DEGREE PROJECT IN TECHNOLOGY,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2016

Anti-analysis techniques to weaken author classification accuracy in compiled executables

JOHAN WIKSTRÖM AND MACAULLY MUIR

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF COMPUTER SCIENCE AND COMMUNICATION

KTH Computer Science
and Communication

Anti-analysis techniques to weaken author classification accuracy in compiled executables

DEGREE PROJECT IN COMPUTER SCIENCE, FIRST LEVEL
STOCKHOLM, SWEDEN 2016

JOHAN WIKSTRÖM
MACAULLY MUIR

Supervisor: Michael Schliephake
Examiner: Örjan Ekeberg

CSC, KTH 2016-05-11

Abstract

Programming languages such as C/C++ allow for great flexibility in how code can be written. This leads to programmers developing their own "code style" that can be used to identify them among a group of other programmers, in a setting such as a programming competition. Recent research has shown that some of the identifying stylistic features present in source code survive the compilation process, and that authorship classification can be performed on the compiled executables alone. This was originally performed by Rosenblum et al. in their 2011 paper on the subject.

This thesis takes the approach of Rosenblum et al. and investigates how the author classification process is affected by changes in the compilation process of the training dataset, specifically different levels of optimisation (-O1 to -O3) and static linkage. We find that full optimisation yields a 10% drop in accuracy in datasets with 413 and 20 authors respectively. Static linkage results in a significant drop in accuracy in datasets with 20 and 10 authors, respectively. In both cases, the classifiers still perform significantly better than random chance and as such these methods cannot guarantee anonymity to the programmer. It is not clear how these results translate to other datasets, although there is reason to believe they would be reproducible using other classifiers found in the literature.

Referat

Tekniker för att reducera möjligheten att identifiera författare till kompilerade program

Programspråk såsom C och C++ ger programmeraren stor valfrihet i hur koden kan se ut. Detta leder till att man utvecklar sin egen “kodstil” som kan användas för att känna igen en programmerare i en större grupp, såsom vid en programmeringstävling. Ny forskning har visat att några av de identifierande stilfaktorerna som finns i källkoden överlever kompileringsprocessen och att författarklassificering kan göras med de kompilerade programmen enbart. Detta visades för första gången av Rosenblum et al. i en artikel publicerad år 2011.

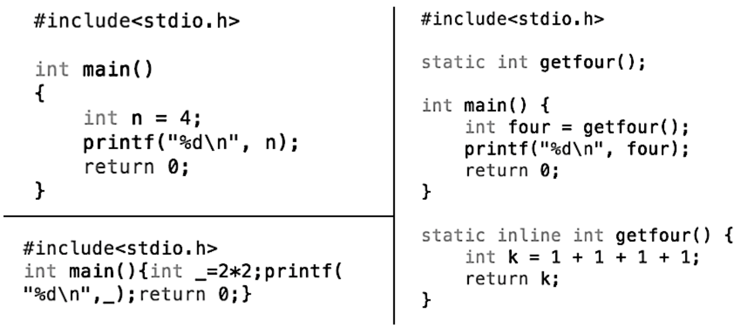
Denna uppsats använder sig av samma metoder som Rosenblum et al. och undersöker hur förändringar i kompileringsprocessen påverkar klassificeringens noggrannhet. Effekterna av olika optimeringsnivåer (-O1 till -O3) samt statisk länkning undersöks. Vi ser att full optimering ger en 10% förminskning av noggrannhet i datamängder med 413 respektive 20 författare. Statisk länkning leder till en betydligt minskad noggrannhet i datamängder med 20 respektive 10 författare. I båda fallen presterar klassificeringen ändå bättre än slumpen, vilket gör att dessa metoder därmed inte kan garantera programmerarens anonymitet. Det är svårt att säga huruvida dessa resultat kan reproduceras i andra datamängder. Dock finns det anledning att tro att liknande resultat skulle erhållas om andra klassificerare i litteraturen skulle användas.

Contents

Chapter 1

Introduction

Languages such as C and C++ are flexible enough to allow the programmer to write solutions to the same algorithmic problem in a multitude of different ways, not unlike how the English language allows for many different ways of explaining the same concept. As with the written word, this leads to programmers developing their own “code style” (Krsul and Spafford 1997). Stylistic properties such as the usage of whitespace, bracket placement, naming conventions etc. all contribute to the programmer’s code style, as well as implementation differences such as the usage of for vs. while loops, iteration vs. recursion and so on. Of course, many problems have multiple algorithmic solutions and the programmer’s preferences between them also contribute to their unique style. Figure 1.1 shows how the trivial task of printing the number 4 to stdout can be accomplished in vastly different ways in the C programming language.



```
#include<stdio.h>

int main()
{
    int n = 4;
    printf("%d\n", n);
    return 0;
}
```

```
#include<stdio.h>
int main(){int _=2*2;printf(
"%d\n",_);return 0;}
```

```
#include<stdio.h>

static int getfour();

int main() {
    int four = getfour();
    printf("%d\n", four);
    return 0;
}

static inline int getfour() {
    int k = 1 + 1 + 1 + 1;
    return k;
}
```

Figure 1.1. Three different C programs that print the number 4 to stdout. When compiled with gcc’s -O3 flag each program generates the same executable, verified by comparing MD5 hashes.

It has been shown in multiple studies that a programmer’s code style is unique within a limited population, such as a subset of students at RMIT University (Burrows, Uitdenbogerd, and Turpin 2009) or contestants in the Google Code Jam programming competition (Caliskan-Islam et al. 2015a). Moreover, unique code style can be used to reliably identify programmers from their source code alone. Caliskan-Islam et al. developed methods that could identify programmers from 1600 candidates with 93% accuracy in the above study. As seen in figure 1.2, identifying authors from their source code is essentially a machine learning problem.

