CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

Detecting security related code by using software architecture

MSc Software Engineering

Paulius Urbonas

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2018

Master's thesis 2018

Detecting security related code by using software architecture

PAULIUS URBONAS

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2018

Detecting security related code by using software architecture
PAULIUS URBONAS

Supervisor: Michel Chaudron, Department of Computer Science and Engineering
Examiner: Regina Hebig, Department of Computer Science and Engineering

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Detecting security related code by using software architecture
PAULIUS URBONAS
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This thesis looks into automatic detection of security related code in order to eliminate this problem. Since manual code detection is tiresome and introduces human error we need a more efficient way of doing it. We explore code detection by using software architecture and code metrics to extract information about the code and then using this information with machine learning algorithms. By extracting code metrics and combining them with Wirfs-Brocks class roles we show that it is possible to detect security related code. We conclude that in order to achieve much better detection accuracy we need to use different kind of methods. This could be software architecture pattern detection to extract additional information.

Keywords: Software architecture, security, code detection, machine learning

Acknowledgements

I would like to thank my supervisor Michel Chaudron and Truong Ho Quang for their patience, support and guidance throughout this thesis.
I would also like to thank everyone else involved that helped me with their advice and provided motivation when it was needed the most.

Paulius Urbonas, Gothenburg, January 2018

Contents

List of Figures

List of Tables

1 Introduction

In this chapter we provide a brief introduction to this thesis, what it is about, areas relevant to it as well as some reasoning behind of why we chose this topic. We start by providing some context then we explain why we believe there is a problem and whats the purpose of this thesis.

1.1 Context

Every major software project starts out as an idea which then software engineers transform into a more concrete vision and project [4]. All software projects tend to grow over time and in some cases become extremely complex solutions even if developers try to re-factor their code [5]. Despite size and complexity of various software projects they are designed with certain features, characteristics and properties in mind. During initial design phase all of the planned features are taken into account when deciding and designing software architecture and components. These design decisions can later on propagate to actual source code and so they may be traced back to software architecture. This information about software architecture or alternatively about the source code can be used to gain better understanding of the software and familiarize yourself about its architecture and functionality [6]. Additionally such architectural knowledge can help developers learn from each other and improve their future endeavours [7] [8].

What if such architectural knowledge could be used to help developers to be more efficient? What if software architecture could provide information necessary for some more specific needs such as finding code with a specific type of functionality or traits? A good example of this would be code related to security and sensitive (or private) data handling. Code can be overlooked in larger components of the system that are focused on different functionality. For example if we are interested in security related code in communication software we might miss it when, for example encryption, would take place only once in a much more complex functionality essentially blending it in with other code. It can be quite difficult to detect needed code and this could be detrimental in cases such as patching vulnerabilities or updating the code to certain regulations and standards. Manual code detection is one of the solutions however on large software projects this would be almost impossible due to the sheer size of the system and its components [9] [10].

1.2 Problem statement

There’s no such thing as perfect software. All software projects have a bug or two and sooner or later every piece of code ends up needing an update. In order to fix bugs and update code we first need to find and locate parts of it that need to be worked with. Finding such code with specific functionality or traits becomes a lengthy and perhaps a complicated process when software size exceeds hundreds of thousands of lines of code. The ability to locate code of interest with certain amount of accuracy would certainly help to save effort and time, especially in larger software projects.

Inefficient code location effects a range of parties from academia all the way to companies and developers who’ve been contracted to solve specific issues. Students and teachers may want to inspect, analyze and learn from specific software components in the system to gain better understanding of how it works. Developers will want to update software and fix bugs and vulnerabilities related to specific functionality. Any other individuals such as freelancers or just someone who is interested in creating their own fork on GitHub may want to locate specific functionality without having any documentation or understanding of the system at all. Having some kind of tool or process which would help to understand the code and locate its features would be beneficial to many and allow everyone to focus on actual work that they need to do.

While smaller software projects may not benefit from quick code detection as much as the large ones, being able to quickly detect needed code is still beneficial. Even if code detection is not 100% accurate it could still reduce the amount of work needed to find the code that you are looking for. At the very least there could be an approach that provides some additional information about the system and it’s features which would point you in the right direction and help to find what you are looking for. In addition information gathered during code detection and location could prove to be useful in other areas as well.

1.3 Purpose of the Thesis

The goal of this thesis is to focus on security and privacy related code detection and to investigate whether code detection is possible if we use metrics and other information extracted from software architecture. For this reason we look at the case of the open source software project K9-Mail [1]. The primary focus is to find out whether it is possible at all to use software architecture in a way where we could extract useful information (that aids in security related code detection) from it. Then the extracted information would be used together with software code metrics in an attempt to detect and locate code of interest (in this case any security related code). The information will be extracted from software architecture UML diagrams, various relationships between classes as well as metrics and by using any other methods that are deemed useful or may have potential in aiding with such code detection. Using more than one method to extract information as well as using different types of information about the source code allows comparison between different techniques, their efficiency and usefulness at code detection and give much clearer picture of what is actually feasible to achieve and what other potential areas the extracted information could be used for. From this we formulate our research questions:

1. **Assumption:** It is possible to identify security relevant code using information from software design and its architecture.
   (a) **RQ1**: Can software architecture provide any means to identify security relevant code?
   (b) **RQ2**: How useful are software metrics when used for security sensitive code detection?
   (c) **RQ3**: Can we improve our results by using additional features extracted from software architecture?
   (d) **RQ4**: How much of an effect do these additional features have?

From here on it is important to assume that we can use software architecture to extract some code metrics and architecture features by using already existing tools. Such assumption is needed to test hypothesis and derive potential methodology. Certain experimental methods will be used with extracted metrics and information in order to locate security related code. Another necessary assumption is that information extracted from software architecture will be good enough to use with machine learning algorithms. Additionally it is assumed that it will be possible to use machine learning in order to detect security relevant code. Naturally these assumptions will be tried and tested during experiments proving which ones are correct if there are any.

2
Theory

In this chapter we introduce the most important concepts used in our experiments. We also include explanations of certain definitions and assumptions that we make. We start off by providing some definitions and explanations of our interpretations. We then also introduce our running example and talk about some of the concepts used in our later experiments. The second half of the chapter focuses on literature and related studies which includes some additional proofs of concept and some research related to the concepts we mention earlier.

2.1 Background

In this section we introduce some of the definitions and explanations of expressions used later on in the paper. We then introduce our running example followed by some of the main concepts we use for our research.

2.1.1 Definitions

Here we provide the main definitions and their explanations that we use in this paper. While some of them are straightforward we want to ensure that their meaning is clear in how it is used in this papers context.

Software architecture - "Software architecture encompasses the set of significant decisions about the organization of a software system including the selection of the structural elements and their interfaces by which the system is composed; behavior as specified in collaboration among those elements; composition of these structural and behavioral elements into larger subsystems; and an architectural style that guides this organization. Software architecture also involves functionality, usability, resilience, performance, reuse, comprehensibility, economic and technology constraints, trade-offs and aesthetic concerns" [11]

Design patterns are generic solutions or approaches for certain software functionality or system design. Design patterns can be split into different categories that work for certain range of generic and even more specific problems. In our case we are interested in software design patterns related to privacy and security such as *authentication enforcer, message inspector or secure proxy* patterns (note that these are just example patterns). Design patterns allow software architects to create and design software more efficiently by using a large variety of already tried and tested solutions to their problems. These design patterns represent certain coding practices and solutions which allow you to directly trace certain patterns from architecture to the code and the other way around.

Function and/or *method* - depending on a programming language developers might be used to saying function or method. Here we use both terms interchangeably. When we use these terms we refer to a part of functionality that encapsulates some piece of code that's set to perform a specific task (which may consist of smaller sub-tasks and even call other methods).

Security and privacy related code - a piece of code, be it a single function or an entire class that contains some sort of logic which could be attributed to either security or privacy. One example of it would be encryption of a message (secrecy) or providing log-in functionality (authentication and authorization). More obscure examples of this would be accessing users profile to either read or alter information (privacy related) or negotiating connection/communication setting (security related). Below we show an example of code that we consider to be security related.

```java
String transportUri = getTransportUri();
if (transportUri != null) {
    Uri uri = Uri.parse(transportUri);
    localKeyStore.deleteCertificate(uri.getHost(), uri.getPort());
}
```

Listing 2.1: Security related code example

In the above example the code handles functionality related to network connectivity and certificate use. These certificates can be important in that they allow the application to make certain connections and communicate with the outside world. For this reason we consider code like this to be security related.

Another example below is of code that is not security related.

```java
public class AccountStats implements Serializable {
    private static final long serialVersionUID = -5706839923710842234L;
    public long size = -1;
    public int unreadMessageCount = 0;
    public int flaggedMessageCount = 0;
    public boolean available = true;
}
```

Listing 2.2: Unrelated code example

In another example above we have a piece of code that acts as a simple object that stores information. All of its variables are public with the only exception of an ID variable. Furthermore this class has no other functionality (methods) and can only store data. Since none of the variables in this class hold any private, sensitive or security related data we consider this to be an unrelated piece of code.

Granularity - Granularity refers to the unit of size at which we have chosen to analyze the code at. In this case we use *.class* files, or simply *JAVA* classes as our unit and level of granularity. The reason for this decision is because classes are an element of structure of design. Since we are trying to use software architecture as our source of information, classes tend to be directly represented in the architecture which makes it fairly easy to relate architectural information to classes and the other way around. Additionally classes have many more characteristics and information that can be extracted from which makes them a solid starting point.

Feature or property - When we refer to a feature or a property (used interchangeably) when we address certain characteristic of the software architecture. Features and properties can be qualitative as well as quantitative. For example we may refer to the amount of classes as a feature of the architecture. Another such example would be relationships between classes or even a simple count of lines of code.

Prediction accuracy - In this paper we mention *prediction accuracy* or sometimes just *accuracy.* This refers to our results and machine learning algorithm ability to classify the data. In other words *prediction accuracy* shows a percentage of correctly classified instances. For example, a prediction accuracy of 80% would mean that machine algorithm in question managed to classify 180 out of 225 instances correctly. Alternatively the prediction accuracy depicts how many of the 225 classifiable instances were predicted correctly.

F-measure (F1 score) - "The F measure (F1 score or F score) is a measure of a test's accuracy and is defined as the weighted harmonic mean of the precision and recall of the test." [12]

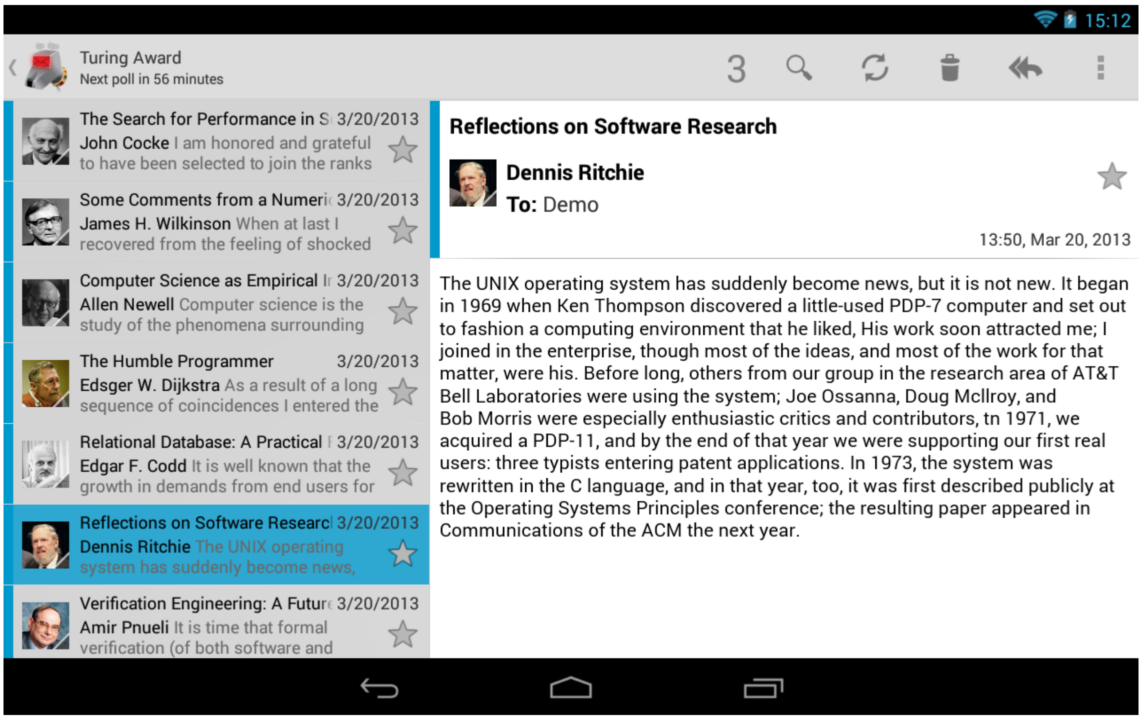
Matthews Correlation Coefficient (MCC) - "The Matthews Correlation Coefficient (MCC) has a range of -1 to 1 where -1 indicates a completely wrong binary classifier while 1 indicates a completely correct binary classifier. Using the MCC allows one to gauge how well their classification model/function is performing." [13]

2.1.2 The Running example

K9-mail is an open source android application that provides e-mail services and allows users to use encryption such as OpenPGP. Since K9-mail is an android application it is written in Java programming language and all of its documentation and source code can be found on its GitHub repository. K9-mail project itself is split into two major sections namely *k9mail* and *k9mail-library* where the former consists of more front-end and the latter of back-end nature. In this thesis we focused our attention on the *k9mail* source code portion of the application because it has more varied code (in terms of back-end and front-end functionality). Another motivation for choosing this part was because the ground truth we obtained is also based on *k9mail* and not on *k9mail-library.*

Figure 2.2 shows a simplified overview of K9-Mail applications architecture [14]. The overview shows how K9-Mail application is constructed. It is also possible to relate this architecture overview to some of the K9-Mail source code directly. We use this overview mostly to better familiarize ourselves with K9-Mail and how it works.

