CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Searching for Android security defects with the help of an NLP machine learning model and existing vulnerability data

Master's thesis in Computer science and engineering

Jakub Dudek

Markus Saarijärvi

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Searching for Android security defects with the help of an NLP machine learning model and existing vulnerability data

Jakub Dudek

Markus Saarijärvi

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Searching for Android security defects with the help of NLP
Jakub Dudek, Markus Saarijärvi

Supervisor: Miroslaw Staron, Department of Computer Science and Engineering.
Advisor: Mirosław Ochodek, Poznań University of Technology
Examiner: Jennifer Horkoff, Department of Computer Science and Engineering.

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2021

Searching for Android security defects with the help of NLP
Searching for Android security defects with the help of NLP
Jakub Dudek, Markus Saarijärvi
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Keywords: Natural Language Processing, Machine Learning, Security, defects, Android, AOSP, code reviews.

Abstract

Background

Over the years, as the modern code review became a common software engineering practice, the underlying benefits of the review process shifted away from finding defects and instead now center around knowledge sharing and communication. As such, there is demand for new tooling which is able to find defects and integrates into the modern code review. One of these tools is the Automatic Code Review Assistant, ACoRA, which by training on previously performed code reviews can automatically find defects in new code. Although ACoRA could in theory be trained to locate any type of software defect, this study limits the scope to only security vulnerabilities.

Aim

ACoRA trains on previously performed code reviews, specifically those code reviews which showcase occurrences of programming defects. The study aims to design and evaluate a new artifact dubbed SeCoRA, intended to facilitate the process of acquiring these code reviews by making use of a database containing existing known security vulnerabilities.

Method

The study follows the design science methodology. Using an unsupervised machine learning model, SeCoRA is able to compare two fragments of code against each other and express how similar they are. Based on this ability, the common vulnerabilities and exposures database can be used to discover code reviews which contain vulnerable code. The assumption here is that if a code review contains code which is similar to an existing vulnerability, that code is also potentially defective. SeCoRA is built specifically to gather these code reviews from the Android Open Source Project.

Results

SeCoRA was evaluated firstly by distinguishing on code in general with both lines and blocks of code. The bigger size of code fragments did not improve the ability of comparing code, and hence, lines of code were used to filter out a set of 1194 code reviews for similar code. Using this approach resulted in 11 code reviews to be found containing potential security defects but did not adhere to the classification from the original code.

Conclusions

Although SeCoRA was able to distinguish between different lines of code, the tool is not sufficiently good to find security related code reviews. Therefore, the results are negative, as the tool does not solve the problem of acquiring the data necessary to train ACoRA. As part of the final discussion, the authors present a project post-mortem and lay the ground for possible future work.

Acknowledgements

We would like to express our most profound appreciation for this opportunity to learn and work with the tool ACoRA. Sincere thanks go to our supervisor, Miroslaw Staron, for the guidance and support throughout the whole thesis. We would also like to thank our advisor, Mirosław Ochodek, for all the time he took explaining, guiding, and discussing how to take advantage of ACoRA to the fullest. This thesis also took use of the work done by Linares-Vasquez et al. [1]. Your work is deeply appreciated, as this thesis may not have been possible without it. Lastly, we would like to thank our examiner, Jennifer Horkoff, for taking her time to read and give helpful feedback and guidance on how to complete the thesis.

Jakub Dudek, Markus Saarijärvi, Gothenburg, June 2021

Contents

Acronyms

ACoRA Automatic Code Review Assistant.
AOSP Android Open Source Project.
API Application Programming Interface.

BERT Bidirectional Encoder Representation from Transformers.

CI Continuous Integration.
CVE Common Vulnerabilities and Exposures.
CWE Common Weakness Enumeration.

DSR Design Science Research.

HTTP Hypertext Transfer Protocol.

LTR Left-To-Right.

MCR Modern Code Review.
ML Machine Learning.
MLM Masked Language Modeling.

NLP Natural Language Processing.
NSP Next Sentence Prediction.

REST Representational State Transfer.
RNN Recurrent Neural Networks.
RTL Right-To-Left.

SeCoRA Security Code Review Assistant.

List of Figures

List of Tables

1

Introduction

Due to the error prone nature of humans and programming, bugs are something that software engineers have to expect and actively work to prevent [3]. Bugs like these, typically security vulnerabilities, can potentially lead to damages costing the company owning the product enormous sums of money[1,2]. Software quality assurance has therefore been a major part of the software engineering discipline going back to the 70s with commonly known approaches such as testing and code inspection [4]. Over the decades, the code inception techniques evolved from formal inspections of code and design towards a more minimal and iterative code review process known as the modern code review (MCR) that can be commonly found in open source projects [5]. Formal inspections have proven to improve the quality of software. Still, the process itself has been seen as cumbersome with overly strict review criteria, e.g., in-person meetings or checklists [6]. In contrast, the modern review process has a more lightweight approach and is not as stern as its precursors.

However, as the MCR became more common, the underlying purpose of code reviews have shifted. Bacheli and Bird [7] show that the actual benefits of reviews do not match the typical expectations of finding programming defects. Instead, aspects such as knowledge sharing and communication are the ones that benefit the most. Moving away from practices like the formal code review or perspective-based reading has increased the need for new ways of finding software defects that integrate into the MCR. Mainly, defects such as security vulnerabilities are of high interest as those can potentially have a great financial impact [8].

A tool that attempts to solve this issue is the Automatic Code Review Assistant (ACoRA). ACoRA is a toolchain intended to facilitate the creation and usage of code review related machine learning models. In short, ACoRAs objective is to learn from previously performed code reviews in order to perform new reviews automatically. When fully functioning, the tool cannot only highlight defective code but also classify the cause of the defect based on a taxonomy specified during the learning process. This classification is then also additionally enhanced by presenting examples of code review comments which have been seen with previous similar instances of the same type of defect. The tool can, in general, operate on any type of defect but can also

[1] `https://www.nytimes.com/2016/12/14/technology/yahoo-hack.html`

[2] `https://www.cnbc.com/2018/09/26/uber-to-pay-148-million-for-2016-data-breach-and-cover-up.html`

be specialized to focus on one type only, which in the case of this study is security.

ACoRA's training can be divided into two distinct steps:

1. Given samples of code, the tool learns the semantics and syntax of the language used to write the samples. This training is performed in an unsupervised fashion.
2. With the ability to understand code, the tool is given examples of code reviews showcasing programming defects and, in a supervised fashion, learns to detect and classify those defects into, e.g., a security taxonomy.

As with most machine learning solutions, an ample amount of data is needed to provide enough examples for the model to learn from. Gathering data for ACoRA's first training step is straightforward as any software repository of substantial size can be used. On the other hand, the process is much more complicated during the second step, as there is no clear indicator of whether a particular review relates to security or not and what type of defect it contains. This limitation brings forward the question which drives this study; *Given the code review history of a software repository, how does one discern which reviews exemplify specific security defects?*

To answer this question, the authors designed a new software artifact dubbed as the Security Code Review Assistant (SeCoRA). SeCoRA builds on top of the simpler model created during the unsupervised training of ACoRA. This model learns the semantics and syntax of code and has the ability to express how similar two code fragments are to each other. SeCoRA takes this ability and attempts to use existing known security vulnerabilities and use them as a means to discern which code reviews relate to security defects and which don't. The assumption made here is that if a code review contains code which is very similar to a known vulnerability, there is a high chance that the code itself is also defective. *The aim of the study is thus to evaluate to what degree can SeCoRA solve the problem of discerning security related code reviews among a project's review history.*

What makes ACoRA stand out and gives this study significance is the novelty of the underlying machine learning technique. As code contains rich and complicated structural information, other natural language processing (NLP) solutions typically require some sort of pre-processing [9] such as generating an Abstract Syntax Tree before providing it to the machine model [10]. Instead, ACoRA makes use of Google's Bidirectional Encoder Representation from Transformers [11] (BERT), a technique that requires no pre-processing and is able to take in the code as is. At the time of writing this text, to our knowledge, ACoRA is the only tool which uses BERT as a means to find defects in code.

1.1 Limitations

Because of time constraints, there were certain measures taken to delimit the size of the study. While ACoRA could potentially operate on any type of defect, the

artifact designed throughout the study focuses only on security. This choice was made based on which defect would bring the most value. The authors decided that between different defect types, finding security defects has the most impact. Additionally, in theory ACoRA could be applied to any software project, but for simplicity the study focuses only on the Android Open Source Project (AOSP). AOSP was selected based on the following factors:

- The substantial size of the project and its rich code review history.
- An open API which makes it possible to gather code review data.
- The large amount of previously performed studies on the topic of Android .

1.2 Research questions

Below we formalise and explain the research questions which form the core of this study. Each question is given an identifier which is used as a reference through the rest of this text.

RQ 1: *To what degree can SeCoRA distinguish between code fragments in general?* SeCoRA's capability to distinguish between different code fragments is of importance to assess if the tool can be used to find new security vulnerabilities.

RQ 2: *How can we acquire security-labeled code fragments to identify new security vulnerabilities?* The data serves as a source of known existing examples of security vulnerabilities and makes it possible for SeCoRA to search for similar fragments.

RQ 3: *Which security vulnerabilities can we find in AOSP code reviews when searching with fragments from **RQ2**?* When combining SeCoRA's ability from **RQ1** and the code fragments from **RQ2** we try to find new security vulnerabilities in the AOSP code reviews.

2

Related Work

This chapter describes the *naturalness hypothesis*, a theory that states why natural language processing (NLP) models apply to code, and gives an overview of the modern code review process and the corresponding tooling.

2.1 The Naturalness Hypothesis

Allamanis et al. [9] have suggested a hypothesis, called the *naturalness hypothesis*, stating that code is also a form of communication and therefore follows statistical patterns similar to the natural language. Allmanis et al. formally describe this hyptothesis as:

> **The naturalness hypothesis.** *Software is a form of human communication; software corpora have similar statistical properties to natural language corpora; and these properties can be exploited to build better software engineering tools.*

Based on this hypothesis, one can expect large code corpora to have rich patterns, similar to natural language. This, in turn allows code related software engineering tools to use already existing probabilistic Machine Learning (ML) models typically used with natural language. The first empirical evidence of the *naturalness hypothesis*, showing that ML models originally developed for NLP were effective for source code, was presented by Hindle et al. [12, 13].

In their article, Allamanis et al. review the emerging area of machine learning and statistical NLP methods applied to source code. They also describe software engineering and programming language applications of the probabilistic source code models [9]:

- **Recommender Systems:** makes recommendations to assist software engineering tasks, such as code auto-completion (e.g. [14, 15]).
- **Inferring Coding Conventions:** seeks to prevent some classes of bugs and makes code easier to comprehend, navigate, and maintain (e.g. [16, 17, 18]).
- **Code defects:** are found by probabilistic models trained on source code which assigns high probability to code that appears often in practice, e.i., is natural.

Therefore, code considered very improbable may be buggy (e.g. [19, 20, 21]).

- **Code Translation, Copying, and Clones:** is the usage of ML to translate code from one source language (e.g. Java) to another (e.g. C#) (e.g. [22, 23]).
- **Code to Text and Text to Code:** is the linking of natural language text to source code which has useful applications, such as program synthesis, traceability, search, and documentation (e.g. [24, 25]).
- **Documentation, Traceability, and Information Retrieval:** are more specialized solutions where probabilistic models are integrating information between natural language text and code (e.g. [26, 27]).
- **Program Synthesis**: is concerned with generating full or partial programs from a specification (e.g. [28, 29]).

2.2 The Modern Code Review

Alberto and Bird [7] describe how the code-review process used by many companies today has evolved into a much more lightweight practice when compared with the formal inspections proposed by Fagan [4] in the 70s. Besides the shift in mentality, the modern code review (MCR) has also spawned a vast arrange of platforms and tooling with the sole purpose of supporting MCR. These tools support the underlying process of submitting, reviewing, commenting, and merging software changes. Examples of such platforms are Microsoft's CodeFlow, Gerrit (commonly used by Google), Facebooks Phabricator, and Github. Rigby and Bird [30] provide an overview of the differences in some of these platforms but, most importantly, put together a set of convergent practices that are common between them and setting a core definition of what defines an MCR.

- Modern peer reviews follows a lightweight, flexible process;
- Reviews happen early (before a change is committed), quickly, and frequently;
- Change sizes are small;
- Two reviewers find an optimal number of defects; and
- Reviews has changed from a defect finding activity to a group problem solving activity.

Not only has the MCR process evolved drastically in practice since the 70s, but Alberto and Bird [7] also describe a shift in expectations. While the top motivating factor for applying the practice is still finding defects, MCR users will also associate it with other benefits such as knowledge transfer, team awareness, and improved solutions to problems. The study goes even as far as showing that there is a mismatch between the expectations and the actual outcomes of the reviews and that relying on code reviews as a way for quality assurance may be fraught.

2.2.1 Modern Code review tooling

Code reviews today are commonly a part of a Continuous Integration (CI) workflow. Before the code reaches a reviewer, a dedicated CI server will build, test, and perform static analysis on the proposed changes [31]. This type of CI flow is where tools like ACoRA can process the code and perform automated actions to improve the reviewer's experience.

Zampetti et al. [32] performed a comprehensive analysis of static analysis tooling in CI pipelines from 20 Java open source projects. The tooling covers a wide arrange of tasks such as enforcing adherence to standards/guidelines, adherence to the intended architecture, and fault detection. Their analysis focuses on analyzing the CI process of each project and understanding what tools are used and how they are configured. Particularly interesting is the research question regarding what types of issues are configured to fail the build process and only produce a warning. Results show that most of the broken builds are related to issues with adhering to coding standards. Checks related to likely defects rarely make the build fail and instead produce a warning, possibly because of the high number of false positives.

McGraw and Sultanow [33, 34] show in their work that the results which traditional static analysis tooling produce are largely dependent on the quality of the provided ruleset. This ruleset typically consists of patterns that the tool matches against when performing the analysis. Therefore, if the ruleset is inadequate, so will the resulting output of the tool. This is why machine learning solutions which automatically learn defects are sought after, they better capture the nuances of code that predefined rulesets can struggle with [34].

There have been previous attempts at applying machine learning techniques to static analysis tooling. Sultanow et al. [34] put together a prototype for a static analysis tool that attempts to detect faults in code by applying techniques commonly used for finding patterns in shopping carts. The core idea of the study was to consider code instructions such as method calls and variable declarations as items. The study abstracts away the naming of variables and prepares all code in a codebase to result in an attribute-relation file format. This prepared codebase, seen as clean and verified code, is then used to validate new code for violations. The prototype was used in a small experiment where the codebase was built up by code fragments which included *not-null* checks. The codebase was then tested against a new similar code fragment where the *not-null* check was omitted. The tooling showed promising results, and the system correctly suggested completing the code fragment when such a check was deliberately omitted.

3

Background

In this chapter, the background of the thesis is presented. The chapter is structured in sections that cover: (1) The Android Open Source Project, including the peer review process and Gerrit's API; (2) Common Vulnerabilities and Exposures, which were the type of vulnerability records used in this thesis; (3) BERT, Google's Natural Language Processing Technique; (4) the Nearest Neighbours ball tree algorithm and Minkowski Distance used by ACoRA; and (5) the Levenshtein distance, a similarity distance for comparing words by single-character edits.

3.1 Android Open Source Project

The Android Open Source Project (AOSP) was chosen for this thesis partly based on its substantial size and rich code review history. AOSP repository, led by Google, offers the information and source code needed to create custom variants of the Android OS, port devices, and accessories to the Android platform[1]. The AOSP, in its current form of this thesis, consists of 1,982 repositories, where most of the code is written in either Java, C, or C++.

3.1.1 Android Peer Review Process

Android makes use of the approach of using git that serves as a barrier between developers' private repositories and the official Android source tree [35]. Developers make changes on their local version and then submit the changes for review, where all comments have to be resolved before the changes can be merged. Android is also an example of a review-then-commit policy that has the following additional change approval steps [35]:

1. Verified - Before the review begins, there is a requirement that the changes are verified in order not to break the current master build. This step can be done manually but is most often done automatically.

2. Approved - Anyone can then comment on the change, but someone with appropriate expertise and approval must approve the changes.

[1] `https://source.android.com/`

3. Submitted/Merged - The change has been approved and merged into Google's master branch, where other developers now can fetch the latest version.