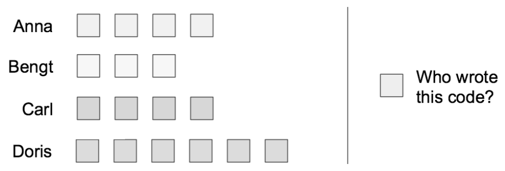


Figure 1.2. The problem of identifying programmers from their source code is essentially a machine learning problem: Given a *training set* of source files with known authors, is it possible to "learn" the style of each author and accurately *classify* a program outside of the training set? This is called the *author classification problem.*

Is the author classification problem solvable if, instead of their source code, we only had access to the programmers' compiled executables? On face value it is not at all clear that the necessary stylistic information survives the compilation process; indeed, the three programs in figure 1.1 would be impossible to differentiate because they all produce the exact same executable when compiled. Nevertheless, in 2011 Rosenblum et al. developed methods that could identify programmers from their compiled executables with 77% accuracy from a set of 20 Google Code Jam contestants (Rosenblum, Zhu, and Miller 2011), showing that the author classification problem for compiled executables is in fact solvable in at least some cases. These methods were improved upon by Caliskan-Islam et al. in 2015, achieving 96% accuracy in a similar dataset (Caliskan-Islam et al. 2015b).

Instead of trying to further improve upon the results of Caliskan-Islam et al., this thesis takes the opposite approach and examines what programmers can do to *prevent* de-anonymisation with the above methods. Specifically, we examine how the methods of Rosenblum et al. (Rosenblum, Zhu, and Miller 2011) perform when the training data is compiled with various levels of compiler optimisation (-O1 to -O3) turned on, as well as studying the effects of statically linking the programs in the training data. This is something that has not been examined in previous research.

1.1 Problem statement

The aim of this thesis is to investigate techniques that reduce the accuracy of models in the author classification problem for compiled executables. We choose an experimental approach and base our analysis around the methods proposed by Rosenblum et al. (Rosenblum, Zhu, and Miller 2011) for mainly practical reasons described in section 1.3. The specific techniques we choose to evaluate are the enabling of different levels of compiler optimisation (-O1 to -O3 using gcc) and statically linking when compiling the models' training data. This is captured by the following problem statement:

> *How do compiler optimisations and static linkage affect the accuracy of the author classification models presented in (Rosenblum, Zhu, and Miller 2011)?*

1.2 Motivation

Methods that solve the author classification problem for compiled executables could potentially be useful in a number of fields such as malware analysis or plagiarism detection. However, little research exists on the robustness of such methods against active adversaries. Common anti-analysis techniques such as the use of packers or virtualisation can add a prohibitive pre-processing step to the classification process (Caliskan-Islam et al. 2015b), but the existence of a truly non-reversible procedure for obfuscating the identifying features of a program is an open research question. Investigating methods that can reliably weaken the accuracy of classifiers in the literature is an important first step to understanding how a more general solution could be constructed, if it is in fact feasible at all.

From a programmer's point of view there are also a number of legitimate reasons why one would want to remain anonymous. It could be that one's association with the software could put them in danger (for example, a programmer that writes anti-surveillance software in an oppressive state). It could also be the case that the programmer simply does not want to be publicly associated with the software. In this regard, it is highly desirable to give programmers the choice of anonymity. Again, weakening the accuracy of classifiers in the literature is an important first step towards this goal.

1.3 Scope

This thesis bases its analysis entirely around the methods presented by Rosenblum et al. in (Rosenblum, Zhu, and Miller 2011). The reasons for doing so are mainly practical; the programs used for feature extraction along with the full dataset of features used in their study are available for download at the authors' homepage. Having access to the same toolchain as the original authors makes for a more reliable comparative study.

Analysis is limited to solutions from the 2010 round of the Google Code Jam programming contest written in C++, one of the datasets used by Rosenblum et al. Section 3.1 provides a more comprehensive description of the datasets used.

Chapter 2

Background

This section begins with an overview of the field. Moreover, the technical concepts required to understand the rest of this thesis is introduced, and finally the state-of-the-art in the field of authorship attribution is presented. Focus will be on deriving authorship from binary programs; however, some related works will be mentioned.

2.1 Overview of the field

Authorship classification belongs to a branch of computer science called computer forensics. The goal of computer forensics is to analyse digital data to obtain information, often in relation to computer crime (Reith, Carr, and Gunsch 2002). In recent years machine learning techniques have been used with great success to identify counterfeited integrated circuits (Huang, Carulli, and Makris 2013), distinguishing different kinds of brain tumours (Zacharaki et al. 2009) and the automated detection of software bugs (Aleem, Capretz, and Ahmed 2015), to name a few examples.

The first attempt at using statistical methods to de-anonymise programmers from their code style was performed by Oman et al. (Oman and Cook 1989) who classified programmers using a hand-picked, language-specific feature set. In 2006, Frantzeskou et al. proposed a method using the analysis of byte level n-grams (continuous sequences of n bytes) present in the source code which greatly improved upon previous methods, along with being language-agnostic (Frantzeskou et al. 2006). In 2015, Caliskan-Islam et al. presented a method based on features of the abstract syntax trees (ASTs) of C/C++ programs that significantly improved the state-of-the-art again (Caliskan-Islam et al. 2015a).