The portion of K9 source code that we will be using for our experiments contain 225 Java *.class* files. 165 of these files have been classified as containing security related code and 55 files that don't contain anything that we consider to be security or privacy related code. In those files we have 354 total JAVA classes and sub-classes out of which 264 have been classified as security relevant and the remaining 90 classes



15:12

Turing Award
Next poll in 56 minutes

3

The Search for Performance in S 3/20/2013
John Cocke I am honored and grateful to have been selected to join the ranks

Some Comments from a Numeri 3/20/2013
James H. Wilkinson When at last I recovered from the feeling of shocked

Computer Science as Empirical I 3/20/2013
Allen Newell Computer science is the study of the phenomena surrounding

The Humble Programmer 3/20/2013
Edsger W. Dijkstra As a result of a long sequence of coincidences I entered the

Relational Database: A Practical 3/20/2013
Edgar F. Codd It is well known that the growth in demands from end users for

Reflections on Software Researc 3/20/2013
Dennis Ritchie The UNIX operating system has suddenly become news,

Verification Engineering: A Futur 3/20/2013
Amir Pnueli It is time that formal verification (of both software and

Reflections on Software Research

Dennis Ritchie
To: Demo

13:50, Mar 20, 2013

The UNIX operating system has suddenly become news, but it is not new. It began in 1969 when Ken Thompson discovered a little-used PDP-7 computer and set out to fashion a computing environment that he liked, His work soon attracted me; I joined in the enterprise, though most of the ideas, and most of the work for that matter, were his. Before long, others from our group in the research area of AT&T Bell Laboratories were using the system; Joe Ossanna, Doug Mcllroy, and Bob Morris were especially enthusiastic critics and contributors, tn 1971, we acquired a PDP-11, and by the end of that year we were supporting our first real users: three typists entering patent applications. In 1973, the system was rewritten in the C language, and in that year, too, it was first described publicly at the Operating Systems Principles conference; the resulting paper appeared in Communications of the ACM the next year.

Figure 2.1: K9 Mail application user interface [1]

were deemed to be unrelated to security or privacy. This gives us approximately 3/4 security related and 1/4 of security unrelated files as well as classes to work with.

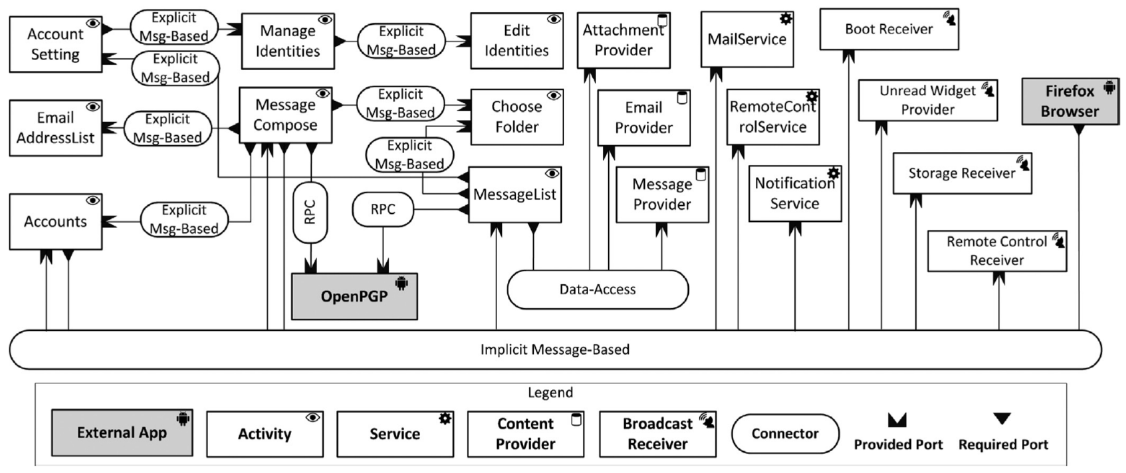


Figure 2.2: K9 Mail application architecture overview

2.1.3 Software Architecture, metrics & patterns

In this section we briefly describe software metrics and what software metrics we will be using in our experiments. We also include short descriptions and explanations of what these metrics represent.

Software metrics - Software metrics are a standard of measure. These metrics provide a quantifiable value of a process or a property that the software posses. Software metrics can refer to both qualitative and quantitative properties such as robustness (qualitative) or lines of code (quantitative).

We use the following software metrics that were extracted by the SourceMonitor tool:

- *lines* - total amount of lines in any given file (including code, comments, white-spaces and anything else in between).
- *statements* - any bit of code that accomplishes some kind of action be it assigning a variable, calling another method or performing a calculation.
- *branches* - branches represent alternative paths that a program could take to execute itself or assigned operation. Branching involves some kind of *if-else* statements or similar logic.
- *calls* - number of total calls found. A call represents an execution of another routine also known as function calls or method calls.
- *comments* - comments are parts of code which are not executed by the program and are meant for documentation or explanation purposes.
- *classes* - "A class is nothing but a blueprint or a template for creating different objects which defines its properties and behaviors" [15]. This metric counts A total number of classes and their sub-classes within analyzed file.
- *methods per class* - this metric shows total amount of functions (or methods) in the class.
- *average statements per method* - this metric compiles an average of statements in each method within a given class.
- *maximum and average complexities* - this metric refers to cyclomatic complexity that shows how many different independent execution paths a single function or method can take. It shows either maximum or average complexity for each file. Note that *branching* is very similar to complexity.
- *maximum and average depth* - depth (code nesting) represents the total amount of loops within other looping statements such as multiple nested if-else statements. Maximum depth shows the deepest found nest within a class while average depth compiles an average of all nested statement depths.

2.1.4 Data-flow and Taint Analysis

Data-flow analysis is a technique that allows you to gather information about data created and its flow in the program [16] [17]. For this purpose a control flow graph (CFG) is used that helps to visualizes the flow and determines at what part of the system changes occur. CFG is also able to show where the data propagates and how it changes throughout its flow. CFG technique is useful for tracking data with regards to optimization and more efficient code execution. CFG technique can also be used to gather information about what kind of data is being generated in the program and how it is being used. Such information can reveal whether there is any sensitive information being transferred between certain points of program or where such data ends up at among other things.

Similar to data-flow analysis, taint analysis also focuses on data flow within an application [18] [19] [20] [21]. One major difference is that taint analysis focuses on the way data is affected by other sources or factors, and how it can be changed in different parts of the program. While data-flow analysis is more focused on efficiency and flows, taint analysis looks at specific instances where this data can be changed not only within the program but also in some instances by outside sources. Taint analysis is heavily security oriented and provides a type of vulnerability detection which points out which parts of the program are vulnerable and where data taints can happen.

As with most analysis techniques both data-flow and taint analyses can be done dynamically and statically. While dynamic analysis requires the code to be run, static analysis can inspect either the entire source code or just specific instances of it such as classes or certain libraries as well as code that may not be complete.

2.1.5 Machine Learning & Algorithms

Here we introduce the machine learning concept and explain what kind of machine learning algorithms we will be using for our experiments. We also include short and brief explanations of each algorithm that we intend to use.

Machine learning - “Machine learning studies computer algorithms for learning to do stuff. We might, for instance, be interested in learning to complete a task, or to make accurate predictions, or to behave intelligently. The learning that is being done is always based on some sort of observations or data, such as examples (the most common case in this course), direct experience, or instruction. So in general, machine learning is about learning to do better in the future based on what was experienced in the past.” [22]

In this thesis machine learning is used to analyze and predict which parts of the system may contain security and privacy sensitive code. For this purpose we are using machine learning tool WEKA and several of WEKAs provided algorithms:

- *ZeroR* - (found in the "rules" category) “ ZeroR is the simplest classification method which relies on the target and ignores all predictors. ZeroR classifier simply predicts the majority category (class)” [23].
- *PART* (Predictive ART or ARTMAP) - (found in the "rules" category) This algorithm “ Autonomously learns to classify arbitrarily many, arbitrarily ordered vectors into recognition categories based on predictive success. This supervised learning system is built up from a pair of Adaptive Resonance Theory modules (ARTa and ARTb) that are capable of self-organizing stable recognition categories in response to arbitrary sequences of input patterns. ” [24]
- *OneR* - (found in the "rules" category) “OneR, short for "One Rule", is a simple, yet accurate, classification algorithm that generates one rule for each predictor in the data, then selects the rule with the smallest total error as its "one rule". To create a rule for a predictor, we construct a frequency table for each predictor against the target. It has been shown that OneR produces rules only slightly less accurate than state-of-the-art classification algorithms while producing rules that are simple for humans to interpret. ” [25]

- *DecisionStump* - (found in the "trees" category) “A decision stump is a Decision Tree, which uses only a single attribute for splitting. For discrete attributes, this typically means that the tree consists only of a single interior node (i.e., the root has only leaves as successor nodes). If the attribute is numerical, the tree may be more complex.” [26]
- *J48* - (found in the "trees" category. This algorithm is an implementation of ID3 and is being developed by the WEKA developers. It is a decision tree based algorithm that works by measuring information gain in each data set and then using this to build a decision tree and classify the data.
- *RandomForest* - another classifier from the "trees" category. This algorithm works by creating multiple decision trees during run-time and then outputs the most often occurring (mode) result of these classifications.
- *RandomTree* - (found in the "trees" category) RandomTree algorithm works essentially by building a decision tree based on random parameters and then classifying data based on it. Unlike RandomForest algorithm it constructs only a single decision tree as opposed to many.
- *REPTree* - (found in the "trees" category) This classifier builds a decision tree based on information gain or variance and prunes it by using reduced-error pruning.
- *LogitBoost* - (found in the "meta" category) This is a boosting classification algorithm that performs an additive logistic regression.
- *MultiClassClassifier* - (found in under the "meta" category) This classifier works by assuming that each classifiable instance can be assigned only one single classification (label) at a given time.
- *BayesNet* - (found in the "bayes" category) “Probabilistic graphical model (a type of statistical model) that represents a set of random variables and their conditional dependencies via a directed acyclic graph (DAG)” [27]
- *NaiveBayes* - (found in the "bayes" category) “The Naive Bayesian classifier is based on Bayes’ theorem with independence assumptions between predictors” [28]
- *Logistic* - (found in the "functions" category) This algorithm performs logistic regression for its classification.
- *kStar* - (found in the "lazy" category) “K* is an instance-based classifier, that is the class of a test instance is based upon the class of those training instances similar to it, as determined by some similarity function. It differs from other instance-based learners in that it uses an entropy-based distance function.” [29]

Most of these algorithms were selected at random from each provided category (found in WEKA) in order to have some variation and see the differences and benefits that each algorithm provided.

Machine learning can be split up into two distinct phases: learning and predicting. The learning phase consists of chosen algorithm analyzing the data in order detect patterns and to create a rule set by which it could then make its own decisions. This learning phase is extremely important for prediction accuracy in the future and therefore quality of the data set used can play a big role in this. In our case we will be using data sets containing software architecture features and code metrics. Therefore it is important that we are always consistent with our measurements and values so that we can be certain that all of the patterns (identified by the algorithms) in the data set actually represent that of the architecture.

The algorithm prediction phase is fairly simple compared to that of learning. In a prediction phase algorithms analyze the provided data set and classify it based on the rule sets derived from the learning stage. In cases where provided data does not fit any of the existing rules, machine learning algorithms most of the time will try to identify best matching rule although such behaviour depends on the algorithm in use.

Algorithm learning and prediction can be done in several ways, however, we will be using 2 different approaches: *cross validation* and specifying what portion of data set should be used for training and predicting.

Cross validation - “In n-fold cross-validation, the original sample is randomly partitioned into n sub-samples. Of the n sub-samples, a single sub-sample is retained as the validation data for testing the model, and the remaining n - 1 sub-samples are used as training data. The cross-validation process is then repeated n times (the folds), with each of the n sub-samples used exactly once as the validation data. The n results from the folds then can be averaged (or otherwise combined) to produce a single estimation ” [30].