During the approval steps mentioned above, the reviewers can place votes and approve the proposed changes. These votes can vary in the span of -2 to +2 and mean:

- -2, This shall not be merged.
- -1, I would prefer this is not merged as is.
- 0, No score.
- +1, Looks good to me, but someone else must approve.
- +2, Looks good to me, approved.

In this thesis, code reviews that could potentially point out security issues are of the highest interest, which leads to the collection of reviews with votes of -1 or -2, as those have the greatest chance of containing security issues.

3.1.2 Gerrit's API

An essential aspect of tooling supporting code reviews in open source projects nowadays is making the data available for collection and analysis. One of the tools supporting the collection of data from open source projects is Gerrit[2] used by Android.

Gerrit comes with a REST-like API available over HTTP, which this thesis uses to collect code reviews. However, the API is also suitable for automated tools to build upon and supports some ad-hoc scripting use cases. Gerrit's REST-like API endpoints[3] are:

/access/ - are endpoints related to the access rights for projects.

/accounts/ - are endpoints related to accounts, for example to get out account information for an specific email address.

/changes/ - are endpoints related to changes and their information within a repository.

/config/ - are endpoints related to configurations, for example to get a gerrit server information.

/groups/ - are endpoints related to group information, for example to get information about administrators, project users, etc.

/plugins/ - are endpoints related to plugins, for example to retrieve information about the plugins a specific server uses.

/projects/ - are endpoints that describes projects.

/documentation/ - are endpoints that describes the documentation search related tasks.

[2] `https://www.gerritcodereview.com/`

[3] `https://gerrit-review.googlesource.com/Documentation/rest-api.html`

Data collection and analysis can be done with different purposes for each endpoint. However, in this study, the **/changes/** endpoint was mainly used as the code, and the reviewers' comments on changes were of interest to find security related issues.

3.2 Common Vulnerabilities and Exposures

This thesis intends to find security defects using an NLP model and existing vulnerability data. The conquest of finding vulnerability data through a literature search, described further in Section 5.3, resulted in the collection of CVE records. The abbreviation stands for Common Vulnerabilities and Exposures, and these records are instances of a bigger taxonomy of Common Weakness Enumerations (CWE). It is MITRE[4], a government-funded organization that puts out these standards to be used by the information security community. The CWE taxonomy can be seen as the broader perspective of weaknesses and can be described as:

- **CWE**[5] - *is a particular software or hardware weakness that has a security consequence. Weaknesses in this case are flaws, faults, bugs or other errors that could be found in software or hardware. To make it clear with CWE is that it has to do with the vulnerability - not the instance within a product or system.*

However, it is not the overall and broader perspective that was of interest for this thesis but rather specific instances of vulnerabilities. This is where the more concrete instances of vulnerabilities in the form of CVE records come into the picture. The records can be described as:

- **CVE**[6] - *refers to a specific instance of a vulnerability within a product or system. The difference is that if CWE is the dictionary of software vulnerabilities, then CVE is a list of known instances of vulnerabilities for known products and systems.*

How an example of a CVE record, used during this thesis, looks like can be seen in Table 1. One can see in this example that it includes an identifier, a description of the instance, and references to documentation or code. These references, if they point to a diff within the AOSP, will later be used to extract vulnerable code.

Table 3.1: An example of a CVE record in the domain of AOSP.

CVE	CVE-2016-3760
Description	Bluetooth in Android 5.0.x before 5.0.2, 5.1.x before 5.1.1, and 6.x before 2016-07-01 allows local users to gain privileges by establishing a pairing that remains present during a session of the primary user, aka internal bug 27410683.

References	`https://android.googlesource.com/platform/hardware/libhardware/+/8b3d5a64c3c8d010ad4517f652731f09107ae9c5`
	`https://android.googlesource.com/platform/packages/apps/Bluetooth/+/122feb9a0b04290f55183ff2f0384c6c53756bd8`
	`https://android.googlesource.com/platform/system/bt/+/37c88107679d36c419572732b4af6e18bb2f7dce`
	...

[4] `https://www.mitre.org/`
[5] `https://cwe.mitre.org/`
[6] `https://cve.mitre.org/`

3.3 BERT

This thesis's underlying machine learning technique is Google's Bidirectional Encoder Representation from Transformers, BERT. The machine learning technique was developed to understand users' search queries even better than before. Unlike Recurrent Neural Networks (RNNs), which are bound to learn either left-to-right (LTR) or right-to-left (RTL) with hidden states, BERT is designed to pre-train deep bidirectional representations from unlabeled text by jointly conditioning on both left and right context in all layers [11]. A BERT model is trained in two phases: (1) unsupervised pre-training to learn the linguistics of a language; and (2) supervised fine-tuning, where the model's parameters are fine-tuned for specific NLP tasks on labeled data. In this study, when using BERT, only the first training step is performed. It is at that point the model gains the ability to compare code fragments using its linguistic capabilities.

The following sections will describe: (1) how BERT takes advantage of the encoder in the Transformers Architecture; and (2) how the model is pre-trained using Masked Language Modeling and Next Sentence Prediction.

3.3.1 BERT's Architecture

A transformer, which BERT is built upon, is a deep learning model that adopts the mechanism of attention, which in other words, is weighing the influence of different parts of the input data. Like RNNs, transformers are designed to handle sequential input data. However, unlike RNNs with hidden states, the transformers do not require that the sequential data is processed in order. Instead, the attention mechanism provides context for any position in the input sequence.

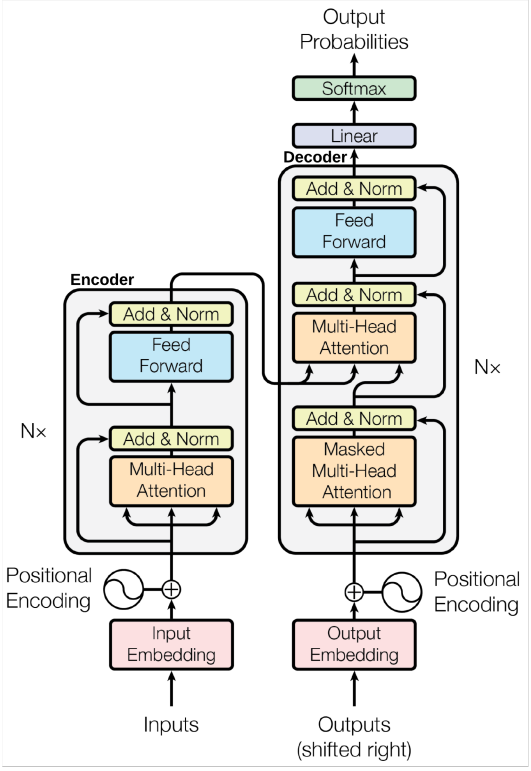

Figure 3.1: The Transformer - Model Architecture [2].

The transformers architecture, in Figure 3.1 shows the encoder structure to the left and the decoder structure to the right. The encoders' purpose is to convert information from an external format to an internal format. On the other hand, the decoders' purpose is to process the internal format and decode it back into a valid format outside of the model, e.g. translating the internal format of text into a new language.

The BERT architecture is simple yet powerful. It only uses the encoder part of a Transformer and stacks them on top of each other to get a deep internal bidirectional representation of data. This stack of encoders in BERT is pre-trained on an ample amount of unlabeled data to obtain a robust internal representation. This internal representation can then be easily fined-tuned for different purposes by adding an extra layer on top of the stack. The following section will go over the two unsupervised training tasks used during the pre-training stage. More information on the second step, where BERT is fine-tuned for specific NLP tasks, can be found in work by Devlin [11].

3.3.2 Pre-training BERT

During the pre-training, the model is trained on large amounts of unlabeled data using two pre-training tasks, Masked Language Modeling and Next Sentence Prediction, described in the following sub-sections.

3.3.2.1 Masked Language Modeling

The first task, being Masked Language Modeling (MLM), is used to train a deep bidirectional representation and is done simply by masking some percentage of the input tokens at random. The masked tokens are then predicted and learned from. Standard conditional language models train LTR or RTL with hidden states, but BERT, being bidirectional, needs masking as it takes the whole sequences at once. Furthermore, if masking is omitted during BERT's training, it would be possible for each word to "see itself" which would lead the model to only learn examples of data instead of its properties and structure.

For performance reasons, the training data generator chooses 15% of the token positions at random for prediction. If the i-th position is chosen, it will replace the i-th token with:

1. the "[MASK]" token 80% of the time;
2. a random token 10% of the time; and
3. leave the i-th token unchanged 10% of the time.

The reason for not using the "[MASK]" token all of the time is the mismatch between pre-training and fine-tuning since the "[MASK]" token does not appear during fine-tuning for specific tasks [11].

3.3.2.2 Next Sentence Prediction

The second task, Next Sentence Prediction (NSP), is used to train the model to understand sentence relationships. The task is essential as many supervised problems such as question answering and natural language interference are based on understanding the relationship between sentences. BERT pre-trains with a binarized NSP task which can be used for any monolingual corpus. The training data for BERT is created and retrieved as examples of two pairs of sentences, A and B. The pairs are binarized and constructed as follows [11]:

1. In 50% of the training examples sentence B is the actual next sentence that follows A (labeled as "IsNext") and;

2. 50% of the time is a random sentence (labled as "NotNext") from the corpus.

With the binarized data described above, the model's task is to predict if sentence B is the sentence that follows sentence A.

3.4 Nearest Neighbours

The underlying implementation of comparing code in ACoRA is based on a proximity search called Nearest Neighbours. As a form of a proximity search, the optimization problem is finding a point in a given set that is closest (or most similar) to a given point. Closeness is typically expressed in terms of a dissimilarity func-

tion. The less similar the objects (points/vectors), the larger the function values. In this study, the points compared are the code embeddings generated by an unsupervised trained BERT model. The following subsections will describe: (1) the Nearest Neighbour ball tree algorithm used within ACoRA; and (2) the Minkowski Distance, a similarity measurement for generated embeddings.

3.4.1 Ball Tree Algorithm

Many tasks in vision, speech, graphics and machine learning require the construction and manipulation of geometric presentations. Systems, such as BERT, build these representations by learning from examples and can be both flexible and robust. A *ball tree* is a data structure which is well suited for geometric learning tasks for its ability to tune itself to the structure of the represented data, for handling high-dimensional entities, and still have a good average-case efficiency [36]. This data structure is used by the tools within this study to actually perform the task of comparing fragments of code based on their similarity.

S. Omohundro [36] describes the ball tree algorithm in its simplest form as:

> "It is an off-line top down algorithm. By this we mean that all of the leaf balls must be available before construction and that the tree is constructed from the root down to the leaves. At each stage the algorithm splits the leaf balls into 2 sets from which ball trees are recursively built. These trees become the left and right children of the root node. The balls are split by choosing a dimension and a value and splitting the balls into those whose center has a coordinate in the given dimension which is less than the given value and those in which it is greater. The dimension to split on is chosen to be the one in which the balls are most extended and the splitting value is chosen to be the median."

An illustration of how the *ball tree* structure would tune itself to capture similar data points can be seen in Figure 3.2.

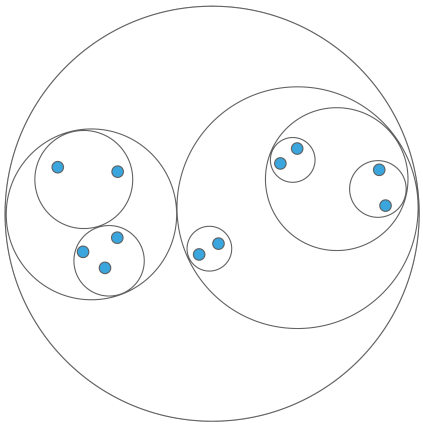

Figure 3.2: Example of how the ball tree algorithm could capture groups of similar data points.

3.4.2 Minkowski Distance

The Minkowski distance is a similarity measurement between two points in a normed vector space and is a generalization of the Euclidean and Manhattan distance. This distance is used by the tools in this study to numerically express the similarity of two code fragments found by the *ball tree* algorithm. A Minkowski value of zero means that two fragments of code are identical, the bigger the value the more different the fragments are according to the tool.

The Minkowski distance of order p (where p is an integer) between two points:

$$X = (x_1, x_2, ...) \; and \; Y = (y_1, y_2, ...)$$

is defined as:

$$D(X, Y) = \left(\sum_{i=0}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

The two special cases for p is 1, which equals the Manhattan distance, and 2, which equals the Euclidean distance.

3.5 Levenshtein Distance

The Levenshtein distance is a straightforward non-machine learning approach to measuring the difference between two fragments of text. The purpose of this algorithm in this study is to serve as a comparison point to the more complicated machine learning tooling. This comparison is important because if the two methods were in full agreement on what fragments are similar or not, there would be no reason to use the much more complicated machine learning solution.

The Levenshtein distance bases its similarity between two fragments by counting the minimum number of single-character edits, insertions, deletions, or substitutions required to change one word into the other. The complete formula comparing between two strings, a and b, is given by $lev(a, b)$ where:

$$lev(a,b) = \begin{cases} max(i,j) & if \; min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & otherwise. \end{cases}$$

The method essentially makes a matrix recursively and takes the shortest path from one word to the other. It should be noted that there may be multiple paths giving the same Levenshtein distance. Table 3.2 demonstrates the matrix created by comparing the word *int* against *bool*:

Table 3.2: Levenshtein matrix comparing the words *int* and *bool*.

		i	**n**	**t**
	0	1	2	3
b	1	1	2	3
o	2	2	2	3
o	3	3	3	3
l	4	4	4	4

One can see above different paths, going from the top left corner to the lower right corner, giving the same Levenshtein distance of 4 as there are multiple ways of making one deletion and three substitutions to go from the word *int* to *bool*.

4

Implementation of SeCoRA

SeCoRA, the primary artifact of this study, is an extension of the already existing ACoRA toolchain. This chapter first explains ACoRA and its limitations. Then, an explanation is given on how SeCoRA intends to solve those limitations. Lastly, a detailed overview of SeCoRA's architecture is shown.

4.1 What is ACoRA?

The core of this study is the tool ACoRA, the Automatic Code Review Assistant, a closed source project created by Mirosław Ochodek, Ph.D., and Miroslaw Staron, Ph.D. To be specific, ACoRA is not a single piece of software but a command line toolset intended to facilitate the development and use of code review related BERT models. The ACoRA toolset helps its user with the following:

- Gathering review data from Gerrit repositories.
- Training the underlying BERT models with code.
- Finding similar lines of code when provided with a sample.
- Classifying code review comments with a provided taxonomy.
- Highlighting potentially problematic code with hints concerning the potential issue.

ACoRA's workflow consists of the following activities as shown in Figure 4.1:

1. ACoRA detects a line that needs the reviewer's attention.
2. ACoRA looks for similar lines that have been commented on. It prepares a summary for the reviewer about those comments - what was the most frequent purpose of the comments (e.g., reviewers requested some changes or asked a question) and what were the subjects of the comment (e.g., code logic, naming, code style, defective code).
3. The reviewer uses the provided information, it helps the reviewer decide if they should comment on the line. That is, if the reviewer sees that the detected line of code adheres to the same problems as the provided information points to.

4. The comment and the commented line are stored in the code reviews database. The comment's purpose and subjects are predicted using a classifier.
5. The information in the code reviews database can be used to train, or re-train, the classification models. In the case of comments, they should be manually classified with a subject and purpose (or at least, the reviewer should act as an oracle for the automated classifications). For the code lines, assembling a new training dataset can be done automatically since we know which lines were commented on.

