POLITECNICO
MILANO 1863

Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione

MASTER OF SCIENCE IN COMPUTER SCIENCE & ENGINEERING

Supervised learning approaches to assist the static analysis of executable files

Master Thesis of:
Pietro De Nicolao
matricola 849761

Advisor:
Prof. Stefano Zanero

Co-advisor:
Davide Quarta

Academic Year 2016-2017

A mia sorella, Laura

Contents

List of Figures

List of Tables

Abstract

In this thesis, we address three problems concerning the static analysis of executable files: the identification of the Instruction Set Architecture (ISA), the discovery of the boundaries of the code section, and the problem of separating code from data.

To disassemble an executable, we need to know its ISA and the boundaries of the code section; however, this information is not always available. We present a classification method to identify the ISA of header-less executable files and a sequential learning method to locate the boundaries of its code section. We test the architecture classifier on a dataset of binaries of 20 different ISAs: the experiments show that our method has a better performance than the state of the art. We test the code section identification method on ELF, PE and Mach-O executables compiled for different sets of architectures, by multiple compilers: the experiments show that our model is always able to identify, with a high accuracy, where the code is located inside the binary files.

Data embedded inside the code section represents a major problem for disassemblers, which may interpret it as valid machine instructions. In literature, most of the proposed approaches to deal with inline data are based on recursive traversal disassembly, which presents severe limitations when dealing with indirect control instructions. We follow a different approach: we train a sequential learning classifier on the bytes representing code and data in pre-tagged executables, and use it to predict the boundaries of the instructions in unseen binaries. The results of our experiments show that our method is capable to distinguish code from data in unseen binary samples with an accuracy of over 99.9%.

Sommario

In questa tesi, affronteremo tre problemi riguardanti l'analisi statica degli eseguibili: l'identificazione dell'Instruction Set Architecture (ISA), l'identificazione della sezione di codice, e la separazione codice-dati.

Per poter disassemblare un file eseguibile, è necessario conoscerne l'ISA e la posizione della sezione contenente il codice. Come si procede se queste informazioni non sono disponibili?

Presentiamo un semplice metodo di apprendimento supervisionato per identificare l'architettura dei file eseguibili e un metodo di apprendimento sequenziale per il riconoscimento dei confini della sezione codice nei binari privi di intestazione. Le prestazioni del nostro classificatore di architetture — valutato su più dataset derivanti da programmi reali e articoli scientifici — superano quelle dell'approccio allo stato dell'arte. Valutiamo il metodo di riconoscimento della sezione di codice su eseguibili ELF, PE e Mach-O compilati per più architetture, da diversi compilatori: il nostro modello è sempre in grado di identificare i byte appartenenti alle sezioni di codice con un'accuratezza elevata (99.8%).

Nell'ambito dell'analisi statica degli eseguibili, i dati inseriti all'interno della sezione di codice da alcuni compilatori rappresentano un problema per i disassembler, che rischiano di interpretarli come istruzioni macchina. In letteratura, la gran parte degli approcci proposti per riconoscere i dati *inline* si basa sul disassembly ricorsivo, che presenta limiti significativi in presenza di salti indiretti. Seguiamo qui un diverso approccio: costruiamo un classificatore sequenziale che apprenda un modello probabilistico dei byte che compongono le istruzioni di codice macchina e i dati, in modo da poterli distinguere. I risultati degli esperimenti mostrano che il metodo presentato è in grado di distinguere il codice dai dati nei binari con un'accuratezza di oltre il 99.9%.

Introduzione[1]

Ambito

L'analisi statica dei file eseguibili — ossia l'analisi del codice eseguibile eseguita senza tracciarne l'esecuzione — è un approccio efficace per estrarre informazioni dai programmi eseguibili quando il codice sorgente non è disponibile. L'analisi statica dei binari fa parte del dominio più ampio dell'ingegneria inversa dei programmi eseguibili. Due degli scopi più importanti dell'analisi statica sono: ottenere un corretto *disassembly* delle istruzioni macchina, e costruire un corretto *Control Flow Diagram* (CFG) del programma.

L'analisi statica dei binari è possibile solo se alcune informazioni sul file eseguibile da analizzare sono note in precedenza: per poter disassemblare un eseguibile, è necessario conoscere l'ISA (*Instruction Set Architecture*) del binario e i confini della sezione del file che contiene il codice eseguibile. La conoscenza dell'ISA è necessaria perché l'ISA definisce la codifica binaria delle istruzioni macchina: la stessa sequenza di byte, infatti, può avere due significati totalmente diversi per due CPU diverse. Conoscere i confini delle regioni contenenti il codice è necessario per assicurarsi che il disassembler lavori sul codice, e non sugli altri dati contenuti all'interno del file eseguibile.

Le informazioni sull'ISA e sulle sezioni del file eseguibile si trovano di solito nell'intestazione dell'eseguibile stesso. In alcuni casi, tuttavia, tale intestazione è assente: questo è il caso di programmi eseguiti direttamente sull'hardware di un dispositivo (firmware). Inoltre, i file eseguibili possono essere deliberatamente offuscati per ostacolare i tentativi di ingegneria inversa: questo è il caso dei *malware*. In tutti questi casi dobbiamo estrarre le informazioni necessarie analizzando il file eseguibile, senza disporre dei relativi metadati.

Anche quando l'ISA ed i confini delle regioni di codice sono noti, il problema di ottenere un corretto *disassembly* dell'eseguibile resta comunque non banale poiché il *disassembler* rischia di interpretare erroneamente come delle istruzioni i dati inseriti all'interno della sezione codice da parte di alcuni compilatori. Infatti, le intestazioni dei formati eseguibili standard (ad esem-

[1] Questo ampio estratto in lingua italiana ottempera all'obbligo previsto dall'art. 4.2 del "Regolamento d'Ateneo degli Esami di Laurea e di Laurea Magistrale con integrazioni della Scuola di Ingegneria Industriale e dell'Informazione", approvato dal Senato Accademico del 23.1.2017 e dalla Giunta della Scuola del 9.2.2017.

pio ELF e PE) non includono le informazioni sulla posizione dei dati inclusi all'interno della sezione di codice, a meno che non siano disponibili gli appositi simboli di debug. Per la maggior parte dei programmi (soprattutto commerciali e *off-the-shelf*), i simboli di debug non sono disponibili, perché faciliterebbero l'ingegneria inversa del prodotto.

Definizione del problema e obiettivi

In questo lavoro proponiamo tre nuovi approcci per risolvere tre problemi aperti nel campo dell'analisi statica dei programmi eseguibili. I tre problemi di cui ci occuperemo sono:

1. **Identificazione dell'*Instruction Set Architecture***: dedurre l'architettura della CPU di un file eseguibile privo di intestazione;

2. **Identificazione della sezione codice**: determinare dove sia collocata la sezione eseguibile, contenente il codice macchina, all'interno di un file eseguibile privo di intestazione;

3. **Problema della separazione tra codice e dati**: all'interno della sezione eseguibile di un file binario, distinguere accuratamente (a livello di byte) tra le istruzioni macchina e i dati *inline*.

Daremo una spiegazione approfondita di questi tre problemi e delle sfide che li accompagnano nel Capitolo 2.

Stato dell'arte

Nella sezione 2.3, presenteremo una rassegna approfondita dello stato dell'arte nella letteratura scientifica relativamente ai problemi sopra descritti.

Il problema della classificazione dei file in base al loro tipo è stato ampiamente affrontato in letteratura (sottosezione 2.3.1), ma solo un lavoro è specificamente orientato all'identificazione dell'architettura dei file eseguibili: l'approccio descritto in (Clemens 2015) consiste in un modello di apprendimento supervisionato che usa le frequenze dei byte come feature per classificare i file binari in base alla loro architettura. Riteniamo che i risultati di questo lavoro possano essere migliorati introducendo feature aggiuntive ed ottimizzando i parametri degli algoritmi di apprendimento.

Il problema dell'identificazione del tipo dei segmenti di file — affrontato ad esempio in (Sportiello e Zanero 2012) — è simile al problema dell'identificazione della sezione contenente il codice negli eseguibili; tuttavia, nel nostro caso la frammentazione non è definita "a priori": necessitiamo di un metodo che segmenti il file e, contemporaneamente, classifichi i segmenti.

Le *Conditional Random Field* (Lafferty, McCallum e Pereira 2001) sono dei modelli di apprendimento supervisionato e strutturato in grado di predire un insieme di variabili le cui dipendenze condizionali sono codificate in un grafo. Le applicazioni delle CRF spaziano nei campi del riconoscimento del linguaggio naturale, della *computer vision* e della bioinformatica. In (Rosenblum et al. 2008), un modello basato su CRF è usato per identificare le funzioni nel codice binario.

Un problema classico nel campo dell'analisi di binari è quello di distinguere il codice dai dati all'interno dei file eseguibili (la sezione 2.3 contiene una rassegna degli approcci esistenti). I *disassembler* esistenti producono risultati erronei qualora i dati si trovino inframmezzati con il codice macchina (analizziamo questo problema nella sottosezione 2.3.3). Gli approcci "classici" si basano sull'*disassembly* ricorsivo (ossia, contrassegnare come codice tutti e soli gli indirizzi raggiungibili dall'esecuzione del programma), e tentano di migliorarne i risultati impiegando diverse euristiche, e l'esecuzione simbolica; nella sottosezione 2.3.5 descriviamo tre approcci di questo tipo. La principale sfida legata a questo tipo di approcci è l'identificazione precisa delle destinazioni dei salti indiretti. (Shoshitaishvili et al. 2016, p. 4)

Un approccio alternativo al problema della separazione tra codice e dati è quello di sviluppare un modello di apprendimento supervisionato sul codice macchina (descriviamo questa famiglia di approcci nella sottosezione 2.3.4). (Wartell et al. 2011) introduce un modello di apprendimento sequenziale (*Prediction by Partial Matching*) per segmentare il codice x86 in istruzioni valide e dati. Il modello ottiene un'accuratezza elevata, classificando correttamente come codice o dati più del 99.8% dei byte. La valutazione del modello è eseguita confrontando manualmente l'output del modello con il *disassembly* generato da IDA Pro, poiché non è disponibile un disassembly "perfetto" dei binari a cui far riferimento; questa limitazione non permette di testare il modello su un numero elevato di binari. Questo approccio supporta una sola architettura (x86), e si basa su alcune euristiche specifiche per quell'architettura: adattare il metodo per supportarne di nuove richiederebbe uno sforzo non trascurabile.

Approccio

Nel Capitolo 3, descriviamo l'approccio che abbiamo sviluppato per risolvere i problemi sopraelencati.

Per risolvere il problema della classificazione degli eseguibili per architettura, miglioriamo l'approccio descritto in (Clemens 2015) costruendo un classificatore basato sulla regressione logistica con feature aggiuntive (le frequenze dei pattern relativi ai prologhi e agli epiloghi delle funzioni); aggiungiamo poi la regolarizzazione L_1 (Lasso) per migliorare le prestazioni del classificatore e la sua robustezza rispetto al rumore. Il metodo è descritto nel dettaglio nella sezione 3.2.

Affrontiamo il problema dell'identificazione della sezione di codice e della separazione tra codice e dati costruendo un modello di apprendimento sequenziale basato sulle *Conditional Random Field* (sottosezione 3.3.3). Il nostro modello esamina la sequenza dei byte nei file binari e apprende, per ciascun byte, la probabilità che esso rappresenti del codice o dei dati, condizionata rispetto ai valori delle *feature* e alla classe assegnata ai byte adiacenti. Il modello, una volta addestrato sui binari del *training set*, è in grado di dire se ciascun byte del binario da analizzare rappresenta del codice o dei dati. Lo scopo del metodo di identificazione della sezione di codice (sezione 3.3) è quello di trovare dei segmenti di codice (e di dati) contigui e di grandi dimensioni all'interno del file, mentre lo scopo del metodo di separazione tra codice e dati (sezione 3.4) è quello di identificare precisamente ciascuna istruzione macchina. Per questo motivo il modello per l'identificazione della sezione di codice include anche una fase di *postprocessing* (sottosezione 3.3.5) per identificare i piccoli segmenti di codice o dati rilevati dal modello, che vengono interpretati come rumore e quindi scartati.

Implementazione

Nel Capitolo 4, descriviamo l'implementazione del nostro approccio. Abbiamo implementato tutti i metodi di apprendimento usando Python 3 e la libreria di *machine learning* scikit-learn, ad eccezione di alcune operazioni di *preprocessing* che sono state eseguite tramite script per Bash e PowerShell.

Il classificatore di architetture — che oltre al modello vero e proprio comprende anche le fasi di preprocessing, di regolazione degli iperparametri e di validazione del modello — è implementato tramite il framework di *machine learning* per Python scikit-learn (sezione 4.2).

Per implementare i modelli di apprendimento sequenziale per l'identificazione della sezione di codice e la separazione tra codice e dati, utilizziamo `pystruct`, un framework di apprendimento strutturato per Python, che implementa le *Conditional Random Field*, apprese tramite SVM strutturali (l'approccio è descritto nel dettaglio in sezione 4.3).

Abbiamo implementato le fasi di *preprocessing* e di apprendimento all'interno di classi Python. Le nostre classi sono dei *wrapper* che incapsulano i modelli generali di classificazione forniti dalle librerie di *machine learning*. In questo modo, esponiamo un'interfaccia semplice e uniforme nascondendo la complessità non necessaria di tali modelli.

Per implementare il nostro approccio al problema della separazione tra codice e dati, necessitiamo dei dati di *training*, ossia un insieme di eseguibili per i quali siano note le posizioni delle istruzioni macchina e dei dati. In letteratura, questo problema viene spesso affrontato utilizzando un *disassembler* — che non sempre forniscono un risultato corretto — per fornire i dati di *training*. Noi scegliamo invece di estrarre i dati di *training* dai simboli di debug generati durante la compilazione degli eseguibili, che sono più affidabili. Questo approccio presenta alcune difficoltà tecniche, che affrontiamo nella sezione 4.5.

Risultati

Nel Capitolo 5, descriviamo la metodologia di valutazione dei metodi proposti in precedenza, e discutiamo i risultati degli esperimenti.

Per verificare i nostri metodi, utilizziamo l'*holdout testing* e la cross-validazione, che sono tecniche standard e consolidate nell'ambito del *machine learning* (sezione 5.3).

Abbiamo valutato estesamente il classificatore di architetture su più dataset di file eseguibili; i risultati mostrano che il nostro metodo ottiene performance migliori dello stato dell'arte (Clemens 2015) su eseguibili ottenuti da articoli scientifici e software reali; ottiene prestazioni soddisfacenti anche sugli *shellcode* e su binari sottoposti a *packing* (sezione 5.4). Il nostro modello è robusto rispetto al rumore: i risultati mostrano che esso è in grado di identificare correttamente l'architettura di file eseguibili contenenti sia sezioni con codice sia sezioni con dati.

Abbiamo valutato il metodo di identificazione delle sezioni di codice su quattro dataset contenenti file binari in tre formati (ELF, PE e Mach-O) e di più architetture hardware, compilati con diversi compilatori e livelli di ottimizzazione (sezione 5.5). I risultati mostrano che il nostro modello riporta sempre ottime prestazioni: la percentuale media di byte correttamente classificati supera il 99,5%.

Abbiamo valutato il metodo per la separazione tra codice e dati su un dataset di file binari per Windows (architetture x86 e x86-64) compilati con i simboli di debug. I risultati mostrano come il nostro metodo raggiunga un'elevata accuratezza (oltre il 99,9% di byte classificati correttamente) su questo dataset.

Abbiamo determinato i valori ottimali per gli iperparametri da utilizzare nei modelli in modo sistematico (sezione 5.7).

Contributi originali

I principali contributi del nostro lavoro di tesi sono i seguenti:

1. un metodo di apprendimento supervisionato, basato su un semplice modello lineare, per identificare l'architettura hardware e l'endianness dei file eseguibili (che possono contenere sia codice che dati);

2. un metodo di apprendimento sequenziale per identificare le sezioni di codice all'interno dei file eseguibili;

3. un metodo di apprendimento sequenziale per separare i byte che rappresentano istruzioni macchina dai dati inseriti all'interno della sezione di codice di un eseguibile.

Organizzazione della tesi

Questa tesi è divisa in otto capitoli.

- Nel Capitolo 1, forniamo un'ampia panoramica sui problemi che intendiamo affrontare, sugli obiettivi e sull'approccio proposto.
- Nel Capitolo 2, forniamo la definizione dei problemi, gli obiettivi della nostra tesi e le sfide da affrontare.
- Nel Capitolo 3, descriviamo in dettaglio gli approcci, gli algoritmi e i modelli che abbiamo scelto di utilizzare per risolvere i problemi descritti nel Capitolo 2.
- Nel Capitolo 4, spieghiamo come abbiamo implementato gli approcci descritti nel Capitolo 3 in un prodotto software concreto.
- Nel Capitolo 5, verifichiamo i nostri metodi su dati reali e diamo un'interpretazione dei risultati.
- Nel Capitolo 6, spieghiamo le limitazioni del nostro approccio.
- Nel Capitolo 7, proponiamo alcune direzioni per eventuali sviluppi futuri basati su questo lavoro.
- Nel Capitolo 8, riassumiamo il lavoro fatto finora e tracciamo le conclusioni.

Chapter 1

Introduction

Domain

Static binary analysis (i.e., analyzing executable code without running it and tracing its execution) is an effective approach to extract information from executable files containing computer programs when the source code is not available. Static binary analysis is part of the wider domain of binary reverse engineering. Two of the most important purposes of static analysis are: obtaining a correct disassembly of the machine code and building a correct Control Flow Graph (CFG).

Static binary analysis is possible only if some information about the executable file to analyze is available. The static analysis tools must at least know the **ISA (Instruction Set Architecture)** of the binary, and **the boundaries of the executable sections of the file** (i.e., the parts of the executable file which contain machine code). The ISA defines the encoding of the CPU instructions of a program into a byte sequence: static analysis tools need this information to decode the machine code, because the same sequence of bytes may have two completely different meanings for two different CPUs. Executable files are structured in sections, which include either code or non-executable bytes ("data"). Without the information about these sections, static analysis tools can not easily tell where the machine code is located inside the executable file.

The information about the ISA and about the sections inside the executable file is found in the header of the executable, if the file is provided in a standard format. Sometimes, however, the header is missing, and we do not have any information at all about an executable file. This is the case of the programs which run directly on a device's hardware (firmwares). Also, executable files may be obfuscated to thwart the attempts of reverse engineering: this is the case of malware files. In all these cases, we need to analyze the executable file without relying on the classical reverse engineering approaches.

Even if the ISA and the boundaries of the executable regions are known, the problem of obtaining a correct disassembly is still not trivial because some compilers insert *inline data* inside the code sections, which may be incorrectly interpreted as machine code by the disassemblers. Indeed, the header data found in the standard executable formats (ELF, PE, Mach-O, ...) does not include the information about the location of inline data inside the code section, unless the compiler is explicitly instructed to generate the debugging symbols. In most commercial and off-the-shelf programs, the debugging information is never included, because it would facilitate the reverse-engineering of the software product.

Problem definition and objectives

In this thesis, we will propose three novel approaches to solve three open problems in the domain of static analysis of executable computer programs. The three problems we will deal with are:

1. **Instruction Set Architecture (ISA) identification**: inferring the CPU architecture of stripped, header-less executable binary files;
2. **Code section identification**: determining where the executable section, containing machine code, is located inside stripped, header-less executable binary files;
3. **Code discovery problem**: inside the executable section of a binary file, accurately distinguish (at the byte level) between valid machine code instructions and inline data.

We will give a precise definition of these problems and describe the challenges associated with them in chapter 2.

State of the art

In section 2.3, we will present a thorough literature review about the state of the art approaches to solve the problems mentioned before.

Generic file type classification is a widely explored problem in literature (subsection 2.3.1); however, we found only one work specifically targeting the identification of the CPU architecture of executable files (subsection 2.3.2). This approach (Clemens 2015) addresses the problem as a machine learning classification problem, by using the frequencies of the bytes in the files as features. We feel that the approach of this work can be improved by introducing additional features and by tuning the parameters of the machine learning algorithms.

The problem of the classification of file segments by type (Sportiello and Zanero 2012) is similar to the problem of code section identification. The fundamental difference is that in our case we do not have any “a priori” file fragmentation: we must both segment the file and classify the segments.

Conditional Random Fields (Lafferty, McCallum, and Pereira 2001) are a supervised, structured classification method to predict a set of variables whose conditional dependencies are encoded in a graph-like structure. CRFs are popular in the domains of natural language processing, computer vision and bioinformatics. In (Rosenblum et al. 2008), the authors build a CRF-based model to identify functions in binary code.

The problem of separating code and data inside executable files (“code discovery problem”) is a classical problem in static binary analysis. Existing disassemblers fail to produce a correct disassembly when data is mixed together with machine code (subsection 2.3.3). The classical approaches rely on recursive disassembly (i.e., labeling as code all and only the locations reachable by the control flow of the program), and try to improve it by employing heuristics and symbolic execution; in subsection 2.3.5 we review three such approaches. The main challenge for these techniques is the precise identification of the targets of indirect jumps. (Shoshitaishvili et al. 2016, p. 4)

An alternative approach to the problem of code-data separation is to train a supervised model on the machine code, learning from the features found in code and data (we review this family of approaches in subsection 2.3.4). (Wartell et al. 2011) proposes a sequential learning model (*Prediction by Partial Matching*) to segment x86 machine code into valid instructions and data. The model obtains a high accuracy, being able to correctly classify more than 99.8% of the bytes as code or data. The model evaluation is done by manually comparing the output of the model with the disassembly from IDA Pro, because precise ground truth for the binaries in the training set is not available. This limitation does not allow to test the method on a large number of binaries. This approach supports a single architecture (x86), and relies on architecture-specific heuristics: it would need some effort to be adapted for new architectures.

Proposed Approach

In chapter 3, we propose an approach to solve the aforementioned problems.

To solve the CPU architecture identification problem, we improve the approach described in (Clemens 2015) by building a Logistic Regression classifier with additional features (i.e., patterns matching the function prologues and epilogues), and by adding Lasso (L_1) regularization to make the classifier more robust to noise. We describe our approach in section 3.2.

We address the code section identification and the code discovery problems by building a sequential learning model based on Conditional Random Fields (subsection 3.3.3). Our model looks at the sequence of bytes in the binary files: for each byte, it learns the probability that it represents code or data, conditioned on its value and on the class labels assigned to itself and to a fixed number of adjacent bytes. The model, trained on pre-tagged binaries, can identify the code-data boundaries (at the byte level) in unseen samples. The purpose of the code section identification method (section 3.3) is to find large, contiguous segments of code inside the file, while the purpose of the code discovery method (section 3.4) is to find the boundaries of each machine-code instruction. For this reason, the code section identification approach also includes a postprocessing phase (subsection 3.3.5) to delete small, noisy segments of code or data.

Implementation

In chapter 4, we describe the concrete implementation of our approach. We implemented all our learning methods using Python 3 and the machine learning framework scikit-learn; we implemented the preprocessing and data collection phases which required the usage of external tools with shell scripts.

We implemented the architecture classifier with the `LogisticRegression` class of scikit-learn (section 4.2). We wrote shell scripts to facilitate the collection and the transformation of some datasets. We implemented the preprocessing, hyperparameter tuning and model evaluation phases by taking advantage of the rich software library provided by scikit-learn.

To implement the sequential classification models for the code section identification and the code discovery methods, we used `pystruct`, a structured learning framework for Python, which implements Conditional Random Fields learned with structural SVMs (the mathematical model is described in section 4.3).

We built the preprocessing and the learning phases as Python classes. We designed our classes as wrappers for the general-purpose classification models provided by the libraries. In this way, we expose a simple, uniform interface by hiding the complexity beneath the general-purpose tools.

To train the code discovery model, we need accurate ground truth for the executables in the training set, i.e., the locations of the machine instructions and of the inline data inside the code section. In literature, the majority of the works we found addresses this problem by using existing disassemblers to produce an evaluation baseline; however, this approach may lead to inaccuracies. We choose to extract the ground truth directly from the debugging symbols generated by the compiler, which are more reliable. This approach presents some technical difficulties, which we describe and address in section 4.5.

Results Summary

In chapter 5, we describe the evaluation methodology for our approaches, the datasets, the experiments we ran and their results.

To validate our approaches, we use cross-validation and holdout testing, which are standard, well-established techniques in the machine learning domain (section 5.3).

We extensively evaluate the architecture classifier over multiple datasets of executables; the results show that our method performs better than the state-of-the-art approach (Clemens 2015) on executables obtained from research papers' datasets and real-world software; it obtains satisfactory performances even on shellcodes and packed binaries, by recognizing the Byte Frequency Distribution of the stub code (section 5.4). Our architecture classifier is robust to noise: the results show that it can correctly identify the architecture of executable files containing both code and data sections.

We evaluate the code section identification method on four datasets, containing binaries of different formats (ELF, PE and Mach-O), of different CPU architectures, compiled with different compilers and multiple levels of optimization (section 5.5). The results show that the performance of our model is consistently high: on average, the percentage of correctly classified bytes exceeds 99.5%.

We evaluate the code discovery method over a dataset of x86 and x86-64 Windows binaries compiled with full debugging symbols to obtain the ground truth. The results show that our method reaches a high accuracy (over 99.9%) on this dataset, correctly classifying nearly all the bytes.

We determine the optimal values for the hyperparameters of the models in a systematic way (section 5.7).

Original Contributions

The main contributions of our work are the following:

1. a supervised classification method to identify the Instruction Set Architecture and the endianness of executable files;

    - our method is robust to noise: it works on executables containing pure machine code, as well as on samples containing code and data, and packed code;

    - our method supports, but does not require, architecture-specific signatures: more architectures can be supported by simply extending the training set;

2. a novel application of a sequential learning model (CRF) to reliably locate the boundaries of the executable regions inside binary files;

3. a novel, simple and reasonably fast approach, based on a sequential learning model (CRF) to distinguish code from data in stripped and

header-less binaries, for two architectures with variable-length instructions (x86, x86-64) and an architecture with fixed-length instructions (ARM);

  - an automated technique to generate a training set of Windows executables for the code discovery method, by extracting the ground truth from the debugging symbols generated by the compiler.

Thesis Organization

This work is structured in eight chapters.

