Australian Government

Department of Defence

Science and Technology

Deep Learning for Cyber Vulnerability Discovery: NGTF Project Scoping Study

de Vel O. [1], *Hubczenko D.* [1], *Kim J.* [1], *Montague P.* [1],
Xiang Y. [2], *Phung D.* [3], *Zhang J.* [3], *Murray T.* [4],
Le T. [3], *Wen S.* [2], *Liu S.* [2], *Nguyen V.* [3],
Lin G. [3], *Nguyen K.* [3], *Le T.* [3], *Nguyen T.* [3], *Nock R.* [5], *Qu L.* [5]

[1] **Cyber & Electronic Warfare Division**

[2] **School of Software and Electrical Engineering, Swinburne University**

[3] **Centre for Pattern Recognition and Data Analytics, Deakin University**

[4] **School of Computing and Information Systems, University of Melbourne**

[5] **Data61, CSIRO**

Defence Science and Technology Group

DST-Group–GD–1039

ABSTRACT

This report is the result of a scoping study undertaken as part of an Australian Defence Department Next Generation Technologies Fund (NGTF) project entitled *Deep Learning for Cyber-Security* (DLC). The report provides a motivation for the study of software vulnerability discovery, briefly reviewing existing techniques for both source and binary code, with an emphasis on machine learning approaches. Noting the spectacular successes in recent years of Deep Learning (DL) techniques in areas such as image recognition, it is proposed to investigate the application of DL techniques to the software vulnerability discovery problem, with a focus on binary code analysis as most relevant to Defence. As part of this effort, consideration is given to the acquisition and generation of suitable training and testing datasets.

RELEASE LIMITATION

For Public Release.

Produced by

Cyber and Electronic Warfare Division
PO Box 1500
Edinburgh, South Australia 5111, Australia

Telephone: 1300 333 362

May, 2019

APPROVED FOR PUBLIC RELEASE

Deep Learning for Cyber Vulnerability Discovery: NGTF Project Scoping Study

Executive Summary

The rapid evolution and growth in scale and complexity of malicious cyber capabilities presents an ever increasing challenge for cyber-security. Rather than react/respond to discovered attacks, a proactive approach is to concentrate on preventative measures through the discovery of software vulnerabilities, particularly in mission-critical applications and systems. Discovered vulnerabilities may be mitigated before they can be actively exploited.

There has been some work in recent years on investigating the use of machine learning (ML) techniques in order to assist software vulnerability discovery. Motivated by the spectacular success of deep learning (DL) approaches in fields such as computer vision and natural language processing, we propose to study the application of DL techniques to software vulnerability discovery. DL approaches have the ability to learn feature representations and complex non-linear structures in datasets exhibiting hierarchies of patterns at fine to coarse scales, and there is every reason to suspect that they may continue to enjoy similar success in the software vulnerability discovery domain. Primary focus will be on the analysis of binary code vulnerabilities as this is of most relevance to Defence, though source code approaches will also be considered.

This report is a scoping study undertaken in the context of an Australian Department of Defence (DoD) Next Generation Technologies Fund (NGTF) project entitled *Deep Learning for Cyber-Security* (DLC).* The project will investigate concepts, techniques and technologies relating to the application of deep learning algorithms to the discovery of software vulnerabilities. An initial focus will be on generation of suitable datasets of known vulnerabilities for training and testing of the techniques under study.

The NGTF DLC project scoping study sought to to:

1. Ground the research programme design in the existing DL and cyber-security literature, so that future work builds on a firm foundation of existing knowledge
2. Develop appropriate datasets of known vulnerable code for both training and testing of DL techniques
3. Develop deep learning based vulnerability discovery techniques and technology solutions
4. Detail a plan of work packages, to ensure appropriate long-term project outcomes

In particular, the DLC project seeks to deliver:

- Ground truth datasets consisting of labelled source/binary codebases
- New DL methods for binary and source code based vulnerability discovery

*The DLC project is supported by the Australian DoD NGTF and contribution of personnel from CEWD-DST Group and Data61.

- Prototypes of DL based software vulnerability discovery tools

The DLC research team consists of members from CEWD-DST Group, Data61, the University of Melbourne, Deakin University and Swinburne University.

Contents

1 INTRODUCTION 1

2 LITERATURE REVIEW 2
2.1 Binary code based methods 2
2.1.1 Binary related analysis with ML 3
2.1.2 Deep learning for binary code analysis 5
2.2 Source code based methods 5
2.2.1 ML-based source code analysis 6
2.2.2 Traditional source code analysis 6
2.3 Lessons learned 7

3 EXPECTED DELIVERABLES 8

4 RESEARCH AND DEVELOPMENT OF GROUND-TRUTH 8
4.1 Collecting existing datasets 8
4.2 Labelling real-world binary code 9
4.3 Dataset augmentation 10
4.4 Ground-truth related work package 10

5 SOFTWARE VULNERABILITY DISCOVERY 11
5.1 Binary code based vulnerability discovery 11
5.1.1 Vulnerability discovery from byte sequences 11
5.1.2 Vulnerability discovery from assembly code 14
5.2 Source code based vulnerability discovery 15
5.2.1 Learning graph representations 15
5.2.2 Building graph-based vulnerability detectors 16
5.2.3 Tackling problems of limited training data 16
5.2.4 Challenges of the approaches based on source code 17
5.3 Multiple source based vulnerability discovery 17
5.4 The main tasks in the vulnerability detection work package 18

6 PROJECT PLANS 18

Figures

1 Vulnerabilities by type [Source: http://www.cvedetails.com/vulnerabilities-by-types.php]. 3
2 The information extracted from the debug file using Microsoft's Debug Interface Access SDK. 9
3 An example of code augmentation. The original code is vulnerable to the *CWE-839: Numeric Range Comparison Without Minimum Check*. The sample code only checks if the given array index is less than the maximum length of the array, but it does not verify the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index. This will result in an out-of-bounds read (CWE-125) and offer attackers a way of accessing sensitive memory. 10

4 The tasks and sub-tasks in the ground-truth related work package, WP1. 11
5 Binary file represented as a sequence of bytes (orange) or as a sequence of machine instructions (green). 13
6 Top: Training the 1D RNNs and classifier sequentially. Bottom: Training the 1D RNNs and classifier simultaneously, where the classifier parameter $\boldsymbol{\theta}_c$ is shared among the 1D RNNs. 20
7 Two functions that each suffers from a buffer overflow vulnerability. There is a common localised pattern in their respective source code hence, when compiled to binary code and then visualized as images, the resulting images share a local 'pixel' pattern that can be efficiently detected using CNN(s). We note that each 4-byte instruction is visualized as one pixel using the CMYK colour system. 21
8 A binary file with three functions: Function 1.x (multiple sections), Function 2, and Function 3. . 21
9 Encoder-decoder architecture with attention mechanism for function scope identification. The orange rectangles denote the hidden states of the encoder, while the green ones are the hidden states of the decoder. 22
10 Example binary code and the resulting assembly code after decompilation. 22
11 Overview of the assembly code based approaches. The assembly code derived from the binary code can be viewed either as a graph (graph view [1]) or a collection of functions (function view [2]). 23
12 Example control flow graph (CFG) based on the assembly code approach. Left: The full control flow graph which is constructed based on the entire assembly file. Right: The example section of the control flow graph. We note that this control flow graph shows the hierarchical, nested, and semantic structure of the binary file. Each block in this control flow graph can be further used as input to deep learning methods to detect vulnerabilities at the block level. 23
13 The encoder-decoder architecture for function scope identification in assembly code. Tokens are embedded into a vector space and $\mathrm{e}(\mathrm{token})_i$ denotes a token embedding. 24
14 Example of assembly code with three functions inside. 24
15 The execution diagram after decompiling the binary code to source code. 25
16 The overall workflow of *AST2Vec* (adapted from [25]). 25
17 The proposed function-level RNN for software vulnerability discovery. 26
18 The tasks in the vulnerability detection work package WP2. 26
19 The plan for the tasks in the *Deep Learning for Cyber project*. 27

1. Introduction

Computer software is a crucial part of the modern world. In a routine day, users depend on myriad software components running on different platforms and ranging from simple applications on hand-held devices, to complicated enterprise software, to critical embedded systems. Coupled with software's ubiquity, security vulnerabilities remain rife, and so vulnerability discovery is a critical concern. Following [14], we define a software vulnerability as follows:

> *In the context of software security,* vulnerabilities *are specific flaws or oversights in a piece of software that allow attackers to do something malicious: expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program.*

The impact of software vulnerabilities has increased each year. Numerous examples and incidents exist in the past two decades in which software vulnerabilities have caused significant damage to companies and individuals [20]. Examples include vulnerabilities in popular browser plugins that have threatened the security and privacy of millions of internet users (e.g., Adobe Flash Player (US-CERT 2015; Adobe Security Bulletin 2015) and Oracle Java (US-CERT 2013)), as well as vulnerabilities in popular and fundamental open-source software that have threatened the security of thousands of companies and their customers around the globe (e.g., Heartbleed (Codenomicon 2014), ShellShock (Symantec Security Response 2014), and Apache Commons (Breen 2015)). At the same time, the number of software vulnerabilities has also increased (see Table 1 and Figure 1).

We seek to investigate the application of advanced methods in machine learning (ML) to accurately and efficiently detect vulnerabilities in large code bases.

The rapid rise of deep learning is in part due to its ability to learn feature representations and complex non-linear structure in datasets. Deep learning has achieved particular successes in data domains such as vision, speech and natural language, which each exhibit hierarchies of patterns at fine to coarse scales. Cyber-security would appear to be amenable to a similar approach, owing to its complex, hierarchical, non-linear detection tasks. We propose that deep learning techniques can be used to model the structural and semantic features related to the presence of vulnerabilities. Applying deep learning for software vulnerability discovery may therefore address the following challenges related to Vulnerability Analysis:

- *Efficiency*: To accurately detect and locate vulnerabilities in code is expensive, tedious and time-consuming. With software having increasingly advanced functions, the number of lines of code contained in software has grown dramatically.

- *Effectiveness*: To specifically identify security vulnerabilities in software is very difficult. It requires a professional (e.g., code auditor) to have not only a deep understanding of the code, but also be equipped with security-related skills. Consequently, there are often security vulnerabilities reported after the software has been in operation for several years.

Despite valuable advances made in recent years, application of ML for cyber-security is still at an early stage. Obstacles that are particularly problematic for applying ML to cyber-security are the large, complex and mixed datasets. In this project, we will address some key challenges by means of suitable innovations:

- To date, the vast majority of vulnerability discovery is based on manual code analysis by security experts. In this proposal, we focus on combining code analysis and machine learning techniques to facilitate intelligent vulnerability discovery. We will investigate new feature learning techniques, which can be applied to transferring or converting known code vulnerabilities to indicators.

- Original high-level source code may be unavailable. Source code could be lost, or inconvenient to use. Thus, it is critical to conduct the research using binary code as the data source. Inspired by the good performance of deep neural networks in various domains, we plan to employ deep neural networks to assist binary code analysis. To boost the research of deep learning for vulnerability discovery, we will establish our own ground-truth datasets using open-source code repositories and the Common Vulnerabilities and Exposures (CVE) system.

Table 1: The statistics of vulnerabilities by types [Source: http://www.cvedetails.com/vulnerabilities-by-types.php] .