The 2011 study by Rosenblum et al. (Rosenblum, Zhu, and Miller 2011) upon which this thesis is based was first to show that author classification in compiled executables is feasible. Features were extracted from the disassembled code and the resulting control flow graph; this is explained in more detail in section 3.2. In 2014 Alrabaee et al. improved upon the accuracy of those techniques and proposed features that were more closely related to programming style (Alrabaee et al. 2014). In

2015, Caliskan-Islam et al. further improved the state of the art using disassemblers and decompilers, and applying their research in de-anonymising programmers from source code to the ASTs of the decompiled binaries (Caliskan-Islam et al. 2015b). In the same report, Caliskan-Islam et al. also perform a short side study on the effects of compiler optimisations have on their methods, concluding that they result in a significant drop in accuracy but not enough to be effective anonymisation tools. No other studies regarding techniques for reducing the accuracy of classifiers for authorship attribution in compiled executables were found.

2.2 Technical Background

This section includes the technical details necessary for understanding the rest of this thesis. Initially it starts with a brief overview of the compilation process and the static analysis. Moreover, this section describes techniques used to extract features from binary executables and gives a short explanation of support vector machines (SVMs) and their uses.

A rudimentary understanding of the compilation process of C/C++ programs is presumed, in particular the basic relationships between source code, assembly code and binary machine code. Chapter 1 of (Aho, Sethi, and Ullman 1986) provides a sufficient overview for the purposes of this thesis.

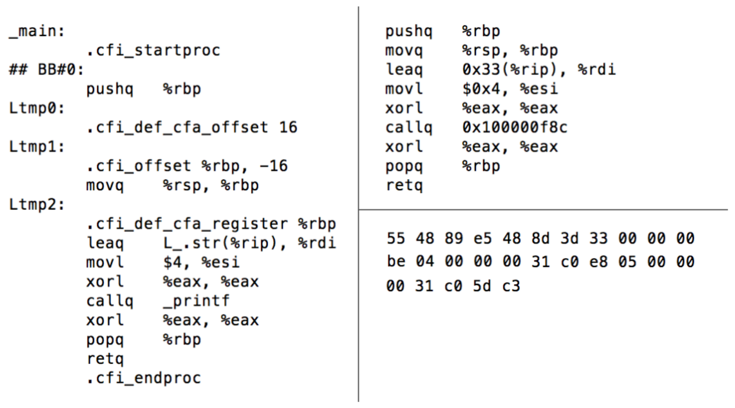
2.2.1 Disassembly

Disassembly is the task of generating a functionally equivalent assembly code representation of a compiled executable. The general idea is to give a "human readable" representation of a program that is easier to work with than the plain binary data. Figure 2.1 shows the compiler generated assembly code for the programs in figure 1.1, its equivalent machine code after compilation and the resulting assembly code after disassembly. Note that lexical data such as comments and whitespace do not survive the compilation process.

In general disassembly is a non-trivial task, especially with executables targeting for complex instruction sets such as x86 (Wartell et al. 2011). Moreover, a number of techniques can be employed to purposely thwart commonly used disassembly algorithms, making the process much harder (Linn and Debray 2003). For the purpose of this thesis, however, this is not an issue; the Dyninst library used for disassembling (see section 3.2 for details) is capable of disassembling all compiled executables in the Google Code Jam dataset.

2.2.2 Control flow analysis

Control flow analysis of compiled executables is performed on assembly code produced by a disassembler. The goal is to partition the code into so-called "basic blocks": sequences of instructions with exactly one entry point and exit point with



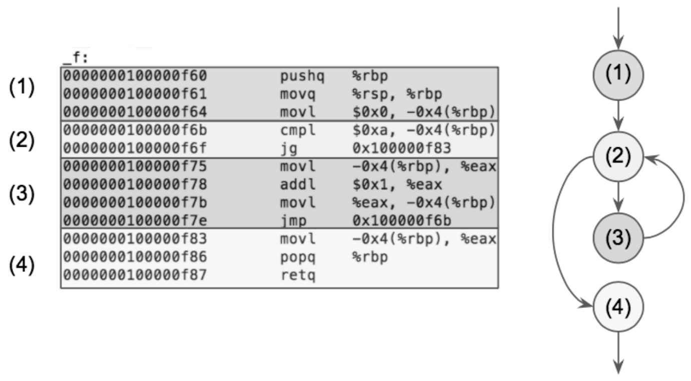
```
_main:
         .cfi_startproc
## BB#0:
         pushq   %rbp
Ltmp0:
         .cfi_def_cfa_offset 16
Ltmp1:
         .cfi_offset %rbp, -16
         movq    %rsp, %rbp
Ltmp2:
         .cfi_def_cfa_register %rbp
         leaq    L_.str(%rip), %rdi
         movl    $4, %esi
         xorl    %eax, %eax
         callq   _printf
         xorl    %eax, %eax
         popq    %rbp
         retq
         .cfi_endproc
```

```
pushq   %rbp
movq    %rsp, %rbp
leaq    0x33(%rip), %rdi
movl    $0x4, %esi
xorl    %eax, %eax
callq   0x100000f8c
xorl    %eax, %eax
popq    %rbp
retq
```

```
55 48 89 e5 48 8d 3d 33 00 00 00
be 04 00 00 00 31 c0 e8 05 00 00
00 31 c0 5d c3
```

Figure 2.1. Compiler generated assembly code (left), the equivalent machine code after compilation (bottom right) and the assembly code generated by disassembling the compiled executable (top right), for the three programs in figure 1.1. Note that stylistic features such as whitespace, labels and comments are lost in the compilation process.

respect to program flow (Allen 1970). This structure is captured in the control flow graph (CFG), a directed graph with the basic blocks as nodes and control flow paths as edges. Figure 2.2 illustrates the basic blocks of a simple assembly code routine.



```
     _f:
(1)  0000000100000f60    pushq   %rbp
     0000000100000f61    movq    %rsp, %rbp
     0000000100000f64    movl    $0x0, -0x4(%rbp)
(2)  0000000100000f6b    cmpl    $0xa, -0x4(%rbp)
     0000000100000f6f    jg      0x100000f83
(3)  0000000100000f75    movl    -0x4(%rbp), %eax
     0000000100000f78    addl    $0x1, %eax
     0000000100000f7b    movl    %eax, -0x4(%rbp)
     0000000100000f7e    jmp     0x100000f6b
(4)  0000000100000f83    movl    -0x4(%rbp), %eax
     0000000100000f86    popq    %rbp
     0000000100000f87    retq
```

Figure 2.2. A simple assembly routine and corresponding CFG. The basic blocks of the routine are labeled (1)-(4).