Unlike cross validation, percentage split allows you to specify exactly what portion of the data set you want to use for which stage. Once the ratio is decided the data set is randomly shuffled and new data sets produced for machine algorithms to use. All of this is done only once and each experiment will be using slightly different data sets due to random shuffle at the beginning of classification.

2.2 Related Work

In this section we discuss some of the related work. We start by introducing the main concept of class roles. Then we continue by mentioning existing work related to different analysis methods which we will be using in our experiments.

2.2.1 Class roles concept

One of the underlying concepts behind this thesis is that software architecture inherently provides information about the finished version of software and its code. This is also one of the basic theories that R. J. Wirfs-Brocks’ proposes [31] in her paper. Wirfs-Brock argues that each individual class has some elements such as class, method or variable names that imply certain functionality. Then Wirfs-Brock argues that using this information and further inspecting the code it is possible to see certain patterns and features that emerge from each class. Using such information Wirfs-Brock assigns a certain role type for each analyzed class. Class roles that Wirfs-Brock assigns serves a certain general purpose that is somewhat distinct from the other roles. The roles (also known as stereotypes) that Wirfs-Brock proposes are: *Information holder, Structurer, Service Provider, Controller, Coordinator* and *Interfacer.* Any class should fit at least one of the proposed stereotypes with most classes having only a single well defined role. This of course is an ideal view of such characterization. Wirfs-Brock also argues that more experienced developers tend to blur the lines between each class and its functionality and make them as flexible as possible in that aspect. In such cases classes become more ambiguous and their roles become much more difficult to clarify. With ambiguous roles without any clarification it becomes possible to assign more than one role to each particular class. This certainly makes the process of characterization more complex but still provides some useful insight about the code.

2.2.2 Text-based searches & analysis

Another study currently in a review process (obtained via private communication [32]) evaluated four different text-based program comprehension techniques in order to see how effective they were at locating security related code. In this study the same K9-mail application was used as a running example and its classes were assigned labels related to their function from a security point of view. Then the four techniques (*grep, Aromatic Keyword-Based Extraction, Latent Dirichlet Allocation, Supervised Latent Dirichlet Allocation*) were used in order to analyze the code and predict its assigned label. Depending on the label assigned and the technique used to predict it the results were varied but mostly positive and indicated that key-word based prediction can be fairly accurate. Some of the issues with these techniques were that in order to be more effective it was necessary to have better structural insight of the software which is not possible with these techniques. Despite this though their most effective technique turned out to be a simple *grep* combined with a classifier which proves the usefulness of machine learning algorithms.

2.2.3 Data-flow and taint analysis

Edward J. Schwartz et. al [18] have published a paper giving a good introduction into dynamic taint analysis. In this paper Schwartz introduces various concepts around taint analysis and related vulnerabilities. There Schwartz explains in details how taint analysis works by giving several examples and providing definitions and policies that are used to detect data taints. These examples with formal definitions provide an excellent explanation of how taint analysis works and what it can be used for.

A study published in 2014 by S.Arzt et. al [20] studied data-flow in Android applications and potential detection of security flaws that come with it. The study focused on taint analysis using *FlowDroid* and *DroidBench* tools for their analysis and result evaluation. The study evaluated the effectiveness and efficiency of various commercially available tools to that of *FlowDroid* and found that *FlowDroid* was able to easily outperform them and offer great accuracy with high taint detection.

Another set of studies [21] [19] looked into bigger picture of Android applications and analyzed larger sets of Android applications to detect taints. W. Kliebers [19] study focused on potential taints that happen between different applications using static analysis methods. Similarly Z. Yang [21] uses the same static analysis methods to detect taints in a much larger set of applications. In both cases static analysis techniques managed to identify a large set of leaks (potential data taints) and localize their origins.

Using knowledge gained from these previous studies we can identify possible uses of taint analysis in order to detect certain leaks with an intent to locate the source code causing them. Since taint analysis has security relevance and identifies flaws in the code that most often lead to certain types of security issues we can assume that detected leaks would lead to certain security sensitive code snippets.

2.2.4 Software architecture pattern detection

Software architecture unarguably provides information about software that captures some of the rationale behind certain decisions and so architectural design patterns captures this rationale really well. K. Keller [6] argues exactly that in his study and uses several reverse-engineering techniques in order to find these design patterns. The study concludes by a successful identification of many of the design patterns found in three large software systems. However, in extremely large and complex projects detection of all of the patterns turned out to be simply unfeasible due to their size and time required to do so. This was also mostly due to a manual nature of such pattern detection.

Another later study by D. Heuzeroth [33] used static and dynamic analysis methods in order to detect a selected set of design patterns. This study showed that a successful and automated pattern detection is possible at least for well known and defined patterns. Even non-standard pattern detection is possible by modifying current known pattern detection templates used in the study. This mostly concerns static analysis with dynamic analysis being somewhat more difficult due to the lack of knowledge and structure definitions. In either case design pattern prediction is possible unlike in the previosuly mentioned study.

Finally another study published in 2006 by N.Tsantalis et. al [34] shows some works with software design patterns and their identification in code. This study attempted to detect a known set of patterns in the given software by using a similarity scoring algorithm. After carefully working out the steps necessary for pattern identification the study was successful in creating a tool that managed to identify all of the defined patterns with very few false negatives and no false positives. The main difference and advantage of this study was the use of similarity scoring algorithm which allowed automatic detection of certain patterns even if there were slight deviations in their implementation.

All of these studies show that combining various techniques and approaches it is possible to detect software design patterns. With some modifications it should also be possible to detect lesser known patterns that tend to deviate from their specification in code implementation. This should make it possible to automatically detect even security related patterns that we are interested in.

2.2.5 Feature location in source code

Feature location is another concept directly related to our topic. Feature location utilizes various techniques in order to analyze the source code and identify features as well as patterns leading to them. A good example of this is a study by T. Eisenbarth

[10] where both dynamic and static analysis techniques are applied in order to locate certain features. By largely automating the entire process with dynamic and static analysis techniques the study also introduces a conceptual mathematical technique in order to investigate binary relations. These binary relations are used to derive correspondences between computational units (sections of code) and features. Such combination of analysis techniques proves to be quite efficient at locating features even if the source code is large and complex.

A good collection and summary of feature location techniques is described by B. Dit [35]. In there B. Dit presents a systematic literature survey of more than 80 articles from different sources. The study includes reviews of dynamic, static, textual, historical and other types of analysis techniques used to locate features. This study also takes time to review the results of each reviewed article providing a quick overview of the effectiveness that each respective analysis technique has in the provided context.

While we are not directly interested in feature location ourselves some of the techniques used can certainly be modified and applied in our own approach. More importantly some of the feature location techniques can serve as an inspiration and a starting point in deriving our own analysis methods to apply on software architecture.

3 Methodology

In this chapter we discuss our methods and procedures used in our experiments. We detail all of the approaches and argue why we deem them interesting and what potential outcomes they can have. Starting with a visualization of our intended approach we continue by explaining some of the metrics that we will use. We also explain how we intend to collect the data and what our data-set will consist of. We end this chapter by mentioning how we will use machine learning and some other approaches related to data extraction.

3.1 Approach and the Data-set

In this section we include a simple diagram showing our intended approach to this project. We also include a description and explanation of the data set that we intend to use as well as explanation about metrics and additional features in the data-set.

3.1.1 Approach visualization

Figure 3.1 shows our overall approach to this project. We start by defining some of the features and what data we want to extract. We then use these definitions and apply them on k9mail source code in order to extract corresponding metrics. We also extract some of the basic source code metrics and add them to the data-set, together with feature metrics and ground truth. Once this is done we use the data-set with machine learning algorithms and get our results. We then analyze these results and refine corresponding features and data set in order to achieve better prediction accuracy. We do this until we collect enough data and results about both, features used and their effect on the machine learning algorithms.

Generally the entire work-flow is fairly simple and is easy to replicate. We start by identifying an interesting feature or metric that we would like to extract. Then we proceed by either extracting it using already existing methods or come up with our own. If this doesn't work out we pick a new feature to work with. Once feature data is extracted we merge it to our already existing data set. After the merge we feed the data to machine learning algorithms for training and prediction. After both training and prediction is completed we analyze the results and record anything of interest. We then repeat the entire process until we run out of time or ideas.

The purpose of more complicated features that we will introduce later on is to introduce a wild-card variable (read feature) in our data-set. Well known metrics and some basic features can be predictable and may not give desired results. Introducing

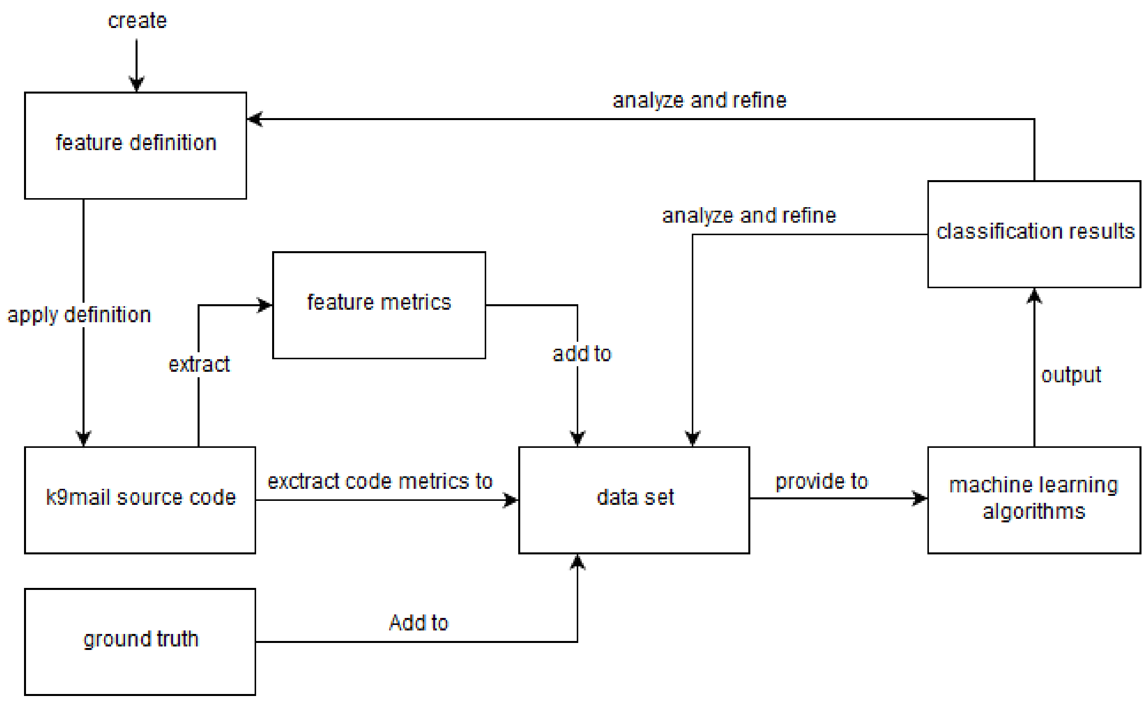


Figure 3.1: A simplified visualization of our methodology

a variable that we know very little off might help to stir the data enough to produce some interesting outcomes. Therefore we take a look at some of the features by investigating ways of obtaining them and even attempting to extract some data.

3.1.2 The Data-set

Our data-set contains a list of all classifiable (read *.class* files) instances. This means that our data-set contains an entire list of *.class* files with their corresponding name as an identifier. Each instance has all of the feature values assigned to it in a numerical form with the only exception being ground truth which holds a boolean *true* or *false* value.

Since Wirfs-Brock class roles initially have textual representation we have assigned a unique numerical value to each defined role. This is needed so that machine learning algorithms could effectively differentiate between the class roles. The values were mapped as follows:

1. - Controller
2. - Service Provider
3. - Structurer
4. - Information Holder
5. - Interfacer
6. - Coordinator

Such class role conversion allows us to easily identify what role each instance has been assigned and use this with machine learning algorithms. It is important to note though that values we assigned are arbitrary and have no other purpose and does not affect the results. It is also worth noting that these values can be any number as long as each role has a unique value that it could be identified by.

3.1.2.1 Basic metrics

Since we are trying to experiment with software architecture features in order to see if some features could be used for code detection, we need some kind of benchmark. For this purpose we are going to use some basic software code metrics that can be extracted by an already existing automated tools such as the SourceMonitor. We already know that software metrics can be extracted from any kind of source code and so we don't have to worry about reliability of such extracted metrics. By using basic metrics with machine learning algorithms we will be able to get our baseline results. Having baseline results will allow us to measure the effectiveness of our machine learning algorithms. While even basic metrics alone could have some effect on the prediction accuracy and machine learning, we will be able to add additional features to measure how the accuracy changes based on the features used.

Granularity level for all of the extracted metrics is set at the class level. This is chosen because classes still contain some architectural information about the code and can still be analyzed at a more detailed level (such as methods or lines of code). It is also convenient since a lot of the automated tools for data extraction use class granularity to extract a lot of additional metrics (ex. avg. complexity, avg. depth etc.).