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	616	177	112	172			2	7		25	16	103			2
2000	903	257	208	206		2	4	20		48	19	139			
2001	1610	403	403	297		7	34	123		83	36	220		2	2
2002	2249	498	553	435	2	41	200	103		127	74	199	2	14	1
2003	1766	381	477	371	2	49	129	60	1	62	69	144		16	5
2004	2591	580	614	410	3	148	291	110	12	145	96	134	5	38	5
2005	5647	838	1627	657	21	604	786	202	15	289	261	221	11	100	15
2006	8584	893	2719	663	91	967	1302	322	8	267	271	184	18	849	30
2007	8338	1101	2601	953	95	706	884	339	14	267	323	242	69	700	44
2008	7382	894	2310	699	128	1101	807	363	7	288	270	188	83	170	74
2009	8103	1035	2185	700	188	963	851	322	9	337	302	223	115	138	735
2010	7652	1102	1714	680	342	520	605	275	8	234	282	238	86	73	1493
2011	5996	1221	1334	770	351	294	467	108	7	197	409	206	58	17	557
2012	7068	1425	1458	843	423	242	758	122	13	343	389	250	166	14	615
2013	6254	1454	1186	859	366	156	650	110	7	352	511	274	123	1	205
2014	9535	1598	1574	850	420	305	1105	204	12	457	2104	239	264	2	401
2015	8671	1792	1825	1079	749	217	776	149	12	577	748	367	248	5	127
2016	8256	2029	1494	1326	717	94	497	99	15	446	843	601	87	7	1
2017	14602	3073	2900	2688	721	464	1444	265	10	612	1640	454	309	18	4
Total	115823	20751	27295	14658	4619	6881	11592	3303	150	5156	8663	4626	1644	2164	4321
% Of All		17.9	23.6	12.7	4.0	5.9	10.0	2.9	0.1	4.4	7.5	4.0	1.4	1.9	3.7

2. Literature Review

2.1. Binary code based methods

Given that many software projects are closed-source, binary code based vulnerability discovery is very important. In addition, compilation and optimisation may alter the structure of the compiled program (e.g. by reordering independent operations, eliminating dead code, or by taking advantage of undefined behaviour in the source program) creating a mismatch between the source code and the compiled binary code [5]. Such optimisations can even introduce additional security vulnerabilities. As a canonical example the act of zeroing a no-longer-used buffer which contained sensitive information, to prevent it being read by attackers, may be optimised away by certain compilers [15]. In this respect, purely

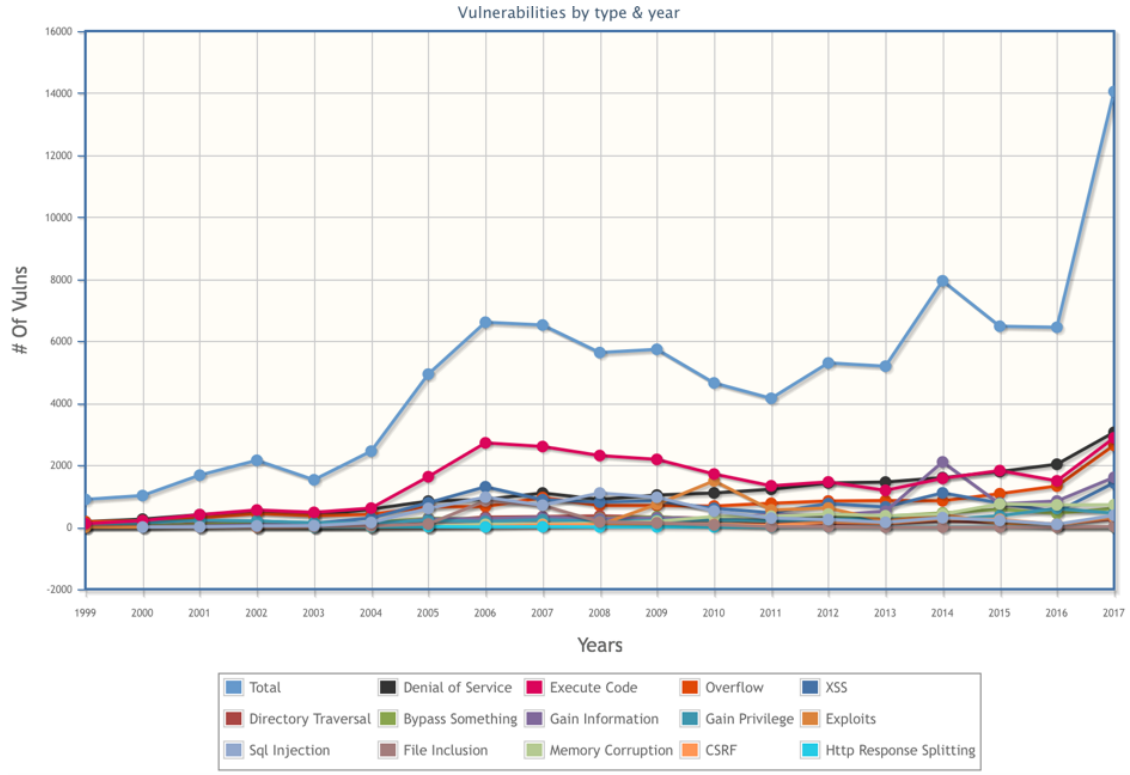

Figure 1: Vulnerabilities by type [Source: http://www.cvedetails.com/vulnerabilities-by-types.php].

performing analysis on the source code level may fail to detect certain vulnerabilities [18]. This section briefly reviews some recent work on binary code based methods for software vulnerability discovery.

2.1.1. Binary related analysis with ML

Among existing methods for binary analysis, machine learning-based methods are beginning to play an important role.

Abraham and de Vel [1] survey recent work applying machine learning for software vulnerability discovery. This survey reviewed the related works from several perspectives including data structures [12, 58], program structures [7], dynamic analysis [23], symbolic execution [10, 3, 32] as well as applications such as malware detection [46, 19]. Another similar survey was published concurrently in [20]. In the following, we review the research reported since [1] was published.

Morrison et al. [36] highlighted that the Windows security development team usually raises concerns about the usefulness of vulnerability prediction models. Morrison et al. examined whether the vulnerability prediction models are accurate and actionable enough to provide helpful recommendations when allocating engineering resources. The authors replicated the vulnerability prediction model of Zimmermann et al. [63] for two releases of the Windows operating system. They conducted experiments using several traditional machine learning algorithms such as Logistic Regression (LR), Naive Bayes (NB), Recursive Partitioning (RP), Support Vector Machine (SVM), Tree Bagging (TB) and Random Forest (RF). The experimental results demonstrate that the recall and precision performance measures of the binary-level prediction outperform the file-level prediction performance measure. However, the binaries are usually too large to be used for practical inspection. The file level prediction performance

measure is preferred by security experts such as code auditors because they can check source code easily. In this respect, the authors concluded that "Vulnerability Prediction Models must be refined to achieve actionable performance possibly through security-specific metrics".

Shar and Tan [49] argued that SQL injection (SQLI) and cross site scripting (XSS) are two common and critical web application vulnerabilities. However, existing prediction models usually focus on general vulnerabilities, and operate at the granularity of software components or files. Some methods also involve process attributes that are often difficult to measure. To address these problems, the authors propose a set of statistical code attributes as an alternative solution to existing taint analysers. Experiments using C4.5, NB and MLP were conducted to evaluate the proposed method.

Yamaguchi et al. [61] described that many approaches of software vulnerability discovery try to construct a large database of vulnerability patterns, which can subsequently be used to search vulnerable code easily. However, the construction of effective search patterns for known vulnerabilities requires a security expert to invest a considerable amount of manual work. It requires the expert to identify related input sources, data flows and corresponding sanitization checks, which usually involves a profound understanding of project-specific functions and interfaces. To address this issue, Yamaguchi et al. developed a scheme that can automatically infer search patterns for taint-style vulnerabilities from C source code. For a sensitive sink, such as a memory or network function, the proposed method combines static program analysis and unsupervised learning techniques to automatically identify corresponding source-sink data flows in a code base and infer sanitisation patterns from which search patterns can be generated to detect potential taint-style vulnerabilities (in which the sanitisation is absent). Experiments have been conducted showing that the method reduces the amount of code to audit by 94.9% on average.

Alves et al. [2] have evaluated several state of the art vulnerability prediction techniques using a large and representative data set. The data set is composed of 2186 vulnerabilities from five widely-used open source projects. Experiments have been conducted using both decision trees and logistic regression. The experimental results vary quite significantly. The precision values ranged from 0.32% to 30.05%, while recall values ranged from 0.36% to 100%. The random forest algorithm achieved very good results in terms of any performance metric.

Eschweiler et al. [18] emphasised the importance of security-critical vulnerability detection in binary code. They presented a new approach to efficiently search for similar functions in binary code. The main idea of their work is to compute the similarity between functions based on the structure of the corresponding control flow graphs. They proposed a set of code features including structural features and numeric features, which do not vary a lot across different compilers, optimization options, operating systems, and CPU architectures. Experimental results showed that structural information is sufficiently robust for identifying function similarity. However, the method is computationally expensive and impractical. To overcome this issue, numeric features have been used to represent the meta information about the function. These numeric features are then embedded in a vector space, where machine learning techniques can be applied to quickly identify a set of features for checking structural similarity. The proposed strategy has been implemented in a tool called discoRE. Experiments have been conducted on real-world firmware images with up to 3 million basic blocks for performance evaluation. The results showed that discoRE was able to outperform state-of the-art semantic approaches w.r.t. speed while still maintaining the same predictive quality (93.93%). It could also could correctly identify buggy functions from ARM, MIPS and x86 architectures (i.e., for three

firmware images) in about 80 ms.

2.1.2. Deep learning for binary code analysis

We report on the existing work where deep learning has been used to recognize software vulnerabilities in binary code. In this subsection, we also include some of the published works on the application of deep learning to malware detection. These are relevant to this project from a technical point of view.

Grieco et al. [23] developed a machine learning-based approach that uses lightweight static and dynamic features for memory corruption vulnerability analysis over binary code. Both static and dynamic features are extracted from binary programs. The static features extracted are the sequences of calls to the standard C library functions in the program. The dynamic features are obtained by analysing program execution traces that contain concrete function calls and arguments. The authors' approach treats each call trace as a text document. Two text-mining techniques using N-grams and word2vec, are employed for data vectorisation. Random oversampling is used to address the class imbalance problem and the dropout training technique is applied to avoid overfitting. The results of this work show that machine learning techniques have potential to significantly increase the number of vulnerabilities found at the operating system scale.

Saxe and Berlin [47] argued that machine learning holds the promise of automating the work required for malware detection. It could potentially learn generalisations about malware and benign software (non-malware). However, existing machine learning-based malware detection methods cannot achieve the low false positive rates and high scalability required to deliver deployable detectors. Saxe and Berlin developed a deployable deep neural network-based malware detector using static features. The method proposes four kinds of features including contextual byte features, portable executable (PE) import features, string 2-D histogram features as well as PE metadata features. It uses a classification model based on deep neural networks and a score calibration model. The deep learning-based malware detection method achieved a detection rate of 95% and a false positive rate of 0.1% in the experiments that were conducted on a dataset of more than 400,000 software binaries.

Raff et al. [43] proposed a static analysis approach for malware detection based on raw byte sequences. They investigated the raw bytes of the file itself and built a neural network to determine maliciousness. A convolutional network architecture was employed to capture more sophisticated spatial correlation in the code file. There were three special considerations: 1) the ability to scale with the sequence length, 2) the ability to consider both the local and global contexts while examining an entire file, as well as 3) an explanatory ability to aid analysis of flagged malware. The network architecture was able to successfully process a raw byte sequence of over two million steps. One of the primary limitations was the GPU memory consumption in the first convolutional layer. It required aggressive pooling of data between layers when attempting to build deep architectures with such long sequences. This resulted in lopsided memory use, and made model parallelism in computational frameworks like TensorFlow difficult to achieve.

2.2. Source code based methods

Traditional source code analysis techniques can be categorised into three classes: *static* analysis techniques that do not execute the program being analysed; in contrast to *dynamic* techniques that do, while *hybrid* techniques that combine both static and dynamic analysis. In this section, we review ML-based vulnerability detection at the source code level. Some basic knowledge of source code analysis is also provided for completeness. For further details we refer the reader to [20, 1].

2.2.1. ML-based source code analysis

Software vulnerability detection in source code is difficult since vulnerabilities are rare compared to other types of software defects. For instance, the famous Heartbleed vulnerability was caused by two missing lines of code. In order to find software vulnerabilities, many static analysis and dynamic analysis tools have been developed.

Recently, machine learning techniques have been introduced to build up software vulnerability discovery models. Statistical features of software code [51] such as software metrics (e.g., size of code, number of dependencies, cyclomatic complexity), code churn metrics (the number of code lines changed) as well as developer activity statistics are used as the input for ML-based software vulnerability detection. However, these features do not have good discriminative capability in terms of semantics. For example, two pieces of code may have the same complexity metrics, but they behave differently. Another disadvantage is that the statistical features are created by knowledgeable domain experts, which may carry outdated experience and underlying biases. In addition, some handcrafted features may not generalise well, which means features that perform well in one project may not perform well in other projects [62].

