), the pre-trained model was not trained for any specific task, the pre-trained T5 models that are supplied as part of some of the commonly used Transformer and ML/DL libraries are already fine-tuned for some tasks. For example, the pre-trained T5 models that are supplied as part the Hugging Face Transformer Library can perform English-to-German Translation and Summarisation (“T5”, 2024).

4.2.6 Sentence Embeddings / Gap Sentence Models

An embedding is a numerical representation of usually a word (token), sentence or text (but other data can also be embedded, for example images), so that the representation contains semantic meaning

of what is being embedded. If we look at the words “dog” and “cat”, they share the fact that they both represent an animal and/or a pet, and this information must be readable from of the numerical representation. The most common way to represent an embedding is in vector space. With vectors we can easily calculate the distance between different embeddings to determine how well their meaning matches each other as illustrated in figure 5. A very common technique to produce vector representation of words (tokens) is called word2vec (Mikolov et al., 2013).

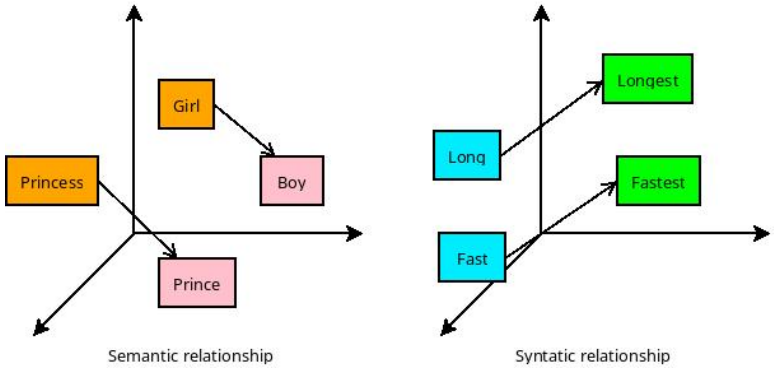

Figure 5: Relationships between words.

Transformers also rely heavily on embeddings to represent data mathematically (Vaswani et al., 2017). Transformer based models usually take word embeddings as input, but as the embeddings can also be for example entire sentences, it is also possibility to create sentence embeddings from the C statements. A statement in C is effectively just a sentence. That would allow the model used to process much larger amount of input data. When the individual tokens are embedded, every token will have a numeric representation.

```
[int, i, =, 0, ;] —> [1, 2, 3, 4, 5]
```

Listing 8: Embeddings contain also information about the distance between the tokens, so they are not mere enumerations like in this example.

When the statement is embedded, the entire statement will have a single numeric representation. Listing 9 shows how an entire function could possibly be represented with the same amount of data as an individual statement. By embedding the statements, the model could possibly process entire C source files as input.

```
int main(void)              -> [1,
{                           ->  2,
        int i = 0;          ->  3,
        printf("%d\n", i);  ->  4,
}                           ->  5]
```

Listing 9: Simplified C statement embeddings.

There are specific models designed for sentence and document embeddings that can be used, for example Doc2vec (Le & Mikolov, 2014) that is based on Word2vec. It is also possible to produce the C statement embeddings by using a specified Transformer.

Once the C statement embeddings have been created, the pre-training model can be trained using specific techniques that have been developed for sentence embeddings. One of those techniques is called Gap Sentence Generation (Zhang et al., 2019). Gap Sentence Generation is a technique that is very similar to the MLM used in BERT (Devlin et al., 2018), except that instead of individual tokens, entire sentences are masked.

4.3 Use Cases

4.3.1 Patch Type Classification

After a mainline Linux kernel is released, the release is considered stable, and it is maintained as separate stable kernel tree until the next mainline Kernel is released. Some stable kernel trees are also designated as long-term maintenance releases. Those are maintained much longer (“Active kernel releases”, 2024).

As explained in section 3.3, all bug fixes from the mainline tree are backported to the stable releases. In most cases the bug fixes have a specific ***Fixes:*** tag, but not all. The PatchNet (Hoang et al., 2019) was designed to identify the fixes that did not have that tag to make sure that also they are applied into the stable kernel trees.

PatchNet is CNN based neural network that attempts to categorise each patch to fix or not fix. So, in practice PatchNet tries to look at the patches as if they were images. PatchNet has reported relatively high accuracy, up to 86% (Hoang et al., 2019, p. 12). Nevertheless, PatchNet still produces a lot of false positives, meaning patches that are not fixes that that PatchNet thinks are fixes. In some cases, those patches have been applied to the stable kernel trees.

One goal for the project will be to see if a Transformer based solution could achieve even higher accuracy than PatchNet. Basic categorisation could possibly be achieved very simply with a BERT model that is fine-tuned for the task. For training data, the *Fixes* tag could be used to label the commits like in listing 10.

```
#/bin/sh
echo commit,fix
git log --no-merges --oneline --pretty=%h|while read l
do
        # Check does the commit contain string "Fixes:".
        if [ -n "$(git show -s $l|grep Fixes:)" ]; then
                echo $l,1
        else
                echo $l,0
        fi
done
```

Listing 10: Shell script that labels the commits that contain the *Fixes* tag with 1 and the others with 0.

The T5 model that is planned for the main task of basic code quality checking in this research could also be used. The benefit from T5 would be that the same model could be trained for both the categorisation task as well as code quality checking task.

4.3.2 Code Quality Understanding

The Linux kernel development-process defines the preferred coding style and many other things that are considered acceptable and not acceptable in the Linux Kernel code ("A guide to the Kernel Development Process", 2024). The development-process gives the rules that a model that can perform the task of code quality checking must understand. Listing 11 shows an example of a code change that introduces a coding style mistake.

```
 int function(void)
 {
+        int i;
+        /* Unnecessary extra newline. */
+
+
        i = 0;
        ...
```

Listing 11: Adding double newlines is a common coding style mistake.

Creating a model that can be used for automated Linux kernel code quality checking is the primary goal in this research. The target is to create a model that first has a basic understanding of Linux kernel code and after that basic understanding of the C code quality in Linux kernel. The model should be able to take a patch as input, detect if the code quality does not meet the guidelines, and then generate proposed improvements if needed.

The root model that has basic level understanding of Linux kernel code does not necessarily need to understand C code quality. It just needs to understand Linux Kernel in general. After the root model

has been trained to understand Linux Kernel Code in general, it can then be fine-tuned to understand also code quality and detect issues with it. So for the root model only the Linux kernel Code is needed, but for the fine-tuned model additional training data that can be used to teach the model about what is good or bad quality code is needed. That additional data can be collected from the mailing lists as explained in section 3.4.

It is also possible to use the Fixes tag as explained in section 3.3, but those patches rarely fix code quality issues. Instead they more often fix more serious issues such as memory leaks and buffer over flows, so by using the Fixes tag, the model would learn about things that are out side the scope of the initial goal. That may not be a problem, but it may complicate the learning process. Later the model must be able to distinguish between a code quality problem and a more serious problem. That is needed at least when the model needs to be able to generate a review in English language on top of the proposed improvements. Because of that, it may be better to separate the task of finding serious issues in the patches from mere code quality issues. That way the first model can be kept simpler.

A model that can detect the quality issues only needs the encoder part from a Transformer based model, so for example BERT can be used for this task (Devlin et al., 2018). For the generative, part where the model needs to first produce proposed improvements, and later review comments in English, decoder is also needed. T5 or BART based model should work well for this task. A T5 based model can also be easily trained for multiple tasks, so for example separating the detection of code quality issues and mode serious issues should not be an issue.

4.3.3 Review Generation

The second goal should be to have a model that can also generate comments together with the proposed fixes to the code. The training data collected from the mailing list archives for the code quality checking task contains also the review comments but using it has a couple of problems. The comments are not always consistent as shown in listing 12. The same problem may have multiple descriptions.

That may not be a problem for the Transformer based models that are trained to understand English language. Bigger issue is that there may not be a clear single comment describing a problem at all. Instead, there may be a long discussion with multiple emails about the problem. Perhaps the biggest problem is that in many cases the comments are hostile in nature (Claburn, 2022) (VK, 2023).

```
> void reserve(void *resource)
> {
        ...
> }
>
> void release(void *resource)
> {
        ...
> }
>
> void my_func(void)
> {
>       void *resource = ...
>
>       reserve(resource)
> }

1. The "resource" is not released.
2. Resource leak.
3. Resource imbalance.
```

Listing 12: Different ways to express the same type of common issue in the review comments.

In practice the review comments collected from the mailing list archives are not usable without checking every single comment. This may be impractical.

Another approach is to start with a model that only generates proposed code fixes for the patches that need improvements, and then organise a community such as the Linux kernel community inside Intel (or any other organisation), or even the public Linux kernel community, to provide "standardised" comments for the common issues. This way the community could agree together what those standard comments could be. That would guarantee that the most common issues would get a similar comment, and more importantly, all the hostility could be ruled out. Because of the fast phase the Linux kernel is developed, possibly on one year would be needed to generate a large enough dataset of review comments. At least with T5 based models the review can be trained afterwards simply as an additional task.

Nevertheless, the focus should initially be on being able to generate the proposed fixes rather than the comments. As the saying goes: "A picture is worth a thousand words". Showing the needed fix as code is enough in most cases - no explanation needed.

4.3.4 Bug Fixing

The Linux kernel gets a constant flow of fixes, and those fixes are not only fixing issues introduced in the recent patches. Some fixes are fixing issues that were introduced years ago, because nobody

hasn’t detected those issues before. The reason for that is in most cases corner cases that requires very specific conditions.

The Linux kernel is no different than all the other large software projects. It is full of existing issues - bugs - that are not yet discovered. It would be very beneficial to be able to find and fix a large portion of those issues, if possible, not only because of the issues themselves, but every time an issue is discovered, it obviously needs to be fixed. That means somebody must put the effort into figuring out how the issues should be fixed, and that is time away from the everything else.

If a model can detect issues in patches, it may be able to find issues that already exist in the Linux kernel source code. This task should not even require any additional training data. This would be simply a specific separate task for the model.

4.3.5 Static Code Analysis

Static Code Analysis means analysis of the code without executing it. Static Code Analysis is the currently used method of finding code quality issues. The target for the initial model in this research is to be able to find similar issues that currently used conventional Static Code Analysing software is designed to find:

- Programming errors
- Coding standard violations
- Undefined values
- Syntax violations
- Security vulnerabilities

Static Code Analysis can be used as an initial goal for the model produced in this study. The model should attempt detect the same issues that are detected by the static code analysers such as the Intel’s Linux kernel Performance/0-Day Test Service (Wu, 2020).