- In chapter 1, we give a broad overview about the problems we will be dealing with, the goals of our work, the proposed approach and the results.
- In chapter 2, we precisely state the definition of the problems, the state-of-the-art solutions to such problems, the goals of our thesis and the challenges we will be facing.
- In chapter 3, we describe in detail the approaches, the algorithms and the models we choose to use to solve the problems stated in chapter 2.
- In chapter 4, we explain how we implemented the approaches described in chapter 3 in a concrete software product.
- In chapter 5, we evaluate our methods on real data and give an interpretation of the results.
- In chapter 6, we analyze the limitations of our approach.
- In chapter 7, we give some suggestions for future research, starting from what we developed in this work.
- In chapter 8, we summarize the work done so far and draw the conclusions.

Chapter 2

Motivation

In this chapter, we will give some background information to contextualize the domain of binary reverse-engineering (section 2.1). After this necessary step, we will precisely define the problems which we are going to address (section 2.2). Then, we will describe the state of the art approaches in the literature (section 2.3). We will conclude by listing the goals and the challenges (section 2.4) associated with the problems mentioned before.

2.1 Background

An executable file contains a computer program, in a format suitable for the operating system to load it in memory and run it. Executable files are generated as the result of the compilation of the source code of computer programs. While the source code of a program is written in a high-level, understandable programming language, the machine code present inside the executable ("binary file") is a set of hardware-dependent, low-level instructions executed by the Central Processing Unit (CPU) of the computer. Machine code in executable files lacks the constructs, the structures and the semantics of the source code; therefore, extracting useful information from the executable file alone, without having the source code available, is a challenging task (Kruegel et al. 2004, p. 1). Moreover, compilers often transform and optimize the code, to reduce the size of the code and to make it run faster. These optimizations make static analysis harder (Andriesse, X. Chen, et al. 2016, p. 8).

Reverse-engineering an executable file means extracting useful information from the binary file, without the source code being available. Static analysis of executable files is a branch of binary reverse-engineering which consists in analyzing the executable file without running it, nor tracing its execution. A typical task in binary reverse engineering is the *disassembly* of the program, i.e., "the extraction of the symbolic representation of the instructions (assembly code) from the program's binary image". (Kruegel et al. 2004, p. 1)

2.1.1 Executable file formats

The software applications running on the most popular desktop and mobile operating systems are shipped into standard executable file formats. In this work, we will focus on three of these formats:

Executable and Linkable Format (ELF) The standard format (*Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2* 1995) for executable files in Linux-based operating systems.

Portable Executable (PE) The standard format (Microsoft 2017[f]) for executable files in the Windows family of operating systems.

Mach-O The standard format (*Mac OS X ABI Mach-O File Format Reference* 2007) for executable files in the operating systems running the Mach kernel (including Apple's macOS).

The standard ELF, PE and Mach-O executable formats include a *header* which provides metadata such as:

- the Instruction Set Architecture (ISA) of the executable;
- the endianness;
- the number, offsets and lengths of the sections (code, static data, relocations, . . .);
- the base address of the executable, i.e., where the operating system loader will place the code in main memory;
- the imported and exported functions;
- the relocations, if present;
- (optionally) the debug symbols;

The executable files in the ELF, PE and Mach-O formats are structured in *sections*; a set of flags in the header determines the role of each section (e.g., executable, writable, read-only data, . . .). In particular, some sections are executable, and others are not. Executable sections contain machine code and (sometimes) inline data mixed with the instructions. Non-executable or “data” sections may contain statically initialized variables, padding bytes, strings, relocations and other information needed by the program.

2.1.2 Disassemblers

Disassemblers are computer programs which take an executable file containing machine code as input and decode it into readable assembly code. Disassemblers are a class of widely used tools in the binary reverse-engineering domain.

Machine code may follow non-linear execution paths: a class of CPU instructions allows to *jump* or conditionally *branch* to any arbitrary code address, or to a relative offset. This means that some regions of the binary file are never reached by the execution. Those unreachable regions may contain valid but “dead” code, or data.

The problem of statically (i.e., without executing the program) determining whether the execution flow will reach a generic location in machine code is an undecidable problem, since it reduces to the halting problem (Wartell et al. 2011, p. 11). This means that also the correct disassembly of binary files is undecidable.

One of the tasks disassemblers must deal with is determining which bytes in the byte stream correspond to valid machine code instructions. Some ISAs have a dense opcode space (Rosenblum et al. 2008): every byte can be interpreted as the beginning of a valid CPU instruction, so it is not possible to tell apart code from data by only looking at the bytes themselves.

There are two families of disassemblers, which deal with this problem in different ways:

Linear disassemblers start from the first byte in the file and decode all the valid instructions sequentially. They may react to the presence of inline data between the instructions in different ways:

- if the data can be decoded as a valid instruction, the disassembler decodes it, even if those bytes do not represent a CPU instruction (false positive);
- if the data cannot be decoded as a valid instruction, the disassembler marks it as invalid (best case);

Some CPU architectures have instructions of variable length: it is not trivial to tell where an instruction starts, because the incorrect decoding of some data as code may “dis-align” the disassembly and lead to the incorrect decoding of the next instructions. x86 disassembly errors tends to “repair” themselves after few instructions: this property is known as *self-repairing disassembly.* (Linn and Debray 2003)

Recursive traversal disassemblers start from the *entry point(s)* of the executable and start disassembling until they find a control flow instruction (jump or branch). When a control flow instruction is reached, the disassembler attempts to determine all the *successors*, i.e., all the addresses the CPU may jump to. This disassembling procedure is recursively restarted from each of the jump targets. The problem of statically determining the set of successors of a branch instruction is, once again, undecidable; the jump target for a branch instruction may be dynamically computed by the program itself and stored in a registry, or in memory. This leads to two classes of errors in recursive disassembly:

- if some jump targets are missed, the disassembler will miss valid code (false negatives);
- if more jump targets than the true ones are detected, the disassembler may decode portions of data as instructions (false positive).

In all the cases where the set of jump targets is precisely determined, recursive traversal disassemblers are able to avoid inline data and disassemble all and only the reachable instructions.

Not all the compilers insert inline data into the instruction stream: for example, the most recent versions of GCC and clang place all the data in the `.rodata` section. Microsoft Visual Studio inserts data and padding bytes directly into the instruction stream, when compiling for the x86 and x86-64 architectures (Andriesse, X. Chen, et al. 2016). ARM code often contains jump tables embedded into the instruction stream (J.-Y. Chen et al. 2013).

The presence of data interleaved with instructions into the executable regions of binary files hinders the correct disassembly by both linear-sweep and recursive disassemblers.

2.2 Problem Statement

In this work, we present a supervised learning approach to solve three tasks relative to the reverse engineering of executable files:

1. **CPU architecture identification**: inferring the CPU architecture of stripped, header-less executable files;

2. **Code section identification**: determining where the code section is located inside stripped, header-less executable files;

3. **Code discovery problem**: inside the executable sections, accurately distinguish (at the byte level) between valid CPU instructions and inline data inserted between instructions.

We will analyze each of these problems in depth in the following sections.

2.2.1 CPU architecture identification

One of the first steps in the analysis of an executable file is to precisely determine the Instruction Set Architecture (ISA) of the executable, together with its endianness.

The **Instruction Set Architecture (ISA)** defines the syntax of the *machine code* of a processor (CPU). In particular, the ISA defines the allowed data types, the instruction set (e.g., the set of all the allowed instructions and their semantics), the encoding of the instructions into binary machine code and the state of the machine (the registers).

Different models of CPUs have different ISAs; a program compiled for an ISA can not run on a CPU with a different ISA.

The **endianness** of a binary file is defined as the order in which the bytes are stored in a CPU register (Blanc and Maaraoui 2005). There are two main endianness conventions:

- **Big-endian**: the Most-Significant-Byte is stored at the lowest address;
- **Little-endian**: the Least-Significant-Byte is stored at the lowest address.

Other orderings of the bytes are possible: they are generically called *middle-endian*.

If the program to analyze is shipped in a standard format, the information about the ISA and the endianness is present in the header of the executable file (see subsection 2.1.1 for details).

In some cases, however, the analyst may have no information at all about an executable. This often happens with binary blobs, or firmwares extracted directly from the hardware of a device. We assume to be in this situation: no header information is given; we only have a sequence of bytes, and no information to interpret it.

It is impossible to disassemble an executable file if the ISA or the endianness are not known, because without this information the machine code inside the executable file is just a sequence of bytes which can not be decoded.

The first problem that we will face in this work is to label any unknown binary file with the correct architecture and endianness.

2.2.2 Code section identification

To accurately disassemble an executable file, the disassembler needs to know where the code section is located inside the file.

The information about the starting offset, the length and the type of each section of the executable file is usually stored into the header. In some cases, however, the header may be missing, and there is no trivial way to know where the machine code is located.

We assume that the ISA of the binary file is known, and no further information about it is available. The problem consists in finding where the sections containing valid executable code are located into the unlabeled binary files.

2.2.3 Code discovery problem

Even inside the code section of an executable, not all the bytes represent valid, executable CPU instructions.

Indeed, some compilers (e.g., Visual Studio) insert small sequences of data inside the code section (Andriesse, Slowinska, and Bos 2017). These small pieces of data are typically strings, jump tables, or padding bytes between functions.

Clearly, when data is present in the instruction stream, the compiler makes sure that the execution flow never reaches it: otherwise, the program would crash or behave incorrectly.

The presence of data inside the code section of executables is one of the major issues preventing the correct disassembly of binary files. In fact, the problem of correct disassembly is undecidable (Wartell et al. 2011, p. 11).

The problem we want to solve is the *code discovery problem* (J.-Y. Chen et al. 2013): given a sequence of bytes containing executable machine code and data, label each byte as code or as data with the highest accuracy possible.

2.3 State of the Art

2.3.1 Generic file type classification

The classification of files by their type is a common problem in computer forensics. Most of the works dealing with file type identification are not specifically targeted toward the CPU architecture identification problem: all the executable files are put in the same class, because a fine-grained classification is beyond the scope of these works. In this research area, we find some useful techniques aimed at classifying binary files by only looking at their contents.

The first publication to propose a machine learning approach to classify computer files by their type, and to extract the features from the Byte Frequency Distribution of files, is **(McDaniel and Heydari 2003)**. The algorithm based on the BFD achieved a modest 27.50% accuracy over 30 classes.

(Li et al. 2005) improves on the same idea by clustering the files by their BFD and representing each file type with a set of centroids. On six classes, the best method achieves an average accuracy of 93.8%. The authors observe that the features can be extended by adding the frequency of N-grams, i.e., the count of the number of occurrences (even overlapping) of any N-byte sequence. The authors observe that 1-grams offer the best tradeoff between accuracy and computational complexity, which is the same result reported by (Clemens 2015).

In **(Sportiello and Zanero 2012)**, Support Vector Machines (a supervised learning model) are used to classify the type of 512-byte file segments, a recurring task in forensic analysis (e.g., when recovering files from a damaged hard drive). In addition to the features extracted from the Byte Frequency Distribution, additional features like Rate of Change, Mean Byte Value, Entropy and Lempel-Ziv complexity are also computed. The average classification performance is 90.4% for the True Positive Rate and 12.4% for the False Positive Rate, on 9 file classes. The authors observe that any attempt to work with a set reduced or compressed features derived from the BFD produces worse results than the model trained on the full BFD. Another important observation, supported by an experiment carried out in the paper, is that by increasing the size of the blocks, the classification performance also increases. The methods described in this paper can be used to classify the executable files by their ISA. This approach is not suitable to solve the code section identification problem, because the segmentation of the executable files is not known *a priori*: we need to solve both the segmentation and the classification problems.

(Penrose, Macfarlane, and Buchanan 2013) contains an extensive literature review about the classification of high-entropy file fragments and claims that existing techniques are insufficient to classify compressed or encrypted file segments (which are beyond the scope of our work).

2.3.2 Classification of executable files

(Clemens 2015) addresses the problem of the identification of the CPU architecture of executable files as a classification task: 9 different machine learning models are trained over a dataset of 20 different CPU architectures.

The features are extracted from the *Byte Frequency Distribution* (BFD) of the files, i.e., there are 256 features and the i-th feature represents the frequency of occurrence of byte i in the file:

$$freq(i) = \frac{count(i)}{N} \tag{2.1}$$

In addition to these features, four bi-gram features are added to detect of the endianness of the binary files. Those are the frequencies of the bi-grams `0x0001`, `0x0100`, `0xfffe`, `0xfeff`. They represent – in both the big endian and little endian encodings – the frequency of the positive integer number "1," and the frequency of the two-byte sequence `0xfffe` which is the prefix of "high" memory addresses.

The paper evaluates and compares 10 different machine learning models (1-NN, 3-NN, Decision Tree, Random Tree, Random Forest, Naïve Bayes, Bayesian Networks, Logistic Regression, SVM, Neural Network) on the same dataset. The best performing algorithm (SVM) reports an accuracy of 98.35%, but all the models perform well, even without hyperparameter tuning.

If we look at the disaggregated data reported for the Logistic Regression model, however, we find that some classes perform worse than others. MIPS (big endian) and MIPS (little endian) have a F-measure of 88.4% and 88.6% respectively; CUDA performs quite bad at 51.6% and AVR stops at 93.6%. All the other architectures have a F-measure greater than 99%.

In the case of CUDA, the poor performance is simply explained by the low number of samples in the dataset (17). The two MIPS architectures differ only for their endianness; we suspect that this may confuse the classifier, even with the bi-gram features. Unfortunately, the paper does not provide a confusion matrix to check which couples of classes are more likely to confuse the model.

The dataset is obtained by extracting the code section of binaries coming from 20 different architectures. The assumption here is that we know where the code section is located in each binary, and that we can isolate it from other data present in the original binary file. This assumption limits the practical applications of this work: if the location of the code section is known, it is likely that a header is present and that we also know the CPU architecture.

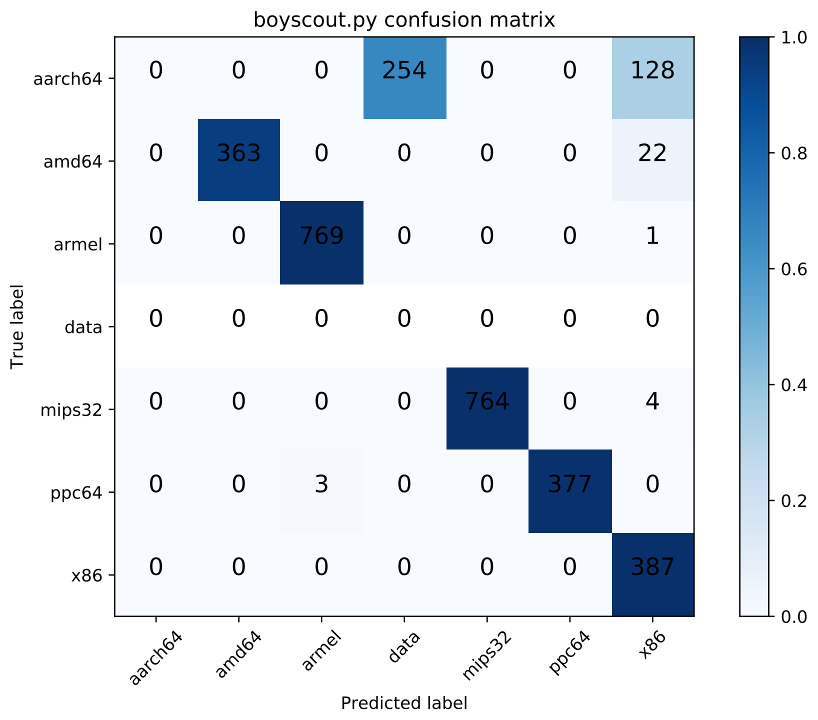
All the models were trained with their default parameters (except for the Neural Network classifier): hyperparameter tuning may increase their performances. Indeed, for some models, the choice of the parameters (e.g., the pruning criterion for Decision Trees, or the regularization strength for Logistic Regression and SVMs) is critical to obtain a correct bias-variance tradeoff. (Bishop 2006, p. 665)

angr's Boyscout

The binary analysis framework `angr` **(Shoshitaishvili et al. 2016)** provides a tool (Boyscout) to identify the CPU architecture of an executable. The tool implements this simple algorithm: for each architecture-endianness couple there is a set of regular expressions matching the byte patterns corresponding to the function prologues and epilogues. One match for a pattern is a “vote” for the corresponding architecture. The architecture-endianness couple with the most votes is returned.

We evaluated Boyscout on a dataset of binaries of different architectures, downloaded from the Debian package repository (we refer to subsection 5.2.3 for a detailed description of this dataset).

To evaluate Boyscout, we loaded each binary of the dataset into `angr`. Since the binaries in our dataset are regular ELF files, `angr` can detect the architecture and the endianness of each file by looking at its header. This provides our ground truth, i.e., the “true” architecture of the binary in the `angr`'s naming scheme. We manually filled in the class label of the binaries which were not recognized by `angr`. Then, we reloaded each file as a “blob” into `angr` and ran the Boyscout architecture detection algorithm. The confusion matrix which compares Boyscout's predictions with the true architecture of the binaries is shown in Figure 2.1.



	aarch64	amd64	armel	data	mips32	ppc64	x86
aarch64	0	0	0	254	0	0	128
amd64	0	363	0	0	0	0	22
armel	0	0	769	0	0	0	1
data	0	0	0	0	0	0	0
mips32	0	0	0	0	764	0	4
ppc64	0	0	3	0	0	377	0
x86	0	0	0	0	0	0	387

Figure 2.1: Confusion matrix of Boyscout on the Debian dataset.

The accuracy of the Boyscout detection algorithm, computed as the percentage of samples for which Boyscout correctly detects both the architecture and the endianness, is 86.6%.

Some of the `amd64` binaries are erroneously classified as `x86`: this is expected since the two architectures are similar. Boyscout is unable to recognize any of the `aarch64` (ARM 64-bit) binaries. This happens because, although the architecture is supported by `angr`, its patterns are missing. If we remove these binaries from the dataset, the accuracy becomes 98.9%.

Boyscout's approach is simple and precise (for a subset of architectures); however, it has the following shortcomings:

1. the tool must include a signature database of function prologues and epilogues for each architecture, which requires constant maintenance (Andriesse, Slowinska, and Bos 2017);

2. no feature selection: every regex match carries the same weight;

3. shorter matches are more likely to occur in random data: architectures with shorter prologues and epilogues are favored;

4. prologues and epilogues can be missing in heavily optimized or obfuscated code.

2.3.3 Real-world disassembler accuracy

Before investigating novel research methods to distinguish code from data in executable files, we will now describe how existing disassembler frameworks behave on real-world binaries.

(Andriesse, X. Chen, et al. 2016) provides an in-depth analysis about how state-of-the-art disassemblers perform on x86 and x86-64 binaries, and reveals a notable mismatch between the true capabilities of the disassemblers and the overly pessimistic expectations reported in 30 academic papers.

The first experiment evaluates the accuracy of the detection of instruction boundaries, evaluating both linear sweep and recursive traversal disassemblers. As mentioned in subsection 2.1.2, linear disassemblers and recursive traversal disassemblers differ about how they discover the instructions inside the binary stream, and these differences affect the behavior of such disassemblers when inline data is mixed into the instruction stream. Linear sweep disassemblers yield the best performance on the most difficult dataset (binaries compiled by Visual Studio), finding 99.92% of the real instructions, with a false positive rate of 0.56% of the decoded instructions. The false negatives are much less, at 0.09%. By comparison, the best recursive traversal disassembler, IDA Pro, has an accuracy between 99% and 96% depending on the optimization level of the binaries.

The performance of linear-sweep disassemblers depends on the amount of non-padding data that is inserted by the compiler into the instruction stream. If more data were present, the performance would decrease. For example, (Linn and Debray 2003) reports that, by inserting "junk" into the instruction stream, they can make linear disassemblers classify 26%-30% of the instructions incorrectly. The same paper explains more sophisticated obfuscation techniques to confuse recursive disassemblers.

So, the comparison among disassemblers and code discovery approaches must be done carefully, since the amount of data inserted in the instruction stream can dramatically change the performance of the approach.

2.3.4 Statistical code discovery methods

In this section, we will review some of the works which address the problem of code discovery as a supervised classification problem. The general idea is to solve the code discovery problem by classifying every byte in the executable file either as "code" or "data". This is done by learning a *probabilistic language model* of the code and data bytes and then compute the most likely class for each byte, by using the values of the surrounding bytes (often referred as *context*) as features.

An important contribution in this field is **(Wartell et al. 2011)**. The algorithm described in the paper follows these high-level steps:

1. the bytes are divided into segments by using an *instruction reference array* and a utility function;

2. *Predication by Partial Matching* (PPM), a model similar to a *k*th order Markov model, classifies each segment as code or as data;

3. additional heuristics are used to improve the classification accuracy (e.g., data is more likely to appear after an unconditional jump).

PPM is an algorithm that performs a dynamic context match, i.e., predicts the probability of appearance of a symbol in a sequence, conditioned on the previous k symbols. If no match is found for a prefix of length k, an *escape event* is recorded and the model considers a prefix of length $k-1$. Two language models are built, one for code and another for data. Each segment is then classified with the class that gives the highest likelihood.

The paper does not explain precisely how the outcomes of the heuristic criteria are merged with the classification result from the PPM model.

The experimental method, evaluated over 11 real-world Windows binaries, correctly classifies the segments with high accuracy (the worst being 99.893%), overcoming the performance of IDA Pro which is recognized as one of the top-performing recursive disassemblers.

The ground truth data is obtained by manually comparing the disassembly given by IDA Pro with the results of the proposed method. This evaluation methodology has the following drawbacks:

1. the quality of the ground truth depends on the skills of the human analyst;

2. if the classification of a segment is mistaken by both the experimental method and IDA Pro, the error is not considered;

3. the need for manual checking does not allow to test the method on a large dataset of binaries;

4. whenever the method changes, the ground truth must be regenerated.

Indeed, the author acknowledges that the development of an automated ground truth generation process is still an open problem.

(J.-Y. Chen et al. 2013) addresses the code discovery problem in ARM binaries. The final objective of this work is to perform a correct *static binary translation*, i.e., the conversion of an executable from an ISA to another without recompiling from source. ARM is a 32-bit RISC[1] architecture with fixed-length instructions; however, the same executable may contain ARM code (32-bit) mixed with Thumb code (16-bit). This means that the instructions can be 4-byte or 2-byte aligned in different regions of the executable:

[1]RISC: *Reduced Instruction Set Computer*

the segmentation is not trivial as in pure ARM code where all the instructions are aligned at the addresses multiples of 4 bytes.

The proposed approach starts by analyzing the binary with a conservative recursive traversal, following jumps only when possible. The regions found in this way can be safely classified as ARM or Thumb by their address. The algorithm identifies embedded data by looking at PC-relative addressing in `load` instructions. Anything that was not classified either as ARM, Thumb or data is translated twice into the resulting executable: first as ARM code, and then as Thumb code. At run-time, the correct version will be chosen (or never reached, in the case of inline data). This approach increases the size of the binary but ensures its correctness.

The algorithm is interesting for the analysis of ARM code, but it is specifically targeted for that architecture. It can not be extended to "true" variable-length instruction ISAs like x86. Also, this algorithm is unable to classify all the bytes as code or data: some are left undecided.

(Karampatziakis 2010) presents the code discovery problem in x86 binaries as a supervised learning problem over a graph. x86 is a CISC[2] variable-length instruction architecture: it represents the most general and difficult case (Andriesse, X. Chen, et al. 2016, p. 3). The proposed approach uses structural SVMs to classify bytes as code or data. A *trellis graph* is used as a data structure to express constraints over the classification outcome. The vertices of the graph represent the bytes of the binary file. For example, if the classifier decides that the byte in position i is the start of an instruction having length 7, this implies that no data is present from i to $i + 6$, and no other instruction can start in that interval. This ensures that the classification outcome is consistent.

The author claims that this model is the most convenient tradeoff between simpler models like linear-chain Conditional Random Fields and more expressive, complex and slower models like semi-CRFs and semi-Markov SVMs.

The ground truth is extracted by disassembling 200 programs for Windows XP with OllyDbg and checking the results against IDA Pro. This may prove problematic because, as we explained in subsection 2.1.2, no disassembler yields a perfect disassembly.

Two types of features are associated with each edge in the graph: byte-level features and instruction-level features. Byte-level features are extracted from an 11-byte window centered around the source of the edge; instruction-level features are obtained from the histograms of instructions in the candidate code blocks. The total number of features is 2.3 million. Three loss functions are defined: Hamming loss, block loss and instruction loss. The Hamming loss expresses the fraction of the bytes correctly classified by the model.

The original paper reports the normalized losses multiplied for the average number of bytes, blocks and instructions. To allow comparisons with the other

[2]CISC: *Complex Instruction Set Computer*

	$\bar{L} \cdot \Delta_{NH}$	$\bar{L}$	Δ_{NH}
Greedy	1916.6	16712	0.1147
SVM^{hmm}	201.3	16712	0.0120
$SVM^{wis}\Delta_I$	115.6	16712	0.0069
$SVM^{wis}\Delta_{NI}$	103.7	16712	0.0062
$SVM^{wis}\Delta_B$	98.2	16712	0.0059
$SVM^{wis}\Delta_{NB}$	87.2	16712	0.0052

Table 2.1: Normalized Hamming losses for the models in (Karampatziakis 2010)

methods, we report in Table 2.1 the normalized Hamming losses (Δ_{NH}) for all the models evaluated in the paper, by dividing the de-normalized Hamming loss ($\bar{L} \cdot \Delta_{NH}$) by the average number of bytes ($\bar{L}$).

(Rosenblum et al. 2008) addresses the problem of Function Entry Point (FEP) identification in stripped binaries (i.e., those not including debug information) for the Intel IA-32 architecture, which is a CISC architecture with variable-length instructions. More specifically, for each byte x_i in the binary, the task is to predict whether x_i is the first byte of a function (the Function Entry Point), or not.

The paper states that simple heuristics such as pattern matching on function prologues are not sufficient to successfully perform this task. Also, it observes that recursive disassembly fails to identify approximately 40% of functions in binary code because of indirect control flow transfers, i.e., jump targets which are resolved at run-time. There are *gaps* between discovered functions, which contain undiscovered code as well as data like constants, padding bytes and jump tables.

The proposed method to identify Function Entry Points employs a *linear-chain Conditional Random Field (CRF)* (Lafferty, McCallum, and Pereira 2001) for structured classification in sequences. Linear-chain CRFs are a supervised, structured, probabilistic learning method able to predict a label for each item in a sequence.

The authors propose two classes of features:

1. **Idioms** are patterns of assembly instructions; feature selection is used to select the most significant ones;

2. **Structural features** express two properties of the binary code:

    a) *Binary call-consistency*: if we assume that byte x_i is a FEP, and the function starting at x_i performs a call to the location x_j, then x_j is likely to be a FEP.

    b) *Binary overlap*: if the functions starting at two candidate FEPs x_i and x_j overlap, is unlikely that both x_i and x_j are actually FEPs.