To overcome these problems, some researchers treat software code as a form of text and leverage natural language processing (NLP) techniques to extract features automatically. Scandariato et al [48] used the bag-of-words (BoW) representation, in which the software source component is seen as a series of terms with associated frequencies. Their research showed that it is possible to build a good quality classifier that can predict whether a file is vulnerable using term frequencies.

Hoa et al. [13] argued that there are two weaknesses to the bag-of-words (BoW) method. One weakeness is that the BoW method ignores the semantics of code tokens. For example, it fails to recognise the semantic relation between ‘for’ and ‘while’. The other weakeness is that the BoW method cannot capture ordering information between source code tokens, and so misses much meaningful information. Hoa et al. developed a novel deep learning-based approach that can automatically learn features for predicting vulnerabilities in source code. In their approach, an LSTM was employed to capture the long context relationships in source code where dependent code elements are scattered far apart. The results of within-project and cross-project prediction demonstrated that the automatically-learned features can significantly improve the classification performance compared to the traditional methods using software metrics.

2.2.2. Traditional source code analysis

Traditional static analysis techniques usually operate over the program’s source code to uncover faults and vulnerabilities, and include methods such as the following.

1. **Rule-based Analysis**: This technique detects faults and vulnerabilities as violations of programming rules and patterns (e.g. attempts to free the same dynamically allocated block of memory more than once, which can lead to double-free type vulnerabilities).

2. **Symbolic Execution**: Although rule-based analysis can discover general faults, it fails to detect sophisticated bugs and is subject to reporting false positives. The symbolic execution technique can unveil deep bugs by exploring all possible execution paths of a program using symbolic inputs, while being able to generate candidate inputs to trigger found faults. Therefore, symbolic execution can generate test suites with high coverage [9, 45]. Modern symbolic execution tools

like KLEE operate over an intermediate program representation, a code format that resides in between source code and its compiled binary, and thus provides a trade-off between the benefits and drawbacks of both source and binary analysis techniques.

Dynamic analysis techniques identify faults and vulnerabilities that manifest themselves at run time.

1. **Taint Analysis**: Taint analysis (a.k.a. taint tracking) monitors the information flow during the execution of a program, to detect, for example, when attacker-supplied data can influence sensitive operations or when confidential data is revealed to an attacker. It can also be used for analysing commodity and legacy programs [37, 38]. Taint analysis has been applied to vulnerability discovery, and is capable of spotting errors in the code such as buffer overruns and format string vulnerabilities [38]. It has been particularly effective at finding vulnerabilities due to improper validation of untrusted data [26] in which, when the necessary validation is absent, a possible vulnerability is detected [60].

2. **Fuzz Testing**: Fuzz testing, or simply 'fuzzing', is a simple yet effective testing method for vulnerability discovery [54]. Fuzzing is also frequently leveraged by attackers for the same purpose [39]. Fuzzing focuses on intentionally sending malformed data to the examined program. Once an exception or error condition is triggered (usually causing the program to crash), the corresponding input will be recorded and the resulting error can be manually analysed to determine whether it represents a vulnerability or not.

Combining static and dynamic analysis techniques for fault and vulnerability discovery is an active research area. The combination approach can achieve better detection performance at the expense of longer computation times and larger resource requirements[17].

2.3. Lessons learned

Binary code based analysis is as important as source code based analysis for software vulnerability discovery.

Recent research shows that simply combining static and dynamic features for software vulnerability discovery could result in poor results [23]. One possible reason is that the training process is affected by the static features. The static features are not as diverse compared with dynamic ones even though they are shared by all the traces of the same program.

The application of deep-learning (DL) techniques to create powerful cross-project software vulnerability discovery models is a promising idea [20]. As far as we know, relatively little work has been done applying deep learning techniques to discover security-critical vulnerabilities at the binary code level. Peng et al. presented a pioneering work using DL for program classification [42], which could potentially be useful for this project. They proposed a deep learning approach to generate a graph-based embedding space for recognising similar binary code functions. We expect that their embedding technique can be extended to vulnerability discovery for software binaries. Recognising function similarity might also be extended to perform vulnerability discovery within specific vulnerability classes. Other recent work embodies similar ideas [59].

There are some other interesting works related to this project. For example, the idea of DL-based function recognition in binary code [23] [50] could be incorporated into our future research of vul-

nerable function discovery. The idea of deep learning over raw byte sequences for malware detection [43] can be borrowed and extended to recognise vulnerable byte sequences. Later, we will describe a new idea for software vulnerability discovery, which combines embedding and 2D CNNs to process raw byte sequences. How to combine feature engineering and deep representation learning is another interesting direction for software vulnerability discovery. Some new features reported in the literature [47], such as byte/entropy histogram features, PE import features, and PE metadata features, can be embedded into neural networks to support better learning.

Lastly, an important fact is that the number of vulnerable functions is likely to be much smaller than the number of non-vulnerable functions in real-world software. We can develop new methods to address this vulnerability imbalance problem based on the existing research on data redistribution [43]. For example, we can employ fuzzy-based information decomposition to re-balance the training data by generating synthetic minority samples [31].

3. Expected Deliverables

The *Deep Learning for Cyber Project* will produce the following key deliverables. The detailed project plan is presented in Section 6.

- **Deliverable 1 (Ground-Truth datasets):** The generation of labelled binary code and source code datasets. These datasets will be used as inputs for developing methods for vulnerability discovery, function identification, and neural alignment (e.g., learning to align binary instructions with the source code from which they were generated).

- **Deliverable 2 (Theory and DL methods):** The creation of new theory and deep learning methods for binary and source code-based vulnerability discovery; new methods of multiple source analysis; new methods of neural alignment and program embedding.

- **Deliverable 3 (Tools and prototypes):** The development of new deep learning tools and software vulnerability discovery prototypes.

- **Deliverable 4 (Publications and reports):** Production of high-quality international conferences/journals publications coauthored with team members from DST Group and Data61; Technical reports and presentations to communicate project progress.

4. Research and Development of Ground-Truth

One crucial goal of this project is to create labeled source code and binary code datasets for software vulnerability research. We plan to label source code and binary code at the function level. The idea is to collect open source code samples and use the CVE framework for labelling software vulnerability information. In addition, we can benefit from the existing datasets and code generation techniques to augment the quantity and quality of the labelled code data. We note that interprocedural cases where vulnerabilities span across function boundaries are regarded as out of scope for our approaches.

4.1. Collecting existing datasets

We summarise the existing datasets that could be used for our project in Table 2. It is worth noting that all of the listed datasets are in source code form, from which corresponding binary datasets can be generated.

4.2. Labelling real-world binary code

It is of utmost importance to construct labeled binary code datasets. In these datasets, we need to annotate sequences of bytes or machine instructions in functions (namely, specify the function to which a byte or machine instruction belongs) and simultaneously label software vulnerabilities associated with these sequences. We note that the first type of label (i.e., scope accompanied with sequences of bytes or machine instructions of a function) is used in the function identification task (See Sections 5.1.1 and 5.1.2). Our key idea is to compile source codes in debug mode and then utilize the resulting debugging information for identifying function scopes.

When compiling source code to binary in debug mode, the compiler generates additional debug information (e.g. a .pdb file in Microsoft Windows) that contains information relating the generated binary back to the original source files that were compiled. This information can be used to identify function scopes in the binary code. For example, using Microsoft's Debug Interface Access SDK[1], we can read out the debug information as shown in Figure 2. The advantage of this approach is that the debugging information is almost guaranteed to be accurate because it is produced directly by the compiler.

```
*** SYMBOLS
** Module: * CIL *

CompilandEnv    : obj = "* CIL *"
CompilandDetails:
    Language: LINK
    Target processor: 80386
    Compiled for edit and continue: no
    Compiled without debugging info: no
    Compiled with LTCG: no
    Compiled with /bzalign: no
    Managed code present: no
    Compiled with /GS: no
    Compiled with /sdl: no
    Compiled with /hotpatch: no
    Converted by CVTCIL: no
    MSIL module: no
    Frontend Version: Major = 0, Minor = 0, Build = 0, QFE = 0
    Backend Version: Major = 14, Minor = 10, Build = 25019, QFE = 0
    Version string: Microsoft (R) LINK

Function       : static, [00001040][0001:00000040], len = 0000004D, void __cdecl admin_panel(void)
                 Function attribute:
                 Function info:
FuncDebugStart :   static, [00001041][0001:00000041]
FuncDebugEnd   :   static, [0000108C][0001:0000008C]
Callee         :   _printf(printf) has type function
Callee         :   _printf(printf) has type function
Callee         :   _printf(printf) has type function
Callee         :   _printf(printf) has type function
Callee         :   _printf(printf) has type function
CallSite       :   [0x0001:0x00000085]  0x00001085  int ()


Function       : static, [00001090][0001:00000090], len = 00000047, void __cdecl login_fail_panel(void)
                 Function attribute:
                 Function info:
FuncDebugStart :   static, [00001091][0001:00000091]
FuncDebugEnd   :   static, [000010D6][0001:000000D6]
Callee         :   _printf(printf) has type function
Callee         :   _printf(printf) has type function
Callee         :   _printf(printf) has type function
Callee         :   _printf(printf) has type function
Callee         :   _printf(printf) has type function
CallSite       :   [0x0001:0x000000cf]  0x000010CF  int ()


Function       : static, [00001000][0001:00000000], len = 00000006, ___local_stdio_printf_options
                 Function attribute:
                 Function info: inl_specified
FuncDebugStart :   static, [00001000][0001:00000000]
FuncDebugEnd   :   static, [00001005][0001:00000005]
Data           :   static, [00003378][0003:00000378], Static Local, Type: unsigned __int64, _OptionsStorage
```

Figure 2: The information extracted from the debug file using Microsoft's Debug Interface Access SDK.

Inline functions may cause some difficulties; however those difficulties are likely to arise with any technique.

[1] https://msdn.microsoft.com/en-us/library/t6tay6cz.aspx.

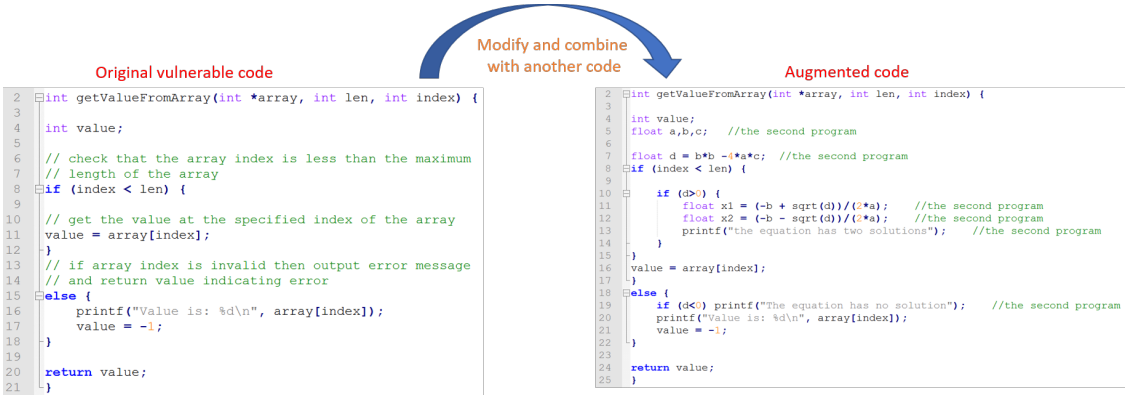

Figure 3: An example of code augmentation. The original code is vulnerable to the CWE-839: Numeric Range Comparison Without Minimum Check. *The sample code only checks if the given array index is less than the maximum length of the array, but it does not verify the minimum value (CWE-839). This will allow a negative value to be accepted as the input array index. This will result in an out-of-bounds read (CWE-125) and offer attackers a way of accessing sensitive memory.*

Debug information from formats like DWARF (used in ELF executables) is also rich enough to allow labelling at finer granularity than entire functions, since it allows identifying the binary instructions to which each source line corresponds. By taking advantage of existing APIs for interpreting the debug information, we may also be able to reduce the amount of human effort in the labelling process by automatically translating labelled source datasets to labelled binary datasets.