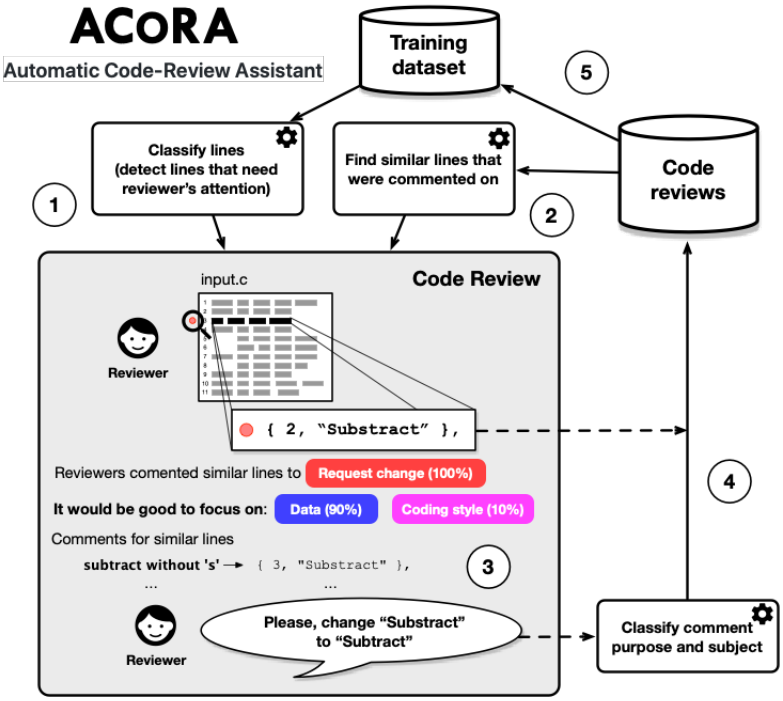

Figure 4.1: ACoRA, an Automatic Code Review Assistant.

Appendix A.1 visualizes the process of training and using ACoRA and the individual scripts within it. The following is a short description of each step:

- **Pretraining BERT4Code**† - This set of scripts takes code from a directory or a repository; converts it to lines in JSON format; creates a vocabulary out of the lines; creates code line pairs that the model can train and predict on; and lastly trains and outputs a "bert4code.h5" file containing the final model.

- **Generate code-embeddings**† - is a script that takes in lines that should be converted to BERT embeddings. The conversion is done with the help of the trained code model and the vocabulary used to train it. These embeddings are used when comparing lines of code.

- **Find similar lines**† - is a script used to compare embeddings from **Generate code-embeddings**. The script takes two types of data: (1) query data; and

†Scripts marked with this symbol were used during this study, the remaining scrips are only desicrbed to explain the capablities of the tool.

(2) data to be queried. The script is able to compare the embeddings from the two types of data and returns a file of found similar lines with a corresponding similarity distance. This is the main script used throughout the study when searching for code based on similarity.

- **Train BERT4Comments** - is a set of scripts used to classify comments from Gerrit. The first run of the scripts will be manually intensive as it requires the user to label the purpose and subject of the comments. However, after the first iteration, the script only asks for manual intervention in high uncertainty cases and automatically labels the rest.

- **Train BERT4CommentedLines** - is a set of scripts that creates a new model for commented lines of code. The model trained on code and the model trained on comments are used here to train a new model on what type of comments was put on specific code.

- **Generate recommendations** - is the last part of ACoRA, which includes the database of examples, the vocabulary used for the code models, the code model that can find problematic code, and the model that is trained to apprehend which comments are common with problematic code. The recommendations can be generated as a batch or in an interactive manner using a web-interface.

4.1.1 Limitations of ACoRA

As with most machine learning solutions, an ample amount of data is needed to train the model sufficiently. ACoRA's training consists of the following steps:

1. **Pretrainig BERT4Code**† is unsupervised training on code fragments with Masked Language Modeling and Next Sentence Prediction;
2. **Train BERT4Comments** is supervised training where a model learns from labeled data to classify a comments subjects and purpose; and
3. **Train BERT4CommentedLines** is supervised training where a model learns to classify what type of comments should be placed on specific code.

Gathering data for ACoRA's first training step is straightforward as code from any software repository of substantial size can be used. On the other hand, the process is much more complicated during the second and third steps, where labeled data is need during the supervised training. When given a project code review history, there is no clear indicator if a review relates to a particular type of security defect. The substantial amount of effort needed to acquire such data is a significant limitation on ACoRA's applicability to an actual real-world project. These limitations bring forward the question which drives this study; *Given the code review history of a software repository, how does one discern which reviews exemplify specific security defects?*.

4.2 What is SeCoRA?

SeCoRA is a command line toolset designed as part of this study specifically to address the limitations mentioned earlier and attempts to automate the process of finding security related code reviews. The tool is built on top of ACoRA and makes use of the simpler unsupervised BERT model. Specifically, SeCoRA uses this model to compare two code fragments against each other and express how similar they are by outputting a Minkowski distance.

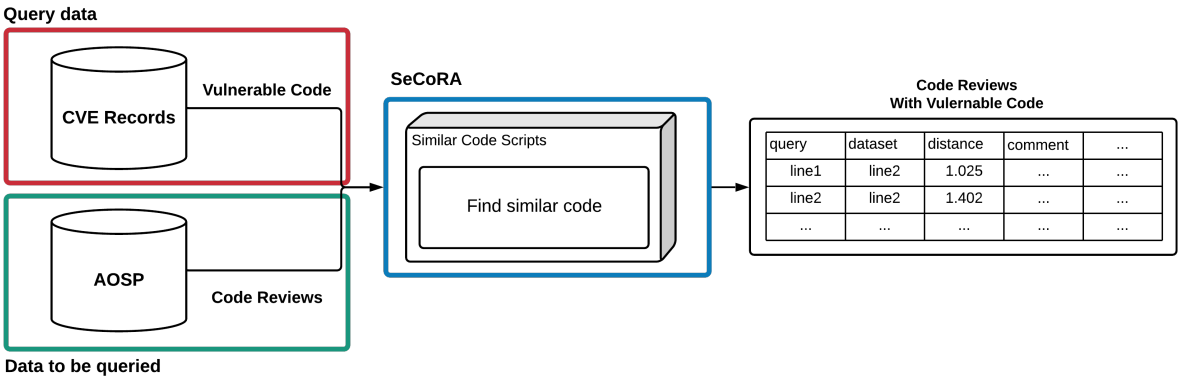

Figure 4.2: Illustration of how to use CVE records to find code reviews with vulnerable code with the help of SeCoRA.

Figure 8.1 shows how SeCoRA uses this ability to compare code to figure out which code reviews relate to security and which don't. The tool takes code fragments which are known to contain vulnerable code and uses them to query AOSP code reviews for similar code. The tool operates on the assumption that if a code review contains code which is similar to known vulnerability, then there is a high probability that the review itself is security related. Furthermore, each entry in the query CVE data has been classified according to a security taxonomy which specifies the type of the underlying security defect (Pointer issues, Improper input validation etc.). This classification should then also apply to any code which the tool has expressed as similar.

The study implements and evaluates SeCoRA on how well it solves the problem of acquiring the necessary training data. The colors used through Figure 8.1 serve as a reference for the reader in future chapters.

4.3 Architecture

The following section describes in detail SeCoRA's architecture as shown in Figure 4.3. The tool is built on top of ACoRA and is responsible for not only comparing code fragments but also gathering all the data from the specific CVE database and AOSP. The bottom layer of the stack shows ACoRA itself. Each inner box within the software layers represents a script, and together they form a toolchain. Scripts

which were not used during the thesis have been grayed out.

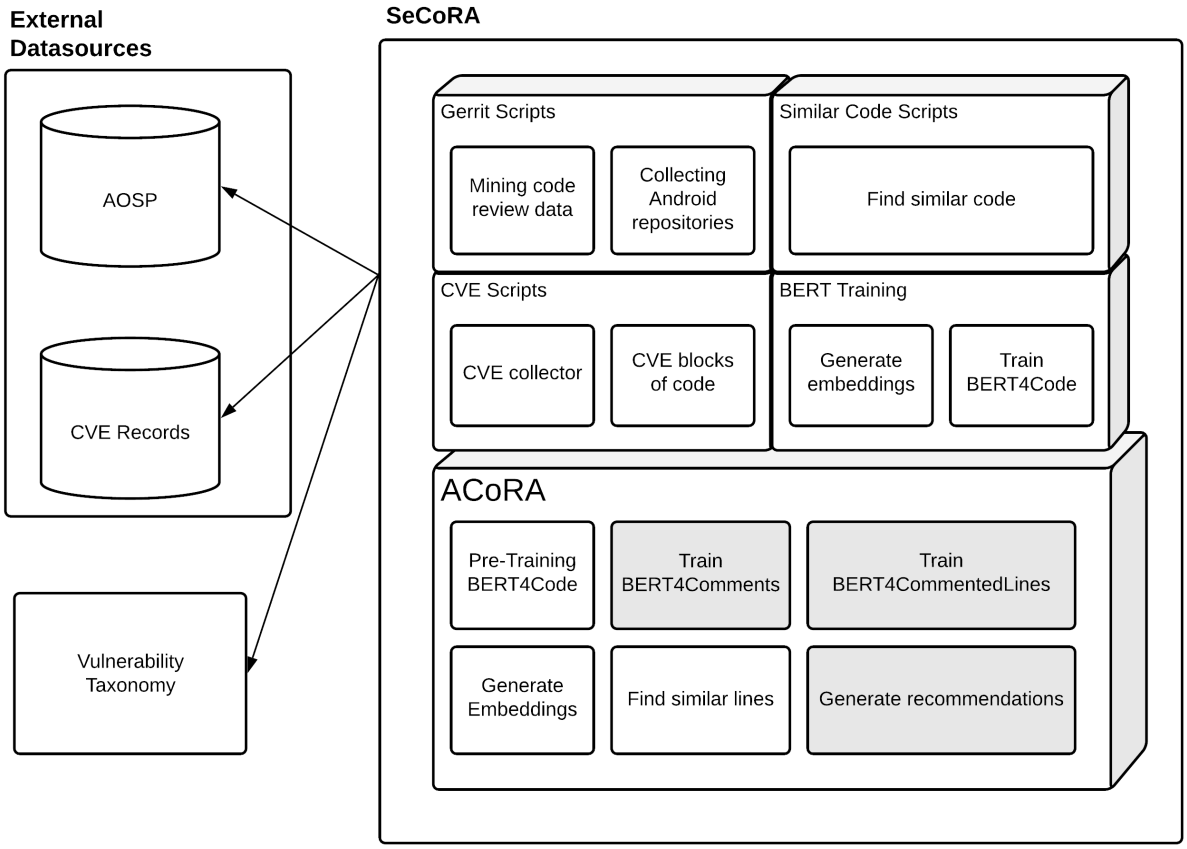

Figure 4.3: Illustration of how to use CVE records to find code reviews with vulnerable code.

4.3.1 Data Acquisition

The left-hand side of Figure 4.3 shows the two external datasources which SeCoRA relies on. From these two data sources, SeCoRA depends on three types of data:

- Code repositories that can be used to pre-train SeCoRA on code (AOSP).
- Samples of security vulnerabilities that can be used as query data (CVE Records); and
- Archive of previously performed code reviews that will be queried for security vulnerabilities (AOSP).

The scripts made to collect data from these datasources and the additional security taxonomy are described in the following subsections.

4.3.1.1 Gerrit Scripts

The mining procedure is done with the *mining code review data* script written specifically for this purpose. The script makes use of the "python-gerrit-api"[1] library which simplifies the process of interacting with Gerrit and queries Gerrit's **/changes/** endpoint for a list of recently submitted changes. Each change is then inspected, only those changes where the review resulted with either a -1 or -2 rating are considered. If the code has received a negative rating from the reviewer, the possibility that it contains a security related defect increases. Upon detecting a negative review, the script then collects all the comments left by the reviewer, corresponding code, and some metadata. For reference, this data is marked with the color green in Figure 8.1. It should be mentioned that the AOSP limits the amount of data shown at a given point to a maximum of 2,500 latest entries.

Collecting Android repositories, the second script acquires code from AOSP, which is used to train SeCoRA's underlying BERT implementation to understand programming syntax. The script essentially web scrapes links to Android projects and clones them, so they are available locally.

4.3.1.2 CVE Scripts

The CVE data marked with the red color in Figure 8.1 needs to be filtered and processed before it can be given to SeCoRA. Each CVE record typically has a link to the corresponding patch, which resolves the underlying vulnerability. These records have to be filtered as not all of the links point to the AOSP. This filtering is done by the *CVE Collector* script, which simply checks the beginning of the url within each CVE record and makes sure it belongs to AOSP. In addition, the script is also capable to further filter the CVE records based on the programming language used within the patch. This makes it possible to specifically collect only those CVE records with code written in for example, Java.

The filtered CVEs are then passed into the *CVE Blocks of code* script, which iterates over the AOSP links and collects the code. Each link points to a diff showing the before and after of a fixed vulnerability. An example is shown in Figure 4.4, the red code represents the line before the fix, and the green is after. Only the red lines are collected as that is the vulnerable code.

```
@@ -501,7 +501,7 @@
        intent.putExtra("STK CMD", cmdMsg);
        intent.putExtra("SLOT_ID", mSlotId);
        CatLog.d(this, "Sending CmdMsg: " + cmdMsg+ " on slotid:" + mSlotId);
-       mContext.sendBroadcast(intent);
+       mContext.sendBroadcast(intent, AppInterface.STK_PERMISSION);
    }

    /**
```

Figure 4.4: Part of a diff taken from a security vulnerability fix, the vulnerability is part of the CVE database.

[1] https://pypi.org/project/python-gerrit-api/

Additionally, the user can specify if she wants to capture any surrounding lines with a **range** parameter. If the range is set to **0**, only single lines are captured, and when set to **1**, it will also capture the line before and after as shown in Figure 4.4.

```
CatLog.d(this, "Sending CmdMsg: " + cmdMsg+ " on slotid:" + mSlotId);
mContext.sendBroadcast(intent);
}
```

And if the **range** parameter would have been set to **2**, it would have created a block from plus/minus two lines of the "deleted line" in the diff, and so forth.

The format of the final resulting data can be seen in Table 4.1.

Table 4.1: The format of records of vulnerable code from CVEs.

Vulnerability Type	AOSP Patch Link	Vulnerable code

4.3.1.3 Taxonomy

This taxonomy divides the CVE's into distinct types and categories based on the type of the underlying vulnerability, such as pointer issues, improper input validation, etc. This classification is performed manually on the CVE records before they are given to the CVE mining scripts. This classification ends up in the final results of the mining and is shown as *Vulnerability type* in Table 4.1.

SeCoRA later uses this field to label any newly found vulnerabilities based on the label set on the code used to find them. So, for example, if a CVE with the type *pointer issues* was used to locate a code review, then that review, in theory, can be labeled with the same type.

Creating such a taxonomy requires extensive knowledge or substantial time and research into a particular domain. Fortunately, previous work has been done on security taxonomies within AOSP[1, 37], which can be reused for this project.

4.3.2 BERT Training

These scripts are responsible for parsing a codebase and using the data to train a BERT model to understand the syntax and semantics of the programming language used within the provided code. The original training scripts within ACoRA were built to break down a codebase into individual single lines. This means that the resulting model is only capable on working with input which is of similar length, a single line. SeCoRA extends these capabilities by making the script able to process and learn code fragments longer than a single line. This capability will be described further in future chapters.

4.3.3 Find similar code

Using the color blue, Figure 8.1 shows how this script performs the main job of SeCoRA, which is comparing fragments of code against each other. This script uses the ball tree algorithm to perform this comparison and outputs a Minkowski distance for each two compared code fragments. The higher the value, the less similar two fragments are to each other, and zero means that they are completely identical. In the case of SeCoRA, this means that any comparison with a low Minkowski distance implies that both of the fragments in the comparison are instances of security vulnerabilities. This comparison script is identical to the one used by ACoRA. The only changes that have been made are small extensions that make it possible to compare fragments of varying size, as the original version was built to only compare single lines.

This machine learning-based comparison solution is interesting because it doesn't simply count how many characters or words are different, like a simpler solution, the Levenshtein distance would. The underlying BERT model learns the actual syntax and semantics of the language instead. It is able to compare code fragments on a more high abstract level and, in theory, should perform better than non-machine learning approaches.

4.3.4 Toolchain Sequence

The scripts described above comprise the overall SeCoRA workflow. A complete flow diagram is shown in Appendix A.2 and can be used as a reference when reading through the workflow steps:

- The **Collecting Android Repository** script acquires and processes AOSP Repositories.
- The code is then given to the **Train BERT4Code** script which trains the underlying BERT implementation.
- The performance of the model is evaluated before moving to the next step, if needed the training step is repeated.
- The **CVE collector** gathers classified vulnerabilities and links to their corresponding code.
- In turn, **CVE blocks of code** iterates over the links and collects the corresponding code.
- The **Mining code review data** script queries Gerrit for negative code reviews and the corresponding data.
- The CVE and code review data are pre-processed using **Generate embeddings**
- **Find similar code** uses the CVE data to query the review data and returns any similar lines.
- The results are evaluated.