The dataset is composed of 616 binaries compiled with GCC on Linux, 443 Windows XP system binaries, and 112 binaries compiled with ICC on Linux. The ground truth is obtained by parsing the debug symbols generated by the compiler, which contain accurate information about FEPs.

The feature selection and training phases for this method are expensive: the authors use a distributed computing environment to reduce the processing time by parallelizing the computations. The feature selection phase took "less than two days" in real time, which translates in 150 compute-days on the cluster.

The performances of the model are superior to the state of the art of the available disassemblers. The $F_{0.5}$ measure for GCC binaries is 98.9%, while in the worst case it is 85.9% for binaries compiled with ICC.

The code discovery problem is in some respects similar to the FEP identification problem, so a structured learning model may be a promising candidate solution; however, simpler and more straightforward features should be used, to avoid the expensive feature-selection phase which has to be repeated for each different dataset.

The purpose of the work presented in **(Shin, Song, and Moazzezi 2015)** is to identify function boundaries, i.e., the first and the last byte of each function. The approach considers directly the bytes as features, with minimal preprocessing. The proposed model is a single-layer bi-directional *recurrent neural network* which works similarly to a linear-chain CRF. Each byte corresponds to one hidden state, which is a node in the neural network. Each node is connected to the previous and to the next node. The model has no concept of "instruction"; instead, it directly associates the bytes with the functions.

This method consistently improves over the results of ByteWeight (Bao et al. 2014), obtaining a F_1 measure in the range of 98%-99%, with a shorter prediction time than ByteWeight. The training time is also shorter than ByteWeight's, but still high: cross-validation on the neural network took 80 compute-hours, vs. the 586.44 compute-hours needed by ByteWeight.

The authors evaluated the model on the same dataset used in the ByteWeight paper, consisting in ELF and PE executables, compiled for x86 and x86-64. ELF executables were compiled with both GCC and ICC, at four different levels of optimization.

The good results obtained in this work suggest that statistical techniques represent a promising research direction in binary analysis, and can be successfully applied to the code discovery problem.

2.3.5 Improving recursive traversal disassembly

(Bao et al. 2014) introduces an analysis framework, called ByteWeight, which solves three problems, described as follows:

1. **Function Start Identification** consists in finding the address of the first byte of each function, as in (Rosenblum et al. 2008).

2. **Function Boundary Identification** consists in finding the starting and ending addresses of each function; it is equivalent to the code discovery problem, if we assume that inline data is only located outside the function body.

3. **Function identification** consists in finding the set of bytes corresponding to each function in the binary code: a solution to this problem would also solve the code discovery problem.

For each of these problems, the paper defines an "oracle" providing the ground truth data. Obtaining accurate ground truth is paramount for statistical and machine learning methods. The *Boundary Oracle* is obtained by parsing debugging symbols derived from the compilation process.

The proposed approach for the function start identification problem consists in a *weighted prefix tree* which learns the function start patterns and their associated likelihood (weights) from the training set, in a compact form. The model accepts as features either byte patterns, or instruction patterns, obtained by disassembling an instruction starting from each byte (*exhaustive disassembly*).

After recovering the function starting points, ByteWeight infers all the instructions and the bytes belonging to each instruction with a recursive disassembly augmented with Value Set Analysis (VSA) to resolve indirect jumps. VSA over-approximates the destinations of indirect jumps. For a detailed description of VSA, we refer to (Shoshitaishvili et al. 2016). ByteWeight builds the Control Flow Graph (CFG) of the binary and recovers the function end addresses by using recursive disassembly, VSA and some heuristics.

ByteWeight is a hybrid approach: first, it employs a machine learning approach to infer the function start addresses, then it uses binary analysis techniques to construct the Control Flow Graph.

The authors evaluate their method over a dataset of 2,064 binaries for Linux and Windows, compiled with debug information. Linux binaries are ELF files, compiled for the x86 and x86-64 architectures, with the GCC and ICC compilers, using 4 different levels of optimizations, with debug symbols. Windows binaries are PE files, compiled for the x86 and x86-64 architectures, with three versions of Visual Studio, with debug symbols (contained in separate PDB files). The ground truth for the dataset is extracted from the symbol tables (for the Linux binaries) and from the PDB files (for the Windows binaries).

The performance of the function start identification algorithm is consistently higher than (Rosenblum et al. 2008) and commonly used disassemblers (Dyninst, BAP, IDA Pro).

The main drawback of ByteWeight's approach is that it requires long training and prediction times. The author claim that the training phase executed on 2,064 binaries took 586.4 compute-hours, while testing took 256 compute-hours for the function boundary recovery algorithm. The most expensive phase of the computation is Value Set Analysis, which resolves the targets of the indirect jumps.

(Kruegel et al. 2004) addresses the code discovery problem in obfuscated binaries, and proposes a hybrid approach which combines control-flow based and statistical techniques. The purpose of this method is to deal with the obfuscation techniques presented in (Linn and Debray 2003), specifically:

1. the insertion of junk bytes in unreachable locations;
2. the replacement of function calls and returns with indirect branch instructions.

The first phase of the algorithm extracts the Control Flow Graph (CFG) of the program by using a variant of the recursive disassembly algorithm. This method provides an algorithm to resolve the conflicts between overlapping basic blocks.

The second phase is aimed at "filling the gaps" between the basic blocks discovered so far, which may contain either data or valid instructions. Inside these gaps, all the possible valid sequences of instructions are identified; the method assigns a *sequence score* to each of them, which reflects the likelihood that the instruction sequence appears in the executable and represents valid, executable code. The gap completion method computes the likelihood of the single opcodes and of the couples of opcodes present in binary code, and stores the results in probability tables. The score for an instruction is computed as the sum of the probabilities of occurrence of the current opcode, and of the current opcode followed by the next one in the observed sequence. Architecture-specific heuristics are also used to improve the probability tables, ruling out "improbable" instructions by setting to zero their probability whenever the instruction sequence matches a certain rule.

A recursive disassembler with speculative linear disassembly and augmented with external information about function starts and ends provides the ground truth for evaluation.

The paper divides the disassembly techniques into general techniques and tool-specific techniques; the latter are tailored for the obfuscation methods in (Linn and Debray 2003).

The average disassembler accuracy, without using tool-specific techniques, is 90.10% as reported in the paper; by using tool-specific techniques, the average accuracy improves to 96.92%. Disassembler accuracy is defined as $1 - \frac{|V-O|}{V}$, where V is the set of valid program instructions and O are the set of instruction that the disassembler discovers.

According to (Rosenblum et al. 2008) a limitation of this approach is that "these heuristics and simple statistical methods cannot adapt to variations in compiler, optimization level, and post-compilation optimization tools". For instance, one of the assumptions is that the function prologues are easily identifiable by looking for common patterns. Extending the function identification algorithm to overcome the obfuscation of function prologues would require tool-specific knowledge and other heuristics.

`angr` (Shoshitaishvili et al. 2016) is a binary analysis framework which performs CFG recovery by recursive disassembly, augmented by advanced, state-of-the-art techniques to solve indirect jumps. Specifically, angr implements symbolic execution, backward slicing and Value Set Analysis with the help of an algebraic solver. Retrieving the full CFG with these techniques is expensive in terms of computation time. Also, it is impractical to analyze header-less binary because these techniques require the executable to be correctly loaded in angr; the base address and the entry point must be provided for the CFG recovery algorithms to work. angr provides an analysis (Girlscout) which should be able to identify the base address, but at the current time the code is broken.[3]

2.4 Goals

The first goal of this thesis is to build a system able to identify the CPU architecture of an arbitrary executable file. The system must accept any sequence of bytes (corresponding to an executable file) as input and return the predicted CPU architecture, or an ordered list of the most likely architectures. If an architecture has both the little-endian and the big-endian variants, those are considered as separate architectures.

The goals that we want to achieve are the following:

Speed The classifier must give the prediction results quickly. Training should also be reasonably fast.

Simplicity The classifier should use a simple, straightforward model, with few parameters to tune.

Domain knowledge The analyst must be able to integrate his knowledge of the domain by adding new features to the model; the features should be sufficiently expressive.

Automatic feature selection The system must be able to tell which features are the most useful to discriminate between architectures.

[3]GitHub issue: `https://github.com/angr/angr/issues/447`

Zero knowledge about the binary The prediction must depend only on the byte sequence of the executable file, without relying on any metadata. Headers, debug info and symbols are always assumed to be missing.

Robustness Noise in binary files should not confuse the classifier. In particular, we do not assume to know where the code section of the executable is located: the classifier should be able to take the whole binary as input and output the correct classification outcome.

Extensibility The analyst must be able to extend the set of supported architectures simply by extending the training set.

The second goal of this work is to build a system to identify the boundaries of the executable sections of unknown, header-less binaries. More precisely, the input of the problem is a binary file which consists of one or more code sections containing machine code, and other sections containing data. We assume to know the CPU architecture of the executable file. The output of the system is a set of ranges corresponding to the predicted locations of the executable sections.

The requirements for this system are:

No external knowledge The only information available is the sequence of bytes and the CPU architecture of the binary file.

Extensibility The analyst must be able to extend the set of supported architectures simply by extending the training set.

Speed Prediction should be done in real time, and training should be reasonably fast.

The third goal of this work is to solve the code discovery problem, i.e., to build a system to classify each byte in the executable section of a binary as code or inline data. The system accepts as an input any arbitrary sequence of bytes; we assume to know the CPU architecture of the binary. The model must classify each byte either as a part of a machine instruction, or as data.

The requirements for this system are:

Very high accuracy State-of-the-art tools perform very well (Andriesse, X. Chen, et al. 2016) in this task, and we want to further improve the performance.

Simplicity and generality We want to build a straightforward, general model which is easily adaptable to new architectures. The model must not rely on architecture-specific heuristics and must use general features.

Speed the model should return its prediction quickly.

Automated generation of ground truth data The ground truth for the training set must be reliable and automatically generated. The generation of the ground truth must not require the manual inspection and disassembly of the binaries.

Variable length instructions The model must be able to deal with the most difficult and general case: variable-length CPU instructions, mixed with data.

Context-sensitiveness To classify the bytes as code or data, their context (preceding and following bytes) must be considered, otherwise some ambiguities cannot be solved.

2.4.1 Challenges

(Clemens 2015) obtains the dataset used for the evaluation of the architecture identification method by extracting the code section from a set of executables. We want to assess whether this method will also work with real-world, “spurious” binaries, i.e., full executables containing not only executable instructions but also data, which is seen as noise by the classifier. We want to challenge the method on some more difficult sets of binaries, e.g., packed binaries or shellcodes.

We also want to add more complex features to the model, without resorting to byte n-grams which would produce too many features (256^n).

The main challenge associated with the code section identification problem is that we must design an integrated approach to segment a binary file and classify the segments at the same time. Indeed, the segmentation and the classification phases are inter-dependent and can not be performed sequentially. This means that the file fragment classification approaches we cited before (e.g., (Sportiello and Zanero 2012)) are not suitable for this task, because they assume that the segmentation of the file is given *a priori*.

The code discovery problem is challenging *per se* because it is known to be theoretically undecidable. In practice, however, there are approaches in literature that yield approximate, but acceptable, results.

Traditional models based on recursive disassembly require complex and advanced techniques to precisely retrieve the targets of indirect jumps. As shown in (Andriesse, X. Chen, et al. 2016), recursive disassemblers perform worse on this task than simple linear disassemblers. Also, they are slow and require that the binary is correctly loaded. The base address must be always provided, otherwise the disassembler will not resolve the jumps to absolute addresses.

Among the works we cited in the literature review, we are particularly inspired by the machine-learning based approaches like those presented in (Wartell et al. 2011), (Shin, Song, and Moazzezi 2015), (Karampatziakis 2010), and

(Bao et al. 2014). These approaches avoid the pitfalls of recursive disassemblers and do not need to recover the targets of indirect jumps; they do not require the knowledge of the entry point or of the base address). The only requirement is to know the ISA of the binary and to have a proper training set for it.

The main problems with the machine-learning approaches we cited before are the long training time and the memory requirements. Indeed, these methods must output a classification for each byte in the binary: for a 1 MB binary, this means that one million variables must be predicted. The problem becomes worse if we consider the feature space, which is often high-dimensional. Even considering only the bytes as features, the natural choice (Shin, Song, and Moazzezi 2015) is to generate 256 features by one-hot encoding. This means that for a 1 MB binary, we have a matrix with 256 million elements. If we use instruction-level features, the feature space becomes even larger: (Karampatziakis 2010) claims to have 2.3 million features; the feature selection phase in (Rosenblum et al. 2008) takes about two compute-days.

If the training phase takes too long, the analyst would need to either have all the necessary models computed beforehand, or to equip himself with a consistent amount of computing power. We seek to avoid either, since we want to be able to experiment with different training sets and we want our solution to be practical for binary analysis.

Sometimes (Bao et al. 2014), not only the training, but also the prediction phase can be slow, too. Instead, we want a system which can be used interactively: at least the prediction phase must run quickly.

Supervised machine-learning approaches carry the issue of the generation and the collection of ground truth data, i.e., the pre-labeled binaries whose bytes are labeled either as code or as data. In (Wartell et al. 2011), the binaries are pre-tagged with IDA Pro (which introduces mistakes, as any other disassembler does) and the evaluation of the approach was done by manually comparing the results of the experimental approach with the disassembly resulting by IDA Pro. This approach is tedious, error-prone, and allows to evaluate the method only on a limited number of binaries. We already discussed its limitations in subsection 2.3.4. In (Karampatziakis 2010), the ground truth is still obtained by using a disassembler (OllyDbg). In both cases, erroneous ground truth may lead to errors both in the training of the model, and in an overly optimistic evaluation of the performance of the model (e.g., when both the model and the ground truth make the same error, the result is evaluated as correct). (Andriesse, X. Chen, et al. 2016) and (Bao et al. 2014) extract the ground truth directly from the debugging symbols. This is the most precise approach, because the ground truth is provided by the compiler itself; however, executable files already compiled with complete debugging symbols are not easy to find (Andriesse, X. Chen, et al. 2016, p. 17), so it is necessary to generate the executables together with the debugging symbols by compiling from source. Parsing the debugging symbols presents technical difficulties when they are not stored in a standardized format: this is the case of PDB files, generated by Microsoft Visual Studio, which require a platform-specific API. (Křoustek et al. 2012)

Chapter 3

Approach

In this chapter, we will describe in detail the approaches we developed to solve the problems defined in chapter 2. First, we will give a broad overview, and then we will discuss each approach in detail.

3.1 Approach Overview

The **architecture classification problem** consists in predicting the CPU architecture and the endianness of an executable file without relying on any header information. To solve this problem, we follow an approach similar to (Clemens 2015), which faces the problem as a machine learning classification task. We assume not to have any metadata about the executables to classify.

We collect a training set of binaries ("samples") compiled for multiple architectures, from which we extract the byte-level features, i.e., the frequencies of occurrence of single bytes and of selected multi-byte patterns. We then train a logistic regression model to predict the ISA and the endianness of unseen binaries from these features.

The logical steps of the architecture identification approach, illustrated in Figure 3.1, are the following:

1. **Data collection**
2. **Feature engineering and preprocessing**
3. **Training**
4. **Prediction**

The **code section identification problem** consists in predicting whether each byte in an executable file belongs to an executable section or not. Our approach works as follows. We collect a training set of labeled binaries of the same architecture of the executables that we want to analyze; then we train a probabilistic sequence learning model (linear-chain CRF) on such training

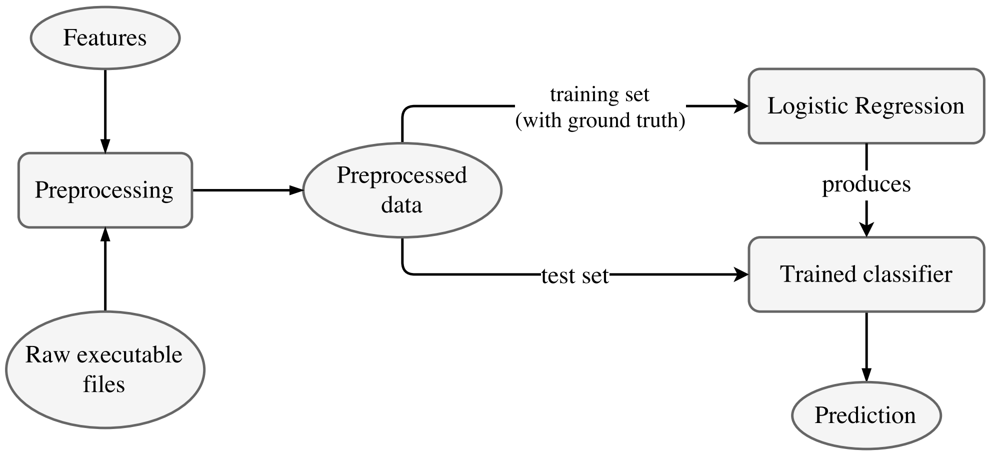


Figure 3.1: Diagram of the logical steps of the architecture classification method.

set. The trained model can identify the code-to-data and the data-to-code transitions in any unlabeled executable. The sequential learner classifies each byte as code or data: as we want to identify one single, contiguous, code section inside the binary file, we develop a postprocessing step to eliminate small blocks of code or data, by merging them with the surrounding region. As we will show in subsection 5.5.1, the postprocessing phase improves the performance of our model by eliminating the noise in the prediction output.

Figure 3.2a describes the data collection and preprocessing steps; Figure 3.2b describes the model learning, prediction and postprocessing steps.

The logical steps of the code section identification approach are the following:

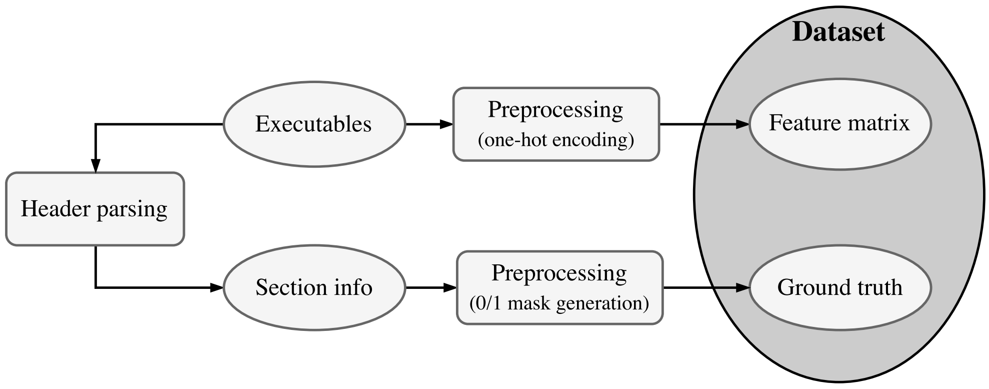
1. **Data collection**
2. **Preprocessing**
3. **Learning the model**
4. **Prediction**
5. **Postprocessing**

The **code discovery problem** consists in predicting whether each byte inside the code section of an executable file belongs to a valid CPU instruction, or should be considered inline data. We solve this problem with the same model used to solve the code section identification problem. First, we collect a training set of executables in which the code/data tagging is known, by compiling some programs with full debug information. We train a probabilistic sequence learning model (linear-chain CRF) on such training set. The trained model will predict, for an unknown binary, which bytes belong to a valid CPU instruction and which ones are to be considered as data.

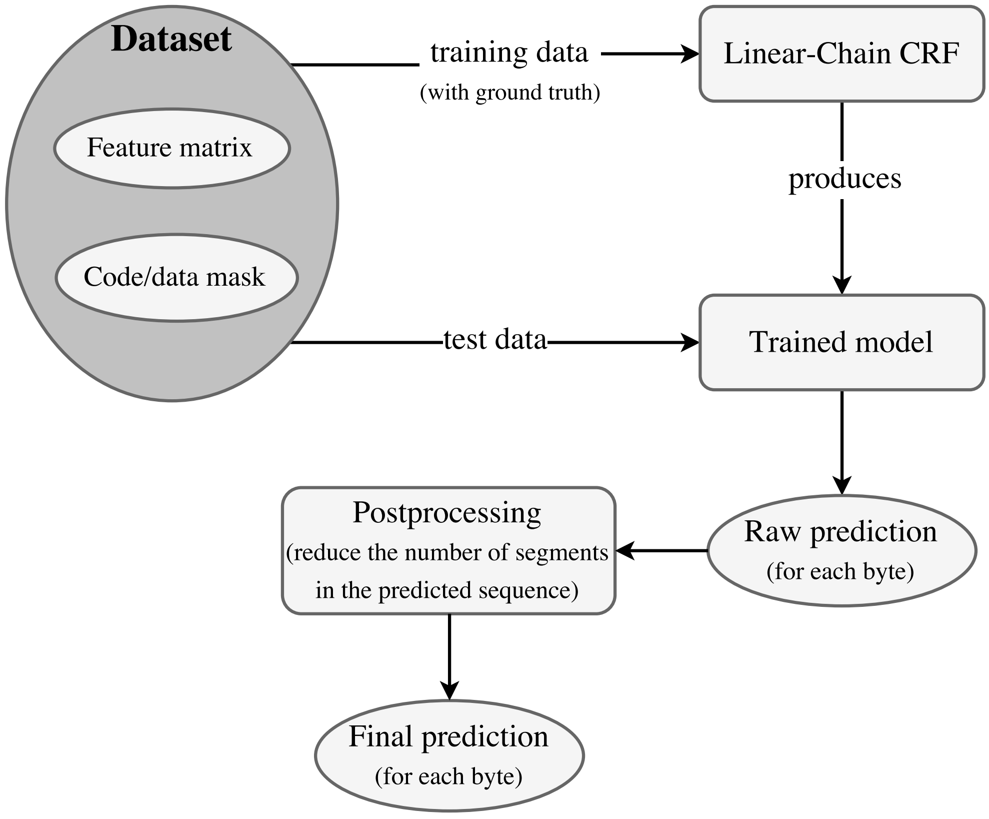
The logical steps of the code discovery approach are the following:

1. **Data collection and preprocessing**
2. **Training**
3. **Prediction**

Figure 3.3a illustrates the data collection and preprocessing phases; Figure 3.3b illustrates the model learning and prediction phases.

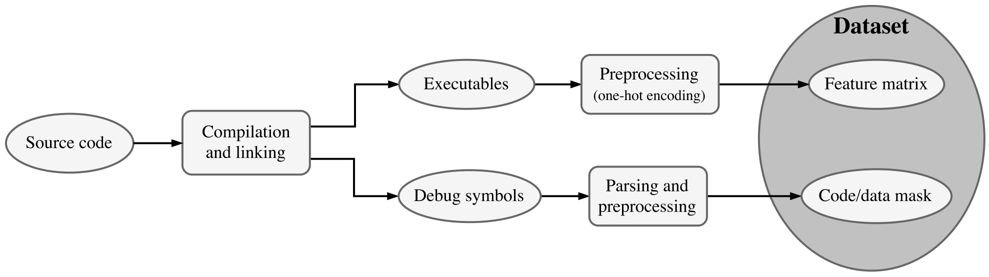


(a) Preprocessing steps to generate the training dataset.

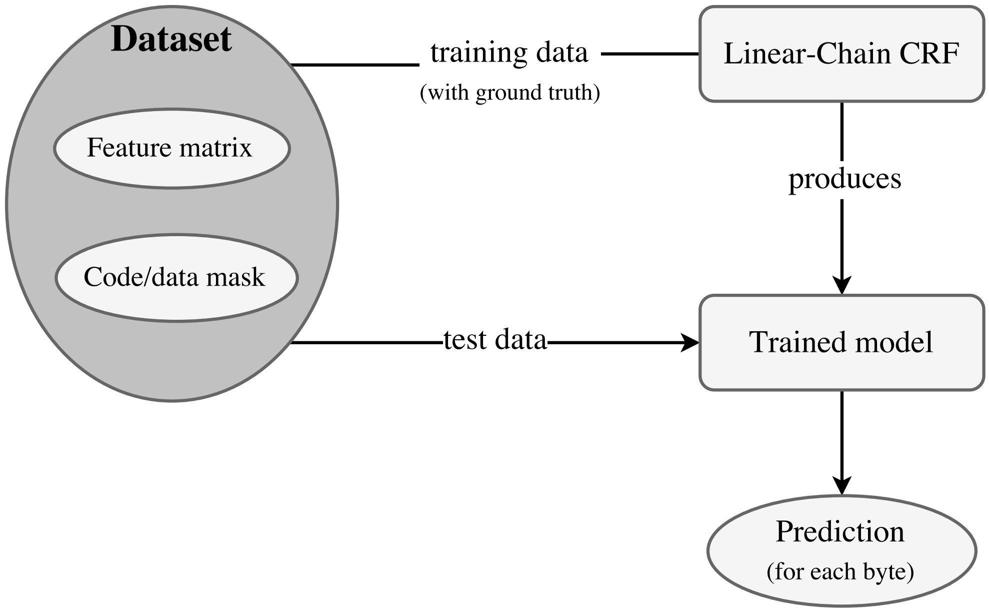


(b) Model learning and prediction phases.

Figure 3.2: Diagrams for the code section identification approach.



(a) Steps for the generation of the dataset.



(b) Model learning and prediction phases.

Figure 3.3: Diagrams for the code discovery approach.

3.2 Architecture classification

3.2.1 Preprocessing and feature engineering

We train our model on a set of executable files, labeled with the correct CPU architecture. The executables can be in any format (e.g., PE or ELF), as we do not rely on any header information.

Since we assume to have no metadata about the executable files (no disassembly is possible at this time), the only choice is to extract the features from the bytes in the files.

We obtain the *Byte Frequency Distribution* (BFD) of each file by computing the frequencies of the 256 possible bytes. The frequency of a byte having integer value i is defined as:

$$f_i = \frac{count(i)}{\sum_{i=0}^{255} count(i)} \qquad \forall i \in [0, 255] \tag{3.1}$$

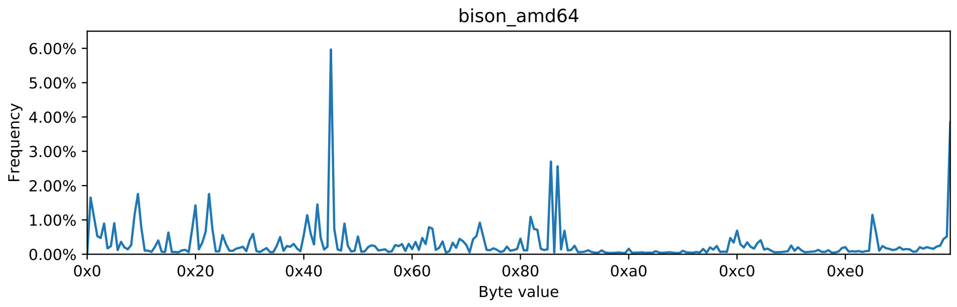
where $count(i)$ is the number of times that the byte i appears in the executable. We choose the BFD as the initial feature set for our model since it is a common and effective choice in literature (subsection 2.3.2). The underlying assumption which supports the choice of the BFD as a set of features is that executables compiled for different architectures have different BFDs.