The reports that test services like the 0-Day Test Service produce are send to the same mailing lists as the patches. That means those reports can be used as training data quite easily, but that may not be necessary, because a report like that on a patch means that the patch must be revised in any case (See Figure 1). But the reports may contain for example explanations that could be used when teaching the model generate the review comments.

4.3.6 Understanding complete C source code file such as a device driver

By teaching a model to understand C code at general level, the model can be used for tasks such as finding issues in C statements and functions, but that will not be enough to teach the model to detect issues in C source file structure. So with only general level understanding of C, the model can not determine if a function should be static or not, if a function or structure is missing a prototype, if a function is called before it has been introduced, and so on.

To train just the basic understanding of C code, or Linux kernel specific C code, a model only needs to be able to read the C functions (see section 3.2.4) or possibly even only individual C statements (see section 3.2.3) as input. That should be enough to give the model an understanding of how C programming language should be read at a general level, but that will not be enough to teach the model to understand how an entire C source file should be organised and used. To understand that the model would have to be able to read the entire C source code files as input.

Because of the size of most files in Linux kernel, the input of the model would grow so large that processing it would require massive amounts of memory. That would be impractical, or possibly even impossible at least with the largest files. Either the size of the file would have to be limited, or the larger files would have to be split.

Most likely a better approach would be to use method like Sentence Embedding described in section 4.2.6. That will reduce the input size and it should allow the model to process entire C source files. Instead of each token, each statement, or possibly even each function in the C source code file, will have its own embedding vector.

4.3.7 Understanding of a Complete Linux Kernel Subsystem

The same techniques that can be used to train a model to understand complete C source files may be possible use to train the model to understand even larger parts of the kernel, entire Linux kernel subsystems.

Linux kernel is organised into components called sub-systems. Each basic operating system function, such as memory management, scheduling and interrupt management, has a dedicated sub-system in Linux kernel. In Linux kernel each bus (e.g. PCI, USB, etc.) has its own dedicated sub-system. Linux kernel has also specific device classes that are used to organise devices further. The device classes in

Linux kernel are for example graphics, sound, net, printer, block and so on. Each device class in Linux kernel is also a sub-system (Corbet et al., 2005).

A sub-system in Linux kernel offers the device drivers the fundamental routines that are used to register and un-register the devices, and in case of data buses, usually also routines that are used to communicate with the device. A routine in most cases is a single function coupled with a single data structure. Though, in some cases a routine may require calling of multiple functions, but in those cases, Linux kernel usually offers a *helper function* like the one in listing 13, which is a wrapper function that calls both (all) actual functions in the correct order.

```
1 int device_register(struct device *dev)
2 {
3         device_initialize(dev);
4         return device_add(dev);
5 }
```

Listing 13: Example of an actual helper function in kernel.

It is possible that a model that has been trained with complete C sources files may be able to determine how a sub-system routine should be used, with what parameters, in which order, and so on. Ideally the model would also understand the full potential that a sub-system specific API can offer. So for example, the model should be able to point out that a routine such as device registration should be done by separately initialising and adding the device in some cases instead of always using the helper device_register(). For that the model would most likely need to understand that a helper routine is constructed from other functions that the sub-system API offers, as well as how those functions are normally used in the drivers.

The model could for example be trained so that each device driver that is used as input during training is included also the entire sub-system code. That way the model could possibly get a better idea about all the functions a sub-system has to offer. By using techniques like sentence embedding that should not be a problem.

4.3.8 Understanding the Whole Linux Kernel

Understanding how the Linux kernel subsystems work individually would be quite useful, but the device drivers always use routines that are offered for all C source code in the library code of the Linux kernel. These routines include the functions used to reserve memory, handle strings and so

on. The device drivers quite often also use routines that are offered by some different Linux kernel sub-system. The devices in Linux kernel can be bind to both a bus and a device class.

Usually, a device also depends on resources that the other Linux kernel sub-systems manage. For example, a PCI device, which is registered as part of the PCI bus sub-system, may need handles to the clocks supplied to the PCI device, which are handled by the clock sub-system, or regulators that are handled by the regulator sub-system, and so on.

So, the drivers, and all the other C source code files, have a lot of internal dependencies, and those dependencies are quite often the source of problems in Linux kernel. The most common problem is that a driver that depends on services provided by other sub-systems, is not made to depend on those other sub-systems in the kernel configuration file or the Makefile that is used when the driver is build.

To give a model an idea about the dependencies in Linux kernel, it needs be trained to understand also the kernel configuration files and Makefiles which are not C source. For an idea about the location of the files, the location of the files needs to be supplied somehow as a parameter to the model.

On top of that, all the different sub-systems have their own sub-folders in the source tree. It would be very useful if the model would understand the structure of the Linux kernel source tree. In some cases, it is not clear under which sub-system a new driver should be placed. This is especially a problem for the so-called multifunctional devices. The multifunctional devices are basically components that contain several different kinds of controllers and other devices. There is a dedicated sub-system in the Linux kernel for those called Multi-Functional Device (MFD) Framework, but unfortunately it is not always possible to use it, but when it can be used, the problem can be solved by simply placing the new driver under the MFD framework sub-folder.

There would be many benefits in having a LLM model that knows details about the entire kernel on top of the individual drivers and sub-systems. To train the model to have such awareness of the entire Linux kernel, possible the same techniques used to teach the model deeper understanding of the individual sub-systems could be used. The techniques used in tools such as Doc2Vec can be used to generate embeddings from a complete sub-system, if necessary, even the entire kernel. A customised Transformer based model could also be created for that purpose.

4.4 Challenges

4.4.1 Limitations with Transformer Based Language Models

The Transformer architecture has been very successful. Thirupathi Thangavel describes the architecture as revolution in Natural Language Processing (NLP) in his paper “Limitations of Transformer Architecture” (Thangavel, 2023). Despite of its success, the Transformer architecture does have limitations which are good to keep in mind when using it for any task. Thangavel lists total of 13 limitations in his article, most of which are challenges related to the architecture itself - complexity, large memory requirement, and so on. But one of the limitations he simply describes as “Lack of Common-Sense Reasoning”.

Peng et al. describe the same issue as inability to compute simple semantic operations in the paper “On Limitations of the Transformer Architecture” (Peng et al., 2024). The example in the paper describes the problem as inability to “identify a grandparent of a person in a genealogy”. The paper is a study on a common phenomenon in Transformer architecture called “hallucination”, where the Transformer based model produces inaccurate and inconsistent output, which in the paper is proven to be cause by the same root issue.

The Transformer architecture, while powerful, does not have what Thangavel calls in his article as *true understanding* (Thangavel, 2023). All the patterns the model can produce will be based on the training data and training data alone.

4.4.2 Training and Inference Resource Requirements

The Large Language Models, like the name suggest, are large, very large. To use the T5 as an example, the T5-base model has 220 million parameters with 12-layers, T5-large has 70 million parameters with 24-layers, T5-3B has more than 2,8 billion parameters, and T5-11B 11 billion parameters (Raffel et al., 2020). The number of parameters alone mean that the models require a lot of resources to be processed.

Larger models obviously require more processing power and time when compared to the smaller models, but perhaps a bigger issue is the considerable memory size requirement. With the Transformer-based models the required memory size is quadratic size of the context window - the context window

size being the number of tokens the model can process and generate (N. F. Liu et al., 2023) (Vaswani et al., 2017).

To mitigate the large resource requirements, the precision of the model can be reduced in various ways. The floating-point precision can be reduced from the full 32-bit to half - 16-bit floating-point precision. The modern CPUs and GPUs also support other floating-point format besides the normal single-precision (32-bit) and half-precision (16-bit) floating-point formats. Intel for example has instructions ("Intel Deep Learning Boost New Deep Learning Instruction bfloat16", 2024) for processing data in a floating-point format called *bfloat16* where the exponent part of the floating-point number is increased from the 5 bits used in the normal 16-bit floating-point format to 8 bits which is used in the normal 32-bit floating-point format. The dynamic range of the bfloat16 is therefore the same as it is with the normal 32-bit floating-points, even though it only occupies 16 bits.

The training of the model can also be optimised in many ways. The patch size - the number of simultaneous inputs - can be reduced. That will reduce the memory requirement by extending the training time.

Specific optimised Transformer-based model architecture can also be used for the resource mitigation. The Longformer model (Beltagy et al., 2020) for example introduces an attention mechanism that scales linearly instead of quadratically like in the normal Transformer-based models.

It may also be possible to use Small Language Models (SLM) that have much less parameters and smaller neural networks (the term SLM may also be used when LLM is trained with a small dataset) (Thomas, 2023). The T5-small for example has only 60 million parameters, which is considerably less compared to the 220 million that the t5-base has (Raffel et al., 2020).

The large resource requirement may mean that the model, even when optimised, must be trained with powerful hardware - possibly much more powerful than any desktop computer. Services such as Google Cloud ("Tensor Processing Units", 2024) and "Intel Developer Cloud", 2024, which can supply powerful dedicated hardware for the training, may be required.

4.4.3 Limited amount of training data

The Linux kernel is a large software project consisting of over 60 000 files and 33 million lines of code (as of Linux kernel v6.7), but when compared to the more than 250 billion pages in Colossal Clean

Crawled Corpus that was used to train T5 (Raffel et al., 2020), it is not much. Alone the Linux kernel C code may not be enough to train a completely new Transformer-based model.

One way to increment the amount of C code used to train the model is to use other open-source projects that are programmed using the C programming language. The Mozilla Firefox web browser is mostly programmed in C, but parts of it are programmed using other programming languages (“Firefox”, 2024). The GNOME Desktop Environment is also largely programmed in C, but GNOME is constructed out of multiple components, each having its separate code repository, which may create difficulties when collecting the code (“GNOME”, 2024). Both these projects are open-source projects, and access to the source code is therefore not a problem, but it is also important to note that neither of them are licensed using the GPLv2 licence that the Linux kernel is licensed with. That is important to consider not only from the legal, but also ethical point-of-view. Other large open-source projects of course exist, but it is unlikely that any of them is programmed using similar style as dictated in the Linux kernel coding standard (“A guide to the Kernel Development Process”, 2024). The C code in these projects may look a bit different.

The patches from the mailing-list archives could also be considered. The Linux Kernel source code repository is built from more then one million commits, each commit being originally a patch that was send to the mailing-lists. The problem with the patches is that they are not pure C code. A patch is an output from the *diff* program that compares two separate files, or like in the case of git repositories, two separate versions of the same file. Because of the different format of the data, it may not be easy to use. It is possible that the data collected from the mailing-list archives is more usable when the model is later fine-tuned for the final task of finding issues from the patches, but not for the training of the base model that is pre-trained with the Linux kernel C source code.