5

Methodology

5.1 DSR

In order to construct the new complementary treatment design to ACoRA, this thesis followed the design science research (DSR) methodology. In design science, the study partially follows the engineering cycle. It is called a cycle because the research is conducted iteratively, which means that all steps are performed multiple times until the objective is reached or the time has run out. The iterative process of the design cycle and where it deviates from the engineering cycle can be seen in Figure 5.1.

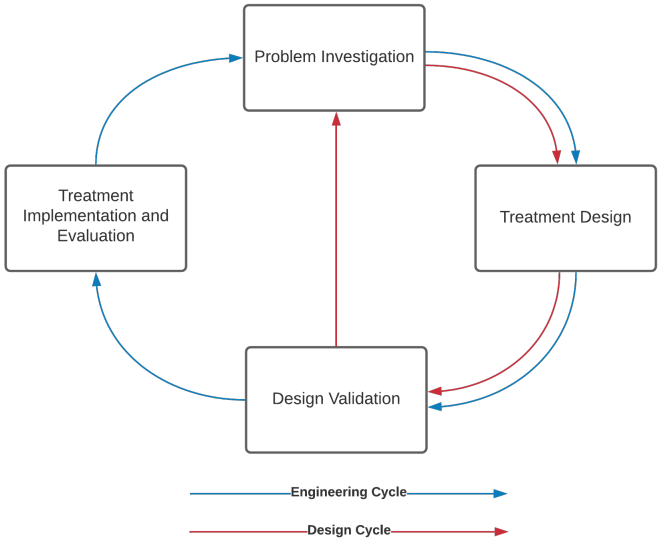

Figure 5.1: The Engineering and Design Cycle.

The steps in the engineering cycle can be described as [38]:

- **Problem investigation** - What phenomena must be improved? Why?
- **Treatment Design** - Design one or more artifacts that could treat the problem.
- **Design Validation** - Would these designs treat the problem?

- **Treatment Implementation & Evaluation**:
  - **Treatment Implementation** - Treat the problem with one of the designed artifacts.
  - **Treatment Evaluation** - How successful has the treatment been? This may be the start of a new iteration through the engineering cycle.

The design cycle deviates from the engineering cycle and rather focuses on if the artifact and design could treat the problem. In short, to design a treatment, one must understand the problem to justify the choice for a treatment, and one must validate it before it is implemented. In contrast, the engineering cycle tells us also that to learn from an implementation, one must evaluate it in the intended environment [38].

This thesis focused on the design cycle, investigating if the artifact and design could treat the problem of discerning and collecting security related code reviews. Although the study was performed iteratively, in order to aid readability, the authors opted to structure the text in a sequential form. Since the study results would remain consistent no matter if the steps were performed sequentially or iteratively, the detailed description of each iteration is omitted.

The remainder of this chapter will cover how the new treatment design was trained and how it was validated to know if the design treats the problem.

5.2 Distinguishing between Code Fragments

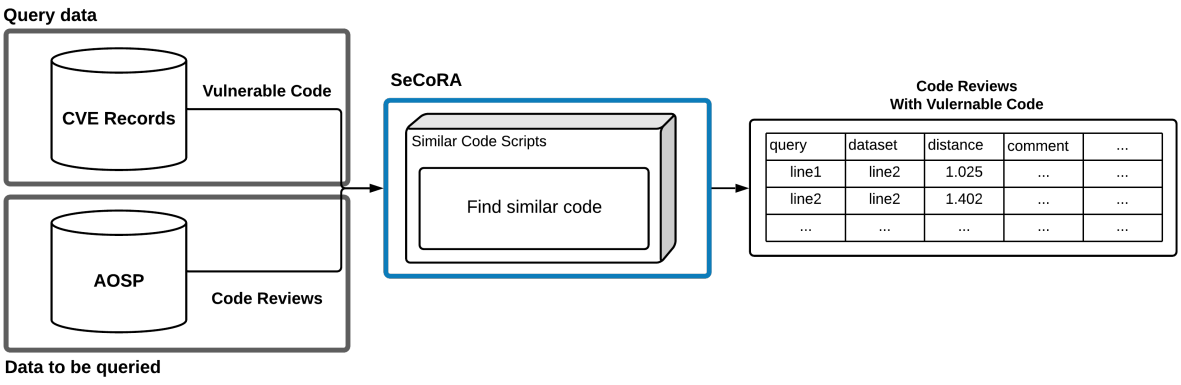

Figure 5.2: Illustration of how to use CVE records to find code reviews with vulnerable code.

Before SeCoRA could be used for finding vulnerabilities, it was essential to answer **RQ1** and understand to what degree the tool is able to distinguish between different arbitrary code fragments. If the tool could not distinguish between arbitrary code, it wouldn't be able to do it for security vulnerabilities either.

For this purpose, two unsupervised models were trained. These are the BERT models which SeCoRA uses for finding similar code as shown using the color blue in Figure 5.2. In order to understand how the size of the code fragments affects the results of the code comparison, the first model was trained on single lines of code while the second on larger code fragments.

The answer to **RQ1** can then be formulated by evaluating those two models on their ability to distinguish code fragments. The evaluation was done by using the model to compare various *if statements*, *for loops* and other code fragments against each other. The general idea is that a working model should be able to express that *if statements* are more similar to other *if statements* than they are to e.g *for loops*.

Because one of the models was trained on blocks, it should be able to handle input larger than the single line model. Therefore the evaluation process also attempts to assess how this affects the block model's overall ability to discern between code fragments. This was done using different input sizes during the evaluation, specifically: single lines, three lines, and complete statements of varying length.

Lastly, to fully answer **RQ1**, an additional evaluation is performed where a model is compared against an existing code comparison solution, the Levenshtein distance. The model, together with some arbitrary code fragments were used to query a repository for similar code. For each result of this query the Levenshtein distance is calculated to see if a different comparison method agrees on which fragments are similar or not.

This chapter breaks down and describes the overall steps of training and evaluating in the same order as above.

5.2.1 Training models on code

The following sections describe: (1) what data was used during the training; (2) how one model was trained on a single line basis; (3) how the second model was trained on a block basis with a sliding window.

5.2.1.1 Data used during training

The AOSP consists of 1,982 repositories, where most of the code is written in either Java, C, or C++. The authors chose to use a Java repository for training as this was the language in which they were most proficient. If a judgment related to code needed to be made at any point, that judgment would be affected by the experience of the judge with the language in question. The choice of which exact repository would be used was made randomly as there was no clear heuristic on why one repository would be a better choice over the other. If one repository was not sufficient and the models were to require more training data, this step could simply be repeated as additional repositories are always available.

The randomly chosen repository was the "platform/base/tools" project. An overview of the repository can be found in Table 5.1.

Table 5.1: Overview of the "platform/tools/base" repository.

Repository	platform/tools/base
Link	https://android.googlesource.com/platform/tools/base/
Size	1.88 GB
Top 5 file extensions	.java (4,439), .kt (3,316), .xml (1,611), .png (1,244), .aapt (501)
Total amount of files	14,182

Before using the repository for training, the code was cleared out from any comments with the help of a regex based script. The decision to remove the comments was based on the notion that they were not something that the model should learn as they cannot possibly be a source of a security vulnerability.

5.2.1.2 Training the single line model

The training was done using the unsupervised Masked Language Modeling and Next Sentence Prediction tasks, which are described in Section 3.3. The first model was trained using single lines of code using the data present in the Java files found in the repository shown in Table 5.1. SeCoRA extracted 1,263,839 individual lines of Java code from the 4,439 Java files that existed in the repository. The extracted lines were then made into 577,956 code-line pairs that the model would train on with its two unsupervised tasks, MLM and NSP. All settings used to train the model can be found in Appendix B.1. The model was set to train for at least 20 epochs or until the loss function stagnated.

5.2.1.3 Training the block model

The second model was trained using the same unsupervised methods, but instead of single lines, bigger code fragments were used, gathered with a sliding window technique. A sliding window is a technique where a window is formed over some part of the data. This window can incrementally slide to capture different portions of it. In this case, the window was used to capture a set amount of tokens referred to as a block.

The following was done to process the training data before the sliding window was used to create blocks:

- removal of empty lines,
- removal of starting and ending white-spaces, and,
- joining all lines on white-space.

Samples of java code, before and after processing are shown in Figure 5.3.

```
1 /**
2  * Tracks a pending task that has been scheduled on the associated Executor.
3  */
4 private abstract static class PendingTask implements Runnable {
5
6     private ScheduledFuture<?> mFuture;
7
8     public void scheduleDelayed(ScheduledExecutorService executor, int timeMs) {
9         mFuture = executor.schedule(this, timeMs, TimeUnit.MILLISECONDS);
10    }
11
12    public boolean isDone() {
13        return mFuture != null && mFuture.isDone();
14    }
15
16    public void cancel() {
17        if (mFuture == null) return;
18        mFuture.cancel(false /*interrupt*/);
19    }
20 }
21
22 private abstract static class PendingTask implements Runnable { private ScheduledFuture<?> mFuture; public void
   scheduleDelayed(ScheduledExecutorService executor, int timeMs) { mFuture = executor.schedule(this, timeMs,
   TimeUnit.MILLISECONDS); } public boolean isDone() { return mFuture != null && mFuture.isDone(); } public void
   cancel() { if (mFuture == null) return; mFuture.cancel(false ); } }
```

Figure 5.3: Example of how code is prepared for the sliding window technique.

In Figure 5.3 code before processing is shown on lines 1-20, and line 22 shows the code after processing has been applied. The example also shows how the multi-line comment, lines 1-3, and the in-line comment, on line 18, are excluded from the final result.

Wang et al. [39] showed in their study that splitting articles into passages with the length of 100 words when sliding a window brought 4% improvement for their specific task. As such, 100 tokens were selected as the initial starting size. However, with such a large window size, 40% of the files resulted in a single block. The specific length could not be used to train the model as one of the unsupervised tasks during training is Next Sentence Prediction, where one block is not enough. Testing different options and considering that a longer passage is wanted, while trying to avoid too many single passages, resulted in a window length of 32 tokens where the number of single line files decreased to 16%. An example of how a block of code would have been split into fragments of code can be seen in Figure 5.4.

```
1 private abstract static class PendingTask implements Runnable { private ScheduledFuture<?> mFuture; public void
  scheduleDelayed(ScheduledExecutorService executor, int timeMs) { mFuture = executor.schedule(this, timeMs,
  TimeUnit.MILLISECONDS); } public boolean isDone() { return mFuture != null
2
3 && mFuture.isDone(); } public void cancel() { if (mFuture == null) return; mFuture.cancel(false ); } }
```

Figure 5.4: Example of how the code from Figure 5.3 would be split with the token length set to 32.

One can see in Figure 5.4 how the code from Figure 5.3 would have been split where the first passage is on line 1 and the second passage is seen on line 3.

SeCoRA was able to extract 86,060 blocks of code from the 4,439 Java files. The

code fragments were then made into 80,603 code-block pairs that the model could train on. All settings used during training the block model can be found in Appendix B.2. The model was set to train until the loss function stagnated.

5.2.2 Evaluation of the models

The following sections describe: (1) How the models were evaluated by using them to compare samples of code; (2) An additional evaluation step where a model is used to query a repository.

5.2.2.1 Evaluation by code comparison

In order to evaluate the performance of the models in their ability to distinguish between different code fragments, some sort of ground truth was needed. The idea was that, when given fragments of code, the model should express that for example, *if statements* are more similar to other *if statements*, than they are similar to e.g *for loops* or *other code*. For this purpose, multiple datasets were constructed containing code to be used in the evaluation, these were:

- Dataset of *if statements*;
- Dataset of *for loops*; and
- Dataset of *other code*, not containing *if statements* or *for loops*.

Considering that two models were trained on varying sizes of code, either lines or blocks, the evaluation also intended to capture how the size of the fragments used during the evaluation impact the comparison results. As such, the datasets mentioned above were constructed in three sizes: (1) single lines of code; (2) three lines of code; and (3) whole blocks of code. Figure 5.5 illustrated how an *if statement* would have been used to construct three different sized code fragments.

Figure 5.5: Illustration of how an *if statement* was divided during the creation of different datasets.

The datasets were then compared to each other using SeCoRA's ability to fit a Nearest Neighbours model using the Minkowski distance, which is described in Section 3.4.1. A subset of *if statements* containing 10% of the total *if statement* data was extracted. The evaluation process assumes that when comparing those 10% against the remaining 90%, on average, the distance of this comparison should be smaller than if the same 10% were compared against *for loops* or *other code*.

An illustration of comparing code is shown in Figure 5.6 below:

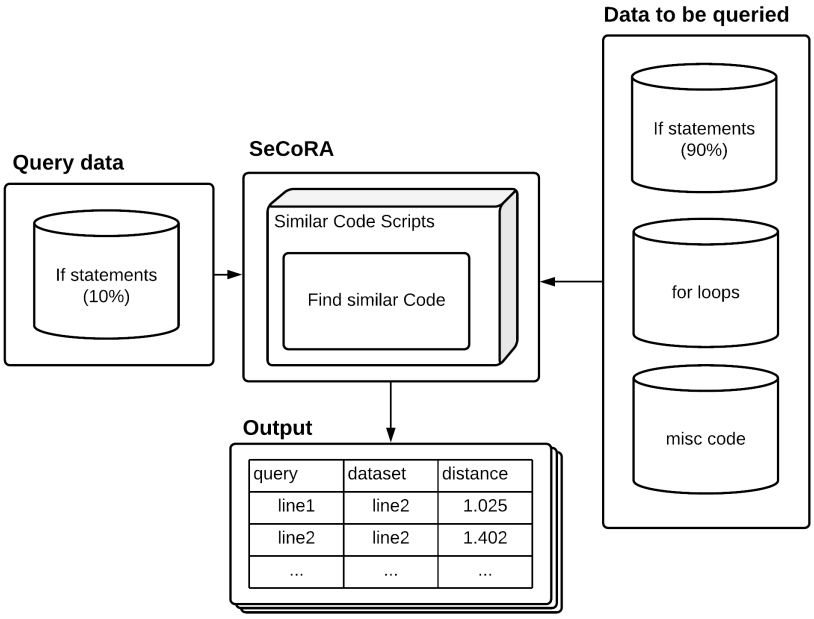

Figure 5.6: Illustration of how SeCoRA was evaluated based on its ability to distinguish different fragments of code.

The last step for the evaluation was to calculate the mean and the standard deviation of the Minkowski distance for each dataset and then plot the results for comparison to assess the results visually.

5.2.2.2 Additional Model Evaluation

In order to further evaluate the degree to which a model can distinguish between different code fragments, an additional evaluation step was performed where the model is put against an existing comparison solution, the Levenshtein distance. This contrast was important because if the model and Levenshtein distance gave the same results, there would be no reason to use SeCoRA as it's vastly more complicated.

For this evaluation, a small set containing just a couple of code fragments was created. The model then uses those fragments to query a repository looking for fragments that it considers similar. For each found fragment, the Levenshtein distance is then calculated. This data is then used to showcase not only what the model

and the Levenshtein method agreed on, but at the same time, point to outliers, examples of fragments that would have been ignored by the Levenshtein method. The following steps outline this evaluation:

1. Use SeCoRA to search for similar lines of code within a repository using artificially made code examples.
2. Calculate Levenshtein distance for the found lines of code.
3. Calculate the mean and standard deviation based on the Levenshtein distance.
4. Calculate the Z-score for the 95th percentile.
5. Find the outliers based on the Z-score.

