MONASH University

Deep Learning for Software Security

Van Khac Nguyen

MSc (Master of Science)

A thesis submitted for the degree of Doctor of Philosophy at

Monash University in 2020

Faculty of Information Technology

Copyright Notice

Abstract

Computer software plays a significant role in modern life. A large number of technologies and software methodologies have been employed to develop enormous varieties of computer software on different platforms (e.g., Linux and Windows) from very simple applications to complex enterprise software systems, which makes software security one of the most impactful and critical problems of cybersecurity. Due to the ubiquity of computer software, software vulnerabilities (SVs) (i.e., specific potential flaws, weaknesses or oversights in computer software that attackers can use to carry out malicious actions) have become a serious and crucial concern in the software industry and the field of software security.

Although there are significant and ongoing efforts being made to develop machine learning and deep learning approaches for detecting dangerous and malicious threads and vulnerabilities that can severely compromise software security, this issue still remains open with several emerging problems involving SV detection and function identification methods that need rigorous and dedicated study.

The *first* challenging problem is how to *transfer efficiently the learning on SVs* from labelled projects to other unlabelled projects. This problem deals with the scarcity of labelled SVs in projects that require the laborious manual labelling of code by software security experts. Labelled vulnerable code is needed in order to train the models, but the process of labelling vulnerable source code is very tedious, error-prone and challenging even for domain experts. This has led to few labelled projects compared with the vast volume of unlabelled ones.

The *second* challenging problem is how to efficiently exploit the semantic and syntactic relationships inside source code to *detect SVs at a fine-grained level* (i.e., the statement level) than the function or program levels. This problem highlights code statements that are highly relevant to the corresponding SV. For most publicly available datasets, SVs are only labelled at the program or function levels, not at the statement level. In doing fine-grained SV detection, we can significantly speed up the process of isolating and detecting SVs, thereby reducing the time and cost involved.

The *third* challenging problem is how to leverage the information from binary programs to *deal with the function identification problem*, a preliminary step in binary analysis for many applications such as malware detection and binary instrumentation, to name a few. This problem seeks to obtain the optimal solutions to deal with all cases (e.g., the function scope identification problem) of the function identification problem.

The thesis aims to rigorously investigate and provide solutions to these three challenging problems by using recent advances in the deep learning field which have been demonstrated to be successful in some application domains such as computer vision and natural language processing. However, due to the complexity of software data, the application of deep neural networks to software security is not straight-forward and requires filling the gaps in understanding in order to propose novel deep learning models that fit the characteristics of the software data. Let us start the journey of bringing deep learning closer to software security.

Declaration

This thesis is an original work of my research and contains no material which has been accepted for the award of any other degree or diploma at any university or equivalent institution and that, to the best of my knowledge and belief, this thesis contains no material previously published or written by another person, except where due reference is made in the text of the thesis.

Signature:

Print Name: Van Khac Nguyen

Date: December, 2020.

Publications During Enrolment

Some parts of the thesis are extended or modified versions of conference and journal papers that have been published or are under review or are going to be submitted, as follows:

Chapter 3:

- © 2019 IEEE. Reprinted, with permission, from **Van Nguyen**, Trung Le, Tue Le, Khanh Nguyen, Olivier de Vel, Paul Montague, Lizhen Qu and Dinh Phung. "Deep Domain Adaptation for Vulnerable Code Function Identification". In The 2019 International Joint Conference on Neural Networks (IJCNN), July 2019.

Chapter 4:

- Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer. In The 24th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD). "Dual-Component Deep Domain Adaptation: A New Approach for Cross Project Software Vulnerability Detection". **Van Nguyen**, Trung Le, Olivier de Vel, Paul Montague, John Grundy and Dinh Phung, © 2020.

Chapter 5:

- **Van Nguyen**, Trung Le, Paul Montague, Olivier de Vel, John Grundy and Dinh Phung. "Information-theoretic Source Code Vulnerability Highlighting". ***Submitted*** to The 25th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), 2021.

Chapter 6:

- **Van Nguyen**, Trung Le, He Zhao, Paul Montague, Olivier de Vel, John Grundy and Dinh Phung. "Information-Theoretic End-to-End Models to Identify Code Statements Causing Software Vulnerability". ***To be submitted*** to The 38th International Conference on Machine Learning (ICML), 2021.

Chapter 7:

- Reprinted by permission from Springer Nature Customer Service Centre GmbH: Springer. In The 24th Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD).

“Code Action Network for Binary Function Scope Identification”. **Van Nguyen**, Trung Le, Tue Le, Khanh Nguyen, Olivier de Vel, Paul Montague, John Grundy and Dinh Phung, © 2020.

Chapter 8:

- © 2020 IEEE. Reprinted, with permission, from **Van Nguyen**, Trung Le, Tue Le, Khanh Nguyen, Olivier de Vel, Paul Montague and Dinh Phung. “Code Pointer Network for Binary Function Scope Identification”. In The 2020 International Joint Conference on Neural Networks (IJCNN), July 2020.

Chapter 9:

- **Van Nguyen**, Trung Le, Olivier de Vel, Paul Montague, Jon Arnold, John Grundy and Dinh Phung. “Cross Project Software Vulnerability Detection with Deep Domain Adaptation”. ***To be submitted*** to Information Software Engineering journal, 2021.

Another paper that does not directly constitute the thesis but has contributed to the knowledge that I have gained during my PhD research is the following:

Hung Nguyen, **Van Nguyen**, Thin Nguyen, Mark E Larsen, Bridianne O’Dea, Duc Thanh Nguyen, Trung Le, Dinh Phung, Svetha Venkatesh and Helen Christensen. “Jointly Predicting Affective And Mental Health Scores Using Deep Neural Networks Of Visual Cues On The Web”. ***Published*** in The 19th International Conference on Web Information Systems Engineering (WISE), 2018.

Acknowledgements

I am deeply appreciative of and grateful to my principal supervisor, Prof. Dinh Phung, for his teaching and general guidance that shaped my thesis and research methods. He has taught me many critical things in terms of theoretical knowledge, problem-solving skills and professional ways of working. I am also very thankful for his sympathy and support whenever I became stuck in my life.

I am forever grateful for the chance to work with and be guided by my co-supervisor, Dr. Trung Le, with his valuable suggestions, brilliant ideas and excellent writing, which I have always been striving for. I am deeply appreciative of his detailed and specific instructions. I am greatly indebted to him and his family. With them, I found my second family and my second home in Australia.

I strongly believe that I am extremely lucky to have such devoted supervisors. My life would have been much more difficult without their instruction and support.

My special thanks to Assoc. Prof. Gholamreza Haffari, Assoc. Prof. Vincent Lee and Dr. Mario Boley, who were the chair and panel members for my PhD research milestones at the Faculty of Information Technology at Monash University, for their helpful comments on my research and sharing of useful experiences.

I am also very thankful to my friend Mahmoud Ibrahim for his time in terms of sharing and talking with me whenever I felt stressed or needed help. Many thanks to my other friends in Prof. Dinh Phung's and Dr. Trung Le's research groups, who have made an encouraging workplace and environment. I enjoyed all the work and leisure time that we shared together.

Accredited professional editor Mary-Jo O'Rourke AE provided copyediting and proofreading services in certain parts of this thesis according to the national university-endorsed 'Guidelines for editing research theses' (Institute of Professional Editors 2019).

My thesis is dedicated to my beloved family: my Father, my Mother, my Sister and my Brothers who have always been there to support me and take care of me in every step that I have taken in my life.

Contents

List of Figures

List of Tables

Notations

$\boldsymbol{x}$	A vector value
$\boldsymbol{U}$, $\boldsymbol{V}$, $\mathbf{W}$, $\mathbf{H}$	Matrices
$\mathbb{R}$	The set of real numbers
$\mathbf{B}$, $\mathbf{D}$, I, $\mathbf{X}$	Sets of input data
S, $\mathbf{T}$	Source and target data domains
$\mathbf{Y}$	A set of targets or labels
X, $\widetilde{X}$, Y	Random variables (a scalar or vector)
$\mathcal{X}$	A value or event space
$\mathbf{W}^\top$	Transposition of the matrix $\mathbf{W}$
x	A scalar value
$D_{KL}(p\|\|q)$	Kullback-Leibler divergence of two distributions p and q
$\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$	Gradient of function $f(\boldsymbol{x})$ with respect to $\boldsymbol{x}$
$\boldsymbol{x} \sim p_{data}$	Data $\boldsymbol{x}$ drawn from the distribution p_{data}
$\mathbb{E}_{\boldsymbol{x}\sim p_{data}}(f(\boldsymbol{x}))$	Expectation of $f(\boldsymbol{x})$ with respect to $p_{data}(\boldsymbol{x})$
I	Identity matrix with dimensionality implied by context
$\mathcal{N}(0, I)$	Normal Gaussian distribution
$\mathcal{N}(\boldsymbol{x}, \boldsymbol{\mu}, \Sigma)$	Gaussian distribution over $\boldsymbol{x}$ with mean $\boldsymbol{\mu}$ and covariance Σ
$I(X, Y)$	Mutual information between two random variables X and Y
$H(X)$	Entropy of the random variable X
$\|\|\boldsymbol{x}\|\|$	L^2 norm of $\boldsymbol{x}$

Abbreviations

i.i.d	Independently and identically distributed
w.r.t	With respect to
Fig.	Figure
Eq.	Equation
SVs	Software vulnerabilities
SVD	Software vulnerability detection
DNNs	Deep feedforward neural networks
ANNs	Artificial neural networks
MLPs	Multi-layer perceptrons
CNNs	Convolutional neural networks
RNNs	Recurrent neural networks
NLP	Natural language processing
LSTMs	Long short-term memory networks
GRUs	Gated recurrent unit networks
Bi-RNNs	Bidirectional recurrent neural networks
VAEs	Variational autoencoders
GANs	Generative adversarial networks
RKHS	Reproducing kernel hilbert space
MK-MMD	Multiple kernel variant of maximum mean discrepancies
CDAN	Code domain adaptation network
VAP	Virtual adversarial perturbation
SCDAN	Semi-supervised code domain adaptation network
GD-DDAN	Generator-discriminator deep code domain adaptation network
GD-SDDAN	Generator-discriminator semi-supervised deep code domain adaptation network
DDAN	Deep code domain adaptation
ICVH	Information-theoretic code vulnerability highlighting
VCP	Vulnerability coverage proportion
VCA	Vulnerability coverage accuracy
S2CVH	Statement-grained source code vulnerability highlighting
CAN	Code action network
CPN	Code pointer network

Chapter 1

Introduction

Software security is a crucial and significant research problem in cybersecurity wherein all aspects of how to detect and analyse dangerous and malicious threats in order to secure given software are speculated on and studied. One of the most impactful problems in software security is software vulnerability detection (SVD), in which we need to develop automatic tools such as machine learning or deep learning approaches to specify whether a given piece of software has any vulnerabilities. For SVD, although approaches based on machine learning and deep learning can help to automate this task, those approaches demand rich datasets or software projects with vulnerability labels. Moreover, because the task of labelling software with vulnerability type is labor-intensive and involves the intervention of security experts with strong knowledge domain of software analysis, many real-world software projects currently are not labelled at all or only partly labelled. This raises a critical research question of how to transfer knowledge from machine learning and deep learning models trained on labelled software projects to achieve models that can accurately predict the vulnerability of software in unlabelled target software projects.

In addition to addressing the first research question of how to develop novel methods for transferring the knowledge learned from labelled software projects to unlabelled ones, this thesis also puts a focus on developing efficient and accurate deep learning-based methods for SVD at a fine-grained level in the second research question. The final research problem studied in this thesis is to develop deep learning-based methods inspired by recent advances in the deep learning field to target the problem of function scope identification in a given piece of binary software. This binary code analysis problem plays an important role as a preprocessing step whose outcomes can be used in other tasks including binary SVD and malware detection.

Although deep learning has achieved breakthrough results in multiple application domains such as visual object recognition [Krizhevsky et al., 2012] and language modelling [Sutskever et al., 2014a], the success of deep learning in software security and software-relevant tasks is still limited. To the best of our knowledge, this is one of the very first theses to study and propose novel deep learning-based methods with the aim of bringing deep learning closer to software security, one of the most important problems in cybersecurity.

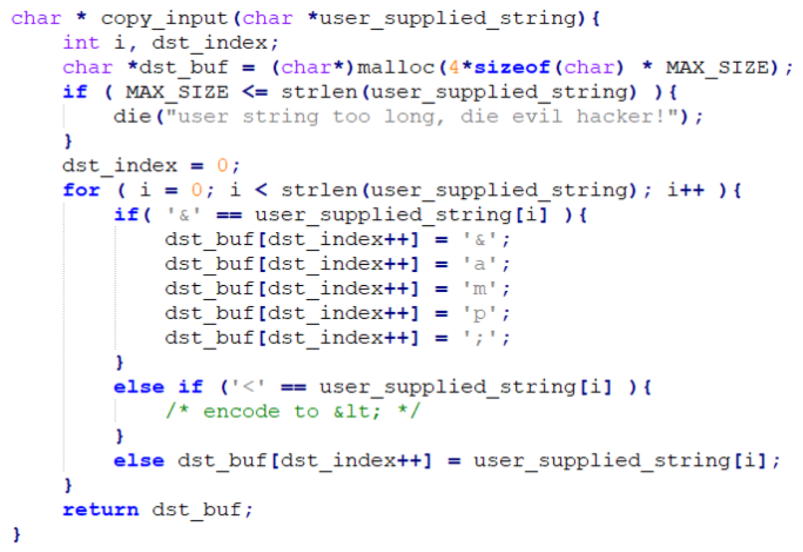
In the introduction chapter, we first present an overall view of software vulnerabilities (SVs) in the software industry and the field of computer security, as well as their potential to cause serious damage to a nation's economy and people's lives. We then review the main approaches for the SVD problem in terms of their strengths and drawbacks, followed by the scope and aims of the thesis. Finally, we summarise the significance and the main contributions of the thesis.

1.1 Software vulnerabilities

Computer software plays a significant role in modern life. A large number of technologies and software methodologies have been employed to develop enormous varieties of computer software on different platforms (e.g., Linux, Windows, Google's Android and Apple's iOS) from very simple applications to complex enterprise software systems. Because of the variety of computer software as well as the diversity in its development processes, a great deal of computer software faces SVs, which can be exploited by hackers or vandals leading to severe and serious economic damage.

In the field of software security, SVs are specific potential flaws, glitches, weaknesses or oversights in software. Attackers can leverage these vulnerabilities to carry out malicious actions, such as exposing or altering sensitive information and disrupting/destroying/taking control of a system/program [Dowd et al., 2006]. Fig. 1.1 shows an example of a software vulnerability (i.e., a heap-based buffer overflow error) in a C/C++ source code function. In particular, this source code aims to encode a user's input string with certain characters. The programmer assumes that the encoding expansion process only expands a given character by a factor of 4 (in the declaration); however, the encoding of each ampersand expands by 5 (in the implementation). As a consequence, when the encoding procedure expands a string, it is possible to overflow the destination buffer if attackers provide a string with many ampersands.

Although much effort has been devoted and many solutions have been proposed for SVD, the number of SVs and the severity of the threat imposed by them have gradually increased and caused considerable damage to individuals and companies [Ghaffarian and Shahriari, 2017]. For example, SVs in popular browser plugins have threatened the security and privacy of millions



```
char * copy_input(char *user_supplied_string){
    int i, dst_index;
    char *dst_buf = (char*)malloc(4*sizeof(char) * MAX_SIZE);
    if ( MAX_SIZE <= strlen(user_supplied_string) ){
        die("user string too long, die evil hacker!");
    }
    dst_index = 0;
    for ( i = 0; i < strlen(user_supplied_string); i++ ){
        if( '&' == user_supplied_string[i] ){
            dst_buf[dst_index++] = '&';
            dst_buf[dst_index++] = 'a';
            dst_buf[dst_index++] = 'm';
            dst_buf[dst_index++] = 'p';
            dst_buf[dst_index++] = ';';
        }
        else if ('<' == user_supplied_string[i] ){
            /* encode to &lt; */
        }
        else dst_buf[dst_index++] = user_supplied_string[i];
    }
    return dst_buf;
}
```

Figure 1.1: Example of a heap-based buffer overflow error in a C/C++ source code function. (source: https://cwe.mitre.org/data/definitions/122.html).

of internet users (e.g., Oracle Java (US-CERT 2013) and Adobe Flash Player (US-CERT 2015; Adobe Security Bulletin 2015)). Additionally, SVs in fundamental and popular open-source software have also threatened the security of thousands of companies as well as their customers around the globe (e.g., ShellShock (Symantec Security Response 2014), Heartbleed (Codenomicon 2014) and Apache Commons (Breen 2015)).

Furthermore, due to the rapid growth of computer software, potential SVs have become universal in software development and deployment processes, creating severe threats to cybersecurity, and leading to costs of about USD 600 billion globally each year [McAfee and CSIS, 2017]. In addition, the quantity of reported SVs has soared for each vulnerability type (e.g., Overflow or Denial of Service (DoS)) year by year. Summary statistics on vulnerabilities by type are given in Table 1.1 and Fig. 1.2. These threats create an urgent need for automatic tools and methods to efficiently and effectively deal with a large amount of vulnerable code with a minimal level of human intervention.

1.2 Software vulnerability detection

Typical approaches proposed to the SVD problem include those based on machine learning and deep learning methods, as depicted in Fig. 1.3. All proposed approaches have their strengths and drawbacks.

Machine learning-based methods. Techniques based on machine learning such as data structures [Cozzie et al., 2008, White and Lüttgen, 2013], program structures [Brumley et al.,

Year	# Vulnerabilities	DoS	Code Execution	Overflow	Memory Corruption	Sql Injection	XSS	Directory Traversal	Http Response Splitting	Bypass something	Gain Information	Gain Privileges	CSRF	File Inclusion	# exploits
1999	894	177	112	172			2	7		25	16	103			2
2000	1020	257	208	206		2	4	20		48	19	139			
2001	1677	403	403	297		7	34	123		83	36	220		2	2
2002	2156	498	553	435	2	41	200	103		127	74	199	2	14	1
2003	1527	381	477	371	2	49	129	60	1	62	69	144		16	5
2004	2451	580	614	410	3	148	291	110	12	145	96	134	5	38	5
2005	4935	838	1627	657	21	604	786	202	15	289	261	221	11	100	15
2006	6610	893	2719	663	91	967	1302	322	8	267	271	184	18	849	30
2007	6520	1101	2601	953	95	706	884	339	14	267	323	242	69	700	44
2008	5632	894	2310	699	128	1101	807	363	7	288	270	188	83	170	74
2009	5736	1035	2185	700	188	963	851	322	9	337	302	223	115	138	735
2010	4652	1102	1714	680	342	520	605	275	8	234	282	238	86	73	1493
2011	4155	1221	1334	770	351	294	467	108	7	197	409	206	58	17	557
2012	5297	1425	1458	843	423	242	758	122	13	343	389	250	166	14	615
2013	5191	1454	1186	859	366	156	650	110	7	352	511	274	123	1	205
2014	7946	1598	1574	850	420	305	1105	204	12	457	2104	239	264	2	401
2015	6480	1792	1825	1079	749	217	778	150	12	577	748	367	248	5	127
2016	6447	2029	1494	1326	717	94	497	99	15	444	843	600	87	7	1
2017	14714	3154	3004	2495	745	508	1518	279	11	629	1639	459	327	18	6
2018	16556	1853	3041	2368	400	517	2042	531	11	708	1424	247	461	31	4
2019	12174	919	2277	1247	296	410	1593	280	4	495	900	129	398	40	
Total	122774	23603	32718	18081	5339	7853	15303	4130	166	6375	10989	5006	2521	2235	4333
% Of All		19.2	26.6	14.7	4.3	6.4	12.5	3.4	0.1	5.2	9	4.1	2.1	1.8	

Table 1.1: Statistics on vulnerabilities by type (source: www.cvedetails.com/vulnerabilities-by-types.php).

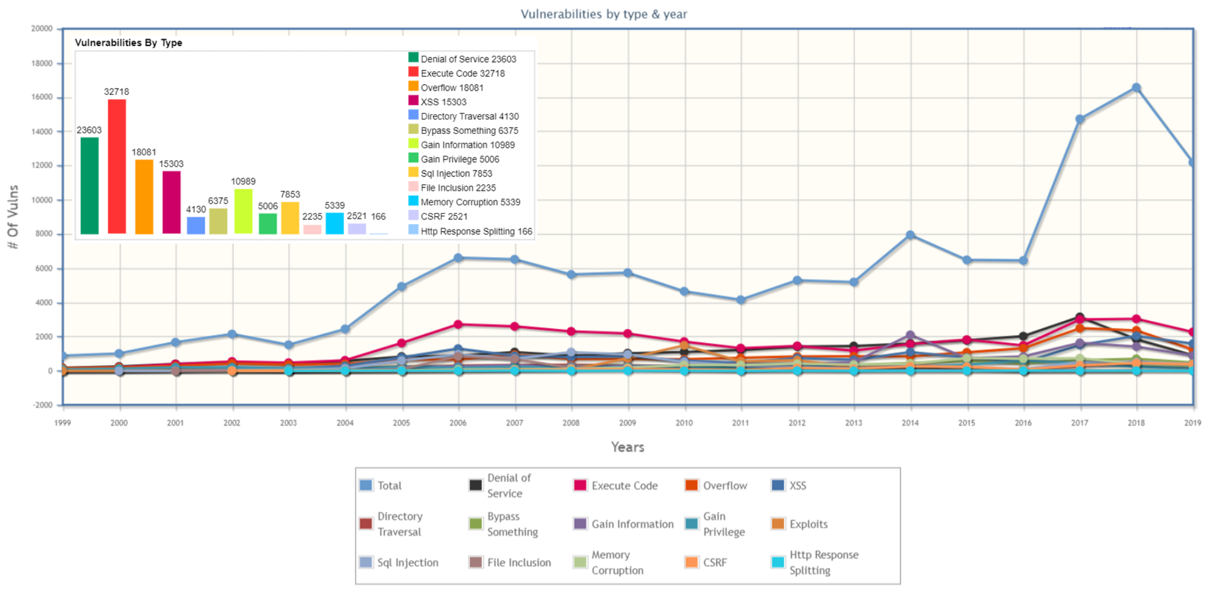


Figure 1.2: Trends of vulnerability types by year (source: www.cvedetails.com/vulnerabilities-by-types.php).

2011], dynamic analysis [Grieco et al., 2016] and symbolic execution [Cadar and Sen, 2013, Avancini and Ceccato, 2013, Meng et al., 2016] play important roles in the SVD problem at the binary code level for cases in which the corresponding source code is not always available. These methods bring the promise of automated work, which can potentially learn generalisations about malware; however, there are few machine learning-based malware detection approaches

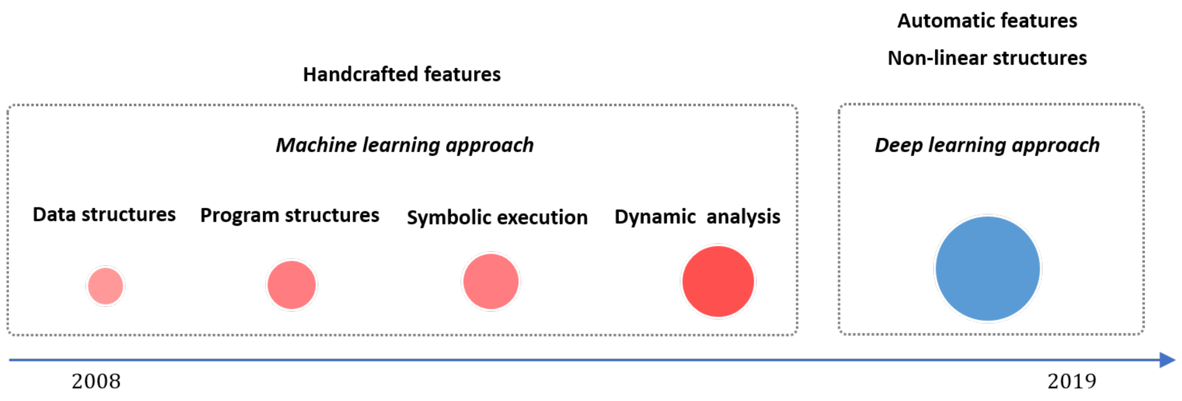


Figure 1.3: Estimated timeline of typical machine learning-based and deep learning-based methods proposed for the SVD problem.

that have low false positive rates and high scalability [Saxe and Berlin, 2015].

Machine learning techniques have also been widely used to create SVD methods at the source code level. In particular, patterns and features (e.g., software metrics [Chidamber and Kemerer, 1994], code churn metrics [Nagappan et al., 2006] or code measures [Ostrand et al., 2004]) representing software source codes are selected as the input for SVD methods. However, these features and patterns cannot figure out and distinguish code regions of different semantics in many cases. Furthermore, some handcrafted features that can perform effectively in one project may not perform well in other projects [Zimmermann et al., 2009] due to poor generalisation.

Deep learning-based methods. The rapid rise of deep learning is in part due to its ability to learn feature representations and complex non-linear structures in datasets. Although deep learning has undergone a renaissance in the past few years and achieved breakthrough results in multiple application domains such as visual object recognition [Krizhevsky et al., 2012], language modelling [Sutskever et al., 2014b] and speech recognition [Hinton et al., 2012], the application of deep learning to cybersecurity is still at an early stage when applied to larger, more complex and mixed datasets.

Some recent works [Li et al., 2018, Dam et al., 2017, Peng et al., 2015, Saxe and Berlin, 2015, Raff et al., 2017] have advanced the application of deep learning for the SVD problem and surpassed SVD methods based on machine learning and data mining [Li et al., 2017] techniques at both binary and source code levels; however, they themselves encounter some critical limitations as follows:

1. They cannot transfer efficiently the learning on SVs obtained from labelled projects to unlabelled projects (i.e, labelled and unlabelled projects may come from different data domains).

2. They cannot detect SVs at a fine-grained and flexible level (i.e., the code statement level), only at the function or program levels.

In computer security, we often encounter situations where source code is not available and only binaries are accessible. In these situations, binary analysis is an essential tool enabling many applications such as malware detection and common vulnerability detection [Perkins et al., 2009]. In binary analysis, function identification is usually the first step and aims to specify function scope in binary programs. In both binary analysis and function identification, tackling the loss of high-level semantic structures in binaries which results from compilers during the process of compilation is likely the most challenging problem. There have been many effective methods for dealing with the function identification problem from heuristic solutions [Kruegel et al., 2004] to complex approaches employing machine learning (e.g., ByteWeight [Bao et al., 2014] and Nucleus [Andriesse et al., 2017]) or deep learning techniques (e.g., [Shin et al., 2015]). These methods have shown promising performance; however, they still encounter the following issue:

3. They cannot address the function scope identification problem, the toughest and most essential sub-problem in the function identification problem, wherein the scope (i.e., the indexes or addresses of all machine instructions in a function) of each function must be specified.

1.3 Scope and aims

This thesis aims to leverage the potential and promising abilities of the deep learning approach to address the shortcomings, mentioned in Section 1.2, existing in most current methods for effectively and efficiently tackling the SVD and function identification problems. In particular, we propose several novel deep learning-based methods for addressing the three following research questions:

(Q.1) How to *transfer efficiently the learning on software vulnerabilities* from labelled projects (i.e., source domains) to other unlabelled projects (i.e., target domains).

(Q.2) How to efficiently exploit the semantic and syntactic relationships inside source code to *detect vulnerabilities at a fine-grained level with more flexible scope* (i.e., the code statement level) than the function or program levels.

(Q.3) How to leverage the information from binaries (i.e., byte instructions) and assemblies (i.e., machine instructions) programs to *deal with all cases* (i.e., the function start identification, function end identification, function boundary identification and function scope identification problems) *of the function identification problem*, especially the function scope identification problem, the toughest and most essential problem.

1.4 Significance

The significance of the thesis is structured around three central lines of work to address the three aforementioned problems remaining in existing current work on software security by answering the three research questions (i.e., Q.1, Q.2 and Q.3) mentioned in Section 1.3. It is worth noting that to the best of our knowledge, this is one of the first thesis projects studying how to leverage the power of deep learning to solve the aforementioned critical and urgent problems in software security which can cause serious and severe damage to a nation's economy and people's lives. This thesis serves as a bridge to connect deep learning approaches proven to be successful in some application domains to software security. The initial success of the approaches that we have proposed in this thesis demonstrates the potential of applying deep learning approaches to real-world problems of software security. we summarise these problems and the significance of the thesis as follows:

Firstly, one of the most crucial issues of the SVD is dealing with the scarcity of labelled vulnerabilities in projects that require laborious manual labelling of code by software security experts. Labelled vulnerable code is needed to train the models and the process of labelling vulnerable source code is very tedious, time-consuming, error-prone and challenging even for domain experts. This has led to few labelled projects compared with the vast volume of unlabelled ones. The first research question of the thesis (i.e., how to efficiently transfer the learning on SVs from labelled projects to other unlabelled projects) aims to address this issue of the SVD problem. Our research is one of the first works proposing deep learning-based methods for the transfer learning problem in SVD.

Secondly, despite their promising performance, current deep learning-based methods are only able to detect SVs at the function-[Lin et al., 2018, Li et al., 2018] or program-[Dam et al., 2017] levels. However, in real-world situations, programs or even functions can consist of hundreds or thousands of code statements and the source of most vulnerabilities often arises from a significantly smaller scope, usually a few core statements. The second research question of the thesis (i.e., how to efficiently exploit the semantic and syntactic relationships inside source codes to detect vulnerabilities at a fine-grained level with more flexible scope than the function or program levels) aims to detect SVs at a fine-grained level, i.e., several code statements within functions or programs. This includes highlighting statements that are highly relevant to the corresponding vulnerability of associated code statements. For most publicly available datasets, vulnerabilities are only labelled at the program or function levels, not at the code statement level. In doing this, we can then significantly speed up the process of isolating and detecting SVs, thereby reducing the time and cost involved. Our research is one of the first works using deep learning approaches to discover effective solutions to detect SVs at the code statement level within functions or programs in SVD.

Finally, function identification is a preliminary step in binary analysis for many applications from malware detection to common vulnerability detection and binary instrumentation, to name a few. However, existing function identification methods [Bao et al., 2014, Shin et al., 2015, Andriesse et al., 2017] cannot address the function identification problem effectively, especially for the hardest one, the function scope identification problem. The third research question of the thesis (i.e., how to leverage the information from binaries and assemblies to deal with all cases of the function identification problem) aims to find the solutions to deal with all cases (i.e., the function start, function end, function boundary and function scope identification problems) of the function identification problem of binary analysis in SVD. Our research introduces some effective remedies that are the answers for solving all cases of the function identification problem.

1.5 Organisation and contributions of the thesis

The main contributions of this thesis are summarised in three main parts (i.e., Parts I, II and III) which reflect and aim to answer the three research questions (i.e., Q.1, Q.2 and Q.3). In what follows, we summarise the organisation and content of each part.

Part I "Deep Domain Adaptation for Software Vulnerability Detection" aims to address the problem relevant to the first research question (Q.1).

Part I presents novel deep learning-based approaches proposed to address the first research question (Q.1) and consists of Chapters 3 and 4. Chapter 3 introduces the work to tackle the problem of transfer learning (aka domain adaptation) from a labelled software project to another unlabelled software project. To the best of our knowledge, this is the first work that has studied deep domain adaptation for SVD. The contribution of this chapter is to propose a novel architecture named the Code Domain Adaptation Network (CDAN) and a novel approach named the Semi-supervised Code Domain Adaptation Network (SCDAN) that is based on the skeleton of CDAN for this specific problem.

Chapter 4 addresses the same problem of the transfer learning for SVD but from a deeper angle in which we first identify the existing drawbacks of the previous work mainly based on the generative adversarial network (GAN) principle which inherently suffers from the mode collapsing problem. The approach we propose in Chapter 4 can tackle the inherent missing mode and boundary distortion problems of GANs, hence significantly improving the first approach mentioned in Chapter 3.

Part II "Learning to Explain Software Vulnerability" aims to address the problem relevant to the second research question (Q.2).

Part II presents another piece of work regarding proposing novel learn-to-explain approaches that can assist us in performing SVD at a fine-grained level. The motivation comes from our observation that a given source code section (i.e., a function or program) can consist of hundreds or thousands of lines of code statements, but only a few of them might be the main source of vulnerabilities. This raises a question regarding whether we can employ or develop learn-to-explain deep learning-based approaches to identify code statements that really contribute to the decision on vulnerabilities and highlighted code statements that are really relevant to the vulnerable code statements in the given source code section. This way of thinking is reasonable because we believe that what the model bases its predictions on concurs with the natural vulnerable characteristics of the source code given the fact that the model learns those characteristics from a mixed variety of source code.

Our contributions to this line of thinking include two novel deep learning-based approaches proposed in Chapters 5 and 6 for which we note that this is the first time the desire for fine-grain-level SVD has been proposed and addressed successfully to some extent. Briefly, the approach we propose in Chapter 5 originates from a novel information-bound formula on which we rely to devise further technical components. Moreover, the novel information-bound formula has been developed to more efficiently exploit the sequential nature of source codes. The second approach, presented in Chapter 6, goes even further in terms of theory in which we leverage the information bottleneck theory [Tishby et al., 2000, Tishby and Zaslavsky, 2015] to the novel information-bound formula for the problem of interest, which then serves as a regularisation term to support the automatic inference of the number of selected vulnerable code statements.

Part III "Deep Sequence-to-sequence Models for Function Scope Identification in Binary Programs" aims to address the problem relevant to the third research question (Q.3).

Part III addresses the third research question and is presented in Chapters 7 and 8. For Part III, we focus on the function scope identification problem for binary software which is one of the most important tasks in binary analysis. More specifically, we have proposed two novel learning-based approaches to tackle the problem. The first approach is developed based on the spirit of pointer networks [Vinyals et al., 2015b], while the second approach, named the Code Action Network, is inspired by the spirit of a Turing machine [Turing, 1938] in which the underlying idea is to equivalently transform the task of function scope identification (i.e., the hardest problem in the function identification problem) into the learning of a sequence of action states including NI (next inclusion), NE (next exclusion) and FE (function end) corresponding to byte instructions (at the byte level) or machine instructions (at the machine instruction level) of binary programs.

The two proposed approaches, especially the Code Action Network, currently achieve state-of-the-art-performance and surpass both other existing approaches based on machine learning and deep learning. This line of work can be leveraged along with the work mentioned in Part II to fulfill real-world binary SVD in which complex binary software is inputted to the system and subsequently split into many binary functions on which we run fine-grain-level detection to highlight the machine instructions likely causing vulnerabilities. It is worth noting that binary SVD is both more useful and more challenging than source code SVD from a practical perspective.

An overall view of the contributions of the thesis is depicted in Fig. 1.4.

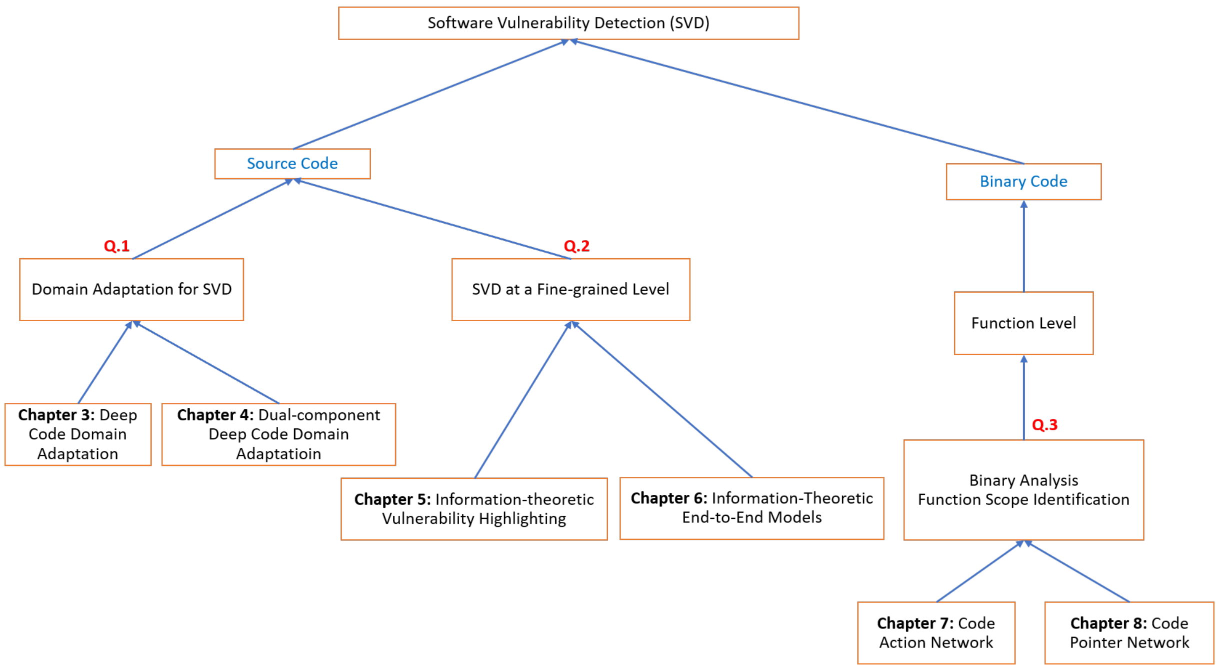

Software Vulnerability Detection (SVD)
Source Code
Binary Code
Q.1
Q.2
Domain Adaptation for SVD
SVD at a Fine-grained Level
Function Level
Chapter 3: Deep Code Domain Adaptation
Chapter 4: Dual-component Deep Code Domain Adaptatioin
Q.3
Chapter 5: Information-theoretic Vulnerability Highlighting
Chapter 6: Information-Theoretic End-to-End Models
Binary Analysis Function Scope Identification
Chapter 7: Code Action Network
Chapter 8: Code Pointer Network

Figure 1.4: Overall view of the thesis on the software vulnerability detection (SVD) problem; the names of all mentioned chapters are simplified for the purpose of visualisation.

Chapter 2

Related Background

In this chapter, we review the related background to the thesis. As previously stated, the thesis mainly focuses on leveraging the potential and promising abilities of deep learning approaches to effectively and efficiently tackle the software vulnerability detection (SVD) problem. We first briefly give an introduction to deep learning in terms of some typical popular architectures, namely, deep feedforward neural networks (DNNs) and convolutional neural networks (CNNs), as well as reviewing information theory (e.g., entropy, mutual information and information bottlenecks). We then provide a fundamental view of deep sequence models in deep learning such as recurrent neural networks (RNNs), bidirectional recurrent neural networks (bi-RNNs) and pointer networks that are specialised and extremely useful for working with sequence data (e.g., source code, time series and text). Finally we discuss deep generative models (e.g., variational autoencoders (VAEs) and generative adversarial networks (GANs)), deep domain adaptation (e.g., using the GAN principle, Wasserstein distance and virtual adversarial training), and spectral graphs for semi-supervised deep domain adaptation.

2.1 Introduction to deep learning

2.1.1 Deep feedforward neural networks

Artificial neural networks (ANNs) were inspired by the idea of building an intelligent machine that can imitate the human brain's architecture. ANNs are at the core of deep learning. They are versatile, powerful and scalable, making them capable of tackling large and highly complex machine learning tasks such as visual object recognition and classification (i.e, classifying billions of images, ImageNets [Russakovsky et al., 2014]), language modelling (e.g., Google translation), SVD, speech recognition (e.g., Apple's Siri), recommendation systems (e.g., Youtube) and self-learning machines (e.g., DeepMind's AlphaGo [Silver et al., 2017], which can beat the human world champion in the game Go by examining millions of past games and then playing against itself).

The standard well-known architectures of ANNs are deep feedforward neural networks (DNNs) (i.e., multi-layer perceptrons (MLPs)). An MLP is composed of one input layer, one or more hidden layers and one output layer. Every layer is fully connected to the next layer (i.e., connection from the input layer to the first hidden layer, from the hidden layer to the next hidden layer, and from the last hidden layer to the output layer). An example architecture of an MLP is depicted in Fig. 2.1.

The DNN shown in Fig. 2.1 has one input layer (i.e., contains input data) with three neurons $\boldsymbol{x} = (x_1, x_2, x_3)$, two hidden layers where the first hidden layer contains four neurons $\boldsymbol{h}^1 = (h^1_1, h^1_2, h^1_3, h^1_4)$ and the second hidden layer has three neurons $\boldsymbol{h}^2 = (h^2_1, h^2_2, h^2_3)$, and one output layer with two neurons $\boldsymbol{o} = (o_1, o_2)$. Depending on different problems (e.g., traffic sign recognition or digit classification), we may have different numbers of neurons in each layer (i.e., the input layer, hidden layers and output layer). For example, in handwritten digit classification (i.e., each digit is an image having 28×28 pixels), the number of neurons in the input layer should be $28 \times 28 = 756$, which is equal to the dimension of the flattened vector of each image. The number of neurons in two hidden layers can be in $\{128, 256, 512\}$ for instance, while the number of neurons in the output layer should be 10 (i.e., handwritten digits go from 0 to 9).

We compute the values of the hidden layers and output layer of the network depicted in Fig. 2.1 as follows:

$$\begin{aligned} \boldsymbol{h}^1 &= \sigma(\mathbf{W}_{xh}^\top \boldsymbol{x} + \boldsymbol{b}) \\ \boldsymbol{h}^2 &= \sigma(\mathbf{W}_{hh}^\top \boldsymbol{h}^1 + \boldsymbol{c}) \\ \boldsymbol{o} &= \phi(\mathbf{W}_{ho}^\top \boldsymbol{h}^2 + \boldsymbol{d}) \end{aligned}$$

where $\mathbf{W}_{xh}$, $\mathbf{W}_{hh}$ and $\mathbf{W}_{ho}$ are the weight matrices from the input layer to the first hidden layer, from the first hidden layer to the second hidden layer and from the second hidden layer to the output layer, while $\boldsymbol{b}$, $\boldsymbol{c}$ and $\boldsymbol{d}$ are the bias vectors of the first hidden layer, the second hidden layer and the output layer respectively. We often use the ReLU function for σ and the softmax function for ϕ in classification problems.

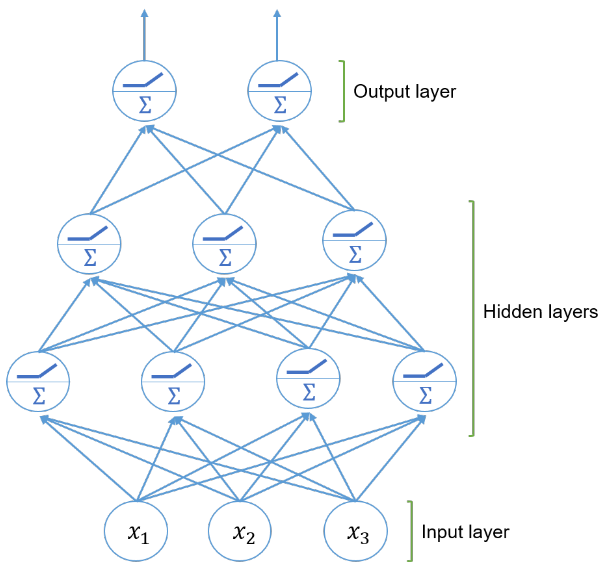


Figure 2.1: Architecture of a deep feedforward neural network with three neurons at the input layer followed by two hidden layers with four and three neurons respectively, while there are two neurons at the output layer.

Training DNNs. In 1986, Rumelhart et al. proposed an idea for training DNNs called back-propagation [Rumelhart et al., 1986] that is described as gradient descent today.

In the back-propagation algorithm, in the first phase, for each mini-batch of the training data the backpropagation algorithm first makes corresponding predictions (the forward pass goes through from the input layer to hidden layers and then to the output layer). The results from the output layer are then evaluated using an objective function (i.e., a cost or loss function). In the second phase, the gradients of the cost function then go through each layer in the network in reverse to measure the error contribution from each connection (reverse pass) and finally slightly tweak the connection weights to reduce the error (gradient descent step).

There are two well-known problems, namely, the vanishing gradients and exploding gradients problems, faced in training DNNs. In the training process, in some cases gradients get smaller and smaller when moving down to the lower (previous) layers. As a result, a gradient descent update leaves the lower layer connection weights almost unchanged and the training process hardly converges to an optimal solution. This problem is called the vanishing gradients problem. In other cases, in the training process the gradients get larger and larger when moving back to the lower layers so many layers get large weight updates and the training process becomes divergent. This is the exploding gradients problem.

Around 2010, especially in the research proposed by Glorot and Bengio [Glorot and Bengio, 2010], the authors mentioned that using a good initialisation strategy for model weight parameters and using appropriately corresponding activation functions can significantly alleviate the vanishing gradients and exploding gradients problems.

2.1.2 Convolutional neural networks

Convolutional neural networks (CNNs) [LeCun et al., 1998] are a kind of DNNs that are specialised for processing image data. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution which is a specialised kind of linear operation. CNNs are DNNs that use convolution in place of general matrix multiplication in at least one of their layers.

CNNs have achieved breakthrough performance on many complex visual tasks [Krizhevsky et al., 2012, Tran et al., 2015] such as visual object recognition, self-driving cars, automatic video classification systems and more. CNNs also successfully show high performance on other tasks such as voice recognition and natural language processing (NLP).

In 1998, Lecun and colleagues [LeCun et al., 1998] introduced a typical famous architecture of a CNN named LeNet-5 as depicted in Fig. 2.2. This architecture shows some common building blocks often used in CNNs including fully connected layers (i.e., as used in DNNs), convolutional layers and pooling layers.

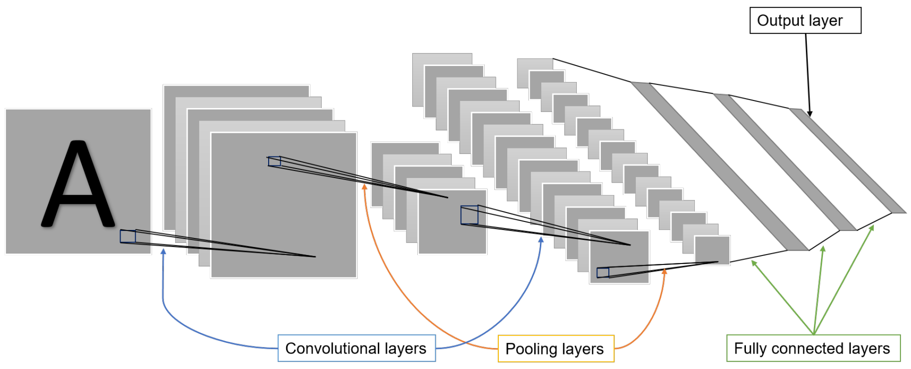


Figure 2.2: Architecture of LeNet-5 (a convolutional neural network) with some building blocks including convolutional layers, pooling layers and fully connected layers.

Convolutional layers. Convolutional layers as shown in Fig. 2.2 are the most important part of a CNN. The neurons in the first convolutional layer are connected to pixels in small rectangles (i.e., the receptive fields) of the input layer. This principle is kept for the connection between other layers (e.g., the connection between the second convolutional layer and the first convolutional layer). The neurons in the second convolutional layer are also connected only some neurons in the first convolutional layer in the receptive fields. This architecture allows the network to concentrate on small low-level (i.e., general) features in some first hidden layers, then assemble them into larger higher-level (i.e., specific) features in some next hidden layers.

In particular, a neuron located in row i and column j of a given layer is connected to the outputs of some neurons in the previous layer located in rows i to $i + f_h - 1$ and columns j to $j + f_w - 1$, where f_h and f_w are the height and width of the receptive field. If we desire to obtain a layer that has the same height and width as the previous layer, we need to add zeros around the inputs, as shown in Fig. 2.3 (left-hand figure). This is called zero padding. Furthermore, we can also gain a much smaller layer by spacing out the receptive fields and the shift from one receptive field to the next is called the stride as depicted in Fig. 2.3 (right-hand figure). In this case, a neuron located in row i and column j in the upper layer is connected to the outputs of the neurons in the previous layer located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, where s_h and s_w are the vertical and horizontal strides.

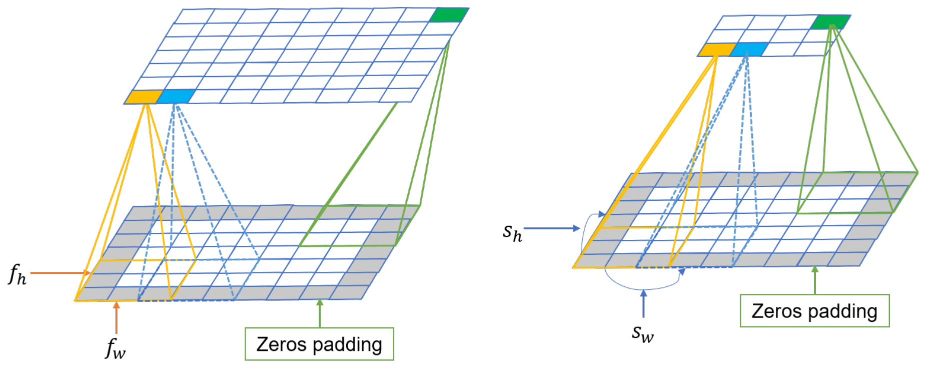


Figure 2.3: Connections between layers in a CNN with zero padding (left-hand figure) and using stride where the vertical and horizontal strides are equal to 2 (right-hand figure).

Filters and feature maps. In reality, a convolutional layer often applies multiple filters (i.e, trainable filters and each filter outputting one feature map) to gain multiple corresponding feature maps, making the convolutional layer capable of detecting multiple features from its inputs. In particular, a neuron located in row i and column j of the feature map k in a given convolutional layer l is connected to the outputs of the neurons in the previous layer $l - 1$, located in rows $i \times s_h$ to $i \times s_h + f_h - 1$ and columns $j \times s_w$ to $j \times s_w + f_w - 1$, across all feature maps (in layer $l - 1$). We note that all neurons in the same row i and column j but in different feature maps of a convolutional layer are connected to the outputs of the exact same neurons in the previous layer.

We can summarise the preceding process by using Eq. (2.1) to further demonstrate how to compute the output of a given neuron of a convolutional layer.

$$o_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{k'}-1} o'_{i\times s_h+u, j\times s_w+v, k'} \times w_{u,v,k',k} \tag{2.1}$$

where

- $o_{i,j,k}$ is the output in row i and column j in the feature map k of convolutional layer l.
- $o'_{i\times s_h+u, j\times s_w+v, k'}$ is the output in row $i \times s_h + u$ and column $j \times s_w + v$ in the feature map k' of the previous convolutional layer $l - 1$.
- $w_{u,v,k',k}$ is the connection weight (i.e., it is updated during the training process) between any neuron in the feature map k of convolutional layer l and its inputs in row u and column v of the corresponding receptive field and feature map k' of the previous convolutional layer $l - 1$ while b_k is the bias of the feature map k of convolutional layer l.

Pooling layers. The main goal of the pooling layers is to subsample (i.e., shrink) the input convolutional layer in order to reduce the computational cost. Pooling layers not only reduce the memory usage, but also reduce the number of parameters, so eliminating the risk of overfitting. Pooling layers have no weights and they only aim to aggregate the inputs using an aggregation function such as the max or mean. We also need to define the size, the stride and the padding type (i.e., zero padding or not padding) for the pooling layers as for the convolutional layers.

2.1.3 Information theory

2.1.3.1 Entropy

We denote X as a random variable if it takes on values from a set of possible values $\mathcal{X}$ with specified probabilities. Entropy is a measure of the uncertainty of a random variable. It aims to measure the averaged amount of information required in order to be able to describe the random variable. Assume that we have a discrete random variable X and a probability mass function $p(x) = p(X = x), x \in \mathcal{X}$ where $\mathcal{X}$ often stands for the event space and seeing any x from $\mathcal{X}$ as observing the event $X = x$. The entropy of $H(X)$ of a discrete random variable X is defined as:

$$H(X) = -\sum_{x\in\mathcal{X}} p(x)\text{log}p(x) \tag{2.2}$$

where the log is to base 2 and the entropy is expressed in bits. The entropy is the number of bits on average required to describe the random variable.

We can further extend the definition of a single random variable as described in Eq. (2.2) to a pair of random variables. Assuming that we have two discrete random variables X and Y drawn from a joint distribution $p(x, y)$, the joint entropy $H(X, Y)$ is defined as:

$$H(X,Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\text{log}p(x,y)$$

Furthermore, we can also define the conditional entropy of a random variable given another random variable. That can be considered the expected value of the entropies of the conditional distributions that is averaged over the conditioning random variable. Assuming that we have two discrete random variables X and Y drawn from a joint distribution $p(x, y)$, the conditional entropy $H(Y|X)$ is defined as:

$$\begin{aligned} H(Y|X) &= \sum_{x\in\mathcal{X}} p(x)H(Y|X=x) \\ &= -\sum_{x\in\mathcal{X}} p(x)\sum_{y\in\mathcal{Y}} p(y|x)\text{log}(y|x) \\ &= -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\text{log}(y|x) \end{aligned}$$

2.1.3.2 Relative entropy and mutual information

Relative entropy. The term "relative entropy" is often used to represent the measure of two distributions. Assume that we have two different distributions p and q with two corresponding probability mass functions $p(x)$ and $q(x)$. The relative entropy, also called the Kullback–Leibler distance, is defined as follows:

$$D_{KL}(p||q) = \sum_{x\in\mathcal{X}} p(x)\text{log}\frac{p(x)}{q(x)} \tag{2.3}$$

where we use some conventions in which $0\text{log}\frac{0}{0} = 0$, $0\text{log}\frac{0}{q} = 0$ and $p\text{log}\frac{p}{0} = \infty$. We note that if there is any value $x \in \mathcal{X}$ such that $p(x) > 0$ and $q(x) = 0$ then $D_{KL}(p||q) = \infty$.

From Eq. (2.3), we can see that the Kullback–Leibler distance is not symmetrical (i.e., $D_{KL}(p||q)$ $\neq D_{KL}(p||q)$) and does not satisfy the triangle inequality (i.e., $D_{KL}(r||p) \leq D_{KL}(q||p) + D_{KL}(r||q)$). These problems are known as the drawbacks of the Kullback–Leibler distance and prevent it from being an optimal measure of the true distance between two distributions.

Mutual information. The term "mutual information" is used for measuring the dependence between two random variables. Mutual information captures how much the knowledge of one random variable reduces the uncertainty of the other.

Assume that we have two random variables X and Y drawn from the joint distribution $p(x,y)$ with two corresponding marginal distribution $p(x)$ and $p(y)$. The mutual information between X and Y denoted by $I(X,Y)$ is the relative entropy between the joint distribution $p(x,y)$ and the product distribution $p(x)p(y)$, and is defined as follows:

$$\begin{aligned} I(X,Y) &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} \\ &= D_{KL}(p(x,y)||p(x)p(y)) \end{aligned} \tag{2.4}$$

The relationship between entropy and mutual information. The formula for the mutual information between two random variables X and Y described in Eq. (2.4) can be further derived as follows:

$$\begin{aligned} I(X,Y) &= \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} \\ &= \sum_{x,y} p(x,y)\log\frac{p(x,y)}{p(x)p(y)} \\ &= \sum_{x,y} p(x,y)\log\frac{p(x|y)}{p(x)} \\ &= -\sum_{x,y} p(x,y)\log p(x) + \sum_{x,y} p(x,y)\log p(x|y) \\ &= -\sum_{x} p(x)\log p(x) + \sum_{x,y} p(x,y)\log p(x|y) \\ &= H(X) - H(X|Y) \end{aligned} \tag{2.5}$$

From Eq. (2.5), we can conclude that mutual information $I(X,Y)$ can be used to denote the reduction in the uncertainty of X using the knowledge of Y. The mutual information $I(X,Y)$ is always larger than or equal to 0 (i.e., $I(X,Y) \geq 0$). The mutual information $I(X,Y)$ obtains the largest value equal to $H(X)$ if and only if X is completely determined by Y while achieving the smallest value equal to 0 if and only if X and Y are independent.

2.1.3.3 Information bottlenecks

An information theory of deep learning. In the supervised learning problem of deep learning, we often aim to predict corresponding outputs (e.g., labels) $\{\boldsymbol{y}_i\}_{i=1}^n$ given inputs $\{\boldsymbol{x}_i\}_{i=1}^n$. A DNN will learn some latent representations (i.e., latent features in a latent space that contain useful information to describe the data) $\{\widetilde{\boldsymbol{x}}_i\}_{i=1}^n$ of the input data in terms of enabling good predictions and generalisations.

Assume that the whole hidden layer in Fig. 2.4 is denoted by a random variable $\widetilde{X}$ while the input layer and the output layer are denoted by random variables X and Y respectively. We can describe this hidden layer by two conditional distributions: the encoder $p(\widetilde{\boldsymbol{x}}|\boldsymbol{x})$ and the decoder $p(\boldsymbol{y}|\widetilde{\boldsymbol{x}})$. This transformation process preserves the information of the input layer X without considering which individual neurons within the hidden layer $\widetilde{X}$ encode which features (i.e., neurons) of X. An optimal encoder process of the mutual information between X and the desired output Y denoted by $I(X, Y)$ can create the most compact encoding (i.e., minimally sufficient statistic) $\widetilde{X}$ of the input data X while $\widetilde{X}$ still has enough information (i.e., $\widetilde{X}$ can capture the important features of X as well as remove the unnecessary parts of X that do not make any contributions to the prediction of Y) to predict Y as accurately as possible.

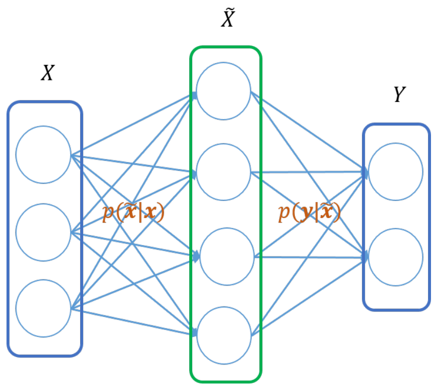


Figure 2.4: Architecture of a simple deep neural network in a supervised learning context.

Information bottlenecks and optimal representations. An information bottleneck [Tishby et al., 2000, Tishby and Zaslavsky, 2015] is proposed to be a computational framework that aims to find the most compact encoding $\widetilde{X}$ of the input data X. In particular, it is the optimal trade-off between the compression $\widetilde{X}$ and the prediction of the desired output Y as described in the following optimisation problem:

$$\min_{p(\widetilde{\boldsymbol{x}}|\boldsymbol{x}), p(\boldsymbol{y}|\widetilde{\boldsymbol{x}}), p(\widetilde{\boldsymbol{x}})} \left\{ I(X, \widetilde{X}) - \beta I(\widetilde{X}, Y) \right\} \tag{2.6}$$

where β specifies the amount of relevant information captured by the encoding process (i.e., the representations $\widetilde{X}$ and $I(\widetilde{X}, Y)$).

2.2 Deep sequence models

2.2.1 Recurrent neural networks

Recurrent neural networks (RNNs) [Rumelhart et al., 1986] are a class of DNNs. RNNs are specialised for processing sequential data (e.g., time series, sentences, documents, or audio samples). RNNs are extremely useful for natural language processing (NLP) systems [Cho et al., 2014a, Sutskever et al., 2014b] such as automatic translation, speech-to-text and sentiment analysis. Leveraging the idea of sharing parameters across different parts of a model, an RNN can not only extend and apply to data of different forms, but also generalise across them.

An RNN is similar to a DNN, except it has connections pointing backward. A visualisation of an RNN's architecture is depicted in Fig. 2.5 (left-hand figure). At each time step t, the state of a recurrent neuron (i.e, the hidden state denoted by $\boldsymbol{h}_t$) will receive the input vector $\boldsymbol{x}_t$ as well as the state vector from the previous step $t-1$ (i.e., $\boldsymbol{h}_{t-1}$) to obtain the state vector $\boldsymbol{h}_t$. In particular, we have:

$$\boldsymbol{h}_t = f(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t)$$

We can unroll the RNN network through time to gain a new visualisation as depicted in Fig. 2.5 (right-hand figure). Each recurrent neuron has two relevant input weights. One is for the input vector $\boldsymbol{x}_t$, and the other is for the state vector $\boldsymbol{h}_{t-1}$ of the previous time step $t-1$. At the time step t, if we denote the weight from the input vector $\boldsymbol{x}_t$ to the state $\boldsymbol{h}_t$ of the current recurrent neuron by $\mathbf{W}_{xh}$ and the weight from the state $\boldsymbol{h}_{t-1}$ of the previous recurrent neuron to the state $\boldsymbol{h}_t$ of the current recurrent neuron by $\mathbf{W}_{hh}$, the state $\boldsymbol{h}_t$ of the current recurrent neuron is computed as follows:

$$\boldsymbol{h}_t = \phi(\mathbf{W}_{xh}^{\top}\boldsymbol{x}_t + \mathbf{W}_{hh}^{\top}\boldsymbol{h}_{t-1} + \boldsymbol{b})$$

where $\boldsymbol{b}$ is the bias vector and $\phi(.)$ is the activation function (e.g., the ReLU or Tanh functions).