In Figure 3.4, we show a graphical representation of the Byte Frequency Distribution computed for the same program, compiled for four CPU architectures. The differences between the three diagrams are evident, and can be spotted even by manual inspection. Two architectures (mips and mipsel) share the same BFD diagram, which we reported only once in Figure 3.4b. The reason these two architectures have an identical BFD is that they differ only by their endianness.

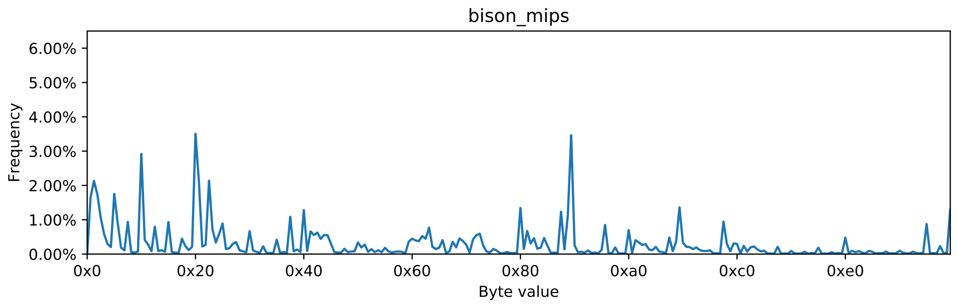
The second set of features is conceptually similar to the BFD: it consists in the frequencies of some specific byte patterns, which are more frequently found in some architectures than in others. These multi-byte patterns are encoded as regular expressions, which in our opinion represent a fair tradeoff between expressive power and matching speed.[1] The choice of the patterns to include in this set of features fell on the patterns of function prologues and epilogues, available in the angr's `archinfo` project (angr 2017).

We included in this set of features the four two-byte patterns used in (Clemens 2015), which help to recognize the endianness of the binary, i.e., the byte bigrams `0x0001`, `0x0100`, `0xfffe`, `0xfeff`. A model using only the Byte Frequency Distribution (i.e., the frequency of the single bytes) would be unable to distinguish two binaries of the same CPU architecture differing only by their endianness, because the two files would differ only in the ordering of the bytes, not in their frequency.

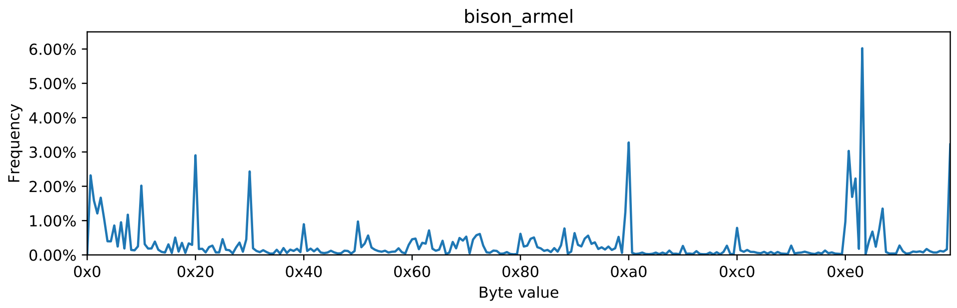
[1] Regular expressions can recognize regular languages, a task which requires linear time w.r.t. the length of the input string. (Reghizzi, Breveglieri, and Morzenti 2013, p. 91)



(a) bison (amd64)



(b) bison (mips, mipsel)



(c) bison (armel)

Figure 3.4: Byte Frequency Distributions of the `bison` executable compiled for four different architectures (amd64, mips, mipsel, armel). Note that the BFDs for the `mips` and `mipsel` architectures are identical, since they differ only for the endianness which is not reflected into the BFD.

The number of matches of these patterns are normalized by the length of the file (number of bytes), to account for executable files of different sizes. Thus, the multi-byte-pattern features are computed as follows:

$$freq(pattern) = \frac{matches(pattern, file)}{len(file)} \tag{3.2}$$

In the preprocessing phase, each file is transformed into a numeric vector containing the values of the features. The preprocessing phase can be parallelized over the files.

3.2.2 Training

In this section, we describe the classification model which learns the parameters of the model from the training set and, once trained, can predict the class (architecture) of any unknown sample.

We choose to use **Logistic Regression** (LR) with L1 regularization (lasso regularization), a linear classification model which learns a vector of parameters w by solving the following minimization problem over the feature matrix X and the vector of labels y:

$$\min_{w,c} \|w\|_1 + C \sum_{i=1}^{n} \log \left(\exp \left(-y_i \left(X_i^T w + c \right) \right) + 1 \right) \tag{3.3}$$

The cost function minimized by Equation 3.3 is composed of two terms: the first is the regularization term, which assigns a penalty to “complex” models with large coefficients; the second term is the logistic loss. The C parameter is the inverse of the regularization strength: higher values of C assign more importance to the second term in Equation 3.3, while lower values of C give more importance to the regularization term. Regularization is a technique used in machine learning to avoid *overfitting*, i.e., to learn models which are accurate on the training set but generalize poorly on unseen samples. We choose L1 regularization because Lasso can generate compact models, by setting to zero the coefficients in w corresponding to less relevant features. In this way, our model performs feature selection as part of the learning phase. For a more thorough explanation of the Logistic Regression model and of the usefulness of regularization in classification models, we refer to (Bishop 2006).

Logistic Regression, as described in Equation 3.3, is only able to discriminate between two classes. We obtain a multi-class classifier by using the *one-vs-the-rest* (OvR) strategy. The OvR strategy consists in fitting one classifier for each class $c \in C$, to distinguish the samples of c (“positive” class) against all the other classes $C \setminus \{c\}$ (considered together as the “negative” class).

Each Logistic Regression classifier outputs a *confidence score* for each sample, i.e., an estimate of the probability that the sample belongs to the positive class. The class label predicted for a sample is the class with the highest confidence score returned by the corresponding classifier.

We chose the Logistic Regression (LR) model for the following reasons:

- LR is a simple, linear model, less prone to overfitting than more complex models;
- the only hyperparameter to tune is the regularization strength;
- LR is a popular model for classification tasks and it is available as an off-the-shelf component from several machine learning libraries;
- training and prediction are fast;
- the model is compact: it can be represented with (num_features $\times$ num_classes) parameters;
- when classifying an unknown sample, LR not only returns the predicted class, but also a confidence score for each class;
- L1-regularization can produce a compact model, by putting to zero the weights of less important features;
- the model weights can give an estimate of the relative importance of the features;
- LR is one of the best-performing models in (Clemens 2015).

3.2.3 Prediction

After fitting the model with the training data, it can predict the CPU architecture of any unseen executable. The preprocessing phase for the binaries to analyze is the same used for the training set.

For any sample $\mathbf{x}$ to predict, the model returns, for each class C, a confidence score: this score can be interpreted as the probability $p(C|\mathbf{x})$ that the sample belongs to that class. The class predicted by the model is:

$$C^* = \arg\max_{C} p(C|\mathbf{x}) \tag{3.4}$$

3.3 Code section identification

3.3.1 Data collection

The training dataset consists in a collection of ELF, PE or Mach-O executable files, whose CPU architecture and location of the executable section are known. These files, according to their specifications, are divided in *sections*. Each section has a different purpose: there are sections containing executable code, static data, debug information and metadata, etc.

The executable file header contains information about the sections of the executable file: we are interested in the starting offset, the length, and the flag telling whether the section is executable or not.

3.3.2 Preprocessing

Since we assume that the binary files to analyze can contain both code and data, we can not disassemble the executable to generate instruction-level features. Instead, we derive the features directly from the byte sequence of the binary file.

For each byte in the file, we take the one-hot representation of its value (from 0 to 255) as a feature vector. For example, for the byte $b_i = 4$, the resulting 256-vector is: $x_i = (0, 0, 0, 0, 1, 0, 0, \ldots, 0)$. This choice derives from the observation that the byte value itself has little meaning for the purposes of our approach, and any ordering among byte values would be meaningless (Shin, Song, and Moazzezi 2015, p. 8), because we are only interested in distinguishing one byte from the others. In other words, the byte values are treated as *categorical features*. One-hot encoding is a typical approach to deal with categorical features in machine learning. One-hot encoding of the bytes turns each binary file in our dataset to a $(N \times 256)$ matrix of ones and zeros, where N is the number of bytes in the file.

We extend the preprocessing phase to also consider, for each byte, the values of the m preceding bytes (*lookbehind*) and of the n following bytes (*lookahead*). m and n are parameters of the model and can be optimized by hyperparameter tuning. These extended features are one-hot encoded as well, and appended to the vector corresponding to each byte. So, from an executable file of N bytes, we derive a matrix of $N \times (256 \cdot (n + m))$ elements.

The remaining part of the preprocessing phase consists in extracting the ground truth, i.e., the location information of the sections of the executable files. The offset and the size of each section containing executable code are found in the file header: from this information, we derive the ground truth in the form of a binary vector (“mask”), of the same length of the number of bytes in the executable file. In this vector, the ones correspond to the bytes belonging to executable sections, while the zeros correspond to the bytes belonging to non-executable sections.

For example, if we have a file of 16 bytes and the executable section spans from 6 to 13, the vector would be:

$$y = (0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0)$$

Executable files may have more than one executable section: the ones in the vector correspond to all the bytes belonging to any section which is marked as “executable” in the file header.

3.3.3 Learning the model

We translate the code section identification problem to a classification problem over a sequence of bytes.

The problem can be formalized as follows: for each training sample, we have a sequence of observed bytes ($\mathbf{X}$) and a sequence of labels ($\mathbf{Y}$):

$$\begin{aligned} \mathbf{X} &= (x_1, x_2, \ldots, x_n) \qquad & x_i &\in [0, 255] \\ \mathbf{Y} &= (y_1, y_2, \ldots, y_n) \qquad & y_i &\in \{0, 1\} \end{aligned} \tag{3.5}$$

We want to learn a model able to predict the label sequence $\mathbf{Y}$ corresponding to any unlabeled input sequence $\mathbf{X}$.

The model of choice is a linear-chain Conditional Random Field (Lafferty, McCallum, and Pereira 2001); the learning is performed by structural SVMs (Taskar, Guestrin, and Koller 2004). All the explanation below follows the steps of the two works cited before.

Conditional Random Fields (CRFs) are probabilistic models to segment and label data structured over a graph. In the simple case of linear-chain CRFs, the graph reduces to an undirected sequence. CRFs are a *structured learning* model, i.e., they do not simply classify each item (byte) in the sequence separately, but consider the structure of the problem. Classifying each byte separately would inevitably lead to a poor model, since in some architectures every byte can be interpreted as a part of a valid instruction, or as data: we must take the context into account.

In a structured learning setting, $\mathbf{X}$ is a set of random variables representing the sequence of observations, and $\mathbf{Y}$ is a set of random variables over the corresponding labels. We assume that some variables in $\mathbf{Y}$ are conditionally dependent: the dependencies are encoded in a graph in which the vertices are the elements of $\mathbf{Y}$ and an edge between two nodes $\mathbf{Y}_v$ and $\mathbf{Y}_w$ represent a conditional dependence between the two variables v and w.

CRFs can be seen as an extension of Hidden Markov Models, with a fundamental difference: while HMMs are generative models and model the joint probability $p(\mathbf{X}, \mathbf{Y})$, CRF do not model the distribution of $\mathbf{X}$ (which in general can be complex): instead, they model $p(\mathbf{X} \mid \mathbf{Y})$, i.e., the probability of $\mathbf{X}$ conditioned on $\mathbf{Y}$. This allows to relax the independence assumptions on the observations, which are a limitation of Hidden Markov Models.

We report the definition of a Conditional Random Field from (Lafferty, McCallum, and Pereira 2001, p. 5):

> Let $G = (V, E)$ be a graph such that $\mathbf{Y} = (\mathbf{Y}_v)_{v \in V}$, so that $\mathbf{Y}$ is indexed by the vertices of G.

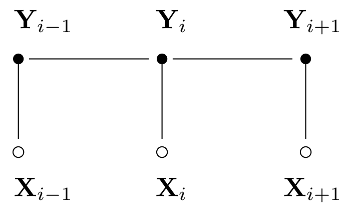


Figure 3.5: Graphical structure of a linear-chain CRF (Lafferty, McCallum, and Pereira 2001, p. 6).

> Then $(\mathbf{X}, \mathbf{Y})$ is a *conditional random field* in case, when conditioned on $\mathbf{X}$, the random variables $\mathbf{Y}_v$ obey the Markov property with respect to the graph: $p(\mathbf{Y}_v \mid \mathbf{X}, \mathbf{Y}_w, w \neq v) = p(\mathbf{Y}_v \mid \mathbf{X}, \mathbf{Y}_w, w \sim v)$, where $w \sim v$ means that w and v are neighbors in G.

Put in simpler words, a CRF is a graph of random variables; each variable is conditionally dependent on all its neighbors, and conditionally independent from all the other variables.

In the case of linear-chain CRFs (Figure 3.5), the graph structure is an undirected linear sequence, meaning that the variable associated with each element $(\mathbf{Y}_v)$ depends on the observations and on the classification of the previous $(\mathbf{Y}_{v-1})$ and the following $(\mathbf{Y}_{v+1})$ element. The dependencies between the variables in $\mathbf{Y}$ obey the Markov property on the graph, i.e., each variable is conditionally independent from all non-neighbor variables.

To compute the conditional probabilities in each node, we define a set of *feature functions*; they can be defined on the vertices, or on the edges:

$$\begin{aligned} f_h(\mathbf{X}, \mathbf{Y}) &= \sum_{i \in V} f_h(\mathbf{X}, \mathbf{Y}_i) \\ g_k(\mathbf{X}, \mathbf{Y}) &= \sum_{(i,j) \in E} g_k(\mathbf{X}, \mathbf{Y}_i, \mathbf{Y}_j) \end{aligned} \tag{3.6}$$

In our approach, the feature function associated to each vertex $\mathbf{Y}_i$ is the one-hot encoded byte value b_i; the feature function associated to each edge is always constant and equal to 1.

Then, we associate unary and binary *potential functions* ϕ respectively to each vertex i and to each edge (i, j) in G. The Markov network model is log-linear, so we can compute the network potentials as the exponential of the weighted sum of the features on the vertices and on the edges:

$$\begin{aligned}
\phi_i(\mathbf{X}, \mathbf{Y}_i) &= \exp\left[\sum_h w_h f_h(\mathbf{X}, \mathbf{Y}_i)\right] && \forall\, i \in V \\
\phi_{i,j}(\mathbf{X}, \mathbf{Y}_i, \mathbf{Y}_j) &= \exp\left[\sum_k w_k g_k(\mathbf{X}, \mathbf{Y}_i, \mathbf{Y}_j)\right] && \forall\, (i,j) \in E
\end{aligned} \tag{3.7}$$

where the weights w_i are the parameters learned by the model.

Finally, we compute the conditional probability distributions as:

$$P(\mathbf{Y} \mid \mathbf{X}) \propto \prod_{i \in V} \phi_i(\mathbf{X}, \mathbf{Y}_i) \prod_{(i,j) \in E} \phi_{i,j}(\mathbf{X}, \mathbf{Y}_i, \mathbf{Y}_j) \tag{3.8}$$

To learn the CRF model, we use Structural Support Vector Machines (SSVMs) (Taskar, Guestrin, and Koller 2004). SSVMs are soft-margin SVMs (Bishop 2006, p. 325) with a loss function designed for multi-label classification.

We report the original primal problem formulation for soft-margin SSVMs from (Taskar, Guestrin, and Koller 2004, p. 4).

$$\begin{aligned}
\min \quad & \frac{1}{2}\|\mathbf{w}\|^2 + C \sum_{\mathbf{x}} \xi_{\mathbf{x}} \\
\text{s.t.} \quad & \mathbf{w}^\top \Delta\mathbf{f}_{\mathbf{x}}(\mathbf{y}) \geq \Delta\mathbf{t}_{\mathbf{x}}(\mathbf{y}) - \xi_{\mathbf{x}} \qquad \forall\, \mathbf{x}, \mathbf{y}
\end{aligned} \tag{3.9}$$

where:

- $\mathbf{w}$ is the vector of weights learned by the model;
- $\mathbf{t}(\mathbf{x})$ is the predicted $\mathbf{y}$ for the input sequence $\mathbf{x}$;
- $\mathbf{f}(\mathbf{x}, \mathbf{y})$ are the *features* or *basis functions*;
- $\Delta\mathbf{f}_{\mathbf{x}}(\mathbf{y}) = \mathbf{f}(\mathbf{x}, \mathbf{t}(\mathbf{x})) - \mathbf{f}(\mathbf{x}, \mathbf{y})$;
- $\Delta\mathbf{t}_{\mathbf{x}}(\mathbf{y}) = \sum_{i=1}^{l} I(\mathbf{y}_i \neq (\mathbf{t}(\mathbf{x}))_i)$ is the number of wrong labels predicted by the model for the input $\mathbf{x}$;
- $\xi_{\mathbf{x}}$ is a slack variable to allow the violation of some constraints when the data is not linearly separable;
- C is the inverse of the regularization strength (the higher it is, the less the model is regularized).

To efficiently solve this optimization problem, we use the Block-Coordinate Frank-Wolfe algorithm for structural SVMs (Lacoste-Julien et al. 2013). The Frank-Wolfe algorithm is an iterative optimization algorithm. Asymptotically, it converges to the solution; it can be stopped after a fixed number of iterations, or when the loss function becomes smaller than a threshold.

This ends the description of the general sequence learning model.

In our case, we train a linear-chain CRF model by using the training data obtained from the preprocessing phase.

3.3.4 Prediction

The fitted model can now be used to detect the location of the code section inside unseen binaries. The features for the binary files to predict are generated with the same preprocessing steps used for the training set.

The output of the model is a prediction for each byte ("code" or "data"). More formally, the model outputs a binary vector of the same format of the ground truth data, i.e., a sequence of ones and zeros of the same length of the input file. The ones correspond to the bytes predicted to be in an executable section, while the zeros correspond to the bytes predicted to be in a non-executable section.

3.3.5 Postprocessing

The code section identification model includes a postprocessing phase to improve the results given by the sequential classification model. The objective of this phase is to ignore the small sequences of code inside the data sections (or vice versa), to return only few contiguous, large sequences of code or data.

The postprocessing phase is needed because, as explained in subsection 2.2.3, code sections may contain small pieces of data which are indistinguishable from the data found in data sections.

The postprocessing phase (Algorithm 1) iteratively removes the smallest contiguous sequence of predictions ("chunk") of code or data, merging it with the surrounding data or code (respectively).

The algorithm takes two parameters:

1. *MinSections*: the minimum number of chunks to keep;
2. *Cutoff*: the maximum size of the chunk that can be eliminated, as a fraction of the biggest chunk.

Algorithm 1 always terminates: at each iteration, either a chunk can be eliminated and the loop iterates again, or no chunk can be eliminated and the procedure returns.

Algorithm 1 Postprocessing algorithm

```
Require: $C$: list of chunks (start, end, tag), $MinSections$, $Cutoff$
  loop
    $M \leftarrow \max_{c \in C} \text{length}(c)$ {size of largest chunk}
    $c_{min} \leftarrow \arg\min_{c \in C} \text{length}(c)$ {smallest chunk}
    if $|C| > MinSections$ and $\text{length}(c_{min}) < Cutoff \cdot M$ then
      invert tag of $c_{min}$ and merge with surrounding chunks
      $C \leftarrow$ updated list of chunks
    else
      return $C$
    end if
  end loop
```

3.4 Code discovery

In this section, we explain our approach to solve the code discovery problem. We developed two different approaches for the architectures with variable-length instructions (x86, x86-64) and for an architecture with fixed-length instructions (ARM). The two approaches share the same sequential learning model; however, they use different features.

3.4.1 Preprocessing (Windows x86)

The generation of a good dataset to train a learner to distinguish code from inline data is not a trivial task. We already discussed the challenges that we have to face and the shortcomings of existing solutions in subsection 2.4.1.

The choice of the binary files to include in the training dataset is limited by some requirements:

1. the executable must have a non-negligible amount of inline data inside the code section;

2. the architecture must have variable-length instructions;

3. ground truth information (i.e., whether each byte is code or data) must be available;

4. ground truth information must be precise; for this reason, we do not want to rely on any disassembler.

To our knowledge (Andriesse, X. Chen, et al. 2016), the only compiler that inserts inline data in the code section of executables is Microsoft Visual Studio; this satisfies the first requirement. To satisfy the second requirement, we consider the x86 and x86-64 architectures, which have variable-length instructions. To satisfy the last two requirements, we choose to use the debugging symbols generated by the compiler to extract the ground truth.

Debugging symbols are generated by the compiler when the developer chooses to build a program with the debug configuration. Debugging symbols help developers to debug their programs, by providing a wealth of information about the structures found in the executable files: functions, variables, line number information, exported symbols, data types, and information about the original source files the program was compiled from. For instance, debug symbols allow to match machine instructions with source lines in the original source code.

Debugging symbols can be found in different formats, depending on the architecture, on the executable format and on the compiler support. (Křoustek et al. 2012) describes which structures can be found in symbol files, and the technical difficulties involved in parsing them.

The symbols in ELF executables are usually stored in the **DWARF** format, which is well-documented (Eager and Consulting 2012) and supported by static analysis tools and disassemblers. DWARF symbols can be embedded in specific sections of the ELF, or they may be distributed in separate files.

On Windows platforms, the executables are shipped in the **PE (Portable Executable)** format, and the compiler (Visual Studio) exports the debug symbols into separate files in the **PDB (Program Database)** format, which is a proprietary format developed by Microsoft. PDB is not standardized ("source code is the ultimate documentation" (Microsoft 2016)) and all the tools needing the information contained in PDBs must rely on Microsoft's APIs, because the PDB file format is subject to change and there is no published standard to rely on. The LLVM Project published a partial documentation about the information contained in PDB files (*The PDB File Format* 2017).

Microsoft provides PDB files for the Windows system executables to help developers debug their code. The idea of including Windows system executables in the training set is attractive; however, the PDB files for the Microsoft binaries are *stripped* (Microsoft 2017[a]) and do not contain *private symbols*. This means that the information about the functions in the binary code is not available and that we can not extract the ground truth. Indeed, the availability of private symbols would simplify the reverse-engineering of Microsoft's software, which is understandably not in the best interests of the company. The impossibility of extracting ground truth from Windows binaries is also reported by (Bao et al. 2014).

Since we need full debug symbols (including function information) for each binary to extract the ground truth, and those symbols are not generally shipped with the executables, we have to compile some programs from source with Microsoft Visual Studio, enabling the generation of complete symbol files.

dia2dump is a tool provided by Microsoft (Microsoft 2017[b]) whose purpose is to parse a PDB file and to extract a textual representation of it. By using dia2dump on the PDB files and by parsing, in turn, the output of dia2dump, we extract the starting and ending offset of each function in the executables, together with the locations of inline data.

We developed Algorithm 2, to tag each byte inside the code section of an executable as code or data starting from the information extracted from the PDB file.

Algorithm 2 Code/data tagging algorithm

```
$tag[i] \leftarrow DATA \quad \forall i$
for every function $f$ do
    $tag[f.begin : f.end] \leftarrow CODE$
    for $d \in f.data$ do
        $tag[d.addr : f.end] \leftarrow DATA$
    end for
end for
for each thunk $t$ do
    $tag[t.begin : t.end] \leftarrow CODE$
end for
return $tag$
```

The rationale for tagging all bytes as data by default is that the information contained in the PDB file does not cover all the bytes in the code section. Indeed, between functions, there is a variable amount of padding bytes (0xCC), not reported in the PDB file.

The rationale for the data tagging step is that the PDB file reports the starting address of each data block, but not its length, which is variable. By manual analysis of the binary files using IDA Pro, we found out that static data inside the functions is always located at the end of the function.

As suggested in (Bao et al. 2014), *thunks* (short callbacks to addresses in the code) are tagged as code.

By manual inspection, we found out that in each binary there are some functions for which there is no information in the PDB file. They are repetitive and short blocks of code; they appear to be exception handlers and polymorphic call handlers. We suspect that the reason there is no function information for these blocks of binary code is that they do not correspond to any function in the original source code. This code is always located in a specific sub-section of the .text section, identified as .text$x in the PDB file. Since we were unable to find further information about these code blocks, there is no ground truth for them, and they are isolated in a known sub-section of the file, we excluded these blocks from the training set. This choice should not lead to any overestimation of the performance of our method, because the excluded code is very repetitive and would be easy to spot in an automated way.

Eventually, we obtain the training set, which consists in two files for each sample:

1. `program.bin`: the sequence of byte we want to analyze (i.e., the contents of the `.text` section of the executable, after excluding the `.text$x` subsection);

2. `program.mask`: a sequence of zeros and ones, of the same length of the `.bin` files, which tells whether each byte is code or data.

3.4.2 Preprocessing (ARM)

ARM is a RISC architecture with fixed-length instructions. This means that every instruction and every data block starts at an address which is a multiple of 4 bytes. The code discovery problem for architectures with fixed instruction length is simpler, because the instructions and the data are already segmented in fixed-size blocks.

The problem of code discovery for ARM executables can be stated as follows: to tell whether each 4-byte word in the executable section is an ARM machine code instruction, or data.

The ground truth for the training of the model is extracted from ARM executables compiled with debugging symbols. Indeed, ARM debugging symbols contain mapping information which tells where code and data are located inside the executable sections (*ELF for the ARM® Architecture* 2015, p. 24). We do not consider the case in which Thumb code (which is 2-byte aligned) is also present; that problem is addressed in (J.-Y. Chen et al. 2013).

To obtain the features from the binaries, we strip them (i.e., remove the symbols) and pass them to a linear disassembler. The linear disassembler, if no mapping symbols are present, tries to interpret each word as an instruction and returns an opcode, or outputs the string `INVALID` if the 4 bytes can not be decoded into a valid ARM instruction.

We take as features, for each 4-byte word, **the opcode returned from the disassembler** and **the first byte of the word**. These two features are one-hot encoded before being passed to the model.