5.3 Acquiring Security-Labeled Data

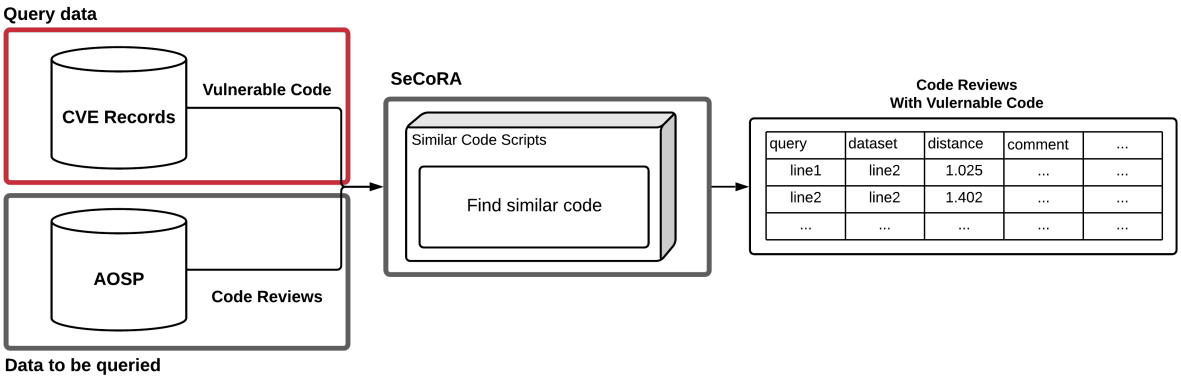

Figure 5.7: Illustration of how to use CVE records to find code reviews with vulnerable code.

For SeCoRA to be able to search for vulnerabilities based on similarity, the tool requires a database of existing known examples of such vulnerabilities to be used as a query. These examples should not only showcase vulnerabilities but also classify them against a taxonomy so that the newly found vulnerabilities can also be classified accordingly. Using a red border, Figure 5.7 shows how this data integrates into the flow of SeCoRA.

Creating this type of database requires extensive security knowledge and a substantial amount of effort and is outside of the scope of this thesis. As such, to gather this data and answer **RQ2** the authors set out to find and use existing databases and research instead.

There were two aspects that required consideration when answering **RQ2**: (1) how do we acquire a taxonomy that could be used to label vulnerable code, and (2) how do we acquire examples of vulnerabilities which are labeled accordingly.

As there has been a noticeable amount of previously performed studies on the topic of security within the area of Android (e.g., [1, 37, 40, 41, 42, 43, 44, 45, 46]), performing a literature search was a promising approach. The idea was that if a paper had a taxonomy, there is also a high chance that the data used to construct it would be available and could be reused for our project. The search was finalized by selecting a data source based on the quantity of the provided classified vulnerabilities.

5.4 Using SeCoRA to find Code Reviews with vulnerabilities

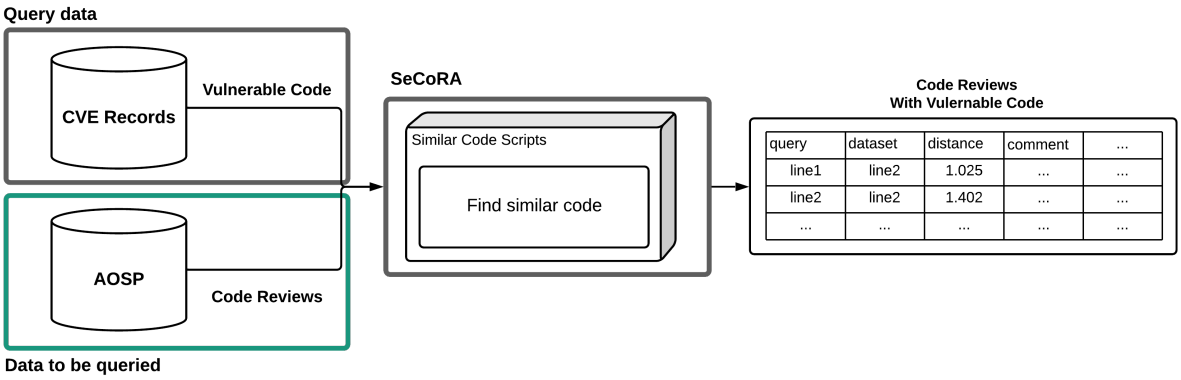

Figure 5.8: Illustration of how to use CVE records to find code reviews with vulnerable code.

With the model acquired when answering **RQ1** and the vulnerability database from **RQ2** SeCoRA can be used to query existing code for vulnerabilities based on similarity.

To assess what vulnerabilities can be found using this approach and to answer **RQ3** the tool was used to query code reviews gathered from the AOSP. As previously mentioned, an assumption is made that, when given code known to be a vulnerability, any similar code is also potentially defective. Using a green border, Figure 5.8 shows how this data integrates into the final flow of SeCoRA. The following sections describe (1) how the code review data was acquired, (2) how SeCoRA, together with the vulnerability database, were used in an attempt to find vulnerabilities, and (3) how the results of the search were evaluated.

5.4.1 Mining AOSP for Code Reviews

Although, in theory, the tool could be used to search for vulnerabilities in any type of code artefact this study focuses specifically on code reviews. This is because of the general goal driving the study, which is acquiring data that can be used to train ACoRA, specifically code reviews where both the code and the corresponding review comment relate to some type of security vulnerability.

These code reviews were gathered through the open Gerrit API of the AOSP. To increase the chance that the collected reviews contain code with vulnerabilities, SeCoRA would only collect those reviews which were marked negatively with either a -1 or -2. As described in Section 3.1.1, a negative vote on a code review expresses a reviewer's position that the changes should not be merged because of some underlying issue, which in this case potentially could be a security vulnerability.

According to the previously described limitations in Section 4.3.1.1, the AOSP restricts the amount of data shown at a given point to a maximum of 2,500 latest entries for any type of query. In addition, out of these 2,500, only a subset will be marked negatively.

The authors felt that this could heavily limit the amount of available data. In order to address this, the data was queried from both open and merged endpoints. The collected data was then filtered for duplications. Figure 5.9 illustrates why this was necessary.

```
57 /*
58 * Reviewer and owner has commented on the code block below.
59 * The comments are shown on the right hand side and span over the
60 * lines 63-65. The comments themselves gets duplicated
61 * on each line of code in the block.
62 */
63 public boolean isEnabled() {            // ["Change to static", "Done"]
64     return PhoneConfigurationManager    // ["Change to static", "Done"]
65             .getInstance()              // ["Change to static", "Done"]
66             .getRadioInterface()
67             .isSupported();
68 }
69
70
71 /*
72 * Reviewer and owner has commented on the code block below also.
73 * The comments are shown on the right hand side and span over the
74 * lines 77-79. The comments themselves gets duplicated
75 * on each line of code in the block here aswell.
76 */
77 public boolean isDisabled() {           // ["Change to static", "Done"]
78     return !PhoneConfigurationManager   // ["Change to static", "Done"]
79             .getInstance()              // ["Change to static", "Done"]
80             .getRadioInterface()
81             .isSupported();
82 }
83
84 /*
85 * One can see above that the code and the comments happends to be
86 * the same for line 65 & 79. The duplication that happens above is
87 * the reason that some examples are filtered out in our scripts.
88 */
89
```

Figure 5.9: Illustration of code and comments that we receive from Gerrit's API which can include duplicates.

Assume the comments in Figure 5.9 are present within the same change, patchset, and in the same file. One can observe that SeCoRA would have collected two occurrences of '.getInstance()', with the exact same comments. The deduplication was performed by transforming all the gathered data into a set where only one instance

of each entry could exist. This example hopefully showcases why no important data would be lost, as only the redundant information is removed.

5.4.2 Searching for Code Reviews with Vulnerabilities

Using the known vulnerabilities found by the literature search in Section 5.3, the code reviews extracted from AOSP, as described in Section 5.4.1, and SeCoRAs ability to find similarities, an attempt was made to find vulnerabilities. Figure 5.10 illustrates this process.

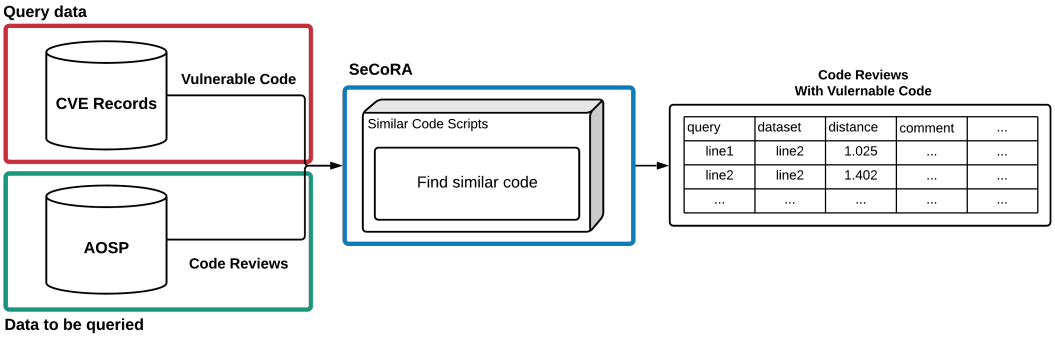

Figure 5.10: Illustration of how SeCoRA was used to search for new vulnerabilities.

SeCoRA compares the similarities between the code from the two different datasets and outputs a Minkowski distance for each found similar code fragment. This process also retains other fields, such as the corresponding review comments, which can later be used to evaluate the results.

Table 5.2 showcases a subset of the fields present in SeCoRAs output. Columns containing metadata have been omitted for brevity and are represented with an ellipsis. Colors used throughout the table's header refer back to Figure 5.10, indicating where the data originates from.

Table 5.2: SeCoRA's output structure, metadata columns have been omitted

...	Query Taxonomy	Query Code	Distance	Found Similar Code	Comment	...
...	Data Handling	if (array.length != 3) {	0.7216	if (ar.exception == null) {	Done	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...

5.4.3 Evaluation of newly found vulnerabilities

In order to assess the results of the search and confirm that the code fragments found by the model are indeed occurrences of vulnerability issues, an evaluation was needed. As shown in Table 5.2, each found code fragment had a corresponding review comment, the content of the comment could be used as a means of confirming

if the underlying code contains a security vulnerability. There are two reasons why the comments are used for evaluation and not the code itself. First, since the authors are not security experts, comments are preferred as they are easier to assess without deep context knowledge. Secondly, as this study aims to acquire training for ACoRA, if the corresponding comment is not security related, the data can't be used for training.

The evaluation consisted of two pass-throughs. Each pass-through consists of the authors manually going over each result, assessing the corresponding comment, and then setting a label. The authors performed these pass-throughs individually from each other and then compared their results. If any of the entries were labeled differently between the authors, a debate was held until an agreement was reached.

During the first pass-through the comments were labeled based on their potential of being a security issue, with either a *zero* or a *one*. The meaning behind each label is:

- *zero* - the comment is not security related e.g. "Done".
- *one* - the comment pointed to something problematic, could be a potential security issue as well as other things.

The second pass-through was then performed only on comments initially labeled with a *one*. The goal during the second pass-through was to derive if the underlying problem which the comment describes is related to a security vulnerability or some other type of defect. During the second pass-through, if a comment previously labeled with a *one* was perceived as security related, the label was changed to a *two*. The meaning of this label can then be formulated as:

- *two* - the comment pointed with a high probability of being a security issue.

As mentioned earlier, the vulnerable code fragments should not only showcase vulnerabilities but also classify them against a taxonomy so that the newly found vulnerabilities could also be classified accordingly. In order to validate this last step, all comments that were labeled as a *two* were reviewed one last time. The original classification for the vulnerable code was supposed to be reflected within the content of the comment. In other words, the *Comment* should in some way reflect the *Query Taxonomy* seen in Table 5.2.

6

Results

This chapter shows the results gathered for each research question. First, regarding **RQ1** the results of the model evaluation are shown. Then, the vulnerability data acquired when answering **RQ2** is described in detail. Lastly, for **RQ3**, SeCoRA was used in an attempt to find new security vulnerabilities by searching through code reviews. The results of this search are evaluated and presented.

6.1 Evaluation of code comparison models

The two unsupervised models trained during the study were evaluated on their ability to distinguish between different code fragments. Both of the models were evaluated using single lines where the first line of each fragment is used. Because one of the models was able to accept larger input than single lines, it was also evaluated using the first three lines of the code fragments and then the complete fragments. An additional second evaluation was also performed where one of the models is compared against an existing solution to the code comparison problem.

This section will go through the results and the data used during each evaluation.

6.1.1 Single line model evaluation

Dataset	**Number of entries**
Query data (*if statements*)	306
if statements	2447
for loops	2087
other code	453

Table 6.1: Overview of data of single lines of code used during model evaluation

An overview of the data used during the single line model evaluation can be found in Table 6.1, 306 *if statements* were compared by the model against the other datasets. For each comparison, the model calculates the Minkowski distance expressing how similar it thinks those two entries are to each other. The complete description of the evaluation process can be found in Section 5.2.2.1.

Figure 6.1 shows the results of the evaluation. Using the Minkowski value, a distribution was calculated for each dataset. The smaller the mean of the distribution the more similar the datasets are to each other according to the model.

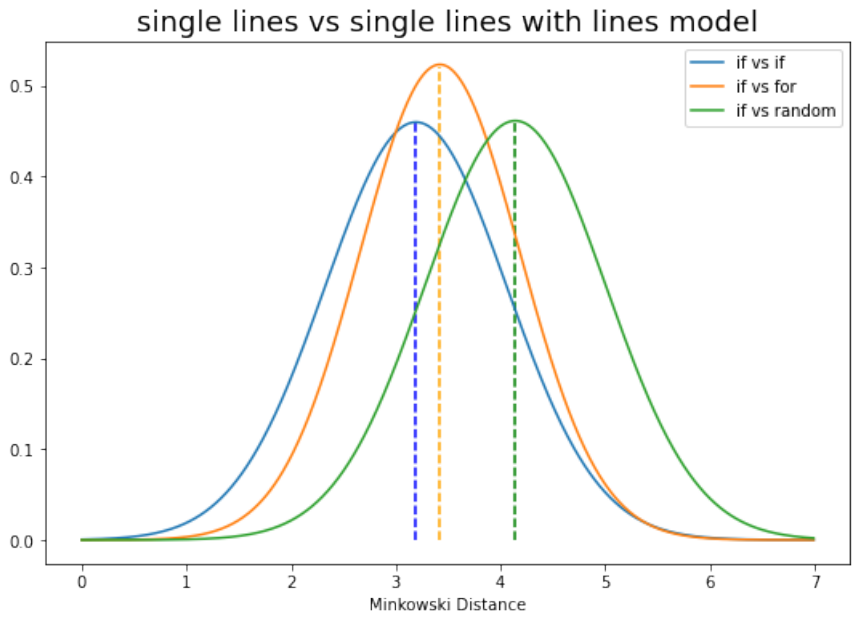

Figure 6.1: Results of comparing a dataset of *if statements* towards other *if statements*, *for loops* and *random code* with the model trained on lines of code.

According to the results, although the difference between the distributions is small, the model is correctly able to identify that the query *if statements* were most similar to other *if statements*. Furthermore, the graph also tells us that according to the model *for loops* are relatively similar to the query data while *other code* is noticeably different.

6.1.2 Block model evaluation

Similarly, the block model was evaluated to understand to what degree it is able to perform the same task as the line model and how changing the size of the fragments affects the results. The same dataset was used for both the single line and block model evaluation. A summary can be found in 6.1.

The evaluation process remains the same as with the single line model but is performed three times with varying size of code fragments. First, only the first line of the code fragment was used, then the first three lines, and lastly, the whole fragment. The results of each evaluation can be found in Figure 6.2, 6.3 and 6.4 respectively.

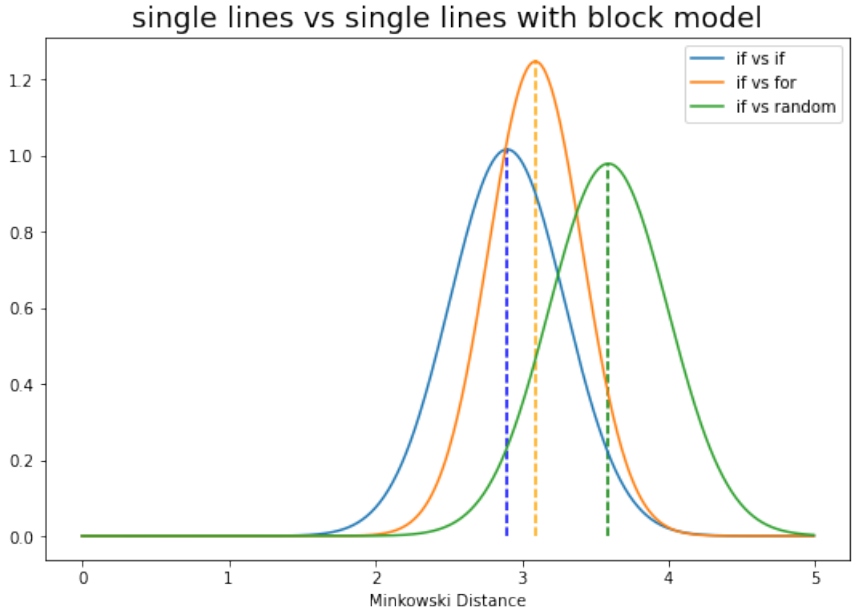

Figure 6.2: Single lines compared to other single lines with block model.