It is important to note that we do not leave out any of the extracted basic metrics. This is done to preserve any potential impact extracted metrics may have on our machine learning algorithms and their prediction accuracy. By excluding some of the metrics we could inadvertently remove an important metric that could have had an unforeseen effect (be it positive or negative). Therefore when we extract our basic metrics we include as many as possible and reasonable to extract by using available tools. By doing this we are also able to get some answers to our RQ1 and RQ2 and to better plan our next step by setting some expectations based on obtained data.

3.1.2.2 Advanced features

Advanced features in this case refers to features that are extracted from software architecture and require more attention and intervention from our part. Advanced features include so called experimental variables which we define and attempt to extract in a procedural way.

We start with additional metrics such as *libraries in use.* In JAVA much like in other programming languages it is possible to include external source code called *libraries.* We can use this as an *included libraries* metric and even refine it to a more specific *included security libraries* metric which shows how many of the libraries contain security related functionality. This will allow us to see if there are any identifiable relations between the amount of additional source code (read *libraries*). included and likelihood of that particular instance to be security related.

Another feature that can be included is based on keywords. By searching for specific terminology and expressions we could quantify our findings and feed them to a machine learning algorithm. Assuming that selected keywords have relation to security it could help machine learning algorithms to better predict whether source code contains any security specific functionality. This would be very similar to our previously mentioned *included libraries* feature.

Additionally we have the class role [31] concept. Class roles are tied to software architecture and represent an innate functionality of an entire *class*. This could mean that by using class role as a feature and quantifying it we could be able to better predict whether a class with a specific role contains any security related code. Further more we could improve such class roles and create our own role definitions in order to describe a class role in terms of its security functionality and context.

There may be more features that can be extracted from software architecture that can help us with security relevant source code detection. However it is impossible to predict them all and thus we limit ourselves to the above mentioned features as our initial experimental set. Additional features could be extracted using but not limited to techniques such as taint analysis or software architecture pattern detection.

Extracting additional advanced features helps us answering RQ3. By looking into broader variety of features we can see whether our results improve or worsen in comparison with previously mentioned basic metrics. This directly relates to RQ3 and allows us to answer which features have any effect on the results as well as how big of an effect they have.

3.2 Data Collection

In this section we explain what approach we took when extracting metrics and other data needed for our experiments. We explain our approaches, the reasoning behind them and extraction of certain features.

3.2.1 Extracting initial feature set

In order to collect the necessary data first we need several things:

- K9-mail source code
- A list of categorized classes
- Ground truth

K9-mail source code is freely available for download to anyone from GitHub [36].

Once we have our source code we can start looking into it and understanding it better. Having source code leads us to being able to classify it using Wirfs-Brocks class roles [31] and create the initial data set to work with. In order to avoid some subjectivity role classification has been done by 3 reviewers (2 experts and one student in Software Engineering field) independently. Then the results were compared and discussed before a final decision was made on which roles to assign to produce the final data set.

Since these class roles are one of the few major factors that are deemed to be useful in code detection we have spent some additional time studying them. In order for each reviewer to better understand and familiarize with both roles and the code we have spent additional time using these class roles by classifying some randomly selected source code of K9-mail. Once individual classification has been done each reviewer shared their results and explanations of their motivation for each instance classification. Most of the discussion time was spent on cases where different roles would be assigned for the same part of source code to eliminate any differences in reasoning and interpretations. Initial disagreement rate was 24.5% meaning that 57 out of 233 classes were assigned different roles and the remaining 176 out of 233 would have the same role assigned by all 3 reviewers.

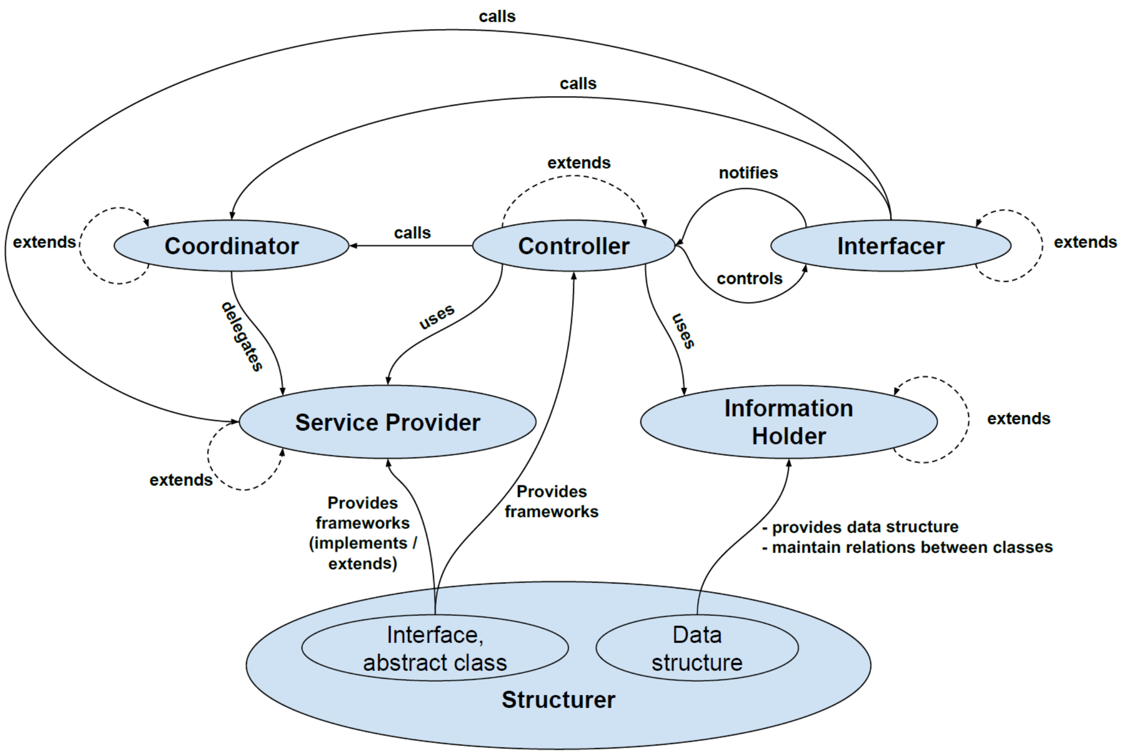


Figure 3.2: Class role relationship diagram created by T. H. Quang [2]

In addition to do better mock classifications a relationship diagram was created (figure 3.2). The diagram was created in order to better understand how each role relates to other respective roles and how class roles relates to the used source code.

Only after several rounds of classifications each reviewer received a final set of 200 semi-randomly selected classes. Since the classes were selected randomly and we had a limited set to work them naturally some of the selected ones were already used in our mock classifications. In the final set 150 of the classes were unique to each reviewer while the remaining 50 were assigned to others as well. None of the reviewers knew which classes were unique (assigned only to them) in order to eliminate any potential bias. These 50 classes were then used as a control to compare overall disagreement between assigned roles. In case the disagreement was too great (more than 20% of assigned roles were different) all of the classes would be re-assigned and re-classified for more accurate results.

The ground-truth has been obtained from another (so far unpublished [32]) study which had several security experts look at the code and decide which classes contained security and privacy relevant code. As far as our knowledge goes a similar strategy was used as with our class roles. We do not modify the ground-truth that we obtained in any way and use it as is in order to keep the consistency.

Finally we obtain our source code metrics by using SourceMonitor tool with all of its default settings. By default SourceMonitor provides us with a total of 12 different metrics to work with. Additionally SourceMonitor has a built-in change tracker which allows us to see any changes in code throughout its development if we want to track such information. Even though we have no intentions on modifying code this allows us to compare different versions of K9-mail since its source code is readily available on GitHub including many older and even versions.

After classification and basic metrics extraction we have our data set that is composed of the following features:

- *Security relevance*
- *Class role*
- *Lines*
- *% Statements*
- *% Branches*
- *Calls*
- *% comments*
- *Classes*
- *Methods per class*
- *Average statements per method*
- *Average complexity*
- *Maximum complexity*
- *Average depth*
- *Maximum depth*

This is our initial feature set which will be expanded as more data and features become available from further experiments.

3.2.2 Java libraries as a feature

In addition to source code metrics we also used *Java libraries in use* as an additional feature. For *Java libraries in use* feature we counted all of the libraries imported by a class and assigned a corresponding value to that instance. One of the reasons for this is that some of the libraries can be considered inherently security related and thus an assumption can be made that if such library is imported, the class will contain some security sensitive code. Finally some of the *import* statements include local (within the project) classes which could be accounted for. Similarly to security related imports, it also could indicate certain tendencies and relationships that otherwise might be overlooked.

We count and assign all of the libraries by using a custom script (see Appendix A) that crawls through the code and looks for any occurrences of word *import.* Since K9-mail is a *JAVA* application all of the libraries are loaded using *import* statement (see an example below).

Also, in order to avoid comments or any other irrelevant statements, we manually look through results and identify any false positive matches and discard them. The keyword list that has been used for extracting security related libraries is included in an Appendix A.

```java
    import java.security.cert.CertificateException;
    import android.content.ContentResolver;
    import com.fsck.k9.provider.EmailProvider;
```

Listing 3.1: Java import statement examples

The example above shows how some of the import statements would look like in our analyzed source code. From that example we can see that each statement starts with a keyword *import* which we then use in our script to detect each such line of code.

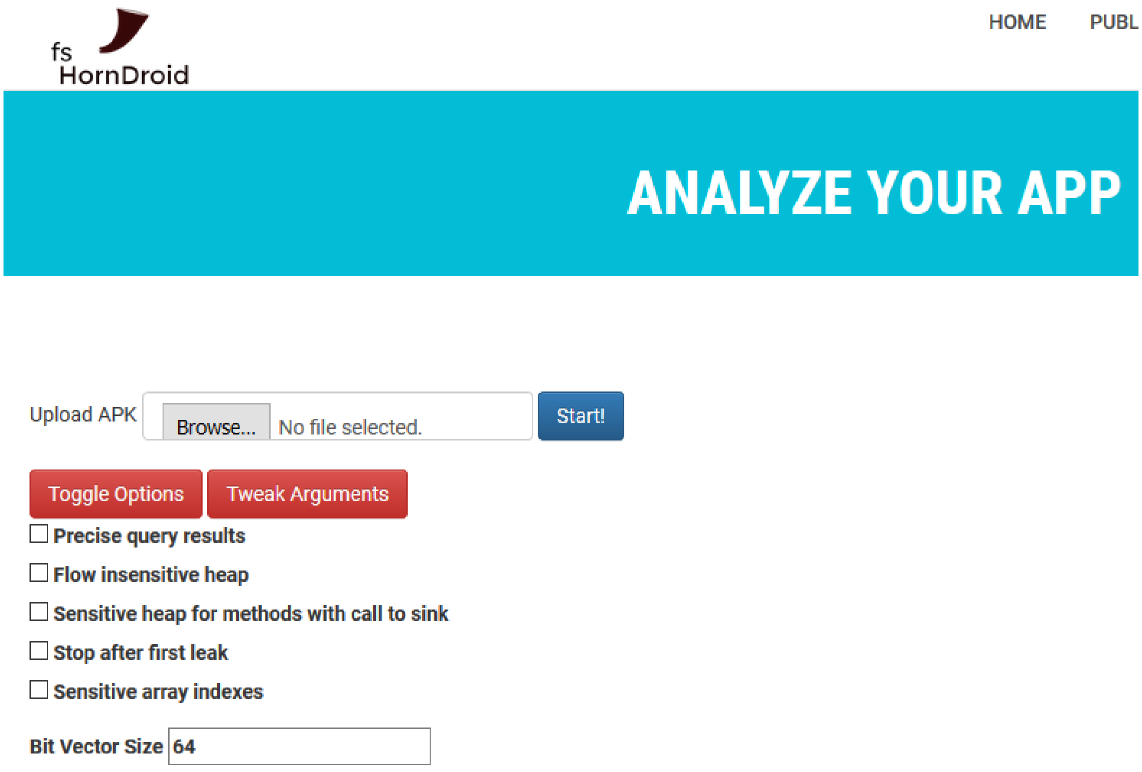
3.2.3 Keywords as a feature

Since there's indication that keyword and text based analysis can be used to detect security sensitive code with fairly decent accuracy we also try to use a similar approach. One of the challenges with keyword based approach is that it requires us to compile a list of keywords that would be relevant for the task at hand. In this case we have crafted our own custom keyword list partially based on existing Java libraries that we know they provide security functionality (see Appendix A for details). Our rationale behind this is that assuming that the library is in use and not included by accident or left out from older functionality, the class that uses it should perform some security or privacy related functionality. We also add some additional security related terms and informal expressions extracted from various reports and online sources related to security. The additional terms and expressions were chosen individually by a student with experience in computer security field. The reason for this is to introduce a little bit more flexibility in our search as well as to have more potential matches.

We use keyword list by crawling through the code and counting any occurrences of these keywords in a similar way as we did with import extraction. This means that our search does not take into account context that keywords were found in as well as including keywords found in comments or dead code (unreachable or unused code). Once the crawling is done we assign match count to each *.class* file that they were found in. In order to test the impact of different set of keywords we also split the list into two parts and ran the scans twice. First run looked for keywords based on the Java libraries while the second run included both, Java libraries as well as misc security related words. The reason for this was to measure whether our selection would have any impact on the overall results or not.