4.3. Dataset augmentation

It is well known that deep learning methods require a large amount of data instances to work efficiently. To increase the quantity and quality of training sample code, in this specific context, we borrow the idea of data augmentation from the area of pattern classification. For example, in the topic of image classification, the images are transformed (e.g., rotated, shifted, or their contrast modified) to create additional training data before feeding them to a deep neural network. In the context of software vulnerability discovery, we can undertake data augmentation by compiling source code with a variety of combinations of target platform, hardware architecture, and code optimization. Another option would be to apply semantic-preserving transformations to the source code before compilation. Such transformations have been well studied in the context of metamorphic compiler testing [55], from which we can draw.

Another possible way of code augmentation is to combine the source code of many programs randomly or genetically. Suppose the involved source code programs use the same compilation process – we can then compile the combined source code to binary code. This could make DL classifiers perform better when tackling divergent vulnerable code patterns. Figure 3 shows a typical example of code augmentation.

Note that combining code samples has to be done carefully to ensure that the labelling information is preserved since, for instance, the merging of two code samples could introduce a new code vulnerability, or might mask existing vulnerabilities.

4.4. Ground-truth related work package

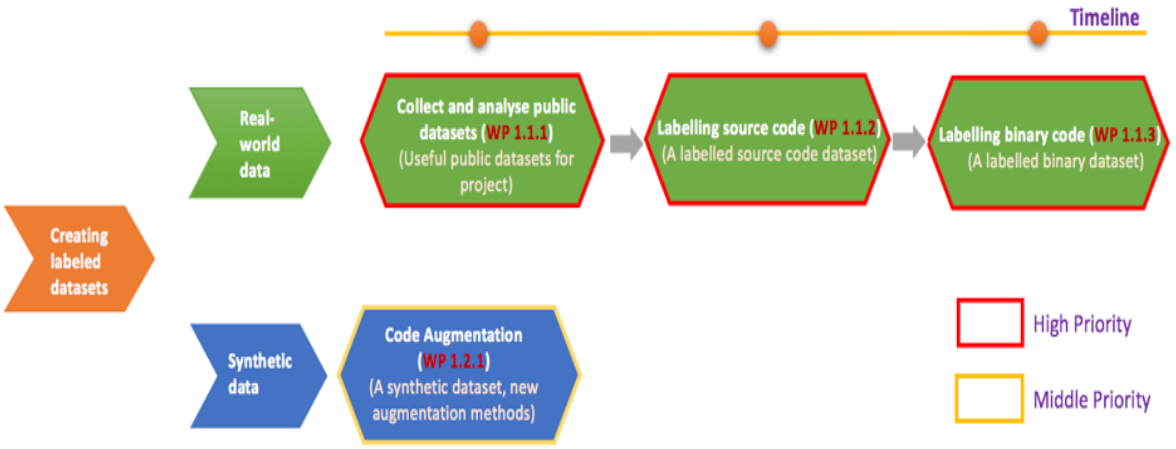

Figure 4: The tasks and sub-tasks in the ground-truth related work package, WP1.

To generate our own ground-truth datasets for software vulnerability discovery we plan to operate as follows. Firstly, we will automatically crawl the GitHub and open source repositories to obtain the source code samples. These source code samples will be labelled with the assistance of CVE security reports. Secondly, we will use code augmentation techniques to enrich the set of labelled source code samples via, for exampe, semantic-preserving source code transformations. Thirdly, the labelled source code will be compiled to binary code and further augmented by applying different sets of compiler flags and compiler versions and so on. Finally, we will apply the ideas of using debug mode for transferring the source code labels to the resulting binary code samples. The *Software Defect Description Language* (SDDL) developed by DST Group will be used to store the ground-truth datasets.

One goal of this project is to create several ground-truth datasets, which will be made available as open source to advance the research on software vulnerability discovery. Work package 1 (i.e., WP1 – the data preparation work package) aims to achieve this goal. Figure 4 shows the main tasks and associated sub-tasks in the work package WP1. Since the ground-truth datasets are fundamental to the research on deep learning, this work package is of high priority and will be undertaken first. The figure also shows the order and priority of the specified tasks.

5. Software Vulnerability Discovery

5.1. Binary code based vulnerability discovery

5.1.1. Vulnerability discovery from byte sequences

A binary file can be viewed as a sequence of bytes or a sequence of machine instructions (See Figure 5). Thus it is natural to employ deep learning models capable of dealing with sequences of objects. In this section, we differentiate between two granularities of analysis : i) at the file level; and ii) at the function level.

5.1.1.1 File-level vulnerability discovery

Although file-level detection is coarse-grained, it is often easy to handle and facilitates the search for vulnerable code. Apart from viewing a binary file as a sequence, we also propose to reorganize the sequence into a two-dimensional matrix so that we are able to explore the correlation between

neighbouring sub-sequences.

5.1.1.1.1 Sequential view of binary files

Binary files are of variable lengths. It is a common practice to apply an RNN to project a sequence into a dense fixed-length vector, which is further fed into a classifier for vulnerability detection. This architecture often requires a substantial amount of labeled training data. Since our labeled training data is sparse, to leverage unlabeled data, we propose two ways of training RNNs in a semi-supervised manner.

Sequential training The first approach starts with applying unsupervised pre-training with a 1D RNN [13], followed by supervised fine-tuning. Inspired by language modelling, the task of unsupervised pre-training is formulated as predicting the next element given a sub-sequence. An element could be a byte or a machine instruction. This task formulation allows the learning of good representations of sequence elements as well as good initialization of the parameters of the RNN. After pre-training, for each binary file, we gain a sequence of hidden states $\boldsymbol{h}_1, ..., \boldsymbol{h}_l$ where l is the length of the sequence. In order to obtain a sequence representation, we can either take $\boldsymbol{h}_l$ or apply a statistical pooling (e.g., taking the average or computing the covariance matrix) over all hidden states if information gets 'lost' by the RNN.

In order to associate sequence representations with vulnerability discovery, at the fine-tuning stage, we apply a classifier (e.g., Support Vector Machine, Deep Neural Network, etc.) on top of the sequence representations (See Figure 6). We will explore various designs of the classifers. One idea is to employ a specific variant of Support Vector Machine that employs the random feature technique [44] to plug the RNN into the classifier so that we can apply standard techniques such as BackProp for training.

Joint training The aforementioned two-stage approach may cause the model to overfit for specific subtasks. In order to encourage more interaction between the 1D RNNs and the classifier, we propose a mechanism that would train a 1D RNN and a classifier simultaneously. In particular, for each binary file, we take the last hidden state generated by the 1D RNN as the input of the classifier (See bottom part of Figure 6). The training objective would take the following form:

$$J\left(\boldsymbol{\theta}_c, \boldsymbol{\theta}_{RNN}, \mathcal{D}\right) = J_c\left(\boldsymbol{\theta}_c, \mathcal{D}\right) + \lambda J_{RNN}\left(\boldsymbol{\theta}_{RNN}, \mathcal{D}\right)$$

where $\mathcal{D}$ is the training set, $\boldsymbol{\theta}_c$ is the model of the classifier, $\boldsymbol{\theta}_{RNN}$ is the model of the 1D RNN, and $\lambda > 0$ is the balance parameter.

Two possible classifiers are under consideration: i) deep feed-forward NNs, and ii) Support Vector Machines using a Fourier random feature with the reparameterization trick [40].

It is worth noting that when setting $\lambda = 0$, we ignore the unsupervised pre-training component, while a large value for λ will make the model focus on the unsupervised task.

5.1.1.1.2 2-dimensional grid of bytes or machine instructions

A binary file could be too long and current RNN techniques may fail to capture long-range dependencies between crucial bytes or machine instructions. An alternative way is to break a long sequence into sub-sequences of fixed length, and arrange them row-wise into a matrix. We can then exploit the correlation of neighbouring sub-sequences easily by using neural network models that have been applied to images, such as CNNs or 2D RNNs. The length of the row then becomes a hyperparameter, which will need to be tuned for target code bases.

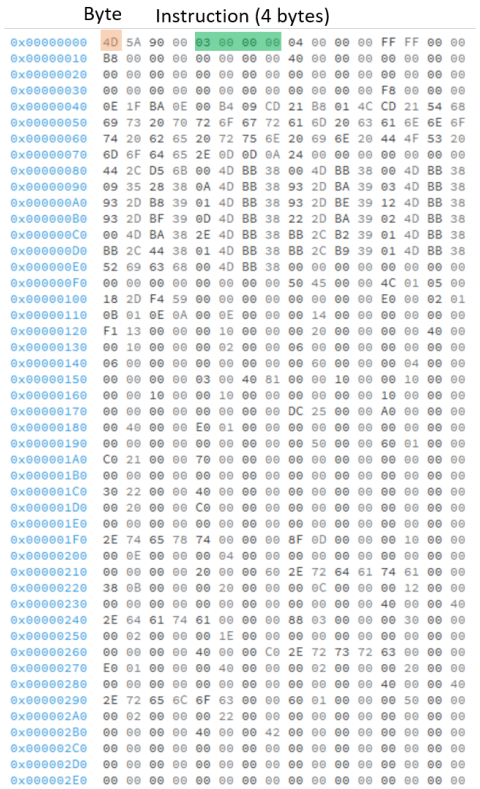

Figure 5: Binary file represented as a sequence of bytes (orange) or as a sequence of machine instructions (green).

2-dimensional grid with CNN Since we view a binary file as a 2-dimensional grid of bytes or machine instructions, we may leverage the efforts of the computer vision community by treating such a input as an image for identifying "vulnerable programming patterns". The CNN is therefore a natural choice for these image-like inputs. Herein, we assume that the binary files of an application contains 0s and 1s or values in hexadecimal format that are not humanly readable.

Given a program's binary file (e.g., an *.exe* file or *.dll* file), it can be converted to a grey scale image or a RGB colour image. In the former case, a binary file can be regarded as a vector of 8-bit unsigned integers and then organized into a 2D array. The array can be visualized as a grey-scale image with each 'pixel' having a value in the range [0,255] (0: black, 255: white). In the latter case, a binary file can be alternatively viewed in hexadecimal format. A mapping can be created to directly map hexadecimal values to a RGB colour, which is in the same format as images with three colour channels. Figure 7 shows two functions that each suffers from a buffer overflow vulnerability.

2-dimensional grid with 2D RNN Since CNNs often fail to capture dependencies between sequence elements that are far from each other, we employ a 2D RNN [22] to capture long-range dependencies. However, doing so requires scanning over all cells of the input, and so the time complexity is high. To speed up the computation and encourage parallelisation, we will consider variants of the 2D RNN including Pyramid 2D RNNs [52] and Pixel RNNs [41].

5.1.1.2 Function-level vulnerability discovery

In many cases, file-level detection is too coarse-grained. Therefore we also consider vulnerability discovery at the function level. This finer-grained detection requires first to detect the scope of each function within the binary (i.e. the machine instructions that implement the function). Since the binary code within each function can be viewed as sequences as well, we can apply the techniques presented in Section 5.1.1 on the sub-sequence of instructions defined by each function's scope. Thus the key challenge is the identification of function scope from a binary (cf. Section 4.2).

Function scope identification. As shown in Figure 8, function scopes can be organized into a hierarchy. Thus it is natural to segment bytes or machine instructions and apply hierarchical clustering [6, 56] on the segments.

The segmentation as well as clustering imposes a special challenge on the representation of sequence elements in a binary file. One possible way is to apply *seq2seq* models [53, 11, 4] or *pointer networks* [57]. The goal of these models is to learn good representations by minimizing reconstruction losses, as illustrated by Figure 9. The *seq2seq* models consist of an encoder and a decoder. The encoder aims to project a sequence into a fixed-length embedding, and the decoder reconstructs the input sequence by taking the embedding as input. For a long program, the models may lose track of important information as the encoder reads long sequences. One possible solution is to incorporate the attention mechanism [4] to focus on key elements in the input sequences.

5.1.2. Vulnerability discovery from assembly code

Viewing binary code as a sequence loses some of their structural information. When we disassemble the binary code into assembly code (See for example Figure 10), it is possible to recover some of this structural information by viewing the resulting code as one or more graphs constructed from the AST, CFG etc. In this section, we propose techniques capable of exploiting both syntactic structures (like ASTs) and semantic structures (like CFGs) of code.