There are also open datasets that contain C code. Project CodeNet (Puri et al., 2021) is a large collection of code written in various programming languages. The C code portion contains 50 000 samples. It should be noted that the code in datasets like Project CodeNet are heavily normalised, which could mean they are not very useful for actual use-case. Fox example, only the most common function names and other identifiers are kept in place in the code, and everything else is replaced with token type names. By normalising the code in the dataset, the accuracy of any model that is trained with it can be increased, but at the same time the data looses valuable context, reducing the number of tasks that it could be used for.

The C code samples in the Project CodeNet are also stripped of any code style information, including new lines. A single C code sample in the dataset like the one in listing 14 usually represents a C file that could be compiled into an object file, but all the code is on single line. The code is also pre-tokenised by separating each token with a space. The data is not usable for fine-tuning, but it may still be useful for training the pre-trained base models.

```
# include < id . id > int main ( ) { int id ; scanf ( string , operator id ) ; printf ( string , id operator id ) ; return 0 ; }
```

Listing 14: A random sample from Project CodeNet dataset.

Possibly the easiest way to overcome the lack of training data issue is to use an already pre-trained model as base for the initial model by taking the weights from it. This requires a study of models that have already been pre-trained. Those will be listed in the section 4.5.

4.4.4 Reviews as Training Data are Hostile and also Incoherent

The arrogant style of communicating inside the open-source community - on the mailing lists - is not only a problem in Linux kernel. Many open-source project is affected by this “culture” (Claburn, 2022). In fact, it is not even fare to say that the problem affects only open-source projects.

The toxic style of communicating nevertheless mean that using the review comments is very problematic. The comments can not be simply collected from the mailing lists without a human checking them because of that.

In the section 4.3.3 an alternative approach for collecting the review comments was proposed where the data would be generated separately for new patches with the help of the community instead of using the already made reviews. That would make sure no hostilities would be in the learning data, and it would also make it easier to standardise the explanation of common issues, for example memory leaks.

While that approach is most likely the way the learning data for the review comments is generated and collected, collecting and labelling the old comments may still be useful. By labelling the comments as hostile or not hostile, the data could then be used to teach a model to detect the hostile comments also in other projects.

Collecting and labelling the old comments must be done by humans, so somebody must put the time and effort into it, but it allows the inconsistent information to be easily removed from the data at the

same time. It is quite common that a comment in a reply to a patch does not state that the code change was good or bad, or even comment the code change at all. The comment may be for some related matter, or for example a question. That kind of comments are not useful and should not be included in the learning data.

4.5 Similar Work

After the success of Transformer-based NLP models, the focus on the development of Code Understanding and Generation by Deep Learning has also been in the use of Transformers, but code understanding deep learning models have been developed even before. LSTM was used with models such as DeepFix (R. Gupta et al., 2017) that had the objective of fixing common C language errors, and DeepCodeReviewer (A. Gupta & Sundaresan, 2018) that was trained to generate code reviews. The LSTM based models usually took advantage of techniques such as word2vec (Mikolov et al., 2013) to learn in unsupervised fashion. Static code with deep learning analysis has also been studied. Bielik et al., 2016 created an approach to learn the rules of static code analysis.

The Transformer-based models that are trained for Code Understanding and Generation in most cases consider code as token sequences, but there are also models that are trained with data that contains information about the structure and dataflow of the code. SynCoBERT (X. Wang et al., 2021) for example was trained to predict the edges in the AST and the next identifiers.

After the introduction of BERT (Devlin et al., 2018), several models were pre-trained with it for the purpose of Code Understanding. CodeBERT (Feng et al., 2020) is trained to understand both programming languages and natural languages. It is designed to perform tasks such as code search and code documentation. CuBERT (Kanade et al., 2020) is a similar model, but it with the difference that it is trained to understand a mix of code and natural language (for example, a patch with review comments). Some of the BERT based models for code understanding and generation have a separate decoder part for the code and natural language generation, BERT being encoder only.

From the T5 (Raffel et al., 2020) based models for code understanding and generation, CodeT5 (Y. Wang et al., 2021) is one of the most common ones. CodeT5 is trained to perform three tasks.

- Text-to-code generation.
- Code autocompletion.

- Code summarization.

The CodeT5 model is useful on its own, but perhaps more importantly, it has been used as a base for other pre-trained and fine-tuned models. The developers of the CodeT5 have published a family of models based on CodeT5 called CodeT5+ (Y. Wang et al., 2023). The CodeReviewer (Li et al., 2022), which is pre-trained directly from CodeT5, is fine-tuned to perform three tasks simultaneously.

- Quality estimation.
- Review generation.
- Code refinement (improvement proposal).

The CodeReviewer therefore has very similar goals as this research. The CodeReviewer model has been trained to understand the grammar of nine of the most popular programming languages which include C.

The models listed above are all openly available as open-source implementations, but there are also commercial solutions and subscription only solutions that are sometimes based on the publicly available modes but may also be based on models that have not been made publicly available at all. These solutions usually rely on the largest models - with most parameters and layers - and they may be trained with massive datasets that may or may not be publicly available, or the datasets may have been collected with methods that are not published from publicly available sources. The GitHub CoPilot (“Copilot”, 2021) is a subscription service that uses the OpenAI Codex model (“OpenAI Codex”, 2021) which is fine-tuned from GPT-3, the third version of the GPT model (Radford et al., 2018), for Code Understanding and Generation tasks.

Many of the models that are trained for Code Understanding and Generation have been trained to understand also C programming language, but most of them are designed primarily to understand Python programming language. None of the Transformer based models have been trained to understand the Linux kernel code. Besides the PatchNet (Hoang et al., 2019), there are no models in general that have been trained to understand the Linux kernel code that were found during this research.

5 Implementation

5.1 Proof-of-Concept

The purpose of the Proof-of-Concept implementation is to demonstrate how the research problem can be mitigated with the existing techniques that were identified during the research.

Some of the existing models that were evaluated in this research appeared to be usable almost directly as a solution for the initial goal- to identify patches with issues and generate proposed improvements. The CodeReviewer (Li et al., 2022) especially was designed for those tasks, and on top of that, also to generate the review comments.

But the CodeReviewer is trained to process only patches instead of plain programming languages. It may therefore not be usable even as a base for some of the other goals listed in section 4.3. It was also trained to process multiple programming languages instead of only C, and it was also not trained to understand any Linux kernel specific details, for example syntax. The maximum input size of the CodeReviewer is 512 tokens which means that larger patches must be split using some logic, but the scripts that prepared the training data for the CodeReviewer seem to simply cut the patches so that the patch always fits into the 512 boundaries instead of attempting to slit the large patches into chunks. So, the model was never though to understand anything about a trailing part of a patch that is fed to the model as input separately.

It was obvious that even if the CodeReviewer was used, it had to be fine-tuned for each task. Instead of attempting to do that in this research, a set of completely new models were trained. The CodeReviewer model architecture and the weights were used, but only in one of the models. This approach provided several benefits.

- Guarantee that the pre-model understands Linux kernel style C code.
- Various T5 based models could be used as base for the model.
- The pre-trained models could be based on BART as well as T5.

By making sure that the pre-trained models are trained to understand Linux Kernel code, they can be fine-tuned for many of the other tasks that the goals in section 4.3 define. Ideally also the different pre-trained models can be fine-tuned for the same tasks easily. That way it becomes possible to compare for example BART based models and T5 based models.

But first the data must be collected and processed. The following sections will explain how that was done and with what tools in this research, followed by description of the training processes.

5.2 The Data Sources

The initial pre-trained model will only need to understand C programming language that is written in the Linux kernel coding style. The model does not need to understand any specific details about the code, like the location of the file, or even what is the exact purpose of the code in Linux kernel, just like the models pre-trained to understand natural languages don't need to understand from which book or article the text they are processing as input is coming from or what that book or article is about. These pre-trained models only need to understand the language. For this purpose, the Linux kernel code should ideally be sufficient as training data.

```
git clone git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

Listing 15: The Linux kernel Code Repository can be cloned with a single command.

The fine-tuned model, that can identify the bad patches and propose the improvements, needs to be trained with additional data from the mailing list archives (see section 3.4). The archives are also git repositories, so for each Linux kernel subsystem that has its own mailing list has also a separate mail archive repository. The archive repository size is limited to 1GB, so a large mailing lists will be archived in multiple repositories ("List archives on lore.kernel.org", 2024). In this research, the primary LKML archive was used. The LKML archive is made from 15 separate repositories. Those were merged into one with the script in listing 16 so that the dataset generation would be easier and faster. Otherwise, all 15 repositories would need to be always checked separately.