The ground truth is simply obtained by passing the binaries, complete with the mapping symbols, to the linear disassembler. If the binaries are not stripped, the linear disassembler can use the symbols to identify inline data, returning a perfect disassembly of the executable file (Andriesse 2016). The disassembler outputs the string `.word` whenever it encounters a word corresponding to data: from there, the ground truth is easily obtained.

Figure 3.6 shows an example of the differences in the output of `objdump` (the linear disassembler in the GNU binutils library) when the debugging symbols are missing.

```
9214: add r3, pc, r3
9218: ldr r2, [r3, r2]
921c: cmp r2, #0
9220: bxeq lr
9224: b 9050
9228: .word 0x0000f1a8
922c: .word 0x00000118
```

(a) With symbols

```
9214: add r3, pc, r3
9218: ldr r2, [r3, r2]
921c: cmp r2, #0
9220: bxeq lr
9224: b 9050
9228: andeq pc, r0, r8, lsr #3
922c: andeq r0, r0, r8, lsl r1
```

(b) Without symbols

Figure 3.6: Disassembly of a portion of an ARM executable, with and without the debugging symbols. In Figure 3.6a, the disassembler correctly identifies the inline data because the debugging symbols are present. In Figure 3.6b, the symbols are not present and the disassembler interprets the last two data words as instructions.

3.4.3 Training and prediction

Starting from the features described in the previous sections, we generate the lookahead and the lookbehind features as explained in subsection 3.3.2.

We train a linear-chain Conditional Random Field (CRF) (Lafferty, McCallum, and Pereira 2001), which we described above in subsection 3.3.3 for the code section identification phase.

The executables to analyze are preprocessed in the same way of the training data (with the obvious exception of the ground truth extraction phase). The trained model reads, as an input, the sequence of items (bytes or words) to classify as code or data; it gives back a sequence of code/data labels, one for each item (byte or word).

Differently from the code section identification approach, we do not apply the postprocessing phase since we are also interested in very small subsequences of code and data.

Chapter 4

Implementation

In this chapter, we will explain how we implemented the approaches described in chapter 3 into concrete software tools. In the first section, we will list all the tools that we used; then, we will describe in detail the implementation of our methods.

4.1 Tools

We used the following software tools and libraries to implement our methods:

- Python 3.6.1 was used as the main programming language to write all our models; the following packages were also used:
  - Scikit-learn 0.19.0 for the machine learning tasks (except the CRFs), feature transformation and model evaluation;
  - NumPy 1.13.1 for efficient operations with vectors
  - SciPy 0.19.1 for the sparse matrix operations;
  - filebytes 0.9.13 for reading ELF, PE and Mach-O headers;
  - python-magic 0.4.13 to determine the type of files;
  - jupyter 1.0.0 for interactive data visualization;
  - matplotlib 2.0.2 for graph plotting;
  - pandas 0.20.3 to preprocess, load and store data;
  - pystruct 0.2.4 for the machine learning tasks involving Conditional Random Fields (CRFs).
- bash 3.2.57 for some of the data collection and preprocessing tasks;
- binutils 2.28 to extract information from ELF files and manipulate them;
- UPX 3.94 to pack executable files;

- macOS 10.12.5 as the primary OS to run all the experiments;
- VirtualBox 5.1.22 as the Virtual Machine Manager to run other operating systems as needed;
- Windows 10 (Evaluation version), inside a VirtualBox virtual machine, to compile the Windows binaries and to extract information from the debugging symbols;
  - Visual Studio 2017 was used to build the Windows programs;
  - PowerShell was used to automate the build process and the extraction of data from the symbol files;
- Debian Linux 8 (“jessie”) was used inside a VirtualBox VM to download the Debian binaries to populate our dataset.

4.2 Architecture classifier

In this section, we will explain how we implemented the architecture classifier described in section 3.2.

4.2.1 Dataset collection

In this section, we will describe the tools that we developed to collect the datasets to train and test our CPU architecture classifier. The composition and the characteristics of each dataset are detailed in section 5.2.

The datasets come from different sources:

- package repositories of Linux distributions;
- datasets from research papers;
- manually collected executables;
- datasets derived from the transformation of other datasets.

We developed two Bash scripts to automate the download and the unpacking of the software archives from the Debian Linux distribution. For each architecture, and for each package, our script attempts to locate and download the corresponding archive. If this step is successful, the script extracts the archive and saves all the executable files in a separate location; all the other files are discarded.

We wrote two Bash scripts to obtain new datasets by transforming existing ones:

- a script to extract the code section from a dataset containing full binary files;

- a script to pack all the executables in a dataset with the UPX packer.

We wrote a Bash script calling `arduino-builder`[1] to compile the Arduino samples (Arduino 2017) to acquire more samples for the `avr` architecture.

4.2.2 Preprocessing

In this section, we will explain how we implemented the feature extraction phase to prepare the data for our classifier.

We implemented the preprocessing step in a method of the Python class of the learner model. The preprocessing step extracts the following features from a sample:

1. the frequency of each byte;
2. the frequency of each multi-byte pattern;
3. the label (i.e., the true architecture of the binary).

The multi-byte patterns are encoded into regular expressions. Regular expression matching is done by the Python Standard Library `re` module. To improve the performance, the regular expressions are pre-compiled beforehand using the `re.compile()` method.

To process an entire dataset, we developed a Python script (`dataset-_loader.py`) which spawns a pool of workers in multiple processes by using the `multiprocessing` module of the Python Standard Library. The number of workers is a parameter of the script. The workers execute the preprocessing step on the files in parallel, using multiple CPU cores.

The features extracted from the executable files are stored into a CSV file (one row per sample). CSV is not the most compact format for storing this kind of information; however, we choose to use it because it is a *de facto* standard: data in the CSV format can be inspected, loaded and converted by a multitude of software tools.

The preprocessing step is done once, before the model training phase: it would be too slow to recompute all the features on the fly whenever the model needs to be trained. Also, once the preprocessing phase is completed, we do not need to have the original binaries at hand to train the classifier; only the file with the features needs to be distributed.

4.2.3 Model training

In this section, we will describe the implementation of the classification model.

We choose to use the Python machine learning framework Scikit-learn to implement our model. Scikit-learn is a well-known, established machine

[1]`https://github.com/arduino/arduino-builder`

learning framework which offers many off-the-shelf supervised and unsupervised machine learning methods, as well as model evaluation facilities (cross validation, holdout, random sampling, computation of common performance metrics) and preprocessing functions. It also supports, for some of the models, parallel execution on multiple CPU cores. Scikit-learn relies on NumPy for numeric computation. NumPy is a Python library, implemented in C, which offers fast, vectorized operations on matrices and vectors.

We implement our classification model in a class (`BinaryClassifier`), written in Python, which internally uses the Scikit-learn's `LogisticRegression` classifier. By implementing a wrapper class for the learner, we isolate all the code related to the model in the same class and hide all the unnecessary complexity of the Scikit-learn `LogisticRegression` class, which is a general-purpose model and exposes a multitude of options and parameters. The `LogisticRegression` already supports one-vs-the-rest models, so we do not have to implement it from scratch.

We follow the Scikit-learn classifier interface (scikit-learn 2017) for our class to be able to use the model evaluation features of Scikit-learn. In particular, we implement the following methods in our learner class:

- `fit(X, y)`: fit the model on the features in X and on the true labels in y;
- `predict(X)`: returns the predicted class for each of the samples represented by the features in X;
- `predict_proba(X)`: similar to `predict(X)`, but instead of returning a single prediction for each sample, it returns a vector containing the probabilities for that sample to belong to each class.

Our wrapper class also implements the methods to preprocess a single sample, and to save and load the trained model.

Our learner is initialized with the following parameters:

- *penalty*: whether to use L1 (lasso) or L2 (ridge) regularization for the Logistic Regression;
- C: the inverse of the regularization strength (if C is smaller, more regularization is applied);
- *num_jobs*: how many parallel jobs to use for model training.

To determine the optimal value for the regularization strength parameter, we use the `GridSearchCV` class of Scikit-learn. `GridSearchCV` performs an exhaustive search in the parameter space, evaluating the model (with cross-validation) on each combination of the parameter values. The user specifies the domain of each parameter.

In our case, we only optimize for the regularization strength (C), in the experiment described in subsection 5.7.1.

4.2.4 Model evaluation

We wrote a Python script to evaluate the model on each dataset, and to store into a JSON file the performance measures, the information about the dataset and the parameters of the evaluated model. We compute the cross-validation prediction for each sample by using the `cross_val_predict` function in Scikit-learn.

We describe our evaluation methodology in greater detail in section 5.3.

4.3 General sequential learning model

In this section, we explain how we developed a sequential learning classifier to implement the code section identification and the code discovery methods.

Both the problems of code section identification and code discovery can be stated as sequence learning and prediction problems; the only things that change are: how an element in the sequence is defined, and which features are used to describe each element.

The classifier we implemented is based on linear-chain Conditional Random Fields, described in section 3.3. We decided to use the models provided by `pystruct` (Müller and Behnke 2014), an open-source and ready-to-use structured learning library implementing CRF-like models with Structural SVMs learners (SSVMs). We chose this framework because of the quality of the documentation, the simplicity of use, and the good classification performance.

We wrap the general-purpose CRF model of `pystruct` into a Python class (`CRFModel`) to hide the complexity of the general-purpose models in `pystruct`, to expose a clear and simple API to the end user, and to ensure the compatibility with the Scikit-learn APIs. Our class wraps the `ChainCRF` and the `FrankWolfeSSVM` classes of `pystruct`, which provide the core learning algorithms; in addition, we implement our postprocessing algorithm (described in subsection 3.3.5).

Table 4.1 shows the format of the input data of our model. Each sample (i.e., sequence) i is represented by a feature matrix (X_i) containing the one-hot encoded features for each element of the sequence, as described in our approach (subsection 3.3.2); the ground truth for a training sample (containing a label for each element in the sequence) is represented by the ground truth vector y_i.

For performance reasons (explained in subsection 4.4.1) our model needs to work with sparse feature matrices. The current version of `pystruct` does not support sparse matrices as an input for its models; however, it is easy

		$\mathbf{y_i}$	$\mathbf{X_i}$					
			0x01	0x02	0x03	$\cdots$	0xFE	0xFF
Address	0x0001	0	0	1	0	$\cdots$	0	0
	0x0002	0	1	0	0	$\cdots$	0	0
	0x0003	1	0	0	1	$\cdots$	0	0
	0x0004	1	0	0	0	$\cdots$	0	1
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
	0xFFFE	1	0	1	0	$\cdots$	0	0
	0xFFFF	0	0	0	0	$\cdots$	1	0

Table 4.1: Exemplification of the input format for our sequential learning model. X_i is the feature matrix for the sample i (a binary file, i.e., a sequence of bytes to classify), containing the one-hot encoded features (columns) for each byte (row) to classify; y_i is the ground truth vector for the sample i, containing one label for each element.

to overcome this problem by applying a simple patch in the code that has already been proposed as a pull request[2] on GitHub.

The interface of `CRFModel` (our class) follows the Scikit-learn classification API and can be used with the model evaluation functions. `CRFModel` implements the following methods:

- `fit(X, y)`: fit the model on the list of feature matrices X and the list of ground truth vectors (true labels) y;
- `predict(X)`: return, for each of the samples in X, the vector of predictions;
- `score(X, y)`: called on a fitted model, computes the average accuracy for the samples;
- `save(filename)`: saves the trained model into a file;
- `load(filename)`: loads the trained model from a file.

Our classifier takes the following initialization parameters:

- `C`: the regularization strength as defined by the `FrankWolfeSSVM` learner of `pystruct`;
- `max_iter`: the maximum number of iterations of the learner;
- `lookahead` and `lookbehind`: how many lookahead or lookbehind bytes to include in the feature matrix.

[2]`https://github.com/pystruct/pystruct/pull/4`

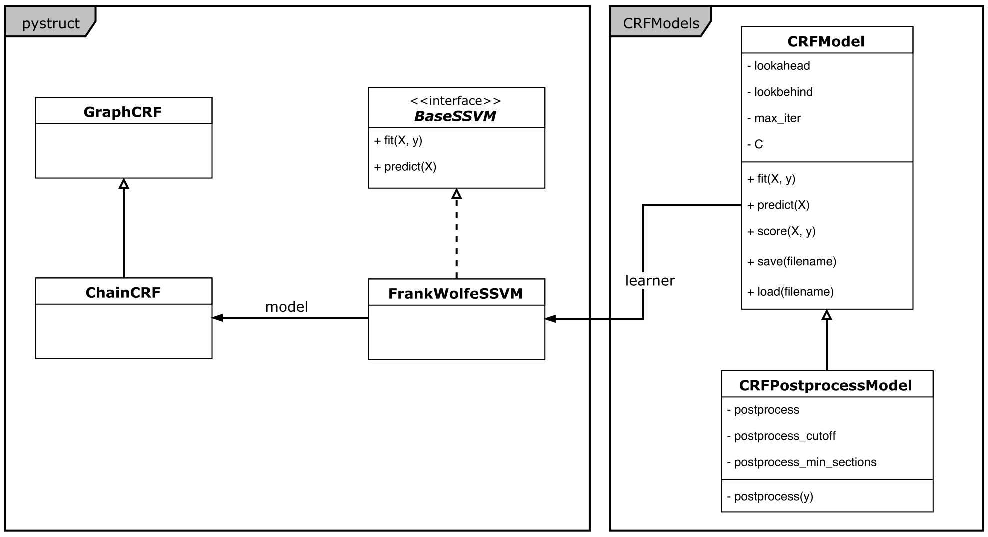


Figure 4.1: Class diagram of our CRF models and of the pystruct API.

Our learner automatically generates the lookahead and the lookbehind features (subsection 3.3.2) in the `fit()` and `predict()` methods. The lookahead and lookbehind features for all the elements in the sequence are generated by vertically shifting a copy of the feature matrix and joining it (horizontally) with the original one. All these operations are performed efficiently, without leaving the sparse matrix format.

Our code section identification method also includes the postprocessing phase (subsection 3.3.5), to be applied to the results of the prediction of the CRF model. We implemented the postprocessing algorithm by subclassing our `CRFModel` and overriding the `predict` method. The subclass (`CRFPostprocessModel`) takes three additional initialization parameters:

1. `post_process`: whether to activate the postprocessing step or not;

2. `postprocessing_cutoff` and `postprocessing_min_sections`: the parameters to be given to Algorithm 1;

The class diagram of our model is shown in Figure 4.1.

4.4 Code section identification

In this section, we will explain how we implemented the code section identification method described in section 3.3. The core learning algorithm is the same described before, in section 4.3.

4.4.1 Preprocessing

In this subsection, we will explain how we extract the features from the binary files in the dataset.

The initial dataset for the code section identification task consists in a collection of ELF, PE or Mach-O executables, all belonging to the same CPU architecture. As explained in subsection 2.1.1, each executable file is divided into "sections," containing either executable code or data; the sections are listed in the file header. The header of the executable file also specifies, for each section, whether it contains executable code or not.

We recall that the purpose of this analysis is to identify the boundaries of the executable sections. We see the binary file as a sequence of bytes, and we want to classify each byte either as belonging to an executable section, or to a non-executable section. We associate a set of *features* to each byte of the executable file. As described in the approach (section 3.3), for each byte we use its one-hot encoded value (0–255) as a feature.

By one-hot encoding the value of each byte, for a binary file of N bytes we produce a feature matrix of $(N \times 256)$ elements. Each element of the feature matrix is an unsigned 8-bit integer, which is the smallest data type supported by NumPy. A representation encoding each element of the matrix in a single bit would be 8 times more space-efficient, but we did not find a practical way to implement it within the NumPy stack. This representation of the features introduces a performance problem: if we had 100 executables of size 1 MB each in our dataset, we would be using 25.6 GB of main memory for the feature matrices!

These feature matrices are, by their nature, *sparse*: since the bytes are one-hot encoded, each row of the matrix contains a single "one" and 254 zeroes. Thus, we can save a significant amount of memory by using a compressed representation of the feature matrix. We use the **CSR (Compressed Sparse Row)** sparse matrix representation (Buluç et al. 2009), which is natively implemented by SciPy. To keep the memory usage as low as possible at all times and to avoid spikes in the memory usage, we generate the feature matrix from the original file directly in the CSR format, without passing from the "dense," standard matrix representation.

More formally, SciPy's CSR matrix creation function requires three vectors:

1. *data*: $data(k)$ is the value of the k-th non-zero element in the matrix, in row-major order;
2. *rows*: $rows(k)$ is the row coordinate of the k-th non-zero element;
3. *cols*: $cols(k)$ is the column coordinate of the k-th non-zero element.

The relationship between the above vectors and the original matrix A is the following:

$$A(i,j) = \begin{cases} data(k) & \text{if } \exists\, k \text{ s.t. } i = rows(k) \wedge j = cols(k) \\ 0 & \text{elsewhere} \end{cases} \tag{4.1}$$

We initialize the CSR matrix by specifying the three vectors as follows: the *data* vector is filled with ones; the *rows* vector is the sequence of integer numbers ranging from 1 to N (the number of bytes); the *cols* vector is the list of the integer values v_i of the bytes in the file ($v_i \in [0, 255]$).

In the preprocessing phase, we do not generate the *lookahead* or *lookbehind* features: those are generated internally by the common sequential learning models described in section 4.3.

Finally, we need to extract the ground truth from the executable files in the training set: this is easy because the ELF, PE and Mach-O headers provide the information about which sections of the binary file are executable. First, we use the Python module `python-magic` which uses the standard UNIX interface (the so-called "magic file"[3]) to determine the file type of the sample (ELF, PE or Mach-O). To parse the header information, we use `filebytes` (Schirra 2017), which is a library written in Python capable of parsing the ELF, PE, and Mach-O headers. The ground truth generation method tags as code ("1") all the bytes belonging to any executable section which (according to the file header), and tags as data ("0") all the remaining bytes in the file. The final result is a binary vector of the same length of the executable file.

To summarize, the final output of the preprocessing phase, for each sample i of length N bytes, is:

- the $(N \times 256)$ feature matrix X_i, containing the one-encoded byte values of the file;
- the ground truth vector y_i, of length N, telling whether each byte belongs to an executable section or not.

4.4.2 Model training

For the code section identification problem, we train the `CRFPostprocessModel` illustrated in Figure 4.1 and described in section 4.3. The model is fitted with the data generated from the preprocessing phase; the executables to analyze follow the same pipeline.

To determine the optimal values for the lookahead and lookbehind lengths, we use the `GridSearchCV` class of Scikit-learn to perform an exhaustive search over the parameter space.

The number of iterations of the Frank-Wolfe block-coordinate SSVM learner also needs to be tuned. In general, more iterations lead to a better accuracy. We manually tuned the maximum number of iterations to obtain a fair tradeoff between the training time and the accuracy of the model.

[3]`https://linux.die.net/man/5/magic`

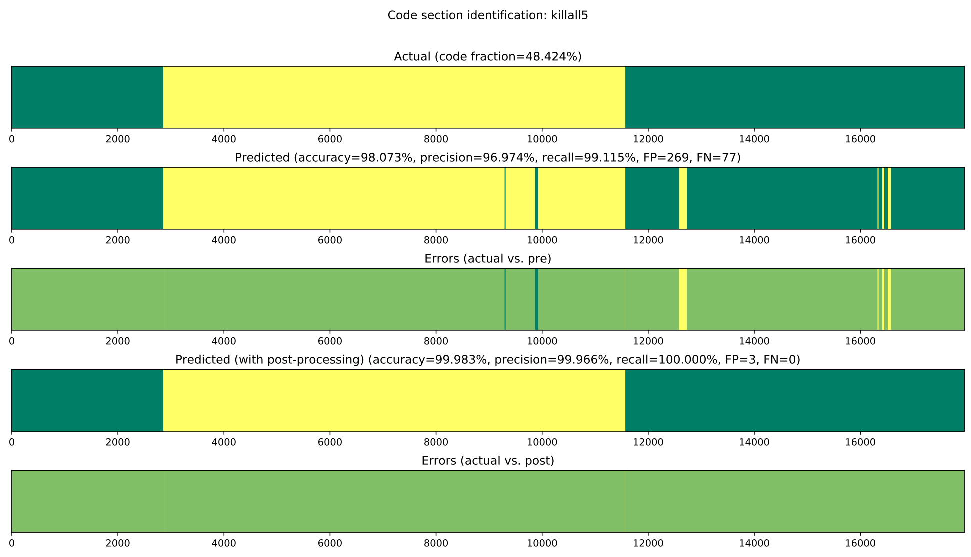


Figure 4.2: Visualization of the results of the code section identification method on a binary file. From top to bottom: the ground truth, the prediction of our model before postprocessing, the prediction errors, the prediction of our model after postprocessing, and the prediction errors after the postprocessing. Precision and recall are computed on the bytes, considering "code" as the positive class.

4.4.3 Model evaluation

To visualize the results of our method, we developed a Jupyter notebook which uses `matplotlib` to compare the ground truth data and the prediction output of our model, by means of a "barcode plot" (Figure 4.2). We mark with two different colors the bytes corresponding to code and data, and we provide the results before and after the postprocessing phase.

We developed a script to evaluate the code section identification model over multiple datasets and save the results in the JSON format. The script uses the model evaluation functions in Scikit-learn. The evaluation methodology is described in detail in section 5.3.

4.5 Code discovery

The purpose of the code discovery method is to predict, within the code section of an executable file, which bytes correspond to valid CPU instructions and which bytes are inline data inserted by the compiler between CPU instructions. In this section, we will explain how we implemented the code discovery approach described in section 3.4.

4.5.1 Preprocessing (Windows x86)

In this section, we will explain how we implemented the automated collection of a dataset for the code discovery method, how we extract the ground truth for the debugging symbols, and how we generate the features for our model.

Dataset collection

The choice of a dataset for our supervised code discovery model is constrained by the need for precise ground truth, i.e., the knowledge about which bytes correspond to valid, executable machine code and which others do not. Specifically, we want to extract such ground truth from the symbol files generated during the compilation process. So, we have to compile from source some Windows programs, and to configure the compiler to generate the symbol files. To do this, we need a set of programs to compile and a configured compilation environment.

Microsoft makes available to developers the *Universal Windows Platform (UWP) app samples* (Microsoft 2017[d]), a set of code samples demonstrating the usage of the Windows APIs. These samples are shipped as ready-to-use Visual Studio projects ("solutions"), so they are a convenient choice for our dataset.

Microsoft makes available to developers a Windows 10 Virtual Machine, with Visual Studio 2017 already pre-installed (Microsoft 2017[c]). We start from this system to configure our compilation environment. MSBuild (Microsoft 2017[e]) — the official Microsoft command-line build automation tool — allows to compile a Visual Studio project with the correct options without interaction, i.e., without having to manually open the project in Visual Studio.

We were not able to automate the build of the C# projects with MSBuild, so we resorted to compile all the C++ projects available in the samples. C++ compilation requires the installation of additional packages in Visual Studio: this is easily done by opening a C++ project in Visual Studio and following the instructions.