3.3 Taint analysis

There are several tools available to perform data-flow analysis. In our case we have mostly focused on taint-analysis due to its security based nature and with an assumption that any leaks detected by it would be security related. Taint-analysis



☐ Precise query results
☐ Flow insensitive heap
☐ Sensitive heap for methods with call to sink
☐ Stop after first leak
☐ Sensitive array indexes

Bit Vector Size 64

Figure 3.3: fsHornDroid interface for analyzing .apk files [3]

works by following various data-flows in the application and detecting functions and processes that interact with specific data instances.

In some cases methods with different security permissions might be able to access the same data source and change it which would be considered data tainting. To discover such taints we can use either dynamic or static analysis techniques. Since dynamic techniques require to run the application in question we have decided to stick to static analysis techniques and tools which allow us to inspect only the parts of K9-mail that we are interested in rather than the whole thing.

One of the newer static taint-analysis tools, *fsHornDroid* [3], allows us to use its online platform for code analysis by simply uploading a source code to analyze (see figure 3.3). Having such a platform is convenient for us since we no longer need to install and configure the tool ourselves and we can get the same result without worrying about different installation configurations.

Additionally, we use another static analysis tool *FlowDroid* [37] which performs the exact same task, however, in this case we need to download not only its source code but also all of the dependency libraries. In both cases the tools provide information about any of the tainted (data) sinks if they identify any.

3.4 Pattern detection

Software pattern detection is a fairly well researched field with several studies proving that it is indeed possible to detect various patterns by analyzing software source code [34] [6] [33]. One caveat with most of the studies though, is that majority of the studies focus on more generic software patterns. While these studies are still relevant they are not as useful when it comes to security. The lack of security-focused studies makes it somewhat difficult to apply the same techniques without putting too much, if not all of the focus on pattern detection in order to identify our security-sensitive source code.

In either case software pattern detection works by reading whatever source you provide and then compares its existing methods and calls with a behavioral template constructed for each specific pattern. For example the *Facade* pattern would have one unique signature and the *Authenticator* pattern would have a different one. By detecting a set of pre-defined patterns we could associate each class file with either the amount of patterns detected or simply with specific patterns. If there is any relation between patterns and source code we could train our classifier to recognize this data and improve accuracy of our predictions. Architectural software patterns used in our case could be anything from more generic ones to the ones specifically tailored with regards to software security. The reason for this is that we do not know if there would be any noticeable relation between any of the patterns and source code without first testing it. Therefore we can not discard any of them even if our intuition might suggest otherwise.

In our case we attempt to use the knowledge gained from previous studies as well as some of the suggested tools in order discover if pattern detection would be possible on our running example and in what ways we could use the extracted information for security relevant code location.

3.5 Machine learning

This section describes steps taken in order to use WEKA and its provided machine learning algorithms. We talk about settings used and the rationale behind some of the methodology decisions.

3.5.1 Working with WEKA

For our classification, we use two different approaches: first, we use a 10-fold cross-validation [30] classification and then we run the classifier again using percentage split of 66% and 34% for training set and classification. We do this in order to compare the results and check which method gives best accuracy and predictions.

Since we have a total of 225 entries in our data, we can afford splitting it into smaller chunks and still expect it to do well with machine learning classifiers. To top it all of in best and worst case scenarios we alter cross-validation folds from 10 to anywhere between 1 and 100. We also change percentage split from 66/34 to 50/50 to check how our results change depending on these parameters. To achieve the absolute best results we also filter out some of the features and code metrics that we use. By doing so we can also measure the relative impact each feature used has on our classification.

We use WEKA by feeding it our data set and selecting machine learning algorithm together with either 10-fold cross-validation or percentage split. Initially experiments start with all of the extracted code metrics and additional features such as imported libraries, keyword found and security roles. We then run each classifier in different modes and record all of the results. We also run all classifiers by using only extracted metrics to get our baseline prediction result. Then we add each additional feature (parameter) to measure their relative effects to prediction accuracy based on our metrics only classification results. We also repeat the same process with best performing classifier in order to measure the relative impact of each source code metrics in the data-set. We do this by using a combination of best performing metrics and then removing single metric and measuring its impact on the result. This is done in order to compare regular metrics with our additional extracted features and measure their relative impact on the accuracy.

Using machine learning we are able to answer or at the very least to get an indication for our RQ1. To be more precise, using machine learning we can answer whether software architecture is able to provide any useful data to identify security relevant code in our context.

3.5.2 Vote casting

Since we are using multiple classifiers to predict and classify our data, we can create a voting system that decides the classification by a majority vote. By comparing each classifiers decision we count votes for security relevance (total of true and false outcomes) and reassign the majority vote to that class. By using vote casting we can potentially weed out less accurate classifiers, assuming that most of the falsely classified instances are specific to the algorithm in use and are not repeated by other machine learning algorithms. Therefore when vote casting we also select only five best performing classifiers and use their predictions only.

We wrote a custom script (see Appendix A) that reads each classifiers decision for specific instances. It works by giving each instance initial value of 0 which is either incremented by +1 for positive matches (classifier thinks the class is security relevant) and decremented by -1 for negative matches (unrelated classification). Since we are using an uneven number of votes (in our case 5 votes) we always end up with either positive or negative value which in the end represents a majority decision. We then compare our voted results with the ground truth and check if overall prediction accuracy has improved or not.

4

Results

In this chapter we discuss the results that we obtained from our experiments. We also include outcomes and reasoning of why certain approaches that we took didn't work out as expected. We start by describing what kind of data we used and define those sets. We then continue by explaining results obtained from experiments using specific data-sets that we described earlier. We then continue by going more in to detail about most successful results where we include truth tables and other related data. The chapter concludes by briefly mentioning all of the other attempts that didn't work out as well as was expected.

4.1 Machine learning results

In this section we discuss our results obtained while working with WEKA machine learning software. Here we mention most of the outcomes from our experiments including some variations of results omitting only those that repeat or have no significant value.

4.1.1 Feature sets and data set used

From our experiments we have derived several feature sets in order to compare impact of features in use. Below we define the feature sets that ended up being most informative and useful (table 4.1):

From here on we will refer to Feature set 1 as FSF (Feature Set Full), Feature set 2 as FSC (Feature Set Class roles), Feature set 3 as FSM (Feature Set Metrics), Feature set 4 as FSB (Feature Set Best).

In our feature-sets *security relevance* represents our ground truth and holds a boolean *true* or *false* value representing its state. *Number of libraries* and *Security libraries* hold a total number of libraries used as per definitions in Chapter 2. The rest of the features in the feature-sets represent code metrics with an exception of class role. *Class role* entry represents an integer value of a class role based on Wirfs-Brocks definitions as per mapping explained earlier in Chapter 3.

The data-set itself consists of a list of 225 JAVA *.class* files with all of the extracted features assigned to them. Then based on the feature set selected, corresponding feature data is selected and fed to the classifiers using WEKA. It is also worthwhile to note that we have a 73.3% vs 26.7% distribution of *.class* files containing security relevant code and those that do not. This means that by naively classifying all of the instances as security relevant we would get a prediction ac-

FSF	**FSC**	**FSM**	**FSB**
Feature set 1	**Feature set 2**	**Feature set 3**	**Feature set 4**
Security relevance	Security relevance	Security relevance	Security relevance
Number of libraries	Number of libraries	Number of libraries	Number of libraries
Security libraries	Security libraries		
Statements	Statements	Statements	Statements
Calls	Calls	Calls	Calls
Classes	Classes	Classes	Classes
Lines	Lines	Lines	
% Branches	% Branches	% Branches	% Branches
% Comments	% Comments		% Comments
Methods per class	Methods per class	Methods per class	Methods per class
Avg. statements	Avg. statements		Avg. statements
Max complexity	Max complexity		
Max depth	Max depth		
Average complexity	Average complexity	Average complexity	Average complexity
Average depth	Average depth		Average depth
	Class Role		Class Role

Table 4.1: Feature sets used for different experiments

curacy of 73.3%. This means that we can use 73.3% prediction accuracy as our control value. If we get a prediction accuracy above this value it would mean that the machine learning algorithm in use is able to correctly guess at least some of the instances. Anything below this value would indicate worse results and some underlying issue either with the data-set, machine learning algorithm or our methodology. A full data-set used with all of the results and extracted information can be found in Appendix A.

You may notice that not all of the features that were discussed earlier in the paper are listed in our feature sets. This is because while initially we did run experiments with other feature-sets containing more features, the results obtained from these experiments were lacking or the results were similar to those of feature-sets described earlier. By this we mean that none of the performed experiments provided any information that could be used and therefore these results were omitted. More about why other experiments didn't provide much useful information we discuss about later in the paper.

4.1.2 Feature set 1 results

We start off our experiments by extracting code metrics and compiling a data sheet representing each *.class* file. Using our FSF we were able to get baseline results which show what we can expect from other attempts, since this set includes most of the features. By performing two different sets of experiments, one with 10-fold cross-validation (experiment 1) and another with splitting the data set 66% for training and 34% for classification (experiment 2) we get our first set of results.

We display these results (table 4.2) sorted by classifiers accuracy (percentage of correctly classified instances). We also include Matthews correlation coefficient (MCC) [13] which ranges from -1 to 1 (higher positive value means better prediction) as well as F-measure that ranges from 0 to 1 (higher positive value represents more accurate predictions). Both MCC and F-measure values represent averages calculated from algorithms prediction performance in terms of true positive and true negative predictions.

Classifier	**Prediction Accuracy**	**Avg. MCC**	**Avg. F-Measure**
LogitBoost	75.1%	0.279	0.726
PART	75.1%	0.279	0.726
RandomForest	74.2%	0.229	0.707
ZeroR	73.3%	0.000	0.621
BayesNet	72.4%	0.025	0.631
DecisionStump	72.4%	0.025	0.631
REPTree	72.0%	0.139	0.676
Logistic	71.1%	0.003	0.630
MultiClassClassifier	71.1%	0.003	0.630
kStar	70.2%	0.210	0.696
RandomTree	69.8%	0.211	0.694
J48	69.8%	0.131	0.673
OneR	69.3%	0.029	0.640
NaiveBayes	49.3%	0.087	0.517

Table 4.2: WEKA results using feature set 1 (FSF). Sorted by best prediction accuracy in 10-fold Cross-validation (experiment 1)

From table 4.2 we can see that *ZeroR* classifier achieved 73.3% accuracy by labelling all of the instances as *true* for security relevance. Since *ZeroR* is a naive classifier it will always choose the most frequently occurring value and assign it to all of the instances. This means that since our data-set is dominated by security relevant instances, *ZeroR* will classify all of them as *true*. By doing this the accuracy achieved by *ZeroR* will directly correspond the percentage distribution of, in our case, security relevant instances. Due to this property and the fact that we do not change our data/set we use *ZeroR* classifier results as our baseline score.

The best achieved accuracy using FSF was 75.1%. In this case two of the classifiers managed to achieve this accuracy which is only 1.8% better than *ZeroR* classifier. Both of them however have a significantly better MCC (0.279) and F-Measures (0.726) meaning their predictions are somewhat more accurate (+0.279 MCC value and +0.105 F-measure). In either case we did not expect this feature set (FSF) to be good enough for accurate prediction since it used only basic source code metrics.

Table 4.3 displays the results of experiment 2 (training set split) with same the FSF. Here we see our baseline accuracy from *ZeroR* to be much lower at 60.5% which is caused by having a slightly different classification set and more even distribution of security relevant and irrelevant classes (due to the split). Best accuracy achieved was 68.4% as well being 6.7% lower than in our previous tests however the difference

Classifier	**Prediction Accuracy**	**Avg. MCC**	**Avg. F-Measure**
RandomTree	68.4%	0.310	0.663
RandomForest	64.5%	0.251	0.540
J48	64.5%	0.204	0.582
OneR	63.2%	0.177	0.600
kStar	63.2%	0.172	0.592
LogitBoost	63.2%	0.162	0.560
NaiveBayes	60.5%	0.063	0.515
PART	60.5%	0.000	0.456
ZeroR	60.5%	0.000	0.456
Bayesnet	60.5%	0.000	0.456
DecisionStump	60.5%	0.000	0.456
REPTree	60.5%	0.000	0.456
Logistic	57.9%	-0.070	0.464
MultiClassClassifier	57.9%	-0.070	0.464

Table 4.3: WEKA results using feature set 1 (FSF). Sorted by best prediction accuracy in 66/34 training set split (experiment 2)

between best and second best classifiers was much greater (3.9%) than in previous 10-fold tests (same result with 3rd best being only 0.9% less accurate). It is also worth to note that *RandomTree* classifier has similar performance even in our earlier test and the top performers of the previous test end up at the mid-field performance.

Another noteworthy difference between two sets of tests is that the *RandomTree* classifier had slightly better MCC value than the *LogitBoost* and *PART* classifiers however its F-measure was a bit lower. However by looking at overall trends it is fairly clear that 10-fold cross validation worked much better overall. It had higher F-measures and only somewhat better MCC values although they were less evenly distributed. In our second set of experiments we can see that performance gradually decreases together with MCC and F-measure values which is not the case in the first set of tests.