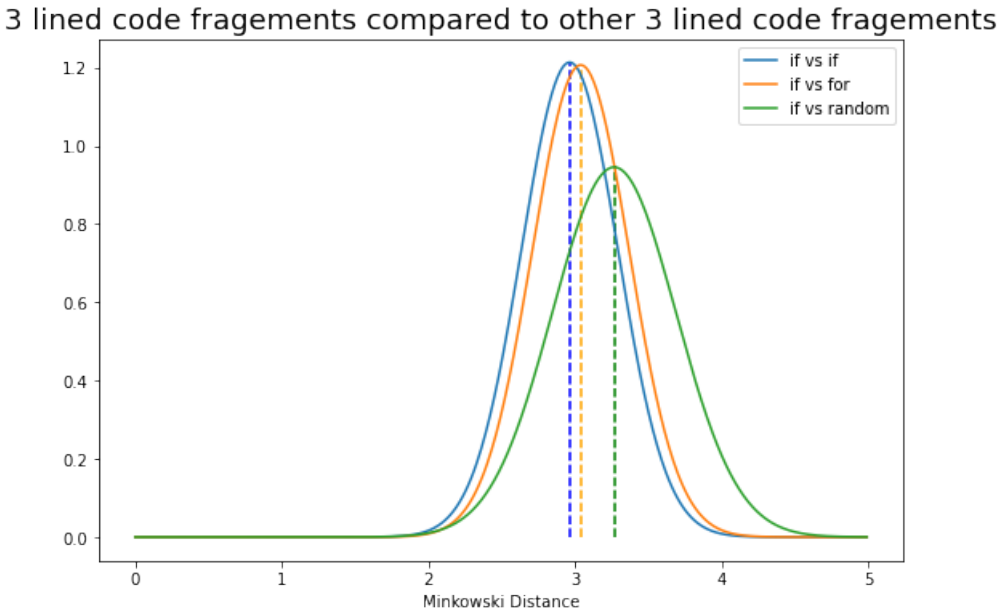

Figure 6.3: Three lines compared to other three lines with block model.

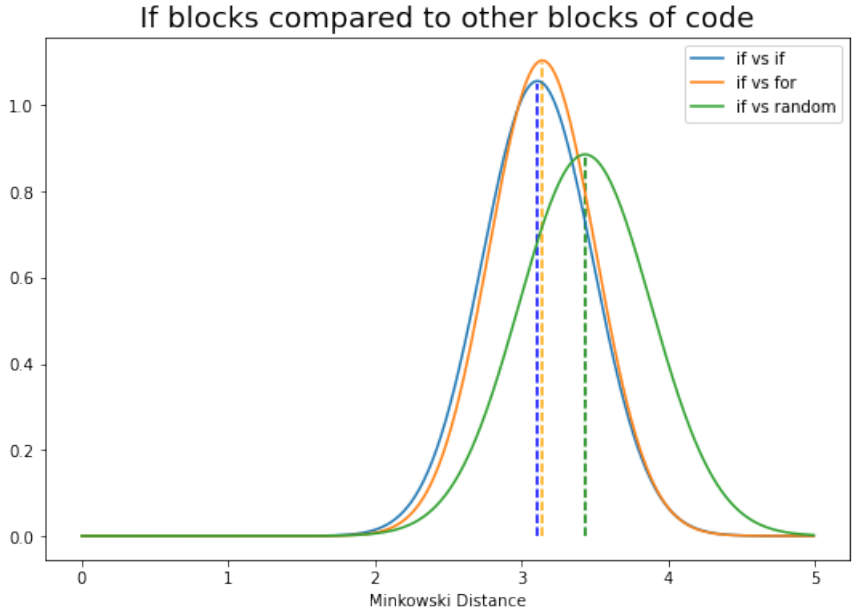

Figure 6.4: Blocks of code compared to other blocks of code with block model.

The means of the distributions in Figure 6.1 and Figure 6.2 show that the single line model and the block model gave very similar results during the single line evaluation.

In addition, graphs in 6.3 and 6.4 tell us that increasing the size of the code fragment when comparing *if statements* against other code makes it harder for the models to tell the different datasets apart. The graphs show that according to the model, the bigger the code fragment used during the evaluation, the more similar the datasets appear to be.

6.1.3 Model Selection

Because of the time limitations, only one of the two models was used throughout the remainder of the study. The remaining test and evaluations on both models would have been too time-consuming.

The choice was made based on the complexity of the models. The authors preferred to test the more straightforward solution first, in case it turned out to be sufficient, and there would be no need for the more complex one. As the block model can handle blocks of varying size as the input, it is the intrinsically more complex model. As such, the single line model was selected and was the only one used through the remainder of the study.

One could also argue that the block model could have been used without increasing the size of the fragments. While this is true, the results from the previous section showed that the models have similar results for the single line evaluation. As such, there is no benefit of using one over the other. Therefore, again, the simpler model is preferred.

6.1.4 Additional Line Model Evaluation

In order to compare the single line model against an existing approach, the Levenshtein distance, a set of artificial code lines were prepared, which are shown in Table 6.2. The single line model then used this set to query one of the AOSP repositories for similar lines. As described in Section 5.2.2.2, for each found line, the Levenshtein distance is manually calculated to see what a non-ML approach would see as an outlier. The results of this search can be found in Table 6.3.

Artificially made code examples
for (int i = 0; i <size; i++) {
if (condition) {
if (a == b) {
if (a != null) {

Table 6.2: The artificially made examples used to search for similar code using single line model.

This evaluation showed that in most cases, the Levenshtein distance agrees with what the model showed to be similar. Although, on the opposite, there are also examples which the model finds similar, but the Levenshtein method sees as drastically different, which can be seen in Table 6.3.

Since similarity is a subjective attribute, there is no simple way to judge if one method is more right than the other when it comes to deviance in their "opinion". Although, with that in mind, the first interesting aspect that should be considered are the *for loops* in the table. Most likely, the Levenshtein method saw them as outliers because of the more complex and lengthy natures of those fragments. The model was able to look past this complexity and still deemed them similar. The second interesting aspect is that the model found the structure of *switch* and *synchronized* statements to be similar to "*if (condition)*" which once again, the Levenshtein distance saw as outliers.

This contrast between the model and Levenshtein is important because if the methods gave the exact same results, there would be no reason to use SeCoRA as it's vastly more complicated. These results suggest that the model is able to capture more complex and interesting patterns and find similarity in fragments which the Levenshtein method would have omitted.

6.2 Acquiring Security-Labeled Data

This section covers how examples of AOSP security vulnerabilities were acquired. First, the results of a literature search are presented, then the data gathered during the literature search is processed so the model can use it.

6.2.1 Data gathering

In order to acquire a source of existing labeled security vulnerabilities, the authors performed a search for existing research surrounding security taxonomies within the domain of the AOSP. An assumption was made that the same research has a high chance of containing a labeled dataset which was used to create that taxonomy. The search was performed both using the Google Scholar search engine and manually within the Mining Software Repositories conference database.

The initial search made used combinations of the following terms:

- Android;
- Security; and
- Taxonomy.

There were heaps of literature found, however, these results did not cover the AOSP as a whole. Instead, these results covered security topics related to third-party applications that would run on the Android operating system. In an attempt to improve the results and find literature more focused on the AOSP, the search terms were extended with the following:

- Common Vulnerabilities; and
- Common Weaknesses.

These terms led the authors to find two major papers covering the domain of AOSP security taxonomies, the works of Jimenez et al. [37] and Linares-Vasquez et al. [1] which both had a corresponding taxonomy that could be reused during this study.

Jimenez et al. analyzed 32 vulnerabilities from a CVE database[1] and based their taxonomy on the underlying issues, corresponding components, and code complexity of patches within each CVE entry. Linares-Vasquez et al. then build on this work and perform a similar analysis but on a substantially larger dataset consisting of 660 AOSP specific CVE entries. Those 660 CVE entries and their corresponding classification can be easily accessed through the authors' website[2] and was therefore reused as the dataset for this study.

An overview of this data can be seen in Figure 6.5, and the taxonomy in whole can be seen in Appendix C.1. One can see in Figure 6.5 the different classifications and their corresponding amount of CVEs.

6.2.2 Vulnerability data processing

As the code comparison model was trained on Java code, the vulnerability dataset acquired during the literature search contains code in other languages and therefore had to be filtered accordingly. The filtering process was done by simply letting the

[1] `https://cve.mitre.org/`

[2] `https://ml-papers.gitlab.io/android.vulnerabilities-2017/appendix/index.html`

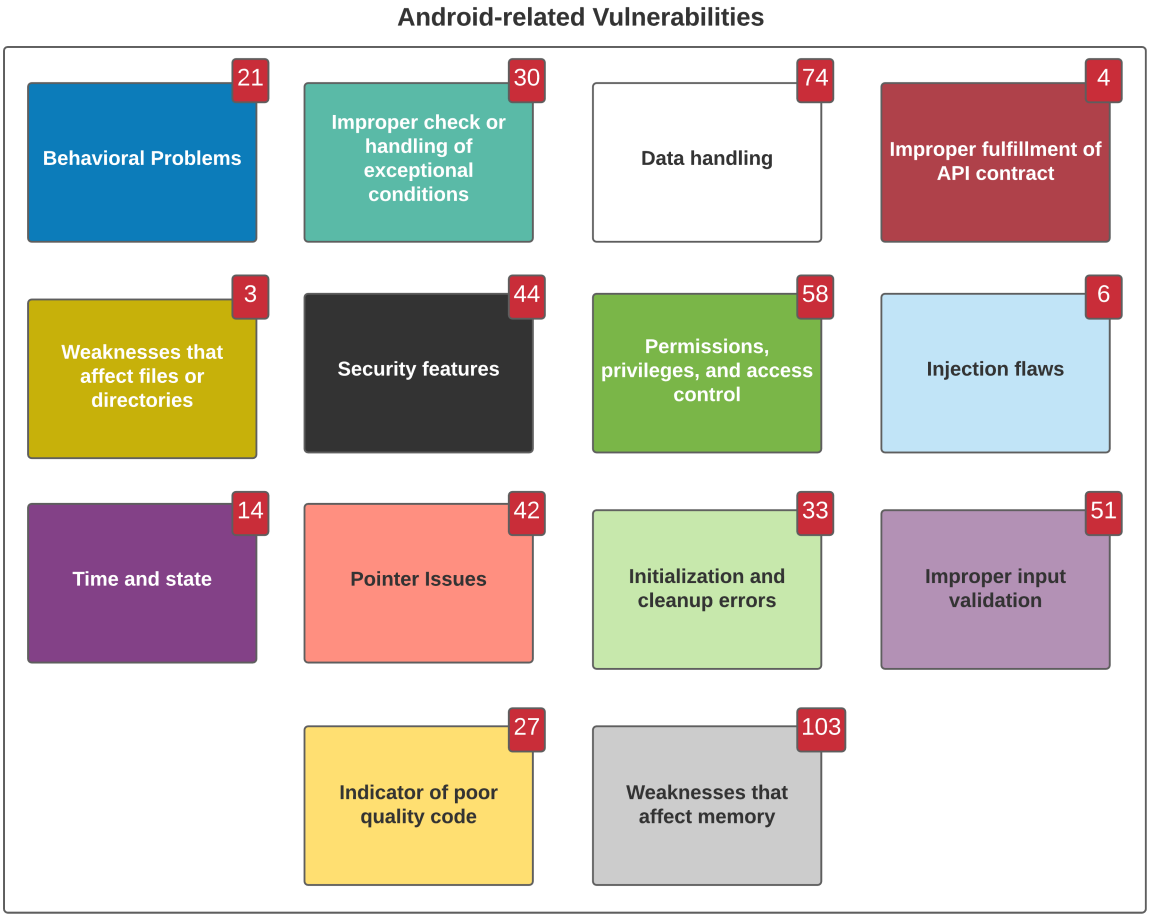

Figure 6.5: High-level overview of Linares-Vasquez et al. Taxonomy with corresponding amount of collected CVEs.

CVE scripts check the extensions of the files within the CVE records and exclude anything except the Java files. Table 6.4 shows an overview of the final query dataset.

Table 6.4: Amount of collected Java files from all CVE records.

Classification	**# files**
Permissions, privileges, and access control	**63**
Data handling	**18**
Improper Input Validation	**9**
Time and State	**3**
Weakness that affect memory	**3**
Security features	**3**
Injection flaws	**2**
Indicator of poor quality code	**1**
Total	**102**

6.3 SeCoRA used to find Security Related Code Reviews

Using the single line model and the acquired dataset of known vulnerability data, SeCoRA can be used to query code for new vulnerabilities. The following sections will cover the following: (1) an overview of the code reviews, which will be queried for vulnerabilities, and (2) the results of the query process are shown and evaluated.

6.3.1 Data to be queried - Code Reviews

Section 5.3 describes the process of gathering code reviews in full detail. SeCoRA was used to mine both the *Open* and *Merged* Gerrit endpoint of the AOSP. The tool specifically gathered only those code reviews which had a negative review rating, based on the assumption that those have a higher chance of containing a security defect. An overview of the data collected can be seen in Table 6.5.

Table 6.5: Statistics of the data pulled from Gerrit.

Date	*Source*	Rows	Unique	Unique %
15/2/2021	Open	849	749	88.22%
15/2/2021	Merged	465	445	95.70%
	Total unique		1194	100%

The columns in the table represent the following:

- Date - is the day which the data was collected from Gerrit.
- Source - is the endpoint where the data was retrieved from.
  - Total unique - is the total unique entries found.
- Rows - is the amount of entries pulled from Gerrit.
- Unique - is the amount of unique entries pulled.
- Unique % - is the percentage of total entries that are unique.

6.3.2 Search results and evaluation

The final step using SeCoRA was to search for code reviews containing vulnerable code. This was done with the help of the single line model and the existing vulnerability data that had been previously collected. As illustrated in Figure 5.10, the existing vulnerability data was then used to query code reviews. The results of this search are summarized and evaluated here according to the process described in Section 5.4.

The evaluation consisted of the authors manual passes going over the newly found similar lines and labeling them based on the following semantics:

- *zero* - the comment pointed by no means to being a security issue, e.g. “Done”.
- *one* - the comment pointed to something problematic, could be a potential security issue as well as other things.
- *Two* - the comment pointed with a high probability of being a security issue.

Table 6.6 shows a summary of all labeled data that was found by SeCoRA:

Table 6.6: Overview over all combinations of code and comments that were classified.

Potential	**Amount**
0	1,130
1	230
2	61
Total	1,421

The top five most common comments which were found, which were labeled with a *zero*, can be seen in Table 6.7 below:

Table 6.7: Overview over the top five most common comments

Potential	**Comment**	**Count**
0	Done	285
0	The code in this case is quite a lot. Can you please wrap it in a private method ...	151
0	Will check	151
0	I think <name> already said this, but it would be better to leave this sort of style fix ...	54
0	Sorry about that. I will keep that in mind.	54
	Total:	695

It should be noted that why a comment can repeat so often is because of multi-line comments. As illustrated in Figure 5.9, if a comment is placed over a block of code, each comment gets duplicated for each line of that fragment. Thus, if one only looks at the unique comments, they see the following overview instead:

Table 6.8: Overview over unique comments

Potential	**Amount**
0	98
1	71
2	11
Total	180

As a final step of the evaluation, each of the eleven matched code reviews was validated. The original classification of the CVE code used to match these code reviews should be reflected within the content of the corresponding comment. So, as an example, for the result to be valid, if the original CVE was marked with the type "Pointer issues", the comment of the review should in some way discuss this problem. This inspection showed that out of those eleven entries, none of the corresponding comments matched this classification.

Table 6.3: Examples of Levenshtein outliers from data collected using the single line model.