```
1 git clone http://lore.kernel.org/lkml/0 lkml
2 cd lkml
3 for i in $(seq 1 14); do
4     git pull http://lore.kernel.org/lkml/$i    \
5         --allow-unrelated-histories            \
6         -X theirs --no-ff
7 done
```

Listing 16: Shell script that merges all the LKML repositories into a single repository.

After the repositories are cloned, the data can be collected and pre-processed into a dataset suitable for training the models.

5.3 The Linux kernel Functions into a Dataset

The Linux kernel C source code could be now used directly by tokenising every file, and then splitting the tokenised data into small enough parts so the models can read them as input - in case of CodeReviewer it is 512 tokens. But as explained in section 3.2, the data would then include also comments and the C pre-processor macros.

If everything from the C source code files is included, also the English language comments, the tokeniser generates larger vocabulary. Using a big vocabulary would mean that the LLMneeds a massive embedding matrix as both input and output layers that would require unprecedented amount of memory to process. Because of that, most Transformer-based models have limited the vocabulary size to approximately 50000. For example, the Hugging Face implementation of T5 (“T5”, 2024) has vocabulary size limited to 50000. This creates another problem.

If the total vocabulary size is considerable larger than the one that the model uses, the model will see only the most common tokens, and the rest, the ones that don’t fit into the 50000 token boundary, are replaced a specific out-of-vocabulary token. That will impact the accuracy of the model. Because of this problem, tricks like normalisation are used in models like CodeNet (Puri et al., 2021), but in this research a decision was made to reduce the total vocabulary size by removing the comments from the code instead. This was done because the initial task for a model in this research is to be able to process code without comments. The comments can be processed separately if needed as the following step, possibly even with a dedicated model. Of course, keeping in mind that if they are processed separately, the full context is lost - meaning, how a comment relates to the actual code. So a dedicated model for the comments alone may not be useful for other things besides checking the English language grammar, or for example translations.

In order to remove the comments from the code, the code has to be parsed first with a tool that can separate the comments and the code, for example with a compiler such as *gcc* (“GNU project C and C++ compiler”, 2024) or *Clang* (“Clang: a C language family frontend for LLVM”, 2024), or a dedicated parser such as *Sparse* (“Sparse”, 2024). The Clang provides a C programming interface called LibClang (“LibClang”, 2024). There are also bindings for Python programming language. Because of the programming interface, Clang was used in this research.

The Clang considers comments in the code a special type of token - a single comment is a single token of type *comment* regardless of how many words, or even how many lines the comment has. Because of that, the easiest way to use it to remove the comment is to walk through every token in the code starting from the beginning, and then simply drop the tokens that are of type *comment*. The only problem is that LibClang does not allow access to its own handle to the source file after parsing it, so everything that's not a token (so white spaces, tabs and newlines) is lost. The application needs to therefore open a separate handle to the source code file, but that is not a big issue.

Every token object in LibClang has all the necessary information, including the token type and the start and end offsets inside the source file are needed, which are the only details needed for in this implementation.

The application needs to only execute the following steps.

1. Parse the source code file - this will automatically tokenise it.
2. Open a handle to the source file.
3. Loop through the tokens.
    (a) If the token is comment, ignore it.
    (b) If the token is not a comment, read its start end end offsets.
    (c) Read the bytes from the file handle using the offsets.
    (d) Read everything between tokens.

To reduce the vocabulary size even further, the C source file can be pre-processed so that all pre-processing macros are replaced with the final C code. But that will also include all the code from the header files which may not be desirable, as the code in the header files will always be the same.

A better way to remove most of the pre-processing code from the C source code files is to only read complete functions and exclude everything else. The pre-processing macros are mostly defined outside of the function scopes. Some macros may still be defined inside the functions, but it is rare, and should not affect the final size of the vocabulary considerable.

Collecting all the individual functions from the C source code files can be done by walking through the root elements - called cursors in Clang - and identifying the elements that are of type *function*. After that, the comments can be easily removed from each function by using the procedure described

above. This way all the C functions in the Linux kernel can be collected relatively easily into a dataset where each function is a sample. A complete Python script that collects the functions and removes the comments from the code can be seen in Appendix 2. The script generates a Comma Separated File (CSV), where each row is a function. The file name and the line where the function is implemented are included as separate columns.

5.4 The Mailing Lists into a Dataset

5.4.1 Data for Categorisation

The Linux kernel code review happens on the public mailing lists as described in section 2.2. The proposed code changes are sent as patches to the mailing lists, and if a patch has problems, the developer will fix those problems, and then send a new version of the patch. There are several ways how all that information could be used for LLM training purposes.

The patches on the mailing lists that have multiple versions make it possible to categorise the patches into good and bad patches - the bad patches are the first versions of the patches that had to be revised, and the good ones are the patches that are accepted and applied to the mainline Linux kernel code repository. The good patches are in practice always the last version of patches that have multiple versions. A good patch may also have only a single version on the mailing list which is applied to the kernel directly.

A patch that has only a single version, that has been rejected, may have been programmed in a way that it does not meet the acceptance criteria of the Linux Kernel review process (“A guide to the Kernel Development Process”, 2024), and could be considered as a *bad quality patch*. But it may have also been rejected from some completely different reason. A common reason to reject a patch is, if the patch introduces an interface that nobody uses - the patch may not have any technical or stylistic problems, but if there are no users, it can not be accepted. Therefore, a patch with a single version that is not accepted can not be simply labelled as *bad*. Those patches would need to be checked more carefully, or not include into the dataset at all.

5.4.2 Data for Proposed Improvement Generation

Since the target task is to be able to also generate the proposed improvements for the patches that have problems on top of being able to detect the ones that have problems in the first place, the

model needs to be trained with actual examples of the improvements. Those need to be extracted by comparing the bad versions to the final good version patch somehow.

Comparing the bad and good versions of the patches can be also done in number of different ways. The simplest approach would be to compare the first versions of the patch to the final good version of the patch directly with a data comparison tool, for example the *diff utility*. But the default format that the *git-diff* ("Git - git-diff documentation", 2024) produces, does not show the complete context of a function - only a few lines right next to the change are included. The git-diff tool can show the full context of the function, but only if it is told to do so with a specific *–function-context* option. Without being able to see the complete function that is being changed, it is very difficult to understand what exactly the change is about. To compare the situation to the natural languages, a patch would change a few words in the middle of a sentence, but without showing the beginning and the end of the sentence.

A better approach is to apply the bad version to the Linux kernel git repository, and then compare it to the good final version by hand. That way the full context of the function that is being change can always be included into output of the tool. But this approach allows even more flexibility. This approach makes it also possible to collect the actual C code of the function instead of just the diff output.

A bad version of a patch can not be applied anywhere in the Linux kernel repository. First the location of the final *good* version of the patch needs to be in the Linux kernel git repository. Then the HEAD of the repository needs to move one commit earlier than the final version of the patch. After that the tree is in the same situation as right before the good final version was added. Now the bad version can be applied.

First all the commits from both the Linux kernel git repository and the LKML git repository are indexed as shown in listing 17. The kernel version used is now chosen to be v6.7.

```
cd \$kernel_root
git log --no-merges --oneline > /tmp/kernel.commits
cd \$lkml
git log --no-merges --oneline > /tmp/lkml.commits
```

Listing 17: Building the commit index.

The patch subject is the commit subject, so by looping through the kernel commits, it is possible to use the subject of the commit to find all versions of the patches from the LKML repository if those patches were sent for review to LKML. To demonstrate how the process works picking a randomly selected sample commit in listing 18. After selecting the random sample, finding all the emails that match the subject from the LKML index in listing 19.

```
shuf -n 1 /tmp/kernel.commits
f49449fbc21e usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
```

Listing 18: For demonstration purposes, selecting a random commit.

```
grep 'usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach' /tmp/lkml.commits
75bc4907f074 Re: [PATCH v4] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
a51fca7739ae Re: [PATCH v4] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
43a9f9dd5650 Re: [PATCH v4] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
91f150ab8f30 [PATCH v4] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
25a7cc2dc960 [PATCH v3] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
19dab1e6c88d Re: [PATCH v2] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
bf1c7904c7f3 [PATCH v2] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
3c0cfb1b0ef7 [PATCH] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
33e445aa06df Re: [PATCH] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
adefcf42ae0e [PATCH] usb: gadget: u_ether: Replace netif_stop_queue with netif_device_detach
```

Listing 19: Finding all the emails matching the commit subject.

The above listing includes not only all the different versions of the patch, but also all the review replies that took place on the LKML. The commits are in chronological order, so the first mail (the bottom line in the listing) is the first version of the patch. It, as well as any other version of the patch, can now be picked from the LKML repository using the commit SHA1 hash 12-byte identifier, and then applied to the kernel repository. All the steps are shown in listing 20.

```
cd $lkml
git show adefcf42ae0e:m > /tmp/first.patch
cd $kernel
git checkout -b good 91f150ab8f30 # The randomly selected commit.
git checkout -b bad 91f150ab8f30^ # The commit before the randomly selected commit.
git am /tmp/first.patch           # Now apply the first version of the patch (to branch "bad")
```

Listing 20: Creating a branch *good* for the final version of the patch, and branch *bad* with the first version of the patch.

Now the kernel repository has two branches named *good*, which points to the final good patch, and *bad*, which has the first version of the patch. Now the comparison of those two separate versions of the patch is easy with tools such as *git-diff* (“Git - git-diff documentation”, 2024). In the above example, only the first version is used, but the same can of course be done with all versions of the patch.

By using the above steps, a dataset can be created by walking through the commits in the Linux kernel repository. The dataset that was collected in this research was a CSV table where each sample contained the following columns:

- commit - The commit SHA1 hash identifier.
- function - The function code from the first version of the patch.
- diff - Output from command ***git diff good bad***.
- out - The function code from the last versions of the patch.

The dataset was prepared especially for a model that can take the *function* input and use the *diff* as the target output. As an alternative, the *out* could also be used instead of *diff*.

The generic currency sing (¤) was used as the separator in the CSV, because it is not used in any of the files in the Linux kernel. That allows the table to be created without any additional checks and conversions that would be otherwise needed to prevent the table from getting corrupted. If for example the default comma character was used as the separator, all the commas in the code would have to be marked as special characters or replaced with some special token.

5.5 Data Analysis

The final dataset was collected with the assumption that if a patch has multiple versions, the last version is always better than the first, but this assumption did not turn out to be completely true.

To ease the analysis, the dataset size was reduced by only considering commits that were added after 2009, so only patches between 2010 and 2023 were processed. The largest functions were also discarded. After the reduction, the dataset had 43000 samples. From the 43000, 200 randomly selected samples were used for audit.

By looking at the randomly selected samples, it was clear that considerable amount of patch revisions ends up improving driver and hardware specific details, for example programming the registers of a device in different order. It was expected, but that kind of information is not useful for training, as it does not contain any generic code quality improvements or bug fixes, and they would require understanding of the underlying hardware rather than C programming language.

Another surprise was that in many cases the patch revision reduced code quality rather than improve it. The problem seemed to be common with revisions where the hardware programming was altered. In those cases, it is more important for the reviewers to confirm that the hardware is programmed correctly in the patch rather than confirming that the overall code quality meets the Linux kernel standards. The issues were related to the code style. No serious issues such as memory leak or buffer overflow were introduced in the samples that were reviewed, however, those can not be completely ruled out.

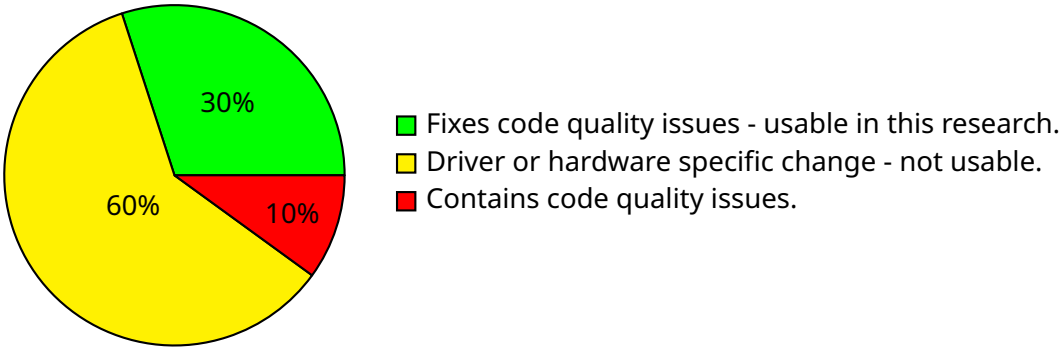

Figure 6: Linux kernel patch revision usability.

Based on the analysis of the 200 samples, approximated two thirds of the dataset is not usable as training data for a model that is meant to detect code quality issues and other generic bugs. The remaining 1/3 of the samples that are useful must be manually selected and moved to a separate dataset. That was not done during this study. The distribution of the samples is visualised in the figure 6.

5.6 The Models

5.6.1 Baseline for the Models

The CodeReviewer (Li et al., 2022) was trained by using the CodeT5 (Y. Wang et al., 2021) as baseline - the architecture and parameters are copied from CodeT5 - to understand code changes in the *diff* format (Li et al., 2022, p 5). The primary goal for the model in this study is to understand C programming language written in the style and structure that is defined for the Linux kernel (“A guide to the Kernel Development Process”, 2024). Because of that small difference, a separate baseline model was trained to understand the Linux kernel explicitly.

Two separate models were pre-prepared. A pre-trained RoBERTa (Devlin et al., 2018) encoder only model that could be fine-tuned for the classification task, and a T5 (Raffel et al., 2020) based model that can also generate the proposed improvement.

5.6.2 Data Preprocessing

After the dataset was collected with the script in appendix 2, the data was tokenised with RoBERTa (Y. Liu et al., 2019) tokeniser. The RoBERTa tokeniser relies on the Byte-Pair Encoding (Sennrich et al., 2015). The RoBERTa tokeniser was used instead of the T5 tokeniser mainly because both the CodeReviewer and CodeT5 use it.

The dataset was then split into training and testing subsets, and finally regrouped into 512-byte samples so they do not exceed the maximum input size of the models. An example script that utilises the Hugging Face Transformer Library (“Hugging Face”, 2024) is available in appendix 3.

5.6.3 Pre-Training RoBERTa Model

After the data is pre-processed, it still needs to be masked as explained in section 4.2.2 in order to be used for MLM training. Luckily the Transformer library from Hugging Face (“Hugging Face”, 2024) offers an existing **Data Collator** class called DataCollatorForLanguageModeling that is designed especially for generation of the random masks for data augmentation automatically (“Data Collator”, 2024). The final RoBERTa model in listing 21 is trained as a TensofFlow model.

5.6.4 Pre-Training T5 Model

The model in listing 22 is build by using the CodeReviewer (Li et al., 2022) base model as the checkpoint. That will make the model use the CodeT5 architecture, which is used in CodeReviewer, and initialise the parameters from the CodeReviewer model parameter values. This approach mimics the way the CodeReviewer was prepared from CodeT5. The model is then trained 10 epochs, which is the same as the CodeReviewer was trained.

The Hugging Face Transformer library does not have a data collator class designed for the generation of the random sentinel tokens that are needed for unsupervised de-noising pre-training of a T5 model, but the Hugging Face community does offer an example how to implement one (“Language model training examples”, 2024). The example is designed to be used with Flax ML/DL library (“Flax”, 2024).