These results show us that it is possible to use some metrics for classification but they are not much better than pure guessing. Adding more features and modifying metrics might change the results.

4.1.3 Feature set 2 results

For our second experiment the same metrics data set has been used with a single addition of Wirfs-Brocks roles as an additional feature. This feature set (FSC) has been used with exactly the same classifiers and parameters as our previous experiment using FSF.

Once again we have our *ZeroR* classifier that acts as a baseline measure at 73.3% accuracy in 10-fold cross-validation experiments. However with an addition of class roles we reach best accuracy of 76.4%. This is an improvement of 3.1% compared to baseline accuracy as well as giving us a slight improvement of 1.3% over FSF results.

Classifier	**Prediction Accuracy**	**Avg. MCC**	**Avg. F-Measure**
LogitBoost	76.4%	0.318	0.739
RandomForest	75.5%	0.267	0.717
ZeroR	73.3%	0.000	0.621
REPTree	72.8%	0.152	0.678
BayesNet	72.4%	0.025	0.631
DecisionStump	72.4%	0.025	0.631
J48	71.5%	0.193	0.695
Logistic	71.1%	-0.021	0.624
MultiClassClassifier	71.1%	-0.021	0.624
kStar	70.2%	0.193	0.692
PART	69.3%	0.132	0.673
OneR	69.3%	0.029	0.640
RandomTree	68.4%	0.206	0.687
NaiveBayes	49.7%	0.073	0.523

Table 4.4: WEKA results using Feature set 2 (FSC). Sorted by best prediction accuracy in 10-fold Cross-validation (experiment 1)

Classifier	**Prediction Accuracy**	**Avg. MCC**	**Avg. F-Measure**
RandomTree	68.4%	0.310	0.663
J48	64.4%	0.204	0.582
RandomForest	64.4%	0.251	0.540
OneR	63.1%	0.177	0.600
kStar	63.1%	0.172	0.592
LogitBoost	63.1%	0.162	0.560
NaiveBayes	60.5%	0.063	0.515
PART	60.5%	0.000	0.456
ZeroR	60.5%	0.000	0.456
BayesNet	60.5%	0.000	0.456
DecisionStump	60.5%	0.000	0.456
REPTree	60.5%	0.000	0.456
Logistic	57.8%	-0.070	0.464
MultiClassClassifier	57.8%	-0.070	0.464

Table 4.5: WEKA results using Feature set 2 (FSC). Sorted by best prediction accuracy in 66/34 training set split (experiment 2)

We can see similar overall improvement for all of the classifiers. This improvement is supported by better MCC and F-measures as well. This indicates that class roles do have positive impact and help our classifiers to perform better. In both feature sets the *LogitBoost* classifier remains as best and most accurate algorithm outperforming the rest.

If we look at our experiment 2 results (table 4.5) there is no clear improvement over FSF. In this case there's no improvement in accuracy, MCC or F-measure values. It is odd considering the fact that class roles had a clear positive impact in

10-fold cross-validation classification. This leaves us with somewhat mixed results where we have some indication of class role positive impact but only in certain circumstances. It is hard to tell whether this improvement in accuracy can be attributed to usefulness of class roles or due to a more random nature of the classifiers that we used.

From here on now we will only include the results obtained from our 10-fold cross-validation experiments. The reason for this is that 10-fold cross-validation experiments are essentially the same as using 66/34 training split experiments except the split is 90/10 and we repeat it 10 more times. We have already obtained some basic data and can see that the results improve and are more accurate using 10-fold cross-validation. Therefore there is no need to continue running two different types of experiments.

4.1.4 Feature set 3 results

Since we got our baseline as well as class role results showing some improvement in accuracy we wanted to test if it was possible to further tweak feature sets in order to achieve even better accuracy by either manipulating classifier settings or changing some features that we use. We do this by at first changing the amount of folds and adjusting training set percentage split. By using FSM in experiment 1 environment (10-fold cross validation) we got improved results as can be seen in table 4.6.

Classifier	Prediction Accuracy	Avg. MCC	Avg. F-Measure
RandomForest	77.3%	0.338	0.744
LogitBoost	74.2%	0.244	0.713
ZeroR	73.3%	0.000	0.621
BayesNet	72.4%	0.025	0.631
DecisionStump	72.4%	0.025	0.631
REPTree	72.4%	0.150	0.679
PART	71.1%	0.166	0.686
kStar	70.2%	0.218	0.698
Logistic	70.2%	-0.108	0.605
MultiClassClassifier	70.2%	-0.108	0.605
J48	69.7%	0.077	0.656
OneR	69.3%	0.029	0.640
RandomTree	64.0%	0.103	0.644
NaiveBayes	51.5%	0.009	0.543

Table 4.6: Feature set 3 (FSM) results using 10-fold cross-validation (experiment 1)

From these results (table 4.6) we can see that we have managed to achieve 77.3% accuracy, which is better than any previously achieved results with or without class roles. This is largely due to our modification of metrics used in our data. For this set we have limited our metrics to: *number of imports, statements, calls, classes, lines, branches, methods per class, average statements per method* and *average complexity.*

Using only those metrics we achieved 4% better precision than our baseline accuracy as well as beating best metric only result by 2.2%. This accuracy is also 0.9% better than our best prediction accuracy using class roles.

4.1.5 Feature set 4 results

Classifier	Prediction Accuracy	Avg. MCC	Avg. F-Measure
LogitBoost	79.5%	0.414	0.770
RandomForest	74.6%	0.248	0.713
J48	74.2%	0.265	0.721
PART	74.2%	0.229	0.707
ZeroR	73.3%	0.000	0.621
BayesNet	73.3%	0.000	0.621
DecisionStump	73.3%	0.000	0.621
REPTree	72.8%	0.109	0.659
OneR	72.0%	0.117	0.667
RandomTree	71.1%	0.250	0.709
Logistic	70.6%	-0.033	0.621
MultiClassClassifier	70.6%	-0.033	0.621
kStar	68.0%	0.164	0.676
NaiveBayes	55.1%	0.137	0.577

Table 4.7: WEKA results using FSB in 20-fold cross-validation classification

In order to check whether previous results were simply a coincidence and class roles don't actually contribute that much for our prediction accuracy we ran another set of experiments where we include class roles in our classification but remove other metrics and change some classifier settings to improve prediction accuracy. By changing our classifier to 20-fold cross validation and filtering metrics provided to *class role, number of imports, statements, calls, number of classes, branches, comments, methods per class, average statements per method, average depth* and *average complexity*, we get the results shown in table 4.7.

As you can see (table 4.7) we have managed to reach 79.5% prediction accuracy using FSB. Unsurprisingly, *LogitBoost* classifier performed best even without a refined list of metrics and settings. In this best case scenario we achieve 6.2% better than baseline accuracy which is also 3.1% better than previously achieved results using FSC. This actually doubles the improvement in our prediction accuracy and clearly indicates benefit of using class roles as a feature. Furthermore if we look at MCC and F-measure values we can also note a clear improvement over any of the previously achieved results.

Since *LogitBoost* achieved best prediction accuracy we can also take a look at its confusion matrix (table 4.8). The confusion matrix shows us how well our algorithm performed in terms of true and false positives as well as true and false negatives.

From this matrix we can see that *LogitBoost* managed to classify almost all of the files containing security code correctly (approx. 95% accuracy) however it

Predicted True	**Predicted False**	**Actual:**
157	8	**True**
38	22	**False**

Table 4.8: LogitBoost confusion matrix using feature set 4 (FSB)

was significantly worse at predicting files without any security related code (approx. 63% accuracy).

Additionally we have an F-measure of 0.872 and MCC of 0.414 for all of the *TRUE* classifications. *FALSE* classifications had an F-measure of 0.489 and MCC of 0.414 (same as *true*). While MCC value remains the same for both types of classifications we can see that F-measure shows that *LogitBoost* is much better at predicting *true* instances. Even if we get a few more instances in *false* wrong it means that if we could achieve slightly better *true* classification we could use this algorithm. To be more precise this algorithm then could be used to discard all predicted *false* instances. Even though by doing so we would have some false-positives we could be sure that we didn't really miss any true-positives.

4.1.6 Individual feature impact results

We would like to find out how much of an impact each code metric that we used has on the overall accuracy. For this purpose we run another set of classifications. This time we do this by removing a single parameter from our data and measuring its relative impact on the accuracy. We run this experiment by using our previously best achieved accuracy settings and algorithm (*LogitBoost* in 20-fold cross-validation) and get the following results:

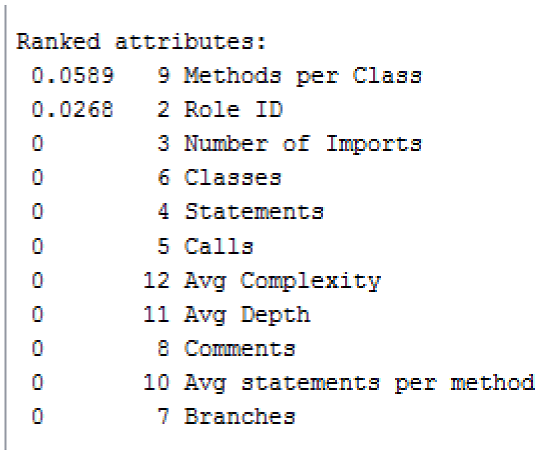
Parameter	*Accuracy impact*
Class role	4.89%
Methods per class	3.56%
Statements	3.12%
Branches	3.12%
Number of imports	3.00%
Average statements per method	2.67%
Classes	2.29%
Comments	1.78%
Average complexity	1.34%
Average depth	0.45%
Calls	0.45%

Table 4.9: Individual metrics relative impact to the prediction accuracy in best case scenario

From this (table 4.9) we can see that the class roles feature had an overall impact of 4.89%. This also is the greatest relative impact for all features. This of course doesn't mean that this is its absolute or actual (real) impact on the accuracy but rather relative one to. Even thought this shows us the change in accuracy using FSB in experiment 1 environment we can still gauge its potential effect even under different circumstances.

Our class roles are followed by methods per class, statements, branches and number of imports which all provide 3% or more relative impact to the prediction accuracy. This is quite interesting since we can see that even seemingly basic code metrics can have fairly measurable impact. Oddly enough though if we would remove our least impact-full features (average depth and calls) from this feature set we would find that our prediction accuracy plummets. This only solidifies idea that these accuracy impact are only relative to the environment they are being observed at.

Even though class roles seem to have the most impact, we can't attribute all of the importance to class role feature alone. As mentioned earlier this is rather due to the combination of all of the features used. However even keeping this in mind it is clear that class role feature is one of the more important factors that affect prediction accuracy. In addition we were unable to achieve any better results by any other feature combinations that did not include class role which certainly highlights its importance.



```
Ranked attributes:
 0.0589   9 Methods per Class
 0.0268   2 Role ID
 0        3 Number of Imports
 0        6 Classes
 0        4 Statements
 0        5 Calls
 0       12 Avg Complexity
 0       11 Avg Depth
 0        8 Comments
 0       10 Avg statements per method
 0        7 Branches
```

Figure 4.1: InfoGain results from WEKA

Additionally we ran *InfoGain* attribute evaluator from WEKA (table 4.1). This evaluator computes how much of an impact each feature has in decreasing the overall entropy (how much it helps to increase prediction accuracy). Greater value represents greater impact of that feature.

Interestingly enough by using this *InfoGain* attribute we see that the biggest impact on the results is done by *methods per class* feature rather than class role (Role ID). Even in this case we can still see class roles being second best feature however the calculated impact is still fairly small (0.0268). This may depend on the implementation of this *InfoGain* evaluator as well as due to the fact that we achieved our best accuracy by tweaking feature set as well as experiment environment. This proves that our results are not universal and more case specific.

From these results we can see that the least we should expect from our classifiers is a prediction accuracy of 73.3% which is fairly high but is caused by the distribution (true / false) of our data set. The best achieved prediction accuracy is 79.5% which gives us approximately 6.2% overall improvement which is quite fair considering the metrics used. However having only 80% accuracy may not be enough in real world environment where accurate predictions are needed. Missing 1/5 of potential results can be a sizable chunk of code that requires manual correction. The only case in which 80% accuracy would be beneficial is if our classifiers could achieve perfect prediction with either no false positives or false-negatives. This way at least we could be certain that at least part of the classification would require no more manual intervention.

4.1.7 True Positives and True Negatives

From all of these results it is worth to mention some of the algorithms that excelled at classifying instances as either security related or unrelated instances. Tables 4.10 and 4.11 shows these results. TP (True Positive) Rate shows how well the algorithm did at correctly identifying instances. FP (False Positive) Rate shows how many of the corresponding instances were miss-classified. In terms of TP Rate the higher the value, the better the result. For FP rate a higher value means more miss-classifications and so lower value is better. The rates were obtained from FSB feature set of experiments.