At the time step t, if we denote $\mathbf{W}_{hy}$ as the weight from the state $\boldsymbol{h}_t$ of the current recurrent neuron to the corresponding output denoted by $\boldsymbol{y}_t$, the output $\mathbf{y}_t$ is computed as follows:

$$\boldsymbol{y}_t = \phi(\mathbf{W}_{hy}^{\top}\boldsymbol{h}_t + \boldsymbol{c})$$

where $\boldsymbol{c}$ is the bias vector and $\phi(.)$ is the activation function (e.g., the softmax function).

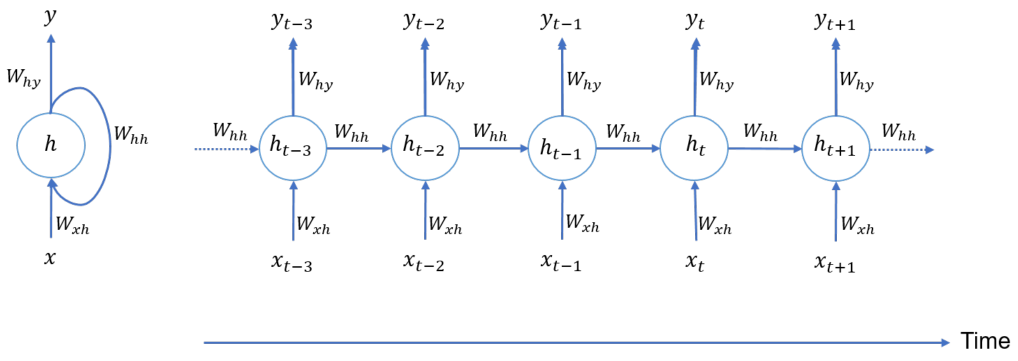


Figure 2.5: Architecture of a recurrent neural network with the outputs y plus the hidden states h of recurrent neurons.

Memory cell. Because the state of a recurrent neuron at the time step t is a function or can be considered a lossy summary of all previous inputs and recurrent neurons at previous time steps, we can say that a recurrent neuron's state has a form of a memory. The part of an RNN preserving some states across time steps is called a memory cell, or just a cell for simplicity.

Multi-layer RNNs. We can stack multiple layers of a memory cell to gain a deep multi-layer RNN. The architecture of a multi-layer RNN can be depicted as in Fig. 2.6. Compared to a basic one-layer RNN, a multi-layer RNN has an additional weight denoted by $\boldsymbol{R}_{hh}$ between the recurrent neurons of different layers. We compute the state $\boldsymbol{h}_t^l$ of the recurrent neuron at the time step t of the layer l with $l > 1$ as follows:

$$\boldsymbol{h}_t^l = \phi(\boldsymbol{R}_{hh}^{\top}\boldsymbol{h}_t^{l-1} + \mathbf{W}_{hh}^{\top}\boldsymbol{h}_{t-1}^l + \boldsymbol{d})$$

where $\boldsymbol{d}$ is the bias vector and $\phi(.)$ is the activation function (e.g., the ReLU or Tanh functions) while $\boldsymbol{h}_t^{l-1}$ and $\boldsymbol{h}_{t-1}^l$ are hidden states of the previous layer $(l-1)$ and the previous time step $(t-1)$ respectively.

Training RNNs. To train an RNN, we first unroll it through time and then use a regular backpropagation process. This approach is named backpropagation through time (BPTT).

Similar to a regular backpropagation process, in the first phase the forward process goes through

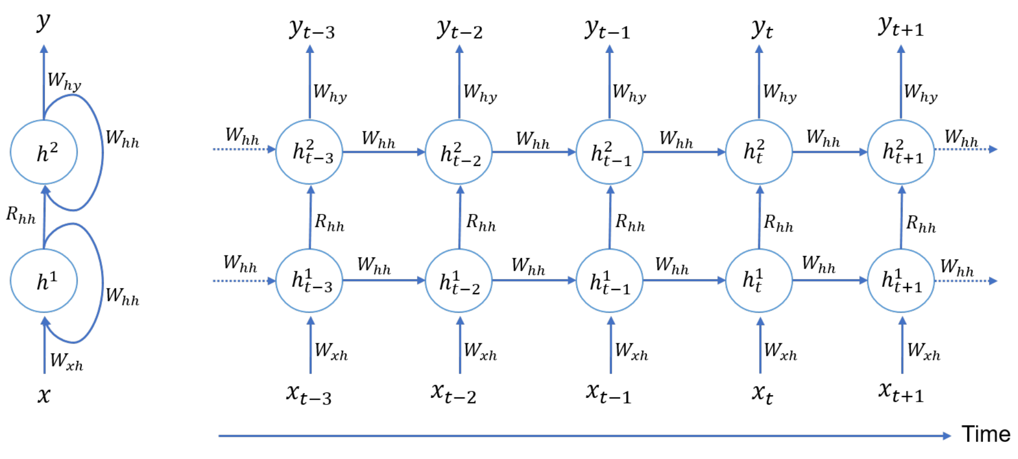


Figure 2.6: Architecture of a deep RNN (left-hand side) and unrolled through time (right-hand side).

the unrolled RNN to obtain an output sequence that is evaluated using an objective function (i.e., a cost or loss function). In the second phase, the gradients of the cost function are propagated backward through the unrolled RNN to update the RNN's parameters using the gradients obtained during BPTT.

To train an RNN for a long input sequence, we need a long unrolled RNN (i.e., it has many time steps), which may suffer from two serious issues, namely, the vanishing and exploding gradients problem (i.e, this problem makes a long unrolled RNN difficult to train to be able to achieve an optimal solution), as emerges in a DNN and the short-term memory problem (i.e., the lost information about some of the first inputs in the memory cell when traversing a long RNN). To deal with the first problem, there have been many approaches introduced such as initialising good model parameters, using nonsaturating activation functions (e.g., ReLU) and applying batch normalisation [Ioffe and Szegedy, 2015] or gradient clipping [Pascanu et al., 2013] techniques. To handle the second problem, various types of memory cells with long-term memory have been introduced like LSTMs [Hochreiter and Schmidhuber, 1997] and gated recurrent units (GRUs) [Cho et al., 2014b].

2.2.2 Long short-term memory networks

Long short-term memory (LSTM) networks are a type of RNNs capable of learning long-term dependencies, first proposed by Hochreiter and Schmidhuber [Hochreiter and Schmidhuber, 1997] and gradually improved over the years by other researches [Sak et al., 2014, Zaremba et al., 2014]. An LSTM network can address the exploding and vanishing gradients problems as well as the short-term memory problem (i.e., the lost information about some of the first inputs in the memory cell of a long RNN) in training RNNs effectively.

The key idea of an LSTM network is about storing a long-term memory. An LSTM network can learn to figure out what information from the inputs should be read and stored in the long-term state denoted by $\boldsymbol{c}_t$ as well as what information should be thrown out from $\boldsymbol{c}_t$. A visualisation of an LSTM network is shown in Fig. 2.7. As depicted in Fig. 2.7, there are four layers in an LSTM network: the main layer and three additional layers (i.e., gate controllers), namely, the forget gate, the input gate and the output gate.

- The main layer at the time step t aims to analyse the current input vector $\boldsymbol{x}_t$ and the previous (short-term) state $\boldsymbol{h}_{t-1}$ to gain the output $\boldsymbol{g}_t$ (i.e., $\boldsymbol{h}_t$ in a basic cell RNN). In an LSTM cell, the layer's output $\boldsymbol{g}_t$ does not go straight out, but instead goes through a gate controller to decide what parts are stored in the long-term state (i.e., $\boldsymbol{c}_t$).
- Using the logistic activation function, the forget gate $\mathbf{f}_t$ aims to learn which parts of the long-term state $\boldsymbol{c}_t$ should be erased. The input gate $\boldsymbol{i}_t$ aims to control which parts of $\boldsymbol{g}_t$ should be added to the long-term state $\boldsymbol{c}_t$ while the output gate $\boldsymbol{o}_t$ aims to learn which parts of the long-term state $\boldsymbol{c}_t$ should be outputted for both $\boldsymbol{h}_t$ and $\boldsymbol{y}_t$.

The following equation (i.e., Eq. (2.7)) summarises the aforementioned computing process of the four layers at the time step t:

$$\begin{aligned}
\boldsymbol{i}_t &= \sigma\left(\mathbf{W}_{xi}^{\top}\boldsymbol{x}_t + \mathbf{W}_{hi}^{\top}\boldsymbol{h}_{t-1} + \boldsymbol{b}_i\right)\\
\mathbf{f}_t &= \sigma\left(\mathbf{W}_{xf}^{\top}\boldsymbol{x}_t + \mathbf{W}_{hf}^{\top}\boldsymbol{h}_{t-1} + \boldsymbol{b}_f\right)\\
\boldsymbol{o}_t &= \sigma\left(\mathbf{W}_{xo}^{\top}\boldsymbol{x}_t + \mathbf{W}_{ho}^{\top}\boldsymbol{h}_{t-1} + \boldsymbol{b}_o\right)\\
\boldsymbol{g}_t &= \tanh\left(\mathbf{W}_{xg}^{\top}\boldsymbol{x}_t + \mathbf{W}_{hg}^{\top}\boldsymbol{h}_{t-1} + \boldsymbol{b}_g\right)\\
\boldsymbol{c}_t &= \mathbf{f}_t \otimes \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \otimes \boldsymbol{g}_t\\
\boldsymbol{y}_t &= \boldsymbol{h}_t = \boldsymbol{o}_t \otimes \tanh(\boldsymbol{c}_{t-1}) \qquad (2.7)
\end{aligned}$$

where $\mathbf{W}_{xi}$, $\mathbf{W}_{xf}$, $\mathbf{W}_{xo}$ and $\mathbf{W}_{xg}$ are the weight matrices from the input vector $\boldsymbol{x}_t$ to each of the four layers while $\mathbf{W}_{hi}$, $\mathbf{W}_{hf}$, $\mathbf{W}_{ho}$ and $\mathbf{W}_{hg}$ are the weight matrices from the previous short-term state $\boldsymbol{h}_{t-1}$ to each of the four layers, and $\boldsymbol{b}_i$, $\boldsymbol{b}_f$, $\boldsymbol{b}_o$ and $\boldsymbol{b}_g$ are the bias vectors to each of the four layers respectively. In general, the output $\boldsymbol{y}_t$ can be different from the short-term state $\boldsymbol{h}_t$ (i.e., $\boldsymbol{y}_t = \phi(\mathbf{W}_{hy}^{\top}\boldsymbol{h}_t + \boldsymbol{b}_y)$) where $\boldsymbol{b}_y$ is the bias vector, $\mathbf{W}_{hy}$ is the weight from $\boldsymbol{h}_t$ to $\boldsymbol{y}_t$, and $\phi(.)$ is the activation function (e.g., the softmax function)).

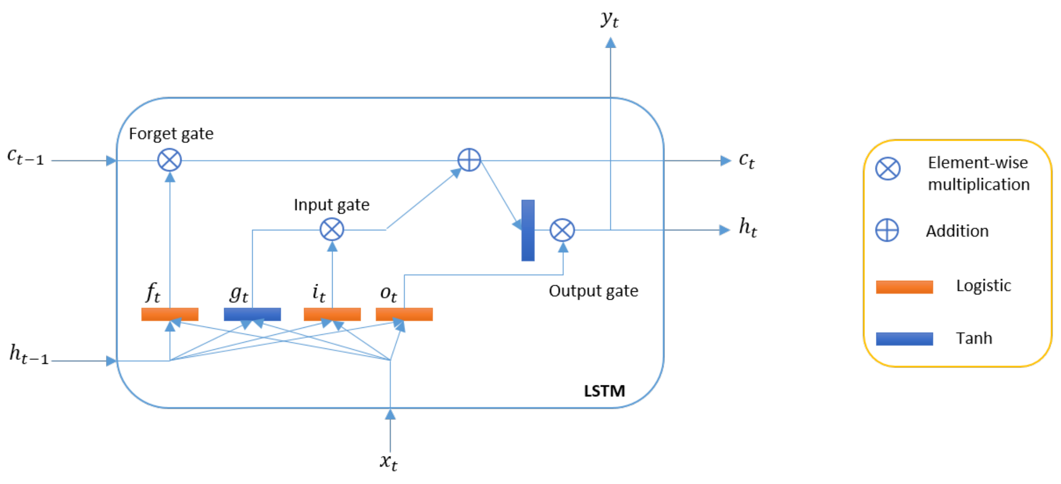


Figure 2.7: Architecture of a long short-term memory (LSTM) network.

2.2.3 Gated recurrent unit networks

A gated recurrent unit (GRU) network [Cho et al., 2014b] depicted in Fig. 2.8 can be considered a simplified version of an LSTM network. Following are the simplifications at the time step t:

- Both short-term $\boldsymbol{h}_t$ and long-term $\boldsymbol{c}_t$ states are merged into one single state denoted by $\boldsymbol{h}_t$.
- A single gate controller $\boldsymbol{z}_t$ is introduced to control both the input gate and the forget gate. If $\boldsymbol{z}_t$ outputs $\mathbf{1}$, the forget gate is open and the input gate is closed and vice versa (i.e., if $\boldsymbol{z}_t$ outputs $\mathbf{0}$, the forget gate is closed and the input gate is open). This process operates whenever a memory should be stored in a location whose current memory will be erased first before having a new incoming memory.
- There is no output gate in a GRU network. However, there is a new gate controller named $\boldsymbol{r}_t$ proposed to control which part of the previous state $\boldsymbol{h}_{t-1}$ will be used in the main layer $\boldsymbol{g}_t$.

Equation (2.8) summarises the aforementioned computing process at the time step t:

$$\begin{aligned}
\boldsymbol{z}_t &= \sigma\left(\mathbf{W}_{xz}^\top \boldsymbol{x}_t + \mathbf{W}_{hz}^\top \boldsymbol{h}_{t-1} + \boldsymbol{b}_z\right) \\
\boldsymbol{r}_t &= \sigma\left(\mathbf{W}_{xr}^\top \boldsymbol{x}_t + \mathbf{W}_{hr}^\top \boldsymbol{h}_{t-1} + \boldsymbol{b}_r\right) \\
\boldsymbol{g}_t &= \tanh\left(\mathbf{W}_{xg}^\top \boldsymbol{x}_t + \mathbf{W}_{hg}^\top (\boldsymbol{r}_t \otimes \boldsymbol{h}_{t-1}) + \boldsymbol{b}_g\right) \\
\boldsymbol{y}_t &= \boldsymbol{h}_t = \boldsymbol{z}_t \otimes \boldsymbol{h}_{t-1} + (1 - \boldsymbol{z}_t) \otimes \boldsymbol{g}_t
\end{aligned} \tag{2.8}$$

where $\mathbf{W}_{xz}$, $\mathbf{W}_{xr}$ and $\mathbf{W}_{xg}$ are the weight matrices from the input vector $\boldsymbol{x}_t$ to each layer $\boldsymbol{z}_t$,

$\boldsymbol{r}_t$ and $\boldsymbol{g}_t$, while $\mathbf{W}_{hz}$, $\mathbf{W}_{hr}$ and $\mathbf{W}_{hg}$ are the weight matrices from the previous state $\boldsymbol{h}_{t-1}$ to each layer $\boldsymbol{z}_t$, $\boldsymbol{r}_t$ and $\boldsymbol{g}_t$. In addition, $\boldsymbol{b}_z$, $\boldsymbol{b}_r$ and $\boldsymbol{b}_g$ are the bias vectors to each layer $\boldsymbol{z}_t$, $\boldsymbol{r}_t$ and $\boldsymbol{g}_t$ respectively. In general, the output $\boldsymbol{y}_t$ can be different from the state $\boldsymbol{h}_t$ (i.e., $\boldsymbol{y}_t = \phi(\mathbf{W}_{hy}^\top \boldsymbol{h}_t + \boldsymbol{b}_y)$) where $\boldsymbol{b}_y$ is the bias vector, $\mathbf{W}_{hy}$ is the weight from $\boldsymbol{h}_t$ to $\boldsymbol{y}_t$, and $\phi(.)$ is the activation function (e.g., the softmax function)).

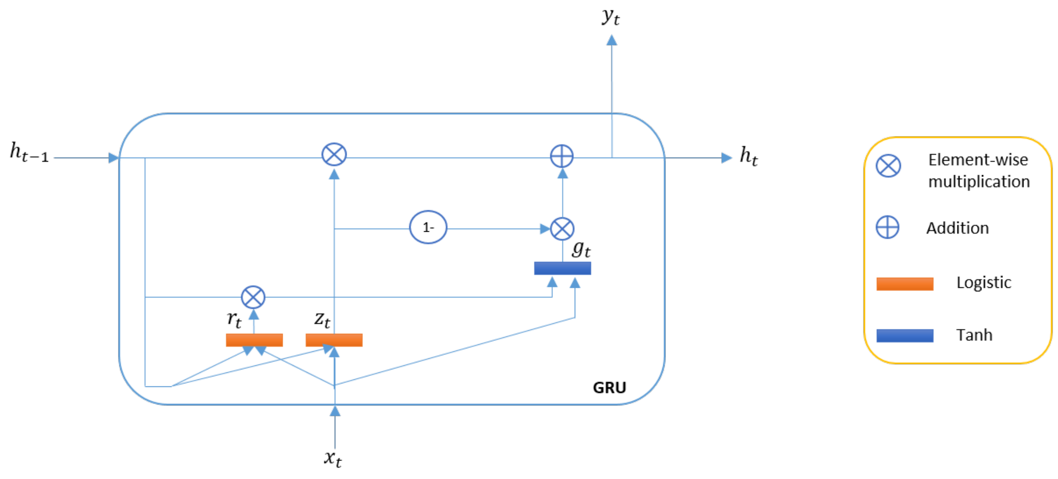


Figure 2.8: Architecture of a gated recurrent unit (GRU) network.

2.2.4 Bidirectional recurrent neural networks

Bidirectional recurrent neural networks (bi-RNNs) [Schuster and Paliwal, 1997] are a class of DNNs. In bi-RNNs, at a specific time step t the input information in the past (i.e., the forward states used in RNNs) and the future (i.e., the backward states, which are usually also useful) can be leveraged to learn the corresponding state and prediction (i.e., for both $\boldsymbol{h}_t$ and $\boldsymbol{y}_t$). This is different from RNNs (i.e., at a specific time frame, RNNs can only capture the input information in the past to learn the corresponding state and prediction). Bi-RNNs can effectively address this limitation of RNNs.

A bi-RNN combines two RNNs, where one RNN moves forward while the other RNN moves backward through time. The overall structure of a bi-RNN is depicted in Fig. 2.9. A bi-RNN takes a sequence of input data $\mathbf{B} = (\boldsymbol{i}_1, \boldsymbol{i}_2, ..., \boldsymbol{i}_l)$ for mapping to a corresponding sequence of target output $\mathbf{Y} = (\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_l)$. The computation of a bi-RNN is as follows:

$$\boldsymbol{h}_k^1 = a(\mathbf{H}^\top \boldsymbol{h}_{k-1}^1 + \boldsymbol{U}^\top \boldsymbol{i}_k)$$

$$\boldsymbol{g}_k^1 = a(\mathbf{H}^\top \boldsymbol{g}_{k+1}^1 + \boldsymbol{V}^\top \mathbf{i}_k)$$

$$\boldsymbol{o}_k = W^\top \begin{bmatrix} \boldsymbol{h}_k^1 \\ \boldsymbol{g}_k^1 \end{bmatrix} \quad \text{and } \boldsymbol{p}_k = \text{softmax}\,(\boldsymbol{o}_k)$$

where $k = 1, ..., l$, $\boldsymbol{h}_0^1$, and $\boldsymbol{g}_{l+1}^1 = \boldsymbol{g}_0^1$ are initial hidden states, $a(\cdot)$ is the element-wise activation function and $\theta = (\boldsymbol{U}, \boldsymbol{V}, \mathbf{W}, \mathbf{H})$ is the model (i.e., the model parameters). We further note that $\boldsymbol{p}_k, k = 1, \ldots, l$ is a *discrete* distribution over the target labels. To find the best model θ^* we need to solve the following optimisation problem:

$$\max_{\theta} \sum_{(\mathbf{B},\mathbf{Y}) \in \mathbf{D}} \log p(\mathbf{Y} \mid \mathbf{B})$$

where we define

$$\log p(\mathbf{Y} \mid \mathbf{B}) = \sum_{k=1}^{l} \log p(\boldsymbol{y}_k \mid \boldsymbol{y}_1, ..., \boldsymbol{y}_{k-1}, \boldsymbol{i}_1, \boldsymbol{i}_2, ..., \boldsymbol{i}_l) = \sum_{k=1}^{l} \log p(\boldsymbol{y}_k \mid \boldsymbol{o}_k)$$

We denote $\mathbf{D}$ as the training set including pairs $(\mathbf{B}, \mathbf{Y})$ of the data and their corresponding target labels, and $\boldsymbol{o}_k$ as a function or it can be considered a lossy summary of the sequence $(\boldsymbol{y}_1, ..., \boldsymbol{y}_{k-1}, \boldsymbol{i}_1, \boldsymbol{i}_2, ..., \boldsymbol{i}_l)$.

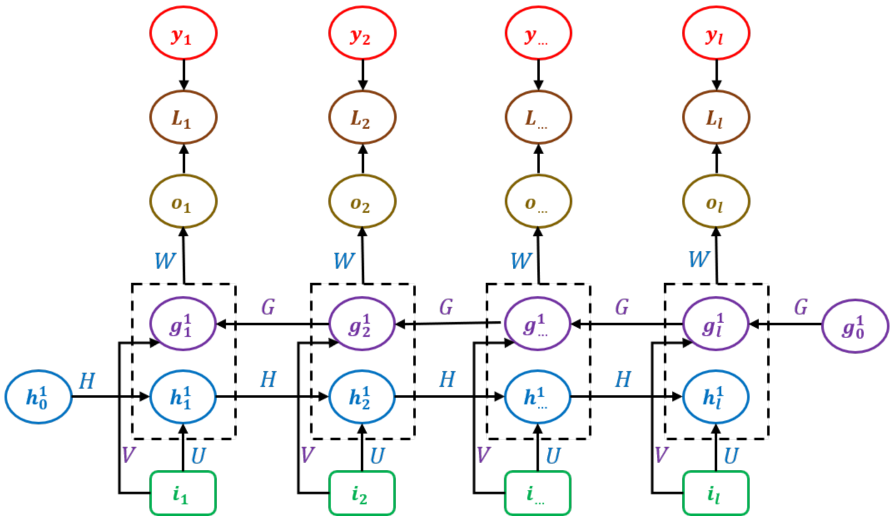


Figure 2.9: Structure of a bidirectional recurrent neural network. The bi-RNN will learn to map the input sequences of items $(\boldsymbol{i}_1, \boldsymbol{i}_2, ..., \boldsymbol{i}_l)$ to the target output sequence $(\boldsymbol{y}_1, \boldsymbol{y}_2, ..., \boldsymbol{y}_l)$ with the loss L_i at each time step t. $\boldsymbol{h}_t$ represents the forward-propagated hidden state (towards the right) while $\boldsymbol{g}_t$ stands for the backward-propagated hidden state (towards the left). At each time step t, the predicted output can benefit from the relevant information from the past from its $\boldsymbol{h}_t$ and from the future from its $\boldsymbol{g}_t$.

2.2.5 Sequence-to-sequence models

Although DNNs are powerful machine learning models, they can only be used with the problems where the dimensionality of their inputs and outputs is known and fixed. This is a considerable limitation of DNNs due to many significant problems (i.e., NLP problems [Sutskever et al., 2014b, Cho et al., 2014a]) best expressed with sequences whose lengths are not known prior such as machine translation and question answering.

A sequence-to-sequence network (seq2seq network or encoder-decoder network) has been proposed to deal with sequence-to-sequence problems. A seq2seq network normally consists of two different RNNs. The first and second RNNs are called the encoder and the decoder respectively. The aim of the encoder is to read and encode an input sequence (i.e., a source sequence), one time step at a time, to obtain the final fixed dimensional vector representation (i.e., context vector) for the input sequence. The context vector is then fed to the second RNN to gain an output sequence (i.e., a target sequence). In seq2seq networks, LSTMs are used for the memory cells in both the encoder and the decoder due to their ability to learn data with long-range temporal dependencies, which makes them a natural choice for this application because of the considerable time lag between the inputs and their corresponding outputs.

Assume that we have an input sequence $(\boldsymbol{x}_1, ..., \boldsymbol{x}_T)$ and its corresponding output sequence $(\boldsymbol{y}_1, ..., \boldsymbol{y}_{T'})$, the goal of a seq2seq network is to estimate (i.e., maximise) the conditional probability $p(\boldsymbol{y}_1, ..., \boldsymbol{y}_{T'}|\boldsymbol{x}_1, ..., \boldsymbol{x}_T)$ as:

$$p(\boldsymbol{y}_1, ..., \boldsymbol{y}_{T'}|\boldsymbol{x}_1, ..., \boldsymbol{x}_T) = \prod_{t=1}^{T'} p(\boldsymbol{y}_t|\boldsymbol{x}_1, ..., \boldsymbol{x}_T, \boldsymbol{y}_1, ..., \boldsymbol{y}_{t-1}) \tag{2.9}$$

The seq2seq network computes this conditional probability by first obtaining the fixed dimensional representation v of the input sequence $(\boldsymbol{x}_1, ..., \boldsymbol{x}_T)$ given by the last hidden state of the encoder and then computing the probability of $(\boldsymbol{y}_1, ..., \boldsymbol{y}_{T'})$ with the decoder with a standard formulation whose initial hidden state is set to the representation $\boldsymbol{v}$ of $(\boldsymbol{x}_1, ..., \boldsymbol{x}_T)$. Equation (2.9) can be rewritten as:

$$p(\boldsymbol{y}_1, ..., \boldsymbol{y}_{T'}|\boldsymbol{x}_1, ..., \boldsymbol{x}_T) = \prod_{t=1}^{T'} p(\boldsymbol{y}_t|\boldsymbol{v}, \boldsymbol{y}_1, ..., \boldsymbol{y}_{t-1}) \tag{2.10}$$

In Eq. (2.10), each $p(\boldsymbol{y}_t|\boldsymbol{v}, \boldsymbol{y}_1, ..., \boldsymbol{y}_{t-1})$ distribution is represented with a softmax over all the words in the vocabulary. Note that we require each sentence to end with a special end-of-sentence symbol EOS, which enables the model to define a distribution over sequences of all possible lengths. The overall process is visualised in Fig. 2.10, where the shown seq2seq network computes the representation of A, B, C and EOS, and then uses this representation to compute the probability of W, X, Y, Z and EOS.

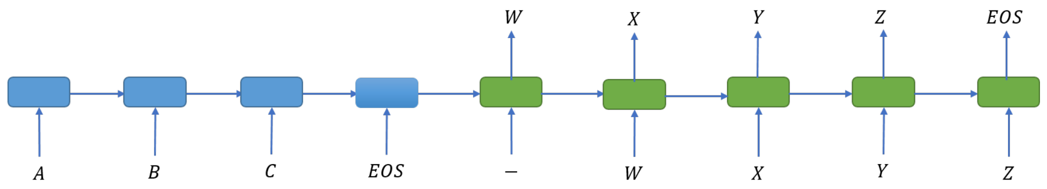


Figure 2.10: Architecture of a seq2seq model reading an input sentence including A, B, C and EOS, and producing W, X, Y, Z and EOS as the output sentence. The model stops making predictions after outputting the end-of-sentence token EOS. Note that the encoder using an LSTM reads the input sentence in reverse. By doing so, we introduce many short-term dependencies in the data, aiming to simplify the optimisation problem.

2.2.6 Attention mechanism

Using a fixed-length context vector to encode the meaning of a whole input sequence, regardless of how long it may be, can be a problem with standard seq2seq networks. For example, assume that we have two sentences, "I cannot be happier with the results of my study in my first year" compared with "I cannot be happy with the results of my study in my first year" with only one word different; however, this different word causes a large meaning difference between these two sentences. Both the encoder and decoder must be nuanced enough to represent this change as a very slightly different point in space. This causes a big challenge for standard seq2seq models.

The attention mechanism was introduced and gradually improved over the years by many researchers such as [Bahdanau et al., 2014] and [Luong et al., 2015], to address the aforementioned problem faced in standard seq2seq models. The general idea is to provide an elegant way for the decoder to pay attention to and focus on specific parts of the source sequence (i.e., it is strongly relevant for predicting a target word at each time step) during the translation process. In particular, this seq2seq model aims to encode the input sentence into a sequence of vectors and chooses a subset of these vectors adaptively while decoding the translation.

Bahdanau and colleagues [Bahdanau et al., 2014] proposed a novel architecture as depicted in Fig. 2.11 for neural machine translation. In this model, the authors used a bi-RNN for the encoder and an RNN for the decoder.

In a standard seq2seq model, we have the following joint probability:

$$p(\boldsymbol{y}_1, ..., \boldsymbol{y}_{T'}) = \prod_{t=1}^{T'} p(\boldsymbol{y}_t | \boldsymbol{v}, \boldsymbol{y}_1, ..., \boldsymbol{y}_{t-1}) \tag{2.11}$$

where $\boldsymbol{v} = f(\boldsymbol{h}_1, ..., \boldsymbol{h}_T)$ and $\boldsymbol{h}_t = g(\boldsymbol{x}_t, \boldsymbol{h}_{t-1})$ with f and g as nonlinear functions. We often use an LSTM cell for g while $f(\boldsymbol{h}_1, ..., \boldsymbol{h}_T) = \boldsymbol{h}_T$ is the context vector generated from the sequence of the encoder's hidden states (i.e., the representation of the input sequence).

In Bahdanau's model, each conditional probability in Eq. (2.11) is defined as:

$$p(\boldsymbol{y}_t|\boldsymbol{v}, \boldsymbol{y}_1, ..., \boldsymbol{y}_{t-1}) = a(\boldsymbol{y}_{t-1}, \boldsymbol{s}_t, \boldsymbol{v}_t)$$

The decoder's hidden state at time step t denoted by $\boldsymbol{s}_t$ is computed as:

$$\boldsymbol{s}_t = b(\boldsymbol{s}_{t-1}, \boldsymbol{y}_{t-1}, \boldsymbol{v}_t)$$

where a and b are nolinear functions.

The probability of each target word $\boldsymbol{y}_t$ is conditioned on a specific context vector $\boldsymbol{v}_t$ while the context vector $\boldsymbol{v}_t$ is the weighted sum of all encoders' hidden states:

$$\boldsymbol{v}_t = \sum_{j=1}^{T} \alpha_{tj} \boldsymbol{h}_j$$

The weight α_{tj} of each encoder's hidden state $\boldsymbol{h}_j$ is computed by

$$\alpha_{tj} = \frac{\exp(c(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j))}{\sum_{k=1}^{T} \exp(c(\boldsymbol{s}_{t-1}, \boldsymbol{h}_k))} \tag{2.12}$$

The term $c(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$ in Eq. (2.12), for example, computes the alignment score, aiming to measure the match between the input at position j and the output at position t. In particular, $c(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j)$ is computed as follows:

$$c(\boldsymbol{s}_{t-1}, \boldsymbol{h}_j) = \boldsymbol{q}^\top \tanh(\mathbf{W}\,[\boldsymbol{s}_{t-1}; \boldsymbol{h}_j]$$

where $\boldsymbol{q}$ and $\mathbf{W}$ are the learnable weight vector and the matrix respectively.

2.2.7 Pointer networks

Pointer networks [Vinyals et al., 2015b] are a novel neural architecture that aims to learn the conditional probability of an output sequence whose elements are discrete tokens which correspond to positions in an input sequence. A pointer network overcomes the limitation of requiring the size of the output dictionary to be fixed prior as in the introduced sequence-to-sequence paradigms. The main idea of a pointer network is repurposing the attention mechanism of [Bahdanau et al., 2014] to create pointers to input elements.

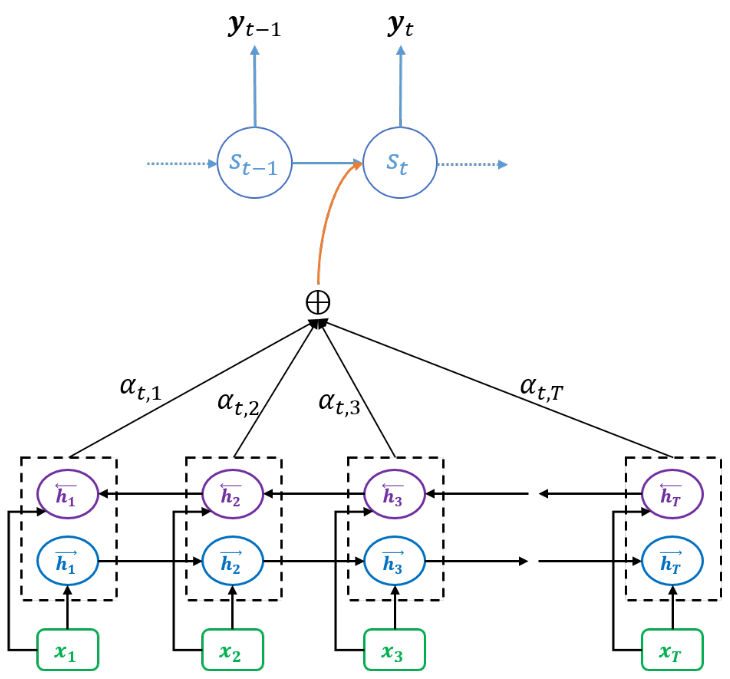


Figure 2.11: Architecture of Bahdanau et al.'s seq2seq attention mechanism model in generating the t-th target word $\boldsymbol{y}_t$ given a source sequence $(\boldsymbol{x}_1, ..., \boldsymbol{x}_T)$.

Assume that we have a training pair (I, O) where $\text{I} = \{\boldsymbol{i}_1, \boldsymbol{i}_2, ..., \boldsymbol{i}_l\}$ and $\text{O} = \{n_1, n_2, ..., n_m\}$ are the sequence of l data points and m indices between 1 and l respectively. A pointer network whose structure can be depicted as in Fig. 2.12 learns the model parameters θ by maximising the following conditional probabilities for the training set:

$$\theta^* = \underset{\theta}{\text{argmax}} \sum_{\text{I,O}} \text{log} p(\text{O}|\text{I}; \theta)$$

where we have defined

$$p(\text{O}|\text{I}; \theta) = \prod_{i=1}^{m} p_\theta(n_i | n_1, ..., n_{i-1}, \text{I}; \theta) \tag{2.13}$$

To solve the problem in Eq. (2.13), a pointer network uses an attention mechanism as follows:

$$\boldsymbol{u}_{ji} = \boldsymbol{v}^\top \tanh(\boldsymbol{U}_h \boldsymbol{h}_j + \boldsymbol{U}_s \boldsymbol{s}_i), \quad j \in \{1, ..., l\}$$

$$p(n_i | n_1, ..., n_{i-1}, \text{I}) = \text{softmax}(\boldsymbol{u}_{ji}), \quad j \in \{1, ..., l\}$$

where $i \in \{1, ..., m\}$. The vector $\boldsymbol{v}$ and the matrices $\boldsymbol{U}_h$ and $\boldsymbol{U}_s$ are learnable parameters. $\{\boldsymbol{h}_1, \boldsymbol{h}_2, ..., \boldsymbol{h}_l\}$ and $\{\boldsymbol{s}_1, \boldsymbol{s}_2, ..., \boldsymbol{s}_m\}$ are the encoder's hidden states and the decoder's hidden states respectively.

The softmax function aims to normalise the vector $\boldsymbol{u}_i$ to be an output distribution over the dictionary of inputs. Unlike the standard attention mechanism, in a pointer network we do not combine the encoder states to provide more extra information to the decoder. We use $\boldsymbol{u}_i$ as pointers to the input data to find the most appropriate input element for the output at the time step i of the decoder. Furthermore, we use the corresponding $\boldsymbol{i}_{n_{i-1}}$ as the input for the decoder's state $\boldsymbol{s}_i$ to set the condition on n_{i-1}.

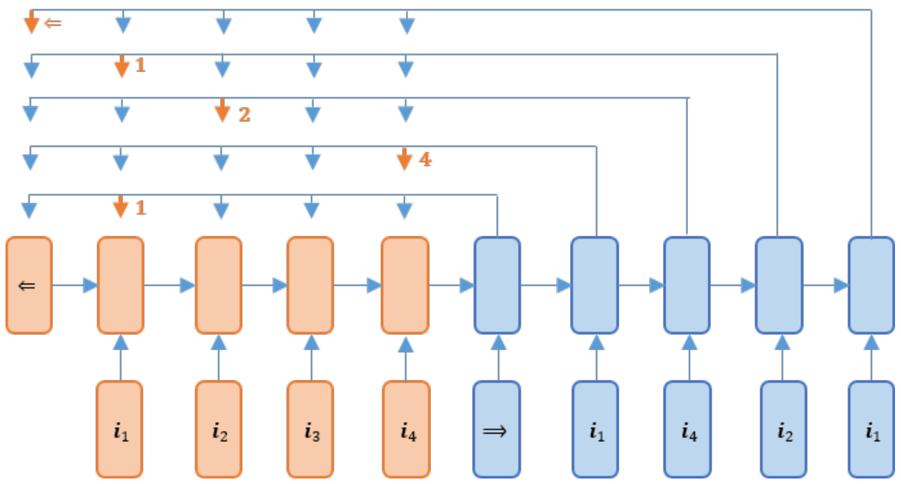


Figure 2.12: Structure of a pointer network, which includes an encoding RNN (on the left-hand side with orange colour) aiming to encode the input sequence to be fed to the generating network (on the right-hand side with light-blue colour). At each step, the output from the generating network using a content-based attention mechanism over the inputs is a softmax distribution with a size equal to the length of the input sequence.

2.3 Deep generative models

2.3.1 Variational autoencoders

Variational autoencoders (VAEs) [Kingma and Welling, 2014] are neural variational models that aim to learn approximations of the data distributions. Assume that a neural variational model is parameterised by θ and the dataset $\mathbf{X} = \{\boldsymbol{x}_i\}_{i=1}^N$ consists of N independent and identically distributed samples of some continuous or discrete variable $\boldsymbol{x}_i$. It aims to maximise the following log-likelihood function:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \log \prod_{i=1}^{N} p_\theta(\boldsymbol{x}_i) \tag{2.14}$$

to obtain the best parameter θ^* of the model to approach the data distribution.

If a neural variational model is representative of the dataset, for every data point $\boldsymbol{x}_i$ there should exist at least one setting of the latent variables which causes the model to generate something very similar to $\boldsymbol{x}_i$. We assume that the data is generated by a random process, as depicted in Fig. 2.13 (left-hand figure), which involves an unobserved continuous random variable $\boldsymbol{z}$ (e.g., $\boldsymbol{z} \sim \mathcal{N}(0, I)$). The process first generates a value $\boldsymbol{z}$ from some prior distribution $p_\theta(\boldsymbol{z})$, and then a value $\boldsymbol{x}_i$ is generated from some conditional distribution $p_\theta(\boldsymbol{x}_i|\boldsymbol{z})$. In this case, the likelihood function mentioned in Eq. (2.14) is written as:

$$p_\theta(\boldsymbol{x}_i) = \int p_\theta(\boldsymbol{x}_i|\boldsymbol{z})p_\theta(\boldsymbol{z})d\boldsymbol{z} \tag{2.15}$$

It is impossible to compute $p_\theta(\boldsymbol{x}_i)$ because we cannot compute the integral $p_\theta(\boldsymbol{x}_i|\boldsymbol{z})$ for all the possible values of $\boldsymbol{z}$. To narrow down the value space in order to facilitate faster searching (i.e., the key idea behind the VAE is to attempt to sample values of z that are likely to have produced $\boldsymbol{x}_i$ and to compute $p_\theta(\boldsymbol{x}_i)$ just from those), an approximation function was proposed to output $q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$, which is parameterised by ϕ given an input $\boldsymbol{x}_i$ as depicted in Fig. 2.13 (right-hand figure).

Eq. 2.15 can be rewritten as:

$$\begin{aligned}
\log p_\theta(\boldsymbol{x}_i) &= \mathrm{E}_{\mathrm{z}\sim \mathrm{q}_\phi(\mathrm{z}|\mathrm{x_i})}\left[\log p_\theta(\boldsymbol{x}_i)\right] \\
&= \mathrm{E}_{\mathrm{z}\sim \mathrm{q}_\phi(\mathrm{z}|\mathrm{x_i})}\left[\log \frac{p_\theta(\boldsymbol{x}_i|z)p_\theta(z)}{p_\theta(z|\boldsymbol{x}_i)}\right] \\
&= \mathrm{E}_{\mathrm{z}\sim \mathrm{q}_\phi(\mathrm{z}|\mathrm{x_i})}\left[\log \frac{p_\theta(\boldsymbol{x}_i|z)p_\theta(z)}{p_\theta(z|\boldsymbol{x}_i)}\frac{q_\phi(z|\boldsymbol{x}_i)}{q_\phi(z|\boldsymbol{x}_i)}\right] \\
&= \mathrm{E}_{\mathrm{z}\sim \mathrm{q}_\phi(\mathrm{z}|\mathrm{x_i})}\left[\log p_\theta(\boldsymbol{x}_i|z)\right] - \mathrm{E}_{\mathrm{z}\sim \mathrm{q}_\phi(\mathrm{z}|\mathrm{x_i})}\left[\log \frac{q_\phi(z|\boldsymbol{x}_i)}{p_\theta(z)}\right] + \mathrm{E}_{\mathrm{z}\sim \mathrm{q}_\phi(\mathrm{z}|\mathrm{x_i})}\left[\log \frac{q_\phi(z|\boldsymbol{x}_i)}{p_\theta(z|\boldsymbol{x}_i)}\right] \\
&= \mathrm{E}_{\boldsymbol{z}\sim q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}[\log p_\theta(\boldsymbol{x}_i|\boldsymbol{z})] - D_{KL}[q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)\|p_\theta(\boldsymbol{z})] + D_{KL}[q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)\|p_\theta((\boldsymbol{z}|\boldsymbol{x}_i)]
\end{aligned}$$

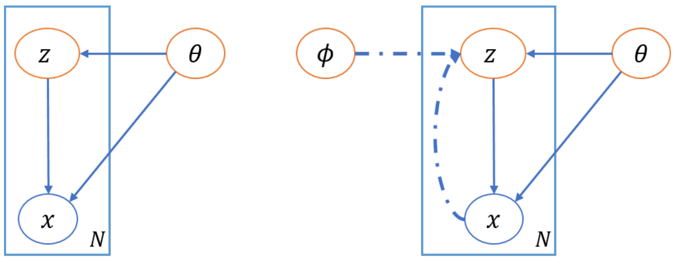


Figure 2.13: Standard VAE model is represented as a graphical model in the left-hand figure and a variant with the variational parameters ϕ is learned jointly with the generative model parameters θ (in the right-hand figure). The rectangle with N shows that we can sample $\boldsymbol{z}$ and $\boldsymbol{x}$ N times when the model parameters ϕ and θ remain fixed.

VAEs have emerged as one of the popular approaches to unsupervised learning of complex distributions because they are built on top of standard function approximators (DNNs) that can be trained with stochastic gradient decent. The key insight behind VAEs is that they are trained to maximise the variational lower bound $L(;\theta)$ on log-likelihood for each data sample $\boldsymbol{x}_i$ of the dataset:

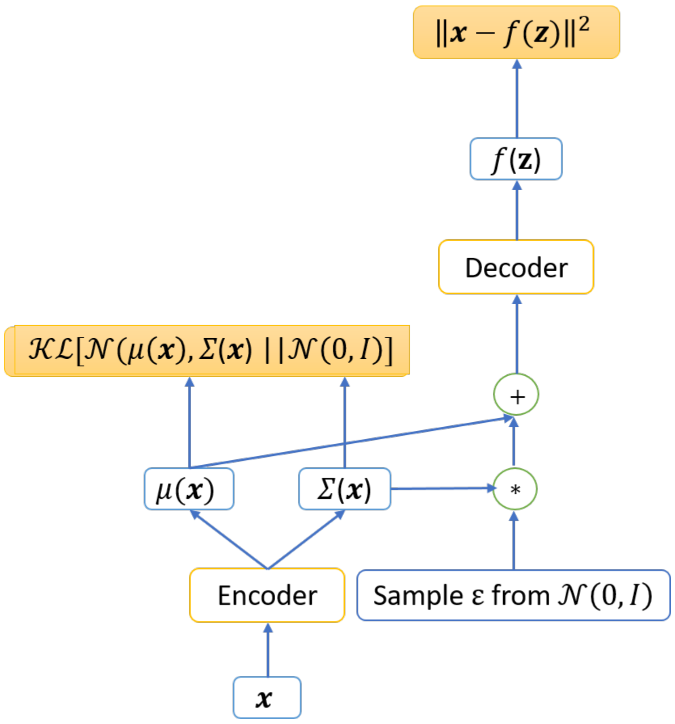
$$L(\boldsymbol{x}_i;\theta) = \mathrm{E}_{\boldsymbol{z}\sim q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)}[\log p_\theta(\boldsymbol{x}_i|\boldsymbol{z})] - D_{KL}[q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)\|p_\theta(\boldsymbol{z})] \tag{2.16}$$

where $q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$ is a posterior distribution, and $p_\theta(\boldsymbol{z})$ is a prior distribution for the latent variable $\boldsymbol{z}$. The first term of Eq. (2.16) can be considered the data reconstruction likelihood (from the given $\boldsymbol{z}$ code). The second term tries to make the posterior $q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$ and the prior $p_\theta(\boldsymbol{z})$ close to each other. One of the easy choices for the prior $p_\theta(\boldsymbol{z})$ is normal Gaussian distribution $\mathcal{N}(0, I)$. This way, an encoder-decoder model can be trained to maximise the lower bound $L(\boldsymbol{x}_i;\theta)$. The encoder of this network is $q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$, which can be modelled as a neural network that learns to output the parameters (mean $\boldsymbol{\mu}$ and variance Σ) of the posterior distribution $q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$ given the data $\boldsymbol{x}_i$ as input. The encoder also tries to force the learning to be close to the chosen prior $p_\theta(\boldsymbol{z})$. On the other hand, the decoder of this model is $p_\theta(\boldsymbol{x}_i|\boldsymbol{z})$, which can be modelled as a neural network that resembles a reconstruction of the data $\boldsymbol{x}_i$ from the code $\boldsymbol{z}$ sampled from the learned posterior $q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)$. Fig. 2.14 shows the typical architecture of a VAE encoder and decoder.

The VAE is one of the most popular approaches to variational learning in deep generative models. However, the main challenge of variational methods is how to find an effective way to minimise $D_{KL}[q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)\|p_\theta(\boldsymbol{z})]$ towards 0 to let $p_\theta(\boldsymbol{x}_i)$ converge (in term of distribution) to the true distribution. It is not straightforward to ensure that $D_{KL}[q_\phi(\boldsymbol{z}|\boldsymbol{x}_i)\|p_\theta(\boldsymbol{z})] = 0$. In practice, variational methods often obtain very good likelihoods, but are known to produce lower quality or blurry samples.

2.3.2 Generative adversarial networks

Generative adversarial networks (GANs) [Goodfellow et al., 2014a] are one of the most successful generative models. They aim to perform implicit density estimation by training the generator G such that $G(\boldsymbol{z})$ fed by $\boldsymbol{z} \sim p_z$ can mimic the true data in a given dataset. From a game perspective, a GAN can be seen as a game of two players: the discriminator D and the generator G. The task of the discriminator is to discriminate the true data and generate samples, while the task of the generator is to make its generated samples indistinguishable from the true samples



$\|x - f(z)\|^2$
$f(z)$
Decoder
$\mathcal{KL}[\mathcal{N}(\mu(x), \Sigma(x) \,||\, \mathcal{N}(0, I)]$
+
$\mu(x)$
$\Sigma(x)$
*
Encoder
Sample ε from $\mathcal{N}(0, I)$
x

Figure 2.14: Typical architecture of a VAE encoder and decoder. The orange boxes are the objectives of the encoder and decoder, while $\boldsymbol{z}$ is sampled using the equation $\boldsymbol{z} = \boldsymbol{\mu} + \text{diag}\,(\boldsymbol{\sigma})^{1/2}\,\boldsymbol{\epsilon}$ and $\boldsymbol{\epsilon} \sim \mathcal{N}(0, I)$. [Doresch, 2016]

(see Fig. 2.15). A GAN is formulated as a minimax problem:

$$\min_G \max_D \left(\mathbb{E}_{p_d}\left[\log D\left(\boldsymbol{x}\right)\right] + \mathbb{E}_{p_z}\left[\log\left(1 - D\left(G\left(\boldsymbol{z}\right)\right)\right)\right]\right)$$

where p_d is the empirical data distribution.

It can be proven that at the Nash equilibrium point, the distribution p_g induced by $G\left(\boldsymbol{z}\right)$ with $\boldsymbol{z} \sim p_z$ is exactly the data distribution, and the discriminator cannot distinguish between the true and generated samples:

$$p_g = p_d$$
$$D\left(\boldsymbol{x}\right) = \frac{1}{2}, \forall \boldsymbol{x}$$

In particular, GANs use the Jensen–Shannon divergence to minimise the divergence between two distributions p_d and p_g. Some (e.g., [Huszar, 2015]) believe the main reason behind GANs' success is their use of the Jensen–Shannon divergence instead of using the Kullback–Leibler divergence.

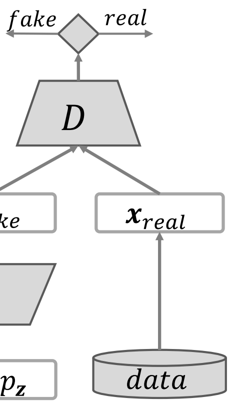


Figure 2.15: Diagram of a generative adversarial network.

Although GANs have shown great performance in many generative tasks to replicate rich real-world content such as images, human language, and music, they have a well-known limitation in term of the mode collapsing problem [Goodfellow et al., 2014a, Santurkar et al., 2018]. With this problem, the generated samples miss some modes in the true data or can only partly recover some modes in the true data.

2.4 Deep domain adaptation

2.4.1 Using the generative adversarial network principle

In many cases, direct access to vast quantities of labelled data for the task of interest (i.e., the target domain) is either costly or otherwise impossible, but labels are readily available for related training sets (i.e., the source domain). The deep domain adaptation approach [Ganin and Lempitsky, 2015] not only aims to learn the representations of the data from both the source and target domains, but also aims to minimise the divergence between the source and target domains. Hence, the learning obtained from the source domain can be effectively transferred to the target domain.

Mathematically, the source and target data are mapped via the generator G to a joint feature space. Inspired by the GAN principle, an additional discriminator is then invoked to bridge the gap between the source and target domains in this joint feature space. Another source classifier C is simultaneously employed to classify the source data with labels. When the gap between the source and target domains in the joint space vanishes, we can use the source classifier C to transfer the learning from the source to the target domains. Denote the source and target datasets by $\mathrm{S} = \left\{\left(\boldsymbol{x}_1^S, y_1\right), \ldots, \left(\boldsymbol{x}_{N_S}^S, y_{N_S}\right)\right\}$ where the labels $y_i \in \{-1, 1\}$ and

$\mathbf{T} = \left\{ \boldsymbol{x}_1^T, \ldots, \boldsymbol{x}_{N_T}^T \right\}$. The generator G, the domain discriminator D, and the source classifier C are trained via the following optimisation problem:

$$\min_C \sum_{i=1}^{N_S} \ell\left(C\left(G\left(\boldsymbol{x}_i^S\right)\right), y_i\right) \tag{2.17}$$

where $\ell\left(y, y'\right)$ specifies the loss function (e.g., the cross-entropy loss).

$$\min_G \max_D \left(\mathbb{E}_{x\sim p_S}\left[\log D\left(G\left(\boldsymbol{x}\right)\right)\right] + \mathbb{E}_{x\sim p_T}\left[\log\left(1 - D\left(G\left(\boldsymbol{x}\right)\right)\right)\right]\right) \tag{2.18}$$

where $p_S(\cdot) = \frac{1}{N_S}\sum_{i=1}^{N_S}\delta_{\boldsymbol{x}_i^S}(\cdot)$ and $p_T(\cdot) = \frac{1}{N_T}\sum_{i=1}^{N_T}\delta_{\boldsymbol{x}_i^T}(\cdot)$ are the empirical distributions over the source and target data. Here we note that $\delta_{\boldsymbol{x}}(\cdot)$ is the atom measure at the sample $\boldsymbol{x}$. With the above notions, the optimisation problem in Eq. (2.18) can be equivalently rewritten as:

$$\min_G \max_D O\left(G, D, C\right) \tag{2.19}$$

where we have defined

$$\mathcal{O}\left(G, D, C\right) = \frac{1}{N_S}\sum_{i=1}^{N_S}\log D\left(G\left(\boldsymbol{x}_i^S\right)\right) + \frac{1}{N_T}\sum_{i=1}^{N_T}\log\left[1 - D\left(G\left(\boldsymbol{x}_i^T\right)\right)\right]$$

Putting the optimisation problems in Eqs. (2.17 and 2.19) together, we arrive at the joint objective function with the trade-off parameter $\lambda > 0$:

$$\mathcal{J}\left(G, D, C\right) = \frac{1}{N_S}\sum_{i=1}^{N_S}\ell\left(C\left(G\left(\boldsymbol{x}_i^S\right)\right), y_i\right) + \lambda\left(\frac{1}{N_S}\sum_{i=1}^{N_S}\log D\left(G\left(\boldsymbol{x}_i^S\right)\right) + \frac{1}{N_T}\sum_{i=1}^{N_T}\log\left[1 - D\left(G\left(\boldsymbol{x}_i^T\right)\right)\right]\right)$$

where we seek the optimal generator G^*, the domain discriminator D^*, and the source classifier C^* by solving

$$\left(C^*, G^*\right) = \text{argmin}_{C,G}\mathcal{J}\left(G,\ D,\ C\right)$$

$$D^* = \text{argmax}_D\mathcal{J}\left(G,\ D,\ C\right)$$

2.4.2 Using the Wasserstein (Earth Mover's) distance

Wasserstein distance. Let $(\mathcal{X}, \eta)$ be a compact metric space where $\eta(\boldsymbol{x}, \boldsymbol{y})$ is the distance function of two data points $\boldsymbol{x} \in \mathbb{R}^d$ and $\boldsymbol{y} \in \mathbb{R}^d$ in $\mathcal{X}$ (a compact metric set; in this case, every open cover of $\mathcal{X}$ contains a finite subcover). We denote $\text{Prob}(\mathcal{X})$ as the space of probability measures defined on $\mathcal{X}$. The σ-th Wasserstein distance between two Borel probability distribution measures (for a Borel probability measure, all continuous functions are measurable) $p_r \in$

Prob($\mathcal{X}$) and $p_g \in$ Prob($\mathcal{X}$) with finite moments of order σ (i.e., $\int \eta(\boldsymbol{x}, \boldsymbol{y})^\sigma dp_r(\boldsymbol{x}) < \infty, \forall \boldsymbol{y} \in \mathcal{X}$) is defined as follows:

$$\begin{aligned} W_\sigma(p_r, p_g) &= \inf_{\gamma \in \Omega(p_r, p_g)} (\int \eta(\boldsymbol{x}, \boldsymbol{y})^\sigma d\eta(\boldsymbol{x}, \boldsymbol{y}))^{1/\sigma} \\ &= \inf_{\gamma \in \Omega(p_r, p_g)} (\mathbb{E}_{\boldsymbol{x} \sim p_r, \boldsymbol{y} \sim p_g} \eta(\boldsymbol{x}, \boldsymbol{y})^\sigma)^{1/\sigma} \end{aligned} \tag{2.20}$$

where $\sigma \geq 1$ while $\Omega(p_r, p_g)$ is the set containing all joint distributions $\gamma(\boldsymbol{x}, \boldsymbol{y})$ whose marginals are $p_r(\boldsymbol{x})$ and $p_g(\boldsymbol{y})$ respectively. Intuitively, $\gamma(\boldsymbol{x}, \boldsymbol{y})$ can be considered a policy where we specify how much "mass" should be transported from $\boldsymbol{x}$ to $\boldsymbol{y}$ to transform the distribution p_r into the distribution p_g.

The Kantorovich-Rubinstein theorem indicates that when $\mathcal{X}$ is separable and $\sigma = 1$, the dual representation [Villani, 2008] (i.e., the main idea is the transformation from minimising the cost in Eq. (2.20) to maximising the profit as in Eq. (2.21) of the Wasserstein distance described in Eq. (2.20) can be written as follows:

$$W_1(p_r, p_g) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{\boldsymbol{x} \sim p_r} [f(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{y} \sim p_g} [f(\boldsymbol{y})] \tag{2.21}$$

where f denotes the mapping from $\mathbb{R}^d$ to $\mathbb{R}$ and the Lipschitz semi-norm $\|f\|_L$ is defined as $\|f\|_L = \sup |f(\boldsymbol{x}) - f(\boldsymbol{y})| / \eta(\boldsymbol{x}, \boldsymbol{y})$ while $\eta(\boldsymbol{x}, \boldsymbol{y}) \geq |f(\boldsymbol{x}) - f(\boldsymbol{y})|$ for all $\boldsymbol{x}$ and $\boldsymbol{y}$.

The Lipschitz constraint plays an important role in the Wasserstein distance because it aims to block f from arbitrarily enhancing small differences. The Lipschitz constraint ensures that if two input data are similar, their outputs from f are similar as well.

Domain adaptation using Wasserstein distance. In the unsupervised learning context, the greatest challenge in domain adaptation is that two different domains have different data distributions. Therefore, the classifier obtained from the source domain may be significantly biased in the target domain. Several recently proposed methods [Redko et al., 2016, Shen et al., 2018] leverage the advantages of the Wasserstein distance in minimising the discrepancy between two different data distributions in order to bridge the gap between the source and target representations through adversarial training. As a result, the proposed models can learn feature representations invariant to the change of domains in an effective way.

The work in [Shen et al., 2018] uses the domain critic introduced in [Arjovsky et al., 2017] to estimate the Wasserstein distance between the source representation (e.g., $\boldsymbol{h}^S = f_g(\boldsymbol{x}^S)$) and the target representation (e.g., $\boldsymbol{h}^T = f_g(\boldsymbol{x}^T)$) distributions (i.e., $p_{\boldsymbol{h}^S}$ and $p_{\boldsymbol{h}^T}$) in the joint space where f_g is a feature extractor parameterised with θ_g by a DNN that maps the data from both source and target data in $\mathbb{R}^m$ into a joint feature space $\mathbb{R}^d$. The domain critic aims to learn a function $f_w : \mathbb{R}^d \to \mathbb{R}$ parameterised with θ_w to map a feature representation to a real corresponding number. The Wasserstein distance between two distributions then can be computed as follows:

$$\begin{aligned} W_1(p_r, p_g) &= \sup_{\|f_w\|_L \leq 1} \mathbb{E}_{p_{\boldsymbol{h}^S}} \left[f_w(\boldsymbol{h}^S) \right] - \mathbb{E}_{p_{\boldsymbol{h}^T}} \left[f_w(\boldsymbol{h}^T) \right] \\ &= \sup_{\|f_w\|_L \leq 1} \mathbb{E}_{p_{\boldsymbol{x}^S}} \left[f_w(f_g(\boldsymbol{x}^S)) \right] - \mathbb{E}_{p_{\boldsymbol{x}^T}} \left[f_w(f_g(\boldsymbol{x}^T)) \right] \end{aligned} \tag{2.22}$$

We can approximate the empirical Wasserstein distance in Eq. (2.22) by maximising the domain critic loss $\mathcal{L}_{wd}$ with respect to θ_w if the $\{f_w\}$ (i.e., the parameterised family of the domain critic function f_w) are all 1-Lipschitz:

$$\mathcal{L}_{wd}(\boldsymbol{x}^S, \boldsymbol{x}^T) = \frac{1}{N_S} \sum_{\boldsymbol{x}^S \in S} f_w(f_g(\boldsymbol{x}^S)) - \frac{1}{N_T} \sum_{\boldsymbol{x}^T \in T} f_w(f_g(\boldsymbol{x}^T))$$

There are some potential ways to enforce the Lipschitz constraint. Arjovsky et al. [2017] proposed to clip the weight θ_w of the domain critic f_w in an interval $[-c, c]$ with the value of $c = 0.01$ after each gradient update step. However, this can cause the problem of capacity underuse, as well as the gradient vanishing or exploding problems [Gulrajani et al., 2017]. Gulrajani et al. [2017] proposed an elegant solution to enforce the gradient penalty $\mathcal{L}_{grad}$ for the parameter θ_w of the domain critic f_w as follows:

$$\mathcal{L}_{grad}(\widetilde{\boldsymbol{h}}) = (\left\| \nabla_{\widetilde{\boldsymbol{h}}} f_w(\widetilde{\boldsymbol{h}}) \right\|_2 - 1)^2$$

where $\widetilde{\boldsymbol{h}} = \alpha \boldsymbol{h}^S + (1 - \alpha)\boldsymbol{h}^T$, $\alpha \in U[0, 1]$, $\boldsymbol{h}^S \sim p_{\boldsymbol{h}^S}$ and $\boldsymbol{h}^T \sim p_{\boldsymbol{h}^T}$.