```python
import pandas as pd
import tensorflow as tf

from datasets import load_from_disk

from transformers import DataCollatorForLanguageModeling
from transformers import TFAutoModelForMaskedLM
from transformers import AdamWeightDecay
from transformers import AutoTokenizer

# The same as with tokenizer.
checkpoint = "distilroberta-base"
epochs = 3

# The tokenizer and the dataset need to be ready at this point.
tokenizer = AutoTokenizer.from_pretrained("./tokenizer")
dataset = load_from_disk("./dataset")

data_collator = DataCollatorForLanguageModeling(tokenizer = tokenizer,
                                                mlm_probability = 0.15,
                                                return_tensors = "tf")

optimizer = AdamWeightDecay(learning_rate = 2e-5, weight_decay_rate = 0.01)

model = TFAutoModelForMaskedLM.from_pretrained(checkpoint)

tf_train_set = model.prepare_tf_dataset(
    dataset["train"],
    shuffle = True,
    batch_size = 16,
    collate_fn = data_collator,
)

tf_test_set = model.prepare_tf_dataset(
    dataset["test"],
    shuffle = False,
    batch_size = 16,
    collate_fn = data_collator,
)

model.compile(optimizer = optimizer)
model.fit(x = tf_train_set, validation_data = tf_test_set, epochs = epochs)
model.save_pretrained("./model")
```

Listing 21: RoBERT-based pre-training model.

```
checkpoint = "Salesforce/codet5p-220m"
epochs = 10

# The tokenizer and the dataset need to be ready at this point.
tokenizer = AutoTokenizer.from_pretrained("./tokeniser")
dataset = load_from_disk("./dataset")

block_size = 569
target_length = 114

model = TFT5ForConditionalGeneration.from_pretrained(checkpoint, from_pt = True)

config = T5Config.from_pretrained(checkpoint, vocab_size = 52000)
model = TFT5ForConditionalGeneration(config)

data_collator = DataCollatorForT5MLM(tokenizer = tokenizer,
                                     noise_density = 0.15,
                                     mean_noise_span_length = 3.0,
                                     input_length = 512,
                                     target_length = 114,
                                     pad_token_id = model.config.pad_token_id,
                                     decoder_start_token_id = model.config.decoder_start_token_id)

tf_train_set = model.prepare_tf_dataset(
    dataset["train"],
    shuffle = True,
    batch_size = 2,
    collate_fn = data_collator,
)
tf_test_set = model.prepare_tf_dataset(
    dataset["test"],
    shuffle = False,
    batch_size = 2,
    collate_fn = data_collator,
)

optimizer = AdamWeightDecay(learning_rate = 2e-5, weight_decay_rate = 0.01)
model.compile(optimizer = optimizer)
model.build()

model.fit(tf_train_set, validation_data = tf_test_set, epochs = epochs)
model.save_pretrained("./model")
```

Listing 22: T5-based pre-training model.

A modified version of the class that can be used with TensorFlow models is in appendix 4. By using this data collator, the final T5 model can be prepared very much the same way as the RoBERTa model.

5.6.5 Fine-Tuned Models

While the pre-trained models were trained using unsupervised denoising training objective, the fine-tuned model is trained with supervised training objective where the model is input sequence of data that it then uses to generate output sequence of data - i.e. it is sequence-to-sequence model. By using the Hugging Face Transformer library, the fine-tuned models can be made much the same ways as the pre-training models.

The dataset that was collected from the LKML is without any filtering. The *function* column is used as the input, and first the *diff* column, that contains the changes in diff format, is used as the target output. The dataset is read from the CSV in listing 23 using the Pandas library. The dataset is first converted to Hugging Face Dataset, and then split to training and testing subsets. The data is then tokenised by using the same tokeniser that was trained for the pre-training model.

```
import pandas as pd
from datasets import Dataset
from transformers import AutoTokenizer

df = pd.read_csv("rawdata.csv")
dataset = Dataset.from_pandas(df)

# Split the dataset (80% train, 20% test).
dataset = dataset.train_test_split(test_size = 0.2)

tokenizer = AutoTokenizer.from_pretrained("./model")

def tokenise(data):
    return tokenizer(data["function"],
                     text_target = data["function"],
                     max_length = 512,
                     truncation = True)

tokens = dataset.map(tokenise, batched = True,
                     remove_columns = dataset["train"].column_names)
```

Listing 23: Data preparation for the fine-tuned model.

The Hugging Face Transformer Library contains a class for the data collator designed for sequence-to-sequence processing DataCollatorForSeq2Seq. It will build the batches from the input and output data, and then dynamically pad it. So it will not do anything extra, like generate the data masks that were needed for the unsupervised training.

To model is defined using the TFAutoModelForSeq2SeqLM class from the HuggingFace Transformer library. The previously pre-trained T5 based model can be supplied using the *from_pretrained* method that all the Auto Classes in the Hugging Face Transformer library have. The Auto Classes can guess the actual pre-trained model type (T5, BERT, BART, etc.), so the same fine-tune model definition (the script) should be usable with all of them. The Auto Classes were used also with the tokenisers. The complete model in listing 24 has the target task of producing code improvement proposals is defined.

```
import tensorflow as tf
from transformers import TFAutoModelForSeq2SeqLM
from transformers import DataCollatorForSeq2Seq

data_collator = DataCollatorForSeq2Seq(tokenizer, model = model, return_tensors = "tf")

train_dataset = tokens["train"].to_tf_dataset(
    batch_size = 1,
    columns = ["input_ids", "attention_mask", "labels"],
    shuffle = True,
    collate_fn = data_collator,
)
test_dataset = tokens["test"].to_tf_dataset(
    batch_size = 1,
    columns = ["input_ids", "attention_mask", "labels"],
    shuffle = False,
    collate_fn = data_collator,
)

optimizer = tf.keras.optimizers.Adam(learning_rate = 2e-5)
model.compile(optimizer = optimizer)
print(model.summary())

model.fit(train_dataset, validation_data = test_dataset, epochs = 20)
model.save_pretrained("fine_tuned_model")
```

Listing 24: The Fine-Tuned T5-based Model.

5.6.6 Training and Results

The pre-trained T5 based model was trained on a single GPU with 8GB of memory. The lack of memory meant that the base model size would have to limited to T5base, and the batch size had to be also limited to 2. The training with 10 epochs took 4 days, which is not bad for an LLM. The loss curve is visualised in figure 7.

The focus of the evaluation was meant to be on the fine-tuned models. The pre-trained models were tested manually with scripts like in listing 25 to confirm that they produce reasonable loss also with random inputs.

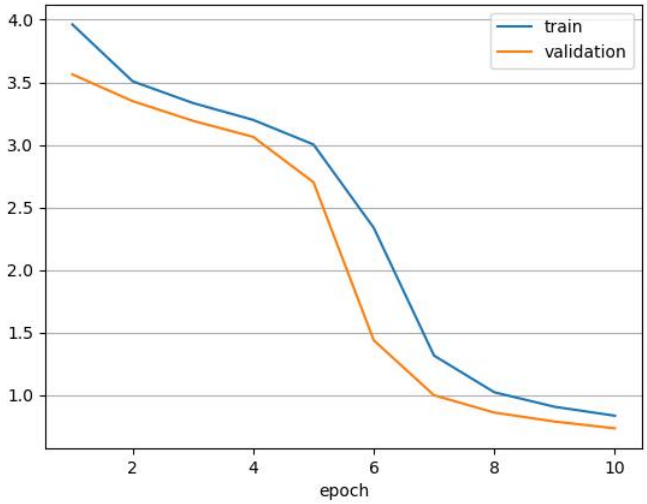

Figure 7: Loss