The CFG provides a structured view of the binary at a higher level than individual instructions. This can be used to identify high-level language constructs such as loops and functions (Cifuentes and Gough 1993) and as such is used by decompilers such as the Hex-Rays decompiler (Hex-Rays 2013), used by Caliskan-Islam et al. in their paper improving on the work of Rosenblum et al. (Caliskan-Islam et al. 2015b). Rosenblum et al. also use the CFG for feature extraction, but do not go as far as to decompile the binaries (Rosenblum, Zhu, and Miller 2011).

2.2.3 Derivatives of the CFG

In their 2011 paper upon which this thesis is based, Rosenblum et al. defined an additional two abstractions based on the CFG of a program: the *coloured control flow graph* (CCFG) and the *coloured call graph* (CCG).

The CCFG is a graph identical to the CFG with an additional *colouring function*. Each instruction in the target instruction set is classified as one of fourteen instruction classes such as “arithmetic instructions” or “branching instructions”, and a node’s colour is simply a 14 bit bitmap representing which instruction types are present in its corresponding basic block. The colouring process is described in more detail in (Rosenblum, Miller, and Zhu 2011), but further knowledge is not needed for the rest of this thesis.

The CCG is created from the CFG by only including the subset of nodes that contain a call instruction, with edges between two nodes representing a path between the corresponding nodes in the original CFG. A colouring function is applied in the same way as for the CCFG, with the colours depending on whether not the node calls a library function, and if so, which library function is called.

2.2.4 Support vector machines

Support vector machines (SVMs) belong to a group of supervised learning methods and are based on statistical learning theory. This thesis will use linear SVMs. SVMs have been around since the 1960s and perform well on for example text and image recognition. The learning algorithm is trained using a set of data and can later be used to identify patterns or to classify new data. In the simplest case with only two different classes A and B (binary classification) and a set of data points, the goal is to find a hyperplane that separates the data points from class A from those of class B and at the same time maximises the margin to the closest data point from each class. This concept can later be extended to work on multiple classes. Moreover, there is a balance in the amount of data the algorithm needs to perform well. More data is not always better; using too much information may cause “overfitting” resulting in poorer predictive performance (Cristianini and Shawe-Taylor 2000).

2.2.5 Mutual information

When using SVMs for classification it is important to be weary of “overfitting” the training data, which can lead to poor predictive performance (Cristianini and Shawe-Taylor 2000). A technique called *feature selection* can be used to prevent this from happening, the general idea being to remove as many features as possible from the training dataset until the model starts to lose predictive power. In order to decide which features to remove first they must be ranked according to some criteria. Mutual information (MI) is one such criteria (Guyon and Elisseeff 2003).

In the case of the author classification problem, ranking is performed as follows. Let X be the multiset of all features generated for a given data set. For all $x_i \in suppX$, let $P(X = x_i)$ denote the observed frequency of x_i in X. Likewise, let Y be the set of all author labels and $P(Y = y) = \frac{1}{|Y|}$ be the observed frequency of the label y in Y and let $P(X = x_i, Y = y)$ denote the joint frequency of x_i and y in the data set. Each unique feature $x_i \in suppX$ is ranked using the mutual information criteria $R(x_i)$ (Guyon and Elisseeff 2003) in the following way:

$$R(x_i) = \sum_{x_i \in X, y \in Y} P(X = x_i, Y = y) \log \frac{P(X = x_i, Y = y)}{P(X = x_i)P(Y = y)}$$

Intuitively the highly ranked features will be unique to a small set of labels in the training data and as such more useful in the classification process, whereas the lowest ranked features can be removed without negatively affecting the predictive power of the model. Removing superfluous features also has the added benefit of reducing the time needed to train the model, which can be important when working with high-dimensional data (Guyon and Elisseeff 2003).

2.2.6 Cross-validation

Cross-validation is a common method used when testing predictions. It is a model used to analyse how well a predictive model will work in practice. When using 10-fold cross-validation the data is randomly divided into 10 equally sized sets, 9 of the sets is used for training and the remaining set is used for the testing. This is repeated for each one of the sets, finally resulting in an average measure of accuracy (Hsu, Chang, Lin, et al. 2003).

2.3 State-of-the-art analysis

As previously discussed, there are as far as we are aware of exactly three papers presenting novel techniques to de-anonymise programmers from compiled binary programs: (Rosenblum, Zhu, and Miller 2011), (Alrabaee et al. 2014) and (Caliskan-Islam et al. 2015b). In this section each approach will be discussed. The focus will be on the techniques used by Rosenblum et al.

2.3.1 Rosenblum et al.

Rosenblum et al. evaluate their data on both a private dataset obtained from an operating systems course (CS537) at the University of Wisconsin and a subset of correct solutions from the 2009 and 2010 rounds of the Google Code Jam (GCJ), specifically the solutions that:

- Were written in C/C++

- Could be compiled using GCC 4.5
- Were submitted by an author that had submitted correct solutions to at least 8 problems in total

In total, the GCJ dataset consisted of 2581 binaries (1,747 binaries from 2010) from 284 authors. Using 10-fold cross-validation on a set of 191 authors from 2010, 51% accuracy was achieved with 1900 features. For a randomly selected subset of 10 authors, 81% accuracy was achieved and for a randomly selected subset of 20 authors 77% accuracy.