We can estimate the empirical Wasserstein distance by solving the following problem:

$$\max_{\theta_w} \{\mathcal{L}_{wd} - \gamma \mathcal{L}_{grad}\}$$

where γ is the trade-off (i.e., balancing) coefficient.

Finally, domain invariant representation learning can be obtained by solving the following minimax problem:

$$\min_{\theta_g} \max_{\theta_w} \{\mathcal{L}_{wd} - \gamma \mathcal{L}_{grad}\} \tag{2.23}$$

Eq. 2.23 is then added to the main objective function as the second term along with the first (main) term (i.e., we minimise the target prediction loss) to form the final optimisation function of the model, for example, as follows:

$$\min_{\theta_g,\theta_c} \left\{ \mathcal{L}_c + \lambda \max_{\theta_w} [\mathcal{L}_{wd} - \gamma \mathcal{L}_{grad}] \right\}$$

where $\mathcal{L}_c$ is the target prediction loss (i.e., $\mathcal{L}_c = \frac{1}{N_S} \sum_{i=1}^{N_S} l(f_c(f_g(\boldsymbol{x}_i^S)), y_i^S)$ where l is the cross-entropy loss function) for the source data and θ_c is the parameters of the source data classifier f_c.

2.4.3 Using the maximum mean discrepancies distance

In recent years, DNNs have shown promising ability in learning transferable features that can generalise to new tasks in domain adaptation effectively; however, the features for transferability often drop significantly in high layers, resulting in increasing domain discrepancy compared with lower layers due to the transformation of learning data representation from general to specific along the networks (e.g., CNNs). Therefore, how to reduce the data bias to enhance the transferability in task-specific layers is an important problem.

In 2015, Long et al. [2015] proposed a novel deep domain adaptation architecture that aims to generalise CNNs for domain adaptation. In this proposed network, the representations in high hidden (i.e., task-specific) layers are mapped (embedded) to a reproducing kernel Hilbert space (RKHS) where we can explicitly match the mean embeddings of different domain distributions. The authors claimed that using an optimal multi-kernel selection method for the mean embedding matching can further reduce the discrepancy between different data domains.

In particular, the proposed architecture focuses on the multiple kernel variant of maximum mean discrepancies (MK-MMD) introduced in [Gretton et al., 2012b]. If we denote the RKHS by $\mathcal{H}_k$, defined on a topological space $\mathcal{X}$, which is endowed with a characteristic kernel k, the mean embedding of a distribution p in $\mathcal{H}_k$ will be a unique element $\mu_k(p)$ with $\mathbb{E}_{\boldsymbol{x}\sim p} f(\boldsymbol{x}) = \langle f(\boldsymbol{x}), \mu_k(p) \rangle_{\mathcal{H}_k}$ for all $f \in \mathcal{H}_k$. The MK-MMD distance between two different distributions p_S and p_T denoted by $d_k(p_S, p_T)$ is defined as the RKHS distance between the mean embeddings of p_S and p_T. We formularise the square of the MK-MMD distance as follows:

$$d_k^2(p_S, p_T) \triangleq \left\| \mathbb{E}_{\boldsymbol{x}^S \sim p_S} \phi(\boldsymbol{x}^S) - \mathbb{E}_{\boldsymbol{x}^T \sim p_T} \phi(\boldsymbol{x}^T) \right\|_{\mathcal{H}_k}^2 \tag{2.24}$$

where the distribution $p_S = p_T$ iff $d_k^2(p_S, p_T) = 0$ [Gretton et al., 2012a] (this is the most important property) and the characteristic kernel associated with ϕ (the feature map, i.e., $k(\boldsymbol{x}^S, \boldsymbol{x}^T) = \left\langle \phi(\boldsymbol{x}^S), \phi(\boldsymbol{x}^T) \right\rangle$) is defined as the convex combination of n positive semi-definite (PSD) kernels $\{k_u\}$:

$$\mathcal{K} \triangleq \left\{ k = \sum_{u=1}^{n} \beta_u k_u : \sum_{u=1}^{n} \beta_u = 1, \beta_u \geq 0, \forall u \right\}$$

In practice, we often use a family of n Gaussian kernels (i.e., positive definite kernels) $\{k_u\}_{u=1}^{n}$ where γ_u in $k(\boldsymbol{x}_i, \boldsymbol{x}_j) = e^{-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/\gamma_u}$ varies between 2^{-10} and 2^{-15} with a multiplicative step-size of $2^{0.5}$ [Gretton et al., 2012b].

Using the kernel trick, we can rewrite Eq. (2.24) as follows:

$$d_k^2(p_S, p_T) = \mathbb{E}_{\boldsymbol{x}_1^S \sim p_S, \boldsymbol{x}_2^S \sim p_S} k(\boldsymbol{x}_1^S, \boldsymbol{x}_2^S) + \mathbb{E}_{\boldsymbol{x}_1^T \sim p_T, \boldsymbol{x}_2^T \sim p_T} k(\boldsymbol{x}_1^T, \boldsymbol{x}_2^T) - 2\mathbb{E}_{\boldsymbol{x}_1^S \sim p_S, \boldsymbol{x}_1^T \sim p_T} k(\boldsymbol{x}_1^S, \boldsymbol{x}_2^T) \tag{2.25}$$

where $\boldsymbol{x}_1^S, \boldsymbol{x}_2^S \overset{iid}{\sim} p_S$, $\boldsymbol{x}_1^T, \boldsymbol{x}_2^T \overset{iid}{\sim} p_T$ and $k \in \mathcal{K}$.

Due to the $O(n^2)$ complexity of the computation mentioned in Eq. (2.25) (i.e., it is not expected when working with deep CNNs), Long et al. introduced an unbiased estimate of the MK-MMD distance in order to enforce the computation to match $O(n)$ linear complexity as follows:

$$d_k^2(p_S, p_T) = \frac{2}{N_S} \sum_{i=1}^{N_S/2} g_k(\boldsymbol{z}_i) \tag{2.26}$$

where $g_k(\boldsymbol{z}_i) \triangleq k(\boldsymbol{x}_{2i-1}^S, \boldsymbol{x}_{2i}^S) + k(\boldsymbol{x}_{2i-1}^T, \boldsymbol{x}_{2i}^T) - k(\boldsymbol{x}_{2i-1}^S, \boldsymbol{x}_{2i}^T) - k(\boldsymbol{x}_{2i}^S, \boldsymbol{x}_{2i-1}^T)$, $\boldsymbol{z}_i \triangleq (\boldsymbol{x}_{2i-1}^S, \boldsymbol{x}_{2i}^S, \boldsymbol{x}_{2i-1}^T, \boldsymbol{x}_{2i}^T)$, $\boldsymbol{x}_{2i-1}^S, \boldsymbol{x}_{2i}^S \overset{iid}{\sim} p_S$ and $\boldsymbol{x}_{2i-1}^T, \boldsymbol{x}_{2i}^T \overset{iid}{\sim} p_T$.

In the domain adaptation scenario, the unbiased estimate of the MK-MMD distance is applied to $\mathcal{D}_S^l = \left\{ \boldsymbol{h}_i^{Sl} \right\}$ and $\mathcal{D}_T^l = \left\{ \boldsymbol{h}_i^{Tl} \right\}$ where $\boldsymbol{h}_i^{Sl}$ and $\boldsymbol{h}_i^{Tl}$ are the l-th (i.e., we can choose different values of l depending on the architecture of the models as well as the number of labelled source data and the number of parameters in the layers that are fine-tuned) layer hidden representations of the source and target data points $\boldsymbol{x}_i^S$ and $\boldsymbol{x}_i^T$ respectively. Therefore, in this case the values of $\boldsymbol{z}_i$ described in Eq. (2.26) will be $\left(\boldsymbol{h}_{2i-1}^S, \boldsymbol{h}_{2i}^S, \boldsymbol{h}_{2i-1}^T, \boldsymbol{h}_{2i}^T \right)$, and the MK-MMD distance between the source and target evaluated on the l-th hidden layer representation is $d_k^2(\mathcal{D}_S^l, \mathcal{D}_T^l)$.

Finally, the MK-MMD distance (i.e, we minimise this distance) is added to the main objective function as the second term along with the first (main) term (i.e., we minimise the target prediction loss) to form the final optimisation function of the model, for example, as follows:

$$\min_{\Theta} \left\{ \frac{1}{N_S} \sum_{i=1}^{N_S} l(\theta(\boldsymbol{x}_i^S), y_i^S) + \lambda \sum_{l=l_1}^{l_2} d_k^2(\mathcal{D}_S^l, \mathcal{D}_T^l) \right\}$$

where Θ is the network parameters, while l is the cross-entropy loss function, and $\theta(\boldsymbol{x}_i^S)$ is the conditional probability that the network assigns the data $\boldsymbol{x}_i^S$ to the corresponding label y_i^S. We denote l_1 and l_2 as the layers used to minimise the MK-MMD distance.

2.4.4 Using virtual adversarial training

The limitation of domain adversarial training. In domain adversarial training, we have the following objective function:

$$\begin{aligned}\mathcal{J}(G,D,C) &= \frac{1}{N_S}\sum_{i=1}^{N_S}\ell\left(C\left(G\left(\boldsymbol{x}_i^S\right)\right),y_i\right)\\ &+\lambda\left(\frac{1}{N_S}\sum_{i=1}^{N_S}\log D\left(G\left(\boldsymbol{x}_i^S\right)\right)+\frac{1}{N_T}\sum_{i=1}^{N_T}\log\left[1-D\left(G\left(\boldsymbol{x}_i^T\right)\right)\right]\right)\end{aligned}$$

where we denote the source and target datasets by S $=\left\{\left(\boldsymbol{x}_1^S,y_1\right),\dots,\left(\boldsymbol{x}_{N_S}^S,y_{N_S}\right)\right\}$ with the labels, for example, $y_i\in\{-1,1\}$ (i.e., -1: non-vulnerable source code and 1: vulnerable source code) and $\mathbf{T}=\left\{\boldsymbol{x}_1^T,\dots,\boldsymbol{x}_{N_T}^T\right\}$ respectively, while $\mathcal{C}$ is the source classifier to classify the source samples, and D is the domain discriminator to discriminate the source and target samples. We use the cross-entropy loss function for l.

We seek the optimal generator G^*, the domain discriminator D^*, and the source classifier C^* by solving:

$$(C^*,G^*)=\text{argmin}_{C,G}\,\mathcal{J}(G,D,C)\text{ and }D^*=\text{argmax}_D\,\mathcal{J}(G,D,C)$$

Using the GAN principle, we encourage the Jensen-Shannon divergence between the latent features of the source and target data via the feature generator G to be small. Ganin and Lempitsky [2015] concluded that when the source classifier error and the feature divergence between the source and target data are both small, domain adaptation tends to be obtained.

However, Shu et al. [2018] pointed out that if the feature generator G has infinite capacity and the source-target supports are disjointed, then G can lead to arbitrary transformations to the target domain in order to match the source feature distribution. In this case, domain adversarial training may not be sufficient for domain adaptation.

Domain adaptation using virtual adversarial training. Shu et al. [2018] proposed applying the cluster assumption [Chapelle and Zien, 2005] (i.e., if the data distribution contains clusters, the data points in the same cluster should come from the same class, and the optimal decision boundaries should not cross high-density regions of the data) to domain adaptation. In particular, the work in [Shu et al., 2018] aims to minimise the conditional entropy in Eq. (2.27) for the target distribution in order to force the source classifier $\mathcal{C}$ to be confident on the unlabelled target data. As a result, the decision boundaries of the source classifier $\mathcal{C}$ are driven away from the target data.

$$\mathcal{L}_c(C, G, \mathbf{T}) = -\mathbb{E}_{\boldsymbol{x}^T \sim \mathbf{T}} \left[C\left(G\left(\boldsymbol{x}^T\right)\right) \log C\left(G\left(\boldsymbol{x}^T\right)\right)\right] \tag{2.27}$$

In practice, the conditional entropy mentioned in Eq. (2.27) must be empirically estimated using the available data. However, if the source classifier $\mathcal{C}$ is not locally-Lipschitz, the conditional entropy may break down [Grandvalet and Bengio, 2005] because without the locally-Lipschitz constraint, the source classifier $\mathcal{C}$ can abruptly change its prediction in the vicinity of the training data. To prevent this problem, Shu et al. [2018] introduced a solution that incorporates the locally-Lipschitz constraint via virtual adversarial training [Miyato et al., 2018] described as follows:

$$\mathcal{L}_v(C, G, \mathbf{D}) = \mathbb{E}_{\boldsymbol{x} \sim \mathbf{D}} \left[\max_{\|r\| \leq \varepsilon} D_{KL}(C\left(G\left(\boldsymbol{x}\right)\right) \| C\left(G\left(\boldsymbol{x}+r\right)\right) \right] \tag{2.28}$$

Note that we minimise Eq. (2.28) to enforce source classifier consistency within the norm-ball neighborhood of each data point $\boldsymbol{x}$, and virtual adversarial training can be applied to both the target and source data (i.e., $\mathbf{D}$ can be S or $\mathbf{T}$). The locally-Lipschitz constraint in Eq. (2.28) is then added to the final objective function as an additional term.

2.4.5 Using spectral graphs for semi-supervised domain adaptation

A spectral graph [Zhu et al., 2009] is a useful tool for capturing the geometrical and distributive information carried in the data. It is usually formulated as an undirected graph whose vertices are the data instances. In the context of semi-supervised learning, we are given a training set $X = X_l \cup X_u$ where $X_l = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{l}$ is labelled data, and $X_u = \{\boldsymbol{x}_i\}_{i=l+1}^{l+u}$ is unlabelled data. We construct the spectral graph $\mathcal{SG} = (\mathcal{V}, \mathcal{E})$ where the vertex set $\mathcal{V}$ includes all labelled and unlabelled instances (i.e., $\mathcal{V} = \{\boldsymbol{x}_i\}_{i=1}^{l+u}$). An edge $e_{ij} = \overline{\boldsymbol{x}_i\boldsymbol{x}_j} \in \mathcal{E}$ between two vertices $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ represents the similarity of the two instances. Let μ_{ij} be the weight associated with the edge e_{ij}. The underlying principle is to enforce that if μ_{ij} is large, then y_i and y_j are expected to receive the same label. The set of edges $\mathcal{SG}$ and its weights can be built as follows:

- Fully connected graph: Every pair of vertices $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ is connected by an edge. The edge weight decreases when the distance $\|\boldsymbol{x}_i - \boldsymbol{x}_j\|$ increases. The Gaussian kernel weight function widely used is given by

$$\mu_{ij} = \exp\left(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2 / \left(2\sigma^2\right)\right)$$

where σ is known as the bandwidth parameter and controls how quickly the weight decreases.

- k-NN: Each vertex $\boldsymbol{x}_i$ determines its k nearest neighbors (k-NN) and makes an edge with each of its k-NN. The Gaussian kernel weight function can be used for the edge weight. Empirically, k-NN graphs with small k tend to perform well.
- ε-NN: We connect $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ if $\|\boldsymbol{x}_i - \boldsymbol{x}_j\| \leq \varepsilon$. Again the Gaussian kernel weight function can be used to weight the connected edges. In practice, ε-NN graphs are easier to construct than k-NN graphs.

It is noteworthy that when constructing a spectral graph, we avoid connecting the edge of two labelled instances since we do not need to propagate a label between them. Fig. 2.16 illustrates an example of a spectral graph constructed using a 3D dataset using k-NN with $k = 5$.

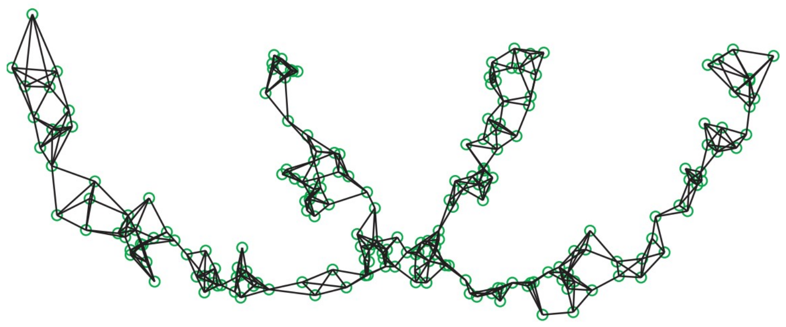


Figure 2.16: Example of a spectral graph.

After building the spectral graph, a semi-supervised learning problem is formulated as assigning labels to the unlabelled vertices. We need a mechanism to rationally propagate labels from the labelled vertices to the unlabelled ones. Again, the key idea here is to encourage $\boldsymbol{x}_i$ to have the same label as $\boldsymbol{x}_j$ if the weight μ_{ij} is large.

To assign labels to the unlabelled instances, it is desirable to learn a mapping function $f : \mathcal{X} \longrightarrow \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ are the domains of the data and labels respectively, such that

- $f(\boldsymbol{x}_i)$ is as close to its label y_i as possible for all labelled instances $\boldsymbol{x}_i$ $(1 \leq i \leq l)$.

- f should be smooth on the whole graph $\mathcal{SG}$, i.e., if $\boldsymbol{x}_i$ is very close to $\boldsymbol{x}_j$ (i.e., $\boldsymbol{x}_i$, $\boldsymbol{x}_j$ are very similar or μ_{ij} is large), the discrepancy between f_i and f_j (i.e., $|f_i - f_j|$) is small.

We arrive at the following optimisation problem:

$$\min_f \left(\infty \times \sum_{i=1}^{l} (f_i - y_i)^2 + \sum_{(i,j)\in\mathcal{E}} \mu_{ij} (f_i - f_j)^2 \right) \tag{2.29}$$

where by convention we define $\infty \times 0 = 0$ and $f_i = f(\boldsymbol{x}_i)$.

The optimisation problem in Eq. (2.29) reaches its minimum when the first term is exactly 0 and the second term is as small as possible. It can be therefore rewritten as a constrained optimisation problem:

$$\min_f \left(\sum_{(i,j)\in\mathcal{E}} \mu_{ij} (f_i - f_j)^2 \right) \tag{2.30}$$
$$\text{s.t.} : \forall_{i=1}^{l} : f_i = y_i$$

To extend the representation ability of the prediction function f, we relax the discrete function f to be real-valued. The drawback of the relaxation is that in the solution, $f(\boldsymbol{x}_i)$ is now real-valued and hence does not directly correspond to a label. This can be addressed by thresholding $f(\boldsymbol{x}_i)$ at zero to produce discrete label predictions, i.e., if $f(\boldsymbol{x}_i) \geq 0$, predict $y = 1$, and if $f(\boldsymbol{x}_i) < 0$, predict $y = -1$.

Part I

Deep Domain Adaptation for Software Vulnerability Detection

Preface to Part I

In Part I, Chapters 3 and 4, we summarise our proposed methods to address the first crucial problem existing in current software vulnerability detection (SVD) methods, which is relevant to the first research question (Q.1): "how to *transfer efficiently the learning on software vulnerabilities* from labelled projects (i.e., source domains) to other unlabelled projects (i.e., target domains)". Owing to the ubiquity of computer software, the SVD problem has become an important problem in the software industry and in computer security. Two significant issues in SVD arise when using machine learning, namely: i) how to learn automatic features that can improve the predictive performance of vulnerability detection, and ii) how to overcome the scarcity of labelled vulnerabilities in projects that require the laborious labelling of code by software security experts.

Chapter 3

Deep Domain Adaptation for Vulnerable Code Function Identification

Owing to the ubiquity of computer software, the SVD problem has become an important problem in the software industry and in computer security. Two significant issues in SVD arise when using machine learning, namely: i) how to learn automatic features that can improve the predictive performance of vulnerability detection, and ii) how to overcome the scarcity of labelled vulnerabilities in projects that require the laborious labelling of code by software security experts.

In this chapter, we address these two crucial concerns by proposing a novel architecture named the Code Domain Adaptation Network (CDAN) which leverages deep domain adaptation with automatic feature learning for SVD. Based on this architecture, we keep the principles and reapply state-of-the-art deep domain adaptation methods to indicate that deep domain adaptation for SVD is plausible and promising. Moreover, we further propose a novel method named the Semi-supervised Code Domain Adaptation Network (SCDAN) that can efficiently utilise and exploit information carried in unlabelled target data by considering them the unlabelled portion in a semi-supervised learning context. The proposed SCDAN method enforces the clustering assumption, which is a key principle in semi-supervised learning.

3.1 Motivations

Software vulnerabilities are specific flaws or oversights in a piece of software that allow attackers to undertake malicious activities: expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program [Dowd et al., 2006]. Due to the ubiquity of computer software, the growth and diversity in its development processes, a great deal of computer software potentially includes software vulnerabilities, and this fact makes the problem of software security vulnerability identification an important concern in the software industry and in the field of computer security. Consequently, we need automatic tools and methods that can accurately detect software vulnerabilities with a minimal level of human intervention. For this purpose, there exist many vulnerability detection systems and methods, ranging from open source to commercial tools, and from manual to automatic methods [Shin et al., 2011, Neuhaus et al., 2007, Yamaguchi et al., 2011, Grieco et al., 2016, Li et al., 2016, Kim et al., 2017, Li et al., 2018].

Another issue in SVD research is the scarcity of labelled projects, since the process of labelling vulnerable source code is a tedious, time-consuming, error-prone and challenging task even for domain experts. As a result, we have few labelled projects compared with a vast volume of unlabelled ones. A viable solution is to apply transfer learning or domain adaptation which aims to devise automatic methods that make it possible to perform transfer learning from the source domain with labels to the target domains without labels.

In this chapter, we first propose the Code Domain Adaptation Network (CDAN), a novel architecture which can tackle software source code and transfer learning from labelled software projects (source domain) to unlabelled projects (target domain). CDAN uses the generator G composed by a bidirectional RNN and a subsequent map to map sequential code tokens from source code to an intermediate layer in the network regarded as the joint feature layer. To bridge the gap between source and target domains in the joint feature layer, we employ an additional discriminator D. From a game perspective, the task of the discriminator D is to discriminate the source and target source codes, while the task of the generator G is to make indistinguishable the source and target source codes. At the Nash equilibrium point of this game, the gap between the source and target projects vanishes and consequently, the supervised source classifier C trained on the source project can be transferred to predict well on the target project.

We further observe that when successfully bridging the gap between the source and target projects, the target source code can be considered as the unlabelled portion of the source project in a semi-supervised learning context. The information carried in the target source code is certainly helpful in boosting the predictive performance. To further utilise the information carried the unlabelled target source codes, we propose the Semi-supervised Code Domain Adaptation Network (SCDAN) wherein the clustering assumption [Chapelle and Zien, 2005], which states that the decision boundary should not cross high-density regions inside data, is enforced. To this end, we simultaneously encourage the source classifier C of the SCDAN to be confident in its decisions for predicting the source and target source codes, and provide smooth predictive outputs $C(\boldsymbol{x})$ on the source and target projects. The first behavior can be achieved via minimisation of the conditional entropy with respect to the source and target distributions [Grandvalet and Bengio, 2005]. To address the second behavior, we use a spectral graph [Zhu et al., 2009] to represent the geometry of data and then solve an optimisation problem to encourage the smoothness of C over the spectral graph.

Leveraging semi-supervised learning with domain adaptation has been studied in shallow domain adaptation [Kumar et al., 2010, Yao et al., 2015]. Mostly related to ours is DIRT-T [Shu et al., 2018] which leverages semi-supervised learning with deep domain adaptation. DIRT-T enforces the clustering assumption by minimising the conditional entropy and using virtual adversarial perturbation (VAP) [Miyato et al., 2018] to impose by smoothness over $C(\boldsymbol{x})$. The underlying idea of VAP is to enforce smoothness locally around each training example $\boldsymbol{x}$. Due to the discrete and sequential nature of source codes, VAP cannot be directly applied to source code domain adaptation. To compare and contrast our proposed methods with DIRT-T in the context of software domain adaptation, we adopt the DIRT-T code to apply VAP in the joint feature layer. However, we observe two things that reduce the success of VAP when applied to source code data: i) VAP only encourages the local smoothness around the training examples and their representations, and ii) VAP requires solving a hard-to-solve optimisation for each training example. In the latter case, the workaround in DIRT-T is to undertake a one-step gradient ascent from a random solution, which is typically distant from the ideal solution.

To demonstrate the advantages of our proposed SCDAN, we conduct experiments using the datasets collected by [Lin et al., 2018] which consist of 6 real-world datasets: FFmpeg, LibTIFF, LibPNG, VLC, Pidgin and Asterisk. We experiment with transfer learning for 6 pairs of source and target domains, namely, FFmpeg$\rightarrow$ LibTIFF, FFmpeg $\rightarrow$ LibPNG, Pidgin $\rightarrow$ LibPNG, VLC$\rightarrow$ LibPNG, Asterisk $\rightarrow$ LibPNG and Pidgin $\rightarrow$ LibTIFF. For the baselines, we reapply the models DDAN, MMD and DIRT-T proposed in [Ganin and Lempitsky, 2015, Long et al., 2015, Shu et al., 2018] using our proposed architecture. The experimental results show that DDAN, DIRT-T and MMD significantly improve and outperform the Deep Code Network approach (VulDeePecker) in [Li et al., 2018] which is purely trained on the source project and then tested on the target project. SCDAN further outperforms DDAN, DIRT-T and MMD by a wide margin. Here we note that when training SCDAN, we only use the semi-supervised principle to enforce the clustering assumption and do not use any labels from the target project. To prove the ability of SCDAN to utilise the label information that may exist in the target project, we further assume that a small portion of the source codes in the target domain has labels and train SCDAN in this setting. As expected, our SCDAN with a few labels in the target domain is superior when compared to the SCDAN without those labels.

To summarise, our contributions in this chapter include the following points:

- Our work is the first to formulate transfer learning from a source of sequences to a target of sequences. Moreover, our contribution is to formulate a novel architecture named CDAN for software vulnerability detection, which is an extremely important problem in cybersecurity. Not only does our work involve both a model contribution and a significant real-world application of DDA, we believe it is a new building block for a wide array of other applications in other domains such as behavior modeling in fintech where temporal dynamics are important, or sequence modeling in computational biology.
- Based on the proposed architecture, we reapply the models DDAN, MMD and DIRT-T proposed in [Ganin and Lempitsky, 2015, Long et al., 2015, Shu et al., 2018]. We subsequently propose SCDAN to more efficiently exploit and utilise information from unlabelled target data. We further demonstrate the effectiveness and advantage of SCDAN by undertaking experiments on 6 real-world datasets. The experimental results show that SCDAN outperforms the baselines by a wide margin.

3.2 Related work for Deep Code Domain Adaptation

Most of previous work in the software vulnerability detection problem [Neuhaus et al., 2007, Shin et al., 2011, Yamaguchi et al., 2011, Li et al., 2016, Grieco et al., 2016, Kim et al., 2017] has been developed based on handcrafted features which are manually chosen by knowledgeable domain experts who may have outdated experience and underlying biases. In many situations handcrafted features normally do not generalize well: features that work well in a certain software project may not perform well in other projects [Zimmermann et al., 2009]. To mitigate the dependency on handcrafted features, the use of automatic features in SVD has been studied recently [Dam et al., 2017, Li et al., 2018, Lin et al., 2018]. Dam et al. [2018] and Lin et al. [2018] shared the same workaround for automatic feature learning. These researches employed a

Recurrent Neutral Network (RNN) to transform sequences of code tokens to vectorial features, which are further fed to a separate classifier (e.g., Support Vector Machine [Cortes and Vapnik, 1995] or Random Forest [Breiman, 2001]) for classification purposes. Due to the independence of learning the vector representation and training the classifier, it is likely that the resulting vector representations of [Dam et al., 2017, Lin et al., 2018] do not fit well with the classifiers, hence compromising the predictive performance. To address this issue, the work of [Li et al., 2018] combined the learning of data representations and the training of the classifier in deep neural networks. A bidirectional RNN is used to take sequential inputs, and the outputs from this bidirectional RNN are then feedforwarded via some dense layers to the output softmax layer for prediction.

Studies in domain adaptation can be broadly categorised into two themes: shallow [Borgwardt et al., 2006, Gopalan et al., 2011] and deep domain adaptations [Ganin and Lempitsky, 2015, Tzeng et al., 2015, Long et al., 2015, Shu et al., 2018, French et al., 2018]. The recent studies [Ganin and Lempitsky, 2015, Tzeng et al., 2015, Long et al., 2015, Shu et al., 2018, French et al., 2018] have showed the advances (i.e., higher predictive performance and capacity to tackle structural data) of deep over shallow domain adaptation. Deep domain adaptation encourages the learning of new representations for both source and target data in order to minimise the divergence between them [Ganin and Lempitsky, 2015, Tzeng et al., 2015, Long et al., 2015, Shu et al., 2018, French et al., 2018]. Source and target data are mapped to a joint feature space via a generator, and the gap between source and target distributions is bridged in this joint space by minimising the divergence between the forwarded distributions. For instance, the methods proposed in [Ganin and Lempitsky, 2015, Tzeng et al., 2015, Long et al., 2015, Shu et al., 2018, French et al., 2018] minimise the Jensen-Shannon divergence between the two relevant distributions relying on the GAN principle [Goodfellow et al., 2014a], while the work of [Long et al., 2015] minimises the maximum mean discrepancy (MMD), and the work of [Courty et al., 2017] minimises the Wasserstein distance between them. However, all aforementioned works mainly focus on transfer learning in the image domain, and to the best of our knowledge, none of the existing works have tackled sequential inputs, especially for source code in the context of software domain adaptation.

3.3 Deep Code Domain Adaptation

Most of the works in Deep Domain Adaptation cope with vector or image data. However, source code in Code Domain Adaptation is sequential data where from a given chunk of source code

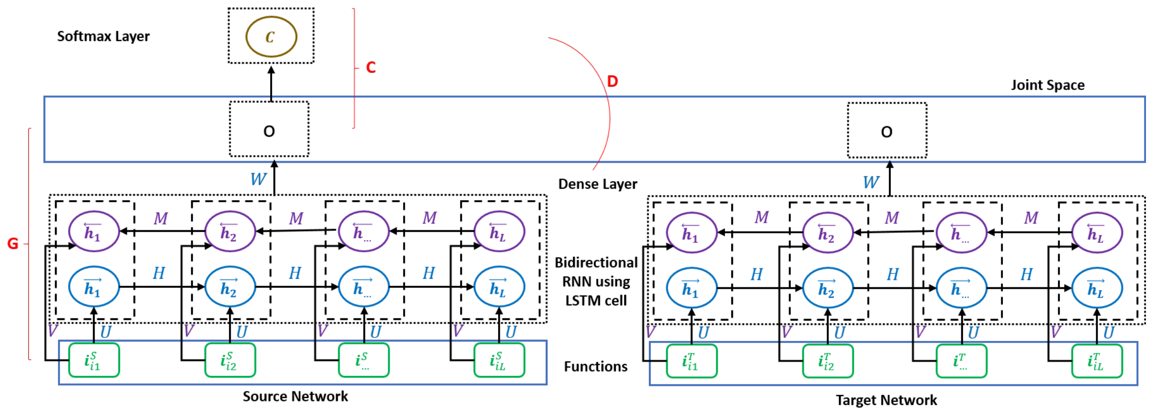


Figure 3.1: Architecture of our Code Domain Adaptation Network (CDAN). The generator G takes the sequential code tokens in vectorial form and maps this sequence to the joint layer (i.e., the joint space). Inspired by the GAN principle, the discriminator D is invoked to discriminate the source and target data. The source classifier C is trained on the source domain with labels. At the Nash equilibrium point, the source and target distributions are identical in the joint space and consequently, the classifier C can be transferred to predict well on the target data. We note that the source and target networks share parameters and are identical.

we first extract a sequence of code tokens and then transform this to a sequence of vectors using embedding matrices. We therefore need to devise a new generator that can efficiently tackle sequential data. In addition, we observe that when the generator can successfully bridge the gap between the source and target domains in the joint feature space, the target data samples without labels can be thought as the unlabelled portion of a semi-supervised problem in the joint feature space. Together with the labelled portion (i.e., the source data samples and their labels), this unlabelled portion contributes to improving the transfer learning performance. We start this section with the problem statement of Deep Code Domain Adaptation, followed by the technical details of the Code Domain Adaptation Network. Finally, we present how to utilise the information carried in the target samples to enable semi-supervised learning in the joint feature space.

3.3.1 The problem statement

Given a source dataset S $= \left\{\left(\boldsymbol{x}_1^S, y_1\right), \ldots, \left(\boldsymbol{x}_{N_S}^S, y_{N_S}\right)\right\}$ where $y_i \in \{0, 1\}$ (i.e., 1: vulnerable code and 0: non-vulnerable code) and $\boldsymbol{x}_i^S = \left[\boldsymbol{x}_{i1}^S, \ldots, \boldsymbol{x}_{iL}^S\right]$ is a sequence of L embedding vectors, and the target dataset $\mathbf{T} = \left\{\boldsymbol{x}_1^T, \ldots, \boldsymbol{x}_{N_T}^T\right\}$ where $\boldsymbol{x}_i^T = \left[\boldsymbol{x}_{i1}^T, \ldots, \boldsymbol{x}_{iL}^T\right]$ is also a sequence of L embedding vectors. We wish to bridge the gap between the source and target domains in the joint feature space so that we can transfer a classifier trained on the source domain to predict well on the target domain.

3.3.2 Data processing and embedding

We preprocess the datasets before injecting them into deep networks. First, we standardise the source code by: removing comments and non-ASCII characters, mapping user-defined variables to symbolic names (e.g., “var1”, “var2”) and user-defined functions to symbolic names (e.g., “func1”, “func2”), and replacing integers, real and hexadecimal numbers with a generic <num> token and strings with a generic <str> token. Second, we embed statements in source code into vectors. Specifically, each statement $\boldsymbol{x}$ consists of two parts including opcode and statement information. We embed both opcode and statement information to vectors, then concatenate the vector representations of opcode and statement information to obtain the final vector representation $\mathbf{i}$ of $\boldsymbol{x}$. For instance, in the (C language) statement “for(var1=num;var1<=num;var1++)”, the opcode is *for*, and the statement information is (var1=1;var1<=10;var1++). To embed the opcode, we multiply the one-hot vector of the opcode by the opcode embedding matrix. To embed the statement information, we tokenise it to a sequence of tokens, e.g., (,var1,=,num,;,var1,<=,num,;,var1,++,), construct the frequency vector of the statement information, and multiply this frequency vector by the statement information embedding matrix. We reserve the opcode *assign* for the assignment statements, e.g., “var3[var5++] = (png_byte)var7;”. In addition, the opcode embedding and statement embedding matrices are learnarable variables in our model.

3.3.3 Deep Code Domain Adaptation with bidirectional recurrent neural network

To handle the sequential data in the context of domain adaptation of software vulnerability detection, we propose an architecture termed as the Code Domain Adaptation Network (CDAN), which uses a bidirectional recurrent neural network (bidirectional RNN) to process the sequential input from both source and target domains (i.e., $\boldsymbol{x}_i^S = \left[\boldsymbol{x}_{i1}^S, \ldots, \boldsymbol{x}_{iL}^S\right]$ and $\boldsymbol{x}_i^T = \left[\boldsymbol{x}_{i1}^T, \ldots, \boldsymbol{x}_{iL}^T\right]$. Denoting the output of the bidirectional RNN by $\mathcal{B}(\boldsymbol{x})$, we then use fully connected layers to connect the output layer of the bidirectional RNN with the joint feature layer wherein we bridge the gap between the source and target domains. The generator is consequently the composition of the bidirectional RNN and the subsequent fully connected layers. We have $G(\boldsymbol{x}) = f(\mathcal{B}(\boldsymbol{x}))$ where $f(\cdot)$ represents the map formed by the fully connected layers. The architecture of the CDAN is presented in Fig. 3.1 wherein the source and target networks share parameters and are identical.

Akin to Deep Domain Adaptation [Ganin and Lempitsky, 2015], we employ the source classifier

$\mathcal{C}$ to classify the source samples and the domain discriminator D to discriminate the source and target samples. The generator G maps sequential code tokens in vectorial form of the source code to the joint layer. The discriminator D is employed to discriminate the source and target data. Together with the generator G which aims to make data in the joint layer indistinguishable, both the discriminator D and generator G cooperate to make the source and target distributions identical in the joint space. We name this model DDAN, and according to the Deep Domain Adaptation [Ganin and Lempitsky, 2015] approach, the objective function of DDAN is as follows:

$$\begin{aligned}\mathcal{J}(G, D, C) = &\frac{1}{N_S}\sum_{i=1}^{N_S} \ell\left(C\left(G\left(\boldsymbol{x}_i^S\right)\right), y_i\right) \\ &+ \lambda\left(\frac{1}{N_S}\sum_{i=1}^{N_S} \log D\left(G\left(\boldsymbol{x}_i^S\right)\right) + \frac{1}{N_T}\sum_{i=1}^{N_T} \log\left[1 - D\left(G\left(\boldsymbol{x}_i^T\right)\right)\right]\right)\end{aligned} \tag{3.1}$$

where we seek the optimal generator G^*, the domain discriminator D^*, and the source classifier C^* by solving:

$$(C^*, G^*) = \text{argmin}_{C,G}\, \mathcal{J}(G, D, C) \text{ and } D^* = \text{argmax}_D\, \mathcal{J}(G, D, C)$$

It is remarkable that DDAN uses information of the target data in the process of bringing source and target data together in the joint space. However, this cannot further utilise and exploit the information of the clustering structure carried in the target data. In what follows, we present how to further exploit the clustering structure information carried in the target data.

3.3.4 Semi-supervised Deep Code Domain Adaptation

We observe that when successfully bridging the gap between the source and target domains in the joint layer of the CDAN, the target samples can be regarded as the unlabelled portion of a semi-supervised learning problem. To enforce semi-supervised learning, the source classifier should satisfy the clustering assumption [Chapelle and Zien, 2005] which states that its decision boundary should not cross high-density regions of the source and target data in the joint feature space. The clustering assumption can be guaranteed if we impose the source classifier C to give a unique prediction result for all samples in a cluster. To address this, we propose to maintain the two following constraints:

1. The source classifier C should be confident in its predictive decisions on the source and target data.

2. The function $C(\cdot)$ should be smooth over the source and target data in the joint space (i.e., if $\boldsymbol{u}$ and $\boldsymbol{v}$ are close in the joint space, the discrepancy between $C(\boldsymbol{u})$ and $C(\boldsymbol{v})$ is small).

Following the study in [Grandvalet and Bengio, 2005], we achieve the first constraint as [Grandvalet and Bengio, 2005]:

$$\min_{C,G} \mathcal{H}(C,G) \tag{3.2}$$

where the conditional entropy $\mathcal{H}(C,G)$ can be computed as:

$$\begin{aligned}\mathcal{H}(C,G) = {} & \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_S}\left[-C(G(\boldsymbol{x}))\log\left[C(G(\boldsymbol{x}))\right]\right] \\ & + \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_S}\left[-\left[1-C(G(\boldsymbol{x}))\right]\log\left[1-C(G(\boldsymbol{x}))\right]\right] \\ & + \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_T}\left[-C(G(\boldsymbol{x}))\log\left[C(G(\boldsymbol{x}))\right]\right] \\ & + \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_T}\left[-\left[1-C(G(\boldsymbol{x}))\right]\log\left[1-C(G(\boldsymbol{x}))\right]\right]\end{aligned}$$

Using the spectral graph constructed on the source and target data in the joint feature space, we can achieve the second constraint. We construct the spectral graph $SG = (\mathcal{V},\ \mathcal{E})$ where the set of vertices $\mathcal{V} = S \cup T$ and the set of edges $\mathcal{E}$ are constructed as discussed earlier. To encourage the smoothness of the function $C(\cdot)$ on the spectral graph, we propose to minimise:

$$\min_{C,G} \mathcal{S}(C,G) \tag{3.3}$$

where we have defined $\mathcal{S}(C,G)$ as:

$$\begin{aligned}\mathcal{S}(C,G) &= \sum_{(\boldsymbol{u},\boldsymbol{v})\in\mathcal{E}} \mu_{\boldsymbol{uv}} KL\left(B_{\boldsymbol{u}}, B_{\boldsymbol{v}}\right) \\ &= \sum_{(\boldsymbol{u},\boldsymbol{v})\in\mathcal{E}} \mu_{\boldsymbol{uv}} \left[C(\boldsymbol{u})\log\frac{C(\boldsymbol{u})}{C(\boldsymbol{v})} + (1-C(\boldsymbol{u}))\log\frac{1-C(\boldsymbol{u})}{1-C(\boldsymbol{v})}\right]\end{aligned}$$

where $B_{\boldsymbol{u}}$ specifies the Bernoulli distribution with $\mathbb{P}(y=1 \mid \boldsymbol{u}) = C(\boldsymbol{u})$ and $\mathbb{P}(y=-1 \mid \boldsymbol{u}) = 1-C(\boldsymbol{u})$, the weight $\mu_{\boldsymbol{uv}} = \exp\left\{-\left\|\boldsymbol{u}-\boldsymbol{v}\right\|^2 / \left(2\sigma^2\right)\right\}$, and $KL\left(B_{\boldsymbol{u}}, B_{\boldsymbol{v}}\right)$ specifies the Kullback-Leibler divergence between two distributions. Here we note that if $\boldsymbol{u} = G(\boldsymbol{x})$ and $\boldsymbol{v} = G(\boldsymbol{t})$ are close in the joint space (i.e., the weight $\mu_{\boldsymbol{uv}}$ is high, $B(\boldsymbol{u})$ (i.e., $C(\boldsymbol{u})$) and $B(\boldsymbol{v})$ (i.e., $C(\boldsymbol{v})$) are encouraged to be close. This shows that the classifier function $C(\cdot)$ is smooth over the source and target data in the joint space.

Combining the optimisation problems in Eqs. (3.1, 3.2, and 3.3), we arrive at the following objective function:

$$\mathcal{I}(C,G,D) = \mathcal{J}(C,G,D) + \alpha\mathcal{H}(C,G) + \beta\mathcal{S}(C,G) \tag{3.4}$$

where α, $\beta > 0$ are the trade-off parameters. We seek the optimal generator G^*, domain discriminator D^*, and source classifier C^* by solving:

$$(C^*, G^*) = \text{argmin}_{C,G}\,\mathcal{I}(G,D,C) \text{ and } D^* = \text{argmax}_{D}\,\mathcal{I}(G,D,C)$$

It is worth noting that the two above techniques are complementary to ensure the clustering assumption. Given a data sample $\boldsymbol{x}$ in a cluster, this data sample stays close to its neighbors in the cluster. The smoothness obtained from the spectral graph imposes the slight difference in the predictive outputs of $\boldsymbol{x}$ of its neighbors, while the minimisation of the conditional entropy ensures a high confidence in the predictive output of $\boldsymbol{x}$. As a result, the classifier gives the same prediction for $\boldsymbol{x}$ and its neighbors, and this behavior is propagated on the entire cluster for which the classifier would give a unique prediction label to all data samples in this cluster.

Virtual adversarial perturbation.

To be able to compare with DIRT-T [Shu et al., 2018], we apply virtual adversarial perturbation (VAP) [Miyato et al., 2018] on the joint feature space rather than on the training example $\boldsymbol{x}$. The reason is that the training example $\boldsymbol{x}$ is a sequence of discrete statements, and it does not make sense to perturb $\boldsymbol{x}$.

For DIRT-T, in the final objective function (3.4), $\mathcal{S}(C,G)$ is replaced by the VAP $\mathcal{L}_v$ which is defined as:

$$\mathcal{L}_v(C,G) = \sum_{\boldsymbol{x}\in\mathcal{S}\cup\mathcal{T}} \max_{\|\mathbf{r}\|\leq\varepsilon} KL\left(C\left(G\left(\boldsymbol{x}\right)\right), C\left(G\left(\boldsymbol{x}\right)+\mathbf{r}\right)\right)$$

Specifically, we adopt the DIRT-T [Shu et al., 2018] code and apply it to our problem. As mentioned previously, we observe two things that make VAP unfit for our problem: i) the VAP mechanism only encourages local smoothness around the training examples $G(\boldsymbol{x})$, and ii) it is challenging to find the $\mathbf{r}_{opt}$ that maximises $KL\left(C\left(G\left(\boldsymbol{x}\right)\right), C\left(G\left(\boldsymbol{x}\right)+\mathbf{r}\right)\right)$. The workaround in DIRT-T [Shu et al., 2018] is to undertake a one-step gradient ascent from a randomly initialised $\mathbf{r}$, and the rendered $\mathbf{r}_{opt}$ is quite distant from the optimal $\mathbf{r}_{opt}$.

Categories	Datasets	#vul-funcs	#non-vul-funcs
Multimedia	FFmpeg	187	5,427
	VLC	25	5,548
	Pidgin	42	8,268
	Asterisk	52	8,796
Image	LibPNG	43	551
	LibTIFF	81	695

Table 3.1: Summary statistics of 6 real-world datasets with the number of vulnerable functions (#vul-funcs) and non-vulnerable functions (#non-vul-funcs).

3.4 Implementation and results

In this section, we present the experimental results for our proposed Semi-supervised Code Domain Adaptation Network (SCDAN) on 6 real-world datasets [Lin et al., 2018]. We also compare the proposed SCDAN with VulDeePecker without domain adaptation, DDAN, MMD and DIRT-T using our proposed architecture (CDAN). In addition, when training the DDAN, MMD, DIRT-T and SCDAN, we do not use any label information from the target domain.

3.4.1 Experimental setup

3.4.1.1 Experimental datasets

We used the real-world dataset collected by [Lin et al., 2018] which contains the source code of vulnerable and non-vulnerable functions obtained from 6 real-world software project datasets, namely, FFmpeg, LibTIFF, LibPNG, VLC, Pidgin and Asterisk. These datasets cover both multimedia and image application categories. The summary statistics of these projects are shown in Table 3.1. In our experiment, some of the datasets from the multimedia category were used as the source domain whilst other datasets from the image category were used as the target domain (see Table 3.2).

3.4.1.2 Model configuration

For training the five mentioned methods including VulDeePecker, MMD, DIRT-T, DDAN and SCDAN, we used a dynamic one-layer bidirectional recurrent neural network using LSTM cells to tackle varied sequence lengths where the size of hidden states is in $\{128, 256\}$. We embedded the opcode and statement information to the $\{150, 150\}$ dimensional embedding spaces respectively. For the source classifier and discriminator, we used a deep neural network with two hidden layers in which the size of each hidden layer is in $\{200, 300\}$. We employed the Adam optimiser

[Kingma and Ba, 2014] with an initial learning rate in $\{0.001, 0.0001\}$, while the mini-batch size is 50. The trade-off parameters λ, α and β are in $\{10^{-1}, 10^{-2}\}$. For SCDAN, we used the fully connected graph for constructing the spectral graph and varied the Gaussian kernel width in $\{2^{-10}, 2^{-9}\}$.

We split the data of the source domain into two random partitions: The first partition contains 80% for training and the second partition contains 20% for validation. We also split the data of the target domain into two random partitions containing 80% for training the model of VulDeePecker, MMD, DIRT-T, DDAN and SCDAN via the objective function in Eq. (3.4) without using any label information and 20% for testing the model. We additionally apply gradient clipping regularisation [Pascanu et al., 2013] to prevent over-fitting when training the model. We implemented our proposed method in Python using Tensorflow [Abadi et al., 2016], an open-source software library for Machine Intelligence developed by the Google Brain Team. We ran our experiments on a computer with an Intel Xeon Processor E5-1660 which had 8 cores at 3.0 GHz and 128 GB of RAM.

3.4.2 Experimental results

3.4.2.1 Code Domain Adaptation for a fully non-labelled target project

Quantitative results. We investigated the performance of our proposed method SCDAN compared with methods using our proposed architecture (CDAN) including VulDeePecker (VULD) [Li et al., 2016] without domain adaptation, DDAN [Ganin and Lempitsky, 2015], MMD [Long et al., 2015] and DIRT-T [Shu et al., 2018] with VAP to be applied in the joint feature layer. The VulDeePecker method was only trained on the source data and then tested on the target data. The DDAN as well as MMD, DIRT-T and SCDAN employed the target data without using any label information for domain adaptation and semi-supervised learning respectively.

In Table 3.2, the experimental results show that the VulDeePecker, MMD, DIRT-T, DDAN and SCDAN techniques achieved a high performance for detecting vulnerable and non-vulnerable functions for all performance measures including FNR, FPR, Recall, Precision and F1-measure. In most cases of the source and target domains, the versions using the domain adaptation technique including MMD, DIRT-T, DDAN and SCDAN achieved much higher performance results compared with VulDeePecker in almost measures, most significantly for SCDAN. Also, SCDAN achieved higher performances for FPR, FNR, Recall, Precision and F1-measure compared with MMD, DIRT-T and DDAN in almost cases of the different source and target domains. Furthermore, SCDAN always obtained the highest F1-measure in all cases. For example, for the

Source → Target	Methods	FNR	FPR	Recall	Precision	F1-measure
Pidgin → LibPNG	VULD	42.86%	1.08%	57.14%	80%	66.67%
	MMD	37.50%	**0%**	62.50%	**100%**	76.92%
	DIRT-T	**33.33%**	1.06%	**66.67%**	80%	72.72%
	DDAN	37.50%	**0%**	62.50%	**100%**	76.92%
	SCDAN	**33.33%**	**0%**	**66.67%**	**100%**	**80%**
FFmpeg → LibTIFF	VULD	43.75%	6.72%	56.25%	50%	52.94%
	MMD	28.57%	12.79%	71.43%	47.62%	57.14%
	DIRT-T	25%	9.09%	75%	52.94%	62.07%
	DDAN	35.71%	6.98%	64.29%	**60%**	62.07%
	SCDAN	**14.29%**	**5.38%**	**85.71%**	57.14%	**68.57%**
FFmpeg → LibPNG	VULD	25%	2.17%	75%	75%	75%
	MMD	**12.50%**	3.26%	**87.50%**	70%	77.78%
	DIRT-T	15.11%	2.20%	84.89%	80%	84.21%
	DDAN	**12.50%**	2.17%	**87.50%**	77.78%	82.35%
	SCDAN	**12.50%**	**1.08%**	**87.50%**	**87.50%**	**87.50%**
VLC→ LibPNG	VULD	57.14%	1.08%	42.86%	75%	54.55%
	MMD	45%	4.35%	55%	60%	66.67%
	DIRT-T	50%	1.09%	50%	**80%**	61.54%
	DDAN	**33.33%**	2.20%	**66.67%**	75%	70.59%
	SCDAN	**33.33%**	**1.06%**	**66.67%**	**80%**	**72.73%**
Pidgin → LibTIFF	VULD	35.29%	8.27%	64.71%	50%	56.41%
	MMD	30.18%	12.35%	69.82%	50%	58.27%
	DIRT-T	38.46%	8.05%	61.54%	53.33%	57.14%
	DDAN	**27.27%**	8.99%	**72.73%**	50%	59.26%
	SCDAN	30%	**5.56%**	70%	**58.33%**	**63.64%**
Asterisk →LibPNG	VULD	**11.11%**	15.38%	**88.89%**	36.36%	51.61%
	MMD	25%	8.67%	75%	42.86%	54.55%
	DIRT-T	14.29%	5.38%	85.71%	54.55%	66.67%
	DDAN	12.50%	7.61%	87.50%	50%	63.64%
	SCDAN	14.29%	**4.30%**	85.71%	**60%**	**70.59%**

Table 3.2: Performance results in terms of false negative rate (FNR), false positive rate (FPR), Recall, Precision and F1-measure of VulDeePecker (VULD), MMD, DIRT-T, DDAN and SC-DAN methods for predicting vulnerable and non-vulnerable code functions on the testing set of the target domain.

case of the source domain (FFmpeg) and target domain (LibPNG), SCDAN achieved the F1-measure (87.50%) compared with the F1-measure (82.35%, 84.21%, 77.78% and 75%) obtained with DDAN, DIRT-T, MMD and VulDeePecker respectively.

Visualisation. We use a t-SNE [Maaten and Hinton, 2008] projection, with perplexity equal to 30, to visualise the feature distributions of the source and target domains in the joint space. Specifically, we project the source and target data in the joint space (i.e., $G(\boldsymbol{x})$) into a 2D space without domain adaptation (VULD) and with domain adaptation (SCDAN). In Fig. 3.2, we observe the cases when performing domain adaptation from a single software project to another: FFmpeg → LibPNG. For the purpose of visualization, we only select a subset of the source project against the entire target project. As shown in Fig. 3.2, without domain adaptation the blue points (which represent the source data) and the green points (which represent the target data) are separate, while with domain adaptation the points are merged, as expected.

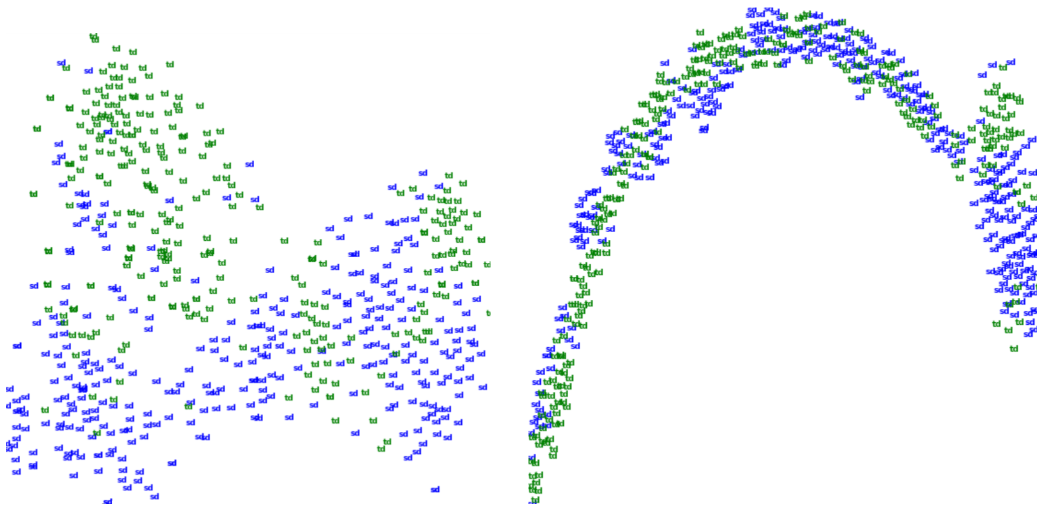


Figure 3.2: 2D t-SNE projection for the case of the FFmpeg → LibPNG domain adaptation. The blue and green points represent the source and target domains respectively. Without undertaking domain adaptation (VULD) the source and target data are separate (left). When undertaking domain adaptation (SCDAN), the source and target data are intermingled (right).

Source → Target	Methods	FNR	FPR	Recall	Precision	F1-measure
FFmpeg → LibPNG	SCDAN	12.50%	1.08%	87.50%	87.50%	87.50%
	SCDAN-10	22.22%	**0%**	77.78%	**100%**	87.50%
	SCDAN-20	16.67%	**0%**	83.33%	**100%**	90.91%
	SCDAN-30	14.29%	**0%**	85.71%	**100%**	92.31%
	SCDAN-40	12.50%	**0%**	87.50%	**100%**	93.33%
	SCDAN-50	**0%**	**0%**	**100%**	**100%**	**100%**
FFmpeg → LibTIFF	SCDAN	14.29%	5.38%	85.71%	57.14%	68.57%
	SCDAN-10	28.57%	1.08%	71.43%	83.33%	76.92%
	SCDAN-20	**0%**	3.23%	**100%**	70%	82.35%
	SCDAN-30	20%	**0%**	80%	**100%**	88.89%
	SCDAN-40	**0%**	1.05%	**100%**	83.33%	90.91%
	SCDAN-50	8.33%	**0.0%**	91.67%	**100%**	**95.65%**

Table 3.3: Results in terms of false negative rate (FNR), false positive rate (FPR), Recall, Precision and F1-measure of SCDAN-10, SCDAN-20, SCDAN-30, SCDAN-40, SCDAN-50 and SCDAN for predicting vulnerable and non-vulnerable functions on the testing set of the target domain. Note that SCDAN-0 is SCDAN.

We observe a strong correspondence between the success of domain adaptation in terms of the classification accuracy of the target domain and the overlap between the domain distributions in such visualization.

3.4.2.2 Code Domain Adaptation for a partly labelled target project

In this experiment, we trained SCDAN for the case when there is a small portion of the training set in the target domain having labels. We then investigated the performance of SCDAN in predicting the vulnerable and non-vulnerable functions on the testing set of the target domain. We investigated five cases. In the first case (SCDAN-10), we set 10% data of the training set of the target domain to have labels. In the second case (SCDAN-20), we assume 20% data of the training set of the target domain have labels. For other cases (SCDAN-30, SCDAN-40 and SCDAN-50), we set 30%, 40% and 50% of the data of the training set of the target domain to have labels respectively.

The results shown in Table 3.3 indicate that using a small fraction of the training set in the target domain that has labels, SCDAN can achieve a much higher performance than (VulDeePecker, MMD, DIRT-T and DDAN) in almost all measures, especially in term of the F1-measure. SCDAN-50 obtains nearly a 9.9% and 14.28% improvement over SCDAN-20 and SCDAN respectively in terms of the F1-measure for the case of the source domain FFmpeg and target domain LibPNG. In the case of the source domain FFmpeg and target domain LibTIFF, SCDAN-50 achieves a 16.15% and 39.49% improvement over SCDAN-20 and SCDAN respectively in term of the F1-measure. In general, SCDAN-10, SCDAN-20, SCDAN-30, SCDAN-40 and SCDAN-50 achieve a high performance for all measures (FNR, FPR, Recall, Precision and F1-measure), especially for SCDAN-50.

3.5 Closing remarks

In this chapter, we addressed two main concerns in automated software vulnerability detection: i) how to automatically learn features that can help improve the predictive performance of vulnerability identification, and ii) how to overcome the scarcity of labelled vulnerabilities in software projects, vulnerabilities that require the laborious labelling of code by software experts. We proposed a novel architecture named the Code Domain Adaptation Network (CDAN) which leverages deep domain adaptation with automatic feature learning for software vulnerability identification, and the Semi-supervised Code Domain Adaptation Network (SCDAN) method. The SCDAN method enforces the clustering assumption to enable the source code information to be combined with the target source code in a semi-supervised context. The experimental results for six real-world software datasets showed that using our proposed architecture (CDAN), deep domain adaptation for software vulnerability detection is plausible and promising. All state-of-the-art deep domain adaptation methods using CDAN significantly outperformed the Deep Code Network without domain adaptation (VULD). Furthermore, the SCDAN method also significantly surpassed the performance of the DDAN, MMD and DIRT-T baseline methods.

Chapter 4

Dual-component Deep Domain Adaptation: A New Approach for Cross-project Software Vulnerability Detection

In the previous chapter, we have proposed a new architecture CDAN and a new method SCDAN to address the transfer learning problem of software vulnerabilities from a labelled source of sequences to an unlabelled target of sequences. However, due to using the generative adversarial network (GAN) principle to bridge the gap between source and target domains as in other deep domain adaptation adversarial approaches, our proposed CDAN architecture and SCDAN method may suffer from the problems of mode collapsing and boundary distortion [Goodfellow, 2016, Santurkar et al., 2018, Hoang et al., 2018, Le et al., 2019a] that negatively impact on the predictive performance.

In this chapter, we propose a new method, the Dual Generator-Discriminator Deep Code Domain Adaptation Network (Dual-GD-DDAN), for tackling the problem of transferring learning from labelled to unlabelled software projects in SVD to resolve the mode collapsing and boundary distortion problems faced in previous approaches.