Query line	# found	# Leven. outliers	Leven. Mean.	Leven. Std.	95th Percentile	Outliers	Leven. Distance

for (int i = 0; i <size; i++)	96	4	7.78	3.82	14.06	for (int i = offset; i <offset + length; i++) {
						for (int i = 0; i <args.length && !error; i++) {
						for (int i = start; i <end - 2; i++, prev = c) {
						for (int i = 0; i <mConnectMessage.length; i++) {
							19
							19
							22
							20

if (condition)	138	7	9.47	2.55	13.66	switch (apkCreatorType) {
						synchronized (LOCK) {
						synchronized (chan) {
						synchronized (clients) {
						synchronized (storage) {
						synchronized (server) {
						synchronized (lock) {
							18
							15
							14
							16
							17
							17
							14

if (a == b)	145	6	8.70	2.42	12.69	if (number == argument) {
						if (properties == null) {
						if (application == null) {
						if (annotation == null) {
						if (component == null) {
						if (definition == null) {
							14
							14
							14
							13
							13
							14

if (a != null)	180	12	6.32	2.57	10.55	if (version == 2) {
						if (version != 2) {
						if (this == target) {
						if (version == 0) {
						if (compare == 0) {
						if (version == 1) {
						if (version == 16) {
						if (node == element) {
						if (application == null) {
						if (properties == null) {
						if (description == null) {
						if (definition == null) {
							11
							11
							11
							11
							11
							11
							12
							12
							11
							11
							12
							11

7

Discussion

In this chapter the significance of the study, threats to validity, ethical considerations and future work are discussed.

7.1 Significance of the study

A common practice employed with the goal of preventing security defects in software projects is the modern code review. Unfortunately, studies have shown that the actual underlying benefits of MCR, in fact, do not match the common expectations of finding defects but instead have the most impact on other aspects such as improving knowledge sharing and communication [8].

A novel tool, ACoRA, has been proposed to address this issue by integrating into MCR and performing automated code reviews with the help of machine learning techniques. However, for ACoRA to operate correctly, it needs a dataset containing examples of already performed code reviews where a defect is present and discussed by the reviewer.

The study proposed SeCoRA, a complementary tool to ACoRA which can be used as a means of gathering security vulnerability data that can be used for training. The tool operates by using a subset of ACoRAs features, specifically, the ability to find similarities between different code fragments. By gathering existing known vulnerable code, in this case, CVE vulnerabilities, SeCoRA was used to query previously performed AOSP code reviews in an attempt to find those which contain vulnerabilities.

In this study, SeCoRA showed the ability to distinguish between datasets of code fragments and could identify similar code, which a more traditional method, the Levenshtein distance, would have seen as drastically different. However, it was also shown that increasing the size of the fragments used during the code comparison process has an overall negative impact on the final comparison results. Based on these findings, using a simpler single line model was preferred.

When using the tool together with a vulnerability database and AOSP code reviews, the tool detected 1,421 code fragments with code similar to an existing vulnerability. By removing duplicates entries and performing a manual evaluation, the authors

concluded that 11 of those fragments had a high potential of being an instance of a security vulnerability. Further investigation showed that out of those 11 potential vulnerabilities, none matched the original vulnerability classification of the fragment used to find it. This, in turn, entails that the results from this thesis are negative, as these potential vulnerabilities are likely not classified correctly and are too few to train SeCoRA/ACoRA to its full potential.

We hope that SeCoRA can inspire other researchers by showing and exemplifying how this thesis tried to compare code fragments in order to find security related code reviews. The results showed that SeCoRA could find security related code reviews by comparing code fragments, but those fragments could not be confidently linked to the code used to find them. As such, the proposed method could be improved but using more robust comparison methods.

7.2 Threats to Validity

External Validity
A threat to the generalizability of the results within this study is the very narrow scope of this thesis. The study was performed on a single project, the Android Open Source Project, using specifically Java code. These choices may have hidden implications, and the results may have been different if the training and comparison data came from other projects or if other programming languages were used.

In addition, SeCoRA used CVE records collected from the work of Linares-Vasquez et al.[1] in order to find new security related code reviews. However, these records were made in the domain of security and specifically in the domain of the AOSP, which also affects the generalizability of the results as SeCoRA could potentially perform better on other types of defects.

Internal Validity
The results of this thesis indicate that SeCoRA can distinguish between code in general when comparing datasets of *if statements*, *for loops*, and *other code*. However, the results may have been noticeably worse or better if other types of code fragments were used.

When interpreting the results, the reader should consider that the code reviews containing vulnerable code may have been found by the tool "accidentally" and then, by chance, also contain comments related to security. This could have been the case as the code reviews used in the study were specifically the ones that have received a negative code review rating.

Lastly, as the authors do not possess extensive knowledge in the security domain, the manual approach taken to evaluate the final results is also a significant threat. This means that the final labeling and judgment made on the code review comments is bound by the technical abilities of the authors and may therefore contain errors.

7.3 Ethical considerations

This thesis explored an automated method to discern which code reviews exemplify security defects with the intent to aid the software discipline. However, ethical considerations must be taken as any vulnerabilities found through the study could be used with malicious intent. As such, none of the potentially faulty code found by SeCoRA is openly disclosed.

In addition, although the majority of the data used in this project is publicly available, with respect to the AOSP data mining limitations, this data will also not be shared.

8

Project Post-mortem and Future Work

The study concluded with negative results as SeCoRA was not able to reliably acquire the data needed to train ACoRA. This chapter acts as a post-mortem of the project and discusses the potential underlying causes behind the negative results. The issues highlighted and discussed here can serve as a foundation for any future work around SeCoRA.

Throughout the text, figures similar to the one below have been used to show the reader the three main elements of this study marked with the colors red, blue, and green. Each section of this chapter corresponds to one of these elements and discusses them respectively.

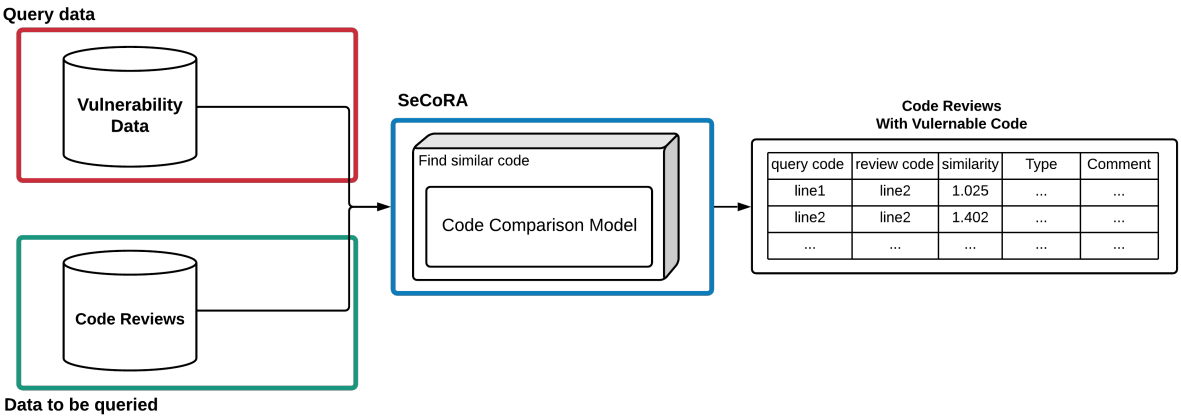

Figure 8.1: Illustration highlighting the three major elements of this study discussed in this chapter.

8.1 Vulnerability Data

Data Quantity
In this study, 102 AOSP Java vulnerabilities and their corresponding code were used as query data. Various aspects of this data can have a significant impact on how well SeCoRA performs. The most apparent approach to potentially improve the results would be to increase the number of known vulnerabilities that the tool uses as query data. An extensive dataset allows the tool to compare the code reviews against more vulnerabilities, increasing the likelihood of finding something.

Data Quality
Other than quantity, the quality of the data should also be considered. The CVE database, where the vulnerabilities used in this study come from, doesn't put much weight into either explaining or exemplifying each defect. Each entry in the database typically references a commit where the corresponding vulnerability was fixed. This commit might have changed code other than the vulnerability itself, and the exact location of the vulnerable code is not specified. SeCoRA is therefore forced to consider the whole commit as a vulnerability. This could be a reason why such a large part of the results, 1,130 out of 1,421, was non-defective code.

Another qualitative aspect of the data which was not considered during the study is the *recognisability* of the vulnerability data. It is not certain how many of the 102 data points can actually be identified using a single line or block of code. A security defect could require intricate knowledge of multiple parts of the system in order to be recognised. Any of these complex vulnerabilities are not useful as query data when used together with SeCoRA. Once again, this issue could be a potential reason behind the negative results.

Future work
For future research, we propose a more focused approach when it comes to vulnerability data. Instead of trying to cover a broad spectrum of vulnerability types, the idea is to select one specific vulnerability. The selection process can be done by sorting through the 102 already known data points and selecting one with a high degree of recognisability. The goal would then be to collect a large number of examples of that specific vulnerability across different contexts, where the exact problematic code is known. This new query data would alleviate the issues mentioned in this section and allow for a more accurate evaluation of SeCoRA.

8.2 Code Comparison Model

Code Comparison Performance
One of the reasons why SeCoRA failed could have simply been that the model's ability to compare code is not sufficiently good for finding security vulnerabilities. The evaluation of the comparison model in this study showed that SeCoRA is able to differentiate between *if statements* and *for loops*. While these results show that the model has successfully learned something during the training processes, they

don't measure if the comparison ability carries over to all code.

Future Work
During this study, the authors learned that evaluating a comparison model is complicated as the similarity can be very subjective. Comparing two fragments of code and expressing their similarity would not only be very hard for a human but would also very likely result in a different answer depending on who is asked. This makes it very difficult to assess if the model is correct or not when it states how similar two pieces of code are.

Thus, the authors suggest a different angle. Instead of measuring if the model can compare code fragments, the tool should be assessed based on its ability to solve the problem at hand, that is, recognising particular vulnerabilities.

Suppose that, as proposed in the previous section, we acquire a large number of different examples of a single vulnerability. The performance of the model can then be evaluated by taking a subset of the data, for example 80%, and using that to query the remaining 20%. The assumption is then that if a model is able to tell that those two subsets are very similar to each other, it will also be able to find other occurrences of that vulnerability.

8.3 Code Reviews

Number of Code Reviews
The final aspect of the study that could motivate the negative results is the size of the code review dataset which was queried for vulnerable code. The authors were not able to find out how often a code review within the AOSP project is related to security. If this is a rare occurrence, then the 2500 reviews used in this study may not have been a sufficiently large search space. This means that the tool may not have been able to correctly identify any vulnerability because not enough code reviews were checked. The results might have looked different if the size of the dataset was increased.

Future Work
It is worth mentioning here that the authors did consider collecting more data but were constrained by the time-consuming and inefficient nature of the manual evaluation process used in the final stages of the study. Using more data would lead to more results to evaluate, which was not possible within the available time frame.

Instead of the manual evaluation, a more efficient process is needed. If the help of an expert can be acquired, it is possible to create an oracle that can be used instead. This oracle consists of code reviews which are known to be security related and where the underlying defect type is known. Using this oracle, SeCoRA can be evaluated more efficiently and more objectively as this method doesn't require any security knowledge from its users and entirely relies on the expert's work.

9

Conclusion

Due to the error prone nature of humans, software quality assurance has been a significant part of the software engineering discipline. While the more cumbersome formal inspection techniques from the 70s have proven to improve software quality, the industry has moved towards a more lightweight approach, the modern code review (MCR). However, as the MCR became more common, the underlying purpose of code reviews has changed, Bacheli and Bird [7] show that the actual benefits of reviews do not match the typical expectations of finding defects but rather improve knowledge sharing and communication.

A novel tool, ACoRA, has been proposed to address this issue by integrating into the MCR and performing automated code reviews with the help of machine learning techniques. However, for ACoRA to operate properly, it needs a dataset containing examples of already performed code reviews where a security defect is present and discussed by the reviewer.

The study proposed an automated approach to solve ACoRA's limitations, SeCoRA, a complementary tool that can be used as a means of gathering data that can be used for training. The tool operates by using a subset of ACoRA's features, specifically, the ability to find similarities between different code fragments. The new treatment design was tested and evaluated on different datasets of code in order to answer **RQ1**: *To which degree can SeCoRA distinguish between code fragments in general?* It showed that it could to some degree distinguish between different datasets of code and that it correctly identified similarities in code that the more well known Levenshtein distance would have found to be different. Next, a literature search was performed in the thesis in order to answer **RQ2**: *How can we acquire security-labeled code fragments to identify new security vulnerabilities?* The literature search resulted in finding a security taxonomy within the domain of Android Open Source Project (AOSP) with 660 Common Vulnerabilities and Exposures (CVE) records. SeCoRA was then used to answer **RQ3**: *Which new security vulnerabilities can we find in AOSP code reviews when searching with fragments from **RQ2**?* SeCoRA was used to search for security issues using single lines from 102 Java files found in the CVE records against 1194 code reviews collected from the AOSP. The search resulted in finding 180 unique code reviews, where 11 of them potentially contained vulnerable code. However, out of those 11 code reviews, manual validation showed that none match the original classification of the code used to find them. As such,

the results from this thesis were negative, and SeCoRA in its current form is not a reliable way of gathering the training data used by ACoRA.

Bibliography

[1] M. Linares-Vasquez, G. Bavota, and C. Escobar-Velasquez, "An empirical study on android-related vulnerabilities," *IEEE International Working Conference on Mining Software Repositories*, no. i, pp. 2–13, 2017.

[2] A. Vaswani, "Attention Is All You Need," no. Nips, 2017.

[3] D. H. O'Dell, "The debugging mindset," *Queue*, vol. 15, no. 1, pp. 1–20, 2017.

[4] M. E. Fagan, "Design and Code Inspections To Reduce Errors in Program Development.," *IBM Systems Journal*, vol. 15, no. 3, pp. 182–211, 1976.

[5] P. C. Rigby, "Contemporary Peer Review in Action :," pp. 56–61.

[6] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "An empirical study of the impact of modern code review practices on software quality," *Empirical Software Engineering*, vol. 21, pp. 2146–2189, oct 2016.

[7] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," *Proceedings - International Conference on Software Engineering*, pp. 712–721, 2013.

[8] M. Ko and C. Dorantes, "The impact of information security breaches on financial performance of the breached firms: An empirical investigation," *Journal of Information Technology Management*, vol. 17, no. 2, pp. 13–22, 2006.

[9] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *arXiv*, vol. 51, no. 4, 2017.

[10] B. Cui, J. Li, T. Guo, J. Wang, and D. Ma, "Code comparison system based on abstract syntax tree," in *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*, pp. 668–673, 2010.

[11] J. Devlin, M. W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *NAACL HLT 2019 - 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies - Proceedings of the Conference*, vol. 1, no. Mlm, pp. 4171–4186, 2019.

[12] H. Abram, T. B. Earl, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 837–847, 2012.

[13] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, p. 122–131, Apr. 2016.

[14] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, 2010.

[15] M. P. Robillard, *Recommendation systems in software engineering.* Springer, 2014.

[16] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 281–293, 2014.

[17] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 38–49, 2015.

[18] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, pp. 2091–2100, PMLR, 2016.

[19] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: bug detection with n-gram language models," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pp. 708–719, 2016.

[20] C.-H. Hsiao, M. Cafarella, and S. Narayanasamy, "Using web corpus statistics for program analysis," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, pp. 49–65, 2014.

[21] E. Fast, D. Steffee, L. Wang, J. R. Brandt, and M. S. Bernstein, "Emergent, crowd-scale programming practice in the ide," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 2491–2500, 2014.

[22] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, pp. 173–184, 2014.

[23] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 585–596, IEEE, 2015.

[24] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 574–584, IEEE, 2015.

[25] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2073–2083, 2016.

[26] R. E. Gallardo-Valencia and S. E. Sim, "Internet-scale code search," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pp. 49–52, IEEE, 2009.

[27] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, pp. 1–30, 2013.

[28] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," *arXiv preprint arXiv:1611.01989*, 2016.

[29] E. Parisotto, A.-r. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," *arXiv preprint arXiv:1611.01855*, 2016.

[30] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," *2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 - Proceedings*, pp. 202–212, 2013.