2.3.2 Alrabaee et al.

Another study by Alrabaee et al. (An Onion approach to Binary code Authorship Attribution) used an approach they called OBA2 to identify author style in binary code (Alrabaee et al. 2014). The study refers to the work of Rosenblum et al. and argues that some of the features used in Rosenblum et al. do not represent author style features. The result of the study shows a higher accuracy than (Rosenblum, Zhu, and Miller 2011).The data used in the study was the same as Rosenblum et al.

2.3.3 Caliskan-Islam et al.

Another report published in December 2015 by Caliskan-Islam et al. improves on the work of Rosenblum. This study has been able to improve author identification accuracy even further also using data from Google Code Jam and c/c++ binary code. With a set of 20 authors they were able to find the correct author with 96% accuracy. Caliskan-Islam (2015) also tested the effect of different optimisations and achieved with 100 authors and no optimisation 78.3% accuracy and with level 3 optimisation 60.1% accuracy. Furthermore, the study found that it was easier to identify more experienced programmers.

Chapter 3

Method

As the goal of this thesis is to investigate the robustness of methods presented by Rosenblum et al. in (Rosenblum, Zhu, and Miller 2011), the methods presented in this section necessarily overlap those presented by Rosenblum et al. in their original paper. To the extent possible, the same datasets, sample sizes, algorithms and toolchain are used. A control study is performed on a dataset compiled without optimisations or static linkage with the same number of authors as in the original paper to ensure that the two implementations match.

This section presents the datasets used in the analysis, as well as the methods used for feature extraction, feature selection, classification and finally evaluation.

3.1 Data and datasets

Analysis is based on data collected from the 2010 round of Google Code Jam (GCJ), one of the datasets used by Rosenblum et al. Specifically, the analysis is restricted to authors that have submitted at least 8 submissions in C++. Furthermore, submissions are required to consist of exactly one file with the extension .cpp which can be compiled and statically linked with gcc version 5.3 on a Fedora 23 x64 installation. A total of 413 authors in the 2010 GCJ dataset match these criteria with 3969 files in total, averaging 9.6 submissions per author. Most authors have 8-9 submissions, as seen in figure 3.2

The submissions are comparatively small on average; 97% of submissions contain less than 200 lines of code, with most submissions containing between 50 and 100 lines of code. Figure 3.1 shows the distribution of file size in the full 413 author dataset.

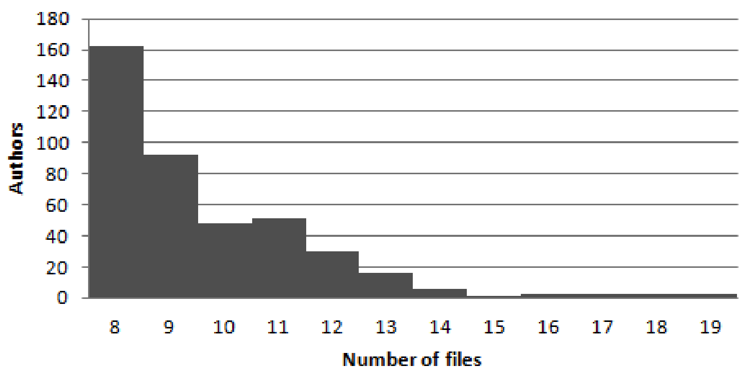


Figure 3.1. Distribution of files over authors in the full 413 author dataset.

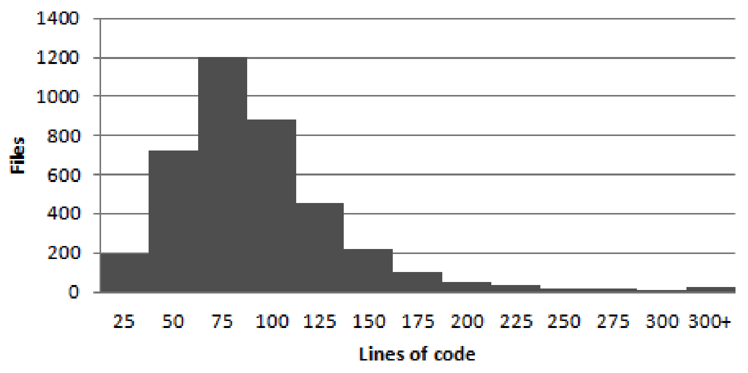


Figure 3.2. Lines of code per file in the full 413 author dataset.

3.2 Feature extraction

Feature extraction is the process of extracting measurable properties from the sample dataset for further analysis. This section aims to provide a short account of the *feature templates* used by Rosenblum et al., on which this thesis is based upon. For an in-depth account of how each feature template contributes to the overall analysis, we refer to their original paper (Rosenblum, Zhu, and Miller 2011).

Because it is not known which properties of the compiled binaries are meaningful for determining authorship, the approach of Rosenblum et al. is to extract a large number of generic features in the hope that authorship information will be captured indirectly. All features are derived from the program's disassembled code, coloured control flow graph (CCFG) and coloured call graph (CCG) (section 2.2.3 explains these concepts in detail) and are based on 6 feature templates:

Idioms

All sequences of 1-3 instructions are extracted and counted from the disassembled code. This is performed intraprocedurally; only sequences of instructions within the same procedure are considered. Wildcard instructions are allowed in order to capture more general instruction patterns.

N-grams
: N-grams (continuous sequences of bytes) of length 3 or 4 are extracted from the program. Again, this is performed intraprocedurally so n-grams outside of procedures will be ignored.

Graphlets
: All connected components of size 3 of the CCFG are extracted.

Supergraphlets
: Each node in the CCFG is collapsed with a random neighbour and all connected components of size 3 of the resulting graph are extracted. The algorithm used to collapse nodes is described in detail in (Rosenblum, Miller, and Zhu 2011)

Call graphlets
: All connected components of size 3 of the CCG are extracted.

