This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

INVITED PAPER

Software Vulnerability Detection Using Deep Neural Networks: A Survey

By Guanjun Lin, Sheng Wen, *Member IEEE*, Qing-Long Han, *Fellow IEEE*, Jun Zhang, *Senior Member IEEE*, and Yang Xiang, *Fellow IEEE*

ABSTRACT | The constantly increasing number of disclosed security vulnerabilities have become an important concern in the software industry and in the field of cybersecurity, suggesting that the current approaches for vulnerability detection demand further improvement. The booming of the open-source software community has made vast amounts of software code available, which allows machine learning and data mining techniques to exploit abundant patterns within software code. Particularly, the recent breakthrough application of deep learning to speech recognition and machine translation has demonstrated the great potential of neural models' capability of understanding natural languages. This has motivated researchers in the software engineering and cybersecurity communities to apply deep learning for learning and understanding vulnerable code patterns and semantics indicative of the characteristics of vulnerable code. In this survey, we review the current literature adopting deep-learning-/neural-network-based approaches for detecting software vulnerabilities, aiming at investigating how the state-of-the-art research leverages neural techniques for learning and understanding code semantics to facilitate vulnerability discovery. We also identify the challenges in this new field and share our views of potential research directions.

KEYWORDS | Cybersecurity; deep neural network (DNN); machine learning (ML); representation learning; software vulnerability.

I. INTRODUCTION

Computer systems form the very foundation of our digital society, empowering people, and businesses worldwide. However, exploitable vulnerabilities in software can pose potential threats to the secure operation of computer systems, and impact on millions of users on a daily basis. This has been proved by the "Heartbleed" [1] and "Shellshock" [3] vulnerabilities disclosed five years ago, and in 2017, the vulnerability in Apache Struts which resulted in 143 million consumers' financial data to be compromised [5], [15], [125], [126].

One of the effective attack mitigation solutions is to apply software vulnerability detection prior to the deployment. The code analysis techniques which are fundamental for vulnerability detection can be categorized into static, dynamic, and hybrid approaches. Static techniques, such as rule/template-based analysis [16], [29], [113], code similarity detection [42], [45], and symbolic execution [13], [83], [106], mainly rely on the analysis of source code, but often suffer from high false positives [56]. Dynamic analysis includes fuzz testing [81], [103] and taint analysis [74], [82] and often have issues with low code coverage. Hybrid approaches combine static and dynamic analysis techniques with the aim to overcome the aforementioned weaknesses. However, they also inherit their drawbacks, and are inefficient to operate in practice [119]. To date, the rapid increase in the number of vulnerabilities disclosed after the release of software products

Manuscript received January 30, 2020; accepted May 4, 2020. *(Corresponding author: Jun Zhang.)*

Guanjun Lin is with the School of Information Engineering, Sanming University, Sanming, Fujian 365004, China (e-mail:daniellin1986d@gmail.com).

Sheng Wen, **Qing-Long Han**, **Jun Zhang**, and **Yang Xiang** are with the School of Software and Electrical Engineering, Swinburne University of Technology, Melbourne, VIC 3122, Australia (e-mail: swen@swin.edu.au; qhan@swin.edu.au; junzhang@swin.edu.au; yxiang@swin.edu.au).

Digital Object Identifier 10.1109/JPROC.2020.2993293

0018-9219 © 2020 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

suggests that current techniques for code inspection and software quality checking require further improvement in terms of efficiency and effectiveness [108].

Data-driven vulnerability discovery using pattern recognition and machine learning (ML) provides alternative solutions for automated and potentially more efficient vulnerability discovery [21], [33], [63], [101]. The ML algorithms can learn latent and abstract vulnerable code patterns, with a potential of significantly improved level of generalization. Existing ML-based approaches mainly operate on source code which provides better human readability. Researchers have applied source-code-based features such as imports (e.g., header files), function calls [73], software complexity metrics, and code changes [94]–[96] as indicators for identifying potentially vulnerable files or code snippets. Furthermore, features and information obtained from version control systems such as developer activities [66] and code commits [79] were also adopted for predicting vulnerabilities.

The application of conventional ML techniques for vulnerability detection still requires experts to define features, which primarily depends on human experience, level of expertise, and depth of domain knowledge [43], [56], [117]. The feature engineering processes can be time-consuming and possibly error-prone, and the resulting handcrafted features can also be objective and task-specific. The emerging deep learning techniques provide new potential to the field of ML-based vulnerability detection. On one hand, the layered structure of neural-network-/deep-learning-based models helps facilitate the learning of abstract and highly nonlinear patterns, capable of capturing the intrinsic structure of complex data. On the other hand, the use of neural networks allows the features to be extracted automatically and with multiple levels of abstraction [53], and possibly with an improved level of generalizability [61], thus relieving experts from labor-intensive and possibly error-prone feature engineering tasks. In addition, the deep learning approaches are capable of discovering latent features that a human expert might never consider to include [89], which significantly expands the feature search space. Therefore, researchers have been motivated to apply the neural models for learning vulnerable code patterns that are indicative of potential vulnerabilities in applications.