```python
from transformers import T5Tokenizer
from transformers import T5ForConditionalGeneration

tokeniser = T5Tokenizer.from_pretrained("./model")
model = T5ForConditionalGeneration.from_pretrained("./model")

input_ids = tokeniser("for <id0> i <id1> 0 <id2> i <id3> 10 <id4> i <id5>)",
                      return_tensors="tf").input_ids
labels = tokeniser("<id0> ( <id1> = <id2> ; <id3> < <id4> ; <id5> ++ <id6>",
                   return_tensors="tf").input_ids

print(model(input_ids = input_ids, labels = labels).loss)
```

Listing 25: Python testing script for the pre-trained T5-based model.

The evaluation of the fine-tuned models was going to be done using the ROUGE metric (Lin, 2004), and if possible, also with the BLEU algorithm (Papineni et al., 2001) that was used to evaluate both CodeT5 (Y. Wang et al., 2021) and the CodeReviewer (Li et al., 2022).

However, during training of the fine-tuned model, the loss curve did not show any movement whatsoever. The main problem was most likely the bad quality of the unfiltered dataset that was used as the training data. The data preprocessing could also be improved for example by taking advantage of sliding window techniques, where the next sample will have the end part from the previous sample as the beginning part, that would allow inclusion of also larger functions into the dataset.

Nevertheless, the conclusion is that the dataset must be filtered for it to be usable for training purposes. The fine-tuned model can not be trained with it as it is now.

5.7 Next Steps

The first thing to do is to filter the already collected dataset and improve the preprocessing. The dataset should be increased in size. This could be done by utilising the *Fixes* tags as shown in section 4.3.1, or by organising the people that review the code to *mark* the revision improvements most suitable for the dataset in the same way common responses for common issues could be produced in section 4.3.3.

Once the first model that can produce high quality improvement proposals, the focus must be on the next goals described in the section 4.3. To increase the understanding of the model, techniques that were studied in this research could be quickly taken into used.

A quick study was also conducted on how embeddings from the C statements could be produced was conducted by using the Doc2Vec (Le & Mikolov, 2014). The script where it was experimented is in appendix 5. The embeddings produced by Doc2Vec could be very useful, but the problem with Gensim Doc2Vec implementation is that it does not appear to be able utilise accelerators for parallel processing such as GPUs. It means producing the embeddings with it is always a bit slow.

An alternative approach is to use a Transformer-based model to produce the embeddings. The CodeT5+ (Y. Wang et al., 2023) model collection contains a model designed to produce the embeddings called *CodeT5+ 110M embedding model*.

6 Conclusions

This study surveyed several DL techniques and models that could be used to aid the Linux kernel development process. The focus was especially on the Code Review Process of the Linux kernel and in the mitigation of the problem caused by the lack of human resources available for the review. The research done during this study constructed a set of steps, each evolving from the previous one, that can be used as the goals for the utilisation of AI in the Linux Kernel Development Process.

The goal of the study was also to achieve the first of those steps, which was defined as ability to detect code style issues and other common quality issues in the patches that are send for review. For this part of the study a dataset was collected from previous patch reviews that were conducted on the public Linux Kernel Mailing List. This data revealed that a revision does not always improve the quality

of the code. Especially code style issues were introduced to the final versions of the patches in some cases. The improvements were also primary so hardware and driver specific, that they would not be suitable for training of a DL model that is meant to find generic issues in the code. To demonstrate the problem with the data and attempt to fine-tune a model that could be used for the initial task by using the complete dataset was executed. The performance of the model was so low that it was not usable for the task. The dataset that was constructed would need to be filtered before it can be used as training data.

The legislation and privacy related to the use of this data was considered very carefully in this study. The Linux kernel is licensed using the GPLv2 licence, which is very restrictive licence. The research viewed several other projects that could be used as additional data sources, but due to the nature of the licence, using code from those projects together with the code from the Linux Kernel may not be possible. Due to the restrictive licence, a base model was pre-trained using only the source code from the Linux kernel. That model performed so well that there is likely no need for additional data sources. The licence and the ethical guidelines require that the use of the data requires a permission from the people who have produced it. This means that the Linux Kernel Development Community must be involved by the time the final applications that utilise the models trained with the data. The use of those models in other software projects besides the Linux kernel may not be possible.

Besides the findings in the quality and the general use of the Linux kernel Source Code as data, the modern DL techniques, algorithms, and especially the Transformer-based models, can be concluded to be very suitable for aiding also the development of the Linux kernel. The models are not going to be usable for tasks such as Code Generation for a complex software such as an Operating System Kernel, due to the lack of the *true understanding* that in practice the task would require. But for the task of finding code quality issues, first at the level of coding style issues, and gradually being able to find more and more complex issues, those models are already very suitable.

7 Discussion

This thesis shows that modern LLMs can be used to aid Linux kernel development process, but perhaps the most important finding in the study was related to the difficulty to collect and process the data needed for the training of the models. Although, the sources for the data were clear, how to utilise those sources required that the tasks for the model was decided first. In this study the initial

task of the model was to find the basic quality problems from the patches, so training the model required that the differences between the patch revisions were collected. This was not as straightforward as it may sound. It took a lot of time and effort to figure out the way how to do it that was described in this thesis.

Besides the difficulties related to the data collecting, the data processing was also not at all straightforward. The programming languages are more complex in some ways compared to the natural languages. This thesis explains how the C programming language is used to form a hierarchical trees of statements, which do not exist in natural languages, and that the C code also contains parts that are not processed as C programming language - mainly the macros and comment parts in the code. With programming languages a decision has to be made regarding which parts of the code are used, as well as in what way. In this thesis entire C functions were used, but alternatively only statements could have been used, or even entire C source code files. The decision impacts the size and performance of the model, so it requires careful consideration.

The ethical and legal consideration is also very important. GPLv2 licence means that it is very difficult to utilise data sources outside of the Linux kernel project. The amount of available training data may always be relatively small because of the restrictive licence.

The major lesson that should be learned from this thesis is that handing the training data can be more demanding and time consuming then it is to create and train the model itself, but it should also be noted that in most cases the problem is about selecting from a set of alternative approaches.

In the future research, those alternative approaches could be compared. As an example, the model performance of the model that was trained with the patch revisions differences could be compared to a separate model that is trained with simply the patches themselves. Comparing alternative approaches will obviously requires more time, as multiple model are trained instead of one, but it will also require high performance hardware with a lot of memory available for the training, because some of the decisions that need to be made will be related to the model sizes.

References

Active kernel releases. (2024). Linux Kernel Organization. Retrieved February 10, 2024, from https://www.kernel.org/category/releases.html

Arxiv.org e-print archive. (2024). Cornell University. Retrieved March 10, 2024, from https://www.arxiv.org/

Beltagy, I., Peters, M. E., & Cohan, A. (2020). Longformer: The long-document transformer. https://doi.org/10.48550/ARXIV.2004.05150

Bielik, P., Raychev, V., & Vechev, M. (2016). Learning a static analyzer from data. https://doi.org/10.48550/ARXIV.1611.01752

Bovet, D. P., & Cesati, M. (2006). *Understanding the linux kernel*. O'Reilly.

Checkpatch. (2024). The kernel development community. Retrieved February 1, 2024, from https://www.kernel.org/doc/html/latest/dev-tools/checkpatch.html

Claburn, T. (2022). *Arrogant, subtle, entitled: 'Toxic' open source GitHub discussions examined*. Retrieved March 10, 2024, from https://www.theregister.com/2022/06/29/toxic_interaction_github_open_source/

Clang: A c language family frontend for llvm. (2024). LLVM Developer Group. Retrieved April 9, 2024, from https://clang.llvm.org/

Copilot. (2021). GitHub. Retrieved April 1, 2024, from https://copilot.github.com/

Corbet, J. (2006). *Where have all the reviewers gone?* Retrieved February 1, 2024, from https://lwn.net/Articles/199110/

Corbet, J., Rubini, A., & Kroah-Hartman, G. (2005). *Linux device drivers, 3rd edition*. O'Reilly.

Data collator. (2024). Hugging Face, Inc. Retrieved March 10, 2024, from https://huggingface.co/docs/transformers/main/en/main_classes/data_collator#transformers.DataCollatorForLanguageModeling

Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2018). Bert: Pre-training of deep bidirectional transformers for language understanding. https://doi.org/10.48550/ARXIV.1810.04805

Ethics guidelines for trustworthy ai. (2019). European Commission. Retrieved April 10, 2024, from https://digital-strategy.ec.europa.eu/en/library/ethics-guidelines-trustworthy-ai

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., & Zhou, M. (2020). Codebert: A pre-trained model for programming and natural languages. https://doi.org/10.48550/ARXIV.2002.08155

Firefox. (2024). Mozilla Foundation. Retrieved March 28, 2024, from https://www.mozilla.org/fi/firefox/

Flax. (2024). Google. Retrieved April 10, 2024, from https://flax.readthedocs.io/en/latest/index.html

Gerrit code review. (2024). Gerrit. Retrieved February 1, 2024, from https://www.gerritcodereview.com/

Git - git-diff documentation. (2024). Git Project. Retrieved April 9, 2024, from https://git-scm.com/docs/git-diff

Gnome. (2024). The GNOME Project. Retrieved March 28, 2024, from https://www.gnome.org/

Gnu general public license, version 2. (1991). Free Software Foundation. Retrieved April 10, 2024, from https://www.gnu.org/licenses/old-licenses/gpl-2.0.html

Gnu project c and c++ compiler. (2024). Free Software Foundation, Inc. Retrieved April 9, 2024, from https://gcc.gnu.org/

A guide to the kernel development process. (2024). The kernel development community. Retrieved February 1, 2024, from https://www.kernel.org/doc/html/latest/process/development-process.html

Gupta, A., & Sundaresan, N. (2018). *Intelligent code reviews using deep learning*. The Association for Computing Machinery Special Interest Group on Knowledge Discovery and Data Mining. Retrieved March 28, 2024, from https://www.kdd.org/kdd2018/files/deep-learning-day/DLDay18_paper_40.pdf

Gupta, R., Pal, S., Kanade, A., & Shevade, S. (2017). Deepfix: Fixing common c language errors by deep learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, *31*(1). https://doi.org/10.1609/aaai.v31i1.10742

Hoang, T., Lawall, J., Tian, Y., Oentaryo, R. J., & Lo, D. (2019). Patchnet: Hierarchical deep learning-based stable patch identification for the linux kernel. IEEE Transactions on Software Engineering. https://doi.org/10.1109/TSE.2019.2952614

Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, *9*(8), 1735–1780.

Hugging face. (2024). Hugging Face, Inc. Retrieved March 10, 2024, from https://huggingface.co/

Intel deep learning boost new deep learning instruction bfloat16. (2024). Intel Corporation. Retrieved March 10, 2024, from https://www.intel.com/content/www/us/en/developer/articles/technical/intel-deep-learning-boost-new-instruction-bfloat16.html

Intel developer cloud. (2024). Intel Corporation. Retrieved March 10, 2024, from https://www.intel.com/content/www/us/en/developer/tools/devcloud/overview.html

International Organization for Standardization. (2023). *Programming languages — C. (ISO/IEC 9899:2023)*. https://www.iso.org/standard/82075.html.

Kanade, A., Maniatis, P., Balakrishnan, G., & Shi, K. (2020). Learning and evaluating contextual embedding of source code. https://doi.org/10.48550/ARXIV.2001.00059

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language*. Prentice Hall.

Language model training examples. (2024). Hugging Face, Inc. Retrieved April 10, 2024, from https://github.com/huggingface/transformers/tree/main/examples/flax/language-modeling

Le, Q. V., & Mikolov, T. (2014). Distributed representations of sentences and documents. https://doi.org/10.48550/ARXIV.1405.4053

Lee, J.-W., Kang, H., Lee, Y., Choi, W., Eom, J., Deryabin, M., Lee, E., Lee, J., Yoo, D., Kim, Y.-S., & No, J.-S. (2021). Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. https://doi.org/10.48550/ARXIV.2106.07229

Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., & Zettlemoyer, L. (2019). Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. https://doi.org/10.48550/ARXIV.1910.13461

Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., & Sundaresan, N. (2022). Automating code review activities by large-scale pre-training. https://doi.org/10.48550/ARXIV.2203.09095

Libclang. (2024). LLVM Developer Group. Retrieved April 9, 2024, from https://clang.llvm.org/docs/LibClang.html

Lin, C.-Y. (2004). Rouge: A package for automatic evaluation of summaries. https://aclanthology.org/W04-1013

The linux kernel archives. (2024). Linux Kernel Organization. Retrieved February 10, 2024, from https://www.kernel.org/

Linux kernel performance*. (2024). Intel. Retrieved February 1, 2024, from https://www.intel.com/content/www/us/en/developer/topic-technology/open/linux-kernel-performance/overview.html

List archives on lore.kernel.org. (2024). Linux Kernel Organization. Retrieved February 10, 2024, from https://www.kernel.org/lore.html

List of maintainers. (2024). The kernel development community. Retrieved February 1, 2024, from https://www.kernel.org/doc/html/latest/process/maintainers.html

Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the middle: How language models use long contexts. https://doi.org/10.48550/ARXIV.2307.03172

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., & Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach. https://doi.org/10.48550/ARXIV.1907.11692

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. https://doi.org/10.48550/ARXIV.1301.3781

Nicholson, D. (2016). *Solving the linux kernel code reviewer shortage*. Retrieved February 1, 2024, from https://opensource.com/business/16/10/linux-kernel-review

Openai codex. (2021). OpenAI. Retrieved April 1, 2024, from https://openai.com/blog/openai-codex

Operating system market share worldwide. (2024). Statcounter. Retrieved February 2, 2024, from https://gs.statcounter.com/os-market-share

Papineni, K., Roukos, S., Ward, T., & Zhu, W.-J. (2001). Bleu: A method for automatic evaluation of machine translation. *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02*. https://doi.org/10.3115/1073083.1073135

Peng, B., Narayanan, S., & Papadimitriou, C. (2024). On limitations of the transformer architecture. https://doi.org/10.48550/ARXIV.2402.08164

Puri, R., Kung, D. S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., Thost, V., Buratti, L., Pujar, S., Ramji, S., Finkler, U., Malaika, S., & Reiss, F. (2021). Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. https://doi.org/10.48550/ARXIV.2105.12655

Radford, A., Narasimhan, K., Salimans, T., & Sutskever, I. (2018). *Improving language understanding by generative pre-training*. OpenAI. Retrieved February 10, 2024, from https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

Radhakrishnan, P. (2021). *Why transformers are slowly replacing cnns in computer vision?* Medium. Retrieved February 10, 2024, from https://becominghuman.ai/transformers-in-vision-e2e87b739feb

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, *21*(140), 1–67. http://jmlr.org/papers/v21/20-074.html

Sennrich, R., Haddow, B., & Birch, A. (2015). Neural machine translation of rare words with subword units. https://doi.org/10.48550/ARXIV.1508.07909

Sparse. (2024). The kernel development community. Retrieved February 1, 2024, from https://www.kernel.org/doc/html/latest/dev-tools/sparse.html

Submitting patches: The essential guide to getting your code into the kernel. (2024). The kernel development community. Retrieved February 1, 2024, from https://www.kernel.org/doc/html/latest/process/submitting-patches.html

Systemization of knowledge (sok) papers. (2024). Journal of Systems Research. Retrieved May 24, 2024, from https://www.jsys.org/type_SoK/

T5. (2024). Hugging Face, Inc. Retrieved March 10, 2024, from https://huggingface.co/docs/transformers/model_doc/t5

Tensor processing units. (2024). Google. Retrieved March 10, 2024, from https://cloud.google.com/tpu/

Thangavel, T. (2023). *Limitations of transformer architecture*. Retrieved March 10, 2024, from https://medium.com/@thirupathi.thangavel/limitations-of-transformer-architecture-4e6118cbf5a4

Thomas, R. J. (2023). *Small but Powerful: A Deep Dive into Small Language Models (SLMs)*. Retrieved March 10, 2024, from https://medium.com/version-1/small-but-powerful-a-deep-dive-into-small-language-models-slms-b793bdb002f2

Towards data science. (2024). Medium Inc. Retrieved March 10, 2024, from https://towardsdatascience.com/

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need. https://doi.org/10.48550/ARXIV.1706.03762

VK, A. (2023). *'Toxic' Linux Community Acts As the Perfect Repellent for New Users*. Retrieved March 10, 2024, from https://analyticsindiamag.com/toxic-linux-community-acting-as-the-perfect-repellent-for-new-users/

Wang, X., Wang, Y., Mi, F., Zhou, P., Wan, Y., Liu, X., Li, L., Wu, H., Liu, J., & Jiang, X. (2021). Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. https://doi.org/10.48550/ARXIV.2108.04556

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D. Q., Li, J., & Hoi, S. C. H. (2023). Codet5+: Open code large language models for code understanding and generation. https://doi.org/10.48550/ARXIV.2305.07922

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. https://doi.org/10.48550/ARXIV.2109.00859

Wu, F. (2020). Linux kernel performance tests [Source Code]. Retrieved February 1, 2024, from https://github.com/intel/lkp-tests

Zhang, J., Zhao, Y., Saleh, M., & Liu, P. J. (2019). Pegasus: Pre-training with extracted gap-sentences for abstractive summarization. https://doi.org/10.48550/ARXIV.1912.08777

Appendix 1. Line counting script