External interaction
: The name of each library routine used by the program is extracted.

Features are extracted for each program in the dataset and the set of unique features in the entire dataset is generated; each feature can then be indexed and recognised across programs. Finally, features are grouped by author and saved in the LibSVM format (Chang and Lin 2011) for analysis.

In a supplement to their 2011 paper, Rosenblum et al. provided the programs they created for feature extraction as open source software. The same programs are used in the analysis presented in this thesis, albeit with minor changes due to outdated dependencies [1]. Both versions use the Dyninst library ("Dyninst: An application program interface (api) for runtime code generation") for disassembly and reconstruction of the CFG.

3.3 Feature selection

The feature extraction process yields a high dimensional dataset with hundreds of thousands to hundreds of millions of features. Given the large number of features, feature selection is required in order to speed up the training of the SVM and avoid overfitting. Features are ranked with the mutual information criteria presented in section 2.2.5, after which models are trained with varying sizes of subsets of the

[1] The original feature extraction source code can be found at `http://pages.cs.wisc.edu/~nater/esorics-supp/`. Our modified versions can be found at `https://github.com/macaullyjames/esorics`, along with installation instructions.

highest ranking features. The predictive power of each model is then evaluated (see section 3.4) and the model that performs best is retained.

Subset sizes of between 800 and 100000 features are evaluated in all datasets. Models trained with all features are also evaluated where doing so is computationally feasible, however they consistently perform worse than models trained on smaller subsets.

3.4 Classification and evaluation

As in the original paper from Rosenblum et al., classification is performed using a linear support vector machine (SVM) (section 2.2.4 explains SVMs in more detail). The svm-scale program from the LIBSVM toolkit is used (Chang and Lin 2011) to scale the feature values to the interval [0, 1], whereafter the LIBLINEAR SVM implementation (Fan et al. 2008) is used for training and evaluating via 10-fold cross-validation.

Chapter 4

Results

As stated in the introduction, the aim of this thesis is to investigate how compiler optimisations and static linkage affect the accuracy of the author classification models presented by Rosenblum et al. in (Rosenblum, Zhu, and Miller 2011). This section begins with the results from the control study before presenting the results in each of these areas. The control study shows that the methods used in this thesis are comparable to those used by Rosenblum et al. Compiler optimisations are shown to decrease the predictive power of the model in all datasets and statically linked programs are significantly harder to classify accurately.

4.1 Method validation

A 191 author subset of the dataset described in section 3.1 is compiled without additional compiler optimisations (-O0). Our model is then evaluated using 10-fold cross-validation on each dataset using a varying number of features, as described in chapter 3. Additionally, we evaluate our model using 20 randomly selected subsets of 20 authors in the same way. Both results are compared to comparable results in the 2011 study by Rosenblum et al.

Our models achieve 41.8% accuracy in the 191 author subset and 71.2% accuracy on average in the 20 random subsets of 20 authors. In comparison, Rosenblum et al. achieve 51% accuracy and 76.8% accuracy in comparable datasets. The chances of randomly choosing the correct author in both datasets are 0.5% and 5% respectively. Figure 4.1 summarises our results compared to that of Rosenblum et al.

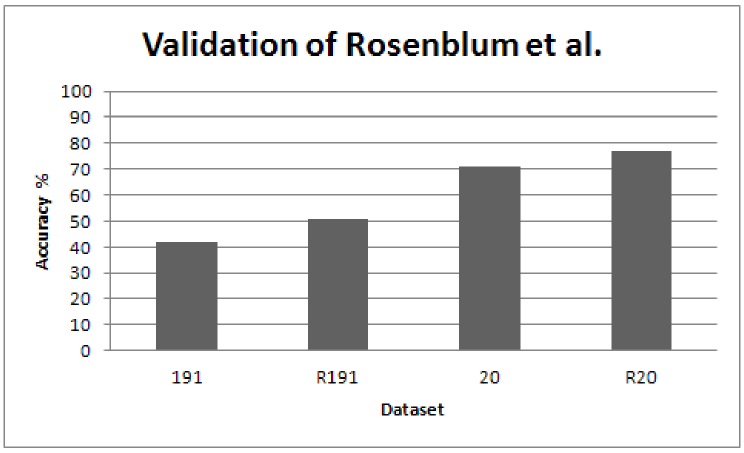


Figure 4.1. Comparison of model accuracies with those of Rosenblum et al.

4.2 Optimisation levels -O1 to -O3

The full 413 author dataset is compiled with optimisation levels -O1 to -O3 and compared to a baseline dataset compiled without additional compiler optimisations enabled (-O0). Compiler optimisations lead to an increase in both unique features and the total number of features extracted from the programs, the -O3 dataset yielding 5 times the number of unique features as the -O0 dataset and a 63% increase in the total number of features. Models trained on the -O1 and -O2 datasets do not perform significantly worse than the baseline (28.6% and 27.6% vs. 32.5%) whereas the -O3 dataset yields an accuracy of just 21.6%. This is summarised in table 4.1.

Table 4.1. How accuracy varies depending on optimisation level in the full 413 author dataset

Optimisation level	Model accuracy
−O0	32.5%
−O1	28.6%
−O2	27.6%
−O3	21.6%

Interestingly, the number of features selected by cross-validation is roughly the same for all optimisation levels. This despite that both the number of unique features and total number of features vary greatly, as seen in table 4.2.

The same analysis is performed on 20 random subsets of 20 authors from the same 413 author dataset. Again, models trained on the -O1 and -O2 subsets do not perform significantly worse than the baseline on average (65.4% and 64.4% vs. 68.8%, respectively) while the -O3 subsets yield an average accuracy of 58.7%. This is summarised in table 4.3.