4.1 Motivations

In the software industry, software vulnerabilities relate to specific flaws or oversights in software programs which allow attackers to expose or alter sensitive information, disrupt or destroy a system, or take control of a program or computer system. The software vulnerability detection problem has become an important issue in the software industry and in the field of computer security. Computer software development employs of a vast variety of technologies and different software development methodologies, and much computer software contains vulnerabilities.

This has necessitated the development of automated advanced techniques and tools that can efficiently and effectively detect software vulnerabilities with a minimal level of human intervention. To respond to this demand, many vulnerability detection systems and methods, ranging from open source to commercial tools, and from manual to automatic methods have been proposed and implemented. Most of the previous works in software vulnerability detection (SVD) [Almorsy et al., 2012, Kim et al., 2017] have been developed based on handcrafted features which are manually chosen by knowledgeable domain experts who may have outdated experience and underlying biases. In many situations, handcrafted features normally do not generalize well. For example, features that work well in a certain software project may not perform well in other projects. To alleviate the dependency on handcrafted features, the use of automatic features in SVD has been studied recently [Li et al., 2018, Lin et al., 2018, Le et al., 2019b]. These works have shown the advantages of automatic features over handcrafted features in the context of software vulnerability detection.

However, most of these approaches lead to another crucial issue in SVD research, namely, the scarcity of labelled projects. Labelled vulnerable code is needed to train these models, and the process of labelling vulnerable source code is very tedious, time-consuming, error-prone, and challenging even for domain experts. This has led to few labelled projects compared with the vast volume of unlabelled ones. A viable solution is to apply transfer learning or domain adaptation which aims to devise automated methods that make it possible to transfer a learned model from the source domain with labels to the target domains without labels. Studies in domain adaptation can be broadly categorised into two themes: shallow [Gopalan et al., 2011] and deep domain adaptations [Ganin and Lempitsky, 2015, Long et al., 2015, Shu et al., 2018]. These recent studies have shown the advantages of deep over shallow domain adaptation (i.e., higher predictive performance and capacity to tackle structural data). Deep domain adaptation encourages the learning of new representations for both source and target data in order to minimise the divergence between them [Ganin and Lempitsky, 2015, Long et al., 2015, Shu et al., 2018]. The general idea is to map source and target data to a joint feature space via a generator, where the discrepancy between the source and target distributions is reduced. Notably, the work of [Ganin and Lempitsky, 2015, Shu et al., 2018] employed generative adversarial networks (GANs) [Goodfellow et al., 2014a] to close the gap between source and target data in the joint space. However, most of aforementioned works mainly focus on transfer learning in the computer vision domain. The work of [Nguyen et al., 2019] is the first work which applies deep domain adaptation to SVD with promising predictive performance on real-world source code projects. The underlying idea is to employ the GAN to close the gap between the source and target domains in the joint space and enforce the clustering assumption [Chapelle and Zien, 2005] to utilise the information carried in the unlabelled target samples in a semi-supervised context.

GANs are known to be affected by the mode collapsing problem [Goodfellow, 2016, Santurkar et al., 2018, Hoang et al., 2018, Le et al., 2019a]. In particular, the study in [Santurkar et al., 2018] recently studied the mode collapsing problem and further classified this into the missing mode problem i.e., the generated samples miss some modes in the true data, and the boundary distortion problem i.e., the generated samples can only partly recover some modes in the true data. It is certain that deep domain adaptation approaches that use the GAN principle will inherently encounter both the missing mode and boundary distortion problems. Last but not least, deep domain adaptation approaches using the GAN principle also face the data distortion problem. The representations of source and target examples in the joint feature space degenerate to very small regions that cannot preserve the manifold/clustering structure in the original space.

Our aim in this chapter is to address not only the deep domain adaptation mode collapsing problem, but also the boundary distortion problem when employing the GAN as a principle in order to close the gap between source and target data in the joint feature space. Our two approaches are: i) apply manifold regularisation for enabling the preservation of manifold/clustering structures in the joint feature space, hence avoiding the degeneration of source and target data in this space, and ii) invoke dual discriminators in an elegant way to reduce the negative impacts of the missing mode and boundary distortion problems in deep domain adaptation using the GAN principle as mentioned before. We name our mechanism when applied to SVD as Dual Generator-Discriminator Deep Code Domain Adaptation Network (Dual-GD-DDAN). We empirically demonstrate that our Dual-GD-DDAN can overcome the missing mode and boundary distortion problems which is likely to happen as in Deep Code Domain Adaptation (DDAN) [Nguyen et al., 2019] in which the GAN was solely applied to close the gap between the source and target domains in the joint space (see the discussion in Sections 4.3.3 and 4.4.3, and the visualization in Fig. 4.3). In addition, we incorporate the relevant approaches – minimising the conditional entropy and manifold regularisation with spectral graph – proposed in [Nguyen et al., 2019] to enforce the clustering assumption [Chapelle and Zien, 2005] and arrive at a new model named Dual Generator-Discriminator Semi-supervised Deep Code Domain Adaptation Network (Dual-GD-SDDAN). We further demonstrate that our Dual-GD-SDDAN can overcome the mode collapsing problem better than SCDAN in [Nguyen et al., 2019], hence obtaining better predictive performance.

We conducted experiments using the datasets collected by [Lin et al., 2018], that consist of five real-world software projects: FFmpeg, LibTIFF, LibPNG, VLC and Pidgin to compare our proposed Dual-GD-DDAN and Dual-GD-SDDAN with the baselines. The baselines consider to include VULD (i.e., the model proposed in [Li et al., 2018] without domain adaptation), MMD, DIRT-T, DDAN and SCDAN as mentioned [Nguyen et al., 2019] and D2GAN [Nguyen et al., 2017] (a variant of the GAN using dual-discriminator to reduce the mode collapse for which we apply this mechanism in the joint feature space). Our experimental results show that our proposed methods are able to overcome the negative impact of the missing mode and boundary distortion problems inherent in deep domain adaptation approaches when solely using the GAN principle as in DDAN and SCDAN [Nguyen et al., 2019]. In addition, our method outperforms the rival baselines in terms of predictive performance by a wide margin.

4.2 Related work for Deep Code Domain Adaptation

In this section, we introduce work related to ours. First, we present the recent work in automatic feature learning for software vulnerability detection. Finally, we present some recent work in deep domain adaptation.

Automatic feature learning in software vulnerability detection minimises intervention from security experts [Li et al., 2018, Lin et al., 2018, Dam et al., 2018]. Particularly, Dam et al. [2018] and Lin et al. [2018] shared the same approach employing a Recurrent Neutral Network (RNN) to transform sequences of code tokens to vectorial features for automatic feature learning, which are then fed to a separate classifier (e.g., Support Vector Machine [Cortes and Vapnik, 1995] or Random Forest [Breiman, 2001]) for classification purposes. However, owing to the independence of learning the vector representations and training the classifier, it is likely that the resulting vector representations of [Lin et al., 2018, Dam et al., 2018] may not fit well with classifiers to enhance the predictive performance. To deal with this problem, the study introduced in [Li et al., 2018] combined the learning of the vector representations and the training of a classifier in deep neural networks. This work leverages a bidirectional RNN to take sequential data as inputs and the outputs from the bidirectional RNN are then fed to a deep feedforward neural network for prediction.

Deep domain adaptation aims to bridge the gap between the source and target domains in a joint space [Ganin and Lempitsky, 2015, Tzeng et al., 2015, Shu et al., 2018, French et al., 2018]. These methods try to minimise a divergence (e.g., Jensen-Shannon divergence, f-divergence, maximum mean discrepancy (MMD) or Wasserstein distance) between the source and target distributions in the joint space. For instance, the methods proposed in [Ganin and Lempitsky, 2015, Tzeng et al., 2015, Long et al., 2015, Shu et al., 2018, French et al., 2018] minimise the Jensen-Shannon divergence between two relevant distributions relying on the GAN principle [Goodfellow et al., 2014a], while Long et al. [2015] proposed to minimise the MMD, and the work of [Courty et al., 2017] minimises the Wasserstein distance between two relevant distributions. The study proposed in [Nguyen et al., 2017] relies on the GAN principle with using two discriminators aiming to tackle the problem of mode collapse encountered in generative adversarial networks (GANs). In addition, most of aforementioned works proposed to transfer a pretrained model on the dataset ImageNet [Deng et al., 2009] to other image sources. The work of [Purushotham et al., 2017] proposed to apply deep domain adaptation for multivariate time-series data. This work based on the Variational RNN [Chung et al., 2015], and the GAN principle was applied on the latent representation. Recently, relying on the GAN principle [Goodfellow et al., 2014a], the work of [Nguyen et al., 2019] proposed to tackle sequential inputs, particularly source code, in software domain adaptation. The underlying idea is to use the GAN principle to close the gap between the source and target domains in the joint space and enforce the clustering assumption to utilise the information of unlabelled target samples in a semi-supervised learning context.

Leveraging semi-supervised learning with domain adaptation has been studied in shallow domain adaptation [Kumar et al., 2010, Yao et al., 2015]. DIRT-T [Shu et al., 2018] leveraged semi-supervised context by enforcing the clustering assumption via minimising the conditional entropy and using virtual adversarial perturbation (VAP) [Miyato et al., 2018] to impose by smoothness over the decision output. Another work of [Nguyen et al., 2019] proposed to leverage semi-supervised context by enforcing the clustering assumption via minimising the conditional entropy and manifold regularisation with spectral graph which was proven to be more appropriate to discrete sequential data like source codes.

4.3 Deep Code Domain Adaptation with GAN

4.3.1 Problem statement

A source domain dataset S $= \{(\boldsymbol{x}_1^S, y_1), \ldots, (\boldsymbol{x}_{N_S}^S, y_{N_S})\}$ where $y_i \in \{0, 1\}$ (i.e., 1: vulnerable code and 0: non-vulnerable code) and $\boldsymbol{x}_i^S = [\boldsymbol{x}_{i1}^S, \ldots, \boldsymbol{x}_{iL}^S]$ is a sequence of L embedding vectors, and the target domain dataset $\mathbf{T} = \{\boldsymbol{x}_1^T, \ldots, \boldsymbol{x}_{N_T}^T\}$ where $\boldsymbol{x}_i^T = [\boldsymbol{x}_{i1}^T, \ldots, \boldsymbol{x}_{iL}^T]$ is also a sequence of L embedding vectors. We wish to bridge the gap between the source and target domains in the joint feature space. This allows us to transfer a classifier trained on the source domain to predict well on the target domain.

4.3.2 Deep Code Domain Adaptation with bidirectional RNN

To handle sequential data in the context of domain adaptation of software vulnerability detection, the work of [Nguyen et al., 2019] proposed an architecture referred to as the Code Domain Adaptation Network (CDAN). This network architecture recruits a bidirectional RNN to process the sequential input from both source and target domains (i.e., $\boldsymbol{x}_i^S = [\boldsymbol{x}_{i1}^S, \ldots, \boldsymbol{x}_{iL}^S]$ and $\boldsymbol{x}_i^T = [\boldsymbol{x}_{i1}^T, \ldots, \boldsymbol{x}_{iL}^T]$). A fully connected layer is then employed to connect the output layer of the bidirectional RNN with the joint feature layer while bridging the gap between the source and target domains. Furthermore, inspired by the Deep Domain Adaptation approach [Ganin and Lempitsky, 2015], the authors employ the source classifier $\mathcal{C}$ to classify the source samples, the domain discriminator D to distinguish the source and target samples and propose Deep Code Domain Adaptation (DDAN) whose objective function is as follows:

$$\begin{aligned}\mathcal{J}(G, D, C) = &\frac{1}{N_S}\sum_{i=1}^{N_S}\ell(C(G(\boldsymbol{x}_i^S)), y_i) \\ &+ \lambda\left(\frac{1}{N_S}\sum_{i=1}^{N_S}\log D(G(\boldsymbol{x}_i^S)) + \frac{1}{N_T}\sum_{i=1}^{N_T}\log[1 - D(G(\boldsymbol{x}_i^T))]\right)\end{aligned}$$

4.3.3 The shortcomings of DDAN

We observe that DDAN suffers from several shortcomings. First, the data distortion problem (i.e., the source and target data in the joint space might collapse into small regions) may occur since there is no mechanism in DDAN to circumvent this. Second, since DDAN is based on the GAN approach, DDAN might suffer from the mode collapsing problem [Goodfellow, 2016, Santurkar et al., 2018]. In particular, Santurkar et al. [2018] have recently studied the mode collapsing problem of GANs and discovered that they are also subject to i) the missing mode

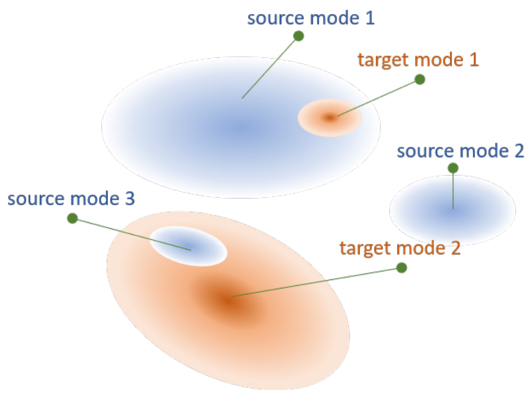


Figure 4.1: Illustration of the missing mode and boundary distortion problems of DDAN. In the joint space, the target distribution misses source mode 2, while the source distribution can only partly cover the target mode 2 in the target distribution and the target distribution can only partly cover the source mode 1 in the source distribution.

problem (i.e., in the joint space, either the target data misses some modes in the source data or vice versa), and ii) the boundary distortion problem (i.e., in the joint space either the target data partly covers the source data or vice versa), which makes the target distribution significantly diverge from the source distribution. As shown in Fig. 4.1, both the missing mode and boundary distortion problems simultaneously happen since the target distribution misses source mode 2, while the source distribution can only partly cover the target mode 2 in the target distribution and the target distribution can only partly cover the source mode 1 in the source distribution.

4.4 Dual Generator-Discriminator Deep Code Domain Adaptation

4.4.1 Key idea of our approach

We employ two discriminators (namely, D_S and D_T) to classify the source and target examples and vice versa and two separate generators (namely, G_S and G_T) to map the source and target examples to the joint space respectively. In particular, D_S produces high values on the source examples in the joint space (i.e., $G_S(\boldsymbol{x}^S)$) and low values on the target examples in the joint space (i.e., $G_T(\boldsymbol{x}^T)$), while D_T produces high values on the target examples in the joint space (i.e., $G_T(\boldsymbol{x}^T)$) and low values on the source examples (i.e., $G_S(\boldsymbol{x}^S)$). The generator G_S is trained to push $G_S\left(\boldsymbol{x}^S\right)$ to the high value region of D_T and the generator G_T is trained to push $G_T(\boldsymbol{x}^T)$ to the high value region of D_S. Eventually, both $D_S(G_S(\boldsymbol{x}^S))$ and $D_S(G_T(\boldsymbol{x}^T))$ are possibly high and both $D_T(G_S(\boldsymbol{x}^S))$ and $D_T(G_T(\boldsymbol{x}^T))$ are possibly high. This helps to mitigate the issues of missing mode and boundary distortion since as in Fig. 4.1, if the target mode 1 can only partly cover the source mode 1, then D_T cannot receive large values from source mode 1. Another important aspect of our approach is to maintain the cluster/manifold structure of source and target data in the joint space via the manifold regularisation to avoid the data distortion problem.

4.4.2 Dual Generator-Discriminator Deep Domain Adaptation Network

To address the two inherent problems in the DDAN mentioned in Section 4.3.3, we employ two different generators G_S and G_T to map source and target domain examples to the joint space and two discriminators D_S and D_T to distinguish source examples against target examples and vice versa together with the source classifier $\mathcal{C}$ which is used to classify the source examples with labels as shown in Fig. 4.2. We name our proposed model as Dual Generator-Discriminator Deep Code Domain Adaptation Network (Dual-GD-DDAN).

Updating the discriminators. The two discriminators D_S and D_T are trained to distinguish the source examples against the target examples and vice versa as follows:

$$\min_{D_S} \left(\frac{(1+\theta)}{N_S} \sum_{i=1}^{N_S} [-\log D_S(G_S(\boldsymbol{x}_i^S))] + \frac{1}{N_T} \sum_{i=1}^{N_T} [-\log[1 - D_S(G_T(\boldsymbol{x}_i^T))]] \right) \tag{4.1}$$

$$\min_{D_T} \left(\frac{1}{N_S} \sum_{i=1}^{N_S} [-\log[1 - D_T(G_S(\boldsymbol{x}_i^S))]] + \frac{(1+\theta)}{N_T} \sum_{i=1}^{N_T} [-\log D_T(G_T(\boldsymbol{x}_i^T))] \right) \tag{4.2}$$

where $\theta > 0$. Note that a high value of θ encourages D_S and D_T place higher values on $G_S\left(\boldsymbol{x}^S\right)$ and $G_T\left(\boldsymbol{x}^T\right)$ respectively.

Updating the source classifier. The source classifier is employed to classify the source examples with labels as: $\min_{\mathcal{C}} \frac{1}{N_S} \sum_{i=1}^{N_S} \ell(\mathcal{C}(G_S(\boldsymbol{x}_i^S)), y_i)$, where ℓ specifies the cross-entropy loss function for the binary classification.

Updating the generators. The two generators G_S and G_T are trained to i) maintain the manifold/cluster structures of source and target data in their original spaces to avoid the data distortion problem, and ii) move the target samples toward the source samples in the joint space and resolve the missing mode and boundary distortion problems in the joint space.

To maintain the manifold/cluster structures of source and target data in their original spaces, we propose minimising the manifold regularisation term as: $\min_G \mathcal{M}(G_S, G_T)$ where $\mathcal{M}(G_S, G_T)$

is formulated as:

$$\mathcal{M}(G_S, G_T) = \sum_{i,j=1}^{N_S} \mu_{ij} ||G_S(\boldsymbol{x}_i^S) - G_S(\boldsymbol{x}_j^S)||^2 + \sum_{i,j=1}^{N_T} \mu_{ij} ||G_T(\boldsymbol{x}_i^T) - G_T(\boldsymbol{x}_j^T)||^2$$

in which the weights are defined as $\mu_{ij} = \exp\{-||h(\boldsymbol{x}_i) - h(\boldsymbol{x}_j)||^2 / (2\sigma^2)\}$ with $h(\boldsymbol{x}) = \text{concat}(\overleftarrow{h_L}(\boldsymbol{x}), \overrightarrow{h_L}(\boldsymbol{x}))$ where $\overrightarrow{h_L}(\boldsymbol{x})$ and $\overleftarrow{h_L}(\boldsymbol{x})$ are the last hidden states of the bidirectional RNN with input $\boldsymbol{x}$.

To move the target samples toward the source samples and resolve the missing mode and boundary distortion problems in the joint space, we propose minimising the following objective function: $\min_D \mathcal{K}(G_S, G_T)$ where $\mathcal{K}(G_S, G_T)$ is defined as:

$$\mathcal{K}(G_S, G_T) = \frac{1}{N_S} \sum_{i=1}^{N_S} [-\text{log} D_T(G_S(\boldsymbol{x}_i^S))] + \frac{1}{N_T} \sum_{i=1}^{N_T} [-\text{log} D_S(G_T(\boldsymbol{x}_i^T))] \tag{4.3}$$

Moreover, the source generator G_S has to work out the representation that is suitable for the source classifier, hence we need to minimise the following objective function:

$$\min_{G_S} \frac{1}{N_S} \sum_{i=1}^{N_S} \ell(\mathcal{C}(G_S(\boldsymbol{x}_i^S)), y_i)$$

Finally, to update G_S and G_T, we need to minimise the following objective function:

$$\frac{1}{N_S} \sum_{i=1}^{N_S} \ell(\mathcal{C}(G_S(\boldsymbol{x}_i^S)), y_i) + \alpha \mathcal{M}(G_S, G_T) + \beta \mathcal{K}(G_S, G_T)$$

where α, $\beta > 0$ are two non-negative parameters.

4.4.3 The rationale for our Dual Generator-Discriminator Deep Domain Adaptation Network approach

Below we explain why our proposed Dual-GD-DDAN is able to resolve the two critical problems that occur with the DDAN approach. First, if $\boldsymbol{x}_i^S$ and $\boldsymbol{x}_j^S$ are proximal to each other and are located in the same cluster, then their representations $h(\boldsymbol{x}_i^S)$ and $h(\boldsymbol{x}_j^S)$ are close and hence, the weight μ_{ij} is large. This implies $G_S(\boldsymbol{x}_i^S)$ and $G_S(\boldsymbol{x}_j^S)$ are encouraged to be close in the joint space because we are minimising $\mu_{ij} ||G_S(\boldsymbol{x}_i^S) - G_S(\boldsymbol{x}_j^S)||^2$. This increases the chance of the two representations residing in the same cluster in the joint space. Therefore, Dual-GD-DDAN is able to preserve the clustering structure of the source data in the joint space. By using the same

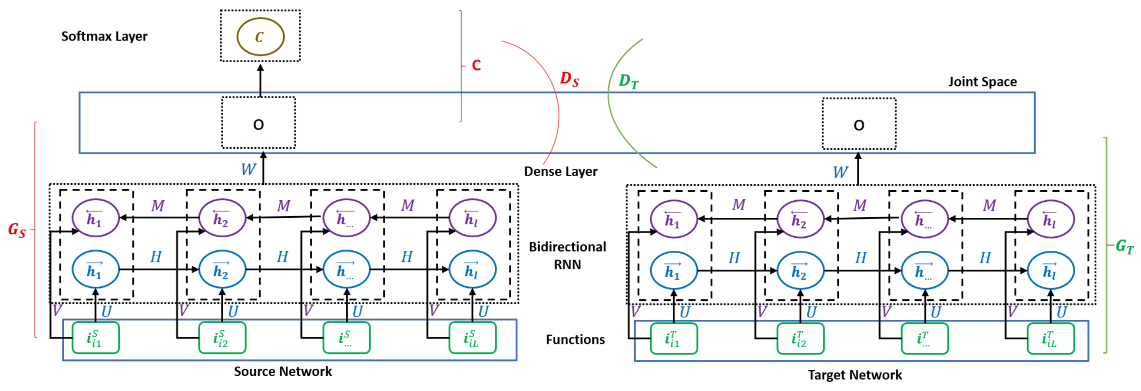


Figure 4.2: Architecture of our Dual-GD-DDAN method. The generators G_S and G_T take the sequential code tokens of the source domain and target domain in vectorial form respectively and map this sequence to the joint layer (i.e., the joint space). The vector representation of each statement $\boldsymbol{x}$ in source code is denoted by $\mathbf{i}$. The discriminators D_S and D_T are invoked to discriminate the source and target data. The source classifier $\mathcal{C}$ is trained on the source domain with labels. We note that the source and target networks do not share parameters and are not identical.

argument, we reach the same conclusion for the target domain.

Second, following Eqs. (4.1, 4.2), the discriminator D_S is trained to encourage large values for the source modes (i.e., $G_S(\boldsymbol{x}^S)$), while the discriminator D_T is trained to produce large values for the target modes (i.e., $G_T(\boldsymbol{x}^T)$). Moreover, as in Eq. (4.3), G_s is trained to move the source domain examples $\boldsymbol{x}^S$ to the high-valued region of D_T (i.e., the target modes or $G_T(\boldsymbol{x}^T)$) and G_T is trained to move the target examples $\boldsymbol{x}^T$ to the high-valued region of D_S (i.e., the source modes or $G_S(\boldsymbol{x}^S)$). As a consequence, eventually, the source modes (i.e., $G_S(\boldsymbol{x}^S)$) and target modes (i.e., $G_T(\boldsymbol{x}^T)$) overlap, while D_S and D_T place large values on both source (i.e., $G_S(\boldsymbol{x}^S)$) and target (i.e., $G_T(\boldsymbol{x}^T)$) modes. The mode missing problem is less likely to happen since, as shown in Fig. 4.1, if the target data misses source mode 2, then D_T cannot receive large values from source mode 2. Similarly, the boundary distortion problem is also less likely to happen since as in Fig. 4.1, if the target mode 1 can only partly cover the source mode 1, then D_T cannot receive large values from source mode 1. Therefore, Dual-GD-DDAN allows us to reduce the impact of the missing mode and boundary distortion problems, hence making the target distribution more identical to the source distribution in the joint space.

4.4.4 Dual Generator-Discriminator Semi-supervised Deep Domain Adaptation Network

When successfully bridging the gap between the source and target domains in the joint layer (i.e., the joint space), the target samples can be regarded as the unlabelled portion of a semi-supervised learning problem. Based on this observation, Nguyen et al. [2019] proposed to enforce the clustering assumption [Chapelle and Zien, 2005] by minimising the conditional entropy and using manifold regularisation with the spectral graph.

Using our proposed Dual-GD-DDAN, the conditional entropy $\mathcal{H}\left(\mathcal{C}, G_S, G_T\right)$ is defined as

$$\begin{aligned}\mathcal{H}\left(\mathcal{C}, G_S, G_T\right) = {} & \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_S}\left[-\mathcal{C}\left(G_S\left(\boldsymbol{x}\right)\right)\log\left[C\left(G_S\left(\boldsymbol{x}\right)\right)\right]\right] \\ & + \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_S}\left[-\left[1-\mathcal{C}\left(G_S\left(\boldsymbol{x}\right)\right)\right]\log\left[1-\mathcal{C}\left(G_S\left(\boldsymbol{x}\right)\right)\right]\right] \\ & + \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_T}\left[-\mathcal{C}\left(G_T\left(\boldsymbol{x}\right)\right)\log\left[\mathcal{C}\left(G_T\left(\boldsymbol{x}\right)\right)\right]\right] \\ & + \mathbb{E}_{\boldsymbol{x}\sim\mathbb{P}_T}\left[-\left[1-\mathcal{C}\left(G_T\left(\boldsymbol{x}\right)\right)\right]\log\left[1-\mathcal{C}\left(G_T\left(\boldsymbol{x}\right)\right)\right]\right]\end{aligned}$$

Let $SG = \left(\mathcal{V}, \mathcal{E}\right)$ where the set of vertices $\mathcal{V} = S \cup T$ be the spectral graph defined as in [Nguyen et al., 2019]. The manifold regularisation term is defined as

$$\begin{aligned}\mathcal{S}\left(\mathcal{C}, G_S, G_T\right) &= \sum_{(\boldsymbol{u},\boldsymbol{v})\in\mathcal{E}} \mu_{\boldsymbol{uv}} D_{KL}\left(B_{\boldsymbol{u}}, B_{\boldsymbol{v}}\right) \\ &= \sum_{(\boldsymbol{u},\boldsymbol{v})\in\mathcal{E}} \mu_{\boldsymbol{uv}}\left[C\left(\boldsymbol{u}\right)\log\frac{C\left(\boldsymbol{u}\right)}{C\left(\boldsymbol{v}\right)} + \left(1-C\left(\boldsymbol{u}\right)\right)\log\frac{1-C\left(\boldsymbol{u}\right)}{1-C\left(\boldsymbol{v}\right)}\right]\end{aligned}$$

where $B_{\boldsymbol{u}}$ specifies the Bernoulli distribution with $\mathbb{P}\left(y=1 \mid \boldsymbol{u}\right) = \mathcal{C}\left(\boldsymbol{u}\right)$ and $\mathbb{P}\left(y=-1 \mid \boldsymbol{u}\right) = 1-\mathcal{C}\left(\boldsymbol{u}\right)$. The weight $\mu_{\boldsymbol{uv}}$ is equal to $\exp\{-\left\|\boldsymbol{u}-\boldsymbol{v}\right\|^2/\left(2\sigma^2\right)\}$, and $D_{KL}\left(B_{\boldsymbol{u}}, B_{\boldsymbol{v}}\right)$ specifies the Kullback-Leibler divergence between two distributions. Here we note that $\boldsymbol{u} = G_S\left(\boldsymbol{x}^S\right)$ and $\boldsymbol{v} = G_T\left(\boldsymbol{x}^T\right)$ are two representations of the source sample $\boldsymbol{x}^S$ and the target sample $\boldsymbol{x}^T$ in the joint space respectively.

To be able to guarantee the clustering assumption for utilising the unlabelled target samples in a semi-supervised context, we leverage the minimisation of conditional entropy and manifold regularisation via spectral graph with our Dual Generator-Discriminator mechanism to propose Dual Generator-Discriminator Semi-supervised Deep Code Domain Adaptation Network (Dual-GD-SDDAN). The new formulas for updating the domain classifier $\mathcal{C}$ and two generators G_S and G_T in Dual-GD-SDDAN are as follows.

Updating the source classifier. The source classifier $\mathcal{C}$ in our proposed Dual-GD-SDDAN is updated by solving

$$\min_{\mathcal{C}}\left(\frac{1}{N_S}\sum_{i=1}^{N_S}\ell\left(\mathcal{C}\left(G_S\left(\boldsymbol{x}_i^S\right)\right), y_i\right) + \gamma\mathcal{H}\left(\mathcal{C}, G_S, G_T\right) + \lambda\mathcal{S}\left(\mathcal{C}, G_S, G_T\right)\right)$$

where γ and λ are two non-negative parameters.

Updating the generators. The generators G_S and G_T in our proposed Dual-GD-SDDAN are updated by minimising the following objective function

$$\frac{1}{N_S}\sum_{i=1}^{N_S}\ell\left(\mathcal{C}\left(G_S\left(\boldsymbol{x}_i^S\right)\right),y_i\right)+\alpha\mathcal{M}\left(G_S,G_T\right)$$
$$+\beta\mathcal{K}\left(G_S,G_T\right)+\gamma\mathcal{H}\left(\mathcal{C},G_S,G_T\right)+\lambda\mathcal{S}\left(\mathcal{C},G_S,G_T\right)$$

4.5 Implementation and results

In this section, firstly, we compare our proposed Dual-GD-DDAN with VulDeePecker without domain adaptation, MMD, D2GAN, DIRT-T and DDAN using the architecture CDAN proposed in [Nguyen et al., 2019]. Secondly, we do Boundary Distortion Analysis to further demonstrate the efficiency of our proposed Dual-GD-DDAN in alleviating the boundary distortion problem caused by using the GAN principle. Finally, we compare our Dual-GD-SDDAN and SCDAN introduced in [Nguyen et al., 2019].

4.5.1 Experimental setup

4.5.1.1 Experimental datasets

We use the real-world datasets collected by [Lin et al., 2018], which contain the source code of vulnerable and non-vulnerable functions obtained from five real-world software projects, namely, FFmpeg (#vul-funcs: 187, #non-vul-funcs: 5,427), LibTIFF (#vul-funcs: 81, #non-vul-funcs: 695), LibPNG (#vul-funcs: 43, #non-vul-funcs: 551), VLC (#vul-funcs: 25, #non-vul-funcs: 5,548) and Pidgin (#vul-funcs: 42, #non-vul-funcs: 8,268) where #vul-funcs and #non-vul-funcs is the number of vulnerable and non-vulnerable functions respectively. The datasets contain both multimedia (i.e., FFmpeg, VLC and Pidgin) and image (i.e., LibPNG and LibTIFF) application categories. In our experiment, datasets from the multimedia category were used as the source domain whilst datasets from the image category were used as the target domain (see Table 4.1).

4.5.1.2 Data processing and embedding

We preprocess datasets before inputting into the deep neural networks (i.e., baselines and our proposed method). Firstly, we standardise the source code by removing comments, blank lines and non-ASCII characters. Secondly, we map user-defined variables to symbolic names (e.g., "var1", "var2") and user-defined functions to symbolic names (e.g., "func1", "func2"). We also replace integers, real and hexadecimal numbers with a generic <num> token and strings with a generic <str> token. Thirdly, we embed statements in source code into vectors. In particular, each statement $\boldsymbol{x}$ consists of two parts: the opcode and the statement information. We embed both opcode and statement information to vectors, then concatenate the vector representations of opcode and statement information to obtain the final vector representation $\mathbf{i}$ of statement $\boldsymbol{x}$. For example, in the following statement (C programming language) "if(func3(func4(num,num),&var2)!=var11)", the opcode is *if*, and the statement information is (func3(func4(num,num),&var2)!=var11). To embed the opcode, we multiply the one-hot vector of the opcode by the opcode embedding matrix. To embed the statement information, we tokenise it to a sequence of tokens (e.g., (,func3,(,func4,(,num,num,),&,var2,),!=,var11,)), construct the frequency vector of the statement information, and multiply this frequency vector by the statement information embedding matrix. In addition, the opcode embedding and statement embedding matrices are learnable variables.

4.5.1.3 Model configuration

For training the eight methods – VulDeePecker, MMD, D2GAN, DIRT-T, DDAN, Dual-GD-DDAN, SCDAN and Dual-GD-SDDAN – we use one-layer bidirectional recurrent neural networks with LSTM cells where the size of hidden states is in $\{128, 256\}$ for the generators. For the source classifier and discriminators, we use deep feedforward neural networks with two hidden layers in which the size of each hidden layer is in $\{200, 300\}$. We embed the opcode and statement information in the $\{150, 150\}$ dimensional embedding spaces respectively. We employ the Adam optimiser with an initial learning rate in $\{10^{-3}, 10^{-4}\}$. The mini-batch size is 64. The trade-off parameters α, β, γ and λ are in $\{10^{-1}, 10^{-2}, 10^{-3}\}$. θ is in $\{0, 1\}$, and $1/(2\sigma^2)$ is in $\{2^{-10}, 2^{-9}\}$.

We split the data of the source domain into two random partitions containing 80% for training and 20% for validation. We also split the data of the target domain into two random partitions. The first partition contains 80% for training the models of VulDeePecker, MMD, D2GAN, DIRT-T, DDAN, Dual-GD-DDAN, SCDAN and Dual-GD-SDDAN without using any label information while the second partition contains 20% for testing the models. We additionally apply gradient clipping regularisation [Pascanu et al., 2013] to prevent over-fitting in the training process of each model. We implement eight mentioned methods in Python using Tensorflow [Abadi et al., 2016] which is an open-source software library for Machine Intelligence developed by the Google Brain Team.

4.5.2 Experimental results

4.5.2.1 Code domain adaptation for a fully non-labelled target project

We investigate the performance of our proposed Dual-GD-DDAN compared with other methods including VulDeePecker (VULD) without domain adaptation [Li et al., 2018], DDAN [Nguyen et al., 2019], MMD [Long et al., 2015], D2GAN [Nguyen et al., 2017] and DIRT-T [Shu et al., 2018] with VAP applied in the joint feature layer using the architecture CDAN introduced in [Nguyen et al., 2019]. The VulDeePecker method is only trained on the source data and then tested on the target data, while the MMD, D2GAN, DIRT-T, DDAN and Dual-GD-DDAN methods employ the target data without using any label information for domain adaptation.

In Table 4.1, the experimental results show that our proposed Dual-GD-DDAN achieves a higher performance for detecting vulnerable and non-vulnerable functions for most performance measures, including FNR, FPR, Recall, Precision and F1-measure in almost cases of the source and target domains, especially for F1-measure. Particularly, our Dual-GD-DDAN always obtains the highest F1-measure in all cases. For example, for the case of the source domain (FFmpeg) and target domain (LibPNG), Dual-GD-DDAN achieves an F1-measure of 88.89% compared with an F1-measure of 84.21%, 84.21%, 80%, 77.78% and 75% obtained with DDAN, DIRT-T, D2GAN, MMD and VulDeePecker respectively.

4.5.2.2 Boundary distortion analysis

Quantitative results. To quantitatively demonstrate the efficiency of our proposed Dual-GD-DDAN in alleviating the boundary distortion problem caused by using the GAN principle, we reuse the experimental setting in Section 5.2 [Santurkar et al., 2018]. The basic idea is, given two datasets S_1 and S_2, to quantify the degree of cover of these two datasets. We train a classifier $\mathcal{C}_1$ on S_1, then test on S_2 and another classifier $\mathcal{C}_2$ on S_2, then test on S_1. If these two datasets cover each other well with reduced boundary distortion, we expect that if $\mathcal{C}_1$ predicts well on S_1, then it should predict well on S_2 and vice versa if $\mathcal{C}_2$ predicts well on S_2, then it should predict

Source → Target	Methods	FNR	FPR	Recall	Precision	F1-measure
Pidgin → LibPNG	VULD	42.86%	1.08%	57.14%	80%	66.67%
	MMD	37.50%	**0%**	62.50%	**100%**	76.92%
	D2GAN	**33.33%**	1.06%	**66.67%**	80%	72.73%
	DIRT-T	**33.33%**	1.06%	**66.67%**	80%	72.73%
	DDAN	37.50%	**0%**	62.50%	**100%**	76.92%
	Dual-GD-DDAN	**33.33%**	**0%**	**66.67%**	**100%**	**80%**
FFmpeg → LibTIFF	VULD	43.75%	**6.72%**	56.25%	50%	52.94%
	MMD	28.57%	12.79%	71.43%	47.62%	57.14%
	D2GAN	30.77%	6.97%	69.23%	**64.29%**	66.67%
	DIRT-T	25%	9.09%	75%	52.94%	62.07%
	DDAN	35.71%	6.98%	64.29%	60%	62.07%
	Dual-GD-DDAN	**12.5%**	8.2%	**87.5%**	56%	**68.29%**
FFmpeg → LibPNG	VULD	25%	**2.17%**	75%	75%	75%
	MMD	12.5%	3.26%	87.5%	70%	77.78%
	D2GAN	14.29%	**2.17%**	85.71%	75%	80%
	DIRT-T	15.11%	2.2%	84.89%	**80%**	84.21%
	DDAN	**0%**	3.26%	**100%**	72.73%	84.21%
	Dual-GD-DDAN	**0%**	**2.17%**	**100%**	**80%**	**88.89%**
VLC→ LibPNG	VULD	57.14%	**1.08%**	42.86%	75%	54.55%
	MMD	45%	4.35%	55%	60%	66.67%
	D2GAN	28.57%	4.3%	71.43%	55.56%	62.5%
	DIRT-T	50%	1.09%	50%	**80%**	61.54%
	DDAN	33.33%	2.20%	66.67%	75%	70.59%
	Dual-GD-DDAN	**28.57%**	2.15%	**71.43%**	71.43%	**71.43%**
Pidgin → LibTIFF	VULD	35.29%	8.27%	64.71%	50%	56.41%
	MMD	30.18%	12.35%	69.82%	50%	58.27%
	D2GAN	40%	7.95%	60%	**60%**	60%
	DIRT-T	38.46%	8.05%	61.54%	53.33%	57.14%
	DDAN	**27.27%**	8.99%	**72.73%**	50%	59.26%
	Dual-GD-DDAN	29.41%	**6.76%**	70.59%	57.14%	**63.16%**

Table 4.1: Performance results in terms of false negative rate (FNR), false positive rate (FPR), Recall, Precision and F1-measure of VulDeePecker (VULD), MMD, D2GAN, DIRT-T, DDAN and Dual-GD-DDAN for predicting vulnerable and non-vulnerable code functions on the testing set of the target domain (best performance in **bold**).

well on S_1. This would seem reasonable since if boundary distortion occurs (i.e., assume that S_2 partly covers S_1), then $\mathcal{C}_2$ trained on S_2 would struggle to predict S_1 well which is much larger and possibly more complex. Therefore, we can utilise the magnitude of the accuracies and the accuracy gap of $\mathcal{C}_1$ and $\mathcal{C}_2$ when predicting their training and testing sets to assess the severity of the boundary distortion problem.

Source → Target	Methods	Accuracy			Accuracy		
		Tr-src/	Ts-tar/	Acc-gap	Tr-tar/	Ts-src/	Acc-gap
Pidgin → LibPNG	DDAN	98.8%	96%	2.8%	97%	92%	5%
	Dual-GD-DDAN	99%	97%	**2%**	97%	95%	**2%**
FFmpeg → LibPNG	Methods	Accuracy			Accuracy		
		Tr-src/	Ts-tar/	Acc-gap	Tr-tar/	Te-src/	Acc-gap
	DDAN	95.9%	92%	3.9%	91%	83.3%	7.7%
	Dual-GD-DDAN	97%	96%	**1%**	98%	95.6%	**2.4%**

Table 4.2: Accuracies obtained by the DDAN and Dual-GD-DDAN methods when predicting vulnerable and non-vulnerable code functions on the source and target domains. Note that Tr-src, Ts-tar, Tr-tar, Ts-src, and Acc-gap are the shorthands of the train source, test target, train target, test source, and accuracy gap respectively. For the accuracy gap, a smaller value is better.

Inspired by this observation, we compare our Dual-GD-DDAN with DDAN using the representations of the source and target samples in the joint feature space corresponding to their best models. In particular, for a given pair of source and target datasets and for comparing each method, we train a neural network classifier on the best representations of the source dataset in the joint space, then predict on the source and target dataset and do the same but swap the role of the source and target datasets. We then measure the difference of the corresponding accuracies as a means of measuring the severity of the boundary distortion. We choose to conduct such a boundary distortion analysis for two pairs of the source (FFmpeg and Pidgin) and target (LibPNG) domains. As shown in Table 4.2, all gaps obtained by our Dual-GD-DDAN are always smaller than those obtained by DDAN, while the accuracies obtained by our proposed method are always larger. We can therefore conclude that our Dual-GD-DDAN method produces a better representation for source and target samples in the joint space and is less susceptible to boundary distortion compared with the DDAN method.

Visualisation. We further demonstrate the efficiency of our proposed Dual-GD-DDAN in alleviating the boundary distortion problem caused by using the GAN principle. Using a t-SNE [Maaten and Hinton, 2008] projection, with perplexity equal to 30, we visualise the feature distributions of the source and target domains in the joint space. Specifically, we project the source and target data in the joint space (i.e., $G(\boldsymbol{x})$) into a 2D space with domain adaptation (DDAN) and with dual-domain adaptation (Dual-GD-DDAN). In Fig. 4.3, we observe these cases when performing domain adaptation from a software project (FFmpeg) to another (LibPNG). As shown in Fig. 4.3, with undertaking domain adaptation (DDAN, the left figure) and dual-domain adaptation (Dual-GD-DDAN, the right figure), the source and target data sampled are intermingled especially for Dual-GD-DDAN. However, it can be observed that DDAN when solely applying the GAN is seriously vulnerable to the boundary distortion issue. In particular, in the clusters/data modes 2, 3 and 4 (the left figure), the boundary distortion issue occurs since the blue data only partly cover the corresponding red ones (i.e., the source and target data do not totally mix up). Meanwhile, for our Dual-GD-DDAN, the boundary distortion issue is much less vulnerable, and the mixing-up level of source and target data is significantly higher in each cluster/data mode.

4.5.2.3 Quantitative results of Dual Generator-Discriminator Semi-supervised Deep Code Domain Adaptation

In this section, we compare the performance of our Dual-GD-SDDAN with Semi-supervised Deep Code Domain Adaptation (SCDAN) [Nguyen et al., 2019] on four pairs of the source and

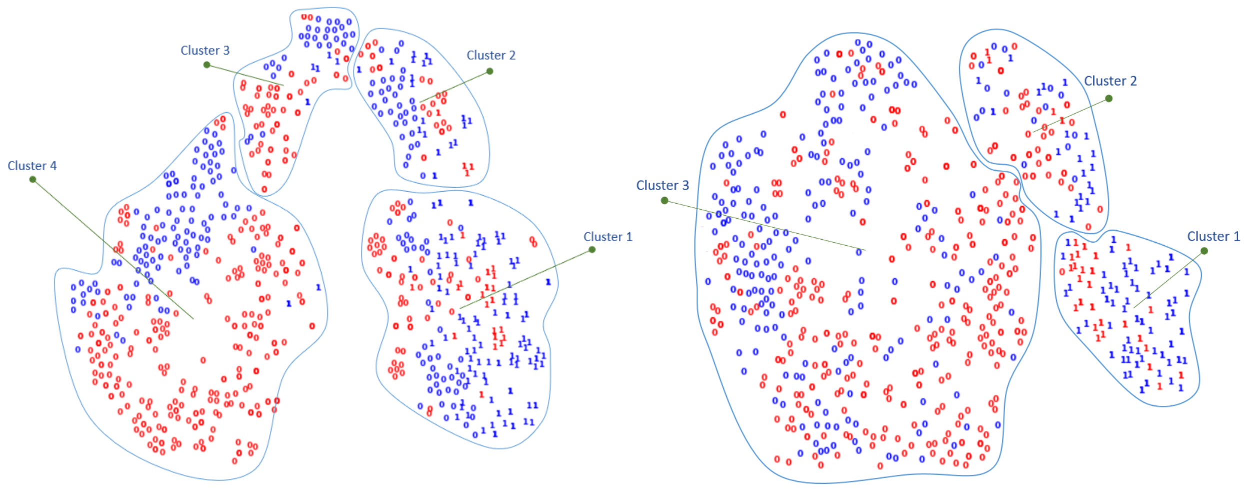


Figure 4.3: 2D t-SNE projection for the case of the FFmpeg → LibPNG domain adaptation. The blue and red points represent the source and target domains in the joint space respectively. In both cases of the source and target domains, data points labelled 0 stand for non-vulnerable samples and data points labelled 1 stand for vulnerable samples.

target domains. In Table 4.3, the experimental results show that our Dual-GD-SDDAN achieves a higher performance than SCDAN for detecting vulnerable and non-vulnerable functions in terms of FPR, Precision and F1-measure in almost cases of the source and target domains, especially for F1-measure. For example, to the case of the source domain (VLC) and target domain (LibPNG), our Dual-GD-SDDAN achieves an F1-measure of 76.19% compared with an F1-measure of 72.73% obtained with SCDAN. These results further demonstrate the ability of our Dual-GD-SDDAN for dealing with the mode collapsing problem better than SCDAN [Nguyen et al., 2019], hence obtaining better predictive performance in the context of software domain adaptation.

Source → Target	Methods	FPR	FNR	Recall	Precision	F1-measure
FFmpeg → LibTIFF	SCDAN	5.38%	**14.29%**	**85.71%**	57.14%	68.57%
	Dual-GD-SDDAN	**3.01%**	35.29%	64.71%	**73.33%**	**68.75%**
FFmpeg → LibPNG	SCDAN	1.08%	**12.5%**	**87.5%**	87.5%	87.5%
	Dual-GD-SDDAN	**0%**	17.5%	82.5%	**100%**	**90.41%**
VLC→ LibPNG	SCDAN	**1.06%**	33.33%	66.67%	**80%**	72.73%
	Dual-GD-SDDAN	4.39%	**11.11%**	**88.89%**	66.67%	**76.19%**
Pidgin → LibTIFF	SCDAN	5.56%	**30%**	**70%**	58.33%	63.64%
	Dual-GD-SDDAN	**2.98%**	37.5%	62.5%	**71.43%**	**66.67%**

Table 4.3: Performance results in terms of false negative rate (FNR), false positive rate (FPR), Recall, Precision and F1-measure of SCDAN and Dual-GD-SDDAN methods for predicting vulnerable/non-vulnerable code functions on the testing set of the target domain (best performance in **bold**).

4.6 Closing remarks

The software vulnerability detection problem is an important problem in the software industry and in the field of computer security. One of the most crucial issues in SVD is to cope with the scarcity of labelled vulnerabilities in projects that require the laborious labelling of code by software security experts. In this chapter, we propose the Dual Generator-Discriminator Deep Code Domain Adaptation Network (Dual-GD-DDAN) method to deal with the missing mode and boundary distortion problems which arise from the use of the GAN principle when reducing the discrepancy between source and target data in the joint space. We conducted experiments to compare our Dual-GD-DDAN method with the state-of-the-art baselines. The experimental results show that our proposed method outperforms these rival baselines by a wide margin in term of predictive performances.

Part II

Learning to Explain Software Vulnerability (Fine-grain-level Vulnerability Detection)

Preface to Part II

In the previous part (i.e., Part I), we have presented our proposed methods for dealing with one of the most crucial issues in software vulnerability detection (SVD), coping with the scarcity of labelled vulnerabilities in projects that require the laborious manual labelling of code by software security experts. Our proposed methods are the first work to formulate the transferred learning of software vulnerabilities from a labelled source of sequences to an unlabelled target of sequences. We believe our proposed novel CDAN architecture is a new building block for a wide array of other applications in other domains such as behaviour modelling in financial technology (fintech) where temporal dynamics are important and sequence modelling in computational biology.

In Part II, Chapters 5 and 6, we summarise our proposed methods to deal with the second problem existing in current SVD methods, which is relevant to the second research question (Q.2): "how to efficiently exploit the semantic and syntactic relationships inside source code to *detect vulnerabilities at a fine-grained level with more flexible scope* (i.e., the statement level) than the function or program levels".

Chapter 5

Information-theoretic Source Code Vulnerability Highlighting

Software vulnerabilities (SVs) are a crucial and serious concern in the software industry and in computer security. A variety of methods have been proposed to detect vulnerabilities in real-world software. Recent methods based on deep learning approaches for automatic feature extraction have improved software vulnerability identification compared with machine learning approaches based on hand-crafted feature extraction. However, these methods can usually only detect SVs at a function-[Lin et al., 2018, Li et al., 2018] or program-[Dam et al., 2017] level, which is much less informative because, out of hundreds (thousands) of code statements in a program or function, only a few core statements contribute to a software vulnerability. This requires us to find a way to detect software vulnerabilities at a fine-grained level. For most publicly available datasets, vulnerabilities are only labelled at the program or function levels, not at the statement level. In doing this, we can then significantly speed up the process of isolating and detecting SVs, thereby reducing the time and cost involved.

In this chapter, we propose a novel learn-to-explain model based on the concept of mutual information that can help us to detect and isolate SVs at a fine-grained level (i.e., statements that are highly relevant to a software vulnerability) in both unsupervised and semi-supervised learning settings. Our proposed method involves two key stages. In the first stage, we train a reference deep learning model with the aim of approximating the true conditional distribution $p(Y \mid F)$ (i.e., the true probabilistic labelling assignment mechanism) of label Y (i.e., vulnerable or non-vulnerable) with respect to the source code F by $p_m(Y \mid F)$ offered by the reference model. In the second stage, we learn another model that aims to explain the reference model by specifying the top-K statements in the given source code that mostly contribute to the vulnerability prediction decision for this model.

5.1 Motivations

In the field of software security, software vulnerabilities (SVs) are specific potential flaws, glitches, weaknesses or oversights in parts of software. Attackers or vandals can leverage these vulnerabilities to carry out malicious actions, such as exposing or altering sensitive information, disrupting or destroying a system, or taking control of a program or computer system [Dowd et al., 2006]. Owing to the rapid growth and dramatic diversity of software, a large amount of computer software potentially contains vulnerabilities, which can create severe threats to cybersecurity, resulting in expenditure costs of about USD 600 billion globally each year [McAfee and CSIS, 2017]. These threats call for an urgent need of advanced approaches (i.e., automatic tools and methods) to efficiently and effectively deal with the large amount of vulnerable code with a minimal level of human intervention.

In this chapter, we propose a novel method that allows us to find and highlight code statements in functions or programs that are truly relevant to the presence of significant code vulnerabilities. Given vulnerable source code (i.e., functions or programs), we aim to highlight the top-K statements that are the most relevant to the vulnerable and non-vulnerable class labels. By referring to the vulnerable source code with a high probability, the highlighted statements contain the core statements that contribute to the overall code vulnerabilities. Moreover, our proposed method specifies and highlights the statements that explain the vulnerability intrinsic to the given source code. Our proposed method involves two key stages. In the first stage, we train a reference deep learning model, with the aim of approximating the true conditional distribution $p(Y \mid F)$ (i.e., the true probabilistic labelling assignment mechanism) of label Y (where $Y = 1$ means a vulnerability and $Y = 0$ means otherwise) w.r.t the source code F by $p_m\left(Y \mid F\right)$ offered by the reference model. In the second stage, we learn another model that aims to explain the reference model by specifying the top-K statements in the given source code that mostly contribute to the vulnerability prediction decision for this model.

The idea is to select a subset of K statements that maximises the mutual information to the corresponding label given by the reference model. A similar information-theoretic metric has been employed in the L2X model [Chen et al., 2018] for instance-wise feature selection for the case of vectorial data. Although that model can be adopted to work for sequential data (e.g., source code), our proposed method is different from L2X in the following aspects: i) our formulation for mutual information takes into consideration the sequential nature of source code (in contrast to L2X), and ii) inspired from this formulation, we propose a novel architecture using a multi-Bernoulli distribution for selecting the top-K mostly relevant statements rather than employing a multinomial distribution as in L2X. The advantage of this approach is two-fold. First, this allows us to better control the random selection process. Second, it allows us to incorporate the information from ground truth in a semi-supervised context wherein we assume that a small portion of source code data might have annotations of core statements that cause a vulnerability. Our key contributions include:

- We propose a novel learn-to-explain model that is based on mutual information and takes into account the sequential nature of data to better evaluate mutual information. Using this theory, we propose a novel architecture based on multi-Bernoulli distribution for random subset of statements selection. Unlike the multinomial distribution used in L2X, our mechanism is more controllable and enables us to train the model in a semi-supervised context. In addition, our proposed model can be used to highlight the core statements that are a subset of the most relevant statements of vulnerable source code. It can also explain how the reference model works by identifying the most important statements that contribute to its prediction.
- We conduct experiments on the datasets collected by [Li et al., 2018], that contain source code of vulnerable and non-vulnerable functions from two real-world software data sources and compare our proposed method to a state-of-the-art baseline L2X approach on these two datasets. We further investigate our proposed method in the semi-supervised context by comparing it to itself in an unsupervised context. We demonstrate that our proposed method can detect vulnerable code statements in functions much more effectively than L2X in unsupervised context and its semi-supervised variant can significantly boost the performance.

5.2 Related work for fine-grain-level vulnerability detection

There are two typical approaches for software vulnerability detection (SVD) including methods based on either hand-crafted or automatic extraction of features. Most previous work in software vulnerability detection [Neuhaus et al., 2007, Shin et al., 2011, Yamaguchi et al., 2011, Almorsy et al., 2012, Li et al., 2016, Grieco et al., 2016, Kim et al., 2017] has been developed based on hand-crafted features of data which are manually chosen by knowledgeable domain experts and may thus carry outdated experience, expertise and underlying biases [Zimmermann et al., 2009].

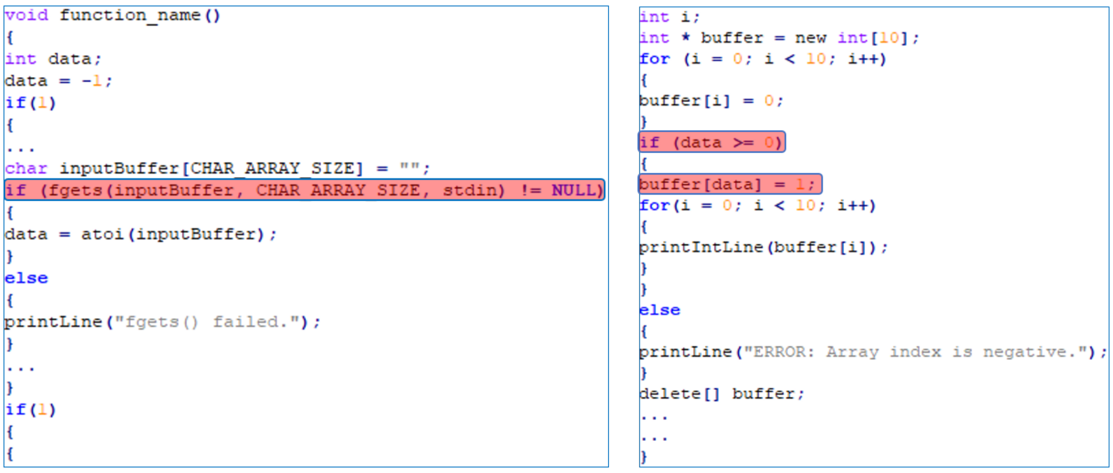
To lessen the dependency on hand-crafted features, the use of automatically learned features for SVD has been recently studied, notably [Dam et al., 2017, Li et al., 2018, Lin et al., 2018]. In particular, these works leverage deep learning models to automatically extract features, and have shown great advances over those based on hand-crafted features.

Despite showing promising performances, current deep learning-based methods are only able to detect software vulnerabilities at the function [Lin et al., 2018, Li et al., 2018] or program [Dam et al., 2017] levels. In real-world situations, programs or even functions are often very long and may consist of hundreds or thousands of lines of code. The source of most vulnerabilities arises from a significantly smaller scope, usually a few core statements. We thus want to be able to detect software vulnerabilities at a more fine-grained level, i.e., several code statements within functions or programs. This includes highlighting statements that are highly relevant to the corresponding vulnerability and associated code statements. In doing this, we can then significantly speed up the process of isolating and detecting software vulnerabilities, thereby reducing the time and cost involved.

To the best of our knowledge, there is one deep learning-based method, named VulDeeLocator [Li et al., 2020] (posted on ArXiv for fine-grain-level vulnerability detection). However, besides the original source codes F and their vulnerability labels Y, VulDeeLocator needs further information relevant to vulnerable code statements to obtain what is called intermediate code-based vulnerability candidate representation from its preprocessing steps, and then use the intermediate code-based representation in the training and testing processes rather than the original source code F. This is totally different from our proposed approach which can be run in an unsupervised setting (only requires the source codes F and their vulnerability labels Y, and not uses information relevant to vulnerable code statements) for the task of fine-grain-level vulnerability detection. VulDeeLocator not only cannot work in the unsupervised setting, but also cannot work directly with source code (i.e., it requires to compile source code to Lower Level Virtual Machine (LLVM) intermediate code. If source code cannot be compiled to LLVM intermediate code, we cannot use VulDeeLocator).

5.3 Motivating example

We give an example of a source code function obtained from the CWE-119 dataset to demonstrate software vulnerability detection (SVD) at a fine-grained level, shown in Fig. 1. This function has a few core vulnerable code statements highlighted in red that are the main source of the vulnerability. The statement "if (fgets (inputBuffer, CHAR_ARRAY_SIZE, stdin) !=



```
void function_name()
{
int data;
data = -1;
if(1)
{
...
char inputBuffer[CHAR_ARRAY_SIZE] = "";
if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL)
{
data = atoi(inputBuffer);
}
else
{
printLine("fgets() failed.");
}
...
}
if(1)
{
{
```

```
int i;
int * buffer = new int[10];
for (i = 0; i < 10; i++)
{
buffer[i] = 0;
}
if (data >= 0)
{
buffer[data] = 1;
for(i = 0; i < 10; i++)
{
printIntLine(buffer[i]);
}
}
else
{
printLine("ERROR: Array index is negative.");
}
delete[] buffer;
...
...
}
```

Figure 5.1: Example of a source code function obtained from the CWE-119 dataset. The left-hand and right-hand figures are the first and second parts of the function. For demonstration purpose and simplicity, we choose a simple source code function. There are some parts of the function omitted for the brevity.

NULL)" is a potential vulnerability because we read data from the console using fgets(). Likewise the two statements "if (data >= 0)" and "buffer[data] = 1;" cause another potential vulnerability because we attempt to write to an index of the array that exceeds the upper bound. Since the real-world source codes might contain hundreds of statements, we want to be able to highlight several statements in the function that are highly relevant to the presence of a vulnerability and contain the core vulnerable statements. In doing so, we can significantly speed up the process of isolating and detecting software vulnerabilities, and therefore reduce the cognitive load of the security analyst.

5.4 Information-theoretic Code Vulnerability Highlighting

We begin with the problem statement of Information-theoretic Code Vulnerability Highlighting (ICVH), followed by the technical details of ICVH in the unsupervised and semi-supervised contexts.