Classifier	**TP Rate**	**FP Rate**
ZeroR	1.000	1.000
BayesNet	1.000	1.000
DecisionStump	1.000	1.000
REPTree	0.958	0.900
Logistic	0.952	0.967
MultiClassClassifier	0.952	0.967
LogitBoost	0.952	0.633
OneR	0.927	0.850
RandomForest	0.921	0.733
PART	0.921	0.750
J48	0.891	0.667
RandomTree	0.812	0.567
kStar	0.794	0.633
NaiveBayes	0.521	0.367

Table 4.10: Algorithms TP and FP rates based on security relevant instance classification. Sorted by best TP Rate.

From table 4.10 we can see that most of the classifiers did really well in terms of True Positives. *ZeroR, BayesNet and DecisionStump* algorithms even managed to classify all of the security related instances correctly. However, we can also see that they have FP rate of 1 which also means that they miss-classified all of the unrelated instances as well. Looking at the other algorithms such as the *REPTree* and the *Logistic* algorithms we can see that they too suffer from the same problem albeit a bit less than the top 3. Ideally in this situation we would want an extremely high TP rate (as close to 1 as possible) and very low FP rate (as close to 0 as possible) to have best prediction results.

If we look at the machine algorithm with the greatest difference between the two rates we find the *LogitBoost* algorithm. It has the highest difference of 0.319 between TP rate and FP rate. This means that the *LogitBoost* algorithm is actually a bit better at predicting instances than the remaining ones. This also means that there is a difference in how it identifies and classifies its instances.

Classifier	**TP Rate**	**FP Rate**
NaiveBayes	0.633	0.479
RandomTree	0.433	0.188
kStar	0.367	0.206
LogitBoost	0.367	0.048
J48	0.333	0.109
PART	0.250	0.079
RandomForest	0.267	0.079
OneR	0.150	0.073
REPTree	0.100	0.042
Logistic	0.033	0.048
MultiClassClassifier	0.033	0.048
BayesNet	0.000	0.000
DecisionStump	0.000	0.000
ZeroR	0.000	0.000

Table 4.11: Algorithms TP and FP rates based on unrelated to security instance classification. Sorted by best TP Rate.

Table 4.11 shows TP and FP rates in terms of True Negatives. Here we can see that our prediction rates are much lower than those of true positives in table 4.10. This means that all of the machine learning algorithms were much worse at identifying instances unrelated to security. The best resulting rate of 0.633 (corresponding to 63.3% prediction accuracy) is held by *NaiveBayes* algorithm. However once again we see that its high TP rate is almost matched by its FP rate meaning that the results aren't terribly accurate or very different from one another.

If we look at an algorithm with the greatest difference between TP and FP rates we find that the *LogitBoost* once again beats all of the remaining machine learning algorithms. Unsurprisingly the difference remains 0.319 between TP and FP rates which is the same as with true positives. It is also worth to note that *LogitBoost* algorithm was our top performer achieving the best prediction accuracy of 79.56%. This proves that *LogitBoost* algorithm is one of the best choices for classification.

4.2 Other attempts

In this section we mention several other approaches taken to extract some information which either didn't result in any successful application or in some cases lead to more complex solutions that would require too much focus and time to complete and thus were considered out of scope for this project. We start by going through our attempt at defining security class roles. Then we move on to security keywords and libraries, taint analysis, pattern detection and some other smaller experiments.

4.2.1 Security roles as a feature

Using Wirfs-Brock class roles may be useful, however none of the roles used are defined with any security concepts in mind and so they may be too generic for our purpose. We attempt to create a similar class role definitions in order to suit more security and privacy related code and *JAVA* classes. Much like original roles, security roles would be somewhat generic but have clear type and purpose and then ideally any single class would have its own independent role based on our definitions. Since Wirfs-Brocks roles are not security or privacy related, our custom roles should ideally have greater impact on classifier prediction accuracy. By using Wirfs-Brocks roles as a template we arrived at the following roles:

- *Security Information Holder* - In this case, much like with Wirfs-Brocks classes, an information holder is a simple data structure or an object. They tend to be static and persist and have mostly *getter* and *setter* methods with very limited additional functionality. They might use other information holders or process data, however the key difference in this case is that the data is clearly security or privacy related.
- *Security Controller* - Controllers are complex classes that handle various security related configurations and uses security providers and information holders. Controllers perform functions directly related to security and/or privacy.
- *Security Provider* - Security providers are similar to *Controllers* in that they also perform security or privacy related functionality. Key difference being that security providers are much smaller and more specific with their functionality. They contain methods that process data, perform certain functions and return the result to their caller. Security providers are essentially meant to be used by other classes rather than work on their own.
- *Security User* - A *user* class in this case is similar to a security provider, however, its main purpose is not security functionality. A *user* class simply tends to use some security or privacy related functionality for a fraction of time with it being a sort of secondary use. An example of this would be displaying users profile name on a pop-up message, in which case the class might need to access users profile data. *User* classes tend to have *void* methods or functions that return data unrelated to security or privacy.
- *Abstract Class* - Abstract classes can be explained as a sort of placeholder for fully implemented classes. Abstract classes always contain one or more methods and variables that are not fully implemented or instantiated. This means that some, if not most of the abstract classes will have no defined functionality making them extremely difficult to classify without seeing full implementation. Therefore we assign abstract classes to a category of their own.
- *Interface* - We refrained from assigning roles to interfaces as well by giving

them their own role.

- *Enum* - *enum* elements are used to declare some kind of enumeration and contains a list of constants used for it. Since this is a fairly specific and only use of *enum* elements we assign them to their own unique role.
- *Unrelated* - An unrelated role represents a class that has no security related functionality.

One of the reasons to why we chose to have separate roles for abstract classes, interfaces and *enum* classes was to check whether splitting them would have any effect with machine learning algorithms. We then use these class roles and assign them to the same set of classes as with regular Wirfs-Brocks classification. This time however the class roles were assigned by a single reviewer without any cross-referencing or dialog from other parties. This makes our classification more subjective and prone to bias, however we wanted to maintain at least some consistency (in terms of interpretations of roles) even if it could lead to biased results. The rest of the classification followed the same procedure as with Wirfs-Brocks roles and was based on the security annotations from the ground truth source code. This entire process took about a month from the initial code review and the final draft of roles.

4.2.2 Security keywords and libraries as a feature

The few additional and promising features such as checking security related imports turned out to be more confusing than helpful. Even though security related libraries would seem to be inherently indicating which instances (*.class* files) contain security related code, this was not the case at all. Regardless of machine learning algorithms used, none of them identified security related libraries feature to be of much importance. Effectively this meant that our security libraries had very little to no measurable impact on the prediction accuracy.This allowed us to remove it from the feature set to minimize information noise (too much data is not always good) and achieve slightly better prediction accuracy that way.

The exact same situation repeated with our attempt to use security keywords as a feature. Any time we would use security keywords as a feature, our prediction accuracy would get worse than without using it. It also gave absolutely no indications of any positive impact. This was the case even after modifying our keyword list several times to refine it and make it more case specific (tailored to fit k9-mail code base) or more generic (adding even unrelated security terms).

Using FSF together with security keywords feature, the best achieved accuracy was only 76%. This is all while FSC alone gave an accuracy of 76.4%. Even though the difference is minimal (0.4%) this was the absolute best achieved accuracy using this feature. We also tried to compile a feature set where security keywords would have some measurable impact but this was the best result we achieved. The best achieved prediction accuracy using a set with security keywords as a feature is 3.1% worse than our absolute best achieved prediction accuracy of 79.5% using FSB.

It still remains unclear what was the precise reason why those features didn't work. While security related libraries are more of a mystery, security keywords feature may have lacked context or used incomplete set of keywords despite of our attempts to make it as comprehensive as possible. This may be very likely since there have been successful attempts at using keyword based search for security code identification in other studies.

4.2.3 Taint analysis

In addition to extracting code metrics and other information we have tried using taint analysis in order to detect data flows related to security. Since taint analysis is security related we assumed that finding data sinks and assigning them to classes would be a good additional feature for our classifiers. However the tools that we used either couldn't detect any taint sources. While we have our doubts whether this was actually the case we were left with no data to work with and thus couldn't make this a useful feature.

Since taint analysis is not directly related to software architecture and our goal was to identify features that could be extracted from it. For this reason we have decided to abandon this topic and feature and move on to other features instead.

4.2.4 Software pattern detection

Software pattern detection seemed another promising feature to use. However in this case a major issue proved to be a lack of pattern necessary definitions. There are hundreds of software patterns available with various degrees of relevance to privacy and security. While this is great in most cases in order to use them we needed to create pattern definitions and behavioral models. Creating just a few of them would take considerable time and dedication which in our case we didn't have enough of. In addition to time constraints we were uncertain of which patterns would work best for K9-mail application and what patterns would be easiest to define. For this reason we have abandoned this topic in order to finish already existing experiments. However this does leave software pattern detection as a great potential feature for follow-up studies since no apparent downsides were identified during our investigation.

4.2.5 Misc attempts

Another attempt was made by trying to reverse engineer K9-mail architecture using some of the tools that analyze the source code and create a representation of it in an UML format (Enterprise Architect [38], StarUML [39], ArgoUML [40]) . This information was then used with a language processing tools (such as GATE [41]) in order to find some relationships between classes that could be used as a feature in our data set. However the reverse engineered architecture proved to be lacking both in accuracy as well as in details. Nonetheless we have attempted to use this data by trying to identify any reoccurring patterns in the architecture. We did this by analyzing frequency of reoccurring architecture elements as well as keywords (specific to the tools used) however without any luck. It may be possible to use this method to extract features in the future however for this we need a much better reverse engineering tool that provides more details about software architecture and relationships between architecture elements.

5
Conclusion

In this chapter we discuss our results and provide our conclusion as well as other thoughts around it. We start this off by offering a discussion of results and our opinion on the overall success of the study as well answering research questions. We then talk about potential threats to validity and what could have gone wrong. We then finish this chapter by mentioning potential future work and give our final conclusion.

5.1 Discussion

In this section of the thesis we mainly provide our thoughts about the thesis and what could have been done differently. We also talk about factors that could have influenced results as well as mentioning what could be done in the future. Here we also talk about threats to validity and discuss potential future work.

5.1.1 Discussion

From our experiments we have obtained some interesting results. In some cases we were able to improve our machine learning algorithm predictions by using features such as class roles, number of libraries used, and various code metrics. In other cases seemingly useful features such as security related libraries or security keywords proved to have no or very little effect on the results. Even scenarios tailored to favor security libraries or keywords the results proved to be disappointing.

Just by using source code metrics we were able to achieve slightly better accuracy than initial baseline results. Providing an additional class role feature to our machine learning algorithms gave us the best results which shows us that there are some links between software architecture and our code. These results also imply that there is some potential to use software architecture features to detect security relevant code.

By manipulating algorithm settings and selecting specific features to use we achieved somewhat decent prediction of approximately 79.56% or almost 4/5. Whether this is adequate or not is not up to us to decide but whatever the case may be our final results show improvement over baseline implementation. However some of the improvement in accuracy can be attributed to our data set and its distribution in terms of related and unrelated source code. Since we have a data set consisting of roughly 3/4 security relevant and a 1/4 non relevant classes we can expect our results to reflect exactly that. Ignoring the skew in distribution we get an improvement of

6.26% for prediction accuracy compared to baseline results. In other words we get a net improvement of 6.26%. While in some cases this may seem insignificant we still see this as an improvement and a positive result.

The biggest caveat with the features we chose, especially with class roles, is that it still takes some considerable amount of time to extract them. While there are tools for extraction of code metrics, class roles require manual intervention which includes certain degree of subjectivity. As long as there are no automatic tools for class role assignment this remains somewhat impractical especially on a large scale projects. On top of that these class roles require some consistency which is difficult to achieve when more than one person is assigning them. Allowing a single person to assign them would reduce the overhead required to classify them and make it a bit faster. However a single person assigning such roles could introduce too much bias and even then could have some inconsistencies. Though one of the positive side effects (in our case) of working with class roles is that it allows whoever is assigning the roles to familiarize themselves with the system and gain much better understanding of it.

In terms of granularity chosen the decision seems to be fairly sound. By using whole classes we are able to relate them to software architecture as well as use several tools to extract additional information about the source code. This saved us considerable amount of time as well as allowed us to experiment with more parameters such as Java security libraries used or potential of software pattern detection. Using any smaller granularity such as methods or lines of code may have allowed us to locate the precise parts of code that relate to security. However such granularity would make it much more difficult to extract additional information about software architecture or see any relations to it. Since one of our objectives is to use software architecture this would be not only impractical but also prevent us from achieving our objective.

It also should be possible to use more different features for security related code detection. However not all of these features would have to be extracted only from software architecture. Data flow diagrams, control flow graphs, sequence diagrams and other artifacts could be used at the current granularity level to extract additional features. One of the biggest issues with extracting features is knowing which ones would have most impact and therefore the only way to find out is by trial and error. Though biggest issue once you start using too many features is that it becomes increasingly more difficult for machine learning algorithms to create accurate prediction models. This becomes a bigger issue when there are no clear patterns in the data and introducing additional features only increases the so called data noise.