Table 4.2. The number of unique features, total number of features and number of features selected by cross-validation for each optimisation level in the full 413 author dataset

Optimisation level	Unique features	Total features	Selected features
−O0	765307	13204201	7000
−O1	2141558	15532246	9000
−O2	2593340	16681152	10000
−O3	3891494	21570182	10000

Table 4.3. Average accuracy and standard deviance per optimisation level for models over 20 random subsets of 20 authors from the full 413 author dataset

Optimisation level	Average accuracy	Standard deviation
−O0	68.8%	5.1%
−O1	65.4%	6.9%
−O2	64.4%	7.0%
−O3	58.7%	5.7%

In summary, we find that the use of compiler optimisation flags does consistently decrease the accuracy of our models, even though they still perform significantly better than random chance. With the highest optimisation level, -O3, a 10% drop in accuracy is observed in both the full 413 author dataset and random subsets of 20 authors.

4.3 Statically linked programs

A subset of 20 authors is compiled and statically linked with optimisation level -O0. This is compared to a baseline dataset of the same 20 authors compiled without additional optimisations (-O0) but dynamically linked. The statically linked programs yield 3859605 unique features and 110725073 features in total. In comparison, the baseline dataset yields 122420 unique features and 618918 features in total, meaning the statically linked programs yield more than 30 times the number of unique features and almost 180 times the number of features in total. Models trained on the statically linked programs also performed considerably worse than the baseline: 28.6% vs. 75.0%. The number of features selected by cross-validation is roughly the same in both cases. These results are summarised in table 4.4.

Table 4.4. Comparison of model accuracy, number of unique features, total number of features and number of features selected by cross-validation for dynamically respectively statically linked programs from the same 20 author dataset

Linkage	Model accuracy	Unique features	Total features	Selected features
Dynamic	75.0%	122420	618918	7000
Static	28.6%	3859605	110725073	9000

The same analysis is performed with 20 randomly selected subsets of 10 authors from the 20 author dataset used above. Again, models trained on the statically linked programs perform significantly worse than models trained on their dynamically linked counterparts, as seen in table 4.5.

Table 4.5. Average accuracy and standard deviance for models over 20 random subsets of 10 authors with static and dynamic linkage, respectively

Linkage	Average accuracy	Standard deviation
Dynamic	81.2%	7.3%
Static	45.5%	6.6%

In summary, statically linking the programs in our dataset leads to a large increase of unique features and total number of features, and significantly reduces the accuracy of our models.

Chapter 5

Discussion

The results presented in chapter 4 show that both compiler optimisations and static linkage negatively effect the performance of the authorship classifiers used. In this section we discuss the validity of these results and whether they can be expected to hold in other datasets. The implications of our research are discussed with respect to programmer anonymity. Practical aspects of using our methods in real-life scenarios and future research.

5.1 Method analysis

The method presented in the 2011 paper by Rosenblum et al. (Rosenblum, Zhu, and Miller 2011) was followed to the best of our abilities. Nevertheless, the comparison study in section 4.1 showed that our methods experienced a near 10% drop in accuracy on a 191 author dataset and an average 5% drop in accuracy in the 20 author subsets. There are, despite our best efforts, several discrepancies between the two methods:

Dataset inconsistencies

Our full dataset contains 413 authors, compared to the 191 authors used by Rosenblum et al. This despite using the same 2010 Google Code Jam data and the same selection criteria (8+ submissions per author, written in C++ etc.).

Errors in the feature extraction process

As described in section 3.2, we make use of the open source programs Rosenblum et al. provide for feature extraction. However, minor modifications to these programs were necessary in order for them to build in our test environment. Moreover, it is not clear exactly how Rosenblum et al. used these programs; for example, their paper does not state the number of times the CCFG is collapsed in order to obtain supergraphlet features.

Different testing environments

The study by Rosenblum et al. was performed in 2011. Since then, the Linux kernel has jumped from version 3.0 to version 4.3, gcc has jumped from version 4.6.0 to version 5.3 and the Dyninst library used for feature extraction (see section 3.2) has undergone 2 major version revisions.

It is also possible that the differences in accuracy are due to error on our part. Nevertheless, we feel that the results are close enough that our general approach can be considered valid.

5.2 Wider implications

Our experiments are performed using the methods of Rosenblum et al. Can the results be expected to hold for other methods of classification? Caliskan-Islam et al. performed a short side study of the implications of compiler optimisation in their 2015 paper regarding author classification for compiled executables (Caliskan-Islam et al. 2015b) and found that compiling with the -O3 flag enabled yielded a drop of accuracy with approximately 18% in a 100 author dataset. We suspect that static linkage would also have similar results with their methods.

As stated in section 3.1, most files in our dataset contain between 50 and 100 lines of code. It can be expected that most useful programs "in the wild" are much larger than this, and it is not clear that our results would be reproducible in a dataset with larger programs. Static linkage has the effect of mixing application code with library code in the resulting executable, and larger programs would have a smaller percentage of library code present. This could potentially negate the benefits of static linkage seen in our experiments.

5.3 Consequences for programmer anonymity

While our research shows that it is relatively easy to dramatically decrease the effectiveness of the classifiers in our datasets, the predictive power of models trained on those datasets are still significantly higher than random chance. To be clear: these methods do not *guarantee* anonymity, but do help reduce the chance of being identified.

When trying to preserve programmer anonymity it is important that any techniques employed be *non-reversible*. If, for example, the statically linked executables could be pre-processed in a way that removed the statically linked library code then the benefits of static linkage would be completely negated; the analyst would simply add an extra pre-processing step to the process. While we have not seen evidence in the literature that compiler optimisations or static linkage are reversible, we have not seen evidence of the contrary either.