Figure 11 gives an overview of our proposed approaches. It is straightforward to disassemble binary code into assembly code by using open source tools. The assembly code derived from binary code can then be viewed either as a graph or a collection of functions. The graph view will lead to the learning of graph representations, while the function view could directly lead to deep learning-based models for vulnerability detection. Figure 12 shows an example of the full control flow graph based on an assembly file.

In this project we are mainly interested in fine-grained analysis, thus we aim to detect vulnerabilities at the function level in both views. For the detection of function scope, we largely reuse the techniques proposed in Section 5.1.1. The main difference is that we map tokens in the assembly code to token embeddings (See Figure 14) before feeding them into models such as pointer networks (See Figure 13). The use of token embeddings will allow us to generalize over similar tokens.

5.1.2.1 Function View

A function in assembly code is a sequence of tokens. Following the same idea as function scope detection, we first map tokens in a function into token embeddings. The embeddings of a function can be viewed as sequences or 2-dimensional matrices so that we can directly apply the techniques proposed in Section 5.1.1 for vulnerability detection. Figure 14 shows an example of assembly code with three functions.

5.1.2.2 Graph View

The graph view attaches functions into a single graph constructed from the assembly code AST or the program's CFG so that it is possible to explore relationships between functions.

There are a number of techniques to learn graph representations. One family of techniques is neural embedding techniques [33, 24, 42, 25], which can be applied to transform the AST/CFG of a function to vectorial graph representations. The DL methods are trained on the top of vectorial graph representations to detect vulnerabilities at the function level.

5.1.2.3 Deep learning based detectors

Given graph representations, we can employ Graph-based Convolution Neural Networks [8, 30, 16, 27], which are able to explore the structural information of graphs.

Another possibility is to use Multi-dimensional RNNs [22, 52, 41], which efficiently exploit the context in grid structures (i.e., in 2D or 3D grids). However, since the input is graphs, we need to extend the multi-dimensional RNNs to exploit vectorial graph representations induced from the previous steps.

5.1.2.4 Challenges of the approaches based on assembly code

1. There could be complex structural relationships between machine instructions after constructing from ASTs or CFGs. Existing DL methods may have difficulty capturing such complicated relationships.

2. At this stage it is unclear what is the best way to learn representations of graph nodes and edges so that the models can explore their structural relationships easily.

3. Since graphs are attached with function scopes, it is not clear how to detect a vulnerability within each function and explore the relationships between functions at the same time.

5.2. Source code based vulnerability discovery

We can obtain yet another representation of binary code by decompiling it to source code. Though decompilation to source code is not always feasible and can be error-prone in the presence of self-modifying code, it should still be highly applicable in our context. Similar to our approach for analysing assembly code, we employ graphs to represent source code. Herein graph nodes are source language tokens and edges indicate their structural relationships. Since labeled training data is limited, our research in this line also focuses on learning graph representations and incorporating them to build deep learning classifiers for vulnerability detection, as shown in Figure 15. Since the training data is limited, and as has been mentioned previously, we tackle this problem with automatic code generation and semi-supervised learning.

5.2.1. Learning graph representations

We consider two ways of building graph representations in an unsupervised way.

Inspired by *Word2Vec* [33] and *Node2Vec* [24], we propose a novel method called *Program2Vec* to formulate the representation learning problem as a task of predicting paths.

By assuming the paths originating from the same node share similar representations, *Program2Vec* formulates the learning of graph representations as a predictive task. In particular, for each node, the

Program2Vec model aims to predict the paths connecting to the node. Formally, given a node v, we maximize the probability of seeing its paths in a spanning tree $Span(v)$ rooted at v. The corresponding optimization problem is formulated as follows:

$$\max_{W} \left(\prod_{v} \prod_{u \in Span(v)} p\left(\pi\left(u\right)|v\right) \right)$$

where $\pi(u)$ denotes the path from the current root v to the vertex u excluding u and Span(v) specifies the spanning tree rooted at v. A deep model with parameters W is applied for constructing path and node representations.

An alternative approach, coined *AST2Vec*, aims to learn good graph representations by reconstructing ASTs. Given a graph representing an AST, we apply first an encoder to project the AST into a fixed-length embedding, as illustrated in Figure 16. This model reads an AST in a bottom-up manner, and optionally employs Structure-based Traversal (SBT) [25] to capture the structure information. Then a decoder takes the embedding as input and tries to reconstruct the input AST. The reconstruction process will enable the model to capture the most essential structural information in ASTs.

5.2.2. Building graph-based vulnerability detectors

Given training datasets labeled with software vulnerability information, we could train a vulnerability detector based on graph representations.

A graph representation learned by *Program2Vec* or *AST2Vec* is a collection of vectors linked by their structural relationships. It is straightforward to apply Graph-based Convolution Neural Networks (GCNNs) [8, 30, 16, 27] to learn the association between vulnerability information and graph representations. We can extend GCNNs to locate vulnerabilities at the function level.

An alternative way is to apply Recursive Neural Networks to read graph representations and attach a vulnerability detector to each function scope. The model could begin with the main function, follow each function call within the main function to reach other functions. The whole process is repeated in a recursive manner until the whole program is read. Each statement within a function is treated as a sequence of tokens. Each token is further mapped to token embeddings, as illustrated in Figure 17.

5.2.3. Tackling problems of limited training data

There is a limited number of labeled training datasets available for vulnerability discovery. In this project we propose two ways to tackle this problem: i) automatic generation of code samples ii) use semi-supervised learning techniques by incorporating code generation with the training of vulnerability detectors.

5.2.3.1 Automatic Code Generation

High-level programming languages allow us to implement the same functionality in multiple ways. In order to make our vulnerability detector invariant to such syntactic variations, we propose to automatically rewrite existing vulnerable functions in different ways. Doing so can increase the diversity of source code while preserving its vulnerability types.

Rewriting can be done either by using classical semantic preserving transformations (as mentioned earlier in Section 4.3), or by using semi-supervised learning techniques, such as combining the ideas of Generative Adversarial Networks (GAN) [21], Conditional Generative Adversarial Networks [34],

and deep reinforcement learning [35] to model correct rewriting operations. The former technique generates code with syntactic variations while preserving the semantics, while the latter technique aims to preserve the vulnerabilities of code after rewriting with possible discrepancies of semantics. For example, if a C program statement `strcpy(buffer, 100)` triggers a buffer overflow, a statement `strcpy(buffer, n)` with `n` > 100 will also lead to buffer overflow. We aim to modify a small section of the vulnerable code. Each rewrite operation is designed to preserve the syntactic correctness of the source code. We expect that the generated source code will remain compilable. In the worst case, it may require very little human effort to debug.

5.2.3.2 Semi-Supervised Learning

GANs and Conditional GANs can also be extended for semi-supervised learning [29, 28]. The GAN architecture is formulated as a two-player game, which consists of a generator and a discriminator. A generator generates fake samples while the discriminator aims to distinguish between true samples and the fake samples from the generator. During training, the generator learns to fool the discriminator by generating samples as similar as possible to the true samples, while the discriminator learns to distinguish them. In semi-supervised learning, we incorporate vulnerability detectors into the GAN architecture so that the model also learns to associate correct labels with both true samples and generated samples. Previous work [28, 29] have shown that a well-trained generator can generate high quality negative samples, possibly also high quality positive samples that greatly improve the performance of the classifier especially when the training data is small. Since the previous work focused on the generation of images and text, the core challenge will be to explore ways of generating high-quality code useful for the corresponding classification task.

5.2.4. Challenges of the approaches based on source code

1. The decompiled source code could feature a complex structure that is beyond the capability of current graph-based models. Novel models may be required to recognise long-range dependencies between programming constructs.

2. The decompiled source code could contain errors and noise. The detection models should be robust against such noise.

5.3. Multiple source based vulnerability discovery

Prior work on vulnerability detection focuses on source code analysis, exploiting the syntax, names of programming constructs, and structures of the code. Given binary code however, one can leverage this previous work by performing decompilation to get automatically generated source code. Unfortunately, such code is often error-prone, noisy and not easy to read. In contrast, some vulnerability patterns are possible to detect even in binary code though such complex patterns are very challenging to detect. In order to combine the best of multiple sources, the key idea is to design targeted models to detect patterns from multiple sources such as binary code and the source code generated by binary decompilation. Such multi-source models are expected to provide higher robustness towards noise, by exploring the ideas of multi-view learning and multi-expert learning.

The same functionality implemented in different programming languages (e.g. C source code vs. compiled assembly code for the same program) can be regarded as different views of the same program. Since different views share the same semantics, we can assume that all views are generated from the same semantic space. For vulnerability detection, we project the features extracted from different sources to the same semantic space, and feed the abstract features in that space to the final

classifier. During the projection, we will exploit the alignment information obtained by decompilation to build shared representations for aligned code fragments. The feature projection process will focus on the features which have strong correction between different views so that noise generated by reverse engineering will be ignored. Since the single source models we proposed are deep neural networks, the feature projectors are essentially some additional neural networks layers on top of the high-level features of the single source model. The challenge herein is to learn the correlations between code fragments in different views.

Multi-source analysis can also be viewed as a way of exploiting the knowledge of multiple experts since each single source model is regarded as an expert for a particular programming language or program representation. The multi-expert strategy aims to combine the predictions, and optionally the respective evidence, of single source models to make the final decisions. Let $\mathbf{x}$ denote the semantics of a program and y indicate if it is vulnerable or not, one possible way is to employ mixture models.

$$p(y|\mathbf{x}) = \sum_{s \in S} p(y|\mathbf{x}_s) p(\mathbf{x}_s|\mathbf{x})$$

where $\mathbf{x}_s$ denotes the program from a particular source s in a space of all sources S. The term $p(y|\mathbf{x}_s)$ is essentially a single source model based on a single source input $\mathbf{x}_\mathbf{s}$ and $p(\mathbf{x}_s|\mathbf{x})$ assigns credibility of the single model based on its correlation to the semantics of the program.

5.4. The main tasks in the vulnerability detection work package

The main goal of the DLC project is to develop deep learning techniques for vulnerability discovery in binary code, which forms the core part of the work package 2 (WP2). Figure 18 gives an overview of the main tasks and their priorities in WP2.

6. Project Plans

Figure 19 shows the plan for the tasks in the *Deep Learning for Cyber project*.

Table 2: Existing datasets of software vulnerabilities.

Dataset	**Description**	**Remark**
Deakin Univ. datasets	These datasets contain software vulnerabilities at the function level obtained from open source code and CVE.	The datasets have been labelled by a PhD student from Deakin Univ.
DST Group dataset	This contains functions or sets of related functions that have software vulnerabilities.	The dataset is shared by DST Group
NIST SAMATE	This contains real software applications with known bugs and vulnerabilities. (SAMATE: Software Assurance Metrics And Tool Evaluation)	Open Source
Open Judge system	This comes from an online Open Judge (OJ) system, which contains a large number of programming problems for students. More than 10 features including problem description have been provided to describe the dataset.	Open Source
DARPA Cyber Grand Challenge	The datasets generated during the DARPA Cyber Grand Challenge.	Open Source
F-Droid and Android OS	This dataset originally contains 20 popular applications, which were collected from F-Droid and Android OS in 2011. The F-Droid repository contains a growing number of more than 2,300 apps.	Open Source
Juliet test suite	The test suite was developed specifically for assessing the capabilities of static analysis tools. It is intended for anyone who wishes to use the test cases for their own testing purposes, or who would like to have a greater understanding of how the test cases were created.	Open Source
Google Project Zero	This contains a small number software vulnerabilities that have been thoroughly analysed in-depth and documented. Project Zero focuses on finding and documenting zero-day vulnerabilities in open source code.	Open Source