5.4.1 The problem statement

Consider a dataset $\mathbf{D} = \{(F_1, Y_1), \ldots, (F_N, Y_N)\}$ where $Y_i \in \{0, 1\}$ (where *1*: vulnerable code and *0*: non-vulnerable code) and $F_i = \left[\boldsymbol{f}_1^i, \ldots, \boldsymbol{f}_{N_i}^i\right]$ is source code with a sequence of N_i statements (i.e., F_i is the i-th source code section in the dataset D) while the lower-case f stands for a statement in the corresponding source code F (e.g., $\boldsymbol{f}_k^i$ is the k-th code statement in the source code F_i). Given a source code $F = [\boldsymbol{f}_1, \ldots, \boldsymbol{f}_L]$, we denote the subset $F_S =$

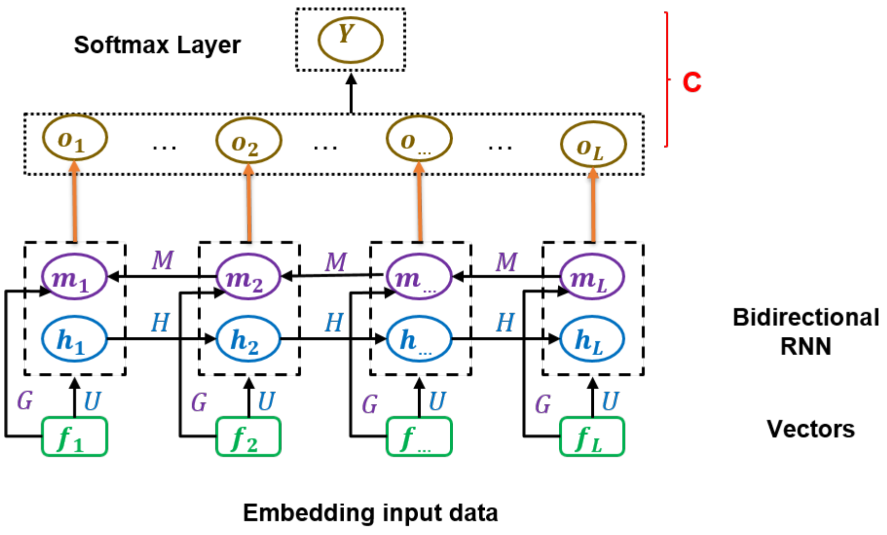


Figure 5.2: Architecture of the reference model using a bi-RNN.

$[\boldsymbol{f}_{i_1}, \ldots, \boldsymbol{f}_{i_K}] = [\boldsymbol{f}_j]_{j \in S}$ where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$ $(i_1 < i_2 < ... < i_K)$. Most existing work in software vulnerability detection involves detecting vulnerabilities at the program or function level. However, in source code only several core statements are highly relevant to a given vulnerability. In this work, we undertake vulnerability detection at a fine-grained level than at the function or program level. In other words, we learn to emphasise the code blocks that are directly and highly relevant to the vulnerabilities. Specifically, given a function F, our task is to select a subset F_S where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$ in such a way that F_S is highly relevant to the presence of a vulnerability.

5.4.2 The reference model

The reference model aims to learn a model distribution $p_m(Y \mid F)$ that can approximate the true distribution $p(Y \mid F)$ where Y is the label of the corresponding source code F and $Y, F \sim p(Y, F) = p(F)p(Y \mid F)$. To obtain $p_m(Y \mid F)$, we use a network architecture with a combination of a bidirectional recurrent neural network (bi-RNN) to learn vector representations of source code and a deep feedforward neural network, which takes the outputs of the bi-RNN as inputs, to model the distribution $p_m(Y \mid F)$. The architecture of the reference model is depicted in Fig. 5.2 where $\boldsymbol{m}_i$, $\boldsymbol{h}_i$ and $\boldsymbol{o}_i = \text{concat}(\boldsymbol{m}_i, \boldsymbol{h}_i)$ with $i = 1, .., L$ are the hidden states and the output of the bi-RNN respectively, while C is the prediction layer, and M, H, U and G are model parameters. We note that without loss of generalisation and to simplify the notion, we use $\boldsymbol{f}_i$ to represent both a symbolic code statement and its embedding vector that is fed to the networks. Data preprocessing and embedding is discussed in the Data Processing and Embedding section in the Experiment section.

5.4.3 The explaining model: Information-theoretic Code Vulnerability Highlighting

The proposed explaining model aims to explain the reference model by specifying the top-K statements (i.e., the top-K vulnerability-relevant statements) in the given source code that mostly contribute to the vulnerability prediction decision of the reference model. Our proposed model does not use any information about ground truth of vulnerable code statements in the training. We name this setting as unsupervised context.

5.4.3.1 Theoretical formulation with mutual information to capture the sequential nature of data

To select the most relevant subset F_S, we aim to maximise the mutual information: $\mathbb{I}\left(F_S, Y\right)$ where we view Y obtained from the reference model as a random variable that is characterised using $p_m\left(Y \mid F\right)$, which is previously trained using the whole training set $\mathbf{D}$. Mathematically, we aim to solve the following optimisation problem:

$$\max \mathbb{E}_{p(F)}\left[\mathbb{E}_{p(S|F)}\left[\mathbb{I}\left(F_S, Y\right)\right]\right] \tag{5.1}$$

Eq. (5.1) means that given source code $F \sim p(F)$ (data distribution), we need to devise the random selection process characterised by $p(S \mid F)$ to select the subset F_S such that the mutual information of F_S and the label Y is maximised. The two main problems here are: i) how to design the random selection process $p(S \mid F)$, and ii) how to obtain $\mathbb{I}(F_S, Y)$. We develop the following relevant theory to efficiently derive $\mathbb{I}(F_S, Y)$ and solve Eq. (5.1). We have,

$$\mathbb{I}\left(F_S, Y\right) = \sum_{k=1}^{K} \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}\left(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}}\right)\right]$$

The following lemma tackles $\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}[\mathbb{I}(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}})]$ and this term can be further derived as follows. We have

$$\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}\left(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}}\right)\right] \approx \mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[\log p_m\left(Y \mid \boldsymbol{f}_{i_{1:k}}\right)\right]\right] + \text{const}$$

The next lemma gives a lower bound to $\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}[\log p_m(Y \mid \boldsymbol{f}_{i_{1:k}})]]$. We can obtain a lower bound to $\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}[\mathbb{E}_{p_m(Y|\boldsymbol{f}_{i_{1:k}})}[\log p_m(Y \mid \boldsymbol{f}_{i_{1:k}})]$ *by*

$$\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[\log Q\left(Y \mid \boldsymbol{f}_{i_{1:k}}\right)\right]\right]$$

for every $Q\left(Y \mid F\right)$. The proofs of the above lemmas can be found in the Appendix. Combining Lemmas 1, 2 and 3, the optimisation problem in Eq. (5.1) can be now rewritten:

$$\max \mathbb{E}_{p(F)}\left[\mathbb{E}_{p(S|F)}\left[\sum_{k=1}^{K}\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[\log Q(Y \mid \boldsymbol{f}_{i_{1:k}})\right]\right]\right]\right] \tag{5.2}$$

5.4.3.2 Network design in the unsupervised context

To design the random selection process for selecting $S = \{i_1, ..., i_K\}$ $(i_1 < i_2 < ... < i_K)$ to form $p(S \mid F)$ and formulate $Q(Y \mid \boldsymbol{f}_{i_{1:k}})$ for $k = 1, \ldots, K$, we employ a bi-RNN with sequence length L. As shown in Fig. 5.4, for each statement $\boldsymbol{f}_k$ or its embedding vector, which is also denoted by $\boldsymbol{f}_k$, we use a Bernoulli random variable Z_k with $\mathbb{P}(Z_k = 1) = \mu_k$ (i.e., we apply the sigmoid activation over μ_k to convert it to a probability) to specify if $\boldsymbol{f}_k$ is selected or not (i.e., $Z_k = 1$ means $\boldsymbol{f}_k$ is selected). We compute $Z_k\boldsymbol{f}_k$ for all k and then feed them to another bi-RNN where we make a prediction of the label at the hidden states with $Z_k = 1$ to mimic $Q(Y \mid \boldsymbol{f}_{i_{1:k}})$. In addition, we have approximated the sub-sequence of statements $[\boldsymbol{f}_{i_1}, ..., \boldsymbol{f}_{i_k}]$ by the sequence $[Z_1\boldsymbol{f}_1, ..., Z_L\boldsymbol{f}_L]$ in which the inactive (not selected) statements are substituted by the vector $\mathbf{0}$.

To render the above random selection process continuous and differentiable for training, we employ the Concrete (Gumbel-softmax) distribution [Jang et al., 2016, Maddison et al., 2016] to undertake relaxation on the Bernoulli random variable Z_k. In particular, we sample V_k from the Concrete distribution as: $[V_k, 1 - V_k] \sim \text{Concrete}(\mu_k, 1 - \mu_k)$.

We now denote $V \odot F$ as $[V_k\boldsymbol{f}_k]_{k=1,...,L}$ and the sequence $V \odot F$ is injected into the second bi-RNN. We denote S_K as the set of indices with the top K values for V_k. The optimisation problem in Eq. (5.2) can be rewritten as follows:

$$\max \mathbb{E}_{p(F)}\left[\mathbb{E}_{p(S|F)}\left[\sum_{k\in S_K}\left[\sum_{y=0}^{1}p_m\left(Y = y \mid V \odot F\right)\log p(Y = y \mid \boldsymbol{h}_k^2, \boldsymbol{m}_k^2)\right]\right]\right] \tag{5.3}$$

where $\log p(Y = y \mid \boldsymbol{h}_k^2, \boldsymbol{m}_k^2)$ relates to the log-likelihood of the second bi-RNN. The architecture of our proposed Information-theoretic Code Vulnerability Highlighting applied to code vulnerability identification is depicted in Fig. 5.4 where $\boldsymbol{m}_i^1$, $\boldsymbol{h}_i^1$, $\boldsymbol{m}_i^2$ and $\boldsymbol{h}_i^2$ with $i = 1, .., L$ are the hidden states of two bi-RNNs while M_j, H_j, U_j, G_j with $j = 1, 2$ and W are model parameters.

Our ICVH model can be applied to both source code and binary code vulnerability identification.

In the testing phase, we choose the code statements whose indices lie in S_K with top K values for μ_k. We can interpret μ_k as the probability to select the k-th code statement in our subset.

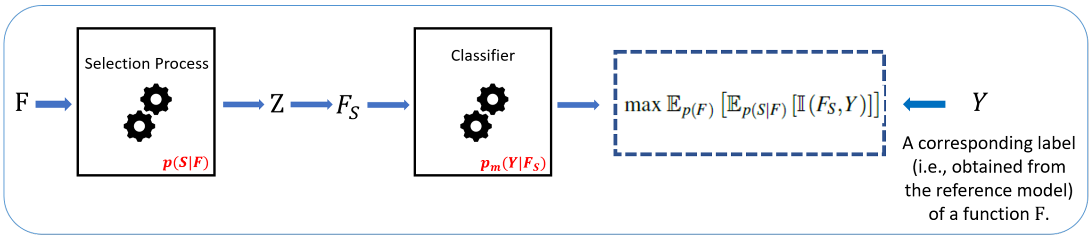

F
Selection Process
$p(S|F)$
Z
F_S
Classifier
$p_m(Y|F_S)$
$\max \mathbb{E}_{p(F)}\left[\mathbb{E}_{p(S|F)}\left[\mathbb{I}(F_S, Y)\right]\right]$
Y
A corresponding label (i.e., obtained from the reference model) of a function F.

Figure 5.3: Training phase of the ICVH model.

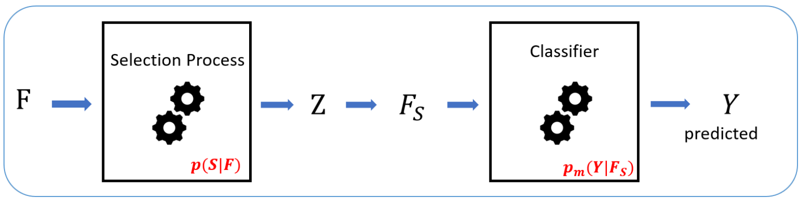


Figure 5.5: Testing phase of our ICVH model using the trained model obtained from the training phase.

For our explaining ICVH model, we aim to obtain high performances not only on approaching

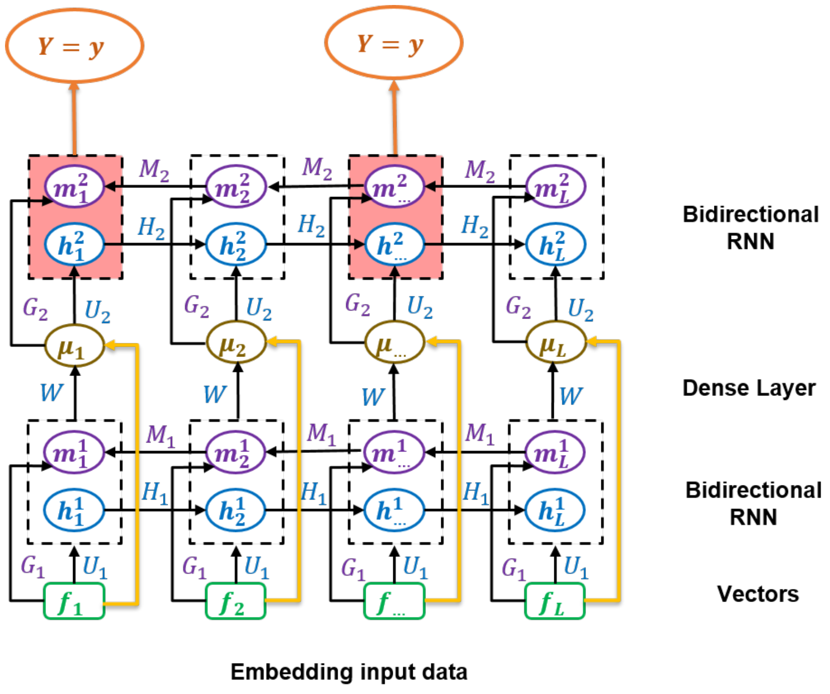


Figure 5.4: Architecture of our Information-theoretic Code Vulnerability Highlighting (ICVH) network using bi-RNNs. The pink cells refer to those code statements in S_K, and we formulate $p(Y \mid \boldsymbol{h}_k^2, \boldsymbol{m}_k^2)$ using the softmax function.

close to (or have a good explanation of) the reference model (i.e., a high F1-score for the label prediction Y obtained from the reference model), but also on the selecting and highlighting process of vulnerable code statements in vulnerable functions (i.e., the VCA and VCP measures which are both mentioned in the experiment section). The working processes of our ICVH model in the training (i.e., in the unsupervised context) and testing phases are visualised in Fig. 5.3 and Fig. 5.5 respectively.

5.4.3.3 Network design in the semi-supervised context

It is very convenient to incorporate the annotations of the core vulnerable statements in the ground truth to our network design. Specifically, let $F_c = [\boldsymbol{f}_{i_1}, ..., \boldsymbol{f}_{i_m}]$ be the core vulnerable statements for a given source code. We maximise the probabilities of selecting the core statements and not selecting other statements in the first bi-RNN as follows:

$$\max \left(\sum_{k \in I_c} \log \mu_k + \sum_{k \notin I_c} \log(1 - \mu_k) \right),$$

where $I_c = [i_1, ..., i_m]$. We then add the above objective function to the main objective function in Eq. (5.3) with the trade-off parameter $\lambda > 0$.

5.5 Implementation and results

First, we compare our Information-theoretic Code Vulnerability Highlighting (ICVH) method with L2X introduced in [Chen et al., 2018] and the random selection method (RSM) where we randomly choose statements from functions in order to compare with ground truths of vulnerable code statements in the unsupervised context. Second, we investigate ICVH in the semi-supervised context in which there is a small portion of data having ground truth of core vulnerable code statements. Finally, we inspect the explanatory capability of ICVH by identifying example misclassifications by the reference model and then analysing the reason for these.

From machine learning and data mining perspectives, it seems that the existing methods in interpretable machine learning [Ribeiro et al., 2016, Shrikumar et al., 2017, Lundberg and Lee, 2017, Chen et al., 2018] with adoption are ready to apply. Unfortunately, besides [Chen et al., 2018], none of others can be adopted to be applicable to the specific context of statement-grained vulnerability detection.

We cannot compare with VulDeeLocator [Li et al., 2020] because: i) it cannot work directly with

Datasets	#vul-funcs	#non-vul-funcs
CWE-119	5,582	5,099
CWE-399	1,010	1,313

Table 5.1: Summary statistics of CWE-119 and CWE-399 datasets with the number of vulnerable functions (#vul-funcs) and non-vulnerable functions (#non-vul-funcs).

source code (i.e., it requires to compile source codes to Lower Level Virtual Machine intermediate code), and ii) it requires information relevant to vulnerable statements for extracting tokens from program code according to a given set of vulnerability syntax characteristics, hence it cannot be operated in the unsupervised setting.

5.5.1 Experimental setup

5.5.1.1 Experimental datasets

We used the real-world datasets collected by [Li et al., 2018] which contain the source code of vulnerable functions (vul-funcs) and non-vulnerable functions (non-vul-funcs) obtained from two real-world software datasets, containing buffer error vulnerabilities (CWE-119) and resource management error vulnerabilities (CWE-399). The summary statistics of these datasets are shown in Table 5.1. For both CWE-119 and CWE-399, we remove functions that are identical. The minimum, mean, and maximum length of functions in CWE-399 and CWE-119 are (4; 51; 177) and (4; 21; 164) respectively. For the 1,010 vulnerable functions of CWE399 and 5,582 vulnerable functions of CWE119, the percentage of vulnerable statements and non-vulnerable statements is 5.50% and 8.13% respectively. These percentages between vulnerable and non-vulnerable statements demonstrate that our proposed VCP and VCA measures are reasonable.

Labelling core vulnerable statements for evaluation. The CWE-399 and CWE-119 data sets have only vulnerable and non-vulnerable labels for their source codes. Our aim in this work is to detect the statements responsible for causing the vulnerability. Although our proposed method does not need the information of vulnerable statements at all in the training phase, this information is necessary in evaluating its performance. To obtain this information regarding the location of vulnerable statements in source code for the CWE-399 and CWE-119 data sets, we further processed these data sets. To obtain the ground truth of vulnerable code statements, we used the description of vulnerability information (i.e., the comments and annotations) in the original source code as well as the differences between the vulnerable versions and the fixed versions (i.e., non-vulnerable versions) of the source code.

5.5.1.2 Data processing and embedding

We preprocess the datasets before injecting them into the deep networks. First, we standardise the source code by: removing comments and non-ASCII characters, mapping user-defined variables to symbolic names (e.g., “var1”, “var2”) and user-defined functions to symbolic names (e.g., “func1”, “func2”), and replacing strings with a generic <str> token. Second, we embed statements in source code into vectors. For instance, in the following statement (C programming language) “if(func3(func4(2,2),&var2)!=var11)”: to embed this code statement, we tokenise it to a sequence of tokens (e.g., if,(,func3,(,func4,(,2,2,),&,var2,),!=,var11,)), construct the frequency vector of the statement, and multiply this frequency vector by the statement embedding matrix. The statement embedding matrix represents the learnable variables in our model.

5.5.1.3 Model configuration

We implemented ICVH and L2X in Python using Tensorflow [Abadi et al., 2016], an open-source software library for Machine Intelligence developed by the Google Brain Team. We ran our experiments on a server with an Intel Xeon Processor E5-1660 which had 8 cores at 3.0 GHz and 128 GB of RAM. The length of each function is padded or cut to 100 code statements. For the reference model (i.e., the learning model), we used a bidirectional recurrent neural network (bi-RNN) using LSTM cells, where the size of the hidden states is in $\{128, 256\}$, combined with a deep feedforward neural network having two hidden layers with the size of each hidden layer in $\{100, 200, 300\}$. For L2X, we used the structure with parameters as mentioned in [Chen et al., 2018] and for each dataset, we used 10 epochs as suggested in [Chen et al., 2018] for the training process. For our ICVH method, regarding the first and second bi-RNN, we used LSTM cells where the size of hidden states is in $\{128, 256\}$. The deep feedforward neural networks consisted of two hidden layers with the size of each hidden layer in $\{100, 200, 300\}$. The trade-off parameter λ is in $\{10^{-1}, 10^{-2}\}$.

We employed the Adam optimiser [Kingma and Ba, 2014] with an initial learning rate in $\{0.001, 0.003\}$, while the mini-batch size is 100 and the temperature τ for the Gumbel-softmax distribution is in $\{0.1, 0.5\}$, for both L2X and ICVH. For the reference (learning) model and explaining model (L2X and our ICVH method), we split the data of each dataset into three random partitions. The first partition contains 80% for training, the second partition contains 10% for validation and the last partition contains 10% for testing. We additionally apply gradient clipping regularisation [Pascanu et al., 2013] to prevent over-fitting when training the model.

5.5.1.4 Measures

To compare the performance of RSM and L2X with our proposed ICVH method, we propose two main measures of interest, namely vulnerability coverage proportion (VCP) and vulnerability coverage accuracy (VCA).

The VCP aims to measure the proportion of correctly detected vulnerable statements over all vulnerable statements in a given dataset. The VCP hence is mathematically defined as $\frac{\#detectedVCS}{\#allVCS}$ where $\#detectedVCS$ is the number of vulnerable code statements detected correctly and $\#allCVS$ is the number of all vulnerable code statements in a dataset.

The VCA is considered more strictly, because it measures the ratio of the successfully detected functions over all functions in a dataset. In addition, a function is considered *successfully detected* by a method if this method can detect successfully all vulnerable statements in this function. Mathematically, the VCA can be expressed as $\frac{\#detectedVFunc}{\#allVFunc}$ where $\#detectedVFunc$ is the number of successfully detected functions and $\#allVFunc$ is the number of functions in a dataset.

In addition to VCP and VCA measures, we also reported the label (i.e., Y) classification F1-score on CWE-399 and CWE-119 data sets for our proposed method and baselines.

5.5.2 Experimental results

5.5.2.1 Learning process (the reference model)

We aim to learn a model distribution $p_m(Y \mid F)$ that can approximate the true distribution $p(Y \mid F)$ where y is the label corresponding to the source code F. To obtain $p_m(Y|F)$, we use the network architecture as depicted in Fig. 5.2. We measure the F1-score of the reference model (learning model) on CWE-119 and CWE-399 real-world datasets. As showed in Table 5.2, using this architecture we obtained a high predictive performance for learning the approximate distribution $p_m(Y|F)$. In particular, the learning model obtained 99.25% and 94.29% for F1-score for CWE-399 and CWE-119 respectively.

In the explaining process described in the next section, we aim to explain the reference model by specifying the most important code statements in each function F that have the most significant role for the reference model to make its decision about the corresponding label Y.

Model	Datasets	F1-score
The reference (learning) Model	CWE-399	99.25%
	CWE-119	94.29%

Table 5.2: Performance results in term of F1-score of the reference (learning) model on the testing set of CWE-399 and CWE-119.

5.5.2.2 Explaining code vulnerability highlighting with selected code statements in the unsupervised context

We compared the performance of our ICVH method with L2X [Chen et al., 2018] and the random selection method (RSM) in the unsupervised context for explaining the reference model and highlighting the vulnerable code statements. In this approach we do not use any information about ground truth of vulnerable code statements in the training process. We wanted to find out the top K statements that mostly influence the decision of the vulnerability of each function. The number of selected code statements for each function used in each method is fixed equal to 10 (i.e., $K = 10$). When comparing L2X and ICVH in the explainable or interpretable model, we not only aim to obtain a high F1-score for a good explanation, but also aim to measure how the selected and highlighted statements cover the core vulnerable statements.

The experimental results in Table 5.3 show that our proposed method (ICVH) achieved a higher performance for both VCP and VCA measures, and F1-score compared with L2X on the CWE-399 and CWE-119 datasets. In particular, for CWE-119, our proposed method (ICVH) achieved 89.13% for VCP and 86.27% for accuracy while L2X achieved 83.21% and 77.74% for VCP and accuracy respectively.

The higher F1-score that was achieved by ICVH shows that it can approximate (or achieve better explainability of) the reference model compared with L2X. The larger VCP and VCA measures show that our proposed method can detect vulnerable code statements in vulnerable functions much more accurately and effectively compared with L2X.

Datasets	K	Methods	VCP	VCA	F1-score
CWE-399	10	RSM	36.36%	30.87%	NA
		L2X	80.41%	71.00%	99.10%
		ICVH (ours)	**86.82%**	**80.46%**	**99.40%**
CWE-119	10	RSM	40.37%	33.28%	NA
		L2X	83.21%	77.74%	97.30%
		ICVH (ours)	**89.13%**	**86.27%**	**99.23%**

Table 5.3: Performance results in terms of VCP, VCA and F1-score on the testing set of CWE-399 and CWE-119 for the random selection method (RSM), L2X and our ICVH method (best performance among methods for each dataset in **bold**).

Datasets	K	Methods	VCP	VCA	F1-score
CWE-399	5	ICVH	69.46%	54.65%	99.21%
		S2-ICVH-5	89.05%	85.42%	**100%**
		S2-ICVH-10	**90.67%**	**88.57%**	99.76%
	10	ICVH	86.82%	80.46%	99.40%
		S2-ICVH-5	91.72%	88.54%	**100%**
		S2-ICVH-10	**95.11%**	**92.86%**	99.76%
CWE-119	5	ICVH	67.53%	58.72%	99.39%
		S2-ICVH-5	90.53%	86.84%	**99.52%**
		S2-ICVH-10	**94.24%**	**91.73%**	99.51%
	10	ICVH	89.13%	86.27%	99.23%
		S2-ICVH-5	95.10%	92.85%	**99.51%**
		S2-ICVH-10	**98.63%**	**98.03%**	99.50%

Table 5.4: Performance results in terms of VCP, VCA and F1-score on the testing set of CWE-399 and CWE-119 for our proposed method in the unsupervised context (ICVH) and semi-supervised context (S2-ICVH) (best performance among methods for each value of K in **bold**).

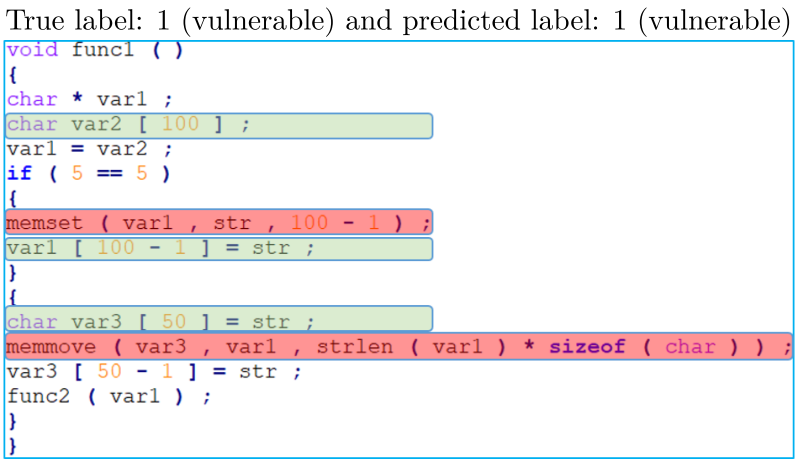
5.5.2.3 Explaining code vulnerability highlighting in the semi-supervised context with the variation of K

We investigated the performance of ICVH for two different contexts, including the unsupervised learning (ICVH) and semi-supervised learning (S2-ICVH) contexts for explaining the reference model and highlighting the vulnerable statements. In the semi-supervised context, we assume that there is a small portion of the training set (i.e., 5% or 10%) having ground truth of vulnerable code statements. We investigated the performance of ICVH from both unsupervised and semi-supervised contexts with some different values of K (i.e., $K = 5, 10$ code statements that are highly relevant to the presence of a vulnerability).

The experimental results in Table 5.4 show that by using a small portion of data having ground truth (i.e., 5% or 10%) of vulnerable code statements, the model performance is significantly increased. For example, for CWE-399, in the case of $K = 10$, the model performance in the unsupervised context (ICVH) achieved 86.82% and 80.46% for VCP and VCA respectively, while the model performance in the semi-supervised context for S2-ICVH-5 (5% of data in the training process having ground truth of vulnerable code statements) and S2-ICVH-10 (10% of data in the training process having ground truth of vulnerable code statements) obtained (91.72% for VCP and 88.54% for VCA) and (95.11% for VCP and 92.86% for VCA) respectively.

The experimental results in Table 5.4 show that the more selected code statements we have, the higher performance in two main measures including VCP and accuracy we obtain. For instance, to dataset CWE-119, in the case with $K = 5$, ICVH obtained 67.53% for VCP and 58.72% for VCA while with $K = 10$, ICVH obtained 89.13% for VCP and 86.27% for VCA respectively.

The high values of F1-score (over 99% for all cases mentioned in Table 5.4) show that our


True label: 1 (vulnerable) and predicted label: 1 (vulnerable)

```
void func1 ( )
{
char * var1 ;
char var2 [ 100 ] ;
var1 = var2 ;
if ( 5 == 5 )
{
memset ( var1 , str , 100 - 1 ) ;
var1 [ 100 - 1 ] = str ;
}
{
char var3 [ 50 ] = str ;
memmove ( var3 , var1 , strlen ( var1 ) * sizeof ( char ) ) ;
var3 [ 50 - 1 ] = str ;
func2 ( var1 ) ;
}
}
```

Figure 5.6: The true and predicted label from the model are shown in the first row. The source code function and selected code statements highlighted relevant to vulnerabilities are shown with $K = 5$.

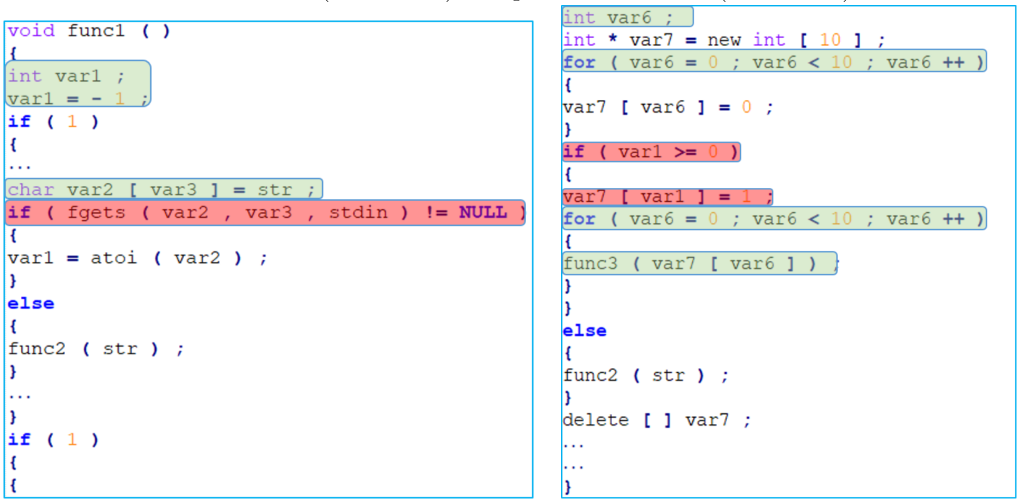
proposed methods can approach very close to (or have a good explanation of) the learning model while the high values of VCP and VCA show that our proposed methods can effectively and efficiently detect vulnerable code statements in vulnerable functions.

5.5.2.4 Visualisation of detected and highlighted code statements

Here we illustrate how we can visualise the highlighted code statements in vulnerable functions, in order to demonstrate the ability of our method to detect and highlight core vulnerable code statements in vulnerable functions to aid security auditors and code developers. We set $K = 5$ for the function in Fig. 5.6 and $K = 10$ for the functions in Fig. 5.7. In these figures, the coloured lines (i.e., the green and red lines) highlight the detected code statements obtained when using our ICVH in the unsupervised context. In addition, each red line specifies the core vulnerable statement obtained from the ground truth and these lines are detected by our method.

For example, in Fig. 5.6, the corresponding function has two core vulnerable statements including "memset (var1 , str , 100 - 1) ;" and "memmove (var3 , var1 , strlen (var1) * sizeof (char)) ;", which lead to a vulnerability, because in this case we initialise var1 as a large buffer that is larger than the small buffer used in the sink (i.e., var1 is larger than var3). Our ICVH method with $K = 5$ can detect these core vulnerable statements that make the corresponding function vulnerable. In Fig. 5.7, the function has some core vulnerable code statements including "if (fgets (var2 , var3 , stdin) != NULL)", which is a potential vulnerability because we read data from the console using fgets(), and "if (var1 >= 0)" and "var7 [var1] = 1 ;" which are also a potential vulnerability in the case we attempt to write to an index of the array that is above the

True label: 1 (vulnerable) and predicted label: 1 (vulnerable)



```
void func1 ( )
{
int var1 ;
var1 = - 1 ;
if ( 1 )
{
...
char var2 [ var3 ] = str ;
if ( fgets ( var2 , var3 , stdin ) != NULL )
{
var1 = atoi ( var2 ) ;
}
else
{
func2 ( str ) ;
}
...
}
if ( 1 )
{
{
```

```
int var6 ;
int * var7 = new int [ 10 ] ;
for ( var6 = 0 ; var6 < 10 ; var6 ++ )
{
var7 [ var6 ] = 0 ;
}
if ( var1 >= 0 )
{
var7 [ var1 ] = 1 ;
for ( var6 = 0 ; var6 < 10 ; var6 ++ )
{
func3 ( var7 [ var6 ] ) ;
}
}
else
{
func2 ( str ) ;
}
delete [ ] var7 ;
...
...
}
```

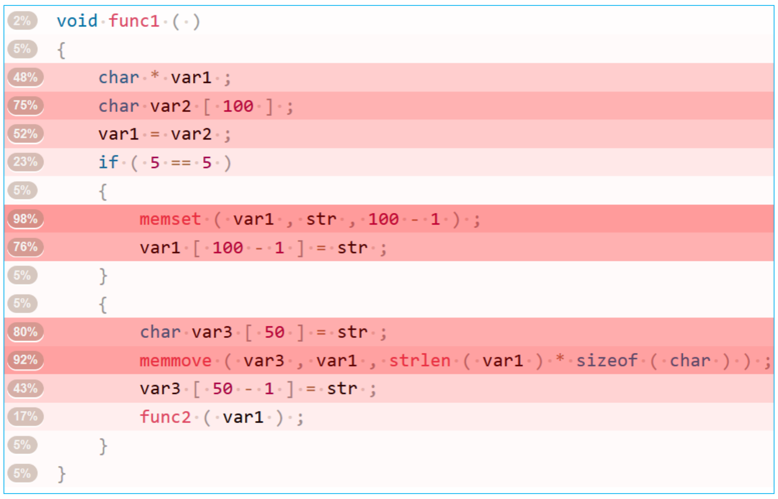
Figure 5.7: The true and predicted label from the model are shown in the first row. The source code function and selected code statements highlighted relevant to vulnerabilities are shown with $K = 10$. The left-hand and right-hand figures are the first and second parts of the function respectively. For demonstration purpose, there are some parts of the function omitted for the brevity.

upper bound. Our ICVH method with $K = 10$ can detect all of these core potential vulnerable code statements that make the corresponding function vulnerable.

Interestingly, we can use the vulnerability relevance probability μ_k associated with each statement to visualise a heat map over the source code as shown in Fig. 5.8. This is intuitive and informative as it shows which statements or blocks of statements are highly relevant to the vulnerabilities.

5.5.2.5 Investigation of misclassification of the non-vulnerable functions

In this section, we investigate the case when some non-vulnerable functions are predicted as vulnerable functions as depicted in Fig. 5.9. These functions appear as non-vulnerable in the ground truth. However, the reference model predicted them as vulnerable. Using $K = 5$, for the left-hand function shown in Fig. 5.9, the green selected code statements "for (var4 $= 0$; var4 $<$ var5 ; var4 ++)" and "var3 [var4] $=$ var2 [var4] ;" can in some cases (e.g., if var2 is larger than var3) lead to a potential vulnerability. For the right-hand function shown in Fig. 5.9, the green selected code statement "wmemset (var1 , str , 50 - 1) ;" will be a vulnerable code statement if we change "50 - 1" into "100 - 1", because in this case we would initialise the source buffer as a buffer that is larger than the buffer used in the sink (i.e., "wcsncat (var4 , var1 , wcslen (var1)) ;"). These are some typical examples for the case when non-vulnerable functions are predicted as vulnerable functions. The main reasons are likely due to: i) the key

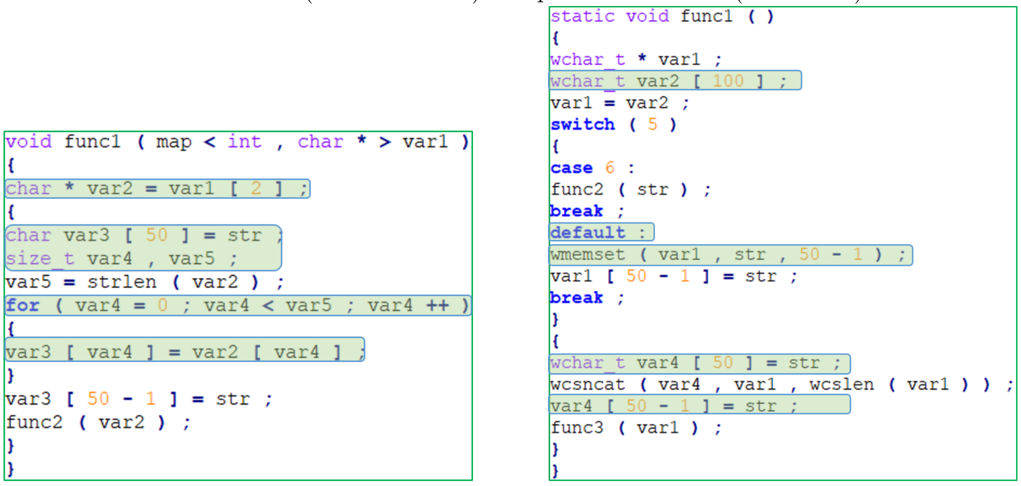


```
2%   void func1 ( )
5%   {
48%      char * var1 ;
75%      char var2 [ 100 ] ;
52%      var1 = var2 ;
23%      if ( 5 == 5 )
5%       {
98%          memset ( var1 , str , 100 - 1 ) ;
76%          var1 [ 100 - 1 ] = str ;
5%       }
5%       {
80%          char var3 [ 50 ] = str ;
92%          memmove ( var3 , var1 , strlen ( var1 ) * sizeof ( char ) ) ;
43%          var3 [ 50 - 1 ] = str ;
17%          func2 ( var1 ) ;
5%       }
5%   }
```

Figure 5.8: Example heat map that represents the vulnerable relevance probabilities over the given source code.

contributed code statements for marking the label of a function can be a potential vulnerability in some specific cases (e.g., depending on behaviour in the calling functions), or ii) the key contributed code statements for marking the label of a function can be a potential vulnerability if we have a minor code change effected in them.

True label: 0 (non-vulnerable) and predicted label: 1 (vulnerable)



```
void func1 ( map < int , char * > var1 )
{
char * var2 = var1 [ 2 ] ;
{
char var3 [ 50 ] = str ;
size_t var4 , var5 ;
var5 = strlen ( var2 ) ;
for ( var4 = 0 ; var4 < var5 ; var4 ++ )
{
var3 [ var4 ] = var2 [ var4 ] ;
}
var3 [ 50 - 1 ] = str ;
func2 ( var2 ) ;
}
}
```

```
static void func1 ( )
{
wchar_t * var1 ;
wchar_t var2 [ 100 ] ;
var1 = var2 ;
switch ( 5 )
{
case 6 :
func2 ( str ) ;
break ;
default :
wmemset ( var1 , str , 50 - 1 ) ;
var1 [ 50 - 1 ] = str ;
break ;
}
{
wchar_t var4 [ 50 ] = str ;
wcsncat ( var4 , var1 , wcslen ( var1 ) ) ;
var4 [ 50 - 1 ] = str ;
func3 ( var1 ) ;
}
}
```

Figure 5.9: The true and predicted labels are show in the first row. The source code functions and selected code statements are shown with $K = 5$.

5.5.2.6 Threats to validity

Key construct validity threat is whether our assessments of the techniques demonstrate identification of vulnerable statements. We used widely accepted measures of accuracy for vulnerability detection from related work in our assessment. Key internal validity threats are the training data used and training approach used. We used a two step process to learn a reference deep learning model to approximate distribution of statement vulnerabilities, and explanatory learned model to specify the top-K statements that contribute to predicted vulnerability. We used two real-world datasets to train and evaluate our models. Key external validity threats include whether our approach will generalise to other vulnerabilities, and whether it will work on other source datasets. We mitigated by using two common but different vulnerabilities, and two real-world and significantly different projects.

5.6 Closing remarks

We have proposed a new method to detect software vulnerabilities at a fine-grained level than the function or program levels in both unsupervised and semi-supervised contexts. Our proposed method aims to maximise the mutual information between selected code statements (i.e., F_S) and the response variable (i.e., Y) of a function or program offered by the reference model trained in the learning phase. By maximising this mutual information, the selected statements are expected to strongly correlate with the existence of a vulnerability, hence potentially containing the core vulnerable statements. In addition, our proposed model is able to play the role of an explanatory model that explains which statements in a given source mostly contribute to the prediction of the reference model. Our experimental results on real-world datasets showed that by using our proposed methods we can detect software vulnerabilities at a fine-grained level effectively and accurately.

Chapter 6

Information-Theoretic End-to-End Models to Identify Code Statements Causing Software Vulnerability

Instead of considering the proposed approach as a learn-to-explain model as mentioned in the previous chapter, in this chapter, we propose a novel end-to-end approach and view our proposed approach as one based on using mutual information to detect the code statements highly relevant to the ground-truth label Y (i.e., vulnerable or non-vulnerable). We aim to tackle the challenge of fine-grained vulnerability detection by formulating it as the problem of learning a set of latent variables for the statements of each individual function, each of which models a statement's relevance to the function's vulnerability. Accordingly, to learn those latent variables, we propose an amortised variational inference framework derived from the maximisation of mutual information.

6.1 Motivations

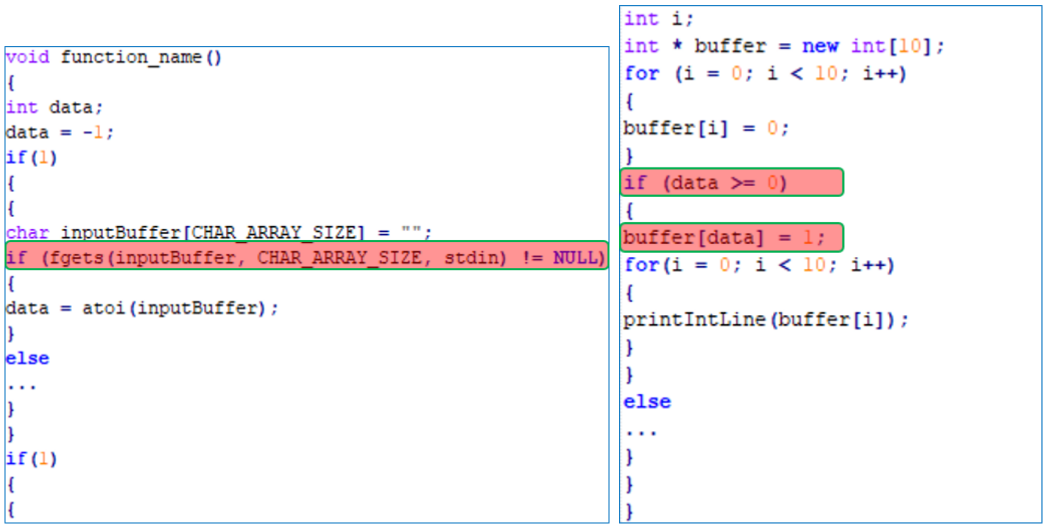
In the field of software security, software vulnerabilities (SVs) are specific potential flaws, glitches, weaknesses or oversights in software. Attackers can leverage these vulnerabilities to carry out malicious actions, such as exposing or altering sensitive information and disrupting/destroying/taking control of a system/program [Dowd et al., 2006]. Due to the rapid growth and dramatic diversity of software, potential vulnerabilities present pervasively in software development and deployment processes, which can create severe threats to cybersecurity, leading to expenditure costs of about USD 600 billion globally each year [McAfee and CSIS, 2017]. These threats call for an urgent need of automatic tools and methods to efficiently and effectively deal with a large amount of vulnerable code with a minimal level of human intervention.

In this chapter, we are interested in software vulnerability detection (SVD) at the source code level, where there exist two typical approaches based on either human expertise or deep (machine) learning approaches. Most previous work in software vulnerability detection [Neuhaus et al., 2007, Shin et al., 2011, Yamaguchi et al., 2011, Almorsy et al., 2012, Li et al., 2016, Grieco et al., 2016, Kim et al., 2017] belongs to the former, which involves the knowledge of domain experts. The performance of these methods can be negatively affected by outdated experience, expertise and underlying biases [Zimmermann et al., 2009]. Recently, deep learning approaches have been used to automatically conduct SVD and have shown great advances, notably in [Dam et al., 2017, Li et al., 2018, Lin et al., 2018].

Despite the promising performance, current deep learning-based methods are only able to detect software vulnerabilities at the function-[Lin et al., 2018, Li et al., 2018] or program-[Dam et al., 2017] level. However, in real-world situations, programs or even functions can consist of hundreds or thousands of lines of code statements and the source of most vulnerabilities often arises from a significantly smaller scope, usually a few core statements. Fig. 1 shows the source code of a simple function that is vulnerable. Among these lines of statements, it turns out that only the statements highlighted in red actually lead to the function's vulnerability. The core statements underpinning a vulnerability are even much sparser in source code of real-world applications.

In this chapter, instead of detecting whether a function[1] is vulnerable, we aim to accurately identify the statements that are highly relevant to a function's vulnerability, which we referred to as the "vulnerability-relevant statements". By highlighting such vulnerability-relevant statements, we can significantly speed up the process of isolating and detecting software vulnerabilities, thereby reducing the time and cost involved. To achieve this aim, we propose a novel probabilistic end-to-end approach, which consists of two components: i) a random selection process ε that picks out a subset $\tilde{F} = \varepsilon(F) \subset F$ (i.e., a source code section consists of many code statements) retaining the most crucial information of the ground-truth label Y, and ii) a classifier acting on $\tilde{F}$ to mimic $p(Y \mid F)$ (i.e., the ground-truth conditional label-data distribution). With this notation, our goal can be formulated to learn the selection process for each F to make $p\left(Y \mid \tilde{F}\right)$ from the classifier as close as possible to $p(Y \mid F)$. To achieve this, two important technical challenges arise: one is the construction of the selection process and the other is the principle to guide the training of the entire system.

[1]It can also be a program or an application. Hereafter, we use "function" to denote a collection of code statements.



```
void function_name()
{
int data;
data = -1;
if(1)
{
{
char inputBuffer[CHAR_ARRAY_SIZE] = "";
if (fgets(inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL)
{
data = atoi(inputBuffer);
}
else
...
}
}
if(1)
{
{
```

```
int i;
int * buffer = new int[10];
for (i = 0; i < 10; i++)
{
buffer[i] = 0;
}
if (data >= 0)
{
buffer[data] = 1;
for(i = 0; i < 10; i++)
{
printIntLine(buffer[i]);
}
}
else
...
}
}
}
```

Figure 6.1: Example of a source code function obtained from the CWE-119 dataset. The left-hand and right-hand figures are the first and second parts of the function with some parts omitted for brevity respectively. For demonstration purpose and simplicity, we choose a simple and short source code function. The statement "if (fgets (inputBuffer, CHAR_ARRAY_SIZE, stdin) != NULL)" is a potential vulnerability because the program reads data from the console without sufficient checking/validation using the "fgets()" system function, and the two statements "if (data >= 0)" and "buffer[data] = 1;" may then attempt to write to an index of the array that exceeds the upper bound.

To address the first challenge, we devise two mechanisms to construct the selection process. Specifically, the first mechanism assumes that each function contains K vulnerability-relevant statements and uses a multinomial distribution to select those K statements [Chen et al., 2018]. To further enhance flexibility, we then propose another mechanism that draws a binary latent vector for each function from independent Bernoulli distributions. An element of the binary vector corresponds to a statement in a function. Importantly, in the semi-supervised context, where a small portion of source code data has vulnerability-relevant statements labelled (i.e., core vulnerable statements) by experts, the second mechanism also enables us to incorporate such ground-truth information of the vulnerability-relevant statements.

To address the second challenge, we first propose a variational inference approach derived from the maximisation of the mutual information between $\tilde{F}$ and Y, which is expected to identify all of the vulnerability-relevant statements of a function. However, this approach may be unable to eliminate the irrelevant statements, that is to say, the statements that it identifies may be a super set of the true vulnerability-relevant statements. To address this, inspired by the information bottleneck theory [Tishby et al., 2000, Tishby and Zaslavsky, 2015], we further propose a regularisation term derived from the minimisation of the mutual information between $\tilde{F}$ and F to ensure that only the vulnerability-relevant statements are kept.

In summary, our main contributions from this chapter can be highlighted as follows:

- We study an important problem of fine-grain-level vulnerability detection, which has a variety of applications in different areas such as software engineering and cybersecurity. Automated deep learning-based techniques for this problem have not yet been well studied.
- We propose a novel probabilistic framework learned by variational inference with various model constructions and training mechanisms, which are derived from an information-theoretic perspective. Our proposed approaches can work effectively and efficiently in both unsupervised and semi-supervised settings, hence providing important modelling tools as well as practical toolboxes for software developers and security experts.
- We comprehensively evaluate our proposed framework with different variants for real-world software datasets in both unsupervised and semi-supervised cases. Our extensive experiments show that our approaches can accurately identify the vulnerability-relevant statements in an end-to-end manner.

6.2 Related work for fine-grain-level vulnerability detection

Deep learning has been applied successfully to source code and binary software vulnerability detection [Dam et al., 2017, Lin et al., 2018, Li et al., 2018, Le et al., 2019b, Nguyen et al., 2019]. However, the mentioned work detects vulnerabilities at either the function or program level, not at the more fine-grained code statement level. Although we conduct experiments to demonstrate our work for fine-grained source code vulnerability detection, ours is also applicable to binary software as in [Le et al., 2019b].

The combination of our first principle (Principle 1) and the multinomial random selection process is similar to [Chen et al., 2018], which was proposed to explain a pre-trained reference model by maximising the relevant mutual information. However, our proposed application is totally different and our work demonstrates that the use of mutual information is an efficient tool for fine-grained vulnerability detection. Other learn-to-explain models [Ribeiro et al., 2016, Shrikumar et al., 2017, Lundberg and Lee, 2017] are though not ready for our application, but might be leveraged to improve our model further. We leave this for future development.

The formulation of our second principle (Principle 2) is an instance of information bottleneck theory [Tishby et al., 2000]. Regarding the application of the information bottleneck theory, Alemi et al. [2016] developed a practical realization of this theory to learn a robust representation for defending against adversarial examples [Goodfellow et al., 2014b], while Wang et al. [2009] applied this theory in the context of semi-supervised learning.

To the best of our knowledge, there is one deep learning-based method, named VulDeeLocator [Li et al., 2020] (posted on ArXiv for fine-grain-level vulnerability detection). However, besides the original source codes F and their vulnerability labels Y, VulDeeLocator needs further information relevant to vulnerable code statements to obtain what is called intermediate code-based vulnerability candidate representation from its preprocessing steps, and then use the intermediate code-based representation in the training and testing processes rather than the original source code F. This is totally different from our proposed approach which can be run in an unsupervised setting (only requires the source codes F and their vulnerability labels Y, and not uses information relevant to vulnerable code statements) for the task of fine-grain-level vulnerability detection. VulDeeLocator not only cannot work in the unsupervised setting, but also cannot work directly with source code (i.e., it requires to compile source code to Lower Level Virtual Machine (LLVM) intermediate code. If source code cannot be compiled to LLVM intermediate code, we cannot use VulDeeLocator).

6.3 Statement-grained Source Code Vulnerability Highlighting (S2CVH)

Most of publicly available datasets only have vulnerability labels (i.e., Y) for the entire source codes (i.e., F) and have no information of code statements causing vulnerabilities. Our proposed methods only require vulnerability labels at the source code level (i.e., Y for the entire source codes) and are capable of pointing out the code statements highly relevant to these vulnerability labels. We hence call this setting as unsupervised, meaning that the training process does not require labels at the code statement level (i.e., ground-truth of vulnerable code statements causing vulnerabilities). In addition, in Section 6.3.4, we assume that a tiny portion of the source codes has labels at the code statement level. We hence name this setting as semi-supervised. Moreover, once trained, given a test source code, our model can provide both the vulnerability label for this entire source code and indicate the code statements inside the source code mostly causing the vulnerability.

We denote a source code section (e.g., a C/C++ function or program) as $F = [\boldsymbol{f}_1, \ldots, \boldsymbol{f}_L]$, which consists of L lines of code statements $\boldsymbol{f}_1, \ldots, \boldsymbol{f}_L$ (L can be a large number, e.g., thousands). In practice, each code statement is represented as a vector, which is extracted by some embedding methods. As those embedding methods are not the focus of this chapter, we leave these details to the experiment section. We assume that F's vulnerability $Y \in \{0,1\}$ (where *1*: vulnerable and *0*: non-vulnerable) is observed (labelled by experts). As previously discussed, there is usually a small subset with K code statements that actually lead to F being vulnerable, denoted as $\tilde{F} = [\boldsymbol{f}_{i_1}, \ldots, \boldsymbol{f}_{i_K}] = [\boldsymbol{f}_j]_{j \in S}$ where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$ $(i_1 < i_2 < \ldots < i_K)$. It is worth noting that for different F, K can be different. Here our goal is to find $\tilde{F}$ (i.e., vulnerability-relevant statements) for each specific F. To select the vulnerability-relevant statements, we propose a learnable random selection process ε, i.e., $\tilde{F} = \varepsilon(F)$, whose training principles and constructions are presented as follows.

6.3.1 Training principles

6.3.1.1 Principle 1

Mutual information is a measure of the dependence between two random variables and it captures how much knowledge of one random variable reduces the uncertainty about the other. Therefore, if we view $\tilde{F}$ and Y as random variables, the selection process ε can be learned by maximising the mutual information between $\tilde{F}$ and Y, formulated as follows:

$$\max_{\varepsilon} \mathbb{I}\left(\tilde{F}, Y\right). \tag{6.1}$$

By the definition of mutual information, we can expand Eq. (6.1) further as the Kullback-Leibler divergence of the product of marginal distributions of $\tilde{F}$ and Y from their joint distribution [Cover and Thomas, 2006]:

$$\begin{aligned}
\mathbb{I}(\tilde{F}, Y) &= \int p\left(\tilde{F}, Y\right) \log \frac{p\left(\tilde{F}, Y\right)}{p\left(\tilde{F}\right) p(Y)} d\tilde{F} dY \\
&= \int p\left(Y \mid \tilde{F}\right) p\left(\tilde{F}\right) \log \frac{p\left(Y \mid \tilde{F}\right) p\left(\tilde{F}\right)}{p\left(\tilde{F}\right) p(Y)} d\tilde{F} dY \\
&= \mathbb{E}_{\tilde{F}}\left[\int p\left(Y \mid \tilde{F}\right) \log \frac{p\left(Y \mid \tilde{F}\right)}{p(Y)} dY\right] \\
&= \mathbb{E}_{\tilde{F}}\left[\int p\left(Y \mid \tilde{F}\right) \log \frac{q\left(Y \mid \tilde{F}\right)}{p(Y)} dY\right] \\
&\quad + \mathbb{E}_{\tilde{F}}\left[D_{KL}\left(p\left(Y \mid \tilde{F}\right) \| q\left(Y \mid \tilde{F}\right)\right)\right] \\
&\geq \mathbb{E}_{\tilde{F}}\left[\int p\left(Y \mid \tilde{F}\right) \log \frac{q\left(Y \mid \tilde{F}\right)}{p(Y)} dY\right] \\
&= \int p\left(Y, \tilde{F}\right) \log \frac{q\left(Y \mid \tilde{F}\right)}{p(Y)} dY d\tilde{F}
\end{aligned}$$

Note that in the above derivation, we introduce a variational distribution $q(Y|\tilde{F})$ to approximate the posterior $p\left(Y \mid \tilde{F}\right)$, hence deriving a variational lower bound of $\mathbb{I}\left(\tilde{F}, Y\right)$ [Alemi et al., 2016, Chen et al., 2018] for which the equality happens if $q\left(Y \mid \tilde{F}\right) = p\left(Y \mid \tilde{F}\right)$. We further proceed as:

$$\begin{aligned}
\mathbb{I}(\tilde{F}, Y) \geq & \int p\left(Y, \tilde{F}, F\right) \log \frac{q\left(Y \mid \tilde{F}\right)}{p(Y)} dY d\tilde{F} dF \\
= & \int p\left(Y \mid F\right) p\left(\tilde{F} \mid F\right) p\left(F\right) \log \frac{q\left(Y \mid \tilde{F}\right)}{p(Y)} dY d\tilde{F} dF \\
= & \mathbb{E}_F \mathbb{E}_{\tilde{F}|F} \left[\int p(Y|F) \log \frac{q(Y|\tilde{F})}{p(Y)} dY \right]
\end{aligned}$$

$$\begin{aligned}
\mathbb{I}(\tilde{F}, Y) \geq \, & \mathbb{E}_F \mathbb{E}_{\tilde{F}|F} \left[\int p(Y|F) \log \frac{q(Y|\tilde{F})}{p(Y)} dY \right] \\
= \, & \mathbb{E}_F \mathbb{E}_{\tilde{F}|F} \left[\sum_Y p(Y|F) \log q(Y|\tilde{F}) \right] + \text{const.}
\end{aligned}$$

Notably, $\tilde{F}|F := \tilde{F} \sim p(\cdot|F) := \varepsilon\left(F\right)$ is the same representation of the random selection process and $p(Y|F)$ is the ground-truth conditional distribution of the code's vulnerability on all of its statements.

To model the conditional variational distribution $q(Y|\tilde{F})$, we introduce a classifier implemented with a neural network, which takes $\tilde{F}$ as input and outputs its vulnerability. With the classifier, our objective is to learn the selection process as well as the classifier to maximise the mutual information:

$$\max_{\varepsilon, q} \left(\mathbb{E}_F \mathbb{E}_{\tilde{F}|F} \left[\sum_Y p(Y|F) \log q(Y|\tilde{F}) \right] \right). \tag{6.2}$$

Principle 1 facilitates a joint training process for the classifier and the selection process. Specifically, the classifier is learned to identify whether a subset of the statements causes a function's vulnerability or not while the selection process is designed to select the best subset according to the feedback of the classifier.

6.3.1.2 Principle 2

With Principle 1, we can learn to pick the statements that are related to a function's vulnerability. However, the principle does not theoretically guarantee to eliminate the statements that are unrelated to vulnerability. Therefore, the set of selected statements can possibly be a superset of the true vulnerability-relevant statements. In the worst case, a selection process can always select all the statements in a function, which is still a valid solution of the above maximisation. To further improve over Principle 1, inspired by information bottleneck theory [Tishby et al., 2000, Slonim and Tishby, 2000], we propose an additional term for training the selection process ε, derived from the following principle:

$$\max_{\varepsilon} \left(\mathbb{I}(\tilde{F}, Y) - \lambda \mathbb{I}(F, \tilde{F}) \right), \tag{6.3}$$

where λ is a hyper-parameter indicating the weight of the second mutual information.

It serves as a regulariser to minimise the mutual information between F and $\tilde{F}$, which encourages $\tilde{F}$ to be as "different" to F as possible. In other words, the selection process prefers to select a smaller subset that excludes the statements unrelated to vulnerability. Accordingly, we can derive an upper bound of the minimisation:

$$\begin{aligned}
\mathbb{I}(\tilde{F}, F) = \int p\left(\tilde{F}, F\right) \log \frac{p\left(\tilde{F}|F\right)}{p(\tilde{F})} d\tilde{F} dF \\
= \int p\left(\tilde{F}, F\right) \log p\left(\tilde{F}|F\right) d\tilde{F} dF - \int p(\tilde{F}) \log p(\tilde{F}) d\tilde{F} \\
\leq \int p\left(\tilde{F}, F\right) \log p\left(\tilde{F}|F\right) d\tilde{F} dF - \int p(\tilde{F}) \log r(\tilde{F}) d\tilde{F} \\
= \mathbb{E}_F \mathbb{E}_{\tilde{F}|F} \left[\log \frac{p\left(\tilde{F}|F\right)}{r(\tilde{F})} \right],
\end{aligned}$$

for any $r\left(\tilde{F}\right)$.

If we combine two terms (i.e., $\mathbb{I}(\tilde{F}, Y)$ and $\mathbb{I}(F, \tilde{F})$) by $\max\left(\mathbb{I}(\tilde{F}, Y) - \lambda \mathbb{I}(F, \tilde{F})\right)$, we can get a unified training objective:

$$\begin{aligned}
\max_{\varepsilon, q} \Bigg(\mathbb{E}_F \mathbb{E}_{\tilde{F}|F} \left[\sum_Y p(Y|F) \log q(Y|\tilde{F}) \right] \\
- \lambda \mathbb{E}_F \mathbb{E}_{\tilde{F}|F} \left[\log \frac{p\left(\tilde{F}|F\right)}{r(\tilde{F})} \right] \Bigg),
\end{aligned} \tag{6.4}$$

where $r\left(\tilde{F}\right)$ is a specified prior distribution that encourages a compact representation of $\tilde{F}$.