To sum all of our work up we can answer our initial research questions:

RQ1: Can software architecture provide any means to identify security relevant code?

A1: A simple answer would be yes, yes it can. However it all depends on your approach and resources available.

RQ2: How useful are software metrics when used for security sensitive code detection?

A2: Just by using default source code metrics such as the ones extracted by SourceMonitor we wont get far. While basic metrics make it possible to somewhat predict where security related code might be, the results won't be very accurate. Furthermore you need to cherry pick a set of metrics in order to be able to use them for prediction since using too many won't give any results at all.

RQ3: Can we improve our results by using additional features extracted from software architecture?

A3: Yes, we can. Filtering out some of the basic metrics and adding additional features related to software architecture such as class roles will improve prediction accuracy. While the improvement may not be extreme it is still noticeable and has a positive impact.

RQ4: How much of an effect do these additional features have?

A4: Based on our data we managed to get a 4.89% improvement in accuracy when we added class roles to the data-set. The overall improvement for best prediction accuracy by using a combination of features was 3.1% when compared to best prediction accuracy without additional features.

5.1.2 Threats to validity

One of the biggest threats to this study is subjectivity during class role assignments. Since Wirfs-Brock roles have flexible definitions in some cases there will be disagreements whether a class has one role or the other. To complicate things even more these class roles assume that each instance (class) has a clear and well defined role which is easy to identify. This is not always the case with some of the instances having multiple roles, or at least flexible functionality that can be attributed to more than one role. This makes class role identification not only complicated but also difficult to get right especially when you have to choose a single role.

Deriving from Wirfs-Brocks roles we have created our own class roles with intent to have a security related class system. This is new system may be heavily biased for a couple of reasons. First of all we derive these new roles after having worked extensively with Wirfs-Brocks roles which could have lead to us making decisions based on what we learned about the system (in this case k9 mail). This would make our derived roles k9 mail specific rather than security related which would skew the results one way or another and not represent the actual truth. Additionally these new security roles are completely new concept and were created by a single individual rather than through a group discussion. This introduces further potential bias and potential for these new roles to be k9 mail application specific rather than objectively security related.

Another similar issue is apparent with our security code. While in most cases it is fairly obvious that a piece of code is performing security or privacy related tasks in others it can be more ambiguous and discrete. We are using our own definitions and interpretations of what we consider to be security and privacy related code which works well in this case but may not in others. We can't be completely sure whether the success or failure of our experiments is the result of our definitions or whether there is a real relation between features used which introduces a certain degree of uncertainty. This also applies to the ground-truth that we have used. We rely and trust completely that the ground-truth is actually representative of security code. Since our ground-truth is obtained from a different, unpublished study, we depend on its accuracy and suffer from any potential mistakes that may have been made on their part. Of course we believe that the ground-truth used is not plagued by any major threats of validity but we still can't exclude such a possibility.

In our feature set we attempt to use a list of security keywords as a feature. This list is compiled by browsing various sources online, source code as well as by using own experience in the security field. This introduces more potential bias and could result in our list that was used being inconsistent, incomplete and heavily tailored to suit exactly our needs. Since software security is a very broad field it has many different terms associated with different types of software and areas. By creating such list on our own we introduce our own bias and potentially skew results. Since when we create this list we already have experience with k9 mail application and know its source code well, it's possible that our keyword list is tailored to suit exactly k9 mail case. It is also possible that we have left out many potential keywords as well. Overall our approach has an advantage that tailoring such a list to suit our needs we could potentially emphasize our results which may be the same even without our biases. Alternatively it could have an opposite effect so the only way to test this is to replicate the study using different keyword list.

We also use native Java libraries as a feature set which we later refine to security related libraries only. Since there are many different native as well as custom Java libraries its extremely difficult to know which ones are which. From our research we have identified several of the Java libraries as being security relevant by checking Java documentation. This has been done by looking for specific functionality such as handling user data, encryption, communication and other similar tasks which we deemed would have security relevance. We did not verify the contents of these libraries due to time constraints and assumed that they simply contain some security functionality. This introduces a fairly major flaw since we can no longer have complete confidence in our results. Since there are no openly available lists of security related libraries online we were limited to this choice only. However due to a relatively small number of Java libraries used we can have some confidence that our list, while perhaps incomplete, still provides the desired results.

One of the additional assumptions we make when checking for security related libraries is that we assume that all of the included Java libraries are actually being used. Since checking for specific library calls would be too time consuming we can't guarantee that an identified security related library in use is being used in a security context. This means that some functionality that relies on such libraries could be using other functionality provided by the library giving us a false positive. We have no real and efficient way of checking whether security functionality is present or being used from these libraries this leaves yet another fairly major flaw when it comes to security related libraries feature.

5.1.3 Future Work

One of the main areas that seems to have some potential for security relevant code detection is software pattern identification. While it would be difficult to choose which patterns to identify, it does seem to be fairly promising, especially if implementation is done right. There already exists tools that are capable of automatic software architecture pattern detection which only need to be adapted for security related patterns. The only question that remains in this case is if such pattern detection would be as accurate as regular patterns. Additionally in case security pattern detection turns out to be effective it would be also necessary to test if these detected patterns could be used in a similar context as this project. Since most this could be done automatically it might be able to replace class roles completely and eliminate most of the required manual work (required with class roles).

Additionally more information about class relationships could be extracted from software architecture. In this project we have used only a small amount of available information about software architecture. This is due to several reasons such as lack of time and need to have more focus. Therefore any future projects around the same topic could attempt to use more information about architecture than in this case. Additional information use could reveal some undiscovered relationships between architecture and code and improve machine learning algorithms prediction accuracy. All of this of course is still speculative at this point since we haven't had enough time to explore this more in-depth. Nonetheless this is one of the areas that could certainly benefit from additional research.

Finally security keywords could be investigated further. The results from using security keywords in our project scope were disappointing. While security in nature such keywords weren't as useful as expected and it might be useful to discover the reason why this is the case. We already know from other studies that using keywords can aid in code classification. Therefore what is left is to find out if our issue in this context stems from the keyword list that we used or if it has to do with the nature of machine learning algorithms. Whatever the case might be spending additional time and using a more thorough process to create security keyword list might answer some of the questions.

5.2 Conclusion

Using regular software metrics extracted from k9 mail source code it was possible to detect security-sensitive code somewhat better than just by purely guessing. However the difference in such case was marginal. Adding additional software architecture features into the mix such as Wirfs-Brocks class roles were able to improve our results by approximately 6.26%. This increase in accuracy gives us a prediction accuracy of almost 80% or 4/5 correct classifications. This means that 1 in 5 instances will be miss classified which severely limits the trust of our classifiers. Since the intention of introducing such classification is to minimize the amount of source code needed to inspect having 80% accuracy doesn't cut it. If we could achieve near perfect 100% prediction accuracy for instances containing or lacking security code we could at least use such classification partially. However none of the machine learning algorithms achieved such prediction accuracy leaving our results a bit lacking in terms of usefulness. Without improving our prediction accuracy we can't use any of the gained information in a meaningful way making it somewhat useless.

In addition to a lack in accuracy we need to dedicate considerable amount of time and effort into assigning class roles to source code. This introduces a certain degree of uncertainty as well as time overhead. All of this combined doesn't really warrant a 6.26% improvement in prediction accuracy over random guessing. This becomes especially evident when we consider that the total prediction accuracy is only 80% making this kind of code location methods not very viable.

Overall we were able to extract some useful information from software architecture (ex. libraries used, class roles). We were then able to use this information in predicting whether a certain instance (part of source code) contains security related code. This prediction however was not very accurate and somewhat better than guessing. Extracting more information from software architecture and using it as additional features (such as software architecture patterns) could still improve prediction accuracy. This leaves us with a positive indication that software architecture can be used in security related code detection and that it has more potential to achieve better prediction accuracy.

Bibliography

[1] K-9 Dog Walkers. K-9 mail. *https://play.google.com/store/apps/details?id=com.fsck.k9*, 2017.

[2] Truong Ho Quang. Private communication. *https://www.chalmers.se/en/staff/Pages/truongh.aspx*.

[3] fsHornDroid. fshorndroid a static analysis tool! *http://134.96.235.13:8080/*.

[4] Paul C Clements. Software architecture in practice. *Diss. Software Engineering Institute*, 2002.

[5] Raghvinder S Sangwan, Pamela Vercellone-Smith, and Phillip A Laplante. Structural epochs in the complexity of software over time. *IEEE software*, 25(4), 2008.

[6] Rudolf K Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st international conference on Software engineering*, pages 226–235. ACM, 1999.

[7] Benjamin Rogers, Yechen Qiao, James Gung, Tanmay Mathur, and Janet E Burge. Using text mining techniques to extract rationale from existing documentation. In *Design Computing and Cognition'14*, pages 457–474. Springer, 2015.

[8] Mohamed Soliman, Matthias Galster, Amr R Salama, and Matthias Riebisch. Architectural knowledge for technology decisions in developer communities: An exploratory study with stackoverflow. In *Software Architecture (WICSA), 2016 13th Working IEEE/IFIP Conference on*, pages 128–133. IEEE, 2016.

[9] Norman Wilde, Juan A Gomez, Thomas Gust, and Douglas Strasburg. Locating user functionality in old code. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 200–205. IEEE, 1992.

[10] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Transactions on software engineering*, 29(3):210–224, 2003.

[11] Mary Shaw and David Garlan. *Software architecture: perspectives on an emerging discipline*, volume 1. Prentice Hall Englewood Cliffs, 1996.

[12] Metacademy. F measure. *https://metacademy.org/graphs/concepts/$f_m$easure*.

[13] David Lettier. You need to know about the matthews correlation coefficient. *https://lettier.github.io/posts/2016-08-05-matthews-correlation-coefficient.html*.

[14] Hamid Bagheri, Joshua Garcia, Alireza Sadeghi, Sam Malek, and Nenad Medvidovic. Software architectural principles in contemporary mobile software: from conception to practice. *Journal of Systems and Software*, 119:31–44, 2016.

[15] Wideskills.com. Java classes and objects. *http://www.wideskills.com/java-tutorial/java-classes-and-objects.*

[16] Frances E. Allen and John Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137, 1976.

[17] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. *Data flow analysis: theory and practice.* CRC Press, 2009.

[18] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and privacy (SP), 2010 IEEE symposium on*, pages 317–331. IEEE, 2010.

[19] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[20] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[21] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Software Engineering (WCSE), 2012 Third World Congress on*, pages 101–104. IEEE, 2012.

[22] Rob Schapire. Cos 511: Theoretical machine learning. *FTP: http://www. cs. princeton. edu/courses/archive/spr08/cos511/scribe notes/0204. pdf*, 2008.

[23] Dr. Saed Sayad. Zeror. *http://chem-eng.utoronto.ca/ datamining/dmc/zeror.htm*, 2010.

[24] Gail A Carpenter, Stephen Grossberg, and John H Reynolds. Artmap: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network. *Neural networks*, 4(5):565–588, 1991.

[25] Dr. Saed Sayad. Oner. *http://chem-eng.utoronto.ca/ datamining/dmc/oner.htm*, 2010.

[26] Webb Geoffrey I Sammut Claude, editor. *Decision Stump*, pages 262–263. Springer US, Boston, MA, 2010.

[27] Wikipedia. Bayesian network. $https://en.wikipedia.org/wiki/Bayesian_network$, 2017.

[28] Dr. Saed Sayad. Naive bayesian. *http://chem-eng.utoronto.ca/ datamining/dmc/naive*$_b$*ayesian.htm*, 2010.

[29] Mark Hall. Kstar. $https://wiki.pentaho.com/display/DATAMINING/KStar$, 2008.

[30] Shyam BV. Bias variance - tradeoff. $https://rpubs.com/shyambv/bias_variance_tradeoff$.

[31] Rebecca J Wirfs-Brock. Characterizing classes. *IEEE software*, 23(2):9–11, 2006.

[32] Riccardo Scandariato. Private communication. *http://dblp.uni-trier.de/pers/hd/s/Scandariato:Riccardo.*

[33] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe. Automatic design pattern detection. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 94–103. IEEE, 2003.

[34] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. Design pattern detection using similarity scoring. *IEEE transactions on software engineering*, 32(11), 2006.

[35] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: a taxonomy and survey. *Journal of software: Evolution and Process*, 25(1):53–95, 2013.

[36] GitHub. k9mail/k-9. *https://github.com/k9mail/k-9/*.

[37] Paderborn University and TU Darmstadt. Flowdroid – taint analysis. *https://blogs.uni-paderborn.de/sse/tools/flowdroid/*.

[38] Sparx Systems Pty Ltd. Enterprise architect. *http://sparxsystems.com/products/ea/*.

[39] Co MKLab. Staruml 2. *http://staruml.io/*.

[40] CollabNet. Argouml. *http://argouml.tigris.org/*.

[41] The University of Sheffield. Gate. *https://gate.ac.uk/*.

A

Appendix A

A.0.1 Complete data set

The complete set of data is available for download here: https://drive.google.com/file/d/1gvd-sBN9Wbd1T6wcF0U1WPIz4aQsOJYL/view