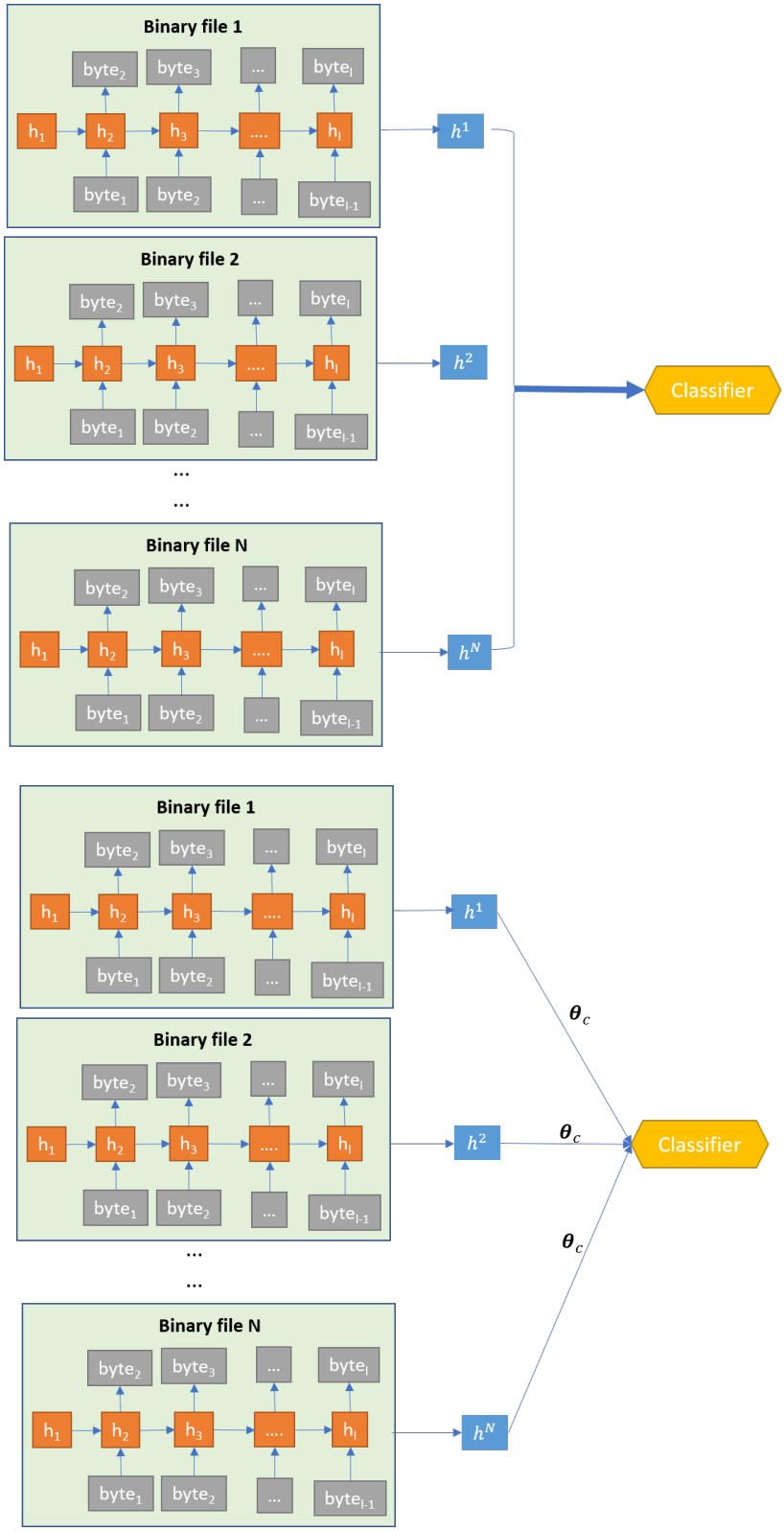

Figure 6: Top: Training the 1D RNNs and classifier sequentially. Bottom: Training the 1D RNNs and classifier simultaneously, where the classifier parameter $\boldsymbol{\theta}_c$ is shared among the 1D RNNs.

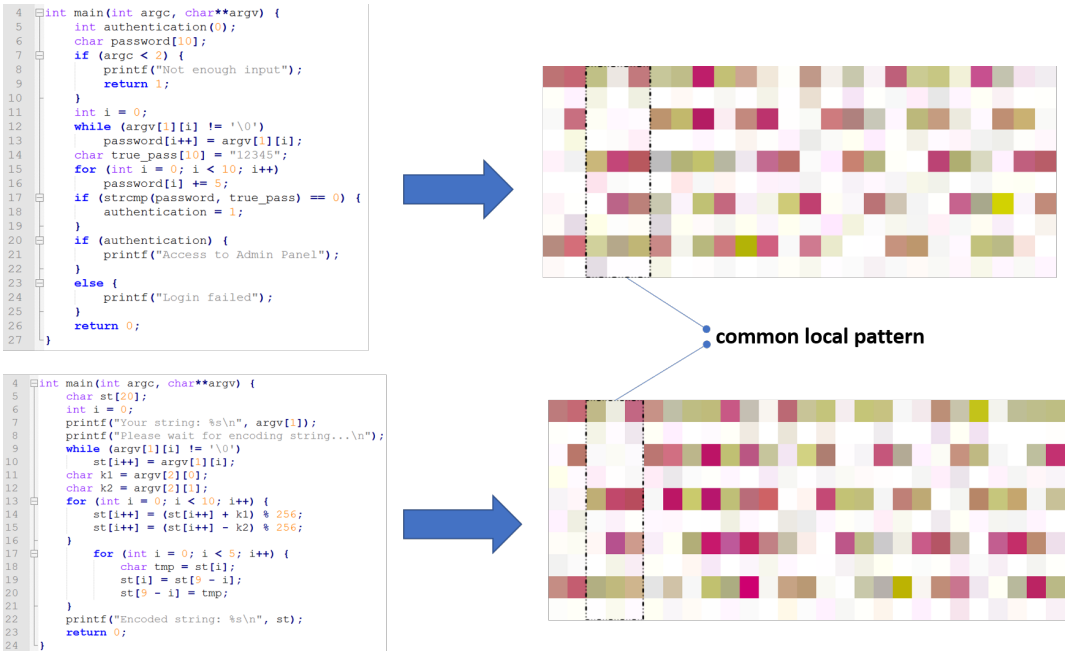

Figure 7: Two functions that each suffers from a buffer overflow vulnerability. There is a common localised pattern in their respective source code hence, when compiled to binary code and then visualized as images, the resulting images share a local 'pixel' pattern that can be efficiently detected using CNN(s). We note that each 4-byte instruction is visualized as one pixel using the CMYK colour system.

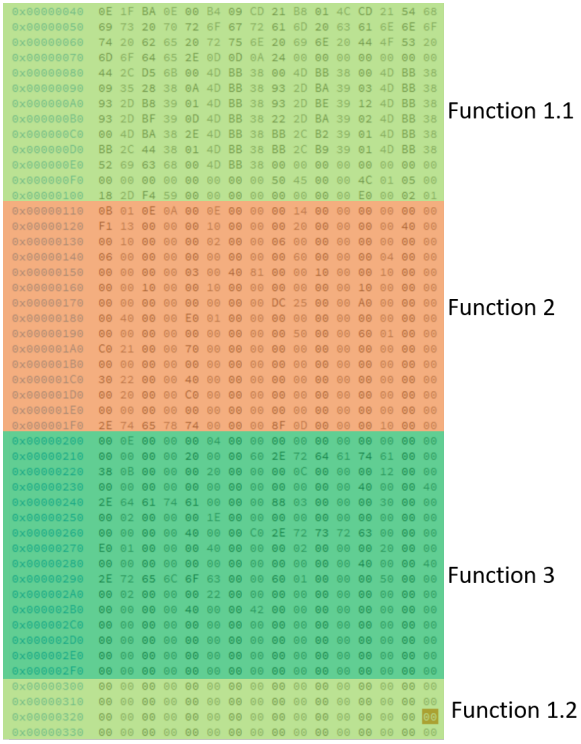

Figure 8: A binary file with three functions: Function 1.x (multiple sections), Function 2, and Function 3.

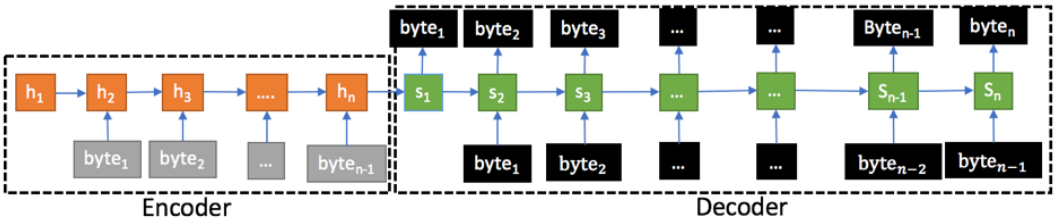

Figure 9: Encoder-decoder architecture with attention mechanism for function scope identification. The orange rectangles denote the hidden states of the encoder, while the green ones are the hidden states of the decoder.

```
.text:0x00001000 __local_stdio_printf_options:
.text:0x00001000         B878334000              mov eax, va_ptr `__local_stdio_printf_options'::`2
.text:0x00001005         C3                      ret
.text:0x00001006 data_0x1006:
.text:0x00001006         db 10 dup(0xCC)
.text:0x00001010 printf:
.text:0x00001010         55                      push ebp
.text:0x00001011         8BEC                    mov ebp, esp
.text:0x00001013         56                      push esi
.text:0x00001014         8B7508                  mov esi, dword ptr [ebp+0x8]
.text:0x00001017         6A01                    push 0x1
.text:0x00001019         FF15B4204000            call dword ptr [__imp____acrt_iob_func]   ; void *
.text:0x0000101F         83C404                  add esp, 0x4
.text:0x00001022         8D4D0C                  lea ecx, [ebp+0xC]
.text:0x00001025         51                      push ecx
.text:0x00001026         6A00                    push 0x0
.text:0x00001028         56                      push esi
.text:0x00001029         50                      push eax
.text:0x0000102A         E8D1FFFFFF              call __local_stdio_printf_options ; unsigned __int
.text:0x0000102F         FF7004                  push dword ptr [eax+0x4]
.text:0x00001032         FF30                    push dword ptr [eax]
.text:0x00001034         FF15B0204000            call dword ptr [__imp____stdio_common_vfprintf]
.text:0x0000103A         83C418                  add esp, 0x18
.text:0x0000103D         5E                      pop esi
.text:0x0000103E         5D                      pop ebp
.text:0x0000103F         C3                      ret
.text:0x00001040 admin_panel:
.text:0x00001040         56                      push esi
.text:0x00001041         8B35AC204000            mov esi, dword ptr [__imp__getchar]   ; void (__cd
.text:0x00001047         660F1F840000000000      o16 nop [eax+eax+0x0]
.text:0x00001050 code_0x1050:
.text:0x00001050         68F8204000              push va_ptr string_AdminPanel     ; "Admin Panel"
.text:0x00001055         E8B6FFFFFF              call printf                       ; int __cdecl pr
.text:0x0000105A         6808214000              push va_ptr string_1Changecon     ; "1. Change con
.text:0x0000105F         E8ACFFFFFF              call printf                       ; int __cdecl pr
.text:0x00001064         6820214000              push va_ptr string_2Changesta     ; "2. Change sta
.text:0x00001069         E8A2FFFFFF              call printf                       ; int __cdecl pr
.text:0x0000106E         683C214000              push va_ptr string_3Exit          ; "3. Exit"
```

Figure 10: Example binary code and the resulting assembly code after decompilation.

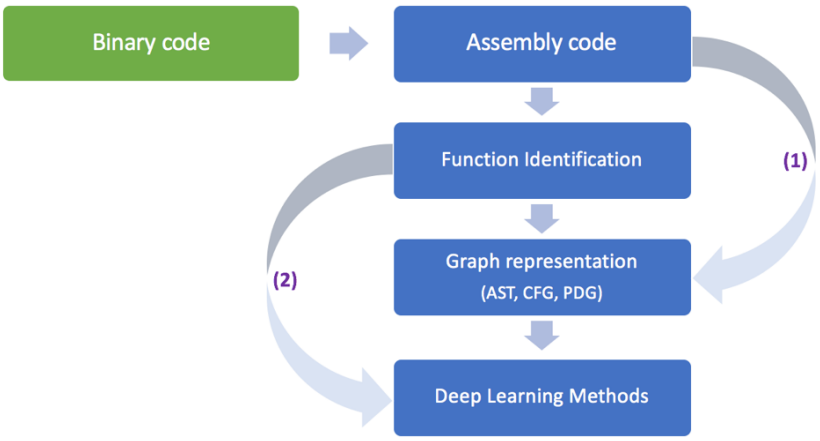

Figure 11: Overview of the assembly code based approaches. The assembly code derived from the binary code can be viewed either as a graph (graph view [1]) or a collection of functions (function view [2]).