We faced another problem with MSBuild: even by specifying the option `/p:DebugType=full`, the resulting debug symbol file lacked the private symbols, i.e., the function information which provides the ground truth. From a verbose run of MSBuild, we identified the problem: MSBuild does not pass the correct flag to the linker. As a workaround, we replaced all the occurrences of `DebugFastLink` with `DebugFull` in the file:

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\
    Community\Common7\IDE\VC\VCTargets\Microsoft.Link.
    Common.props
```

which is one of the configuration files of the linker. In this way, the linker always generates full (public and private) debug symbols.

We wrote a simple PowerShell script to iterate the execution of MSBuild over all the C++ code samples. PowerShell is a shell scripting language designed by Microsoft to provide a more powerful interface than the traditional DOS prompt language. After figuring out the correct combination of options to provide to MSBuild, we compiled the samples for both the x86 and the x86-64 architectures.

After the compilation process, we ran a script to collect all the `.exe`, `.sys` and `.dll` files together with their associated symbol files (PDB files), and move them to a separate location.

Features and ground truth extraction

To parse the PDB files, we used the `dia2dump` (Microsoft 2017[b]) tool by Microsoft. `dia2dump` uses Microsoft's APIs to parse a PDB file and dumps all the information into a plain text file. The sources of `dia2dump` (C++) need to be compiled in Visual Studio, for the x86-64 architecture, with the "Debug" configuration. We discovered that compiling it with the "Release" configuration causes the program to crash at runtime. To run `dia2dump` without errors, we also had to register a DLL provided by Microsoft in the DIA SDK, by using the command:

```
\windows\syswow64\regsvr32.exe "C:\Program Files (x86
    )\Microsoft Visual Studio\2017\Community\DIA SDK\
    bin\amd64\msdia140.dll"
```

After completing these steps, the `dia2dump` executable is ready to run. We developed another PowerShell script to automate the execution of `dia2dump` on all the PDB files in our dataset and save the output into plain text files.

Finally, we convert the textual representation of each PDB file into the final ground truth format, i.e., a binary vector telling, for each byte in the executable file, whether it corresponds to a valid machine instruction or to data. We implemented this step in a Python program, following Algorithm 2 in section 3.4.

We extract the byte-level features from the executable files exactly as we did for the code section identification method (subsection 4.4.1). For each executable file, we only generate the features and the ground truth relative to the code section, discarding all the other sections of the file.

4.5.2 Preprocessing (ARM)

In this section, we will explain how we obtained the training set for an architecture with fixed instruction length (ARM), and how we extracted the features and the ground truth from the executable files.

We gathered the coreutils compiled for ARM[4] with debugging symbols. The debugging symbols are embedded into each ELF executable, in the standard DWARF format (Eager and Consulting 2012).

We use the `objdump` linear disassembler from the GNU binutils library,[5] which is able to yield a perfect disassembly of these executables if the DWARF symbols are present.

To generate the features (subsection 3.4.2) from each file in the training set, we first strip the debugging symbols with the `strip` utility from the binutils, and then we run `objdump` on the stripped file. In this way, our model learns on the same features `objdump` would generate for the executable files to analyze, which miss the debugging symbols.

To extract the ground truth, we run `objdump` on the original ELF files, without stripping the DWARF symbols: in this way, the disassembler knows for certain which 4-byte words are instructions and which words are data. For the latter, the disassembler outputs the string `.word` in place of the opcode.

We implemented the ground truth and feature extraction phases in a Python class, which calls `objdump` on each file and parses the disassembly output.

4.5.3 Model training and prediction

The model training and prediction phases for the code discovery method are similar to the analogous phases for the code section identification method (subsection 4.4.2).

We train the `CRFModel` class (Figure 4.1), which implements the CRF sequential learning model described in section 4.3. Unlike the code section identification approach, this model does not perform any postprocessing on the prediction outputs.

Since we implemented the preprocessing and the learning phases in separate classes, we can use the same learner for both the Windows and the ARM datasets, even if the features used for classification are different.

To determine the optimal values for the lookahead and lookbehind lengths, we use the `GridSearchCV` class of Scikit-learn to perform an exhaustive search over the parameter space.

[4]`https://github.com/BinaryAnalysisPlatform/arm-binaries`
[5]`https://sourceware.org/binutils/docs/binutils/objdump.html`

Chapter 5

Experimental Validation

In this chapter, we will state the goals of our experiments, describe our datasets, explain the evaluation methodology and show the results of our experiments.

5.1 Goals

In this section, we will enumerate all the goals of our experiments.

For the experiments about the **architecture classifier**, we set the following goals:

1. improve the performance of the classifier described in (Clemens 2015);
2. try the model on "complete" binaries containing both code and data (noise), rather than pure machine code, easier to classify;
3. prove that simple features, directly extracted from the bytes, are sufficient to reliably identify the architecture of an executable file;
4. show that our method also works on more challenging datasets, i.e., packed executables, shellcodes and short file fragments;
5. find the optimal value for the regularization parameter (C).

For the experiments about the **code section identification** method, we set the following goals:

1. prove that our method can correctly segment ELF, PE and Mach-O files into executable and non-executable sections;
2. show that the postprocessing algorithm to eliminate noise (Algorithm 1) improves the quality of the predictions;

3. show that our method does not require an excessive amount of resources, i.e., memory and computation time;

4. find the optimal lookahead and lookbehind lengths.

For the experiments about the **code discovery** method, we set the following goals:

1. show that a machine learning approach based on simple features can yield performances which are comparable or superior to the methods based on recursive disassembly;

2. confirm the hypothesis that the sequence of bytes in the instruction flow of an executable program follows a probabilistic *language model*, which can be learned;

3. show that our method does not require an excessive amount of resources, i.e., memory and computation time;

4. show that our method works with architectures having both fixed-length and variable-length instructions;

5. find the optimal lookahead and lookbehind lengths.

5.2 Datasets

In this section, we will describe the origin, the composition and the characteristics of each dataset that we will use in the experiments.

5.2.1 Clemens' dataset

The author of (Clemens 2015) kindly sent to us the dataset of binaries used in the paper, consisting in 16,642 executable files from 20 different architectures. The files in this dataset only contain executable code: the author generated them by extracting the code section from the full binaries.

Of these 16,642 files, 3 files are empty and 622 files appear more than one time, for a total of 1,557 copies. We removed the duplicate and the empty files from the dataset. The number of samples by class in the dataset is shown in Table 5.1, in the first column.

The AVR (Arduino) and CUDA architectures have a small number of samples, which may need to be increased if the performance turns out to be poor for those two classes.

Architecture	**Original dataset**	**Extended dataset**
alpha	1295	1295
amd64	897	897
arm64	1074	1074
armel	903	903
armhf	904	904
avr	292	**365**
cLEMENCy	0	**20**
cuda	20	**133**
hppa	472	472
i386	901	901
ia64	590	590
m68k	1089	1089
mips	903	903
mipsel	903	903
powerpc	900	900
ppc64	766	766
s390	603	603
s390x	604	604
sh4	775	775
sparc	495	495
sparc64	698	698
Total	15084	15290

Table 5.1: Composition of the original Clemens' dataset and of the extended multi-architecture dataset.

5.2.2 Extended multi-architecture dataset

We built a new dataset by adding more executables for the AVR, cLEMENCy and CUDA architectures to the dataset from (Clemens 2015) (subsection 5.2.1).

We compiled the Arduino built-in examples (Arduino 2017) for AVR, and the NVIDIA CUDA 8.0 samples (NVIDIA 2017). Both compilation processes gave a set of ELF files. Following the procedure described in (Clemens 2015), we extracted the code section (`.text`) from each ELF file and added it to the dataset.

cLEMENCy (*The cLEMENCy Architecture* 2017) is a 9-bit middle-endian architecture introduced during the DEF CON 25 (2017) conference. cLEMENCy is a challenging architecture since the 9-bit bytes could break our model, which is based on the frequency of (8-bit) bytes. We took 20 cLEMENCy binaries from the DEFCON challenges and added them to our dataset.

The composition of the updated dataset is shown in the second column of Table 5.1.

Architecture	**Samples**
amd64	386
arm64	382
armel	385
armhf	385
i386	386
mips	384
mipsel	384
ppc64el	380
Total	3072

Table 5.2: Composition of the Debian dataset.

5.2.3 Debian dataset

We collected a set of ELF binaries, for 8 different architectures, from the Debian package repository. To collect the dataset, we wrote a Bash script executing the following steps:

1. randomly choose 300 packages from the full list of Debian packages;[1]
2. for each package, download the corresponding archive for each architecture;
3. unpack the `.deb` archive and extract all the ELF files;
4. insert the ELFs into the dataset, and discard all the other files.

We executed the script inside a Debian virtual machine to take advantage of the package managing tools (`apt` and `dpkg`). Not all the packages were available for all the architectures, hence the number of binaries for each architecture is not the identical. The composition of the Debian dataset is shown in Table 5.2.

5.2.4 Packed Debian dataset

This dataset is derived from the Debian dataset described in subsection 5.2.3.

We ran the UPX packer (Oberhumer, Molnár, and Reiser 2004) on all the binaries in the Debian dataset, with the default configuration. For some binaries, the packing failed, so we discarded them. We removed the `arm64` architecture because packing failed for most of the binaries.

The composition of the resulting dataset is shown in Table 5.3.

[1]`https://packages.debian.org/stable/allpackages`

Architecture	**Samples**
amd64	277
armel	237
armhf	192
i386	249
mips	255
mipsel	257
ppc64el	278
Total	1745

Table 5.3: Composition of the packed Debian dataset.

5.2.5 Shellcodes

Shellcodes are small pieces of executable code which are used to exploit a vulnerable program. When an attacker is able to gain control of the execution of a program, e.g., by exploiting a buffer overflow vulnerability, they will try to inject a machine code payload into the executable memory area of the program and to redirect the execution of the program to such payload. This payload is called *shellcode*, because one of the typical tasks performed by such code is to open a shell with the same privileges of the vulnerable program. Shellcodes, however, may perform other operations: open a socket, launch another program, execute system calls, read and write in memory, etc.

Shellcodes are typically very small in size (tens or hundreds of bytes). In fact, smaller shellcodes are more practical to use from the point of view of the attacker, because in some scenarios the shellcode must fit into a buffer of limited size for the attack to succeed. Shellcodes consist in handwritten machine code; shellcode authors employ a number of tricks to keep the shellcodes as short as possible and to escape the input validation routines of the vulnerable program (e.g., sometimes it is necessary to write a shellcode without the `0x00` byte, the C string terminator). Shellcodes may exhibit "strange" byte patterns which are different from the ones which can be found in the previously described datasets. The small size of the shellcodes is also a challenge for our method, since the Byte Frequency Distribution becomes noisy on short byte sequences.

We collected a dataset of shellcodes for 6 different architectures, from the websites shell-storm.org[2] and Exploit Database[3]. The main problem with this dataset is the small number of samples per architecture. Indeed, it is difficult to collect a large number of non-repetitive shellcodes, because the number of tasks performed in practice by real-world shellcodes is limited. Also, most shellcodes are written for the x86/x86-64 architectures.

[2] `http://shell-storm.org/shellcode/`

[3] `https://www.exploit-db.com/shellcode/`

Architecture	**Samples**
amd64	20
arm	21
i386	19
mipsel	10
ppc	9
sparc	10
Total	89

Table 5.4: Composition of the shellcode dataset.

Architecture	**Samples**
x64	1097
x86	1100
Total	2197

Table 5.5: Composition of the ByteWeight dataset.

The composition of the dataset is shown in Table 5.4.

5.2.6 ByteWeight dataset

The authors of (Bao et al. 2014) made their dataset of executable files publicly available.[4]

The dataset contains Linux ELF binaries compiled from GNU coreutils, binutils and findutils with a combination of the following:

1. compilers: GNU GCC and Intel ICC;
2. levels of optimization: O0, O1, O2, O3;
3. architectures: x86 and x86-64.

The dataset also contains a smaller number of Windows PE executables compiled for the same two architectures, with four levels of optimization, with the Visual Studio compiler. The composition of the dataset is shown in Table 5.5.

This dataset contains binaries from two similar architectures (x86 and x86-64), which may pose a challenge for the architecture classifier; also, the executables are obtained by a variety of compilers, operating systems, and optimization levels.

[4]`http://security.ece.cmu.edu/byteweight/`

Architecture	**Samples**
x64	151
x86	141
Total	292

Table 5.6: Composition of the Windows dataset.

5.2.7 Mach-O dataset

We collected 165 binaries, in the Mach-O executable format, from the `/bin` and `/usr/bin` system directories of an Apple macOS 10.12 installation. These binaries are provided by Apple as part of the operating system, which is based on the Mach kernel.

5.2.8 Arduino (AVR) dataset

We built a dataset of 73 binaries for the AVR architecture by compiling the Arduino samples (Arduino 2017) for the Arduino UNO[5] hardware.

We automated the build with a shell script; the resulting executables are in the ELF format.

5.2.9 Windows dataset

To evaluate our code discovery method, we needed a dataset of binaries with inline data in the code section, belonging to an architecture with variable-length instructions. So, we compiled the *Universal Windows Platform (UWP) app samples* (Microsoft 2017[d]) for the x86 and x86-64 architectures, with full debugging symbols. In section 4.5, we explain in detail the compilation process, and the preprocessing steps to extract the features and the ground truth. Table 5.6 shows the composition of the dataset.

5.2.10 ARM coreutils dataset

To test our code discovery method on an architecture with fixed instruction length, we obtained the binaries of the GNU coreutils[6] compiled for ARM with full debugging symbols and four different levels of optimization: `-O0`, `-O1`, `-O2`, `-O3`.

This dataset consists in 103 binaries for each level of optimization, for a total of 412 executable files.

The code section of the binaries, on average, contains more than 90% of code. By manually inspecting the binaries (using both `objdump` and `radare2`),

[5]`http://www.arduino.org/products/boards/arduino-uno`

[6]Source: `https://github.com/BinaryAnalysisPlatform/arm-binaries`

we found that the majority of data in these ARM binaries consists in jump tables, as we expected after reading (Andriesse 2016).

5.3 Experimental Setup

In this section, we will explain the evaluation methodology we followed for our experiments.

All the experiments were run on a laptop with a 2.9 GHz Intel Core i7 CPU (4 virtual cores) and 8 GB of RAM, running macOS 10.12.6.

5.3.1 Architecture classifier

In the machine learning domain, the simplest way to evaluate the performance of a supervised machine learning algorithm is *holdout testing*, i.e., splitting the dataset in two parts: a training set and a test set. The model is fitted on the training set and evaluated on the test set. The main disadvantage of holdout testing is that the test set can not be used for the training of the model: so, the quality of the fitted model may decrease. For this reason, the test set can not be too large; but a reduction of the size of the test set leads to a greater uncertainty of the performance estimate.

A smarter way to use the data for model evaluation is *k-fold cross-validation*. k-fold cross-validation is a model evaluation technique consisting in randomly splitting the dataset into k non-overlapping folds. For k times we train the model on $k-1$ folds and evaluate it on the remaining fold.

For the architecture classifier, we evaluate the predictions given by **5-fold cross-validation** on the dataset, i.e., for each sample, we take the prediction returned when it was in the test fold. In this way, we can evaluate the model on the whole dataset, avoiding the issues of holdout testing on small datasets. Once the predictions for all the samples are obtained, we compute the performance metrics for each class and for the whole dataset.

We consider the following metrics, expressed in terms of True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN):

1. **Accuracy** is the fraction of correctly predicted samples:

$$accuracy = \frac{1}{N}\sum_{i=1}^{N} I(\hat{y}_i = y_i) = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.1}$$

2. **Precision** for a class $c \in C$ is defined as the fraction of the samples that truly belong to class c among those that are predicted to belong to class c by the model:

$$precision(c) = \frac{TP(c)}{TP(c) + FP(c)} \tag{5.2}$$

3. **Recall** for a class $c \in C$ is defined as the fraction of the samples that are predicted to belong to class c among those truly belonging to class c:
$$recall(c) = \frac{TP(c)}{TP(c) + FN(c)} \tag{5.3}$$

4. **F-measure** for a class $c \in C$ is the harmonic mean of precision and recall:
$$F_1(c) = \frac{2 \cdot precision(c) \cdot recall(c)}{precision + recall} = \frac{2TP(c)}{2TP(c) + FN(c) + FP(c)} \tag{5.4}$$

To report meaningful "global" values for these metrics, we choose to macro-average them over the classes. By averaging over the classes (instead that over the samples), we give to each class the same importance. Indeed, we do not want to under-estimate the errors on the classes with fewer samples.

For example, macro-averaged precision is defined as follows:
$$precision_{macro} = \frac{1}{|C|} \sum_{c \in C} precision(c) \tag{5.5}$$

5.3.2 Code section identification and code discovery

We evaluate the code section identification method and the code discovery method by computing the fraction of correctly classified bytes with relation to the ground truth.

Unless specified, we kept the regularization strength parameter C of our CRF model at the default value of 1; the choice of this parameter does not seem to influence the prediction if we stop the model after a fixed number of iterations (and not at the convergence point).

The reported metrics (accuracy, precision, recall, F-measure) are computed for each sample over its bytes, and then averaged among the samples to obtain global figures. We decided to average the metrics over the binary files with the same weight in order to give to each executable file the same importance. For precision and recall, we assume that the positive class is "code" and that the negative class is "data".

We also report, for each dataset, the percentage of bytes which are labeled as code in the ground truth. This number is useful to interpret the accuracy metric (e.g., if the code percentage was 99%, it would be trivial to obtain 99% accuracy with a model that always predicts "code," and that score would have little meaning).

Figure 5.1 visualizes the ground truth and the prediction output of our code discovery method applied to a single executable file. Performance metrics are also reported.

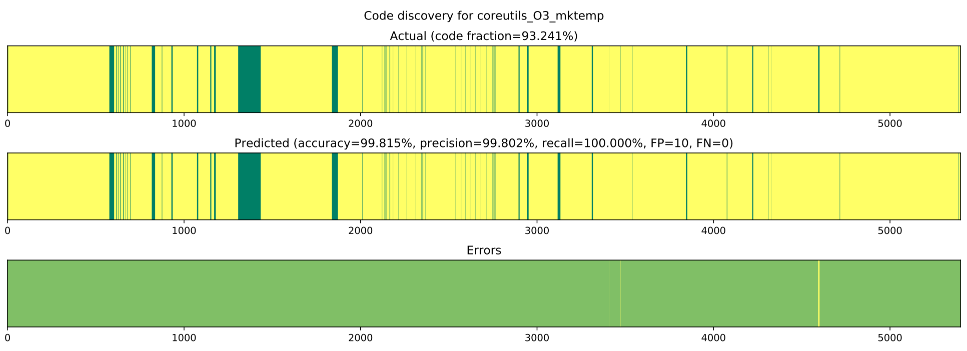


Figure 5.1: Visualization of the code discovery model applied on an ARM binary.

5.3.3 Hyperparameter tuning

Most supervised learning models require the analyst to specify some *hyperparameters*. These hyperparameters are not the same of the *parameters of the model* learned from the training data: they need to be specified *a priori*.

In our case, the parameters we have to choose are the regularization strength (C) for the architecture classifier and the lookahead/lookbehind lengths for the code section identification and the code discovery methods.

In some cases, the hyperparameters of a model do not have any intuitive meaning and cannot be derived from the domain knowledge. A widely employed approach to determine the optimal value for these hyperparameters is to select the values maximizing a certain performance measure of the model (e.g., accuracy).

We select the values of our hyperparameters by performing a *grid search*. The grid search algorithm trains and evaluates (possibly, in parallel) the model with all the possible combinations of the hyperparameters. The analyst has to provide the domain of the values of each hyperparameter. The grid search algorithm estimates (via cross-validation testing) the performance of the model for each choice of the parameters.

The values of the hyperparameters associated with the highest performance are chosen. The model is finally re-trained over all the training samples with these optimal hyperparameters, and tested again on a test set which was never used for hyperparameter tuning, to avoid overfitting and get an unbiased performance estimate.

5.4 Experiments: Architecture classifier

In this section, we will describe the experiments we performed on the architecture classification method, described in section 3.2 and in section 4.2.

5.4.1 Clemens' dataset, basic features

The purpose of this experiment is to replicate the results in (Clemens 2015) with a model which is as similar as possible to the one proposed in the original paper. We ran the classifier on the original dataset (Table 5.1), considering only the original features, namely: the Byte Frequency Distribution and the 4 bi-grams for the endianness detection. We disabled the extended features, i.e., the regular expressions matching the function prologues and epilogues.

Table 5.7 shows the performances computed via 5-fold cross-validation over the entire dataset; we also report the performance of the Logistic Regression model in (Clemens 2015). The global accuracy obtained by our model is 99.8%, higher than the accuracy reported in (Clemens 2015) for the Logistic Regression model (97.94%), and also higher than SVMs, the best-performing model in the paper (98.35%). The F-measures for the single classes are similar to the ones reported in the paper, except for MIPS and MIPSEL, which perform better (we obtained 99.06%, 98.95% vs. 88.4%, 88.6%). For the CUDA architecture, the original paper reports an accuracy of 51.6%: we got a better figure of 95%, but we think that any comparison on this architecture makes little sense because of the very small number of samples.

The original paper uses the `SimpleLogistic` implementation by Weka,[7] which does not perform any regularization; all the parameters are set to their default values and no hyperparameter tuning is attempted. As explained in section 3.2, we employed L1 regularization to avoid overfitting with the Logistic Regression model: this may explain the higher performance achieved on the same dataset. See subsection 5.7.1 for the details about the tuning of the regularization parameter.

5.4.2 Clemens' dataset, extended features

This experiment uses the same dataset and the same model as the previous one; the only difference is that we added the extended features, i.e., the regexes matching the prologues and the epilogues of the functions in the binary code. The goal of this experiment is to determine whether the addition of those features improves the performance of the model.

The results (Table 5.8) show that the extended features do not improve significantly the performance of the classifier, which already performs well without them. The global F-measure is 99.36% vs. 99.18% of the model without extended features.

[7]`http://weka.sourceforge.net/doc.dev/weka/classifiers/functions/SimpleLogistic.html`

Class	**Support**	**Precision**	**Recall**	**AUC**	**F1**	**F1 (Clemens)**
alpha	1295	0.9977	0.9992	0.9995	0.9985	0.997
amd64	897	1.0	0.9967	0.9983	0.9983	0.990
arm64	1074	0.9981	0.9991	0.9995	0.9986	0.994
armel	903	1.0	1.0	1.0	1.0	0.998
armhf	904	0.9978	0.9989	0.9994	0.9983	0.996
avr	292	1.0	0.9623	0.9812	0.9808	0.936
cuda	20	0.9	0.9	0.9499	0.9	0.516
hppa	472	1.0	1.0	1.0	1.0	0.993
i386	901	1.0	1.0	1.0	1.0	0.998
ia64	590	1.0	1.0	1.0	1.0	0.995
m68k	1089	0.9982	0.9991	0.9995	0.9986	0.993
mips	903	0.99	0.9911	0.9953	0.9906	0.884
mipsel	903	0.9879	0.9911	0.9952	0.9895	0.886
powerpc	900	0.9989	0.9989	0.9994	0.9989	0.989
ppc64	766	0.9961	1.0	0.9999	0.998	0.996
s390	603	0.9983	0.9983	0.9991	0.9983	0.998
s390x	604	1.0	0.9983	0.9992	0.9992	0.998
sh4	775	0.9949	0.9987	0.9992	0.9968	0.993
sparc	495	0.9939	0.9939	0.9969	0.9939	0.988
sparc64	698	0.9986	0.9986	0.9992	0.9986	0.993
Total	15084	0.9925	0.9912	0.9955	0.9918	0.9566

Table 5.7: Performance of the architecture classifier on the dataset of (Clemens 2015), without the extended features. In the last column, we report the per-class F-measure of the Logistic Regression model in (Clemens 2015).

5.4.3 Extended multi-architecture dataset

In this experiment, we evaluate the classifier on the extended multi-architecture dataset (subsection 5.2.2), with all the features enabled.

The results (Table 5.9) show that the cLEMENCy architecture performs worse than the other classes, but still obtains a F-measure of 92%. This reduced performance could be explained by the very low number of samples (20). The Byte Frequency Distribution of cLEMENCy binaries does not have any apparent meaning, since the bytes are aligned on a 9-bit basis: however, our classifier is still able to correctly classify the majority of the samples.

The addition of further CUDA samples improved the performance on that class, confirming our hypothesis that the lower performance was simply due to the lower number of samples.

Class	**Support**	**Precision**	**Recall**	**AUC**	**F1**
alpha	1295	0.9992	0.9992	0.9996	0.9992
amd64	897	0.9989	0.9978	0.9988	0.9983
arm64	1074	0.9991	0.9991	0.9995	0.9991
armel	903	1.0	1.0	1.0	1.0
armhf	904	0.9989	0.9989	0.9994	0.9989
avr	292	1.0	0.9623	0.9812	0.9808
cuda	20	0.9474	0.9	0.95	0.9231
hppa	472	1.0	1.0	1.0	1.0
i386	901	1.0	1.0	1.0	1.0
ia64	590	1.0	1.0	1.0	1.0
m68k	1089	0.9991	0.9991	0.9995	0.9991
mips	903	0.9978	0.9989	0.9994	0.9983
mipsel	903	0.9956	0.9978	0.9988	0.9967
powerpc	900	0.9978	0.9989	0.9994	0.9983
ppc64	766	0.9987	1.0	1.0	0.9993
s390	603	0.9967	0.9983	0.9991	0.9975
s390x	604	1.0	0.9983	0.9992	0.9992
sh4	775	0.9949	0.9987	0.9992	0.9968
sparc	495	0.986	0.996	0.9977	0.991
sparc64	698	0.9971	0.9971	0.9985	0.9971
Total	15084	0.9954	0.992	0.996	0.9936

Table 5.8: Performance of the architecture classifier on the dataset of (Clemens 2015), with the extended features.

Execution time

For this dataset, we measured the execution time of the preprocessing and of the cross-validation phases, which were run in parallel on 4 cores on our system:

- the preprocessing script (one-time execution) processed 784 MB of executable files in 112 minutes, resulting in an average throughput of 0.12 MB/s;
- the 5-fold cross-validation took 30.6 s.

5.4.4 Debian dataset

In this experiment, we tested the model on the Debian dataset (subsection 5.2.3). This dataset includes full ELF files, not only their code section isolated from all the other contents of the binary file. The classification of full

Class	**Support**	**Precision**	**Recall**	**AUC**	**F1**
alpha	1295	0.9985	0.9992	0.9995	0.9988
amd64	897	1.0	0.9978	0.9989	0.9989
arm64	1074	0.9972	0.9991	0.9994	0.9981
armel	903	1.0	1.0	1.0	1.0
armhf	904	0.9967	0.9989	0.9993	0.9978
avr	365	1.0	0.9726	0.9863	0.9861
clemency	20	0.9048	0.95	0.9749	0.9268
cuda	133	0.9773	0.9699	0.9849	0.9736
hppa	472	1.0	0.9979	0.9989	0.9989
i386	901	1.0	1.0	1.0	1.0
ia64	590	1.0	1.0	1.0	1.0
m68k	1089	1.0	0.9991	0.9995	0.9995
mips	903	0.9989	0.9989	0.9994	0.9989
mipsel	903	0.9945	0.9989	0.9993	0.9967
powerpc	900	1.0	0.9989	0.9994	0.9994
ppc64	766	0.9974	1.0	0.9999	0.9987
s390	603	1.0	0.9983	0.9992	0.9992
s390x	604	1.0	1.0	1.0	1.0
sh4	775	0.9949	0.9987	0.9992	0.9968
sparc	495	0.992	0.996	0.9978	0.994
sparc64	698	0.9971	0.9971	0.9985	0.9971
Total	15290	0.9928	0.9939	0.9969	0.9933

Table 5.9: Performance of the architecture classifier on the extended multi-architecture dataset.

binaries is more challenging because the data contained in the non-executable sections is noise and may confuse the classifier.

The results of the experiment (Table 5.10) show that our classifier achieves high performances even when dealing with binaries containing both code and data sections. There are no significant differences in performance among the classes.

5.4.5 Packed Debian dataset

In this experiment, we test our classifier on the dataset described in subsection 5.2.4, which is composed by the binaries in the Debian dataset, packed with UPX. Packed code is indistinguishable from random data; however, packed binaries include a "stub," i.e., a small unencrypted routine to load and decrypt the payload containing the rest of the program.

The results — reported in Table 5.11 — show that our model achieves good performances (F-measure = 99%) even on this dataset of packed binaries,

Class	**Support**	**Precision**	**Recall**	**AUC**	**F1**
amd64	386	0.9922	0.9922	0.9956	0.9922
arm64	382	1.0	0.9974	0.9987	0.9987
armel	385	0.9948	0.9974	0.9983	0.9961
armhf	385	0.9974	0.9974	0.9985	0.9974
i386	386	0.9948	0.9948	0.997	0.9948
mips	384	1.0	1.0	1.0	1.0
mipsel	384	0.9974	0.9948	0.9972	0.9961
ppc64el	380	0.9974	1.0	0.9998	0.9987
Total	3072	0.9968	0.9968	0.9981	0.9968

Table 5.10: Performance of the architecture classifier on the Debian dataset.

Class	**Support**	**Precision**	**Recall**	**AUC**	**F1**
amd64	277	0.9964	0.9928	0.996	0.9946
armel	237	0.9958	0.9916	0.9954	0.9937
armhf	192	0.9844	0.9844	0.9912	0.9844
i386	249	0.9762	0.988	0.992	0.982
mips	255	0.9961	0.9922	0.9957	0.9941
mipsel	257	0.9961	0.9883	0.9938	0.9922
ppc64el	278	0.9857	0.9928	0.995	0.9892
Total	1745	0.9901	0.99	0.9942	0.99

Table 5.11: Performance of the architecture classifier on the packed Debian dataset.

containing only a small portion of unencrypted code.

5.4.6 Shellcodes

In this experiment, we evaluate our classifier over the shellcode dataset subsection 5.2.5, composed by short sequences of handwritten machine code.

The results, obtained by 5-fold cross-validation, are reported in Table 5.12. As expected, the performance is lower than in other datasets, but the results show that our classifier achieves acceptable performances even on small amounts of binary code.

5.4.7 ByteWeight dataset

In this experiment, we test the model on the dataset from the ByteWeight paper (subsection 5.2.6). The results — reported in Table 5.13 — show that our method works well on this dataset composed of binaries derived from

Class	**Support**	**Precision**	**Recall**	**AUC**	**F1**
amd64	20	1.0	0.95	0.975	0.9744
arm	21	1.0	0.9524	0.9762	0.9756
i386	19	0.9	0.9474	0.9594	0.9231
mipsel	10	1.0	1.0	1.0	1.0
ppc	9	1.0	1.0	1.0	1.0
sparc	10	0.9091	1.0	0.9937	0.9524
Total	89	0.9682	0.975	0.9841	0.9709

Table 5.12: Performance of the architecture classifier on the shellcode dataset.

Class	**Support**	**Precision**	**Recall**	**AUC**	**F1**
x64	1097	0.9821	1.0	0.9909	0.991
x86	1100	1.0	0.9818	0.9909	0.9908
Total	2197	0.991	0.9909	0.9909	0.9909

Table 5.13: Performance of the architecture classifier on the ByteWeight dataset.

multiple compilers with different optimization levels. Also, our method can reliably distinguish two architectures having an overlapping instruction set (x86 and x86-64).

5.4.8 Size analysis

The goal of this experiment is to study the performance of our classifier on file fragments of varying sizes.

At first, we tried to filter our datasets to extract only the samples smaller than a certain size threshold, but this resulted in an extreme imbalance in the number of samples per class, and in an unreliable performance estimate of the model. So, we followed a different approach to generate samples of arbitrary sizes.

We extract the code section from each ELF file in the Debian dataset (subsection 5.2.3). Then, for each fragment size s between 8 bytes and 64 KiB, we take a sub-sequence of s bytes from each file, starting at a random position between 0 and length(file) $- s$. If length(file) $\leq s$, we take the entire file. In this way, we generate sets of fragments of different sizes.

We evaluate the classifier on each set of fragments via 10-fold cross-validation, considering the macro-averaged F-measure as the performance metric. All the features (byte frequencies and regexes) are enabled; the regularization parameter is $C = 10000$.

The results — reported in Figure 5.2 — show that even for small code

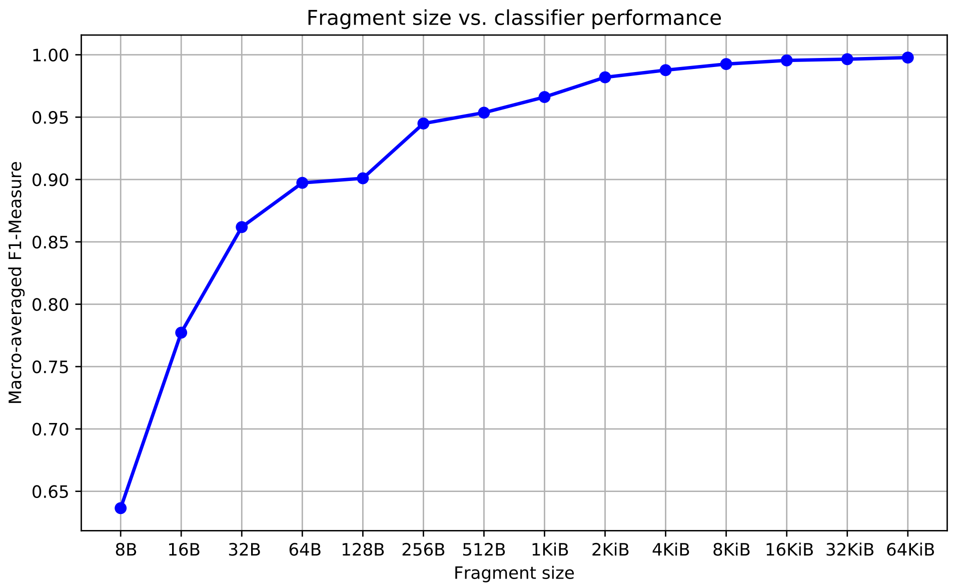


Figure 5.2: The classifier performance (macro-averaged F-measure) depending on the fragment size.

fragments (128 bytes), our classifier reaches a F-measure of 90%. For 512-byte fragments, the F-measure is over 95%.

5.5 Experiments: Code section identification

In this section, we will describe the experiments we performed on the code section identification method, described in section 3.3 and section 4.4.

5.5.1 Debian dataset

In this experiment, we evaluate the code section identification method on a subset of the Debian dataset (subsection 5.2.3). The algorithm is trained and tested separately on the binaries of each CPU architecture. The results are obtained with 5-fold cross-validation.

We used the following parameters of the model (their meaning is described in subsection 4.4.2):

- lookahead and lookbehind length: 1;
- 20 iterations;
- $C = 1$ (regularization strength);
- postprocessing cutoff: 0.1;

Arch	**Support**	**Code %**	**Accuracy**	**Precision**	**Recall**	**F1**	**Time (s)**
`amd64-post`	41	40.69	0.9995	0.9984	1.0	0.9992	9.6
`amd64-pre`	41	40.69	0.9984	0.9969	0.9992	0.998	9.5
`arm64-post`	33	47.83	0.9995	0.9989	1.0	0.9995	7.8
`arm64-pre`	33	47.83	0.9931	0.9934	0.9922	0.9927	7.3
`armel-post`	33	59.22	0.9983	0.9997	0.9977	0.9987	6.5
`armel-pre`	33	59.22	0.981	0.992	0.9749	0.9832	6.8
`armhf-post`	46	46.32	0.9997	0.9995	0.9999	0.9997	8.1
`armhf-pre`	46	46.32	0.9847	0.9881	0.9753	0.9813	8.1
`i386-post`	40	44.17	0.9995	0.9985	1.0	0.9992	16.3
`i386-pre`	40	44.17	0.9946	0.9914	0.9966	0.9939	16.2
`mips-post`	40	41.51	0.9995	0.9983	0.9999	0.9991	21.1
`mips-pre`	40	41.51	0.9958	0.9926	0.9955	0.994	20.7
`mipsel-post`	40	43.64	0.9919	0.9901	1.0	0.9941	8.1
`mipsel-pre`	40	43.64	0.9873	0.9807	0.9943	0.9866	8.1
`powerpc-post`	19	57.69	0.9992	0.9976	0.9999	0.9988	21.7
`powerpc-pre`	19	57.69	0.9911	0.9858	0.9962	0.9908	17.9
`ppc64el-post`	40	41.66	0.9985	0.9951	1.0	0.9975	15.7
`ppc64el-pre`	40	41.66	0.9916	0.9904	0.9924	0.9912	15.4

Table 5.14: Performance of the code section identification method over the different architectures of the Debian dataset, with and without postprocessing. The last column reports the average time spent on each cross-validation fold.

- minimum number of sections for postprocessing: 3.

Table 5.14 reports the results for each architecture, before and after applying the postprocessing phase described in subsection 3.3.5. The results show that the F-measure is over 99% for all the architectures with the postprocessing enabled.

The postprocessing algorithm consistently improves the performances of our model by removing the "noisy," small segments returned by the CRF. If we decrease the number of iterations of the SSVM learner, the contribution of the postprocessing algorithm becomes even more important. Figure 5.3 shows how the postprocessing algorithm compensates for the errors of a model trained with a low number of iterations. Thus, our postprocessing algorithm allows to reduce the training time of the model without sacrificing the quality of the predictions.

5.5.2 ByteWeight dataset

In this experiment, we test the code section identification method on the ByteWeight dataset we used before for the architecture classifier (subsection 5.2.6). This dataset is heterogeneous: it includes executables compiled with three different compilers, with four levels of optimization and for two operating systems.

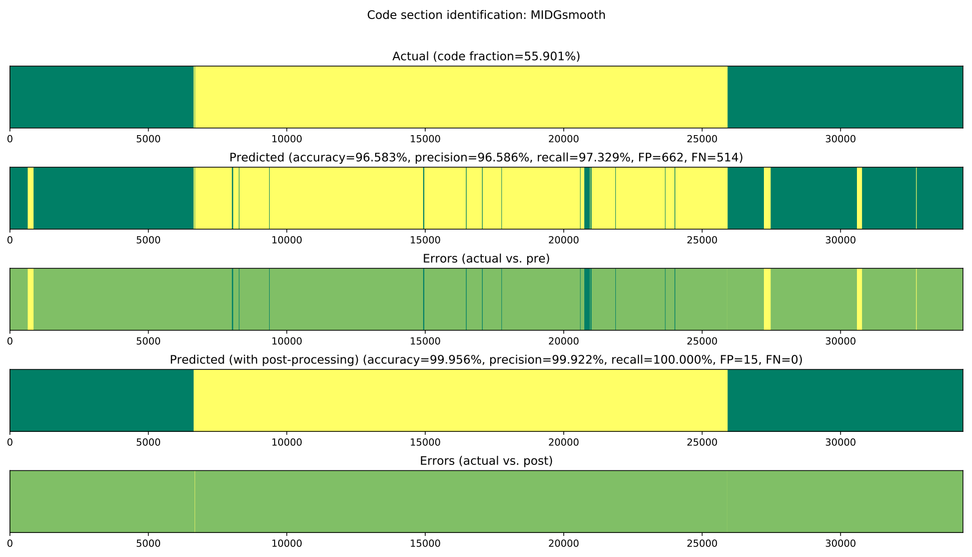


Figure 5.3: Visualization of the results of the code section identification method with a low number of iterations (10). The prediction — before post-processing — is inaccurate; however, after the preprocessing phase, there are no false negatives and the false positives are significantly reduced.

Dataset	Support	Code %	Accuracy	Precision	Recall	F1	Time (s)
ByteWeight (x64)	40	27.13	0.9992	0.9994	0.9987	0.999	174.8
ByteWeight (x86)	40	27.14	0.9998	0.9997	0.9996	0.9996	162.5
Mach-O	165	27.59	1.0	0.9998	1.0	0.9999	82.0
Arduino	73	9.56	0.9999	0.9993	1.0	0.9997	8.6

Table 5.15: Performances of the code section identification method on the ByteWeight, Mach-O and Arduino datasets. The last column reports the average time spent on each cross-validation fold.

We perform the experiment separately for the x86 and the x86-64 architectures. From the original dataset, we sample 40 executables for each architecture, and we compute the metrics by 5-fold cross-validation. We enable the postprocessing algorithm; set a lookahead and a lookbehind length of 1 byte; set the number of iteration of the SSVM learner to 20.

The results — reported in Table 5.15 — show that our model performs well even on this heterogeneous dataset.

5.5.3 Mach-O dataset

In this experiment, we test the code section identification method over the Mach-O dataset described in subsection 5.2.7. The evaluation procedure and the hyperparameters are identical to those used for the experiment over the Debian dataset (subsection 5.5.1).

The results — reported in Table 5.15 — show that our method works correctly with real-world Mach-O binaries.

5.5.4 Arduino dataset

In this experiment, we test the code section identification method over the Arduino (AVR) dataset described in subsection 5.2.8. The evaluation procedure and the hyperparameters are identical to those used for the experiment over the Debian dataset (subsection 5.5.1).

The results — reported in Table 5.15 — show that our method works correctly also with Arduino binaries compiled for the AVR architecture.

5.6 Experiments: code discovery

In this section, we will describe the experiments we performed to evaluate the code discovery method described in section 3.4 and section 4.5.

5.6.1 Windows dataset

In this experiment, we evaluate the code discovery method (section 3.4) over the dataset of Windows binaries (subsection 5.2.9) compiled with full debug symbols. In section 4.5, we explain in detail how this dataset has been generated.

We configured the model with the following parameters:

- $C = 1$
- $lookahead = lookbehind = 4$
- $max\_iter = 30$

The lookahead and the lookbehind lengths were determined by grid search (subsection 5.7.3).

For performance reasons, we trained and evaluated our model on a randomly chosen subset of the binaries in the dataset: since the binaries are large (the median size of the binaries is 1.68 MB), the model can be trained on a small number of samples without any problem. We performed holdout testing, reserving (for each architecture) 10 executables for training and 40 for testing.

The results — reported in Table 5.16 — show that the accuracy and the F-measure of our method on this dataset are over 99.9% for both the x86 and the x86-64 architectures. The accuracy of our model is in line with the mean accuracy (99.98%) of the approach in (Wartell et al. 2011), evaluated on 11 test binaries; however, to perform a proper comparison, the two methods should be evaluated on the same test set.