5.4 Future research

As stated in section 5.2, it is not clear that similar results would be achieved in datasets containing larger programs. Also, most datasets encountered in the literature are chosen so that authors solve the *same* set of problems; Google Code Jam submissions, hand-ins for a university-level class etc. There are many potential applications of author classification, such as malware analysis, where datasets can be expected to consist of programs with mutually different functionality. Further research is needed to investigate if the current state-of-the-art can be used in these circumstances.

The fact that many programs are developed in teams also needs to be addressed. Is it possible to classify teams of authors in the same way? What if two teams enforce the same code style guidelines? What if team members change frequently? Again, further research is needed. A greater understanding of how the stylistic characteristics of source code are transformed during the compilation process could lead to both better feature extraction methods and anti-analysis techniques.

Chapter 6

Conclusion

This thesis has shown that the predictive power of authorship classification methods for compiled executables presented in the literature, specifically those presented in the 2011 paper of Rosenblum et al. (Rosenblum, Zhu, and Miller 2011), are significantly weakened when working with executables compiled with full optimisation turned on or when the executables are statically linked. Nevertheless, the accuracy of the models is still much higher than random chance in all cases and the use of these techniques does not guarantee anonymity.

Further research is required to determine the real-world implications of our findings, particularly with respect to alternative datasets. A greater understanding of how high-level stylistic features of source code are transformed during the compilation process could potentially lead to both better classifiers and better anti-analysis techniques.

Bibliography

Aho, Alfred V, Ravi Sethi, and Jeffrey D Ullman (1986). *Compilers, Principles, Techniques.* Addison wesley.

Aleem, Saiqa, Luiz Fernando Capretz, and Faheem Ahmed (2015). "Comparative performance analysis of machine learning techniques for software bug detection". In: *Computer Science and Information Technology (CS & IT-CSCP 2015)*, pp. 71–79.

Allen, Frances E (1970). "Control flow analysis". In: *ACM Sigplan Notices.* Vol. 5. 7. ACM, pp. 1–19.

Alrabaee, Saed et al. (2014). "OBA2: an onion approach to binary code authorship attribution". In: *Digital Investigation* 11, S94–S103.

Burrows, Steven, Alexandra L Uitdenbogerd, and Andrew Turpin (2009). "Application of information retrieval techniques for source code authorship attribution". In: *Database Systems for Advanced Applications.* Springer, pp. 699–713.

Caliskan-Islam, Aylin et al. (2015a). "De-anonymizing programmers via code stylometry". In: *24th USENIX Security Symposium (USENIX Security 15)*, pp. 255–270.

Caliskan-Islam, Aylin et al. (2015b). "When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries". In: *arXiv preprint arXiv:1512.08546.*

Chang, Chih-Chung and Chih-Jen Lin (2011). "LIBSVM: a library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 2.3, p. 27.

Cifuentes, Cristina and K John Gough (1993). "A methodology for decompilation". In: *Proceedings of the XIX Conferencia Latinoamericana de Inform atica.* Citeseer, pp. 257–266.

Cristianini, Nello and John Shawe-Taylor (2000). *An introduction to support vector machines and other kernel-based learning methods.* Cambridge university press.

Fan, Rong-En et al. (2008). "LIBLINEAR: A library for large linear classification". In: *The Journal of Machine Learning Research* 9, pp. 1871–1874.

Frantzeskou, Georgia et al. (2006). "Source code author identification based on n-gram author profiles". In: *Artificial Intelligence Applications and Innovations.* Springer, pp. 508–515.

Guyon, Isabelle and André Elisseeff (2003). "An introduction to variable and feature selection". In: *The Journal of Machine Learning Research* 3, pp. 1157–1182.

Hex-Rays, SA (2013). *Hex-Rays decompiler.*

Hsu, Chih-Wei, Chih-Chung Chang, Chih-Jen Lin, et al. (2003). "A practical guide to support vector classification". In:

Huang, Ke, John M Carulli, and Yiorgos Makris (2013). "Counterfeit electronics: A rising threat in the semiconductor manufacturing industry". In: *Test Conference (ITC), 2013 IEEE International.* IEEE, pp. 1–4.

Krsul, Ivan and Eugene H Spafford (1997). "Authorship analysis: Identifying the author of a program". In: *Computers & Security* 16.3, pp. 233–257.

Linn, Cullen and Saumya Debray (2003). "Obfuscation of executable code to improve resistance to static disassembly". In: *Proceedings of the 10th ACM conference on Computer and communications security.* ACM, pp. 290–299.

Oman, Paul W and Curtis R Cook (1989). "Programming style authorship analysis". In: *Proceedings of the 17th conference on ACM Annual Computer Science Conference.* ACM, pp. 320–326.

Reith, Mark, Clint Carr, and Gregg Gunsch (2002). "An examination of digital forensic models". In: *International Journal of Digital Evidence* 1.3, pp. 1–12.

Rosenblum, Nathan, Barton P Miller, and Xiaojin Zhu (2011). "Recovering the toolchain provenance of binary code". In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis.* ACM, pp. 100–110.

Rosenblum, Nathan, Xiaojin Zhu, and Barton P Miller (2011). "Who wrote this code? identifying the authors of program binaries". In: *Computer Security–ESORICS 2011.* Springer, pp. 172–189.

Source, Open. "Dyninst: An application program interface (api) for runtime code generation". In: *Online, http://www. dyninst. org.*

Wartell, Richard et al. (2011). "Differentiating code from data in x86 binaries". In: *Machine Learning and Knowledge Discovery in Databases.* Springer, pp. 522–536.

Zacharaki, Evangelia I et al. (2009). "Classification of brain tumor type and grade using MRI texture and shape in a machine learning scheme". In: *Magnetic Resonance in Medicine* 62.6, pp. 1609–1618.