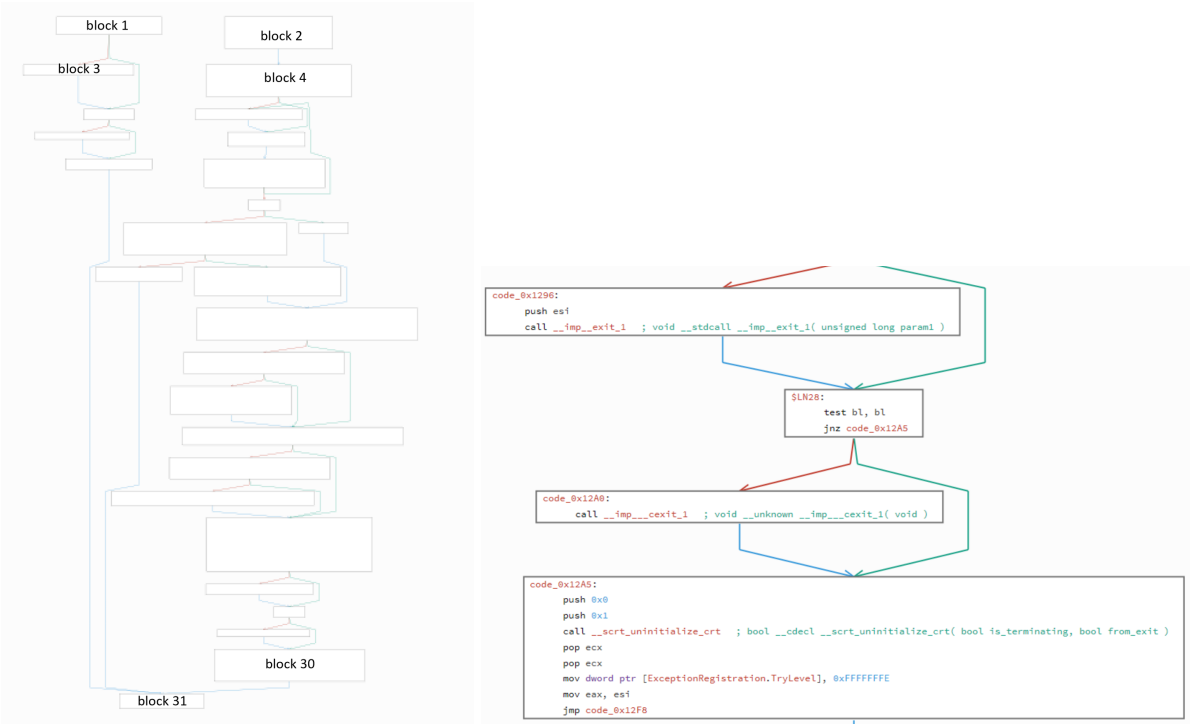

Figure 12: Example control flow graph (CFG) based on the assembly code approach. Left: The full control flow graph which is constructed based on the entire assembly file. Right: The example section of the control flow graph. We note that this control flow graph shows the hierarchical, nested, and semantic structure of the binary file. Each block in this control flow graph can be further used as input to deep learning methods to detect vulnerabilities at the block level.

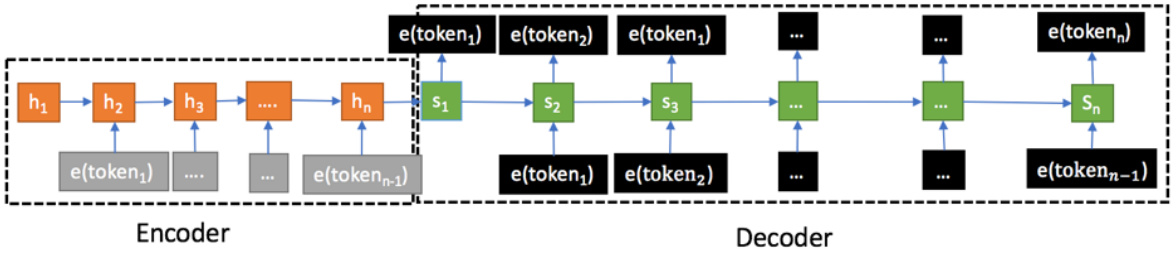

Figure 13: The encoder-decoder architecture for function scope identification in assembly code. Tokens are embedded into a vector space and $e(token)_i$ denotes a token embedding.

```
.text:0x00001000 __local_stdio_printf_options:
.text:0x00001000         B878334000              mov eax, va_ptr `__local_stdio_printf_options'::`2
.text:0x00001005         C3                      ret
.text:0x00001006 data_0x1006:
.text:0x00001006         db 10 dup(0xCC)
.text:0x00001010 printf:
.text:0x00001010         55                      push ebp                                          Function 1.1
.text:0x00001011         8BEC                    mov ebp, esp
.text:0x00001013         56                      push esi
.text:0x00001014         8B7508                  mov esi, dword ptr [ebp+0x8]
.text:0x00001017         6A01                    push 0x1
.text:0x00001019         FF15B4204000            call dword ptr [__imp____acrt_iob_func]   ; void *
.text:0x0000101F         83C404                  add esp, 0x4
.text:0x00001022         8D4D0C                  lea ecx, [ebp+0xC]
.text:0x00001025         51                      push ecx
.text:0x00001026         6A00                    push 0x0
.text:0x00001028         56                      push esi
.text:0x00001029         50                      push eax
.text:0x0000102A         E8D1FFFFFF              call __local_stdio_printf_options ; unsigned __int  Function 2
.text:0x0000102F         FF7004                  push dword ptr [eax+0x4]
.text:0x00001032         FF30                    push dword ptr [eax]
.text:0x00001034         FF15B0204000            call dword ptr [__imp____stdio_common_vfprintf]
.text:0x0000103A         83C418                  add esp, 0x18
.text:0x0000103D         5E                      pop esi
.text:0x0000103E         5D                      pop ebp
.text:0x0000103F         C3                      ret
.text:0x00001040 admin_panel:
.text:0x00001040         56                      push esi
.text:0x00001041         8B35AC204000            mov esi, dword ptr [__imp__getchar]   ; void (__cde
.text:0x00001047         660F1F840000000000      o16 nop [eax+eax+0x0]
.text:0x00001050 code_0x1050:                                                                      Function 3
.text:0x00001050         68F8204000              push va_ptr string_AdminPanel     ; "Admin Panel"
.text:0x00001055         E8B6FFFFFF              call printf                       ; int __cdecl pr
.text:0x0000105A         6808214000              push va_ptr string_1Changecon     ; "1. Change con
.text:0x0000105F         E8ACFFFFFF              call printf                       ; int __cdecl pr
.text:0x00001064         6820214000              push va_ptr string_2Changesta     ; "2. Change sta
.text:0x00001069         E8A2FFFFFF              call printf                       ; int __cdecl pr  Function 1.2
.text:0x0000106E         683C214000              push va_ptr string_3Exit          ; "3. Exit"
```

Figure 14: Example of assembly code with three functions inside.

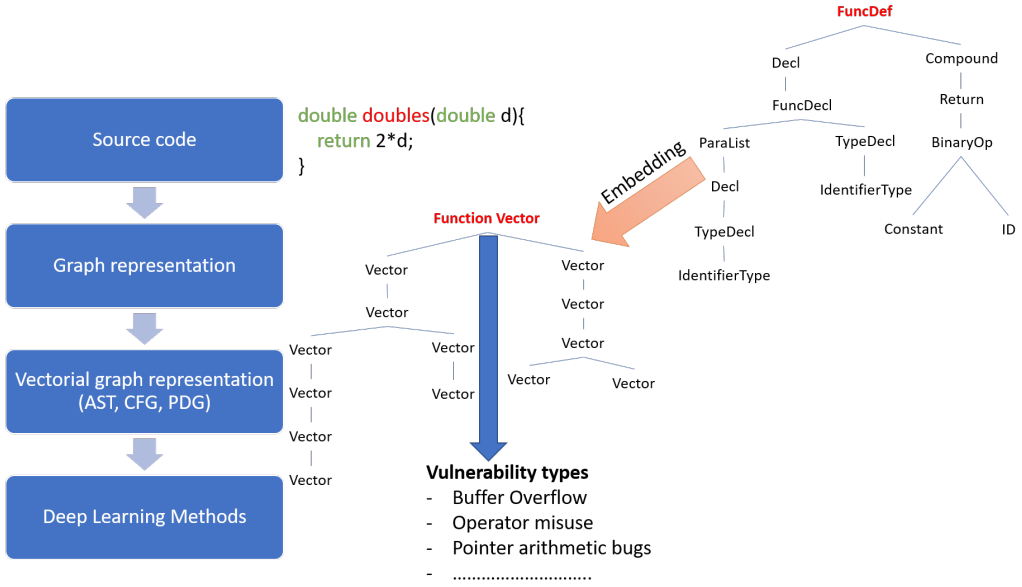

Figure 15: The execution diagram after decompiling the binary code to source code.

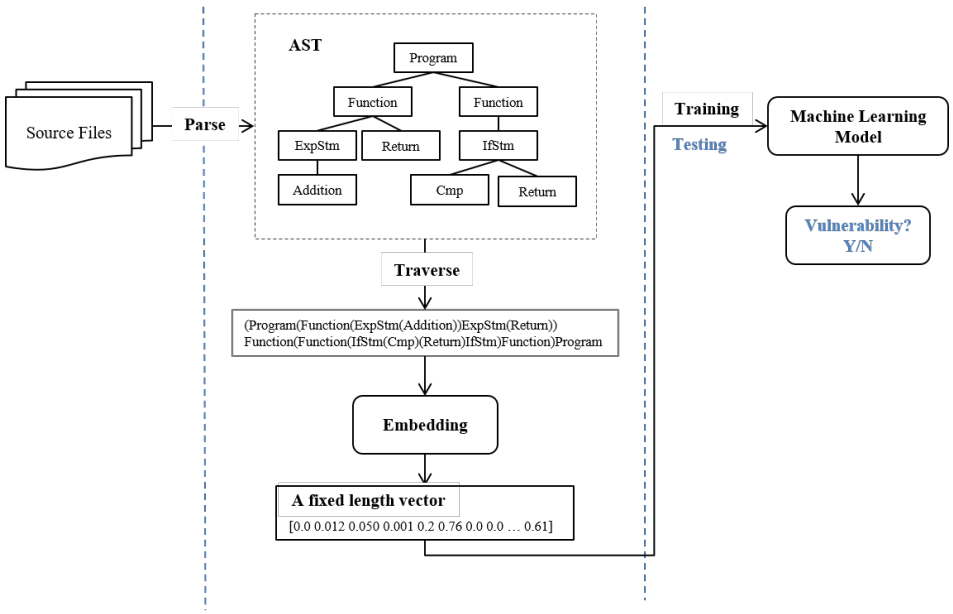

Figure 16: The overall workflow of AST2Vec *(adapted from [25]).*

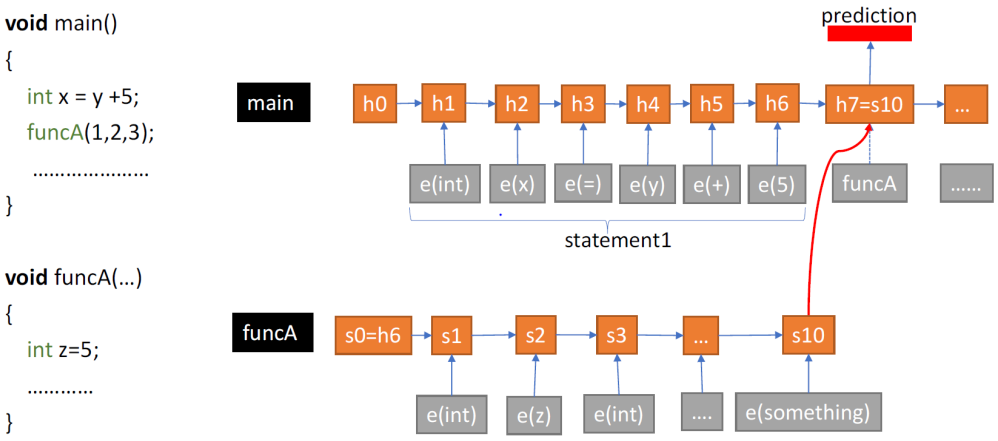

Figure 17: The proposed function-level RNN for software vulnerability discovery.