6.3.2 Construction of the selection process

After discussing the training objectives derived from the two principles, we introduce the proposed constructions of the selection process ε. Without loss of generality, we assume that all source codes have the length of L statements (i.e., filling with $\mathbf{0}$ (s) for shorter source codes and truncating longer source codes). Next, for each function, we introduce a binary latent vector $Z \in \{0,1\}^L$, each element of which, z_i, indicates whether $\boldsymbol{f}_i$ is related to F's vulnerability. As the generation of Z depends on F, we denote $Z(F)$. With Z, we can further construct $\tilde{F} = \varepsilon(F)$ by $\tilde{F} = Z(F) \odot F$, where $\odot$ represents the element-wise product. To construct Z, we propose two specific strategies with the multivariate Bernoulli (multi-bernoulli) distribution and multinomal (multi-nomial) distribution respectively.

6.3.2.1 Multi-bernoulli random selection process

For this case, we model $Z \sim \prod_{i=1}^{L} \text{Bernoulli}(p_i)$, which yields: $\tilde{\boldsymbol{f}}_i = \begin{cases} \boldsymbol{f}_i & \text{with probability } p_i \\ \mathbf{0} & \text{with probability } 1-p_i \end{cases}$.

We further construct $p_i = \omega_i(F;\alpha)$ where ω is a neural network parameterized by α, taking F as input, and outputting a probability. To learn ω with back-propagation by Principle 1 and 2, we apply a Gaussian mixture model for the continuous relaxation:

$$p(\tilde{f}_i|F) = p_i \mathcal{N}(\tilde{\boldsymbol{f}}_i \mid \boldsymbol{f}_i, \sigma^2) + (1-p_i)\mathcal{N}(\tilde{\boldsymbol{f}}_i \mid 0, \sigma^2),$$

where $\sigma > 0$ is a small number.

To facilitate the learning of Principle 2, we derive $\mathbb{I}(\tilde{F}, F)$:

$$\begin{aligned}
\mathbb{E}_F \mathbb{E}_{\tilde{F}|F}\left[\log \frac{p\left(\tilde{F}|F\right)}{r(\tilde{F})}\right] &= \mathbb{E}_F \mathbb{E}_{\tilde{F}|F}\left[\sum_{i=1}^{L} \log \frac{p\left(\tilde{\boldsymbol{f}}_i|F\right)}{r(\tilde{\boldsymbol{f}}_i)}\right] \\
&= \sum_{i=1}^{L} \mathbb{E}_F \left[\int p(\tilde{F} \mid F) \log \frac{p\left(\tilde{\boldsymbol{f}}_i|F\right)}{r(\tilde{\boldsymbol{f}}_i)} d\tilde{F}\right] \\
&= \sum_{i=1}^{L} \mathbb{E}_F [D_{KL}(p(\tilde{\boldsymbol{f}}_i|F) \| r(\tilde{\boldsymbol{f}}_i))].
\end{aligned}$$

Minimizing $\mathbb{I}(\tilde{F}, F)$ is now equivalent to minimizing the KL divergence between $p(\tilde{\boldsymbol{f}}_i|F)$ and $r(\tilde{\boldsymbol{f}}_i)$. Therefore, one can view $r(\tilde{\boldsymbol{f}}_i)$ as the prior distribution, which is constructed by $r(\tilde{\boldsymbol{f}}_i) = \mathcal{N}(\tilde{\boldsymbol{f}}_i|0, \sigma^2)$. Given the fact that $p(\tilde{\boldsymbol{f}}_i|F)$ is a Gaussian mixture distribution, the intuition is that the prior prefers the small values centered at zero. In this way, $p(\tilde{\boldsymbol{f}}_i|F)$ is encouraged to select fewer number of statements, which explains why Principle 2 serves as a regulariser. Moreover, the KL divergence $D_{KL}(p(\tilde{\boldsymbol{f}}_i|F) \| r(\tilde{\boldsymbol{f}}_i))$ can be computed by the following approximation [Gal and Ghahramani, 2016]:

$$\frac{\omega_i\left(F;\alpha\right)}{2\sigma^2}\left\|\boldsymbol{f}_i\right\|^2+(\log\sigma+\frac{1}{2}\sigma^2)+\text{const.}$$

We employ another neural network $g\left(\tilde{F};\beta\right)$ to define $q(Y|\tilde{F})$. We apply the Gumbel softmax distribution [Maddison et al., 2016, Jang et al., 2016] to do continuous relaxation that allows us to jointly train $\omega\left(\cdot;\alpha\right)$ and $g\left(\cdot;\beta\right)$. Let $a_i, b_i \overset{iid}{\sim}$ Gumbel$(0,1)$, we sample: $Z_i\left(F;\alpha\right)\sim$ Concrete$(\log\omega_i\left(F;\alpha\right),\log\left(1-\omega_i\left(F;\alpha\right)\right))$ where $\tau>0$ is the temperature variable as follows:

$$Z_i\left(F;\alpha\right)=\frac{\exp((\log\omega_i\left(F;\alpha\right)+a_i)/\tau)}{\exp((\log\omega_i\left(F;\alpha\right)+a_i)/\tau)+\exp((\log\left(1-\omega_i\left(F;\alpha\right)\right)+b_i)/\tau)}$$

The final objective function for Eq. (6.4) is rewritten:

$$\max_{\alpha,\beta}\Bigg(\mathbb{E}_F\mathbb{E}_{a,b}\left[\sum_Y p(Y|F)\log g_Y\left(F\odot Z\left(F;\alpha\right);\beta\right)\right]$$
$$-\lambda\mathbb{E}_F\left[\sum_{i=1}^{L}\frac{\omega_i\left(F;\alpha\right)}{2\sigma^2}\left\|\boldsymbol{f}_i\right\|^2\right]\Bigg). \tag{6.5}$$

We note that Eq. (6.5) is the optimisation for Principle 2, whereas that for Principle 1 only involves the first term in this objective function. It is worth noting that the objective function in Eq. (6.5) is meaningful as i) the first term indicates that the variational classifier $g\left(\cdot;\beta\right)$ and the selection network $\omega\left(\cdot;\alpha\right)$ need to be in cooperation in such a way that the latter picks out relevant code statements that provide sufficiently information for the former to predict well the vulnerability label, and ii) the second term favors compact and sparse subsets of code statements by means of penalizing less relevant code statements. As such, the selection network is encouraged to choose the most compact and relevant statements to the vulnerability label. In addition, as shown in Eq. (6.5), the selection process tends to penalize statements having high embedding vector length.

6.3.2.2 Multi-nomial random selection process

In addition to the multi-bernoulli random selection process, inspired by [Chen et al., 2018], we propose another construction, which is a process of selecting maximally K vulnerability-relevant statements out of L statements. Specifically, the selection process consists of K draws from a categorical distribution, each of which randomly selects one statement. Next, we pick those statements that have been selected at least once to form the set of $\tilde{F}$. Similar to [Chen et al., 2018], we use the concrete distribution [Maddison et al., 2016, Jang et al., 2016] as the continuous relaxation of the categorical distribution. The parameter of the concrete distribution is modeled by a neural network $\omega(\cdot, \alpha)$, i.e., Concrete($\log \omega(F; \alpha)$). Given $C^j(F) \sim$ Concrete($\log \omega(F; \alpha)$), we then define the vector $V(F; \alpha) \in \mathbb{R}^L$ as:

$$V_i(F; \alpha) = \max_{1 \le j \le K} C_i^j(F) \; (1 \le i \le L),$$

and arrive at the following optimisation problem for jointly training $\omega(\cdot; \alpha)$ and $g(\cdot; \beta)$ as:

$$\max_{\alpha, \beta} \left(\mathbb{E}_F \mathbb{E}_a \left[\sum_Y p(Y|F) \log g_Y(F \odot V(F; \alpha); \beta) \right] \right). \tag{6.6}$$

In addition, the resulting optimisation problem in Eq. (6.6) is developed by leveraging Principe 1 and the multi-nomial selection process, hence can be thought as an adoption of L2X [Chen et al., 2018] to our specific problem of statement-grained vulnerability detection. However, this variant S2CVH1-mulN (Principle 1 + multi-nomial) is not capable of penalizing unrelated code statements (i.e., the number of selected code statements K must be specified beforehand) and extending to the semi-supervised setting.

6.3.2.3 Combination of principles and random selection processes

Here we first compare the two proposed principles. Specifically, Principle 2 is an extension of Principle 1, by additionally minimizing $\mathbb{I}(F, \tilde{F})$, which is encouraged to select a smaller subset that excludes the statements unrelated to a vulnerability. For the multi-bernoulli selection process, we do not need to specify K in advance, as it selects the statements with independent Bernoulli distributions while for the multi-nomial one, K has to be set as a hyper-parameter. On this point, the multi-bernoulli one is therefore more flexible. Moreover, one can figure out that Principle 2 only works with the multi-bernoulli selection process, which helps to automatically learn the number of vulnerability-relevant statements for an individual function (we simply choose a statement $\boldsymbol{f}_i$ into $\tilde{F}$ if $\omega_i(F; \alpha) = \mathbb{P}(Z_i = 1) \ge 0.5$). Finally, another important advantage of multi-bernoulli is its ability to work in the semi-supervised setting, which will be elaborated on later.

By combining the principles and selection processes, three variants are proposed in this chapter, i.e., S2CVH1-mulN (Principle 1 + multi-nomial), S2CVH1-mulB (Principle 1 + multi-bernoulli), and S2CVH2-mulB (Principle 2 + multi-bernoulli). We summarise these variants in Table 6.1 where we denote some notations as: i) P1 stands for "requiring to specify the number of selected statements K beforehand", ii) P2 stands for "allowing to automatically figure out the number of selected statements", and iii) P3 stands for "allowing to incorporate the annotations of the core vulnerable statements in ground truth to our network design in the semi-supervised setting". In the experiment section, we will examine those comparisons empirically.

Model variant	Principle	Selection process	Property
S2CVH1-mulN	1	Multi-nomial	P1
S2CVH1-mulB	1	Multi-bernoulli	P2, P3
S2CVH2-mulB	2	Multi-bernoulli	

Table 6.1: Combination of two principles and two random selection processes.

6.3.3 Working process of our proposed end-to-end approach

In the training phase, our proposed methods only require the source code data (i.e., F) and the corresponding source-code vulnerability labels (i.e., Y) and do not need any information of vulnerable code statements (i.e., the unsupervised setting). In particular, our model includes two key components: the selection network $\omega\left(\cdot;\alpha\right)$ and the variational classifier $g\left(\cdot;\beta\right)$. In what follows, we present the technical details of the training and testing phases.

6.3.3.1 Training phase

Given a source code F including L statements, the selection network outputs $\omega\left(F;\alpha\right)$ either in $[0,1]^L$(for the multi-bernoulli selection process) or in the simplex $\Delta_{L-1}=\{\boldsymbol{\pi}\in\mathbb{R}^L:\left\Vert \boldsymbol{\pi}\right\Vert_1=1\,\text{and}\,\boldsymbol{\pi}\geq\mathbf{0}\}$ (for the multi-nomial selection process) for which each coordinate $\omega_i\left(F;\alpha\right)$ represents the influence level of the corresponding statement to the vulnerability label. Subsequently, we sample a binary latent vector $Z\in\{0,1\}^L$ (i.e., the generation of Z depends on F, so we denote $Z(F)$) and form the subset $\tilde{F}=Z(F)\odot F$. We note that $\tilde{\boldsymbol{f}_i}=\boldsymbol{f}_i$ or $\mathbf{0}$ depending on $Z_i(F)=1$ or 0 and the probability of the event $\tilde{\boldsymbol{f}_i}=\boldsymbol{f}_i$ is proportional to the coordinate $\omega_i\left(F;\alpha\right)$, hence the statement with higher influence level is more likely selected. The subset $\tilde{F}$ is then fed to the variational classifier $g\left(\cdot;\beta\right)$ to predict the vulnerability label (i.e., Y) as accurately as possible according to the objective functions in Eq. (6.6) or Eq. (6.5). Therefore, the selection network and the variational classifier need to cooperate to fulfill the task. Specifically, the selection network is trained to pick out the most influential statements in order that the variational classier has enough information to predict well the vulnerability label. Moreover, Principles 1 and 2 base on mutual information, which eventually leads to minimise the loss of the variational classifier w.r.t the ground-truth vulnerability label, but different from Principle 1, our Principle 2 has a sparsity penalty term that serves as a decent regulariser to further filter out the irrelevant statements for achieving smaller subsets. The diagram of our training phase is shown in Fig. 6.2.

6.3.3.2 Testing phase

After the training phase, the selection network is capable of selecting the most vulnerability-relevant statements of a given source code F by means of offering high value for the corresponding coordinates $\omega_i(F;\alpha)$, meaning that $\omega_i(F;\alpha)$ represents the influence level of the statement $\boldsymbol{f}_i$. We hence can base on the magnitude of $\omega_i(F;\alpha)$ to pick out the most relevant statements. Particularly, for the multi-bernoulli selection process, we simply decide to select $\boldsymbol{f}_i$ if $\omega_i(F;\alpha) \geq 0.5$, whilst for the multi-nominal selection process, we pick the top K relevant statements based on the top K values of $\omega(F;\alpha)$. The diagram of our testing phase is shown in Fig. 6.3.

6.3.4 Semi-supervised setting

In real-world source code data, we may encounter situations where the core vulnerable statements in a small proportion of functions are manually annotated and we call this as the semi-supervised setting. For the multi-bernoulli selection process, it is very convenient to incorporate the annotations of the core vulnerable statements. Specifically, let $F_c = [\boldsymbol{f}_{i_1}, ..., \boldsymbol{f}_{i_m}]$ be the core vulnerable statements of an annotated function. For all the annotated functions, we can leverage such ground-truth information by adding the maximisation of a log likelihood as an additional training objective:

$$\max\left(\sum_{k\in I_c}\log p_k + \sum_{k\notin I_c}\log(1-p_k)\right),$$

where $I_c = [i_1, ..., i_m]$. We then add the above objective function to the main objective function in Eq. (6.5) with the trade-off parameter $\nu > 0$.

6.4 Implementation and results

6.4.1 Proposed approach and baselines

We revise the baselines to compare with our proposed methods in terms of our Principles 1 and 2, and the multi-bernoulli and multi-nominal selection processes. From machine learning and data mining perspectives, it seems that the existing methods in interpretable machine learning

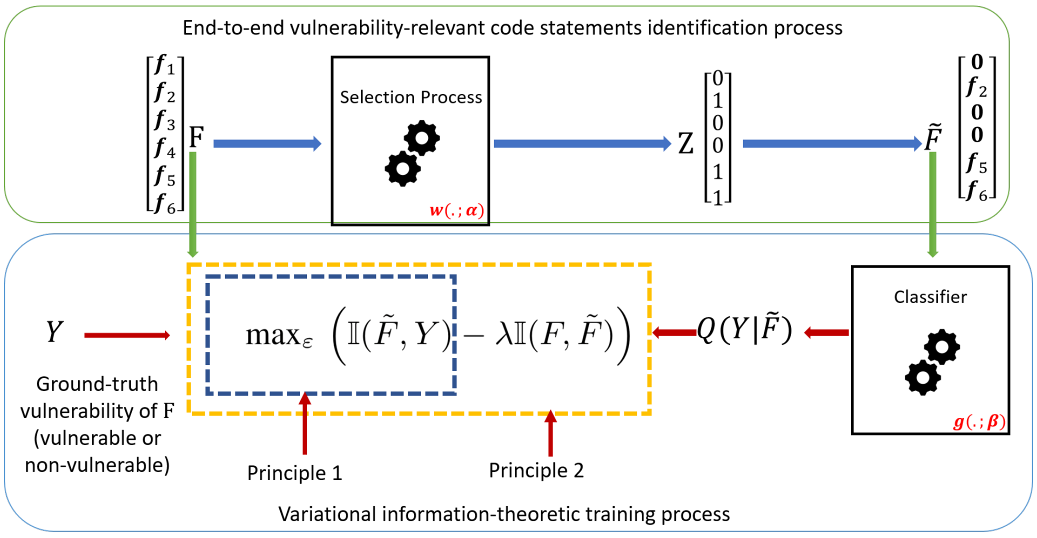

End-to-end vulnerability-relevant code statements identification process
$[f_1, f_2, f_3, f_4, f_5, f_6]$ F
Selection Process
$w(.;\alpha)$
Z $[0,1,0,0,1,1]$
$\tilde{F}$ $[0, f_2, 0, 0, f_5, f_6]$
Y
Ground-truth vulnerability of F (vulnerable or non-vulnerable)
$\max_\varepsilon \left(\mathbb{I}(\tilde{F}, Y) - \lambda \mathbb{I}(F, \tilde{F})\right)$
$Q(Y|\tilde{F})$
Classifier
$g(.;\beta)$
Principle 1
Principle 2
Variational information-theoretic training process

Figure 6.2: Training phase of our proposed end-to-end approach. For the visualization purpose, assuming that we have a source code section $F = [\boldsymbol{f}_1, \boldsymbol{f}_2, \boldsymbol{f}_3, \boldsymbol{f}_4, \boldsymbol{f}_5, \boldsymbol{f}_6]$ going through the selection network to obtain the output probabilities for statements in F as $Z = [0, 1, 0, 0, 1, 1]$. Therefore, we have $\widetilde{F} = [\mathbf{0}, \boldsymbol{f}_2, \mathbf{0}, \mathbf{0}, \boldsymbol{f}_5, \boldsymbol{f}_6]$. The values of F and $\widetilde{F}$ are then used in the objective functions in Eq. (6.6) or Eq. (6.5) in the training process.

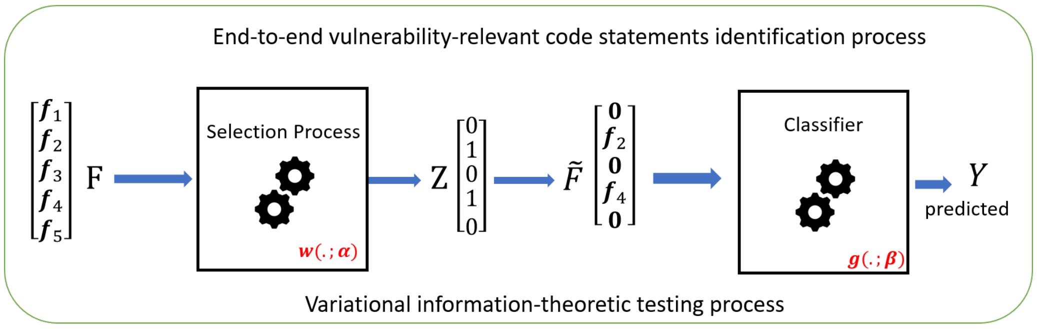

End-to-end vulnerability-relevant code statements identification process
$[f_1, f_2, f_3, f_4, f_5]$ F
Selection Process
$w(.;\alpha)$
Z $[0,1,0,1,0]$
$\tilde{F}$ $[0, f_2, 0, f_4, 0]$
Classifier
$g(.;\beta)$
Y
predicted
Variational information-theoretic testing process

Figure 6.3: Testing phase of our proposed end-to-end approach using the trained model obtained from the training phase. For the visualization purpose, assuming that we aim to obtain the vulnerable statements and the label for a source code section $F = [\boldsymbol{f}_1, \boldsymbol{f}_2, \boldsymbol{f}_3, \boldsymbol{f}_4, \boldsymbol{f}_5]$. Firstly, F goes through the trained selection network to obtain the output $Z = [0, 1, 0, 1, 0]$. Therefore, we have $\widetilde{F} = [\mathbf{0}, \boldsymbol{f}_2, \mathbf{0}, f_4, \mathbf{0}]$. The values of $\widetilde{F}$ are then fed to the trained variational classifier network to predict the vulnerability label Y of F.

[Ribeiro et al., 2016, Shrikumar et al., 2017, Lundberg and Lee, 2017, Chen et al., 2018] with adoption are ready to apply. Unfortunately, besides [Chen et al., 2018], none of others can be adopted to be applicable to the specific context of statement-grained vulnerability detection.

We cannot compare with VulDeeLocator [Li et al., 2020] because: i) it cannot work directly with source code (i.e., it requires to compile source codes to Lower Level Virtual Machine intermediate code) and ii) it requires information relevant to vulnerable statements for extracting tokens from program code according to a given set of vulnerability syntax characteristics, hence it cannot be operated in the unsupervised setting.

In fact, among three variants of our proposed approach, namely, S2CVH1-mulN, S2CVH1-mulB and S2CVH2-mulB (c.f. Table 6.1), the variant S2CVH1-mulN, which is developed based on Principle 1 (maximising the mutual information) and the multi-nominal selection process can be regarded as an adoption of L2X [Chen et al., 2018] towards our application. We hence can consider S2CVH1-mulN as a baseline to compare with the other two in terms of the effectiveness of principles and selection processes. Specifically, we demonstrate that Principle 2 is superior Principle 1 in this specific context since the later can maintain comparable performance with the former, whilst always selecting smaller subset in a source code. Moreover, the multi-bernoulli selection process is also superior the multi-nominal one as it is convenient to extend to the semi-supervised setting and capable of automatically inferring the number of selected code statements, whilst the number of selected code statements K must be specified beforehand in the another selection process.

Another naive baseline that we consider is the random selection method (RSM), where we randomly choose statements from functions in order to match with the ground truth of vulnerable code statements in the unsupervised setting.

In addition to comparing to the baselines in the unsupervised setting, we conduct experiments for S2CVH2-mulB in the semi-supervised setting for which we assume a tiny portion of source codes has ground-truth of core vulnerable statements causing vulnerabilities. With this setting, we aim to illustrate that our proposed multi-bernoulli selection process is useful and efficient in this practical context. Finally, we inspect the explanatory capability of our proposed method (i.e., S2CVH2-mulB) by identifying example misclassification generated by the model and then analyse the reason for these.

6.4.2 Experimental setup

6.4.2.1 Experimental datasets

We used the real-world datasets collected by [Li et al., 2018] which contain the source code of vulnerable functions (vul-funcs) and non-vulnerable functions (non-vul-funcs) obtained from two real-world software project datasets, consisting of buffer error vulnerabilities (the CWE-119 vulnerability category has 5,582 vul-funcs and 5,099 non-vul-funcs) and resource management error vulnerabilities (the CWE-399 vulnerability category has 1,010 vul-funcs and 1,313 non-vul-funcs). For both the CWE-119 and CWE-399 categories, we removed functions that are identical. The minimum, mean, and maximum length of functions in CWE-399 and CWE-119 are (4; 51; 177) and (4; 21; 164) respectively.

6.4.2.2 Labelling core vulnerable statements for evaluation

The CWE-399 and CWE-119 datasets have only vulnerable and non-vulnerable labels for their source codes. Our aim in this work is to detect the statements responsible for causing the vulnerability. Although our proposed methods do not need the information of vulnerable statements at all in the training phase, this information is necessary in evaluating their performance. To obtain this information regarding the location of vulnerable statements in source code for the CWE-399 and CWE-119 datasets, we further processed these datasets. To obtain the ground truth of vulnerable code statements, we used the description of vulnerability information (i.e., the comments and annotations) in the original source code as well as the differences between the vulnerable versions and the fixed versions (i.e., non-vulnerable versions) of the source code. In addition, for 1,010 vulnerable functions of CWE399 and 5,582 vulnerable functions of CWE119, the percentage between vulnerable statements and non-vulnerable statements is 5.50% and 8.13% respectively.

6.4.2.3 Data processing and embedding

We preprocessed the datasets before injecting them into deep networks. First, we standardised the source code by: removing comments and non-ASCII characters, mapping user-defined variables to symbolic names (e.g., "var1", "var2") and user-defined functions to symbolic names (e.g., "func1", "func2"), and replacing strings with a generic <str> token. Second, we embedded source code statements into vectors. For instance, considering the following statement (using C programming language) "if(func3(func4(2,2),&var2)!=var11)", to embed this code statement, we tokenised it to a sequence of tokens (e.g., if,(,func3,(,func4, (,2,2,),&,var2,),!=,var11,)), and then we used a 150-dimensional token embedding followed by a Dropout layer with a dropped fixed probability $p = 0.2$ and (a 1D convolutional layer with the filter size 150 and kernel size 3, and a 1D max pooling layer) or (a 1D max pooling layer) to encode each statement in a function F. Finally, a mini-batch of functions in which each function had L encoded statements were fed to the models.

6.4.2.4 Measures and evaluation

To evaluate the performance of the proposed methods and baselines in detecting the core vulnerable code statements, we proposed two measures: vulnerability coverage proportion (VCP) and vulnerability coverage accuracy (VCA).

The VCP aims to measure the proportion of correctly detected vulnerable statements over all vulnerable statements in a given dataset. The VCP hence is mathematically defined as $\frac{\#detectedVCS}{\#allVCS}$ where $\#detectedVCS$ is the number of vulnerable code statements detected correctly and $\#allCVS$ is the number of all vulnerable code statements in a dataset.

The VCA is considered more strictly, because it measures the ratio of the successfully detected functions over all functions in a dataset. In addition, a function is considered *successfully detected* by a method if this method can detect successfully all vulnerable statements in this function. Mathematically, the VCA can be expressed as $\frac{\#detectedVFunc}{\#allVFunc}$ where $\#detectedVFunc$ is the number of successfully detected functions and $\#allVFunc$ is the number of functions in a dataset.

In addition to VCP and VCA measures, we also reported the label (i.e., Y) classification accuracy (ACC) on CWE-399 and CWE-119 datasets for our proposed methods (i.e., S2CVH1-mulN, S2CVH1-mulB and S2CVH2-mulB).

6.4.2.5 Model configuration

For training our proposed methods including S2CVH1-mulN, S2CVH1-mulB and S2CVH2-mulB, we implemented these methods in Python using Tensorflow [Abadi et al., 2016]. The length of each function is padded or truncated to with $L = 100$ code statements. The trade-off parameter λ is in $\{10^{-1}, 2\times10^{-1}, 10^{-2}, 2\times10^{-2}, 2\times10^{-3}\}$ while σ is in $\{2\times10^{-1}, 3\times10^{-1}, 2\times10^{-2}, 3\times10^{-2}\}$ and ν is in $\{10^{-1}, 10^{-2}\}$. For the networks $\omega(\cdot;\alpha)$ and $g(\cdot;\beta)$, we used deep feedforward neural networks having three and two hidden layers with the size of each hidden layer in $\{100, 300\}$ respectively. The dense hidden layers are followed by a ReLU function as nonlinearity and Dropout [Srivastava et al., 2014] with a retained fixed probability $p = 0.8$ as regularisation. The last dense layer of the network $\omega(\cdot;\alpha)$ for learning a discrete distribution is followed by a sigmoid function for the multi-bernoulli selection process or a softmax function for the multinomial selection process while the last dense layer of the network $g(\cdot;\beta)$ is followed by a softmax function for predicting.

We employed the Adam optimiser [Kingma and Ba, 2014] with an initial learning rate of 10^{-3}, while the mini-batch size is 100 and the temperature τ for the Gumbel softmax distribution is in $\{0.5, 0.7\}$. We split the data of each dataset into three random partitions. The first partition contains 80% for training, the second partition contains 10% for validation and the last partition contains 10% for testing. For each dataset, we used 10 epochs for the training process. We additionally applied gradient clipping regularisation to prevent over-fitting. For each method, we ran the corresponding model 5 times and reported the averaged VCP and VCA measures as well as the ACC. We ran our experiments on an Intel Xeon Processor E5-1660 which has 8 cores at 3.0 GHz and 128 GB RAM.

For the random selection method (RSM), we first randomly chose K (e.g., 10 or 15) code statements from each function in the CWE-119 and CWE-399 datasets. We then compute the VCP and VCA measures of RSM in these datasets in the unsupervised setting to compare with our proposed methods (i.e., S2CVH1-mulN, S2CVH1-mulB and S2CVH2-mulB).

6.4.3 Experimental results

6.4.3.1 Code vulnerability highlighting with selected code statements in the unsupervised setting

We compared the performance of RSM with our proposed methods, namely, S2CVH1-mulN, S2CVH1-mulB and S2CVH2-mulB in the unsupervised setting (i.e., we do not use any information of ground truth of vulnerable code statements in the training process) for highlighting the vulnerable code statements.

The experimental results in Table 6.2 for the CWE-399 vulnerability category show that our S2CVH2-mulB method achieved the most effective performance in terms of average-K (i.e., for our S2CVH1-mulB and S2CVH2-mulB methods, these can automatically obtain the optimal number of selected code statements for each function, so we can then compute the average number of selected statements for all functions in the dataset), VCP and VCA compared with the S2CVH1-mulB, S2CVH1-mulN and RSM methods. In particular, S2CVH2-mulB has a average-K value 9.3 and obtained 87.2% for VCP and 82.0% for VCA while S2CVH1-mulB has a average-K value 10.9 and obtained 82.4% for VCP and 75.0% for VCA.

Our S2CVH1-mulN method obtained 79.1% for VCP and 69.0% for VCA using K = 15. For the S2CVH1-mulN method, we need to specify the number of selected statements in each function beforehand, so we have the same fixed value of selected statements for all functions in the dataset. In contrast, RSM using K = 15 can only achieve a value of 48.7% for VCP and 38.0% for VCA. The S2CVH1-mulN and RSM methods need to select more statements (K) to achieve high values for VCP and VCA compared with S2CVH1-mulB and S2CVH2-mulB methods, especially for the S2CVH2-mulB method.

The experimental results in Table 6.3 for the CWE-119 vulnerability category again show that our S2CVH2-mulB method obtained the most effective performance in terms of average-K, VCP

Dataset	Methods	average-K/K	VCP	VCA	ACC
CWE-399	S2CVH2-mulB	**9.3**	**87.2%**	**82.0%**	**92.9%**
	SSCVH1-mulB	10.9	82.4%	75.0%	87.1%
	S2CVH1-mulN	10	73.7%	61.0%	88.0%
	adopted from L2X [Chen et al., 2018]	15	79.1%	69.0%	88.9%
	RSM	15	48.7%	38.0%	NA

Table 6.2: Performance results for VCP, VCA and ACC on CWE- 399 (best performance in **bold**).

Dataset	Methods	average-K/K	VCP	VCA	ACC
CWE-119	S2CVH2-mulB	**6.9**	**94.2%**	**91.8%**	89.3%
	SSCVH1-mulB	7.2	92.4%	87.6%	89.0%
	S2CVH1-mulN	7	89.1%	84.5%	89.0%
	adopted from L2X [Chen et al., 2018]	10	93.5%	90.9%	**90.0%**
	RSM	10	49.9%	46.9%	NA

Table 6.3: Performance results for VCP, VCA and ACC on CWE-119 (best performance in **bold**).

and VCA compared with S2CVH1-mulB, S2CVH1-mulN and RSM. In particular, for a average-K of 6.9, S2CVH2-mulB achieved 94.2% for VCP and 91.8% for VCA.

We note that all of our proposed methods including S2CVH2-mulB, S2CVH1-mulB and S2CVH1-mulN achieved a high classification accuracy on the label prediction (i.e., Y) of functions. All of the methods obtained an classification accuracy (ACC) on the label prediction higher than 87% for both CWE-399 and CWE-119. This means that all of our proposed methods have achieved a good highlighting performance in terms of making label predictions and highlighting the vulnerable code statements.

6.4.3.2 Code vulnerability highlighting in the semi-supervised setting

We investigate the performance of our S2CVH2-mulB method in the unsupervised setting compared with the semi-supervised setting for highlighting the vulnerable statements. In the semi-supervised setting, we assume that there is a small portion of the training set (i.e., 5% named S2CVH2-mulB-S5, 10% named S2CVH2-mulB-S10) having ground truth of labelled vulnerable code statements.

The experimental results in Table 6.4 for both CWE-399 and CWE-119 show that by using a small portion of data having ground truth (i.e., 5% or 10%) of vulnerable code statements, the model performance is increased for CWE-399 and significantly increased for CWE-119. For instance, for CWE-119, the model performance in the unsupervised setting (S2CVH2-mulB) with a average-K value 6.9 obtained 94.2% for VCP and 91.8% for VCA while the model performance in the semi-supervised setting for S2CVH2-mulB-S5 and S2CVH2-mulB-S10 having average-K value (6.6 and 6.3) achieved (96.2% for VCP and 94.7% for VCA) and (99.5% for VCP and

99.5% for VCA) respectively.

Datasets	Methods	average-K	VCP	VCA	ACC
CWE-399	S2CVH2-mulB	9.3	87.2%	82.0%	92.9%
	S2CVH2-mulB-S5	8.9	87.8%	82.0%	95.1%
	S2CVH2-mulB-S10	**8.8**	**88.5%**	**83.0%**	**96.4%**
CWE-119	S2CVH2-mulB	6.9	94.2%	91.8%	89.3%
	S2CVH2-mulB-S5	6.6	96.2%	94.7%	92.8%
	S2CVH2-mulB-S10	**6.3**	**99.5%**	**99.5%**	**94.2%**

Table 6.4: Performance results for VCP, VCA and ACC on CWE-399 and CWE-119 for our S2CVH2-mulB in the unsupervised and semi-supervised settings (best performance in **bold**).

In the semi-supervised setting, both S2CVH2-mulB-S5 and S2CVH2-mulB-S10 achieved a high classification accuracy (ACC*,* higher than 92%) for both CWE-399 and CWE-119 on the label prediction (i.e., Y) of functions. This means that in the semi-supervised setting, our method (S2CVH2-mulB) achieves a better highlighting performance compared with that in the unsupervised setting in terms of making label prediction and highlighting the vulnerable code statements.

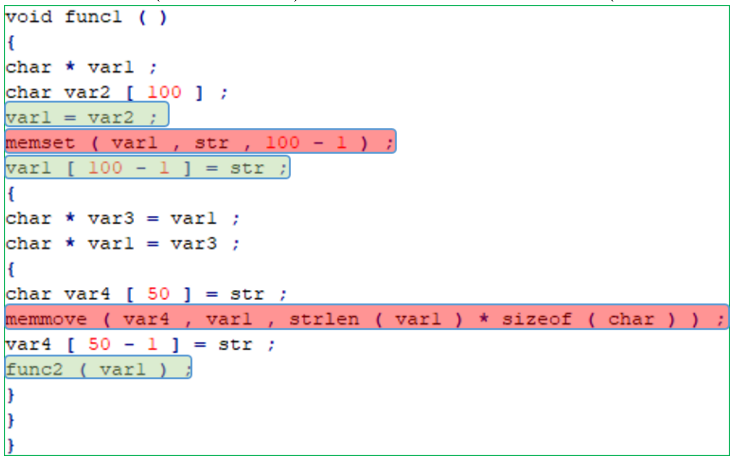
6.4.3.3 Explanatory capability of our proposed method

Visualisation of detected and highlighted code statements. In this section, we illustrate how we visualise the highlighted code statements in vulnerable functions. This shows the ability of our proposed method as a useful tool for developers to identify vulnerable statements. Using our S2CVH2-mulB method, the number of code statements detected in each function will be selected automatically. In Figures (6.6, 6.5 and 6.6), the coloured lines (i.e., the green and red lines) highlight the detected code statements obtained when using S2CVH2-mulB in the unsupervised setting. In addition, each red line specifies the core vulnerable statement obtained from the ground truth and these lines are also detected by our method.

For example, in Fig. 6.6, the corresponding function has two core vulnerable statements including "memset (var1 , str , 100 - 1) ;" and "memmove (var4 , var1 , strlen (var1) * sizeof (char)) ;", which lead to a vulnerability, because in this case we initialise var1 as a large buffer that is larger than the small buffer used in the sink (i.e., var1 is larger than var4*).* Our S2CVH2-mulB method, with the number of automatically detected code statements equal to 5, can detect these core vulnerable statements that make the corresponding function vulnerable.

In Fig. 6.5, the function has some core vulnerable code statements including "if (fgets (var2 , var3 , stdin) != NULL)", which is a potential vulnerability because we read data from the console using fgets(), and "if (var1 > wcslen (var7))" which is also a potential vulnerability

True label: 1 (vulnerable) and predicted label: 1 (vulnerable)

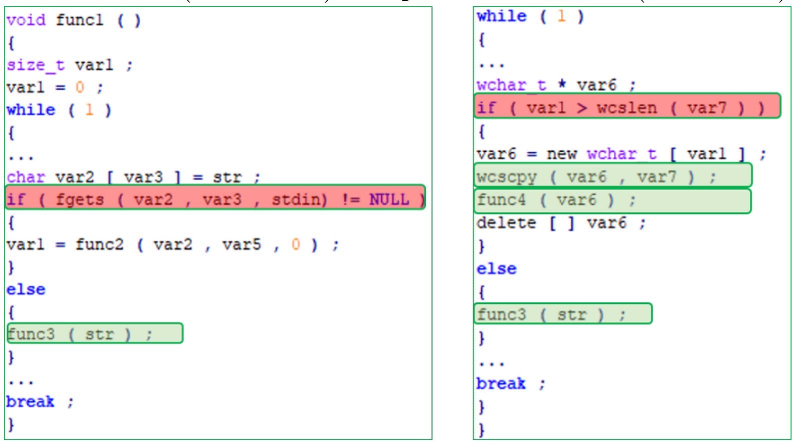


```
void func1 ( )
{
char * var1 ;
char var2 [ 100 ] ;
var1 = var2 ;
memset ( var1 , str , 100 - 1 ) ;
var1 [ 100 - 1 ] = str ;
{
char * var3 = var1 ;
char * var1 = var3 ;
{
char var4 [ 50 ] = str ;
memmove ( var4 , var1 , strlen ( var1 ) * sizeof ( char ) ) ;
var4 [ 50 - 1 ] = str ;
func2 ( var1 ) ;
}
}
}
```

Figure 6.4: The true and predicted labels from the model are shown in the first row. The source code function and selected code statements highlighted relevant to vulnerabilities are shown with colours (see text).

because there is no maximum limitation for memory allocation. Our S2CVH2-mulB method, with the number of automatically detected code statements equal to 6, can detect and highlight all of these core potential vulnerable code statements that make the corresponding function vulnerable.

True label: 1 (vulnerable) and predicted label: 1 (vulnerable)



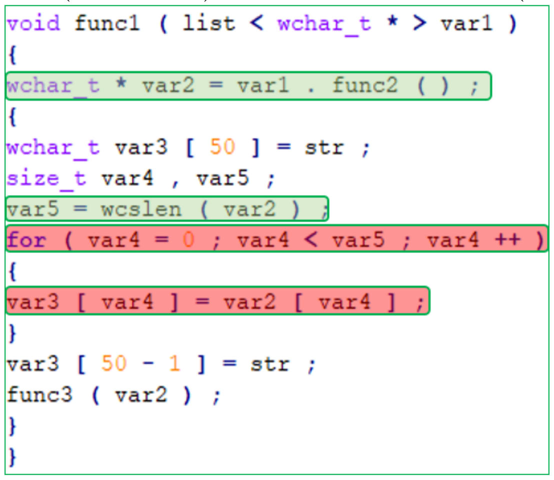
```
void func1 ( )
{
size_t var1 ;
var1 = 0 ;
while ( 1 )
{
...
char var2 [ var3 ] = str ;
if ( fgets ( var2 , var3 , stdin) != NULL )
{
var1 = func2 ( var2 , var5 , 0 ) ;
}
else
{
func3 ( str ) ;
}
...
break ;
}
```

```
while ( 1 )
{
...
wchar_t * var6 ;
if ( var1 > wcslen ( var7 ) )
{
var6 = new wchar_t [ var1 ] ;
wcscpy ( var6 , var7 ) ;
func4 ( var6 ) ;
delete [ ] var6 ;
}
else
{
func3 ( str ) ;
}
...
break ;
}
}
```

Figure 6.5: The true and predicted labels from the model are shown in the first row. The source code function and selected code statements highlighted relevant to vulnerabilities are shown with colours. The left-hand and right-hand figures are the first and second parts of the function with some parts omitted for brevity respectively.

In Fig. 6.6, the function has two key vulnerable statements including “for (var4 = 0 ; var4 < var5 ; var4 ++)” and “var3 [var4] = var2 [var4] ;”, which can in some cases (e.g., if var2 is larger than var3) lead to a potential vulnerability. Our S2CVH2-mulB method, with the number of automatically detected code statements equal to 4, can detect these key vulnerable statements that make the corresponding overall function vulnerable.

True label: 1 (vulnerable) and predicted label: 1 (vulnerable)



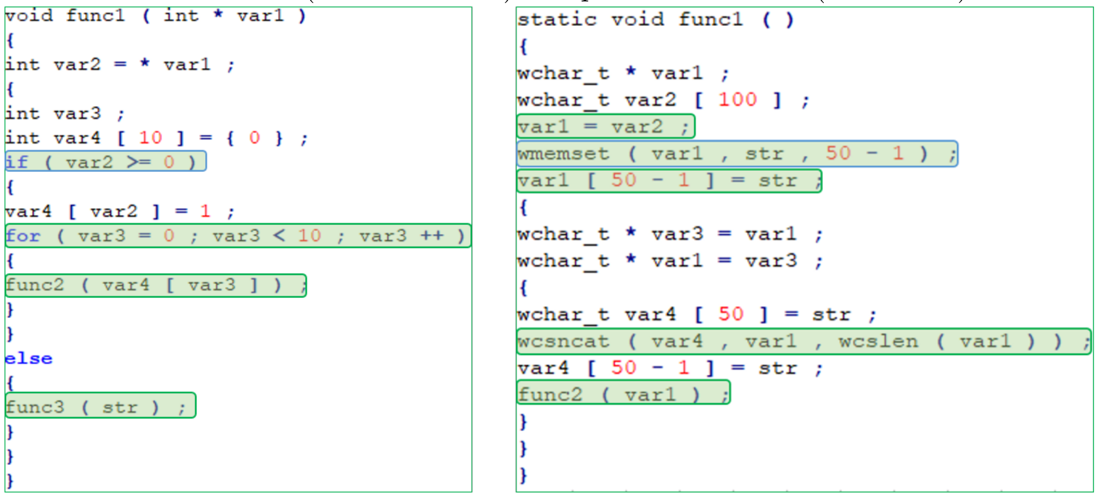
```
void func1 ( list < wchar_t * > var1 )
{
wchar_t * var2 = var1 . func2 ( ) ;
{
wchar_t var3 [ 50 ] = str ;
size_t var4 , var5 ;
var5 = wcslen ( var2 ) ;
for ( var4 = 0 ; var4 < var5 ; var4 ++ )
{
var3 [ var4 ] = var2 [ var4 ] ;
}
var3 [ 50 - 1 ] = str ;
func3 ( var2 ) ;
}
}
```

Figure 6.6: The true and predicted labels from the model are shown in the first row. The source code function and selected code statements highlighted relevant to vulnerabilities are shown with colours (see text).

Investigation for the case of the misclassification of non-vulnerable functions. In this section, we investigate the case when some non-vulnerable functions are predicted as vulnerable functions (i.e., false positives). Consider the examples depicted in Fig. 6.7. These functions appear as non-vulnerable in the ground truth. However, the model predicted them as vulnerable. In Fig. 6.7, the selected code statements (i.e., using our S2CVH2-mulB method) are shown with colours. For the left-hand function shown in Fig. 6.7, the selected code statements “if (var2 >= 0)” can in some cases (e.g., attempting to write to an index of the array that is above the upper bound) lead to a potential vulnerability. For the right-hand function shown in Fig. 6.7, the selected code statement “wmemset (var1 , str , 50 - 1) ;” will be a vulnerable code statement if we change “50 - 1” into “100 - 1”, because in this case we would initialise the source buffer as a buffer that is larger than the buffer used in the sink (i.e., “wcsncat (var4 , var1 , wcslen (var1)) ;”.

These are some typical examples for the case when non-vulnerable functions are predicted as vulnerable functions. The main reasons are likely due to: i) the key contributed code statements for marking the label of a function can be a potential vulnerability in some specific cases (e.g., depending on behaviour in the calling functions), or ii) the key contributed code statements for marking the label of a function can be a potential vulnerability if we have a minor code change applied to them.

True label: 0 (non-vulnerable) and predicted label: 1 (vulnerable)



```
void func1 ( int * var1 )
{
int var2 = * var1 ;
{
int var3 ;
int var4 [ 10 ] = { 0 } ;
if ( var2 >= 0 )
{
var4 [ var2 ] = 1 ;
for ( var3 = 0 ; var3 < 10 ; var3 ++ )
{
func2 ( var4 [ var3 ] ) ;
}
}
else
{
func3 ( str ) ;
}
}
}
```

```
static void func1 ( )
{
wchar_t * var1 ;
wchar_t var2 [ 100 ] ;
var1 = var2 ;
wmemset ( var1 , str , 50 - 1 ) ;
var1 [ 50 - 1 ] = str ;
{
wchar_t * var3 = var1 ;
wchar_t * var1 = var3 ;
{
wchar_t var4 [ 50 ] = str ;
wcsncat ( var4 , var1 , wcslen ( var1 ) ) ;
var4 [ 50 - 1 ] = str ;
func2 ( var1 ) ;
}
}
}
```

Figure 6.7: The true and predicted labels are show in the first row. The source code functions and selected code statements are shown with colours.

6.5 Closing remarks

We have proposed new methods to detect software vulnerabilities at the source code statement level, a more fine-grained level than at the function or program level, in both unsupervised and semi-supervised settings. In particular, our S2CVH1-mulN and S2CVH1-mulB methods aim to maximise the mutual information between selected code statements $\tilde{F}$ and the response variable label Y of a function or program. By maximising the mutual information, the selected statements are expected to strongly correlate with the existence of a vulnerability, hence potentially containing the core vulnerable statements. Compared with the S2CVH1-mulN and S2CVH1-mulB methods, our S2CVH2-mulB method has an additional regularisation term that minimises the mutual information between the selected code statements $\tilde{F}$ and all code statements F (i.e., by minimizing this mutual information, we aim to eliminate the irrelevant information of F to obtain the most compact representation $\tilde{F}$ that can predict well the label Y). The experimental results on real-world datasets have shown that our proposed methods can effectively and accurately detect software vulnerabilities at a fine-grained level.

Part III

Deep Sequence-to-sequence Models for Function Scope Identification in Binary Programs

Preface to Part III

In the previous part (i.e., Part II), we have presented our proposed approaches to address the problem of the fine-grained vulnerability detection which has a variety of applications in different areas such as software engineering and cybersecurity. Automated deep learning-based techniques for this problem have not yet been well studied. Our proposed approaches can work effectively and efficiently in both unsupervised and semi-supervised settings, hence providing important modelling tools as well as practical toolboxes for software developers and security experts. By detecting and highlighting such vulnerability-relevant statements, we can significantly speed up the process of isolating and detecting software vulnerabilities (SVs), thereby reducing the time and cost involved.

In Part III, Chapters 7 and 8, we summarise our proposed methods to deal with the third problem, existing in current software vulnerability detection (SVD) methods, which is relevant to the third research question (Q.3): "how to leverage the information from binaries (i.e., byte instructions) and assemblies (i.e., machine instructions) programs to *deal with all cases* (i.e., the function start identification, function end identification, function boundary identification and function scope identification problems) *of the function identification problem*, especially the function scope identification problem, the toughest and most essential problem".

Chapter 7

Code Action Network for Binary Function Scope Identification

Function identification is a preliminary step in binary analysis for many applications from malware detection to common vulnerability detection and binary instrumentation, to name a few. In this chapter, we propose the Code Action Network (CAN), whose key idea is to encode the task of function scope identification to a sequence of three action states: NI (next inclusion), NE (next exclusion) and FE (function end) in order to efficiently and effectively tackle function scope identification, the hardest and most crucial task in function identification. A bidirectional recurrent neural network is trained to match binary programs with their sequence of action states. To work out function scopes in a binary, this binary is first fed to a trained CAN to output its sequence of action states, which can be further decoded to discover the function scope in the binary.

7.1 Motivations

In computer security, we often encounter situations where source code is not available or impossible to access and only binaries are accessible. In these situations, binary analysis is an essential tool enabling many applications such as malware detection, common vulnerability detection [Perkins et al., 2009]. Function identification is usually the first step in many binary analysis methods. This aims to specify function scopes in a binary and is a building block to a diverse range of application domains including binary instrumentation [Laurenzano et al., 2010], vulnerability research [Pewny et al., 2015], and binary protection structures with Control-Flow Integrity. In both binary analysis and function identification, tackling the loss of high-level semantic structures in binaries which results from compilers during the process of compilation is likely the most challenging problem.

There have been many effective methods for dealing with the function identification problem from heuristic solutions (statistical methods for binary analysis) to complicated approaches employing machine learning or deep learning techniques. In an early work, Kruegel et al. [2004] through his research which leveraged statistical methods with control flow graphs concluded that the task of function start identification can be trivially solved for regular binaries. However, later research in [Zhang and Sekar, 2013] argued that this task is non-trivial and complex in some specific cases wherein it is too challenging for heuristics-based methods to discover all function boundaries. Other influential works and tools that rely on signature database and structural graphs include IDA Pro, Dyninst, (Binary Analysis Platform) BAP, and Nucleus [Andriesse et al., 2017]. Andriesse et al. [2017] has recently proposed a new signature-less approach to function detection for stripped binaries named Nucleus which is based on structural Control Flow Graph analysis. More specifically, Nucleus identifies functions in the intraprocedural control flow graph (ICFG) by analyzing the control flow between basic blocks, based on the observation that intraprocedural control flow tends to use different types and patterns of control flow instructions than inter-procedural control flow.

Inspired from the idea of a Turing machine, we imagine a memory tape consisting of many cells on which machine instructions of a binary are stored. The head is first pointed to the first machine instruction located in the first cell. Each machine instruction is assigned to an action state in the action state set {NI, NE, FE} depending on its nature. After reading the current machine instruction and assigning the corresponding action state to it, the head is moved to the next cell and this procedure is halted as we reach the last cell in the tape (see Section 7.4.1). Eventually, the sequence of machine instructions in a given binary is translated to the corresponding sequence of action states. Based on this incentive, in this chapter, we propose a novel method named the Code Action Network (CAN) whose underlying idea is to equivalently transform the task of function scope identification to learning a sequence of action states. A bidirectional Recurrent Neural Network is trained to match binary programs with their corresponding sequences of action states. To predict function scopes in any binary, the binary is first fed to a trained CAN to output its corresponding sequence of action states on which we can then work out function scopes in the binary. The proposed CAN can tackle binaries for which there exist external gaps between functions and internal gaps inside functions wherein each internal gap in a function does not contain instructions from other functions. By default, our CAN named as CAN-B operates at the byte level and can cope with all binaries that satisfy the aforementioned condition. However, for the binaries that can be further disassembled into machine instructions, another variant named as CAN-M is able to operate at the machine instruction level. CAN-M can efficiently exploit the semantic relationship among bytes in an instruction and instructions in a function as well as requiring much shorter sequence length compared with the Bidirectional RNN in [Shin et al., 2015] which also works at the byte level. In addition, our proposed CAN-B and CAN-M can directly address the function scope identification task, hence inherently offering the solution for other simpler tasks including the function start/end/boundary identifications.

We undertake extensive experiments to compare our proposed CAN-B and CAN-M with state-of-the-art methods including IDA, Bidirectional RNN, ByteWeight no-RFCR, and ByteWeight on the dataset used in [Shin et al., 2015, Bao et al., 2014]. The experimental results show that our proposed CAN-B and CAN-M outperform the baselines on function start, function end and function boundary identification tasks as well as achieving very good performance on function scope identification and also surpass the Nucleus [Andriesse et al., 2017] on this task. Our proposed methods slightly outperform the Bidirectional RNN proposed in [Shin et al., 2015] on the function start and end identification tasks, but significantly surpass this method on the function boundary identification task – the more important task. This demonstrates the capacity of our methods in efficiently utilising the contextual relationship carried in consecutive machine instructions or bytes to properly match the function start and end entries for this task. As expected, our CAN-M obtains the best predictive performances on most experiments and is much faster than the Bidirectional RNN proposed in [Shin et al., 2015]. Particularly, CAN-M takes about 1 hour for training with 20,000 iterations which is nearly 4 times faster than the Bidirectional RNN proposed in [Shin et al., 2015] using the same number of iterations for training and the same number of bytes for handling input. This is due to the fact that CAN-M operates at the machine instruction level, while the Bidirectional RNN proposed in [Shin et al., 2015] operates at the byte level.

We also do error analysis to qualitatively compare our CAN-M and CAN-B with the baselines. We observe that there are a variety of instruction styles for the function start and function end (e.g., in the experimental dataset, there are a thousand different function start styles and function end styles). In their error analyses, Shin et al. [2015] and Bao et al. [2014] mentioned that for functions which encompass several function start styles or function end styles, their proposed methods tend to make mistakes in predicting the function start or end bytes with many false positives and negatives. However, it is not the case for our proposed methods, since we further observe that for the functions which contain more than one function start style or function end style which account for 98.38% and 28% of the testing set respectively our proposed

CAN-M has 0.24% and 1.09% false positive rates respectively.

7.2 Related work for function identification

Machine learning has been applied to binary analysis and function identification in particular. The seminal work of [Rosenblum et al., 2008] modeled function start identification as a Conditional Random Field (CRF) in which binary offsets and a number of selected patterns appear in the CRF. Since the inference on a CRF is very expensive, though feature selection and approximate inference were adopted to speed up this model, its computational complexity is still very high. ByteWeight [Bao et al., 2014] is another successful machine learning based method for function identification aiming to learn signatures for function starts using a weighted prefix tree, and recognises function starts by matching binary fragments with the signatures. Each node in the tree corresponds to either a byte or an instruction, with the path from the root node to any given node representing a possible sequence of bytes or instructions. Although ByteWeight significantly outperformed disassembler approaches such as IDA Pro, Dyninst, and Binary Analysis Platform (BAP), it is not scalable enough for even medium-sized datasets [Shin et al., 2015].

Deep learning has undergone a renaissance in the past few years, achieving breakthrough results in multiple application domains such as visual object recognition [Krizhevsky et al., 2012], language modelling [Sutskever et al., 2014a], and software vulnerability detection [Le et al., 2019b, Nguyen et al., 2020a, 2019]. The study in [Shin et al., 2015] is the first work which applied a deep learning technique for the function identification problem. In particular, a bidirectional recurrent neural network (bidirectional RNN) was used to identify whether a byte is a start point (or end point) of a function or not. This method was proven to outperform ByteWeight [Bao et al., 2014] while requiring much less training time. However, to address the boundary identification problem with [Shin et al., 2015], a simple heuristic to pair adjacent function starts and function ends was used (see Section 5.3 in that paper). Consequently, this approach is not able to efficiently utilise the context information of consecutive bytes and machine instructions in a function and the pairing procedure might lead to inconsistency since the networks for function start and end were trained independently. Furthermore, this method cannot address the function scope identification problem, the hardest and most essential sub problem in function identification, wherein the scope (i.e., the addresses of all machine instructions in a function) of each function must be specified.

7.3 Function identification

This section discusses the function identification problem. We begin with definitions of the sub problems in the function identification problem, followed by an example of source code in the C language and its binaries compiled with the optimisation level O1 using gcc on the Linux platform for the x86-64 architecture.

7.3.1 Problem definitions

Given a binary program P, our task is to identify the necessary information (e.g., function starts, function ends) in its n functions $\{f_1, ..., f_n\}$ which is initially unknown. Depending on the nature of information we need from $\{f_1, ..., f_n\}$, we can categorise the task of function identification into the following such problems.

Function start/end/boundary identification.

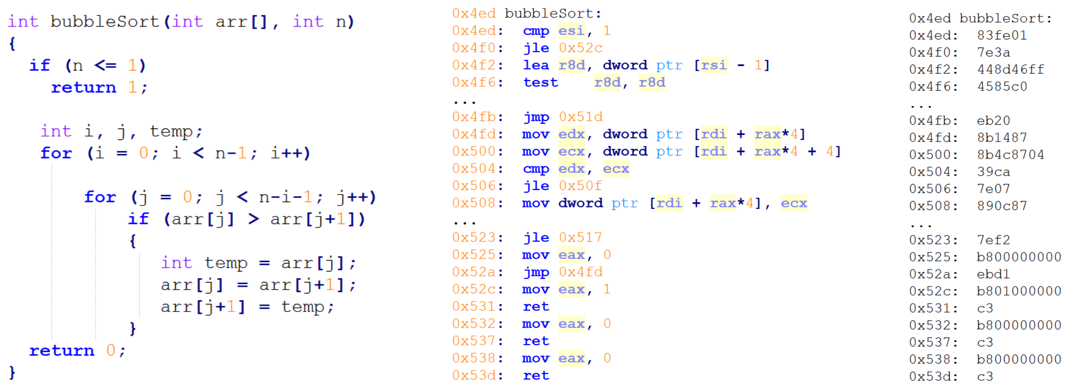
In the first problem, we need to specify the set S $= \{s_1, ..., s_n\}$ which contains the start instruction byte for each of the corresponding functions in $\{f_1, ..., f_n\}$. If a function (e.g. f_i) has multiple start points, s_i will be the first start instruction byte for f_i. In the second problem, we need to identify the set E $= \{e_1, ..., e_n\}$ which contains the end instruction byte for each of the corresponding functions in $\{f_1, ..., f_n\}$. If a function (e.g. f_i) has multiple exit points, e_i will be the last end instruction byte for f_i. In the last problem, we have to point out the set of (start, end) pairs SE $= \{(s_1, e_1), ..., (s_n, e_n)\}$ which contains the pairs of the function start and the function end for each of the corresponding functions in $\{f_1, ..., f_n\}$.

Function scope identification.

This is the hardest problem in the function identification task. In this problem, we need to find out the set $\{(f_{1,s_1}, ..., f_{1,e_1}), ..., (f_{n,s_n}, ..., f_{n,e_n})\}$ which specifies the instruction bytes in each function $f_1, ..., f_n$ in the given binary program P. Here we note that because functions may be not contiguous, the instruction bytes $(f_{i,s_i}, ..., f_{i,e_i})$ may also be not contiguous. It is apparent that the solution of this problem covers the three aforementioned problems. Since our proposed CAN addresses this problem, it inherently offers solutions for the other problems.

7.3.2 Running example

In Fig. 7.1, we show an example of a short source code fragment for a function in the C programming language, the corresponding assembly code in the machine instruction and corresponding hexadecimal mode of the binary code respectively which was compiled using gcc with the optimisation level O1 for the x86-64 architecture on the Linux platform. We further observe that in real binary code, the patterns for the entry point vary over a wide range and can start with *push, mov, movsx, inc, cmp,* or, *and,* etc. In the example, the assembly code corresponding with the optimisation level O1 on Linux has three *ret* statements. Furthermore, in real binary code, the ending point of a function can vary in pattern beside the *ret* pattern. These make the task of function identification very challenging. For the challenges of the function scope identification task, we refer the readers to [Bao et al., 2014, Shin et al., 2015] and the discussions therein.



```c
int bubbleSort(int arr[], int n)
{
  if (n <= 1)
    return 1;

   int i, j, temp;
   for (i = 0; i < n-1; i++)

       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
           {
              int temp = arr[j];
              arr[j] = arr[j+1];
              arr[j+1] = temp;
           }
  return 0;
}
```

```
0x4ed bubbleSort:
0x4ed:  cmp esi, 1
0x4f0:  jle 0x52c
0x4f2:  lea r8d, dword ptr [rsi - 1]
0x4f6:  test    r8d, r8d
...
0x4fb:  jmp 0x51d
0x4fd:  mov edx, dword ptr [rdi + rax*4]
0x500:  mov ecx, dword ptr [rdi + rax*4 + 4]
0x504:  cmp edx, ecx
0x506:  jle 0x50f
0x508:  mov dword ptr [rdi + rax*4], ecx
...
0x523:  jle 0x517
0x525:  mov eax, 0
0x52a:  jmp 0x4fd
0x52c:  mov eax, 1
0x531:  ret
0x532:  mov eax, 0
0x537:  ret
0x538:  mov eax, 0
0x53d:  ret
```

```
0x4ed bubbleSort:
0x4ed:  83fe01
0x4f0:  7e3a
0x4f2:  448d46ff
0x4f6:  4585c0
...
0x4fb:  eb20
0x4fd:  8b1487
0x500:  8b4c8704
0x504:  39ca
0x506:  7e07
0x508:  890c87
...
0x523:  7ef2
0x525:  b800000000
0x52a:  ebd1
0x52c:  b801000000
0x531:  c3
0x532:  b800000000
0x537:  c3
0x538:  b800000000
0x53d:  c3
```

Figure 7.1: Example source code of a function in C language programming (left), the corresponding assembly code (middle) with some parts omitted for brevity and the corresponding hexadecimal mode of the binary code (right).