```
#!/bin/sh
files=$(find -name "*.[hc]"|wc -l)"
echo "files: $(echo $files|wc -l)"
lines=0
for f in in $files; do
    let lines=$lines+$(wc -l $f|cut -d ' ' -f 1)
    echo -ne "\r$lines"
done
echo -e "lines: \r$lines")
```

Appendix 2. Script that collects C functions from source files

```
#
# Collect kernel C functions to CSV file
#
# Run in Linux kernel source directory! Requires files.txt file that lists the
# source files that you want to use. Create that for example like this:
#
#   find arch/ crypto/ block/ drivers/ fs/ kernel/ lib/ mm/ net/ sound/ -name "*.c" > files.txt
#
# The output is a file named "kernel_functions.csv.gz".
#
# Depends on pandas and libclang.
#

import clang.cindex
import pandas as pd

index = clang.cindex.Index.create()
options = clang.cindex.TranslationUnit.PARSE_DETAILED_PROCESSING_RECORD

# Only include the code part in the data by removing the comments.
def strip_comments(code, node):
    last = None
    data = ""
    for token in node.get_tokens():
        if (token.kind == clang.cindex.TokenKind.COMMENT):
            last = token
            continue
        # Read also the white spaces/newlines/tabs before the token.
        if last: # All the other tokens except the first.
            data += code[last.extent.end.offset:token.extent.end.offset]
        else: # This is the first token.
            data += code[token.extent.start.offset:token.extent.end.offset]
        last = token
    return [node.location.file.name, node.location.line, data]
```

```
functions = []
with open("files.txt", "r") as files:
    lines = files.read().splitlines()
    for i, file in enumerate(lines):
        code = open(file, "r").read()
        unit = index.parse(file, options = options)
        print("% 4d/%d: %s" % (i + 1, len(lines), file.rstrip('\n')))

        # Walk through all the child cursors (definitions and declarations).
        for node in unit.cursor.get_children():
            # Ignore the included files.
            if not node.location.file or node.location.file.name != file:
                continue
            # If you comment out this condition, also the structure definitions
            # are included.
            if node.kind != clang.cindex.CursorKind.FUNCTION_DECL:
                continue
            # Function (and other) prototypes are excluded.
            if node.is_definition() and node.kind.is_declaration():
                functions.append(strip_comments(code, node))
                # If you want to include also the comments:
                # functions.append([node.location.file, node.location.line, node.spelling])

df = pd.DataFrame(functions, columns = ["file", "line", "function"])
df.to_csv("kernel_functions.csv.gz")
```

Appendix 3. Data Preprocessing

```
import pandas as pd
from datasets import Dataset
from transformers import AutoTokenizer

checkpoint = "distilroberta-base"
block_size = 64
num_proc = 20

df = pd.read_csv("kernel_v6_7_functions.csv.gz", index_col = 0)
df = df.dropna(axis = 0, how = "any")

dataset = Dataset.from_pandas(df)

def get_training_corpus():
    return (
        dataset[i : i + 1000]["function"]
        for i in range(0, len(dataset), 1000)
    )

#
# Train tokenizer.
#
print("Train tokenizer:")
training_corpus = get_training_corpus()
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
tokenizer = tokenizer.train_new_from_iterator(training_corpus, 50000)
# Save to separate sub-folder.
tokenizer.save_pretrained("./tokenizer")

def tokenise(data):
    return tokenizer([" ".join(f) for f in data["function"]])