Arch	Support	Code %	Accuracy	Precision	Recall	F1	Time (s)
Windows (x64)	50	72.84	0.9997	0.9997	0.9999	0.9998	470.9
Windows (x86)	50	69.06	0.9996	0.9997	0.9997	0.9997	467.1

Table 5.16: Performance of the code discovery method on the Windows dataset. The last column reports the running time of the training and testing phases.

This experiment shows that a sequential learning model, trained on simple byte-level features, is able to effectively separate code from data in Windows binaries.

5.6.2 ARM dataset

We evaluate the code discovery method for fixed-length instruction architectures (subsection 3.4.2) over the dataset containing the GNU coreutils binaries compiled for ARM with full debugging symbols (the dataset is described in subsection 5.2.10).

We expect the performance to be higher than the variable-length case, because these binaries are easier to predict, indeed:

1. the segmentation of each "atom" of code and data is fixed and known (4-byte words), so the model can work directly on these blocks, and not on the bytes;
2. the features are generated by a linear disassembler (`objdump`) which can detect the 4-byte words which cannot be decoded into valid ARM instructions.

The number of iterations of the SSVM learner is 20; the lookahead and lookbehind parameters are set to 1 instruction (i.e., 4 bytes). We evaluate the model separately for each level of optimization of the binaries in the dataset, to avoid overfitting: we do not want to train and test the model on two binaries of the same program, differing only for the level of optimization of the compiler. We performed 5-fold cross-validation to compute the performance metrics.

The results — reported in Table 5.17 — show that the predictions of our model are almost perfect (accuracy of over 99.9%) for the binaries of any optimization level. These results show that the problem of code discovery for ARM executables can be solved by a probabilistic language model using simple features derived from a linear sweep disassembler.

Dataset	Support	Code %	Accuracy	Precision	Recall	F1	Time (s)
coreutils (`-O0`)	103	94.64	1.0	1.0	1.0	1.0	9.6
coreutils (`-O1`)	103	92.41	0.9998	0.9998	1.0	0.9999	6.6
coreutils (`-O2`)	103	92.12	0.9998	0.9998	1.0	0.9999	6.3
coreutils (`-O3`)	103	92.86	0.9998	0.9998	1.0	0.9999	7.5

Table 5.17: Performance of the code discovery method on the ARM coreutils dataset. The last column reports the average time spent on each cross-validation fold.

5.7 Hyperparameter tuning

In this section, we will describe the experiments we executed to determine the optimal hyperparameters for our models (subsection 5.3.3), and we will comment the results.

5.7.1 Hyperparameter tuning on the architecture classifier

The purpose of this experiment is to determine the optimal value for the C parameter (i.e., the inverse of the regularization strength) for the L_1 (Lasso) regularization of the Logistic Regression classifier.

To find the optimal value for C, we proceed as follows. First, we randomly split the dataset into a training set (75%) and a test set (25%). We run a grid search with cross-validation on the training set: the grid search algorithm, for each provided value of C, performs a 10-fold cross-validation and computes the estimate of the mean performance metric and the standard error of the estimate of that mean. We choose the macro-averaged F-measure as the performance metric to optimize, to give the same importance to all the classes, as explained in section 5.3. At the end, the grid search returns the performance metrics associated with each value of C. We select the value of C which gives the highest performance.

Finally, we test the model with the "best" value of C on the test set — which was not used for the grid search — to check whether the performance is still good. This step is needed because it is possible for the grid search algorithm to choose a hyperparameter giving a high performance on the training set, but a poor performance on unseen samples. In other words, we risk to overfit on the choice of the hyperparameters.

In Figure 5.4, we show the average F-measure together with its standard error for the Debian dataset, evaluated for different values of C. The results show that a correct selection of the regularization parameter gives a slightly better result than no regularization (high values of C). We checked that the model does not lose performance on the test set.

We also tested our method with a higher regularization strength ($C < 10$): in this case, the model underfits and the performance is poor.

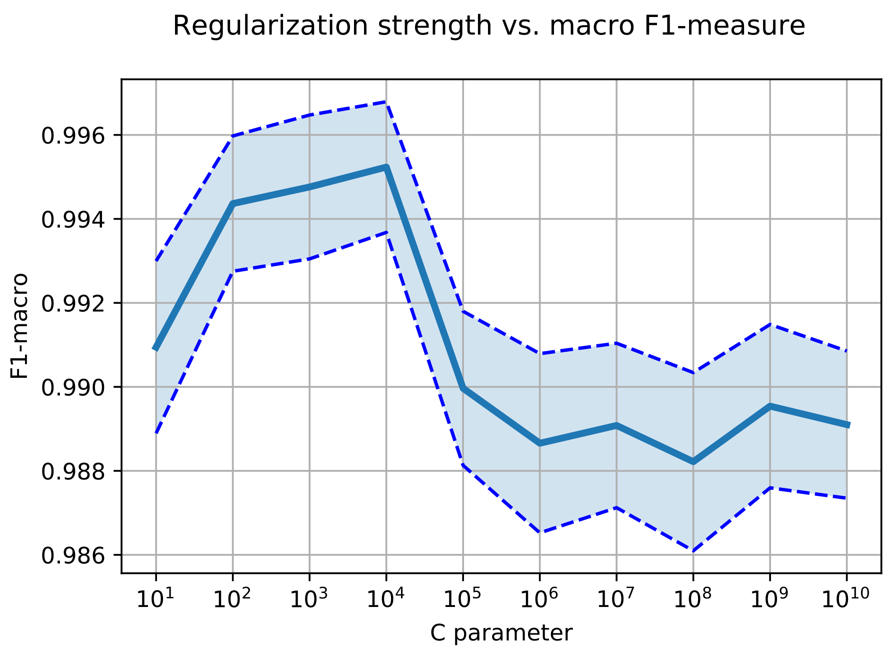


Figure 5.4: The average performance of the classifier on the Debian dataset for different values of C. The upper and lower bounds show the standard error of the measure.

5.7.2 Hyperparameter tuning for code section identification

The purposes of this experiment are to determine the optimal lookahead/lookbehind lengths for the code section identification method, and to compare the performances of the model with and without applying the postprocessing algorithm. For simplicity, we set the lookahead length to be always equal to the lookbehind length; however, our model allows to specify different lengths.

We used the binaries from the `amd64` architecture of the Debian dataset; we randomly select two thirds of them to perform the grid search, and one third to perform the final testing. We execute a grid search with 5-fold cross-validation, by evaluating all the possible choices in the following parameter space:

1. lookahead/lookbehind lengths from 0 to 8;
2. postprocessing enabled or disabled.

The number of iterations for the training of the SSVM learner is 10. For the postprocessing phase, we used the following parameters (see subsection 3.3.5 for an explanation):

- `postprocessing_cutoff=0.1`

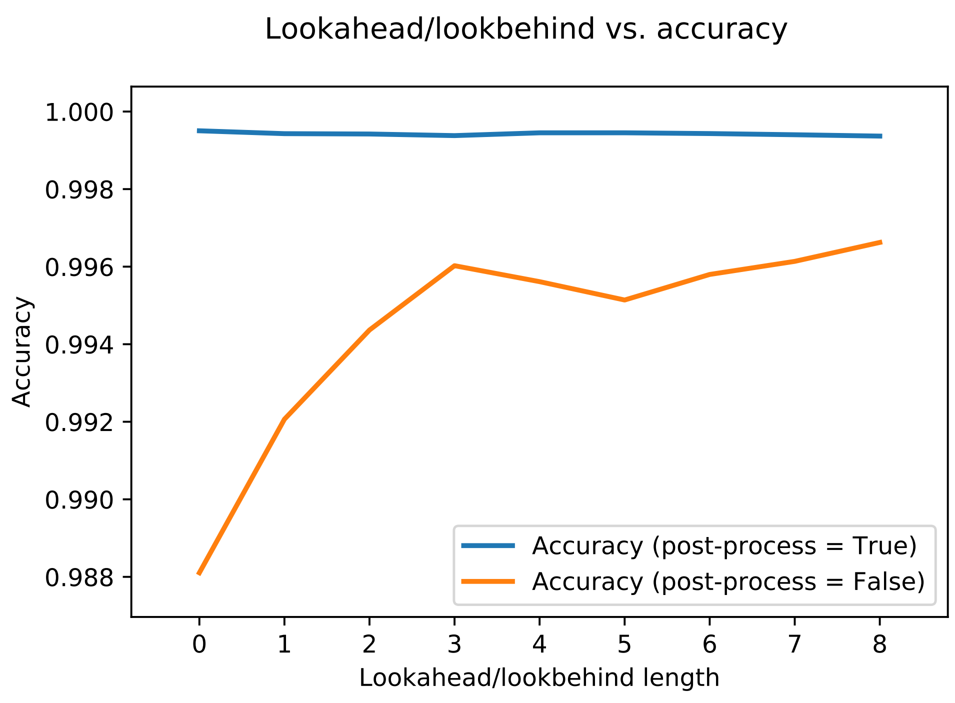


Figure 5.5: Accuracy of the code section identification method, with and without postprocessing, depending on the lookahead-lookbehind length.

- `postprocessing_min_sections=4`

The results — reported in Figure 5.5 — show that, for any choice of the lookahead length, the model with the postprocessing step enabled always outperforms the model without it. The accuracy in the model with postprocessing is consistently high: it does not change depending on the choice of the hyperparameter.

According to this data, we decided to set a minimal lookahead and lookbehind length of 1 byte for our experiments, and to enable the postprocessing phase.

5.7.3 Hyperparameter tuning for code discovery on the Windows dataset

The goal of this experiment is to select the optimal values of the lookahead and lookbehind lengths for the code discovery method. As before, for simplicity, we only consider equal lookahead and lookbehind lengths. By increasing the lookahead length, we expect to increase the model accuracy at the expense of a longer training time and of an increased memory utilization for the additional features.

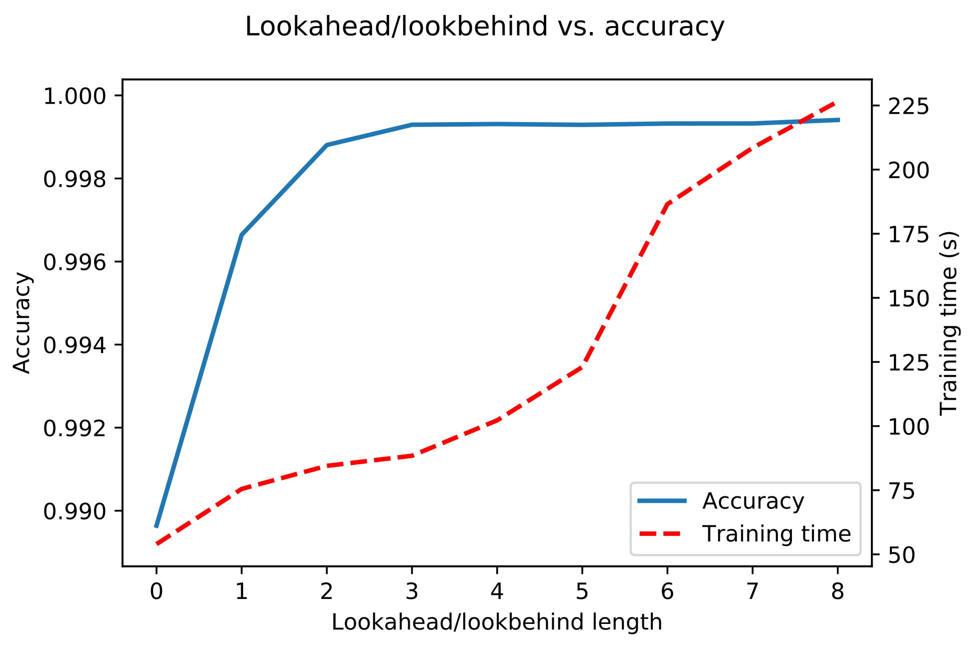


Figure 5.6: The accuracy of the model and the training time vs. the lookahead-lookbehind length, for the Windows dataset.

We run a grid search over the lookahead/lookbehind parameters, ranging from 0 (no lookahead/lookbehind) to 8 (for each byte, 16 surrounding bytes are considered).

We randomly pick 9 binaries from the Windows dataset and perform a 3-fold cross-validation for each value of the hyperparameter. We set the number of the iterations of the SSVM learner to 10.

The results — reported in Figure 5.6 — show that there is no significant performance improvement when the lookahead length is higher than 3 bytes. This means that a *third-order Markov model* (Bishop 2006, p. 608) approximates our data well enough. The model without any lookahead and lookbehind obtains an acceptable accuracy (99.0%).

For the experiments, we choose a lookahead/lookbehind length of 4 bytes as a reasonable tradeoff between model accuracy and training time.

5.7.4 Hyperparameter tuning for code discovery on the ARM dataset

The purpose of this experiment is to select the optimal lookahead/lookbehind lengths for the code discovery method for ARM (an ISA with fixed instruction length). Once again, for simplicity we set the lookbehind length equal to the lookahead length.

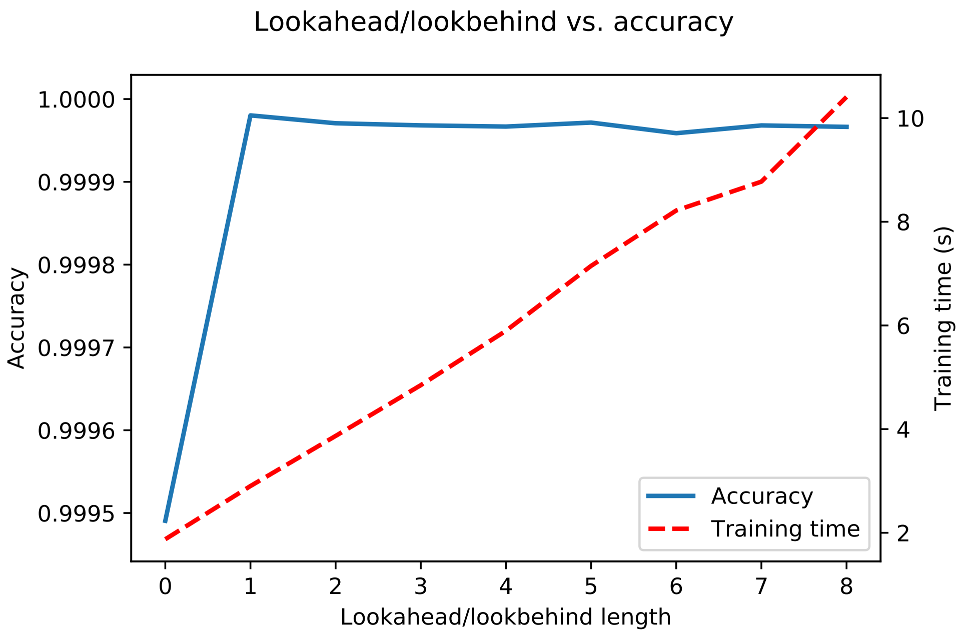


Figure 5.7: The accuracy of the model and the training time vs. the lookahead-lookbehind length, for the ARM dataset.

We execute a grid search testing the lookahead lengths from 0 to 8. For each choice of the parameter, we evaluate the accuracy of the model with a 10-fold cross-validation. We use the ARM coreutils dataset (subsection 5.2.10), considering the binaries with optimization level `-O1`. We use 75% of the dataset for the grid search and the 25% for the final testing. We fixed the number of iterations of the SSVM learner to 10.

The results — reported in Figure 5.7 — show that the model without lookahead nor lookbehind scores well; a slight performance improvement can be obtained by setting a lookahead/lookbehind length equal to 1 word. No benefits are obtained by employing longer lookahead lengths. As expected, the training time increases linearly with the lookahead length.

Chapter 6

Limitations

In this chapter, we will analyze the limitations of our approaches, as well as some ideas about how to overcome them. As usual, we will address the limitations concerning the three analyses separately.

6.1 Architecture classifier

The first limitation of our CPU architecture classifier is intrinsic to all the supervised learning methods: the approach is only as good as the training set. If the training set used to train the model is not sufficiently rich, or it is not representative of the executable files to be analyzed, our method will inevitably fail.

Our classifier works by considering the frequencies of the bytes and of the patterns of function prologues and epilogues. An adversary could actively confuse our classifier by including the patterns typical of the "wrong" architecture in the data sections of an executable, or by employing advanced packing techniques. A simple obfuscation technique would be to include a fabricated "data" section containing a large amount of code of another architecture into the binary file. Resisting to such attacks is beyond the scope of our work.

Our method may fail to recognize binaries generated by a compiler not represented in the training set, if it produces different byte patterns.

Our model relies on a fixed list of patterns, and it is not able to derive the features on its own. A valuable contribution would be to implement an automated feature engineering phase, i.e., a component which would automatically select the most useful patterns. As we will explain in section 7.1, such extension is not trivial, and it would introduce further problems which would need to be addressed. We decided to develop a simpler model, which runs fast, has a good performance and requires tuning a single parameter.

6.2 Code section identification

We were not able to evaluate our approach on real-world firmwares because of the lack of ground truth data, i.e., the precise location of the code and the data sections.

The approach we presented suffers from the typical limitations of supervised learning methods: if the training set is not representative of the samples that the analyst wants to predict, the model will fail. In our case, different compilers, or compiler settings (e.g., optimization levels), may produce different patterns. If the byte patterns in the binary to analyze are significantly different from those present in the training set, the model will fail to identify the code and data segments.

Our model is not able to recognize code contained in compressed or encrypted files. Indeed, the outputs of ideal compression and encryption algorithms are indistinguishable from random data; otherwise, respectively, the file could be compressed further or it may leak information (Penrose, Macfarlane, and Buchanan 2013, p. 2). Such files do not exhibit any recognizable byte pattern that our model can learn.

Our analysis may be actively defeated by putting in the executables' data sections some executable code, or by interleaving a large amount of data with code in the executable section. Defending against these kinds of obfuscation is beyond the scope of our work.

We recall that the aim of this analysis is to retrieve the boundaries of the *code section*, not to determine which bytes correspond to correct machine code (that is the purpose of the code discovery method). Sometimes the code section of executable files contains inline data, which may be similar to the data located *outside* the code section; our model could classify such small data sequences as data even if they belong to the code section. This explains the need for a postprocessing step to obtain few, contiguous data and code sequences. For this reason, the tuning of the parameters of the postprocessing step (Algorithm 1) is critical if the code section contains many inline data sequences.

Thus, the analyst needs to select the parameters of the postprocessing phase, and this makes the analysis less easy to perform. The optimal values may be determined by hyperparameter tuning on the training set (i.e., select the parameters maximizing the performance metric).

6.3 Code discovery

A significant issue which emerged in all the attempts to solve the code discovery problem in literature (subsection 2.3.4) is the reliability of the ground truth data. As we discussed in subsection 2.4.1, some works in this field choose to use existing disassemblers to provide ground truth data; others rely on manual analysis. Our approach — like (Bao et al. 2014; Rosenblum et al. 2008; Andriesse, Slowinska, and Bos 2017) — uses the debugging symbols generated by the linker to get precise ground truth data; however, the debug symbols are not always complete: there are some functions in the executable code which are not covered by the symbols, and the sizes of the data items are not provided (section 3.4).

This problem could be partially overcome by using state-of-the-art disassemblers in conjunction with the debugging symbols. We followed this approach to generate the ground truth for the ARM binaries (section 4.5) by letting `objdump` parse the DWARF symbols, but we were not able to do the same for the Windows binaries because of the technical difficulties involved in getting the disassemblers to work with Microsoft's DIA API.

As we saw in the implementation details, extracting the ground truth data from the PDB files present technical difficulties and requires a working Windows environment. We were not able to find any tool able to correctly parse PDB files on its own without resorting to Microsoft's implementation. Also, the ground truth can only be extracted at compile time from the programs whose source code is available; in other words, it is not possible to extract the ground truth from a program which had already been compiled, and whose debugging symbols are not available.

Our model for variable-length instruction (section 3.4) has no notion of "instruction": we train and evaluate the sequential learner over plain sequences of bytes, and define the accuracy of our model as the fraction of correctly classified bytes in each sample. This byte-level accuracy does not correspond to the disassembler accuracy: if the disassembler starts disassembling even one byte off, it may incorrectly decode a whole block of instructions. Different loss functions — like the "instruction loss" and the "block loss" (Karampatziakis 2010) — can be used to evaluate disassembly errors at the instruction and at the block level; however, computing them requires more precise ground truth.

Another limitation of this approach is that the training set must be representative of the binaries to analyze; hence, it is necessary to collect a dataset of executables with the corresponding symbol files for the correct architecture.

Chapter 7

Future Works

In this chapter, we will discuss some possible future research directions, starting from the ideas described in this work.

7.1 Architecture classifier

Our supervised learning approach to classify the CPU architecture of executable files may be extended by introducing an automated feature engineering phase, to automatically identify the most relevant byte patterns instead of relying on a fixed set of patterns, as we do in the current approach. This improved feature engineering phase would identify the "best" patterns, i.e., those which discriminate the most between the architectures. These patterns could be either simple multi-byte sequences, or regular expressions.

This idea, however, would add significant complexity to the model, introducing additional problems. Sequential pattern mining is computationally expensive, because the number of patterns to test increases exponentially with their length. Moreover, such a model could overfit on features extraneous to the code (e.g., padding sequences or common data patterns). In (Rosenblum et al. 2008), the authors performed automated pattern mining to generate the features for their function identification approach; the feature selection phase consumed over 150 compute-days (2 days in real time) in a highly parallel and distributed environment.

7.2 Code section identification

Our code section identification approach could be used to solve the more general problem of type identification of file fragments, a recurring task in forensic analysis. Also, our model could be applied to classify the sub-components of a compound file, i.e., a file which contains segments of heterogeneous types (e.g., a PDF file containing text, bitmap images, and vector graphics).

Our approach can also be used to find the executable sections inside firmware blobs. Unfortunately, it would not be easy to evaluate such an approach on firmwares because of the lack of ground truth data.

Our method could be further extended to identify other kinds of sections inside executable files (jump tables, headers, the GOT, relocations, ...).

7.3 Code discovery

The quality and the coverage of the ground truth provided by the debugging symbols may be further improved. As we explained in section 3.4, the debugging symbols do not cover some portions of the binaries. In our experiments, we discarded those portions since we already had plenty of training data.

The information from the debugging symbols could be combined with those coming from a disassembler. This hybrid approach would help to cover a wider portion of the code, and would provide other useful features to use in the model. In (Andriesse, X. Chen, et al. 2016), the authors follow an alternative approach: a linear disassembly is started at each byte corresponding to the beginning of every line of code. This approach does not give a *complete* disassembly, because not every machine instruction has a corresponding line in the source code; however, it could be used in combination with other techniques (as the paper does).

The model we use (linear-chain CRF) is a flexible and general-purpose sequence classification model: in principle, any feature can be assigned to the items of the sequence. In our implementation, we only used the one-hot encoded byte values, but other features may be inserted, e.g., those resulting from a disassembler or from the knowledge of the opcodes of the ISA.

Our current model does not have any notion of “instruction”: the Conditional Random Field simply labels each byte as “code” or “data,” without hard constraints over the classification results. This means that our approach could potentially tag as “code” some subsequences which are not valid instructions in the ISA. This risk is balanced by the advantages of a simple approach: learning is fast and no external knowledge is needed.

A more advanced model could include the knowledge about the ISA, e.g., by allowing only valid instructions to be classified as code. Introducing this constraint is not trivial: the model would need to perform structured learning on two different levels of abstraction, simultaneously: bytes and instructions. The improved model would have to label each byte either as code or data, and to group the bytes corresponding to code into valid instructions.

In (Wartell et al. 2011) the instruction set is taken into account, but the segmentation algorithm only considers byte-level features. In (Karampatziakis 2010), the constraints given by the instruction set are encoded in a graph structure in which each byte is a node.

A further step forward would be to learn also over the sequence of *instructions* identified by the model itself, in the case of variable length instructions. New instruction-level features would be introduced (e.g., the opcodes). We were able to assign features to the instructions only in the fixed-length case, where the instructions are trivially segmented in 4-byte words and can be directly labeled (subsection 3.4.2); the linear-chain CRF model is not powerful enough to assign features to subsequences of bytes in the variable-length instruction case.

Semi-CRFs (Sarawagi and Cohen 2005) are an extension of Conditional Random Fields, which allow to assign features and labels to subsequences (i.e., instructions) of elements (i.e., bytes). Training semi-CRFs is computationally more complex than "plain" CRFs; nevertheless, they may represent a valid approach to build a more advanced model to solve the code discovery problem in executable files.

Our code discovery approach for ARM binaries may be used to improve the results of the code discovery method for ARM/Thumb mixed ISA binaries (J.-Y. Chen et al. 2013), or extended to directly support Thumb (16-bit instructions) code.

Chapter 8

Conclusions

In the Introduction (chapter 1), we stated the following goals:

1. develop a supervised learning method to identify the CPU architecture (ISA) and the endianness of header-less executable files;
2. develop a sequential learning method to identify the boundaries of the code sections in a header-less binary file;
3. develop a sequential learning method to solve the code discovery problem, i.e., distinguish the machine code from data inside the code section of stripped, header-less executable files.

In chapter 3, we presented the approach we chose to follow.

We faced the architecture identification problem as a supervised classification problem. We collected the training sets for the classifier from multiple sources (packages from a Linux distribution, datasets from scientific papers, shellcodes, packed binaries). We derived the features from the Byte Frequency Distribution and from the frequencies of multi-byte patterns representing function prologues and epilogues; developed the classifier using Logistic Regression with Lasso regularization; performed hyperparameter tuning to determine the optimal regularization strength.

To solve the code section identification problem, we developed a sequential learning model based on Conditional Random Fields learned via Structural SVMs. We used the one-hot encoded byte values as features. The model learns the conditional probabilities of the sequences of bytes in the code and data sections of the binaries in the training set. We adapted the sequential learning library (`pystruct`) to support sparse feature matrices, to reduce the memory and CPU requirements of our method. We developed a postprocessing algorithm, whose purpose is to reduce the noise in the prediction output, by coalescing the predictions for each byte into large, contiguous sections of the file. Once it has been trained, our model can divide any binary file into code and data sections. The strength of our approach, compared to the existing file fragment classification methods (subsection 2.3.1), is that our model, by predicting the class of each byte, solves both the segmentation and the classification problems at the same time.

We developed a similar sequential learning approach to solve the code discovery problem, i.e., distinguish the bytes corresponding to machine code instructions from the bytes corresponding to inline data inside the executable section of a binary file. We considered the most difficult case: a dataset of binaries for two architectures with variable-length instructions (x86 and x86-64). We built our training set by compiling a set of Windows programs with full debugging symbols, which include the offsets of the functions and of the inline data; we extracted the ground truth by parsing the debugging symbols. We trained the same sequential learning model described for the code section identification method (linear-chain CRF) on the training set, using the byte values as features. The trained model predicts whether each byte of an unseen executable file is part of a valid machine instruction, or if it is inline data.

We also built a variant of this method to detect inline data inside binaries compiled for a fixed-instruction-length architecture (ARM). This model is simpler and faster, because in ARM binaries the instructions and the data blocks are always aligned to 4-byte words inside the file. We obtained the features by running a linear disassembler over the binary files and by collecting the opcodes decoded from each 4-byte word. The DWARF debugging symbols inside the executables provide the ground truth. We used the same CRF-based classifier as before; the only difference is that the sequential learner classifies 4-byte words instead of bytes.

In chapter 5, we tested our methods on real data and evaluated the performance of our models.

We evaluated the architecture classifier on 20 different architectures; our model scored well with an averaged F-measure over 99%. The datasets include both files containing exclusively machine code, and complete binaries containing code and data sections. We also tested our classifier on more challenging datasets (shellcodes and packed binaries): we still obtained a more than satisfactory performance.

We evaluated the code section identification method on binaries coming from: Debian packages for multiple architectures, a dataset of ELF and PE files from (Bao et al. 2014), and a macOS system. Our method scored well on all the datasets: on average, it classified correctly over 99.5% of the bytes of the test samples. The results of the experiments show that the postprocessing algorithm we developed consistently improves the performance of the model. We determined the optimal values of the parameters of the model by hyperparameter tuning.

We evaluated the code discovery method on a dataset of Windows binaries compiled with full debugging symbols. The model correctly classified over 99.9% of the bytes in the binaries. We also tested the code discovery method on a dataset of ARM binaries. In this case, we obtained an accuracy of 99.98%. We determined the optimal values of the parameters of the model by hyperparameter tuning.

An interesting development for the architecture classifier would be to add an automated feature engineering phase, i.e., an algorithm to autonomously generate and select the most relevant patterns for the classification task. We decided not to implement such method, to keep our classifier simple and fast; our implementation, instead, relies on a fixed set of patterns. The approach could be also extended to classify non-executable files by their types. We do not expect our model to be resistant to deliberate obfuscation efforts specifically aimed to conceal the CPU architecture of executable files.

The code section identification method is not able to detect code inside compressed or encrypted files, and can be defeated by obfuscation techniques explicitly targeted to confuse our model (e.g., by putting large amounts of dead code in the data sections of the file). Our approach may be extended to detect not only the executable regions of a binary files, but also other sections which exhibit peculiar patterns (e.g., jump tables, relocations, headers…). An approach similar to ours can be used, in general, to divide a file into segments and classify the segments by their type (this may be useful, for example, in the computer forensics domain, when file carving is not a suitable option).

The code discovery method can be improved by integrating the sequential learning model with a disassembler, e.g., by encoding the disassembler output into additional features. Our model works at the byte level and has no knowledge of the notion of “instruction”: the approach may be extended by introducing additional constraints — derived from the knowledge of the ISA — to enforce the consistency of the prediction output.

Previous works (subsection 2.3.5) widely explored the strengths and the limitations of the static analysis approaches based on the recursive traversal of the code, and proposed increasingly sophisticated analyses over the instruction flow. While some of these approaches have obtained remarkable results, they fall short whenever the execution flow cannot be statically traced with certainty (e.g., when complex indirect jumps are present). The main limitation of this kind of approaches is that statically discovering all the reachable code regions in an executable file is an undecidable problem in computer science; the problem can not be resolved in the general case once and for all.

In this work, we address the problem of code/data separation by looking at the machine code from a purely syntactical, probabilistic point of view: our method consists in learning a language model to compute the likelihood of a byte to be code (or data).

We feel that the application of machine learning and statistical techniques to the static analysis of executable files will give substantial contributions in this research field.

Bibliography

Andriesse, Dennis (2016). *Inline Jump Tables on ARM & Function Detection Using The .eh_frame Section.* URL: `https://writings.mistakenot.net/arm-jump-tables-and-eh_frame/` (visited on 2017-08-16).

Andriesse, Dennis, Xi Chen, et al. (2016). "An In-Depth Analysis of Disassembly on Full-Scale x86/x64 Binaries". In: *25th USENIX Security Symposium (USENIX Security 16).* Austin, TX: USENIX Association, pp. 583–600. ISBN: 978-1-931971-32-4. URL: `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/andriesse`.

Andriesse, Dennis, Asia Slowinska, and Herbert Bos (2017). "Compiler-Agnostic Function Detection in Binaries". In: URL: `https://syssec.mistakenot.net/papers/eurosp-2017.pdf` (visited on 2017-04-05).

angr (2017). *angr/archinfo.* URL: `https://github.com/angr/archinfo` (visited on 2017-07-31).

Arduino (2017). *Built-In Examples.* URL: `https://www.arduino.cc/en/Tutorial/BuiltInExamples` (visited on 2017-08-01).

Bao, Tiffany et al. (2014). "ByteWeight: Learning to Recognize Functions in Binary Code". In: *Proceedings of the 23rd USENIX Conference on Security Symposium.* SEC'14. San Diego, CA: USENIX Association, pp. 845–860. ISBN: 978-1-931971-15-7. URL: `http://dl.acm.org/citation.cfm?id=2671225.2671279`.

Bishop, Christopher M. (2006). *Pattern Recognition and Machine Learning.* Springer. ISBN: 978-0387-31073-2.

Blanc, Bertrand and Bob Maaraoui (2005). *Endianness, or: Where is Byte 0?* 3B Consultancy. URL: `http://3bc.bertrand-blanc.com/endianness05.pdf`.

Buluç, Aydin et al. (2009). "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks". In: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures.* ACM, pp. 233–244.

Chen, Jiunn-Yeu et al. (2013). "Effective Code Discovery for ARM/Thumb Mixed ISA Binaries in a Static Binary Translator". In: *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems.* CASES '13. Montreal, Quebec, Canada: IEEE

Press, 19:1–19:10. ISBN: 978-1-4799-1400-5. URL: `http://dl.acm.org/citation.cfm?id=2555729.2555748`.

Clemens, John (2015). “Automatic classification of object code using machine learning”. In: *Digital Investigation* 14, S156–S162. ISSN: 17422876. DOI: `10.1016/j.diin.2015.05.007`. URL: `http://linkinghub.elsevier.com/retrieve/pii/S1742287615000523` (visited on 2017-04-05).

Eager, Michael J and Eager Consulting (2012). *Introduction to the DWARF debugging format*. URL: `http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf`.

ELF for the ARM® Architecture (2015). ARM IHI 0044F. ARM Limited. URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ihi0044f/IHI0044F_aaelf.pdf`.

Karampatziakis, Nikos (2010). “Static Analysis of Binary Executables Using Structural SVMs”. In: *Advances in Neural Information Processing Systems 23*. Ed. by J. D. Lafferty et al. Curran Associates, Inc., pp. 1063–1071. URL: `http://papers.nips.cc/paper/3925-static-analysis-of-binary-executables-using-structural-svms.pdf`.

Křoustek, J. et al. (2012). “Accurate Retargetable Decompilation Using Additional Debugging Information”. In: *Proceedings of the Sixth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE’12)*. Rome, IT: IARIA, pp. 79–84.

Kruegel, Christopher et al. (2004). “Static Disassembly of Obfuscated Binaries”. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM’04. San Diego, CA: USENIX Association, pp. 18–18. URL: `http://dl.acm.org/citation.cfm?id=1251375.1251393`.

Lacoste-Julien, Simon et al. (2013). “Block-coordinate Frank-Wolfe Optimization for Structural SVMs”. In: *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*. ICML’13. Atlanta, GA, USA: JMLR.org, pp. I-53–I-61. URL: `http://dl.acm.org/citation.cfm?id=3042817.3042825`.

Lafferty, John D., Andrew McCallum, and Fernando C. N. Pereira (2001). “Conditional Random Fields: Probabilistic Models for Segmenting and Labeling Sequence Data”. In: *Proceedings of the Eighteenth International Conference on Machine Learning*. ICML ’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., pp. 282–289. ISBN: 1-55860-778-1. URL: `http://dl.acm.org/citation.cfm?id=645530.655813`.

Li, Wei-Jen et al. (2005). “Fileprints: Identifying file types by n-gram analysis”. In: *Information Assurance Workshop, 2005. IAW’05. Proceedings from the Sixth Annual IEEE SMC*. IEEE, pp. 64–71. URL: `http://ieeexplore.ieee.org/abstract/document/1495935/` (visited on 2017-04-05).

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*. CCS ’03. Washington D.C., USA: ACM, pp. 290–299. ISBN: 1-58113-738-9. DOI: `10.`

1145/948109.948149. URL: http://doi.acm.org/10.1145/948109.948149.

Mac OS X ABI Mach-O File Format Reference (2007). Apple, Inc. URL: http://www.idea2ic.com/File_Formats/MachORuntime.pdf (visited on 2017-08-20).

McDaniel, M. and M. H. Heydari (2003). “Content based file type detection algorithms”. In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, p. 10. DOI: 10.1109/HICSS.2003.1174905.

Microsoft (2016). *Microsoft/microsoft-pdb.* URL: https://github.com/Microsoft/microsoft-pdb (visited on 2017-07-06).

— (2017[a]). *Debugging with Symbols - Windows Dev Center.* URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ee416588(v=vs.85).aspx (visited on 2017-07-09).

— (2017[b]). *Dia2dump Sample | Microsoft Docs.* URL: https://docs.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/dia2dump-sample (visited on 2017-07-09).

— (2017[c]). *Download a Windows 10 virtual machine - Windows app development.* URL: https://developer.microsoft.com/en-us/windows/downloads/virtual-machines (visited on 2017-07-13).

— (2017[d]). *Microsoft/Windows-universal-samples.* URL: https://github.com/Microsoft/Windows-universal-samples (visited on 2017-07-13).

— (2017[e]). *MSDN | MSBuild.* URL: https://docs.microsoft.com/en-us/visualstudio/msbuild/msbuild (visited on 2017-07-13).

— (2017[f]). *PE Format.* URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547 (visited on 2017-08-20).

Müller, Andreas C. and Sven Behnke (2014). “pystruct - Learning Structured Prediction in Python”. In: *Journal of Machine Learning Research* 15, pp. 2055–2060. URL: http://jmlr.org/papers/v15/mueller14a.html.

NVIDIA (2017). *CUDA Samples :: CUDA Toolkit Documentation.* URL: http://docs.nvidia.com/cuda/cuda-samples/index.html (visited on 2017-08-01).

Oberhumer, MFXJ, László Molnár, and John F Reiser (2004). *UPX: the Ultimate Packer for eXecutables.* URL: https://upx.github.io/.

Penrose, Philip, Richard Macfarlane, and William J. Buchanan (2013-12). “Approaches to the classification of high entropy file fragments”. In: *Digital Investigation* 10.4, pp. 372–384. ISSN: 17422876. DOI: 10.1016/j.diin.2013.08.004. URL: http://linkinghub.elsevier.com/retrieve/pii/S174228761300090X (visited on 2017-04-05).

Reghizzi, Stefano Crespi, Luca Breveglieri, and Angelo Morzenti (2013). *Formal Languages and Compilation.* 2nd ed. Springer. ISBN: 978-1-4471-5513-3. DOI: 10.1007/978-1-4471-5514-0.

Rosenblum, Nathan et al. (2008). “Learning to Analyze Binary Computer Code”. In: *Proceedings of the 23rd National Conference on Artificial Intel-*

ligence - Volume 2. AAAI'08. Chicago, Illinois: AAAI Press, pp. 798–804. ISBN: 978-1-57735-368-3. URL: `http://dl.acm.org/citation.cfm?id=1620163.1620196`.

Sarawagi, Sunita and William W Cohen (2005). "Semi-Markov Conditional Random Fields for Information Extraction". In: *Advances in Neural Information Processing Systems 17*. Ed. by L. K. Saul, Y. Weiss, and L. Bottou. MIT Press, pp. 1185–1192. URL: `http://papers.nips.cc/paper/2648-semi-markov-conditional-random-fields-for-information-extraction.pdf`.

Schirra, Sascha (2017). *FileBytes*. URL: `https://github.com/sashs/filebytes` (visited on 2017-07-13).

scikit-learn (2017). *APIs of scikit-learn objects*. URL: `http://scikit-learn.org/stable/developers/contributing.html#apis-of-scikit-learn-objects` (visited on 2017-07-12).

Shin, Eui Chul Richard, Dawn Song, and Reza Moazzezi (2015). "Recognizing Functions in Binaries with Neural Networks". In: *Proceedings of the 24th USENIX Conference on Security Symposium*. SEC'15. Washington, D.C.: USENIX Association, pp. 611–626. ISBN: 978-1-931971-232. URL: `http://dl.acm.org/citation.cfm?id=2831143.2831182`.

Shoshitaishvili, Y. et al. (2016). "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138–157. DOI: `10.1109/SP.2016.17`.

Sportiello, Luigi and Stefano Zanero (2012). "Context-based file block classification". In: *IFIP International Conference on Digital Forensics*. Springer, pp. 67–82. URL: `http://link.springer.com/chapter/10.1007/978-3-642-33962-2_5` (visited on 2017-04-05).

Taskar, Ben, Carlos Guestrin, and Daphne Koller (2004). "Max-margin Markov networks". In: *Advances in neural information processing systems*, pp. 25–32.

The cLEMENCy Architecture (2017). Legitimate Business Syndicate. URL: `https://blog.legitbs.net/2017/07/the-clemency-architecture.html` (visited on 2017-08-31).

The PDB File Format (2017). LLVM Project. URL: `https://llvm.org/docs/PDB/index.html` (visited on 2017-08-19).

Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2 (1995). TIS Committee. URL: `http://refspecs.linuxbase.org/elf/elf.pdf` (visited on 2017-07-22).

Wartell, Richard et al. (2011). "Differentiating code from data in x86 binaries". In: *Machine Learning and Knowledge Discovery in Databases*, pp. 522–536. URL: `http://www.springerlink.com/index/L875Q3N675707N56.pdf` (visited on 2017-04-05).