7.4 Code Action Network for function identification

7.4.1 Key idea

In what follows, we present the key idea of our CAN. In a binary, there are external gaps between functions as well as internal gaps inside a non-contiguous function. The external gaps might contain data, jump tables or padding-instruction bytes which do not belong to any function (e.g., additional instructions generated by a compiler such as *nop* and *int3*). The internal gaps in general might contain data, jump tables or instructions from other functions (e.g., nested functions). We further assume that the internal gaps do not contain any instruction from other functions. It means that if there exist functions nested in a function, our CAN ignores these internal functions. However, we believe that the nested functions are extremely rare in real-world binaries. For example, in the experimental dataset, we observe that there are only 506 nested functions over the total of 757,125 functions (i.e., the occurrence rate is 0.067%).

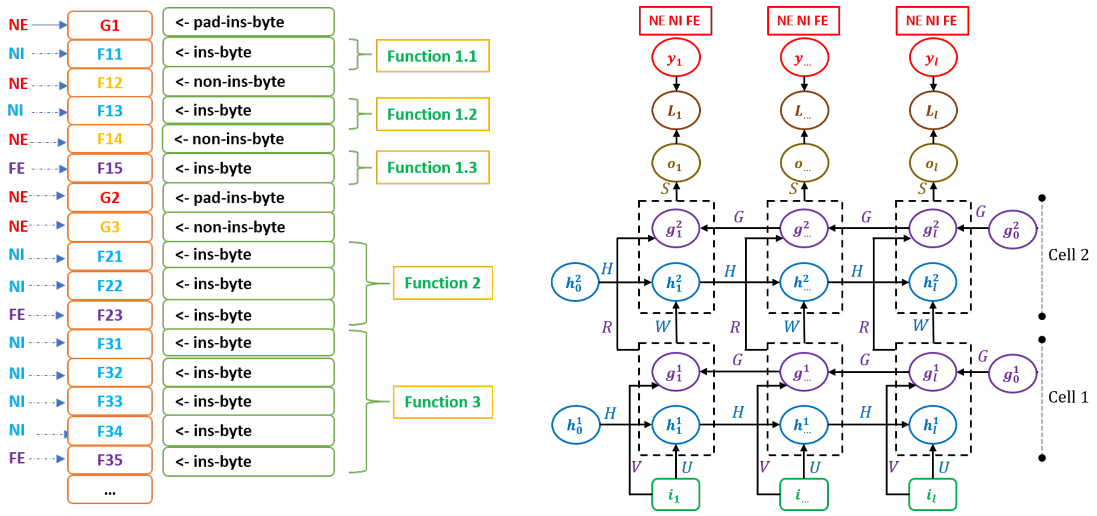


Figure 7.2: (The left-hand figure) Key idea and architecture of the Code Action Network. Assume that we have a sequence of instruction bytes in three functions where the functions may not be contiguous and there exist gaps between the functions. The Code Action Network transforms this sequence of instruction bytes to those of action states (i.e., NI, NE and FE). (The right-hand figure) The architecture of the Code Action Network. Each output value takes one of three action states NI, NE or FE. The Code Action Network will learn to map the input sequences of items $(\mathbf{i}_1, \mathbf{i}_2, ..., \mathbf{i}_l)$ to the target output sequence $(\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_l)$ with the loss L_i at each time step t. The h represents for the forward-propagated hidden state (toward the right) while the g stands for the backward-propagated hidden state (toward the left). At each time step t, the predicted output $\mathbf{o}_t$ can benefit from the relevant information of the past from its h and the future from its g.

The key idea of CAN is to encode the task of function scope identification to a sequence of three action states NI (next inclusion), NE (next exclusion) and FE (function end). With the aforementioned assumption, the binaries of interest consist of several functions and the functions in a binary do not intermingle, that is, each function only contains its machine instructions, data, or jump-tables and do not contain any machine instruction of other functions. Each function can be therefore viewed as a collection of bytes where each byte is from a machine instruction of this function (i.e., instruction byte) or data/jump-tables inside this function (i.e., non-instruction byte). To clarify how to proceed over a binary function given a sequence of action states, let us imagine this binary program including many instruction and non-instruction bytes as a tape of many cells wherein each cell contains a instruction or non-instruction byte and a pointer firstly points to the first cell in the tape. The action state NI includes the current instruction or non-instruction byte in the current cell to the current function and moves the pointer to the next cell (i.e., the next instruction or non-instruction byte). The action state NE excludes the current instruction or non-instruction byte in the current cell from the current function and moves the pointer to the next cell. The action state FE counts the current instruction or non-instruction byte in the current cell, ends the current function, starts reading a new function, and moves the pointer to the next cell.

To further explain how to transform a binary program to a sequence of action states, we consider an example binary code depicted in Fig. 7.2 (the left-hand figure). Assume that we have a sequence of instruction and non-instruction bytes, which belong to Function 1, Function 2 and Function 3 respectively where the functions may be not contiguous and there exist gaps between the functions (e.g., the gap between Function 1 and Function 2 includes the padding-instruction byte (pad-ins-byte) G2 and the non-instruction (non-ins-byte) byte G3). The pointer of CAN firstly points to G1, labels this padding-instruction byte (pad-ins-byte) as NE since G1 does not belong to any function, and moves to the instruction byte F11. The instruction byte F11 is labelled as NI since it belongs to the function Function 1. The pointer then moves to the non-instruction byte F12 which can come from a jump-table or data and labels it as NE because F12 does not belong to any function. After that, the pointer moves to the instruction byte F13 and the non-instruction byte F14 subsequently. F13 and F14 are then labelled as NI and NE respectively since F13 belong to the function Function 1 while F14 does not belong to any function, and the pointer moves to the instruction byte F15 and labels it as FE since it is the end of the function Function 1 and we need to start reading the new function (i.e., the function Function 2). The pointer subsequently moves to the instruction byte G2 and the non-instruction G3 which can come from a jump-table or data and labels them as NE since they do not belong to any function. The pointer then traverses across the instruction bytes F21, F22 and F23, and labels them as NI, NI and FE respectively. The pointer now starts reading the new function (i.e., the function Function 3). This process is repeated until the pointer reaches the last instruction or non-instruction byte and we eventually identify all functions.

It is worth noting that if binaries can be disassembled and a function in these binaries can be thus viewed as a collection of instructions and non-instructions, we can perform the aforementioned idea at the machine instruction level wherein each cell in the tape represents an instruction or non-instruction of a binary. The advantages of performing the task of function identification at the machine instruction level include: i) the sequence length of the bidirectional RNN is significantly reduced and ii) the semantic relationship among bytes in a machine instruction and machine instructions can be further exploited. As a consequence, the gradient exploding and vanishing which often occur with long RNNs can be avoided and the model is easier to train while obtaining higher predictive performance and much shorter training times as shown in our experiments.

7.4.2 Preprocess input statement

Byte level. To process data for the byte level, we simply take the raw bytes in the text segment of the given binary and input them to CAN-B.

Machine instruction level. To process data for the machine instruction level, we first use Capstone[1] to disassemble the binaries and preprocess the machine instructions obtained from the text segment of a binary before inputting them to CAN-M. This preprocessing step aims to work out fixed length inputs from machine instructions. For each machine instruction, we employ Capstone to detect entire machine instructions, then eliminate redundant prefixes to obtain core parts that contain the opcode and other significant information.

We note that the core part obtained may have various sizes of 4 to 8 bytes or even more for the x86 and x86-64 architectures. For the x86 architecture, we keep the first 4 bytes from the left for machine instructions which are longer than 4 bytes and (2) padding 0 to the right of the machine instructions which have fewer than 4 bytes. For example, the machine instruction "move edx, dword ptr [ebp - 0x4dc]" has the corresponding value "8b9524fbffff" in hex format which is 6 bytes long, we then remove the last 2 bytes and keep the first 4 bytes to get the numerical value "8b9524fb". For the machine instruction "mov dword ptr [ebp - 0xc], eax" which contains 3 bytes "8945f4" in the hex format, we then pad with 0 to fill in the fourth byte and gain the numerical value "8945f400". Likewise, for the x86-64 architecture, we keep the first 8 bytes from the left for machine instructions which are longer than 8 bytes and (2) padding 0 to the right of the machine instructions which have fewer than 8 bytes. For example, the machine instruction "mov qword ptr [rbp - 0xe0], 0" has the corresponding value "48c78520ffffff00000000" in hex format which is 11 bytes long, we then remove the last 3 bytes and keep the first 8 bytes to get the numerical value "48c78520ffffff00". For the machine instruction "mov rax, qword ptr [rsp + 0x20]" which contains 5 bytes "488b442420" in the hex format, we then pad with 0 to fill in the sixth, seventh and eighth bytes and gain the numerical value "488b442420000000". In addition, the reason for choosing the first 4 or 8 bytes as aforementioned is that more than 75% of instructions in the x86 or x86-64 architectures have their sizes less than or equal 4 or 8 bytes respectively and the first 4 or 8 bytes in the core parts of the x86 or x86-64 instructions contain the most crucial information (e.g., the opcode or other significant information). Moreover, when a binary can be disassembled, the relevant software like Capstone can further identify non-instruction items and we pad these items with 0 to ensure that their sizes are a multiple of 4 or 8 according to the x86 and x86-64 architectures. Certainly, a naive possible solution for both

[1] www.capstone-engine.org

x86 and x86-64 architectures is to preserve the machine instructions with the maximal length in each architecture and pad the shorter machine instructions. However, this solution leads to extremely high and imbalanced inputs, hence making the network hard to train and tame.

7.4.3 Code Action Network architecture

7.4.3.1 Training procedure

The Code Action Network (CAN) is a multicell bidirectional RNN whose architecture is depicted in Fig. 7.2 (the right-hand figure) where we assume the number of cells over the input is 2. Our CAN takes a binary program $\mathbf{B} = (\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_l)$ including l instructions (and non-instructions) for CAN-M or instruction bytes (and non-instruction bytes) for CAN-B and learns to output the corresponding sequence of action states $\mathbf{Y} = (y_1, y_2, ..., y_l)$ where each y_k takes one of three action states *NI* (i.e., $y_k = 1$), *NE* (i.e., $y_k = 2$) or *FE* (i.e., $y_k = 3$). The computational process of CAN is as follows:

$$\mathbf{h}_k^1 = \tanh(H^\top \mathbf{h}_{k-1}^1 + U^\top \mathbf{i}_k);\ \mathbf{g}_k^1 = \tanh(G^\top \mathbf{g}_{k+1}^1 + V^\top \mathbf{i}_k);\ \mathbf{h}_k^2 = \tanh(H^\top \mathbf{h}_{k-1}^2 + W^\top [\begin{matrix}\mathbf{h}_k^1 \\ \mathbf{g}_k^1\end{matrix}])$$

$$\mathbf{g}_k^2 = \tanh(G^\top \mathbf{g}_{k+1}^2 + R^\top [\begin{matrix}\mathbf{h}_k^1 \\ \mathbf{g}_k^1\end{matrix}]);\ \mathbf{o}_k = S^\top [\begin{matrix}\mathbf{h}_k^2 \\ \mathbf{g}_k^2\end{matrix}];\ \mathbf{p}_k = \text{softmax}(\mathbf{o}_k)$$

where $k = 1, ...l$, $\mathbf{h}_0^1$, $\mathbf{h}_0^2$, $\mathbf{g}_{l+1}^1 = \mathbf{g}_0^1$, $\mathbf{g}_{l+1}^2 = \mathbf{g}_0^2$ are initial hidden states and $\theta = (U, V, W, H, G, R, S)$ is the model. We further note that $\mathbf{p}_k$, $k = 1, \ldots, l$ is a discrete distribution over the three labels NI, NE and FE.

To find the best model θ^*, we need to solve the following optimisation problem:

$$\max_\theta \sum_{(\mathbf{B},\mathbf{Y}) \in \mathcal{D}} \log p\left(\mathbf{Y} \mid \mathbf{B}\right) \tag{7.1}$$

where $\mathcal{D}$ is the training set including pairs $(\mathbf{B},\mathbf{Y})$ of the binaries and their corresponding sequence of action states.

Because o_k is a function (lossy summary) of $\mathbf{i}_{1:l}$, we further derive $\log p(\mathbf{Y} \mid \mathbf{B})$ as:

$$\log p\left(\mathbf{Y} \mid \mathbf{B}\right) = \sum_{k=1}^{l} \log p\left(y_k \mid y_{1:k-1}, \mathbf{i}_{1:l}\right) = \sum_{k=1}^{l} \log p\left(y_k \mid \mathbf{o}_k\right)$$

Substituting back to the optimisation problem in Eq. (7.1), we arrive the following optimisation problem:

$$\max_{\theta} \sum_{(\mathbf{B},\mathbf{Y})\in\mathcal{D}} \sum_{k=1}^{l} \log p\left(y_k \mid \mathbf{o}_k\right)$$

where $p(y_k \mid \mathbf{o}_k)$ is the $\mathbf{y}_k$- th element of the discrete distribution $\mathbf{p}_k$ or in other words, we have $p(y_k \mid \mathbf{o}_k) = \mathbf{p}_{k,y_k}$.

7.4.3.2 Testing procedure

In what follows, we present how to work out the function scopes in a binary using a trained CAN. The machine instructions (and non-instructions) for CAN-M or instruction (and non-instruction) bytes for CAN-B in the testing binary are fed to the trained model to work out the predicted sequence of action states. This predicted sequence of action states is then decoded to the function scopes inside the binary. As shown in Fig. 7.3, the binary in Fig. 7.2 when inputted to the trained CAN outputs the sequence of action states NE, NI, ..., NI and FE, and is later decoded to the scopes of the functions Function 1, Function 2 and Function 3.

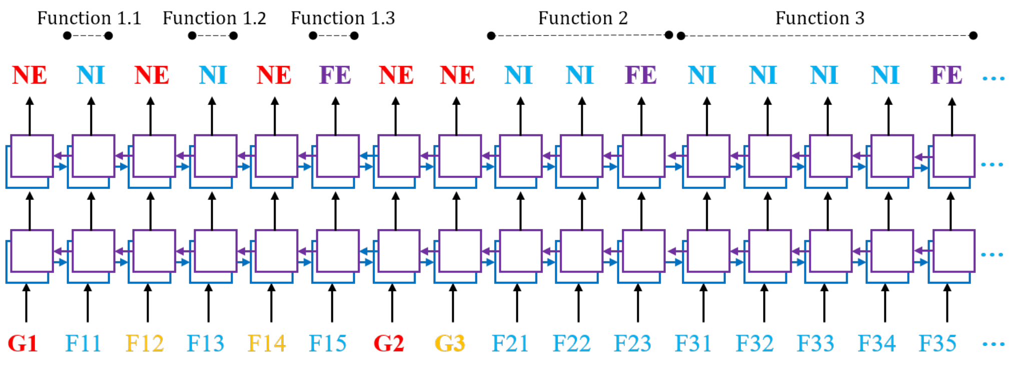

Function 1.1
Function 1.2
Function 1.3
Function 2
Function 3
NE NI NE NI NE FE NE NE NI NI FE NI NI NI NI FE ...
G1 F11 F12 F13 F14 F15 G2 G3 F21 F22 F23 F31 F32 F33 F34 F35 ...

Figure 7.3: Testing procedure of our Code Action Network. The sequence of machine instructions (and non-instructions) or instruction bytes (and non-instruction) bytes in a binary program is fed to the trained Code Action Network to work out the sequence of action states. Subsequently, the sequence of action states is decoded to the set of functions in this binary.

7.5 Implementation and results

In this section, firstly, we present the experimental results of our proposed Code Action Network for the machine instruction level (CAN-M) and the byte level (CAN-B) compared with other baselines including IDA, ByteWeight (BW) no-RFCR, ByteWeight (BW) [Bao et al., 2014], the Bidirectional RNN [Shin et al., 2015], and Nucleus [Andriesse et al., 2017]. Secondly, we perform error analysis to qualitatively investigate our proposed methods. We also investigate the model behaviour of our CAN-M with various RNN cells and with different size for hidden states.

7.5.1 Experimental setup

7.5.1.1 Experimental dataset

We used the dataset from [Bao et al., 2014, Shin et al., 2015], which consists of 2,200 different binaries including 2,064 binaries obtained from the findutils, binutils, and coreutils packages and compiled with both icc and gcc for Linux at four optimisation levels O0, O1, O2 and O3. The remaining binaries for Windows are from various well-known open-source projects which were compiled with Microsoft Visual Studio for the x86 (32 bit) and the x86-64 (64 bit) architectures at four optimisation levels Od, O1, O2 and Ox.

7.5.1.2 Experimental setting

We divided the binaries into three random parts; the first part contains 80% of the binaries used for training, the second part contains 10% of the binaries used for testing, and the third part contains 10% of the binaries for validation. For CAN-M, we used a sequence of 250 hidden states for the x86 architecture and 125 hidden states for the x86-64 architecture where the size of hidden states is 256. For CAN-B, akin to the Bidirectional RNN in [Shin et al., 2015], we used a sequence length of 1,000 hidden states for the x86 and x86-64 architectures. We employed the Adam optimiser with the default learning rate 0.001 and the mini-batch size of 32. In addition, we applied gradient clipping regularisation [Pascanu et al., 2013] to prevent the over-fitting problem when training the model. We implemented the Code Action Networks in Python using Tensorflow [Abadi et al., 2016], an open-source software library for Machine Intelligence developed by the Google Brain Team.

7.5.2 Experimental results

7.5.2.1 Code Action Network versus baselines

We compared our CAN-M and CAN-B using the long short-term memory (LSTM) cell and the hidden size of 256 with IDA, the Bidirectional RNN, ByteWeight (BW) no-RFCR, and ByteWeight (BW) in the task of function start, function end, function boundary and function scope identification. For the well-known tool IDA as well as the Bidirectional RNN, ByteWeight no-RFCR, and ByteWeight methods, we reported the experimental results presented in [Bao et al., 2014] and [Shin et al., 2015]. Obviously, the task of function scope identification wherein we need to specify addresses of machine instructions in each function is harder than that of function boundary identification. To compute the function scope results, given a predicted function by CAN variants, we considered their start and end instructions for CAN-M and start and end bytes for CAN-B, and then evaluated measures (e.g., Precision, Recall and F1-score) based on this pair. In addition, in the function scope identification task, a pair is counted as a correct pair if all predicted bytes or machine instructions accompanied with this pair forms a function that exactly matches to a valid function in the ground truth. In contrast, in the function boundary identification task, we only require the start and end positions of this pair to be correct.

The experimental results in Table 7.1 show that our proposed CAN-M and CAN-B achieved better predictive performances (i.e., Recall, Precision and F1-score) compared with the baselines in most cases (PE x86, PE x86-64, ELF x86 and ELF x86-64). For the function boundary identification task, our CAN-B and CAN-M significantly outperformed the baselines in all measures, especially for CAN-M. Interestingly, the predictive performance of our proposed methods on the harder task of function scope identification was higher or comparable with that of the baselines on the easier task of function boundary identification. In comparison with the Bidirectional RNN proposed in [Shin et al., 2015], our proposed methods slightly outperform it on the function start and function end identification tasks, but significantly surpass this method on the function boundary identification task - the more important task. This result demonstrates the capacity of our methods in efficiently utilising the contextual relationship carried in consecutive machine instructions or bytes to properly match the function start and end entries for this task. Regarding the amount of time taken for training, our CAN-M took approximately 3,490 seconds for training in 20,000 iterations, while our CAN-B and the Bidirectional RNN using the same number of iterations with the sequence length 1,000 took about 12,030 seconds (i.e., roughly four times slower). This is due to a much smaller sequence length of CAN-M compared with CAN-B and the Bidirectional RNN.

7.5.2.2 Code Action Network versus Bidirectional RNN, ByteWeight and Nucleus

We also compared the average predictive performance for case by case including the function start, function bound and function scope identifications of our CANM and CAN-B using the hidden size of 256 and LSTM cell with the Bidirectional RNN, ByteWeight, and Nucleus in both Linux and Windows platforms. For Nucleus [Andriesse et al., 2017], we reported the experimental results reported in that paper. The experimental results in Table 7.2 indicate that our CAN-M and CAN-B again outperformed the baselines, while CAN-M obtained the highest predictive performances in all measures (Recall, Precision and F1-score).

7.5.2.3 Error analysis

For a qualitative assessment, we performed error analysis of our CAN-M and CAN-B for all cases including PEx86, PEx64, ELFx86 and ELFx64.

At the machine instruction level, we observed that there are 4,714, 4,464, 3,320 and 8,147 different types of machine instructions for function start while there are 1,926, 5,523, 9,082 and 11,421 different types of machine instructions for function end in the PEx86, PEx64, ELFx86 and ELFx64 datasets respectively. At byte level, we found that there are 91, 49, 41 and 53 different types of instruction bytes for function start while there are 166, 125, 133 and 126 different types of bytes for function end in the PEx86, PEx64, ELFx86 and ELFx64 datasets respectively. Obviously, these diverse ranges in the function start and function styles make the task of function identification really challenging. In all four cases (PEx86, PEx64, ELFx86 and ELFx64), the compilers in use often add padding between functions such as *nop* and *int3*.

We summarise some observations for the performance of our methods as follows:

- Shin et al. [2015] and Bao et al. [2014] commonly mentioned that for the functions that contain either several function start or function end styles inside, their models tend to confuse in determining the true start or end points, hence offering many false positives. This is due to a high level of ambiguity in the start or end entries for these functions. However, it is not the case for our proposed CAN-M and CAN-B. For example, at the machine instruction level with PE x86, we found that the functions which contain more than one function start style or function end style account for 98.38% and 28.00% of the testing set and when predicting these functions, our proposed CAN-M has 0.28% false negative rate and 0.24% false positive rate as well as 1.56% false negative rate and 1.09% false positive rate.

- Our proposed methods also share the same behavior as the method in [Shin et al., 2015] in predicting some first and last items in an input sequence, that is, the CAN-M and CAN-B sometimes offer false positives and negatives when predicting some first and last instructions or bytes in an input sequence. More specifically, if an input sequence involves several functions, the start of the first function and the end of the last function are more likely to be predicted incorrectly. This is possibly due to the scarcity of context before or after them. For example, at the machine instruction level with PE x86, we record that there is about 2.39% of input sequences which contain function ends at some first and last input items. When predicting these function end entries, our proposed CAN-M obtains 21.21% false positive rate and 27.27% false negative rate.

Task	Architectures	ELF x86			ELF x86-64			PE x86			PE x86-64		
	Methods	R	P	F1	R	P	F1	R	P	F1	R	P	F1
(f.s)	IDA	58.34%	70.97%	64.04%	55.50%	74.20%	63.50%	87.80%	94.67%	91.11%	93.34%	98.22%	95.72%
	BW no-RFCR	96.17%	98.36%	97.25%	97.57%	99.11%	98.33%	92.13%	96.75%	94.38%	96.22%	97.74%	96.97%
	BW	97.94%	98.41%	98.17%	**98.47%**	99.14%	98.80%	95.37%	93.78%	94.57%	97.98%	97.88%	97.93%
	Bidirectional RNN	99.06%	*99.56%*	99.31%	97.80%	98.80%	98.30%	98.46%	99.01%	98.73%	*99.09%*	*99.52%*	*99.30%*
	CAN-B	*99.23%*	99.41%	*99.32%*	*98.19%*	*99.05%*	*98.62%*	*98.95%*	*99.53%*	*99.24%*	**99.20%**	99.46%	**99.33%**
	CAN-M	**99.35%**	**99.61%**	**99.48%**	98.02%	**99.34%**	**98.68%**	**99.52%**	**99.67%**	**99.59%**	99.05%	***99.53%***	99.29%
(f.e)	Bidirectional RNN	97.87%	98.69%	98.28%	95.03%	97.45%	96.22%	98.35%	99.24%	98.79%	**99.20%**	99.28%	**99.24%**
	CAN-B	*99.16%*	*99.38%*	*99.27%*	**98.34%**	*99.20%*	**98.77%**	*98.82%*	*99.39%*	*99.10%*	*99.15%*	*99.30%*	*99.22%*
	CAN-M	**99.30%**	**99.56%**	**99.43%**	*97.97%*	**99.29%**	*98.63%*	**99.56%**	**99.71%**	**99.64%**	99.12%	**99.31%**	99.21%
(f.b)	IDA	56.53%	70.63%	62.80%	53.46%	72.84%	61.66%	87.10%	93.93%	90.39%	93.24%	98.11%	95.61%
	BW no-RFCR	90.58%	92.85%	91.70%	91.59%	93.17%	92.37%	90.48%	95.03%	92.70%	91.35%	92.87%	92.10%
	BW	92.29%	92.78%	92.53%	92.52%	93.22%	92.87%	93.91%	92.30%	93.10%	93.13%	93.04%	93.08%
	Bidirectional RNN	95.34%	97.75%	96.53%	89.91%	94.85%	92.31%	95.27%	97.53%	96.39%	97.33%	**98.43%**	97.88%
	CAN-B	*98.08%*	*98.29%*	*98.18%*	**96.45%**	*97.24%*	**96.84%**	*97.81%*	*98.36%*	*98.08%*	**97.89%**	98.27%	**98.08%**
	CAN-M	**98.43%**	**98.68%**	**98.55%**	*96.13%*	**97.34%**	*96.73%*	**98.99%**	**99.14%**	**99.06%**	*97.63%*	*98.39%*	*98.01%*
(f.sc)	CAN-B	*98.03%*	*98.25%*	*98.14%*	**96.28%**	*97.10%*	**96.69%**	*97.75%*	*98.31%*	*98.03%*	**97.83%**	*98.22%*	**98.02%**
	CAN-M	**98.40%**	**98.65%**	**98.52%**	*95.94%*	**97.21%**	*96.57%*	**98.97%**	**99.12%**	**99.05%**	*97.52%*	**98.28%**	*97.90%*

Table 7.1: Comparison of our Code Action Network and baselines (Best in **bold**, second best in underline). Noting that f.s, f.e, f.b and f.sc stand for func. start, func. end, func. boundary and func. scope respectively, while R, P and F1 represent Recall, Precision and F1-score respectively.

Tasks	Function Start			Function Bound			Function Scope		
Methods	Recall	Precision	F1	Recall	Precision	F1	Recall	Precision	F1
Nucleus	94%	96%	94.99%	88%	96%	91.83%	88%	96%	91.83%
ByteWeight	97.44%	97.30%	97.37%	92.96%	92.84%	92.90%	-	-	-
Bidirectional RNN	98.60%	99.22%	98.92%	94.46%	97.14%	95.78%	-	-	-
CAN-B	*98.89%*	*99.36%*	*99.12%*	*97.56%*	*98.04%*	*97.80%*	*97.47%*	*97.97%*	*97.72%*
CAN-M	**98.99%**	**99.54%**	**99.26%**	**97.80%**	**98.39%**	**98.09%**	**97.71%**	**98.32%**	**98.01%**

Table 7.2: Comparison with the baselines including the Bidirectional RNN, ByteWeight and Nucleus using average scores for all architectures (x86 and x86-64) for both Linux and Windows of our Code Action Network. The experimental results for Nucleus are from the original paper using the same dataset (best performance in **bold**, second best in underline).

7.5.2.4 Variation in RNN cells

In Table 7.3, we studied the performances of our CAN-M with various RNN cells including long short-term memory (CAN-M-LSTM) and gated recurrent unit (CAN-M-GRU) when the hidden size was set to 256. It can be observed that the model using LSTM achieved better performance in almost all measures including Recall, Precision and F1-score than that using GRU in most cases (PE x86, PE x86-64, ELF x86 and ELF x86-64).

7.5.2.5 Variation in the size of the hidden states

In Table 7.4, we studied the performances of our CAN-M using different hidden sizes varying in the range of {128, 256} while employing the LSTM cell. The experimental results show that CAN-M-LSTM-256 obtained higher performance in most measures including Recall, Precision and F1-score compared with CAN-M-LSTM-128 in all cases (PE x86, PE x86-64, ELF x86 and ELF x86-64), while CAN-M-LSTM-128 gained higher performance in Precision for function start and function end in three cases (PE x86, PE x86-64 and ELF x86).

Architectures	ELF x86			ELF x86-64		
Methods	Recall	Precision	F1-score	Recall	Precision	F1-score
CAN-M-LSTM *(func. start)*	**99.35%**	**99.61%**	**99.48%**	**98.02%**	**99.34%**	**98.68%**
CAN-M-LSTM *(func. end)*	**99.30%**	99.56%	**99.43%**	**97.97%**	**99.29%**	**98.63%**
CAN-M-LSTM *(func. bound)*	**98.43%**	**98.68%**	**98.55%**	**96.13%**	**97.34%**	**96.73%**
CAN-M-LSTM *(func. scope)*	**98.40%**	**98.65%**	**98.52%**	**95.94%**	**97.21%**	**96.57%**
CAN-M-GRU *(func. start)*	98.52%	99.45%	98.98%	96.91%	97.67%	97.29%
CAN-M-GRU *(func. end)*	98.67%	**99.60%**	99.13%	96.84%	97.59%	97.21%
CAN-M-GRU *(func. bound)*	97.18%	98.05%	97.61%	93.64%	94.28%	93.96%
CAN-M-GRU *(func. scope)*	97.09%	97.99%	97.54%	93.02%	93.72%	93.37%
Architectures	PE x86			PE x86-64		
Methods	Recall	Precision	F1-score	Recall	Precision	F1-score
CAN-M-LSTM *(func. start)*	**99.52%**	**99.67%**	**99.59%**	**99.05%**	**99.53%**	**99.29%**
CAN-M-LSTM *(func. end)*	**99.56%**	**99.71%**	**99.64%**	**99.12%**	**99.31%**	**99.21%**
CAN-M-LSTM *(func. bound)*	**98.99%**	**99.14%**	**99.06%**	**97.63%**	**98.39%**	**98.01%**
CAN-M-LSTM *(func. scope)*	**98.97%**	**99.12%**	**99.05%**	**97.52%**	**98.28%**	**97.90%**
CAN-M-GRU *(func. start)*	98.86%	99.32%	99.09%	97.63%	98.78%	98.20%
CAN-M-GRU *(func. end)*	98.97%	99.43%	99.20%	97.43%	98.58%	98.00%
CAN-M-GRU *(func. bound)*	97.54%	97.98%	97.76%	95.54%	96.59%	96.06%
CAN-M-GRU *(func. scope)*	97.46%	97.91%	97.68%	95.26%	96.38%	95.82%

Table 7.3: Comparison of variants of CAN using different types of RNN cells including LSTM and GRU.

Architectures	ELF x86			ELF x86-64		
Methods	Recall	Precision	F1-score	Recall	Precision	F1-score
CAN-M-LSTM-128 *(func. start)*	99.15%	**99.63%**	99.39%	97.90%	98.49%	98.19%
CAN-M-LSTM-128 *(func. end)*	99.12%	**99.60%**	99.36%	97.76%	98.34%	98.05%
CAN-M-LSTM-128 *(func. bound)*	98.23%	98.69%	98.46%	95.53%	96.05%	95.79%
CAN-M-LSTM-128 *(func. scope)*	98.19%	98.66%	98.42%	95.23%	95.78%	95.50%
CAN-M-LSTM-256 *(func. start)*	**99.35%**	99.61%	**99.48%**	**98.02%**	**99.34%**	**98.68%**
CAN-M-LSTM-256 *(func. end)*	**99.30%**	99.56%	**99.43%**	**97.97%**	**99.29%**	**98.63%**
CAN-M-LSTM-256 *(func. bound)*	**98.43%**	**98.68%**	**98.55%**	**96.13%**	**97.34%**	**96.73%**
CAN-M-LSTM-256 *(func. scope)*	**98.40%**	**98.65%**	**98.52%**	**95.94%**	**97.21%**	**96.57%**
Architectures	PE x86			PE x86-64		
Methods	Recall	Precision	F1-score	Recall	Precision	F1-score
CAN-M-LSTM-128 *(func. start)*	99.32%	**99.80%**	99.56%	98.44%	**99.68%**	99.06%
CAN-M-LSTM-128 *(func. end)*	99.38%	**99.87%**	99.62%	98.22%	**99.46%**	98.84%
CAN-M-LSTM-128 *(func. bound)*	98.67%	**99.14%**	98.90%	**97.82%**	98.18%	98.00%
CAN-M-LSTM-128 *(func. scope)*	98.65%	**99.12%**	98.88%	97.29%	98.11%	97.70%
CAN-M-LSTM-256 *(func. start)*	**99.52%**	99.67%	**99.59%**	**99.05%**	99.53%	**99.29%**
CAN-M-LSTM-256 *(func. end)*	**99.56%**	99.71%	**99.64%**	**99.12%**	99.31%	**99.21%**
CAN-M-LSTM-256 *(func. bound)*	**98.99%**	**99.14%**	**99.06%**	97.63%	**98.39%**	**98.01%**
CAN-M-LSTM-256 *(func. scope)*	**98.97%**	**99.12%**	**99.05%**	**97.52%**	**98.28%**	**97.90%**

Table 7.4: Comparison of variants of CAN using different hidden state size in the range of {128, 256}. The best results (%) are emphasised in **bold**.

7.6 Closing remarks

In this chapter, we have proposed the novel Code Action Network (CAN) for dealing with the function identification problem, a preliminary and significant step in binary analysis for many security applications such as malware detection, common vulnerability detection and binary instrumentation. Specifically, the CAN leverages the underlying idea of a multicell bidirectional recurrent neural network with the idea of encoding the task of function scope identification to a sequence of three action states NI (next inclusion), NE (next exclusion) and FE (function end) in order to tackle function scope identification, the hardest and most crucial task in function identification. The experimental results show that the CAN can achieve state-of-the-art performance in terms of efficiency and efficacy.

Chapter 8

Code Pointer Network for Binary Function Scope Identification

In this chapter, we propose the Code Pointer Network (CPN), which leverages the underlying idea of a pointer network to efficiently and effectively tackle function scope identification (i.e., the hardest and most crucial task in function identification). Our proposed CPN includes one encoder and one decoder. The encoder takes the sequence of all machine instructions in a binary, while the decoder reads out the function scopes in the given binary. In addition, our proposed CPN can directly address the function scope identification task, hence inherently offering the solutions for other simpler tasks including function start, function end and function boundary identification.

8.1 Motivations

In computer security, we often face the situation where source code is not available or impossible to access and only binaries are accessible. In these situations, binary analysis is an essential tool enabling many extensive applications such as malware detection [Caballero et al., 2010], common vulnerability detection [Perkins et al., 2009]. Function identification is usually the first step in many binary analysis methods. This aims to identify function scopes in a binary and can contribute to a wide variety of application domains including searching for vulnerabilities [Pewny et al., 2015], binary instrumentation [Laurenzano et al., 2010], binary protection structures with Control-Flow Integrity [Prakash et al., 2015], and binary software vulnerability detection [Le et al., 2019b, Nguyen et al., 2020a]. One of the most challenging problems in both binary analysis and function identification is how to tackle the scarcity of high-level semantic structures in binaries which might originate from compilers during the process of compilation.

There have been many effective approaches proposed for solving the function identification problem from simple heuristics solutions to more complicated methods employing machine learning or deep learning techniques. In an early work of the function identification problem, Kruegel et al. [2004] pointed out that the task of function start identification is trivially solved for regular binaries. However, later research of [Perkins et al., 2009] and [Zhang and Sekar, 2013] revealed that this task is non-trivial and complex in some specific cases where it is too difficult for heuristics-based methods to figure out all function boundaries. ByteWeight [Bao et al., 2014] is a machine learning based method for function identification. It learns signatures for function starts using a weighted prefix tree, and recognises function starts by matching binary fragments with the signatures. Each node in the tree corresponds to either a byte or an instruction, with the path from the root node to any given node representing a possible sequence of bytes or instructions. Although ByteWeight significantly outperformed disassembler approaches such as IDA Pro, Dyninst [Bernat and Miller, 2011], BAP [Brumley et al., 2011], and the CMU Binary Analysis Platform, it is not scalable enough for even medium-sized datasets. In particular, it took about 587 compute-hours for training a dataset of 2,064 binaries [Shin et al., 2015]. Recently, Andriesse et al. [2017] proposed a new solution for function identification, which is based on Control Flow Graph analysis. This method was comparable with ByteWeight in terms of predictive performance while requiring less computational time.

The study in [Shin et al., 2015] is the first work that applied a deep learning technique for the function identification problem. In particular, a bidirectional recurrent neural network was used to identify whether a byte is a start point (or an end point) of a function or not. This method was proven to outperform ByteWeight while requiring much less training time. However, this method cannot address the function scope identification problem, the toughest and most essential sub problem in function identification, wherein the scope (i.e., the indexes or addresses of all machine instructions in a function) of each function must be specified. The only way to fulfill this task using this method is to first pair corresponding function starts and function ends and make assumption that the scope inside each pair comprises a function. In addition, if both start points and end points are simultaneously necessary, two separate bidirectional RNNs must be trained independently and this certainly cannot exploit the semantic relationship among start points, end points, and other machine instructions in a function.

In this chapter, we propose a method named the Code Pointer Network (CPN) that employs the idea of a pointer network [Vinyals et al., 2015b] in the specific context of function identification. Our proposed CPN includes one encoder and one decoder. The encoder takes the sequence of all machine instructions in a binary while the decoder reads out function scopes in the given binary. In addition, unlike the work of [Shin et al., 2015], our proposed CPN can directly address the function scope identification task, hence it inherently offers the solutions for other simpler tasks including function start, function end, and function boundary identifications. We establish extensive experiments to compare our proposed Code Pointer Network with the Bidirectional RNN proposed in [Shin et al., 2015] on 120,000 binaries compiled from 120,000 C/C++ programs generated by Csmith [Yang et al., 2011] – a famous tool for generating codes using for the purpose of testing compilers. The experimental results shows that our proposed method significantly outperforms the baseline on function start, function end, and function scope identification tasks, especially for the hardest task of function scope identification. Regarding the amount of time taken for training, our proposed CPN is three times faster than baseline (i.e., around two hours in comparison with six hours) using the same number of iterations (i.e., 40,000 iterations).

8.2 Related work for function identification

In this section, we introduce some work related to ours. We firstly present the pointer network and its applications, and then move to discuss work in function identification.

A pointer network [Vinyals et al., 2015b] is a seq2seq model that deals with the case when the target dictionary is varied and non-fixed. The underlying idea of a pointer network is to employ a content-based attention mechanism to form a discrete distribution over inputs. Pointer networks have been applied to several real-world problems including finding convex hulls, Delaunay triangulation, the traveling salesman problem [Vinyals et al., 2015b], sorting an array [Vinyals et al., 2015a], and natural language processing [Gulcehre et al., 2016].

Function identification is not a real problem when working with source codes [Nguyen et al., 2019, 2020c]. However, this is a preliminary step in binary analysis. Besides the works that used heuristics, several machine learning works have attempted to solve this problem. The seminal work of [Rosenblum et al., 2008] modeled function start identification as a Conditional Random Field (CRF) in which binary offsets and a number of selected idioms (patterns) appear in the CRF. Since the inference on a CRF is very expensive, the computational complexity of this work is very high. Bao et al. [2014] used a weighted prefix tree to learn signatures for function starts, and then recognise function starts by matching binary fragments with the signatures. However, because of its high computational complexity, this work is not suitable for large-scale data sources. Recent methods [Shin et al., 2015, Nguyen et al., 2020b] proposed to employ a bidirectional RNN for function start (or function end) and function scope identification respectively. However, those work cannot be applied directly to or cannot address all cases (e.g., in the case, there exist functions nested in a function) of the toughest and most essential problem – the function scope identification problem. In 2017, Andriesse et al. [2017] based on Control Flow Graph analysis to propose a new solution for function identification.

8.3 Function identification

This section addresses the function identification problem. We begin with definitions of the sub problems in the function identification problem, followed by a typical example of source code generated by Csmith and its binaries compiled under optimisation levels O1 and Ox. Finally, we discuss on the challenges in this task.

8.3.1 Problem definitions

Given a binary program P, our task is to identify the necessary information in its n functions $\{f_1, ..., f_n\}$ which is initially unknown. According to the nature of information we need from $\{f_1, ..., f_n\}$, we can categorise the task of function identification into the following such problems.

Function start identification.

In this problem, we need to specify the set S $= \{s_1, ..., s_n\}$ which contains the first machine instructions of the corresponding functions in $\{f_1, ..., f_n\}$.

Function end identification.

In this problem, we need to identify the set E $= \{e_1, ..., e_n\}$ which contains the end machine instructions of the corresponding functions in $\{f_1, ..., f_n\}$.

Function boundary identification.

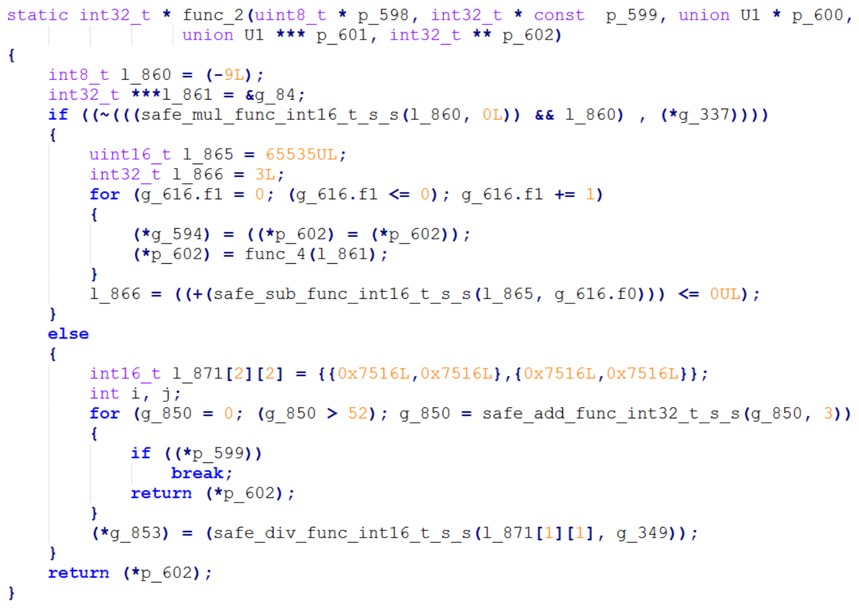
The function boundary identification problem is harder than the function start and function end identification problem. In this problem, we have to point out the set of (start, end) pairs SE $= \{(s_1, e_1), ..., (s_n, e_n)\}$ which contains the pairs of the function start and the function ends of the corresponding functions in $\{f_1, ..., f_n\}$.

Function scope identification.

This is the hardest problem in the function identification task. In this problem, we need to find out the set $\{(f_{1,s_1}, ..., f_{1,e_1}), ..., (f_{n,s_n}, ..., f_{n,e_n})\}$ which specifies the machine instructions in each function $f_1, ..., f_n$ in the given binary program P. It is apparent that the solution of this problem covers those of three aforementioned problems. We note that our proposed CPN addresses this hardest problem, hence inherently offering solutions for the other problems.

8.3.2 Running example

In Fig. 8.1, we show a typical example of a function generated by Csmith. According to our observation, source codes generated by Csmith have a wide range of variety in both control and data flows. Fig. 8.2 shows the assembly code of the source code in Fig. 8.1 which was compiled with the optimisation levels O1 and Ox on the Windows platform. It can be observed that the entry pattern for each optimisation level is different. Besides that the assembly code corresponding with the option Ox has three *rets* (i.e., return instruction) and the last *ret* is the real end point while the assembly code corresponding with the option O1 has only one *ret*. We further observe that in the real generated binary codes, the patterns for the entry point vary in a wide range and can start with *push*, *mov*, *movsx*, *inc*, *cmp*, *or*, *and*, etc. These make the task of function identification very challenging.



```c
static int32_t * func_2(uint8_t * p_598, int32_t * const  p_599, union U1 * p_600,
                        union U1 *** p_601, int32_t ** p_602)
{
    int8_t l_860 = (-9L);
    int32_t ***l_861 = &g_84;
    if ((~(((safe_mul_func_int16_t_s_s(l_860, 0L)) && l_860) , (*g_337))))
    {
        uint16_t l_865 = 65535UL;
        int32_t l_866 = 3L;
        for (g_616.f1 = 0; (g_616.f1 <= 0); g_616.f1 += 1)
        {
            (*g_594) = ((*p_602) = (*p_602));
            (*p_602) = func_4(l_861);
        }
        l_866 = ((+(safe_sub_func_int16_t_s_s(l_865, g_616.f0))) <= 0UL);
    }
    else
    {
        int16_t l_871[2][2] = {{0x7516L,0x7516L},{0x7516L,0x7516L}};
        int i, j;
        for (g_850 = 0; (g_850 > 52); g_850 = safe_add_func_int32_t_s_s(g_850, 3))
        {
            if ((*p_599))
                break;
            return (*p_602);
        }
        (*g_853) = (safe_div_func_int16_t_s_s(l_871[1][1], g_349));
    }
    return (*p_602);
}
```

Figure 8.1: Example source code of a function generated by Csmith.

```
0x1a10 func_2:
0x1a10:     movzx    eax, byte ptr [0x42d009]
0x1a17:     not      eax
0x1a19:     push     esi
0x1a1a:     mov      esi, dword ptr [esp + 0x18]
0x1a1e:     test     eax, eax
0x1a20:     je  0x1a42
0x1a22:     xor      eax, eax
0x1a24:     mov      word ptr [0x42d060], ax
...
0x1a62:     idiv     ecx
0x1a64:     jmp      0x1a6b
0x1a66:     mov      eax, 0x7516
0x1a6b:     movsx    ecx, ax
0x1a6e:     mov      dword ptr [0x42d090], ecx
0x1a74:     mov      eax, dword ptr [esi]
0x1a76:     pop      esi
0x1a77:     ret
```

(3.a) Compiled using MVS with O1.

```
0x2000 func_2:
0x2000:     movzx    eax, byte ptr [0x42d009]
0x2007:     not      eax
0x2009:     test     eax, eax
0x200b:     je  0x203a
0x200d:     xor      eax, eax
0x200f:     mov      word ptr [0x42d060], ax
0x2015:     mov      eax, dword ptr [esp + 0x14]
0x2019:     lea      esp, dword ptr [esp]
...
0x2037:     mov      eax, dword ptr [eax]
0x2039:     ret
0x203a:     movsx    ax, byte ptr [0x42d00a]
...
0x2069:     mov      eax, dword ptr [eax]
0x206b:     ret
0x206c:     mov      eax, 0x7516
...
0x207b:     mov      eax, dword ptr [eax]
0x207d:     ret
```

(3.b) Compiled using MVS with Ox.

Figure 8.2: Assembly codes of the example source code in Fig. 8.1 compiled using Microsoft Visual Studio (MVS) with the x86 architecture and Window platform under the optimisation levels O1 (left) and Ox (right).

8.3.3 Challenges of function identification tasks

In what follows, we list some challenges of the task of function identification. These challenges originate from various behaviors of compilers when compiling source codes under various combinations of optimisation levels (e.g., O1, O2, and O3 or Ox), processor architectures (e.g., x86 and x64), and platforms (e.g., Windows or Linux).

Not every machine instruction belongs to a function.

Compilers may introduce additional instructions for alignment and padding between or within a function which leads to some machine instructions that do not belong to any function.

Functions may be non-contiguous.

There may exist gaps between functions which can jump tables, data, or even instructions for completely different functions. In addition, as observed by [Harris and Miller, 2005], function sharing code can also lead to non-contiguous functions.

Functions may have multiple entries.

High-level languages use functions as an abstraction with a single entry. When compiled, however, functions may have multiple entries as a result of specialization. In addition, the number of patterns for function start can be enormous and varied according to optimisation levels, processor architectures, and platforms.

8.4 Code Pointer Network for function identification

In this section, we present our proposed Code Pointer Network (CPN) that can tackle the function identification task. We start with the discussion of how to process instructions to input to our CPN, followed by technical details for the training and testing procedures.

8.4.1 Processing input statement

We compiled the source code programs for the x86 architecture with three different optimisation levels (O1, O2 and Ox). Each machine instruction may have a different size which can be 4, 5, 6, 7, 8 bytes or even more. To the best of our knowledge, the first 4 bytes in x86 architecture mostly contains the crucial information for a machine instruction (i.e., the opcode and other crucial addresses), hence before feeding these machine instructions to the CPN, we preprocess them as follows: (1) keep the first 4 bytes from the left for machine instructions which are longer than 4 bytes and (2) padding 0 to the right of the machine instructions which have fewer than 4 bytes. For example, the machine instruction "mov dword ptr [0x42b008], 0x0" has the corresponding value "C70508B0420000000000" in hex format which is 10 bytes long, we then remove the last 6 bytes and keep the first 4 bytes to get the numerical value "C70508B0". For the machine instruction "xor al, al" which contains 2 bytes "32C0" in the hex format, we then pad with 0 to fill in the third and fourth bytes, and gain the numerical value "32C00000".

8.4.2 Code Pointer Network

8.4.2.1 Training procedure

The Code Pointer Network architecture is depicted in Fig. 8.3. Our CPN takes as input a binary program including a sequence of many machine instructions which may belong to different functions. The task of the CPN is to read out the scope of the first function in the input binary program. To this end, as in a typical pointer network, our proposed CPN consists of two components: one encoder and one decoder. The encoder's task is to encode the sequence of machine instructions in a binary program, while the decoder tries to decode the encoded output of the encoder to read out the indexes of the machine instructions in the first function in the given binary program. In addition, to signal the end of the function, we introduce the specific symbol EOF whose value is $\mathbf{i}_E$ which is randomly initialised. This value $\mathbf{i}_E$ is inputted to the CPN encoder right after the last instruction in the binary program (cf. Fig. 8.3).

To reduce the sequence length of the CPN encoder, at a time, we input two consecutive functions

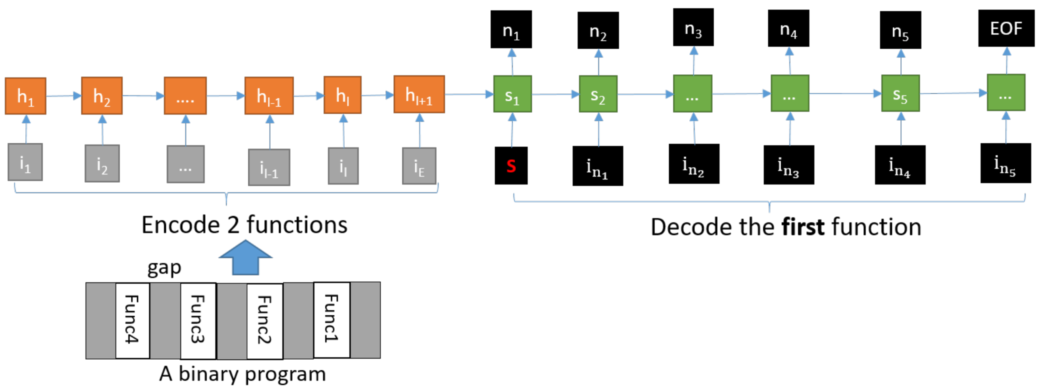


Figure 8.3: Architecture of the Code Pointer Network (CPN).

including gaps between functions if existing in a binary program to the encoder. For example, as shown in Fig. 8.3, a binary program that consists of 4 functions forms 3 pairs of two consecutive functions (i.e., (gap, func1, gap, func2, gap), (gap, func2, gap, func3, gap) and (gap, func3, gap, func4, gap)), and the encoder takes each pair at a time. Let us now define the input sequence $\mathrm{I} = \{\mathbf{i}_1, \mathbf{i}_2, ..., \mathbf{i}_{l+1}\}$ (i.e., the machine instructions in a pair) and the output sequence $\mathrm{O} = \{n_1, n_2, ..., n_m, n_{m+1}\}$ where n_{m+1} specifies the index of the symbol EOF and which includes the indexes of machine instructions in the first function in the input pair. We learn the model parameters θ by maximizing the following conditional probabilities over the training set $\mathcal{D}$ which includes the pair I and the indexes of machine instructions in its first function O:

$$\theta^* = \underset{\theta}{\text{argmax}} \sum_{(\mathrm{I},\mathrm{O})\in\mathcal{D}} \log p(\mathrm{O}|\mathrm{I};\theta)$$

where we have defined

$$p(\mathrm{O}|\mathrm{I};\theta) = \prod_{k=1}^{m} p(n_k|n_1, ..., n_{k-1}, \mathrm{I};\theta) \tag{8.1}$$

We now denote $\{\mathbf{h}_1, \mathbf{h}_2, ..., \mathbf{h}_{l+1}\}$ and $\{\mathbf{s}_0, \mathbf{s}_1, \mathbf{s}_2, ..., \mathbf{s}_m\}$ with $\mathbf{s}_0 = \mathbf{s}$ as the encoder hidden states and decoder hidden states respectively. Since $\mathbf{s}_k$ is a function of $n_1, ..., n_{k-1}, \mathrm{I}$ or a lossy summary of this sequence, we can reasonably simplify $p(\mathrm{O}|\mathrm{I};\theta)$ as follows:

$$p(\mathrm{O}|\mathrm{I};\theta) = \prod_{k=1}^{m} p(n_k|\mathbf{s}_k;\theta)$$
$$\log p(\mathrm{O}|\mathrm{I};\theta) = \sum_{k=1}^{m} \log p(n_k|\mathbf{s}_k;\theta)$$

To define the probability $\log p(n_k|\mathbf{s}_k;\theta)$ where $n_k \in \{1, 2, \dots, l+1\}$, we first compute the align-

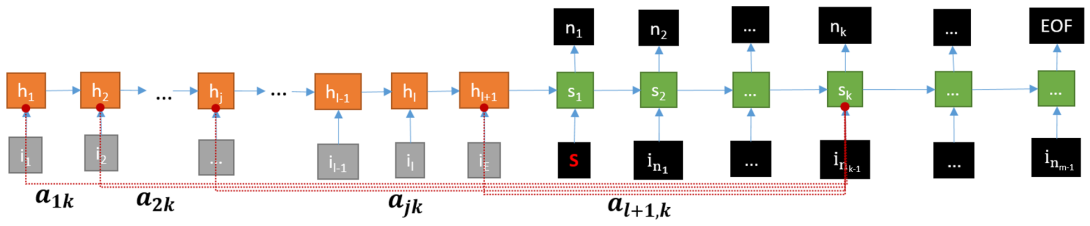


Figure 8.4: Alignment scores in the CPN.

ment scores between the decoder hidden state $\mathbf{s}_k$ and the encoder hidden states $\mathbf{h}_j$, $\forall j = 1, \dots, l+1$:

$$\mathbf{a}_{jk} = \mathbf{v}^{\mathbf{T}}\tanh(\mathrm{U}_h\mathbf{h}_j + \mathrm{U}_s\mathbf{s}_k) \tag{8.2}$$

$$\text{or } \mathbf{a}_{jk} = \mathbf{v}^{\mathbf{T}}\tanh\left(\mathrm{U}\begin{bmatrix}\mathbf{h}_j\\ \mathbf{s}_k\end{bmatrix}\right) \tag{8.3}$$

where the vector $\mathbf{v}$ and the matrices U, U_h, Us are learnable parameters.

We then apply the softmax to $[\mathbf{a}_{jk}]_{j=1}^{l+1}$ to gain the vector $\mathbf{b}_k$ and define $p(n_k|\mathbf{s}_k;\theta)$ as the n_k-th element in this vector:

$$\mathbf{b}_k = \text{softmax}\left([\mathbf{a}_{jk}]_{j=1}^{l+1}\right)$$
$$p(n_k|\mathbf{s}_k;\theta) = \mathbf{b}_{k,n_k}$$

8.4.2.2 Predicting procedure

In the predicting procedure, given a specific binary program, we first input this binary program into the encoder of the trained model to read out the first function (machine instructions and their positions). The detected function is then eliminated from the binary program and the remaining binary code is once again inputted to the CPN. This process is repeated until the last function. We visualise the process for the predicting procedure in Fig. 8.5.

8.5 Implementation and results

In this section, we present the experimental results of our proposed Code Pointer Network compared with the Bidirectional RNN [Shin et al., 2015]. We also investigate the performance of our CPN with various RNN cells (e.g., long short-term memory (LSTM) and gated recurrent

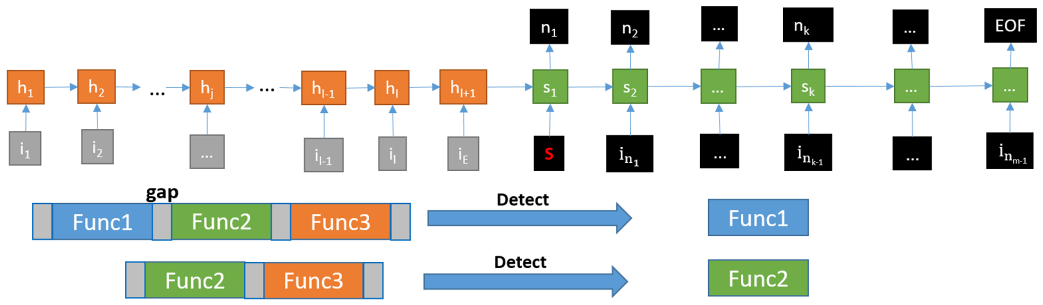


Figure 8.5: Predicting procedure of the CPN.

unit (GRU) cells), and alignment score formulas as shown in Eqs. (8.2 and 8.3). We note that we could not experiment with ByteWeight [Bao et al., 2014] since this method is not scalable enough for our data (i.e, ByteWeight took about 587 compute-hours for training a dataset of 2,064 binaries [Shin et al., 2015] while our data has 120,000 random C/C++ programs). In addition, the Bidirectional RNN has been proven to outperform ByteWeight.

8.5.1 Experimental setup

8.5.1.1 Data collection

We used Csmith [Yang et al., 2011], which is a well-known tool for generating random C/C++ programs conforming dynamically and statically to the C99 standard, to generate 120,000 random C/C++ programs. Each generated program has a number of functions in the range {2, 3, 4, 5}. Binaries were then compiled from the source programs using Microsoft Visual Studio in the debug mode with one of three different optimisation levels O1 (for creating the smallest code), O2 (for creating the fastest code), and Ox (with full optimisation options including smallest and fastest code) under x86 (32 bit) architecture. Finally, we used DIA2Dump[1] to read the debug files (i.e., *.pdb) for creating the labelled dataset.

8.5.1.2 Experimental setting

We divided the binaries into three random parts; the first part contains 80% of the binaries used for training, the second part contains 10% of the binaries used for testing, and the third part contains 10% of the binaries for validation. For each the method, we trained the competitive methods over 40,000 iterations.

For the Bidirectional RNN, we used identical settings and the architecture proposed in [Shin et al., 2015]. In particular, we chopped the binaries into chunks of 1,000 bytes. This means

[1]https://docs.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/debug-interface-access-sdk

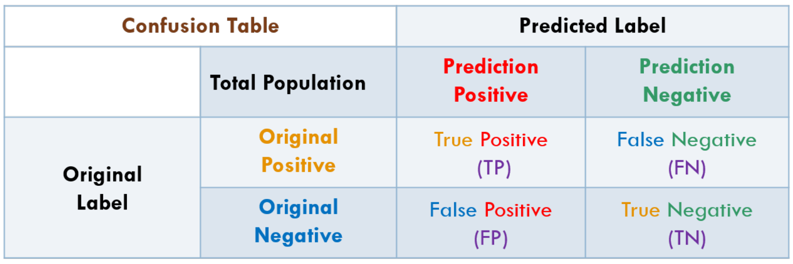
that we run recurrent neural networks forward and backward on a 1000-byte sequence from the corresponding binaries. The size of the hidden state for the forward and backward RNN is 32. Then the forward and backward RNN are concatenated to feed into a linear transformation and the soft-max function. This process produces a probability distribution to identify whether a byte corresponds to the beginning (or end) of a function or not. We employed the RMSprop optimiser with the learning rate varying in the range of $\{0.01, 0.1\}$ and the batch size is set to 32. We trained the models for function start identification task and function end identification task separately. The Bidirectional RNN proposed in [Shin et al., 2015] is not directly applicable to function scope identification. To make it applicable to this task, we first paired the corresponding function starts and function ends detected by two bidirectional RNNs and assume that each pair forms a boundary for a function where all machine instructions in this boundary are counted as that in this function and counted it as a correct function prediction if this matches exactly a real function.

For our model in the training process, we used the encoder with a sequence of 100 hidden states where the hidden state size is 128. For each binary, we concatenate sequentially two functions as input for the Code Pointer Network in the encoder process. We employed the Adam optimiser with the default learning rate 0.001 and the batch size 128. In addition, we applied gradient clipping regularisation [Pascanu et al., 2013] to prevent the over-fitting problem when training the model. Because our CPN solves the function scope identification task, it can be inherently applicable to the function start and function end identification tasks.