# Split the dataset (80% train, 20% test) and tokenize.
print("Tokenize the data:")
dataset = dataset.train_test_split(test_size = 0.2)
```

```
tokenised = dataset.map(tokenise, batched = True,
                        remove_columns = dataset["train"].column_names)

def regroup_functions(examples):
    # Concatenate all texts.
    concatenated_examples = {k: sum(examples[k], []) for k in examples.keys()}
    total_length = len(concatenated_examples[list(examples.keys())[0]])
    # Just dropping the remainder.
    if total_length >= block_size:
        total_length = (total_length // block_size) * block_size
    # Split by chunks of block_size.
    result = {
        k: [t[i : i + block_size] for i in range(0, total_length, block_size)]
        for k, t in concatenated_examples.items()
    }
    return result

print("Regroup the data:")
dataset = tokenised.map(regroup_functions, batched = True, num_proc = num_proc)

# Saving also the final dataset to separate sub-folder for now.
dataset.save_to_disk("./dataset")
```

Appendix 4. Data Collator for T5 MLM

```
import numpy as np
import tensorflow as tf
from transformers import BatchEncoding
from transformers import PreTrainedTokenizerBase
from typing import Dict, List

def shift_tokens_right(input_ids , pad_token_id: int, decoder_start_token_id: int) :
    """
    Shift input ids one token to the right.
    """
    shifted_input_ids = np.zeros(input_ids.shape, dtype = np.int32)
    shifted_input_ids[:,1:] = input_ids[:,:-1]
    shifted_input_ids[:,0] = decoder_start_token_id
    shifted_input_ids[shifted_input_ids==-100] = pad_token_id
    return shifted_input_ids

class DataCollatorForT5MLM:
    """
    Data collator used for T5 span-masked language modeling.
    It is made sure that after masking the inputs are of length `data_args.max_seq_length` and targets are also of fixed length.
    For more information on how T5 span-masked language modeling works, one can take a look
    at the `official paper <https://arxiv.org/pdf/1910.10683.pdf>`__
    or the `official code for preprocessing
    <https://github.com/google-research/text-to-text-transfer-transformer/blob/master/t5/data/preprocessors.py>`__ .

    Args:
        tokenizer (:class:`~transformers.PreTrainedTokenizer` or :class:`~transformers.PreTrainedTokenizerFast`):
            The tokenizer used for encoding the data.
        noise_density (:obj:`float`):
            The probability with which to (randomly) mask tokens in the input.
        mean_noise_span_length (:obj:`float`):
            The average span length of the masked tokens.
        input_length (:obj:`int`):
            The expected input length after masking.
        target_length (:obj:`int`):
            The expected target length after masking.
        pad_token_id: (:obj:`int`):
            The pad token id of the model
        decoder_start_token_id: (:obj:`int):
            The decoder start token id of the model
    """

    def __init__(self,
        tokenizer: PreTrainedTokenizerBase,
        noise_density: float,
        mean_noise_span_length: float,
        input_length: int,
        target_length: int,
        pad_token_id: int,
        decoder_start_token_id: int):

        self.tokenizer = tokenizer
        self.noise_density = noise_density
        self.mean_noise_span_length = mean_noise_span_length
        self.input_length = input_length
        self.target_length = target_length
        self.pad_token_id = pad_token_id
        self.decoder_start_token_id = decoder_start_token_id

    def __call__(self, examples: List[Dict[str, list]]) -> BatchEncoding:
        # convert list to dict and tensorize input
```

```
    batch = BatchEncoding(
        {k: np.array([examples[i][k] for i in range(len(examples))]) for k, v in examples[0].items()}
    )

    input_ids = batch["input_ids"]
    batch_size, expandend_input_length = input_ids.shape

    mask_indices = np.asarray([self.random_spans_noise_mask(expandend_input_length) for i in range(batch_size)])
    labels_mask = ~mask_indices

    input_ids_sentinel = self.create_sentinel_ids(mask_indices.astype(np.int8))
    labels_sentinel = self.create_sentinel_ids(labels_mask.astype(np.int8))

    batch["input_ids"] = self.filter_input_ids(input_ids, input_ids_sentinel)
    batch["labels"] = self.filter_input_ids(input_ids, labels_sentinel)

    if batch["input_ids"].shape[-1] != self.input_length:
        raise ValueError(
            f"`input_ids` are incorrectly preprocessed. `input_ids` length is {batch['input_ids'].shape[-1]}, but"
            f" should be {self.input_length}."
        )

    if batch["labels"].shape[-1] != self.target_length:
        raise ValueError(
            f"`labels` are incorrectly preprocessed. `labels` length is {batch['labels'].shape[-1]}, but should be"
            f" {self.target_length}."
        )

    # to check that tokens are correctly preprocessed, one can run `self.tokenizer.batch_decode(input_ids)` and `self.tokenizer.batch_decode(labels)` here...
    batch["decoder_input_ids"] = shift_tokens_right(
        batch["labels"], self.pad_token_id, self.decoder_start_token_id
    )

    return batch

def create_sentinel_ids(self, mask_indices):
    """
    Sentinel ids creation given the indices that should be masked.
    The start indices of each mask are replaced by the sentinel ids in increasing
    order. Consecutive mask indices to be deleted are replaced with `-1`.
    """
    start_indices = mask_indices - np.roll(mask_indices, 1, axis=-1) * mask_indices
    start_indices[:, 0] = mask_indices[:, 0]

    sentinel_ids = np.where(start_indices != 0, np.cumsum(start_indices, axis=-1), start_indices)
    sentinel_ids = np.where(sentinel_ids != 0, (len(self.tokenizer) - sentinel_ids), 0)
    sentinel_ids -= mask_indices - start_indices

    return tf.convert_to_tensor(sentinel_ids, np.int32)

def filter_input_ids(self, input_ids, sentinel_ids):
    """
    Puts sentinel mask on `input_ids` and fuse consecutive mask tokens into a single mask token by deleting.
    This will reduce the sequence length from `expanded_inputs_length` to `input_length`.
    """
    batch_size = input_ids.shape[0]

    input_ids_full = np.where(sentinel_ids != 0, sentinel_ids, input_ids)
    # input_ids tokens and sentinel tokens are >= 0, tokens < 0 are
    # masked tokens coming after sentinel tokens and should be removed
    input_ids = input_ids_full[input_ids_full >= 0].reshape((batch_size, -1))
    input_ids = np.concatenate(
        [input_ids, np.full((batch_size, 1), self.tokenizer.eos_token_id, dtype=np.int32)], axis=-1
```

```
    )
    return input_ids

def random_spans_noise_mask(self, length):
    """This function is copy of `random_spans_helper
    <https://github.com/google-research/text-to-text-transfer-transformer/blob/84f8bcc14b5f2c03de51bd3587609ba8f6bbd1cd/t5/data/preprocessors.py#L2682>`__ .

    Noise mask consisting of random spans of noise tokens.
    The number of noise tokens and the number of noise spans and non-noise spans
    are determined deterministically as follows:
    num_noise_tokens = round(length * noise_density)
    num_nonnoise_spans = num_noise_spans = round(num_noise_tokens / mean_noise_span_length)
    Spans alternate between non-noise and noise, beginning with non-noise.
    Subject to the above restrictions, all masks are equally likely.

    Args:
        length: an int32 scalar (length of the incoming token sequence)
        noise_density: a float - approximate density of output mask
        mean_noise_span_length: a number

    Returns:
        a boolean tensor with shape [length]
    """

    orig_length = length

    num_noise_tokens = int(np.round(length * self.noise_density))
    num_nonnoise_tokens = length - num_noise_tokens
    # avoid degeneracy by ensuring positive numbers of noise and nonnoise tokens.
    num_noise_tokens = min(max(num_noise_tokens, 1), length - 1)
    # num_noise_tokens should be less than num_noise_tokens and num_nonnoise_tokens
    num_noise_spans = int(np.round(min(num_noise_tokens, num_nonnoise_tokens) / self.mean_noise_span_length))

    # avoid degeneracy by ensuring positive number of noise spans
    num_noise_spans = max(num_noise_spans, 1)

    # pick the lengths of the noise spans and the non-noise spans
    def _random_segmentation(num_items, num_segments):
        """Partition a sequence of items randomly into non-empty segments.
        Args:
            num_items: an integer scalar > 0
            num_segments: an integer scalar in [1, num_items]
        Returns:
            a Tensor with shape [num_segments] containing positive integers that add
            up to num_items
        """
        mask_indices = np.arange(num_items - 1) < (num_segments - 1)
        np.random.shuffle(mask_indices)
        first_in_segment = np.pad(mask_indices, [[1, 0]])
        segment_id = np.cumsum(first_in_segment)
        # count length of sub segments assuming that list is sorted
        _, segment_length = np.unique(segment_id, return_counts=True)
        return segment_length

    noise_span_lengths = _random_segmentation(num_noise_tokens, num_noise_spans)
    nonnoise_span_lengths = _random_segmentation(num_nonnoise_tokens, num_noise_spans)

    interleaved_span_lengths = np.reshape(
        np.stack([nonnoise_span_lengths, noise_span_lengths], axis=1), [num_noise_spans * 2]
    )
    span_starts = np.cumsum(interleaved_span_lengths)[:-1]
    span_start_indicator = np.zeros((length,), dtype=np.int8)
    span_start_indicator[span_starts] = True
```

```
span_num = np.cumsum(span_start_indicator)
is_noise = np.equal(span_num % 2, 1)

return is_noise[:orig_length]
```

Appendix 5. Doc2Vec Experiment

```
import logging
import pandas as pd
from gensim.models.doc2vec import TaggedDocument
from gensim.models.doc2vec import Doc2Vec
from gensim.test.utils import get_tmpfile

print("Reading DataFrame...", flush = True)

df = pd.read_csv("tokens.csv", index_col = 0, dtype = str, keep_default_na=False)

print(df)

print("Convert to TaggedDocument", flush = True)

sentences = []
for index, row in df.iterrows():
    sentences.append(TaggedDocument(row["tokens"].split(), [row["file"], row["line"]]))
    if not index % 1000:
        print("\r%d/%d" % (index, len(df)), end = '')
print("\r%d/%d" % (index + 1, len(df)))

logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)

model = Doc2Vec(dm = 1, vector_size = 100, min_count = 3, workers = 28, epochs = 40,
                alpha = 0.025, min_alpha = 0.001)

model.build_vocab(sentences)
model.train(sentences, total_examples = model.corpus_count, epochs = model.epochs)

with open("doc2vec.model", "wb") as f:
    model.save(f)
```