[31] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, 2014.

[32] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines," *IEEE International Working Conference on Mining Software Repositories*, pp. 334–344, 2017.

[33] G. Mcgraw, "Static analysis identifies many Automated Code Review Tools for Security," tech. rep.

[34] E. Sultanow, A. Ullrich, S. Konopik, and G. Vladova, "Machine learning based static code analysis for software quality assurance," *2018 13th International Conference on Digital Information Management, ICDIM 2018*, pp. 156–161, 2018.

[35] M. Mukadam, C. Bird, and P. C. Rigby, "Gerrit software code review data from android," *IEEE International Working Conference on Mining Software*

Repositories, pp. 45–48, 2013.

[36] S. M. Omohundro, "Five balltree construction algorithms," tech. rep., 1989.

[37] M. Jimenez, M. Papadakis, T. F. Bissyande, and J. Klein, "Profiling Android Vulnerabilities," *Proceedings - 2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016*, pp. 222–229, 2016.

[38] R. J. Wieringa, *Design science methodology: For information systems and software engineering*. 2014.

[39] Z. Wang, P. Ng, X. Ma, R. Nallapati, and B. Xiang, "Multi-passage BERT: A globally normalized BERT model for open-domain question answering," *EMNLP-IJCNLP 2019 - 2019 Conference on Empirical Methods in Natural Language Processing and 9th International Joint Conference on Natural Language Processing, Proceedings of the Conference*, no. 1, pp. 5878–5882, 2020.

[40] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps," *SIGPLAN Not.*, vol. 49, pp. 259–269, jun 2014.

[41] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Trans. Comput. Syst.*, vol. 32, jun 2014.

[42] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale," in *Trust and Trustworthy Computing* (S. Katzenbeisser, E. Weippl, L. J. Camp, M. Volkamer, M. Reiter, and X. Zhang, eds.), (Berlin, Heidelberg), pp. 291–307, Springer Berlin Heidelberg, 2012.

[43] L. Jia, J. Aljuraidan, E. Fragkaki, L. Bauer, M. Stroucken, K. Fukushima, S. Kiyomoto, and Y. Miyake, "Run-Time Enforcement of Information-Flow Properties on Android," in *Computer Security – ESORICS 2013* (J. Crampton, S. Jajodia, and K. Mayes, eds.), (Berlin, Heidelberg), pp. 775–792, Springer Berlin Heidelberg, 2013.

[44] A. Nadkarni, B. Andow, W. Enck, and S. Jha, "Practical DIFC Enforcement on Android," in *25th USENIX Security Symposium (USENIX Security 16)*, (Austin, TX), pp. 1119–1136, USENIX Association, aug 2016.

[45] A. Nadkarni and W. Enck, "Preventing Accidental Data Disclosure in Modern Operating Systems," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, (New York, NY, USA), pp. 1029–1042, Association for Computing Machinery, 2013.

[46] Y. Xu and E. Witchel, "Maxoid: Transparently Confining Mobile Applications with Custom Views of State," in *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, (New York, NY, USA), Association for Computing Machinery, 2015.

A

Appendix 1

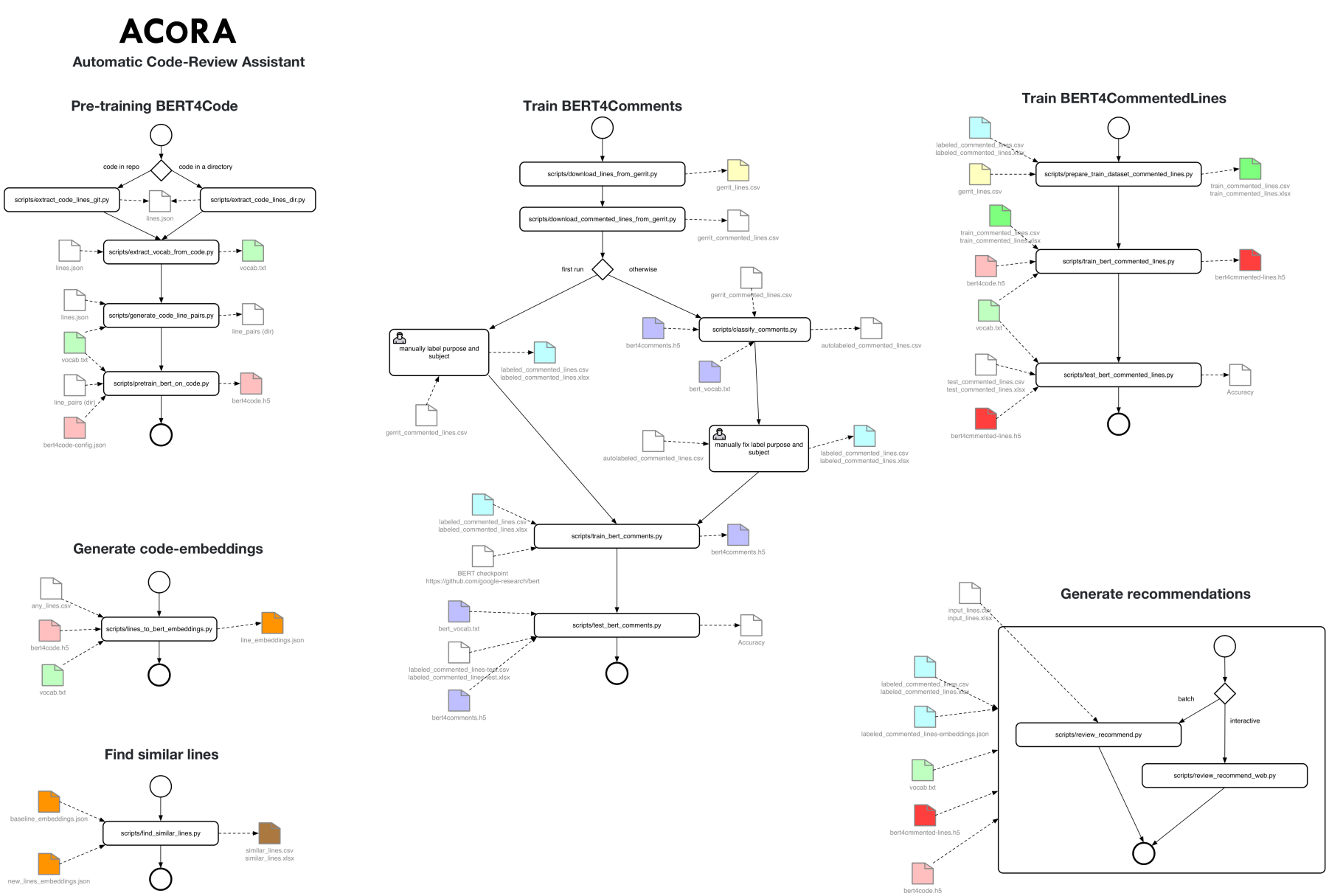

Figure A.1: ACoRA's scripts and pipelines.

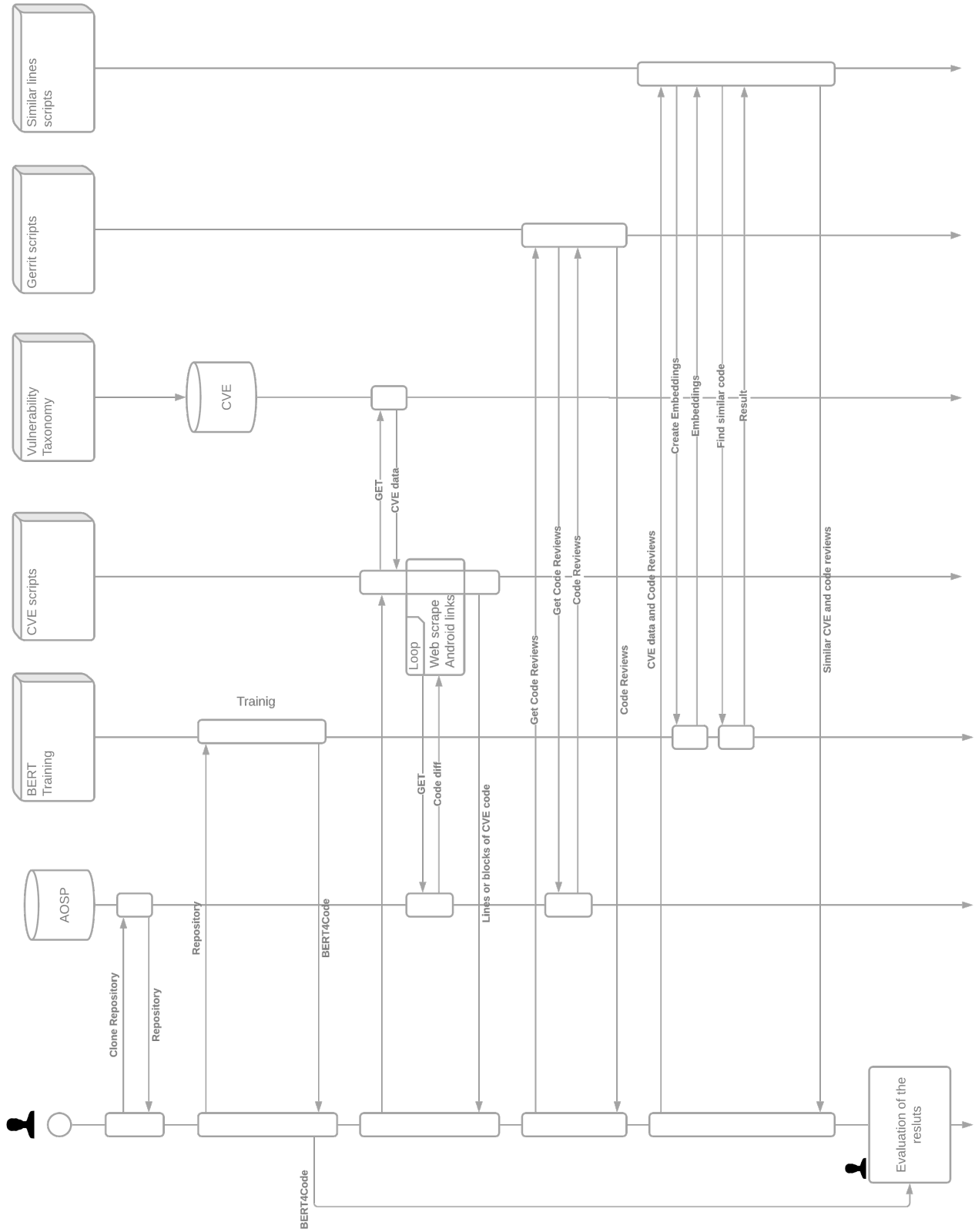

Figure A.2: Flow diagram of the SeCoRA's toolchain

B

Appendix 2

Two underlying models for SeCoRA were trained during the thesis. The first model was trained on a single line basis and the second model was trained on bigger fragments of code. The models were trained on the same repository of code as described in Section 5.2.1.1 but with different parameters. The following two sections gives details about the parameters used during training for the two models.

B.1 Training parameters single line model

The single line model was trained with the following configuration below:

Table B.1: Configuration for training single line model on code.

Hyper-parameter	**Value/Setting**
hidden_size	384
hidden_act	gelu
initializer_range	0.2
vocab_size	10,000
hidden_dropout_prob	0.1
num_attention_heads	8
type_vocab_size	2
max_position_embeddings	256
num_hidden_layers	4
intermediate_size	2,048
attention_probs_dropout_prob	0.1

The model was initially set to train with the following settings below:

Setting	**Value**
batch_size	8
epochs	20
steps per epoch	18,061

Table B.2: Settings for training single line model on code.

The model eventually dropped out from training as the loss function had converged after 12 epochs.

B.2 Training parameters block model

The block model was trained with the following configuration below:

Table B.3: Configuration for training single line model on code.

Hyper-parameter	**Value/Setting**
hidden_size	384
hidden_act	gelu
initializer_range	0.2
vocab_size	10,000
hidden_dropout_prob	0.1
num_attention_heads	8
type_vocab_size	2
max_position_embeddings	512
num_hidden_layers	4
intermediate_size	2,048
attention_probs_dropout_prob	0.1

The model was initially set to train with the following settings below:

Setting	**Value**
batch_size	32
epochs	20
steps per epoch	2,518

Table B.4: Settings for training single line model on code.

The model's loss function did however still drop so the model was not taken out of training until after 50 epochs when the loss function had converged.

C

Appendix 3

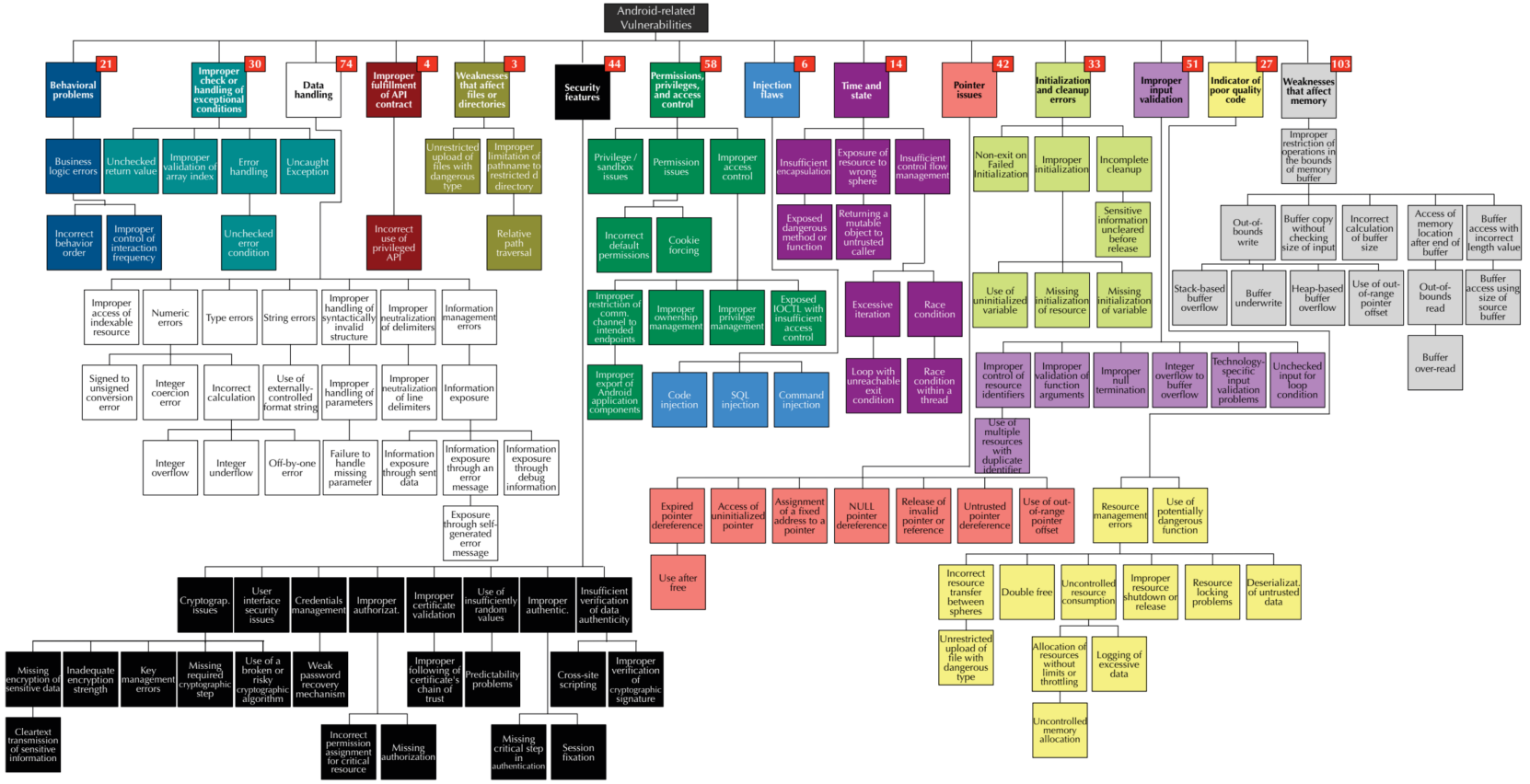

Figure C.1: Taxonomy of android-related vulnerabilities that was constructed by Linares-Vasquez et al. [1].