We implemented the Code Pointer Network and the Bidirectional RNN in Python using Tensorflow [Abadi et al., 2016], an open-source software library for Machine Intelligence developed by the Google Brain Team. We ran our experiments on an Intel Xeon Processor E5-1660 which has 8 cores at 3.0 GHz and 128 GB of RAM.

8.5.1.3 Metrics

In order to evaluate performances of function identification methods, we employ three measures including recall (R), precision (P) and F1-score (F1) which are widely used to report predictive performances on imbalanced datasets. This is due to the fact that the number of function starts, function ends, etc., are fewer than the number of machine instructions. Given a dataset with two kinds of labels: positive and negative labels, the precision is the fraction of the number of true positive instances among the number of original positive instances. The recall is the faction of the number of true positive instances among the number of predicted positive instances. The



Confusion Table		Predicted Label	
	Total Population	Prediction Positive	Prediction Negative
Original Label	Original Positive	True Positive (TP)	False Negative (FN)
	Original Negative	False Positive (FP)	True Negative (TN)

Figure 8.6: Confusion table.

F1-score is the most important measure which aggregates both precision and recall. The recall, precision and F1-score have the following forms:

$$
\begin{aligned}
\mathrm{P} &= \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FP}} \\
\mathrm{R} &= \frac{\mathrm{TP}}{\mathrm{TP}+\mathrm{FN}} \\
\mathrm{F1} &= \frac{2\times \mathrm{P}\times\mathrm{R}}{\mathrm{P}+\mathrm{R}}
\end{aligned}
$$

where TP, FP and FN are the number of true positives, false positives and false negatives respectively which can be defined using a confusion table as shown in Fig. 8.6.

Optimisation	O1								
Methods	Function Start			Function End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN	**96.53%**	97.26%	**96.89%**	94.09%	**94.64%**	**94.36%**	**92.77%**	**93.15%**	**92.96%**
Bidirectional RNN	86.30%	**98.14%**	91.84%	**96.42%**	81.58%	88.38%	81.56%	82.17%	81.87%
Optimisation	O2								
Methods	Function Start			Function End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN	**95.18%**	93.27%	**94.23%**	**89.60%**	**87.87%**	**88.73%**	**87.55%**	**88.20%**	**87.87%**
Bidirectional RNN	82.48%	**98.16%**	89.63%	87.86%	72.52%	79.45%	73.14%	78.43%	75.69%
Optimisation	Ox								
Methods	Function Start			Function End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN	**94.43%**	**93.77%**	**94.10%**	**89.07%**	**88.57%**	**88.82%**	**87.34%**	**88.01%**	**87.67%**
Bidirectional RNN	74.39%	80.02%	77.10%	79.62%	70.14%	74.58%	71.18%	73.21%	72.18%

Table 8.1: Comparison of our Code Pointer Network and the Bidirectional RNN. The best results (%) are emphasised in **bold**.

8.5.2 Experimental results

8.5.2.1 Code Pointer Network versus Bidirectional RNN

We compare our method using Eq. (8.3) for computing alignment scores and long short-term memory (LSTM) for RNN cell with the Bidirectional RNN proposed in [Shin et al., 2015]. The experimental results show that our proposed method achieves better performance in most cases in terms of predictive performance and training time. In particular, Table 8.1 indicates that

Optimisation	O1								
Methods	Function Start			Function End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN-LSTM-01	**95.32%**	96.42%	**95.87%**	**92.75%**	**90.85%**	**91.79%**	**90.27%**	**91.19%**	**90.73%**
CPN-GRU-01	92.78%	**96.45%**	94.58%	90.14%	90.52%	90.33%	84.91%	87.40%	86.14%
CPN-RNN-01	91.32%	96.16%	93.68%	90.07%	90.42%	90.24%	85.26%	86.54%	85.90%
Optimisation	O2								
Methods	Function Start			Functions End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN-LSTM-01	**93.09%**	**93.38%**	**93.23%**	87.04%	87.36%	87.20%	**86.75%**	**87.96%**	**87.35%**
CPN-GRU-01	93.03%	93.25%	93.14%	**87.48%**	**87.69%**	**87.58%**	84.35%	85.45%	84.90%
CPN-RNN-01	91.76%	92.86%	92.30%	86.52%	87.63%	87.07%	83.41%	84.24%	83.82%
Optimisation	Ox								
Methods	Function Start			Functions End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN-LSTM-01	**92.47%**	**93.16%**	**92.81%**	**86.35%**	**85.54%**	**86.94%**	**86.56%**	**87.71%**	**87.12%**
CPN-GRU-01	90.81%	93.94%	92.35%	85.87%	88.71%	87.27%	84.59%	84.73%	84.65%
CPN-RNN-01	90.53%	92.43%	91.47%	84.95%	87.56%	86.24%	83.32%	84.17%	83.74%

Table 8.2: Comparison of the variants of CPN using different types of RNN cells including LSTM, GRU and the standard RNN cell. The best results (%) are emphasised in **bold**.

our proposed CPN achieved better predictive performance (i.e., R: Recall, P: Precision, and F1: F1-score) with a wide margin in most cases, especially for the highest optimisation level Ox our CPN significantly outperformed the baseline in all measures. Regarding the training time, our CPN is approximately three times faster than the baseline. In particular, with the same number of iterations (i.e., 40,000 iterations), the CPN took around 2 hours to finish, while the baseline took around 6 hours.

8.5.2.2 Variations in RNN cells

In Table 8.2, we compare the performances of our CPN using different RNN cells such as long short-term memory (CPN-LSTM-01) and gated recurrent unit (CPN-GRU-01) with the basic RNN cell (CPN-RNN-01) with Eq. (8.2) for computing alignment score. It can be observed that LSTM achieved better performance than GRU which in turn performed better than the basic RNN cell in most cases.

8.5.2.3 Variation in attention mechanism techniques

In Table 8.3, we compare the performances of our CPN using different formulas for computing alignment score as in Eq. (8.2) (CPN-LSTM-01) with Eq. (8.3) (CPN-LSTM-02) while employing the LSTM cell. The experimental results show that CPN-LSTM-02 obtained better performances in most cases compared with CPN-LSTM-01.

Optimisation	O1								
Methods	Function Start			Function End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN-LSTM-01	95.32%	96.42%	95.87%	92.75%	90.85%	91.79%	90.27%	91.19%	90.73%
CPN-LSTM-02	**96.53%**	**97.26%**	**96.89%**	**94.09%**	**94.64%**	**94.36%**	**92.77%**	**93.15%**	**92.96%**
Optimisation	O2								
Methods	Function Start			Function End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN-LSTM-01	93.09%	**93.38%**	93.23%	87.04%	87.36%	87.20%	86.75%	87.96%	87.35%
CPN-LSTM-02	**95.18%**	93.27%	**94.23%**	**89.60%**	**87.87%**	**88.73%**	**87.55%**	**88.20%**	**87.87%**
Optimisation	Ox								
Methods	Function Start			Function End			Function Scope		
	R	P	F1	R	P	F1	R	P	F1
CPN-LSTM-01	92.47%	93.16%	92.81%	86.35%	85.54%	86.94%	86.56%	87.71%	87.12%
CPN-LSTM-02	**94.43%**	**93.77%**	**94.10%**	**89.07%**	**88.57%**	**88.82%**	**87.34%**	**88.01%**	**87.67%**

Table 8.3: Comparison of variants of CPN using Eq. (8.2) (CPN-LSTM-01) and Eq. (8.3) (CPN-LSTM-02) for computing alignment score. The best results (%) are emphasised in **bold**.

8.6 Closing remarks

In this chapter, we have proposed the novel Code Pointer Network for dealing with the function identification problem, a preliminary and significant step in binary analysis for many security applications such as malware detection, common vulnerability detection and binary instrumentation. Specifically, the Code Pointer Network leverages the underlying idea of a pointer network in order to tackle the function scope identification, the hardest and most crucial task in function identification. The experimental results show that the Code Pointer Network can achieve the state-of-the-art performances in terms of efficiency and efficacy.

Chapter 9

Conclusion

In this chapter, we review all of our contributions to the software vulnerability detection (SVD) problem in terms of addressing some of the crucial problems existing in current SVD methods. At the end of the chapter, we discuss some possible extensions to our work mentioned in Chapters 3 and 7.

9.1 Summary

In this thesis, we have successfully advanced the deep learning approach to propose novel deep learning-based methods to address the remaining problems existing in current SVD methods. These problems are relevant to the three aforementioned research questions:

(Q.1) How to *transfer efficiently the learning on software vulnerabilities* from labelled projects (i.e., source domains) to other unlabelled projects (i.e., target domains).

(Q.2) How to efficiently exploit the semantic and syntactic relationships inside source code to *detect vulnerabilities at a fine-grained level with more flexible scope* (i.e., the statement level) than the function or program levels.

(Q.3) How to leverage the information from binaries (i.e., byte instructions) and assemblies (i.e., machine instructions) programs to *deal with all cases* (i.e., the function start identification, function end identification, function boundary identification and function scope identification problems) *of the function identification problem*, especially the function scope identification problem, the toughest and most essential problem.

Our contributions in Part I, Deep Domain Adaptation for Software Vulnerability Detection, are novel deep learning-based methods, i.e., the answers to research question (Q.1) "How to *transfer efficiently the learning on software vulnerabilities* from labelled projects (i.e., source domains) to other unlabelled projects (i.e., target domains)".

In the work summarised in Chapter 3, Deep Domain Adaptation for Vulnerable Code Function Identification, we first propose the Code Domain Adaptation Network (CDAN), a novel architecture which can tackle software source code and transfer learning from labelled software projects (source domain) to unlabelled projects (target domain) using the GAN principle. At the Nash equilibrium point, the gap between the source and target domains vanishes, and consequently the supervised source classifier trained on the source domain can be transferred to predict well in the target domain. We observe that when successfully bridging the gap between the source and target domains, the target source code can be considered the unlabelled portion of the source domain in a semi-supervised learning context. The information carried in the target source code is certainly helpful in boosting the predictive performance. To further utilise the information carried in the unlabelled target source code, we propose the Semi-supervised Code Domain Adaptation Network (SCDAN), wherein the clustering assumption [Chapelle and Zien, 2005] is enforced. In particular, we simultaneously encourage the source classifier of the SCDAN to be confident in its decisions for predicting the source and target source code, and provide smooth predictive outputs in the source and target domains.

Our work is the first to formulate transferred learning from a source of sequences to a target of sequences. Moreover, our contribution is formulation of a novel architecture named CDAN for SVD, which is an extremely important problem in cybersecurity. Not only does our work involve both a model contribution and a significant real-world application of DDA, we believe it is a new building block for a wide array of other applications in other domains such as behaviour modelling in fintech where temporal dynamics are important and sequence modelling in computational biology.

Based on the proposed architecture (CDAN), we reapply the models DDAN, MMD and DIRT-T proposed in [Ganin and Lempitsky, 2015, Long et al., 2015, Shu et al., 2018]. We subsequently propose SCDAN to more efficiently exploit and utilise information from unlabelled target data. We further demonstrate the effectiveness and advantage of ***S***CDAN by undertaking experiments on six real-world datasets. The experimental results show that SCDAN outperforms the baselines by a wide margin.

In the work summarised in Chapter 4, Dual-component Deep Domain Adaptation: A New Approach for Cross-project Software Vulnerability Detection", we aim to address the problems of both mode collapsing and boundary distortion in deep domain adaptation methods employing the GAN as a principle in order to close the gap between source and target data in the joint feature space. Our approaches apply manifold regularisation for enabling the preservation of manifold/clustering structures in the joint feature space, hence avoiding the degeneration of source and target data in this space, and invoking dual discriminators in an elegant way to reduce the negative impacts of the mode collapsing and boundary distortion problems in deep domain adaptation using the GAN principle as mentioned. We name our mechanism when applied to SVD the Dual Generator-Discriminator Deep Code Domain Adaptation Network (Dual-GD-DDAN).

In addition, we incorporate the relevant approaches in terms of minimising conditional entropy and manifold regularisation with a spectral graph proposed in [Nguyen et al., 2019] to enforce the clustering assumption [Chapelle and Zien, 2005] and arrive at a new model named the Dual Generator-Discriminator Semi-supervised Deep Code Domain Adaptation Network (Dual-GD-SDDAN). The experimental results show that our Dual-GD-SDDAN can overcome the mode collapsing and boundary distortion problems better than SCDAN in [Nguyen et al., 2019], hence obtaining better predictive performance.

Our contributions in Part II, Learning to Explain Software Vulnerability, are novel deep learning-based methods, i.e., the answers to research question (Q.2): "How to efficiently exploit the semantic and syntactic relationships inside source code to *detect vulnerabilities at a fine-grained level with more flexible scope* (i.e., the statement level) than the function or program levels".

In the work summarised in Chapter 5, Information-theoretic Source Code Vulnerability Highlighting, we propose a novel learn-to-explain model that is based on mutual information and takes into account the sequential nature of data to better evaluate mutual information. Using this theory, we propose a novel architecture based on multi-Bernoulli distribution for random subsets of statement selection (i.e., we aim to highlight the top-K statements that are the most relevant to the vulnerable and non-vulnerable class labels). Unlike the multinomial distribution used in L2X [Chen et al., 2018], our mechanism is more controllable and enables us to train the model in a semi-supervised context. In addition, our proposed model can be used to highlight the core statements that are a subset of the most relevant statements of vulnerable source code. It can also explain how the reference model works by identifying the most important statements that contribute to its prediction.

We conduct experiments on the datasets collected by [Li et al., 2018] that contain source code of vulnerable and non-vulnerable functions from two real-world software data sources and compare our proposed method to a state-of-the-art baseline L2X approach on these two datasets. We further investigate our proposed method in a semi-supervised learning context by comparing it to itself in an unsupervised learning context. We demonstrate that our proposed method can detect vulnerable code statements in functions much more effectively than L2X in unsupervised context and its semi-supervised variant can significantly boost the performance.

In the work summarised in Chapter 6, Information-theoretic End-to-end Models to Identify Code Statements Causing Software Vulnerability, we study the important problem of fine-grained vulnerability detection, which has a variety of applications in different areas such as software engineering and cybersecurity. Automated deep learning-based techniques for this problem have not yet been well studied. In particular, we propose a novel probabilistic framework learned by variational inference with various model constructions and training mechanisms which are derived from an information-theoretic perspective. Our proposed approaches can work effectively and efficiently in both unsupervised and semi-supervised contexts. These approaches have great potential to serve as powerful tools for practitioners including developers and security experts.

We comprehensively evaluate our proposed framework with different variants for real-world software datasets in both unsupervised and semi-supervised cases. Our extensive experiments show that our approaches can accurately identify the vulnerability-relevant statements in an end-to-end manner.

Our contributions in Part III, Deep Sequence-to-sequence Models for Function Scope Identification in Binary Programs, are novel deep learning-based methods, i.e., the answers to research question (Q.3): "How to leverage the information from binaries (i.e., byte instructions) and assemblies (i.e., machine instructions) to *deal with all cases* (i.e., the function start identification, function end identification, function boundary identification and function scope identification problems) *of the function identification problem*, especially the function scope identification problem, the toughest and most essential problem".

In the work summarised in Chapter 7, Code Action Network for Binary Function Scope Identification, we propose a novel method named the Code Action Network (CAN) whose underlying idea is to equivalently transform the task of function scope identification into learning a sequence of action states. Inspired by the idea of a Turing machine, we imagined a memory tape consisting of many cells on which machine instructions of a binary are stored. The head is first pointed to the first machine instruction located in the first cell. Each machine instruction is assigned to an action state in the action state set {NI, NE, FE} depending on its nature. After reading the current machine instruction and assigning the corresponding action state to it, the head is moved to the next cell and this procedure is halted as we reach the last cell in the tape.

We undertake extensive experiments to compare our proposed CAN-B (at the byte level) and CAN-M (at the machine instruction level) with state-of-the-art methods including IDA, the Bidirectional RNN, ByteWeight no-RFCR, and ByteWeight on the dataset used in [Shin et al., 2015, Bao et al., 2014]. The experimental results show that our proposed CAN-B and CAN-M outperform the baselines on function start, function end and function boundary identification tasks, as well as achieving very good performance on function scope identification and also surpassing the Nucleus [Andriesse et al., 2017] on this task. Moreover, our method CAN-M takes about 1 hour for training with 20,000 iterations which is nearly four times faster than the Bidirectional RNN proposed in [Shin et al., 2015] using the same number of iterations for training and the same number of bytes for handling input. This is due to the fact that CAN-M operates at the machine instruction level, while the Bidirectional RNN proposed in [Shin et al., 2015] operates at the byte level.

In the work summarised in Chapter 8, Code Pointer Network for Binary Function Scope Identification, we propose a novel method named the Code Pointer Network (CPN) that employs the idea of a pointer network [Vinyals et al., 2015b] in the specific context of function identification. Our proposed CPN includes one encoder and one decoder. The encoder takes the sequence of all machine instructions in a binary, while the decoder reads out the function scopes in the given binary. In addition, unlike the work of [Shin et al., 2015], our proposed CPN can directly address the function scope identification task, hence inherently offering the solutions for other simpler tasks including function start, function end, and function boundary identifications. We establish extensive experiments to compare our proposed CPN with the Bidirectional RNN proposed in [Shin et al., 2015].

The experimental results shows that our proposed method significantly outperforms the baseline on function start, function end and function scope identification tasks, especially for the hardest task of function scope identification. Regarding the amount of time taken for training, our proposed CPN is three times faster than baseline (i.e., around two hours in comparison with six hours) using the same number of iterations (i.e., 40,000 iterations).

9.2 Future work

In this section, we discuss some possible future extensions to our work mentioned in Chapters 3 and 7.

In Chapter 3, our CDAN architecture has some limitations, and below we propose our solutions.

Firstly, in the CDAN architecture, we use a bidirectional RNN with long short-term memory (LSTM) cells for the generator G to model the relationship between code statements in each function. The LSTM cell can address the exploding and vanishing gradients problems, as well as the short-term memory problem known to exist in RNNs and bidirectional RNNs. However, LSTM models are difficult to train because they require memory bandwidth-bound computations. In particular, an LSTM model uses four linear layers (multilayer perceptron) that require a large memory bandwidth to be computed for each cell and for each sequence time step. Moreover, in practice, the long short-term memory cell may still encounter the short-term memory problem when dealing with a long sequence length of data.

To address this problem, we propose applying self-attention layers [Vaswani et al., 2017] to the generator G used in CDAN. Self-attention layers have demonstrated more efficiency in modelling long term dependencies in temporal sequences compared to conventional sequential models such as RNNs, LTSMs and GRUs. The major advantage of the self-attention architecture is that at each step we have direct access to all the other steps using the self-attention mechanism, which leaves almost no room for information loss as far as message passing is concerned. We can look at both future and past elements at the same time which also brings the benefit of bidirectional RNNs. As a result, in considering a source code function as a temporal sequence and each code statement in a function as an element in this temporal sequence, we can apply the idea of the self-attention mechanism in further effectively obtaining the dependency relationship among code statements in each function.

Secondly, in the computational process of CDAN architecture, there is a possible issue with using $H = \text{concat}(\boldsymbol{h}_1, ..., \boldsymbol{h}_L)$ to obtain the output $\boldsymbol{o}$. In particular, if the value l (i.e., the time steps) used in the bidirectional RNN $\mathcal{B}(\boldsymbol{x})$ is too large (e.g, $L > 100$), the concatenation vector from $\boldsymbol{h}_1$ to $\boldsymbol{h}_L$ denoted by $H = \text{concat}(\boldsymbol{h}_1, ..., \boldsymbol{h}_L)$ can be considerably long. It is difficult for the network to find which statements it should focus on to achieve high label prediction performance (i.e., for a given section of source code, there are only some specific code statements that contribute to the vulnerability of the source code).

To deal with this problem, we propose applying the attention mechanism to the outputs of the generator G. This allows us to focus on the code statements in the source code that contribute more to the vulnerability detection decision of the source classifier, as well as forming a joint space that better fits the transfer of learning from the source to target domains.

The new architecture CDAN-A that integrates the attention mechanism and self-attention layers into CDAN can help to further exploit the semantic relationships among statements in source code and bring higher predictive performance. The architecture of CDAN-A is presented in Fig. 9.1 wherein, as before, the source and target networks share parameters and are identical. The representation of each function in the joint space named $\boldsymbol{o} = G(\boldsymbol{x})$ is formed as follows:

$$O = [\boldsymbol{o}_1, \boldsymbol{o}_2, ..., \boldsymbol{o}_L]$$
$$\boldsymbol{\gamma} = \text{softmax}(Ow^\top)$$
$$\boldsymbol{o} = \boldsymbol{\gamma}^\top O$$

Where w is a trainable vector while O is the output of the self-attention layers. Each vector $\boldsymbol{o}_j$ with $j \in [1, ..., L]$ stands for the representation of the *i-th* statement in the function. The vector $\boldsymbol{\gamma}$ plays a role as a weighted coefficient vector with the sum of all its element values equal to 1. Each element value in vector $\boldsymbol{\gamma}$ will specify the weight corresponding to each statement that contributes to the label prediction process (vulnerable label or non-vulnerable label) of the function. The output vector $\boldsymbol{o} = \boldsymbol{\gamma}^\top O$ is a linear combination between O and $\boldsymbol{\gamma}$ aiming not only to focus on statements that mostly contribute to the vulnerability prediction decision of the model, but also to form a joint space that better fits the transfer of learning from the source to target domains.

In particular, given a source code function $\boldsymbol{x}_i$ including L code statements $[\boldsymbol{x}_{i1}, \ldots, \boldsymbol{x}_{iL}]$, the representation of $\boldsymbol{x}_i = [\boldsymbol{x}_{i1}, \ldots, \boldsymbol{x}_{iL}]$ in the joint (latent) space represented as $O = [\boldsymbol{o}_1, \ldots, \boldsymbol{o}_L]$ using self-attention layers is formed as follows:

- Scaled Dot-Product Attention: We first compute the query Q, the key K and the value V matrices. We do that by packing our embeddings $[\boldsymbol{i}_{i1}, \ldots, \boldsymbol{i}_{iL}]$ where $\boldsymbol{i}_{ij} \in \mathbb{R}^d, \forall j \in \{1, .., L\}$ (i.e., where $\boldsymbol{i}_{ik}$ is the embedding vector of the statement $\boldsymbol{x}_{ik}$) into a matrix X and multiplying it by the weighted matrices being trained including $W_Q \in \mathbb{R}^{d \times d_k}$, $W_K \in \mathbb{R}^{d \times d_k}$ and $W_V \in \mathbb{R}^{d \times d_v}$ where $d_k = d_v = d/n$. We then calculate the outputs of the Scaled Dot-Product Attention as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^\intercal}{\sqrt{d_k}})V$$

We use a multi-headed version with n heads as introduced in [Vaswani et al., 2017]:

$$O^l = \text{MultiHead-Attention}(X) = \text{Concat}(\text{head}_1, ..., \text{head}_n)W^O$$

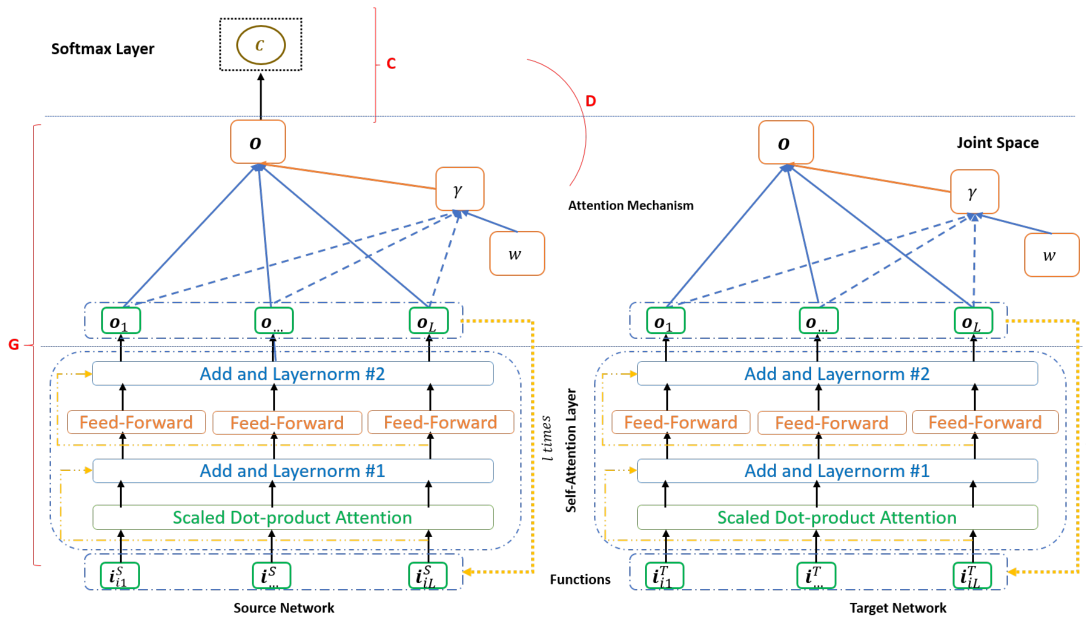


Figure 9.1: Architecture of our Code Domain Adaptation Network using the attention mechanism technique (CDAN-A). The generator G takes a sequence of code statements (i.e., each code statement in vectorial form) and maps this sequence to the joint layer (i.e., the joint space). We apply the attention mechanism to form the generator G that maps the input source code to the joint space. We note that the source and target networks share parameters and are identical. Recall that $\boldsymbol{i}_{ik}$ is the embedding vector of the statement $\boldsymbol{x}_{ik}$.

where $\text{head}_i = \text{Attention}(O^l W_{Q_i}, O^l W_{K_i}, O^l W_{V_i})$ with $O^0 = X$.

- Add and LayerNorm #1: The output from the Scaled Dot-Product Attention O^l is combined with O^{l-1} (with $O^0 = X$) to gain the input for the first LayerNormalisation (LayerNorm #1):

$$O^l = \text{LayerNormalisation}(O^l + O^{l-1})$$

- Feedforward neural network: The outputs O^l from the first Layernormalisation (LayerNorm #1) will be inputted to a feedforward neural network $g(., \alpha)$ consisting of two neural layers with the first layer using the ReLU activation function:

$$O^l_g = g(O^l, \alpha)$$

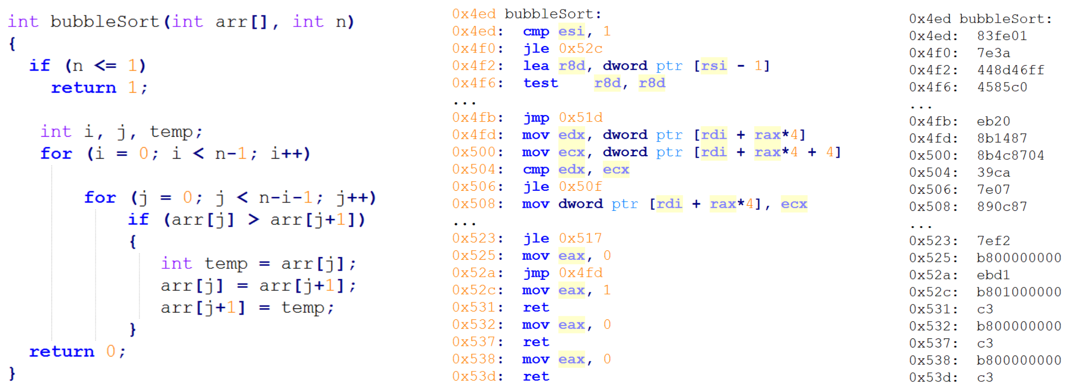
- Add and LayerNorm #2: The outputs O^l_g obtained from the feedforward neural network $g(., \alpha)$ is combined with the output from the LayerNorm #1 (O^l) before being inputted to the second LayerNormalisation (LayerNorm #2) to gain the outputs for the current self-attention layer:

$$O^l = \text{LayerNormalisation}(O^l + O^l_g)$$

Based on our new proposed CDAN-A architecture, we propose the SCDAN-A method that leverages the ideas of Semi-supervised Deep Code Domain Adaptation (SCDAN) with our CDAN-A architecture.

In Chapter 7, to further take advantage of the context information of the machine instructions around each machine instruction when predicting its label, we propose to integrate an attention mechanism into CAN at the last layer (i.e. at the second layer) to obtain CAN-A.

The attention mechanism applied in CAN-A allows us to further incorporate the context information of machine instructions before and after a given machine instruction when predicting its label, and thus achieve higher predictive performance. For example, in Fig. 9.2, we can see that predicting the label (i.e. NI) of the machine instruction (mov edx, dword ptr [rdi + rax*4]) at the address (0x4fd) can benefit from utilising the context information of the machine instructions around it.



```
int bubbleSort(int arr[], int n)
{
  if (n <= 1)
    return 1;

   int i, j, temp;
   for (i = 0; i < n-1; i++)

       for (j = 0; j < n-i-1; j++)
           if (arr[j] > arr[j+1])
           {
              int temp = arr[j];
              arr[j] = arr[j+1];
              arr[j+1] = temp;
           }
  return 0;
}
```

```
0x4ed bubbleSort:
0x4ed:  cmp esi, 1
0x4f0:  jle 0x52c
0x4f2:  lea r8d, dword ptr [rsi - 1]
0x4f6:  test    r8d, r8d
...
0x4fb:  jmp 0x51d
0x4fd:  mov edx, dword ptr [rdi + rax*4]
0x500:  mov ecx, dword ptr [rdi + rax*4 + 4]
0x504:  cmp edx, ecx
0x506:  jle 0x50f
0x508:  mov dword ptr [rdi + rax*4], ecx
...
0x523:  jle 0x517
0x525:  mov eax, 0
0x52a:  jmp 0x4fd
0x52c:  mov eax, 1
0x531:  ret
0x532:  mov eax, 0
0x537:  ret
0x538:  mov eax, 0
0x53d:  ret
```

```
0x4ed bubbleSort:
0x4ed:  83fe01
0x4f0:  7e3a
0x4f2:  448d46ff
0x4f6:  4585c0
...
0x4fb:  eb20
0x4fd:  8b1487
0x500:  8b4c8704
0x504:  39ca
0x506:  7e07
0x508:  890c87
...
0x523:  7ef2
0x525:  b800000000
0x52a:  ebd1
0x52c:  b801000000
0x531:  c3
0x532:  b800000000
0x537:  c3
0x538:  b800000000
0x53d:  c3
```

Figure 9.2: Example source code of a function in C language programming (left); the corresponding assembly code (middle) with some parts omitted for brevity and the corresponding hexadecimal mode of the binary code (right).

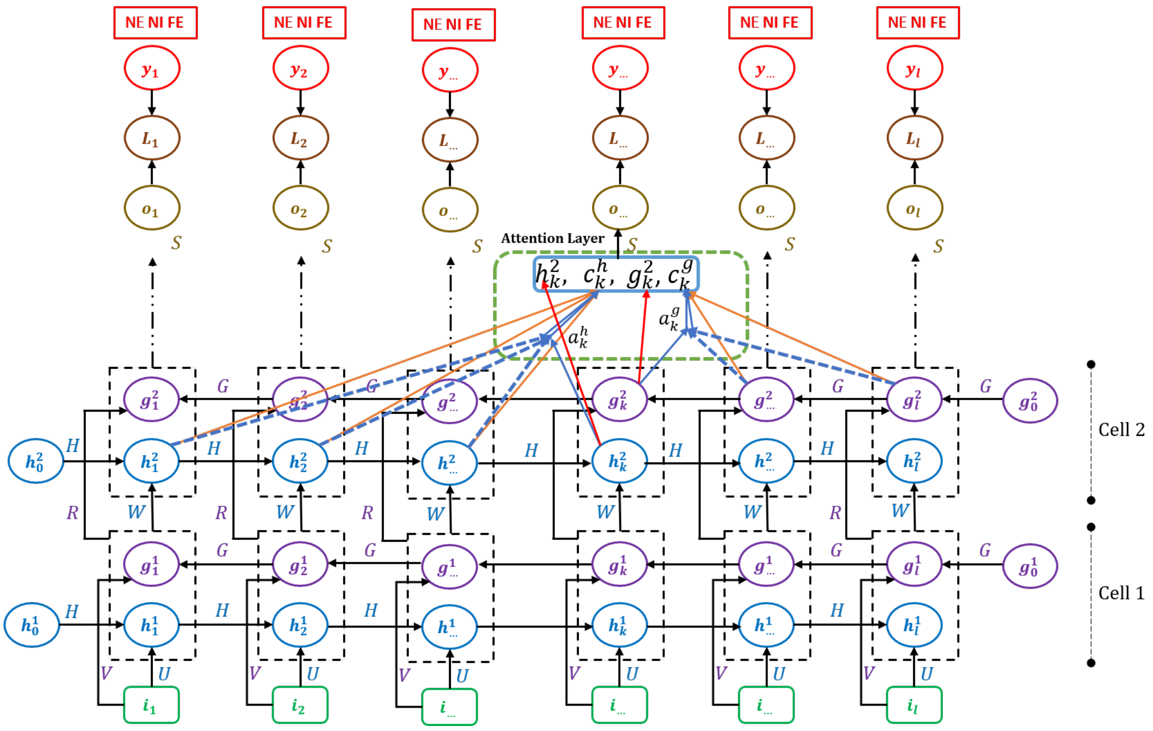


Figure 9.3: Architecture of the Code Action Network with the attention mechanism (CAN-A). The context vector $\mathbf{c}_k^h$ captures the context of the previous machine instructions $\mathbf{i}_1, ..., \mathbf{i}_{k-1}$, while the context vector $\mathbf{c}_k^g$ captures the context of the next machine instructions $\mathbf{i}_{k+1}, \ldots, \mathbf{i}_l$. As a consequence, the context of the machine instructions around the machine instruction $\mathbf{i}_k$ is taken into account when predicting the label of this machine instruction.

CAN-A is also a multicell bidirectional RNN whose architecture is depicted in Fig. 9.3 where we assume the number of cells over the input is 2. CAN-A takes a binary program $\mathbf{B} = (\mathbf{i}_1, \mathbf{i}_2, \ldots, \mathbf{i}_l)$ including l instructions (non-instructions) for CAN-AM or instruction bytes (non-instruction bytes) for CAN-AB and learns to output the corresponding sequence of action states $\mathbf{Y} = (\mathbf{y}_1, \mathbf{y}_2, ..., \mathbf{y}_l)$ where each $\mathbf{y}_k$ takes one of three action states NI (i.e., $\mathbf{y}_k = 1$), NE (i.e., $\mathbf{y}_k = 2$) or FE (i.e., $\mathbf{y}_k = 3$). The computational process of CAN-A is as follows:

$$\mathbf{h}_k^1 = \tanh(H^\top \mathbf{h}_{k-1}^1 + U^\top \mathbf{i}_k); \quad \mathbf{g}_k^1 = \tanh(G^\top \mathbf{g}_{k+1}^1 + V^\top \mathbf{i}_k)$$

$$\mathbf{h}_k^2 = \tanh\left(H^\top \mathbf{h}_{k-1}^2 + W^\top \begin{bmatrix} \mathbf{h}_k^1 \\ \mathbf{g}_k^1 \end{bmatrix}\right); \quad \mathbf{g}_k^2 = \tanh\left(G^\top \mathbf{g}_{k+1}^2 + R^\top \begin{bmatrix} \mathbf{h}_k^1 \\ \mathbf{g}_k^1 \end{bmatrix}\right)$$

$$s^h\left(\mathbf{h}_k^2, \mathbf{h}_i^2\right) = \left(v^h\right)^\top \tanh\left(W^h\left[\mathbf{h}_k^2\ \mathbf{h}_i^2\right]\right), \forall i < k$$

$$a_k^h(i) = \frac{\exp\left(s^h\left(\mathbf{h}_k^2, \mathbf{h}_i^2\right)\right)}{\sum_{j=1}^{k-1} \exp\left(s^h\left(\mathbf{h}_k^2, \mathbf{h}_j^2\right)\right)}; \qquad \mathbf{c}_k^h = \sum_{i=1}^{k-1} a_k^h(i)\, \mathbf{h}_i^2$$

$$s^g\left(\mathbf{g}_k^2, \mathbf{g}_i^2\right) = \left(v^g\right)^\top \tanh\left(W^g\left[\mathbf{g}_k^2\ \mathbf{g}_i^2\right]\right), \forall i > k$$

$$a_k^g(i) = \frac{\exp\left(s^g\left(\mathbf{g}_k^2, \mathbf{g}_i^2\right)\right)}{\sum_{j=k+1}^{l} \exp\left(s^g\left(\mathbf{g}_k^2, \mathbf{g}_j^2\right)\right)}; \quad \mathbf{c}_k^g = \sum_{i=k+1}^{l} a_k^g(i)\,\mathbf{g}_i^2$$

$$\mathbf{o}_k = S^\top \text{concat}(\mathbf{h}_k^2, \mathbf{c}_k^h, \mathbf{g}_k^2, \mathbf{c}_k^g); \quad \mathbf{p}_k = \text{softmax}\left(\mathbf{o}_k\right)$$

where $\mathbf{h}_0^1$, $\mathbf{h}_0^2$, $\mathbf{g}_{l+1}^1 = \mathbf{g}_0^1$, $\mathbf{g}_{l+1}^2 = \mathbf{g}_0^2$ are initial hidden states and $\theta = (U, V, W, H, G, R, S, W^h, W^g, v^h, v^g)$ is the model. We further note that $\mathbf{p}_k, k = 1, \ldots, l$ is a discrete distribution over the three labels NI, NE and FE.

It is worth noting that the context vector $\mathbf{c}_k^h$ captures the context of the previous machine instructions $\mathbf{i}_1, ..., \mathbf{i}_{k-1}$ and the context vector $\mathbf{c}_k^g$ captures the context of the next machine instructions $\mathbf{i}_{k+1}, \ldots, \mathbf{i}_l$, hence the context of the machine instructions around the machine instruction $\mathbf{i}_k$ is taken account of when predicting the label of the machine instruction $\mathbf{i}_k$.

To find the best model θ^*, we need to solve the following optimisation problem:

$$\max_\theta \sum_{(\mathbf{B},\mathbf{Y})\in\mathcal{D}} \log p\left(\mathbf{Y} \mid \mathbf{B}\right) \tag{9.1}$$

where $\mathcal{D}$ is the training set including pairs $(\mathbf{B},\mathbf{Y})$ of the binaries and their corresponding sequence of action states.

Appendix A

Supplementary Proofs

In this appendix, we provide formal proofs and derivations for some lemmas mentioned in Chapter 5.

A.1 Lemmas for the Information-theoretic Code Vulnerability Highlighting method

The existing work in software vulnerability detection involves detecting vulnerabilities at the program or function level. In fact, in source code only several core statements are highly relevant to a given vulnerability. In this work, we undertake vulnerability detection at a fine-grained level than at the function or program levels. In other words, we learn to emphasise the code blocks that are directly and highly relevant to the vulnerabilities. The proposed approach involves using the information-theoretic mutual information to identify the vulnerable scope. Specifically, let us consider source code $F = [\boldsymbol{f}_1, \ldots, \boldsymbol{f}_L]$ with label $y \in \{0, 1\}$ (i.e., $y = 1$ means vulnerable and $y = 0$ means non-vulnerable). Our task is to select a subset $F_S = [\boldsymbol{f}_{i_1}, \ldots, \boldsymbol{f}_{i_K}] = [\boldsymbol{f}_j]_{j \in S}$ where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$ $(i_1 < i_2 < ... < i_K)$ in such a way that F_S is highly relevant to the presence of a vulnerability. The information-theoretic quantity of interest for this aim is the mutual information: $\mathbb{I}(F_S, Y)$ where the random variable Y is characterised using $p_m(Y \mid F)$ which is previously trained using the whole training set $\mathcal{D} = \{(F_i, y_i)\}_{i=1,\ldots,N}$ wherein each $F = [\boldsymbol{f}_j]_{j=1,\ldots,L}$ consists of its code statements or machine instructions. Mathematically, we aim to solve the following optimisation problem:

$$\max \mathbb{E}_{p(F)} \left[\mathbb{E}_{p(S|F)} \left[\mathbb{I}(F_S, Y) \right] \right] \tag{A.1}$$

The interesting part is how to manipulate $\mathbb{I}\left(F_S, Y\right)$ where $F_S = [\boldsymbol{f}_{i_1}, \ldots, \boldsymbol{f}_{i_K}]$ consists of sequential data. Let $F_S = [\boldsymbol{f}_{i_1}, \ldots, \boldsymbol{f}_{i_K}] = [\boldsymbol{f}_j]_{j \in S}$ where $S = \{i_1, \ldots, i_K\} \subset \{1, \ldots, L\}$. We note that $\boldsymbol{f}_{i_{1:n}}$ is $[\boldsymbol{f}_{i_1}, \ldots, \boldsymbol{f}_{i_n}]$.

We have:

Lemma 1.

$$\mathbb{I}\left(F_S, Y\right) = \sum_{k=1}^{K} \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}\left(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}}\right)\right]$$

Proof. We have with noting that $\boldsymbol{f}_{i_{1:0}} = \emptyset$:

$$\begin{aligned}
\mathbb{I}\left(F_S, Y\right) &= \mathbb{E}_{p_m(Y, F_S)}\left[\log \frac{p_m\left(Y, F_S\right)}{p_m\left(Y\right) p_m\left(F_S\right)}\right] \\
&= \mathbb{E}_{p_m(Y, F_S)}\left[\log \frac{p_m(Y, \boldsymbol{f}_{i_K} \mid \boldsymbol{f}_{i_{1:K-1}}) \prod_{k=1}^{K-1} p_m\left(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}{p_m\left(Y\right) \prod_{k=1}^{K} p\left(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}\right] \\
&= \mathbb{E}_{p_m(Y, F_S)}\left[\log \frac{p_m(Y, \boldsymbol{f}_{i_K} \mid \boldsymbol{f}_{i_{1:K-1}}) \prod_{k=1}^{K-1} p_m\left(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}{p_m\left(Y\right) \prod_{k=1}^{K} p\left(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right) \prod_{k=1}^{K-1} p_m\left(Y \mid \boldsymbol{f}_{i_{1:k}}\right)}\right] \\
&= \mathbb{E}_{p_m(Y, F_S)}\left[\sum_{k=1}^{K} \log \frac{p_m\left(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}{\prod_{k=1}^{K}\left[p\left(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right) p_m\left(Y | \boldsymbol{f}_{i_{1:k-1}}\right)\right]}\right] \\
&= \sum_{k=1}^{K} \mathbb{E}_{p_m(Y, F_S)}\left[\log \frac{p_m\left(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}{\prod_{k=1}^{K}\left[p\left(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right) p_m\left(Y | \boldsymbol{f}_{i_{1:k-1}}\right)\right]}\right]
\end{aligned}$$

$$\begin{aligned}
\mathbb{I}\left(F_S, Y\right) &= \sum_{k=1}^{K} \mathbb{E}_{p_m\left(Y, \boldsymbol{f}_{i_{1:k}}\right)}\left[\log \frac{p_m\left(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}{p_m\left(Y | \boldsymbol{f}_{i_{1:k-1}}\right) p\left(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}\right] \\
&= \sum_{k=1}^{K} \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}\left(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}}\right)\right]
\end{aligned}$$

□

The following lemma tackles $\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}\left(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}}\right)\right]$, and this quantity can be further derived as follows.

Lemma 2. We have

$$\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}\left(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}}\right)\right] \approx \mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y | \boldsymbol{f}_{i_{1:k}}\right)}\left[log\, p_m\left(Y \mid \boldsymbol{f}_{i_{1:k}}\right)\right]\right] + const$$

Proof. We derive as follows:

$$\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{I}\left(\boldsymbol{f}_{i_k}, Y \mid \boldsymbol{f}_{i_{1:k-1}}\right)\right] = \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{E}_{p_m\left(Y, \boldsymbol{f}_{i_k} | \boldsymbol{f}_{i_{1:k-1}}\right)}\left[\log \frac{p_m\left(Y, \boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}{p_m\left(Y | \boldsymbol{f}_{i_{1:k-1}}\right) p\left(\boldsymbol{f}_{i_k} \mid \boldsymbol{f}_{i_{1:k-1}}\right)}\right]\right]$$

$$
\begin{aligned}
&\approx \mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{E}_{p_m\left(Y,\boldsymbol{f}_{i_k}|\boldsymbol{f}_{i_{1:k-1}}\right)}\left[\log\frac{p_m\left(Y,\boldsymbol{f}_{i_k}\mid\boldsymbol{f}_{i_{1:k-1}}\right)}{p_m\left(Y\mid F\right)p\left(\boldsymbol{f}_{i_k}\mid\boldsymbol{f}_{i_{1:k-1}}\right)}\right]\right]\\
&=\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{E}_{p_m\left(Y,\boldsymbol{f}_{i_k}|\boldsymbol{f}_{i_{1:k-1}}\right)}\left[\log\frac{p_m\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)p\left(\boldsymbol{f}_{i_k}\mid\boldsymbol{f}_{i_{1:k-1}}\right)}{p_m\left(Y\mid F\right)p\left(\boldsymbol{f}_{i_k}\mid\boldsymbol{f}_{i_{1:k-1}}\right)}\right]\right]\\
&=\mathbb{E}_{\boldsymbol{f}_{i_{1:k-1}}}\left[\mathbb{E}_{p_m\left(Y,\boldsymbol{f}_{i_k}|\boldsymbol{f}_{i_{1:k-1}}\right)}\left[\log\frac{p_m\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)}{p_m\left(Y|F\right)}\right]\right]\\
&=\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[\log p_m\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\right]\right]+\text{const}
\end{aligned}
$$

□

Noting that we approximate $p_m(Y\mid\boldsymbol{f}_{i_{1:k-1}})$ by $p_m(Y|F)$.

The next lemma gives a lower bound to $\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[\log p_m\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\right]\right]$ by

$$\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[\log Q\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\right]\right]$$

for every $Q\left(Y\mid F\right)$.

Lemma 3. *We can obtain a lower bound to* $\mathbb{E}_{\boldsymbol{f}_{1:k}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[log p_m\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\right]\right]$ *by*

$$\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[log Q\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\right]\right]$$

for every $Q\left(Y\mid F\right)$.

Proof. This is obvious from:

$$
\begin{aligned}
\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[\log p_m\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\right]\right]&=\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[\mathbb{E}_{p_m\left(Y|\boldsymbol{f}_{i_{1:k}}\right)}\left[\log Q\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\right]\right]\\
&\quad+\mathbb{E}_{\boldsymbol{f}_{i_{1:k}}}\left[D_{KL}\left(p_m\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\|Q\left(Y\mid\boldsymbol{f}_{i_{1:k}}\right)\right)\right]
\end{aligned}
$$

□

Bibliography

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

Alexander A. Alemi, Ian Fischer, Joshua V. Dillon, and Kevin Murphy. Deep variational information bottleneck. *arXiv preprint arXiv:1612.00410*, 2016.

Mohamed Almorsy, John Grundy, and Amani S. Ibrahim. Supporting automated vulnerability analysis using formalized vulnerability signatures. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 100–109, 2012. ISBN 978-1-4503-1204-2.

Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 214–223, 2017.

Andrea Avancini and Mariano Ceccato. Comparison and integration of genetic algorithms and dynamic symbolic execution for security testing of cross-site scripting vulnerabilities. *Information and Software Technology*, 55(12):2209–2222, 2013.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv e-prints*, abs/1409.0473, 2014.

Tiffany Bao, Jonathan Burket, and Maverick Woo. Byteweight: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium (USENIX Security 14)*, 2014.

Andrew R. Bernat and Barton P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, 2011.

Karsten M. Borgwardt, Arthur Gretton, Malte J. Rasch, Hans-Peter Kriegel, Bernhard Schölkopf, and Alex J. Smola. Integrating structured biological data by kernel maximum mean discrepancy. *Bioinformatics*, 22(14):e49–e57, 2006. ISSN 1367-4803.

Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

David Brumley, Ivan Jager, Thanassis Avgerinos, and Dward J. Schwartz. Bap: A binary analysis platform. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, 2011.

Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.

Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

Olivier Chapelle and Alexander Zien. Semi-supervised classification by low density separation. In *Proceedings of the Tenth International Workshop on Artificial Intelligence and Statistics (AISTATS)*, pages 57–64. Citeseer, 2005.

Jianbo Chen, Le Song, Martin J. Wainwright, and Michael I. Jordan. Learning to explain: An information-theoretic perspective on model interpretation. *CoRR*, abs/1802.07814, 2018.

Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20, 1994.

KyungHyun Cho, Bart V. Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *CoRR*, abs/1409.1259, 2014a.

Kyunghyun Cho, Bart V. Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2014b.

Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarth Goel, Aaron Courville, and Yoshua Bengio. A recurrent latent variable model for sequential data. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2980–2988. Curran Associates, Inc., 2015.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

Nicolas Courty, Rémi Flamary, Devis Tuia, and Alain Rakotomamonjy. Optimal transport for domain adaptation. *IEEE transactions on pattern analysis and machine intelligence*, 39(9): 1853–1865, 2017.

Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory (Wiley Series in Telecommunications and Signal Processing).* Wiley-Interscience, USA, 2006. ISBN 0471241954.

Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *OSDI*, volume 8, pages 255–266, 2008.

Hoa K. Dam, Truyen Tran, Trang Pham, Shien W. Ng, John Grundy, and Aditya Ghose. Automatic feature learning for vulnerability prediction. *CoRR*, abs/1708.02368, 2017.

Hoa K. Dam, Truyen Tran, Trang T. Pham, Shien W. Ng, John Grundy, and Aditya Ghose. Automatic feature learning for predicting vulnerable software components. *IEEE Transactions on Software Engineering*, 2018.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, pages 248–255, 2009.

Carl Doresch. Tutorial on variational autoencoders. *CoRR*, abs/1606.05908, 2016.

Mark Dowd, John McDonald, and Justin Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities.* Addison-Wesley Professional, 2006. ISBN 0321444426.

Geoffrey French, Michal Mackiewicz, and Mark Fisher. Self-ensembling for visual domain adaptation. In *International Conference on Learning Representations*, 2018.

Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059, 2016.

Yaroslav Ganin and Victor Lempitsky. Unsupervised domain adaptation by backpropagation. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1180–1189, 2015.

Seyed M. Ghaffarian and Hamid R. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):56, 2017.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS). Society for Artificial Intelligence and Statistics*, 2010.

Ian Goodfellow. Nips 2016 tutorial: Generative adversarial networks. *arXiv preprint arXiv:1701.00160*, 2016.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014a.

Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014b.

Raghuraman Gopalan, Ruonan Li, and Rama Chellappa. Domain adaptation for object recognition: An unsupervised approach. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 999–1006, 2011. ISBN 978-1-4577-1101-5.

Yves Grandvalet and Yoshua Bengio. Semi-supervised learning by entropy minimization. In *Advances in neural information processing systems*, pages 529–536, 2005.

Arthur Gretton, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. A kernel two-sample test. *Journal of Machine Learning Research*, pages 723–773, 2012a.

Arthur Gretton, Dino Sejdinovic, Heiko Strathmann, Sivaraman Balakrishnan, Massimiliano Pontil, Kenji Fukumizu, and Bharath K. Sriperumbudur. Optimal kernel choice for large-scale two-sample tests. In *Advances in Neural Information Processing Systems 25*, pages 1205–1213. 2012b.

Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. Toward large-scale vulnerability discovery using machine learning. In *Proceedings* *of the Sixth ACM Conference on Data and Application Security and Privacy*, pages 85–96, 2016. ISBN 978-1-4503-3935-3.

Caglar Gulcehre, Sungjin Ahn, Ramesh Nallapati, Bowen Zhou, and Yoshua Bengio. Pointing the unknown words. *arXiv preprint arXiv:1603.08148*, 2016.

Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. Improved training of wasserstein gans. volume abs/1704.00028, 2017.

Laune C. Harris and Barton P. Miller. Practical analysis of stripped binary code. *SIGARCH Comput. Archit. News*, 33(5):63–68, December 2005. ISSN 0163-5964.

Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

Quan Hoang, Tu Dinh Nguyen, Trung Le, and Dinh Phung. Mgan: Training generative adversarial nets with multiple generators. In *In International Conference on Learning Representation*, 2018.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9, 1997.

Ferenc Huszar. How (not) to train your generative model: Scheduled sampling, likelihood, adversary? *CoRR*, abs/1511.05101, 2015.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A scalable approach for vulnerable code clone discovery. In *IEEE Symposium on Security and Privacy*, pages 595–614. IEEE Computer Society, 2017.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2014.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems 25*, 2012.

Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, 2004.

Abhishek Kumar, Avishek Saha, and Hal Daume. Co-regularization based semi-supervised domain adaptation. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 478–486. 2010.

Michael A. Laurenzano, Mustafa M. Tikir, Laura Carrington, and Allan Snavely. Pebil: Efficient static binary instrumentation for linux. *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2010.

Trung Le, Quan Hoang, Hung Vu, Tu Dinh Nguyen, Hung Bui, and Dinh Phung. Learning generative adversarial networks from multiple data sources. In *International Joint Conference on Artificial Intelligence 2019*, 2019a.

Tue Le, Tuan Nguyen, Trung Le, Dinh Phung, Paul Montague, Olivier De Vel, and Lizhen Qu. Maximal divergence sequential autoencoder for binary software vulnerability detection. In *In International Conference on Learning Representations*, 2019b.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

Xiang Li, Jinfu Chen, Zhechao Lin, Lin Zhang, Zibin Wang, Minmin Zhou, and Wanggen Xie. A mining approach to obtain the software vulnerability characteristics. In *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pages 296–301, 2017.

Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: An automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications*, ACSAC '16, pages 201–213, 2016. ISBN 978-1-4503-4771-6.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *CoRR*, abs/1801.01681, 2018.

Zhen Li, Deqing Zou, Shouhuai Xu, Zhaoxuan Chen, Yawei Zhu, and Hai Jin. Vuldeelocator: A deep learning-based fine-grained vulnerability detector. *arXiv preprint arXiv:2001.02350*, 2020.

Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. Cross-project transfer representation learning for vulnerable function discovery. In *IEEE Transactions on Industrial Informatics*, volume 14, 2018.

Mingsheng Long, Yue Cao, Jianmin Wang, and Michael I. Jordan. Learning transferable features with deep adaptation networks. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning*, pages 97–105, 2015.

Scott M. Lundberg and Su-In Lee. A unified approach to interpreting model predictions. In *Advances in Neural Information Processing Systems*, pages 4765–4774, 2017.

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. *CoRR*, abs/1508.04025, 2015.

Laurens V. Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008.

Chris J. Maddison, Andriy Mnih, and Yee W. Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

McAfee and CSIS. There's nowhere to hide from the economics of cybercrime. 2017.

Qingkun Meng, Shameng Wen, Bin Zhang, and Chaojing Tang. Automatically discover vulnerability through similar functions. In *Progress in Electromagnetic Research Symposium (PIERS)*, pages 3657–3661. IEEE, 2016.

Takeru Miyato, Shin ichi Maeda, Masanori Koyama, and Shin Ishii. Virtual adversarial training: A regularization method for supervised and semi-supervised learning. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2018.

XNachiappan Nagappan, Thomas Ball, and Brendan Murphy. Using historical in-process and product metrics for early estimation of software failures. In *International Symposium on Software Reliability Engineering*, 2006.

Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zelle. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 529–540, 2007. ISBN 978-1-59593-703-2.

Tu D. Nguyen, Trung Le, Hung Vu, and Dinh Phung. Dual discriminator generative adversarial nets. In *In Advances in Neural Information Processing*, 2017.

Tuan Nguyen, Trung Le, Khanh Nguyen, Olivier D. Vel, Paul Montague, John Grundy, and Dinh Phung. Deep cost-sensitive kernel machine for binary software vulnerability detection. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2020a.

Van Nguyen, Trung Le, Tue Le, Khanh Nguyen, Olivier DeVel, Paul Montague, Lizhen Qu, and Dinh Phung. Deep domain adaptation for vulnerable code function identification. In *International Joint Conference on Neural Networks (IJCNN)*, 2019.

Van Nguyen, Trung Le, Tue Le, Khanh Nguyen, Olivier D. Vel, Paul Montague, John Grundy, and Dinh Phung. Code action network for binary function scope identification. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2020b.

Van Nguyen, Trung Le, Olivier D. Vel, Paul Montague, John Grundy, and Dinh Phung. Dual-component deep domain adaptation: A new approach for cross project software vulnerability detection. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, 2020c.

Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Where the bugs are. In *International Symposium on Software Testing and Analysis*, 2004.

Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *CoRR*, abs/1211.5063, 2013.

Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*, pages 547–553. Springer, 2015.

Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

Aravind Prakash, Xunchao Hu, and Heng Yin. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *National Diabetes Services Scheme (NDSS)*, 2015.

Sanjay Purushotham, Wilka Carvalho, Tanachat Nilanon, and Yan Liu. Variational recurrent adversarial deep domain adaptation. In *International Conference on Learning Representations (ICLR)*, 2017.

Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*, 2017.

Ievgen Redko, Amaury Habrard, and Marc Sebban. Theoretical analysis of domain adaptation with optimal transport. *CoRR*, abs/1610.04420, 2016.

Marco T. Ribeiro, Sameer Singh, and Carlos Guestrin. Why should i trust you?: Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144. ACM, 2016.

Nathan E. Rosenblum, Xiaojin Zhu, Barton P. Miller, and Karen Hunt. Learning to analyze binary computer code. In *AAAI*, pages 798–804, 2008.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning internal representations by error backpropagation. In *Parallel Distributed Processing*, volume 1, 1986.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *CoRR*, abs/1409.0575, 2014.

Hasim Sak, Andrew W. Senior, and Fran**c**coise Beaufays. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition. *CoRR*, abs/1402.1128, 2014.

Shibani Santurkar, Ludwig Schmidt, and Aleksander Madry. A classification-based study of covariate shift in GAN distributions. In *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 4480–4489. PMLR, 10–15 Jul 2018.

Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 11–20. IEEE, 2015.

Mike Schuster and Kuldip K. Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 1997.

Jian Shen, Yanru Qu, Weinan Zhang, and Yong Yu. Wasserstein distance guided representation learning for domain adaptation. *CoRR*, abs/1707.01217, 2018.

Eui C. R. Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *24th USENIX Security Symposium (USENIX Security 15)*, 2015.

Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

Avanti Shrikumar, Peyton Greenside, and Anshul Kundaje. Learning important features through propagating activation differences. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3145–3153. JMLR. org, 2017.

Rui Shu, Hung H. Bui, Hirokazu Narui, and Stefano Ermon. A DIRT-t approach to unsupervised domain adaptation. In *International Conference on Learning Representations*, 2018.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550, 2017.

Noam Slonim and Naftali Tishby. Agglomerative information bottleneck. In *Advances in neural information processing systems*, pages 617–623, 2000.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, 2014a.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014b.

Naftali Tishby and Noga Zaslavsky. Deep learning and the information bottleneck principle. In *2015 IEEE Information Theory Workshop (ITW)*, pages 1–5. IEEE, 2015.

Naftali Tishby, Fernando C. Pereira, and William Bialek. The information bottleneck method. *arXiv preprint physics/0004057*, 2000.

Du Tran, Lubomir D. Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. Learning spatiotemporal features with 3d convolutional networks. *CoRR*, abs/1412.0767, 2015.

Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. a correction. In *Proceedings of the London Mathematical Society*, 1938.

Eric Tzeng, Judy Hoffman, Trevor Darrell, and Kate Saenko. Simultaneous deep transfer across domains and tasks. *CoRR*, 2015.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

Cédric Villani. *Optimal Transport: Old and New*. Springer, 2008.

Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391*, 2015a.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. In C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems 28*, pages 2692–2700. Curran Associates, Inc., 2015b.

Yang Wang, Gholamreza Haffari, Shaojun Wang, and Greg Mori. A rate distortion approach for semi-supervised conditional random fields. In *Advances in Neural Information Processing Systems*, pages 2008–2016, 2009.

David H. White and Gerald Lüttgen. Identifying dynamic data structures by learning evolving patterns in memory. In *TACAS*, pages 354–369. Springer, 2013.

Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 13–23, 2011.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. ISSN 0362-1340.

Ting Yao, Yingwei Pan, Chong-Wah Ngo, Houqiang Li, and Tao Mei. Semi-supervised domain adaptation with subspace learning for visual recognition. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 00, pages 2142–2150, June 2015.

Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

Mingwei Zhang and R Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, 2013.

Xiaojin Zhu, Andrew B. Goldberg, Ronald J. Brachman, and Thomas Dietterich. *Introduction to Semi-Supervised Learning*. Morgan and Claypool Publishers, 2009. ISBN 1598295470, 9781598295474.

Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, 2009. ISBN 978-1-60558-001-2.