Figure 18: The tasks in the vulnerability detection work package WP2.

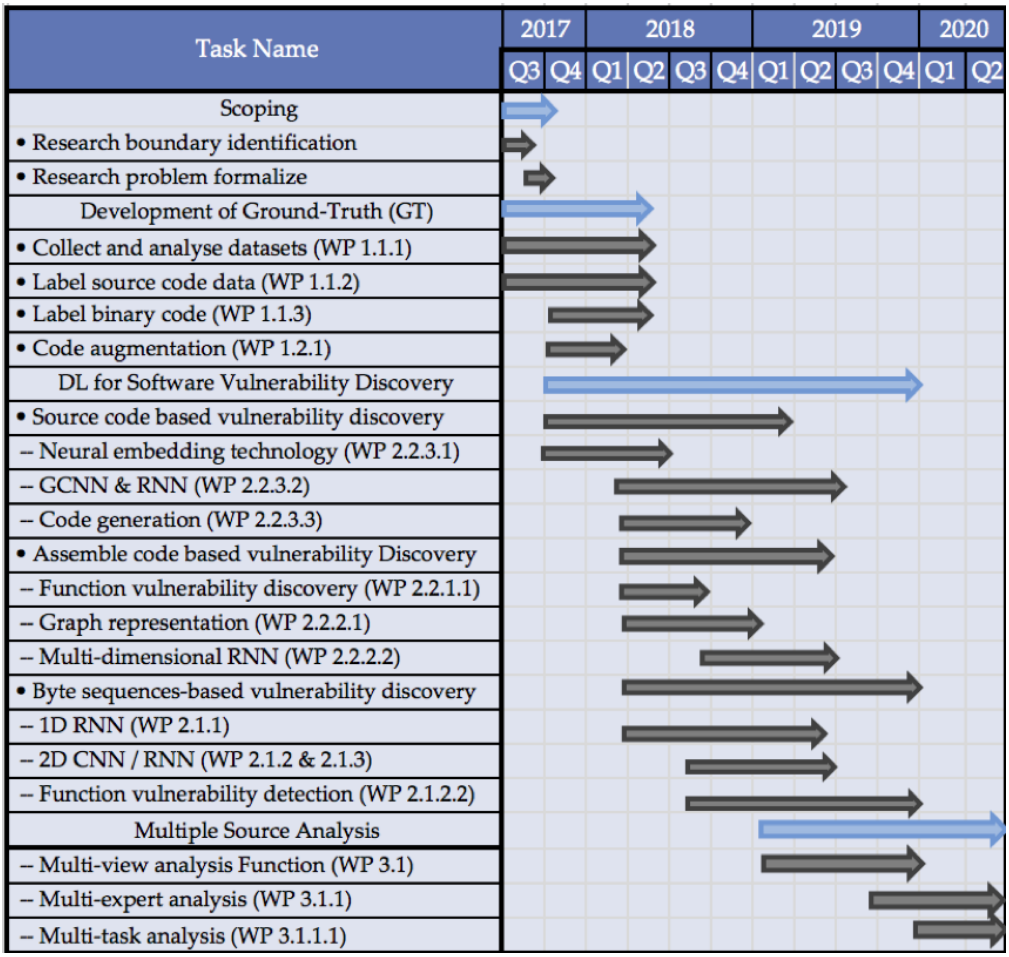

Figure 19: The plan for the tasks in the Deep Learning for Cyber project.

References

[1] T. Abraham and O. de Vel. A review of machine learning in software vulnerability research, 2017. Private, unpublished draft.

[2] H. Alves, B. Fonseca, and N. Antunes. Experimenting machine learning techniques to predict vulnerabilities. In *Dependable Computing (LADC), 2016 Seventh Latin-American Symposium on*, pages 151–156. IEEE, 2016.

[3] A. Avancini and M. 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.

[4] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[5] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. Wysinwyx: What you see is not what you execute. *Verified software: theories, tools, experiments*, pages 202–213, 2008.

[6] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[7] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. Bap: A binary analysis platform. In *International Conference on Computer Aided Verification*, pages 463–469. Springer, 2011.

[8] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun. Spectral networks and locally connected networks on graphs. *CoRR*, abs/1312.6203, 2013.

[9] C. Cadar, D. Dunbar, D. R. Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[10] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[11] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[12] A. Cozzie, F. Stratton, H. Xue, and S. T. King. Digging for data structures. In *OSDI*, volume 8, pages 255–266, 2008.

[13] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose. Automatic feature learning for vulnerability prediction. *arXiv preprint arXiv:1708.02368*, 2017.

[14] M. Dowd, J. McDonald, and J. Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[15] V. D'Silva, M. Payer, and D. Song. The correctness-security gap in compiler optimization. In *IEEE Symposium on Security and Privacy Workshops*, pages 73–87, 2015.

[16] D. K. Duvenaud, D. Maclaurin, J. Iparraguirre, R. Bombarell, T. Hirzel, A. Aspuru-Guzik, and R. P. Adams. Convolutional networks on graphs for learning molecular fingerprints. In *Advances in Neural Information Processing Systems 28*, pages 2224–2232. 2015.

[17] M. D. Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, pages 24–27, 2003.

[18] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovre: Efficient cross-architecture identification of bugs in binary code. In *NDSS*, 2016.

[19] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[20] S. M. Ghaffarian and H. R. Shahriari. Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey. *ACM Computing Surveys (CSUR)*, 50(4):56, 2017.

[21] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[22] A. Graves and J. Schmidhuber. Offline handwriting recognition with multidimensional recurrent neural networks. In *Advances in neural information processing systems*, pages 545–552, 2009.

[23] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. 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. ACM, 2016.

[24] A. Grover and J. Leskovec. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 855–864, 2016.

[25] X. Hu, Y. Wei, G. Li, and Z. Jin. Codesum: Translate program language to natural language. *arXiv preprint arXiv:1708.01837*, 2017.

[26] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[27] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016.

[28] A. Kumar, P. Sattigeri, and P. T. Fletcher. Improved semi-supervised learning with gans using manifold invariances. *CoRR*, abs/1705.08850, 2017.

[29] W. Lai, J. Huang, and M. Yang. Semi-supervised learning for optical flow with generative adversarial networks. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*, pages 353–363, 2017.

[30] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel. Gated graph sequence neural networks. *CoRR*, abs/1511.05493, 2015.

[31] S. Liu, J. Zhang, Y. Xiang, and W. Zhou. Fuzzy-based information decomposition for incomplete and imbalanced data learning. *IEEE Transactions on Fuzzy Systems*, 2017.

[32] Q. Meng, S. Wen, B. Zhang, and C. Tang. Automatically discover vulnerability through similar functions. In *Progress in Electromagnetic Research Symposium (PIERS)*, pages 3657–3661. IEEE, 2016.

[33] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26*, pages 3111–3119. 2013.

[34] M. Mirza and S. Osindero. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784*, 2014.

[35] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[36] P. Morrison, K. Herzig, B. Murphy, and L. Williams. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, page 4. ACM, 2015.

[37] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[38] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[39] J. Neystadt. Automated penetration testing with white-box fuzzing. *MSDN Library*, 2008.

[40] T. D. Nguyen, T. Le, H. Bui, and D. Phung. Large-scale online kernel learning with random feature reparameterization. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[41] A. v. d. Oord, N. Kalchbrenner, and K. Kavukcuoglu. Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*, 2016.

[42] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin. Building program vector representations for deep learning. In *International Conference on Knowledge Science, Engineering and Management*,

pages 547–553. Springer, 2015.

[43] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*, 2017.

[44] A. Rahimi and B. Recht. Random features for large-scale kernel machines. In *Advances in neural information processing systems*, pages 1177–1184, 2008.

[45] D. A. Ramos and D. R. Engler. Under-constrained symbolic execution: Correctness checking for real code. In *USENIX Security Symposium*, pages 49–64, 2015.

[46] I. Santos, J. Devesa, F. Brezo, J. Nieves, and P. G. Bringas. Opem: A static-dynamic approach for machine-learning-based malware detection. In *International Joint Conference CISIS*, pages 271–280. Springer, 2013.

[47] J. Saxe and K. 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.

[48] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering*, 40(10):993–1006, 2014.

[49] L. K. Shar and H. B. K. Tan. Predicting sql injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Information and Software Technology*, 55(10):1767–1780, 2013.

[50] E. C. R. Shin, D. Song, and R. Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security Symposium*, pages 611–626, 2015.

[51] Y. Shin, A. Meneely, L. Williams, and J. 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.

[52] M. F. Stollenga, W. Byeon, M. Liwicki, and J. Schmidhuber. Parallel multi-dimensional lstm, with application to fast biomedical volumetric image segmentation. In *Advances in Neural Information Processing Systems*, pages 2998–3006, 2015.

[53] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[54] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[55] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *17th Asia Pacific Software Engineering Conference (APSEC)*, pages 270–279. IEEE, 2010.

[56] Y. W. Teh, M. I. Jordan, M. J. Beal, and D. M. Blei. Hierarchical dirichlet processes. *Journal of the American Statistical Association*, 101, 2004.

[57] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. In *Advances in Neural Information Processing Systems*, pages 2692–2700, 2015.

[58] D. H. White and G. Lüttgen. Identifying dynamic data structures by learning evolving patterns in memory. In *TACAS*, pages 354–369. Springer, 2013.

[59] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural network-based graph embedding for cross-platform binary code similarity detection. *arXiv preprint arXiv:1708.06525*, 2017.

[60] F. Yamaguchi, F. Lindner, and K. Rieck. Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX conference on Offensive technologies*, pages 13–13. USENIX Association, 2011.

[61] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 797–812. IEEE, 2015.

[62] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. 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*, pages 91–100. ACM, 2009.

[63] T. Zimmermann, N. Nagappan, and L. Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 421–428. IEEE, 2010.

DEFENCE SCIENCE AND TECHNOLOGY GROUP DOCUMENT CONTROL DATA			1. DLM/CAVEAT (OF DOCUMENT)
2. TITLE Deep Learning for Cyber Vulnerability Discovery: NGTF Project Scoping Study			3. SECURITY CLASSIFICATION (FOR UNCLASSIFIED LIMITED RELEASE USE (L) NEXT TO DOCUMENT CLASSIFICATION) Document (U) Title (U) Abstract (U)
4. AUTHORS de Vel O., Hubczenko D., Kim J., Montague P., Xiang Y., Phung D., Zhang J., Murray T., Le T., Wen S., Liu S., Nguyen V., Lin G., Nguyen K., Le T., Nguyen T., Nock R., Qu L.			5. CORPORATE AUTHOR Defence Science and Technology Group PO Box 1500 Edinburgh, South Australia 5111, Australia
6a. DST GROUP NUMBER DST-Group–GD–1039	6b. AR NUMBER	6c. TYPE OF REPORT General Document	7. DOCUMENT DATE May, 2019
8. OBJECTIVE ID	9. TASK NUMBER	10. TASK SPONSOR	
11. MSTC		12. STC	
13. DOWNGRADING/DELIMITING INSTRUCTIONS		14. RELEASE AUTHORITY Chief, Cyber and Electronic Warfare Division	
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT *For Public Release.* OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SA 5111			
16. DELIBERATE ANNOUNCEMENT No Limitations			
17. CITATION IN OTHER DOCUMENTS No Limitations			
18. RESEARCH LIBRARY THESAURUS machine learning, deep learning, software vulnerability, cyber-security			
19. ABSTRACT This report is the result of a scoping study undertaken as part of an Australian Defence Department Next Generation Technologies Fund (NGTF) project entitled *Deep Learning for Cyber-Security* (DLC). The report provides a motivation for the study of software vulnerability discovery, briefly reviewing existing techniques for both source and binary code, with an emphasis on machine learning approaches. Noting the spectacular successes in recent years of Deep Learning (DL) techniques in areas such as image recognition, it is proposed to investigate the application of DL techniques to the software vulnerability discovery problem, with a focus on binary code analysis as most relevant to Defence. As part of this effort, consideration is given to the acquisition and generation of suitable training and testing datasets.			