Although significant research effort has been put into developing automated vulnerability detection solutions, human intelligence still plays an important role in guiding and/or supplementing the development/usage of different types of tools and approaches to aid the vulnerability search process [108]. Many static tools and ML-based detection systems were developed based on the experience of knowledgeable security experts/testers to approximate how experts/testers search for vulnerabilities. For example, early rule-based static code analysis tools extract rule templates based on best programming practices. Violations of the predefined rules will be alerted. ML-based detection approaches are also heavily dependent on carefully engineered feature sets extracted from code analysis, e.g., abstract syntax trees (ASTs), and control-flow graphs (CFGs). In a sense, the feature engineering processes reflect what experts/testers think are the most important characteristics which reveal the vulnerabilities in software. Nevertheless, a detection system cannot in the foreseeable future search for vulnerabilities like experts/testers do due to the difficulty for experts/testers to transform their knowledge and understanding of vulnerabilities to feature vectors learnable by a detection system. What the system learns from the feature sets can also be biased by various factors (e.g., the expressiveness of the model, data overfitting, noises in the data, etc.).

The goal of this research is to enable vulnerability detection systems to perform more accurately and effectively like an experienced expert or tester does and in an automated and efficient manner. Therefore, there is a gap to bridge so that an ML-based detection system can eventually learn and understand the vulnerable code semantics as humans do. In this article, we define this gap as the semantic gap, and which will be thoroughly discussed in Section II-A2. Thus far, researchers of ML, software engineering, and cybersecurity communities have been endeavoring to bridge this gap. Particularly, with deep learning techniques being applied for vulnerability detection, the neural models [e.g., the recurrent neural networks (RNNs)] are capable of reasoning and understanding of code semantics [7]. Compared with conventional ML-based approaches that fully rely on handcrafted features, deep learning approaches have the potential to take one step further to narrow the semantic gap by learning intricate patterns and high-level representations of software codes that reveal code semantics.

In this article, we review a recent work, which applied deep-learning-/neural-network-based approaches for vulnerability discovery, and evaluate how different types of feature representations learned by neural networks help to capture the characteristics of vulnerable code snippets, thus contributing to bridging the semantic gap. We also identify weaknesses of the reviewed studies and questions unanswered in this research field, aiming to propose possible future research directions.

A. Structure

This article is organized as follows. In Section II, we first provide the definitions of software vulnerability summarized from previous studies. Then, we discuss the semantic gap between two parties: the human code inspector and the existing detection systems. Finally, we briefly review how conventional ML-based studies managed to bridge the semantic gap. In Section III, we provide some background knowledge of different types of neural networks applied by the reviewed studies and categorize them into four categories based on different types of feature representations learned by the neural networks. We review studies on each category to reveal how each work process the code and utilize the neural network to learn vulnerable

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

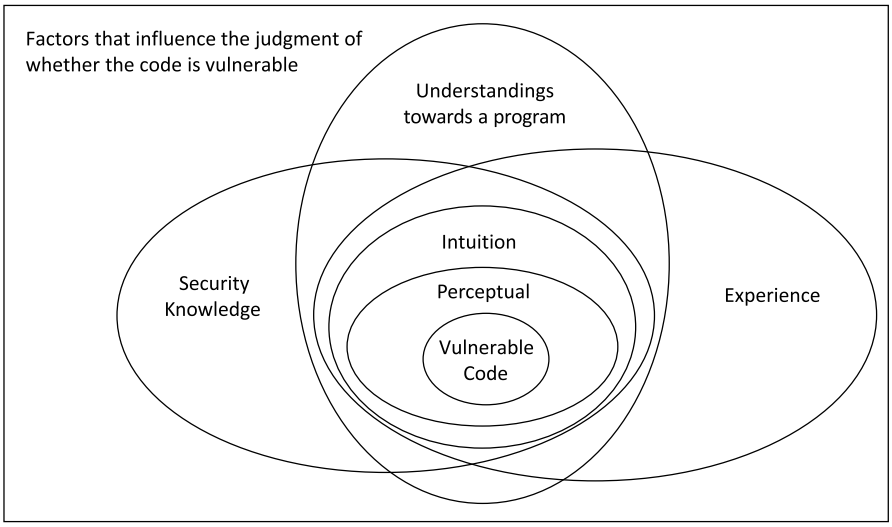


Fig. 1. ***When a practitioner examines a code snippet, determining whether it contains a potential vulnerability depends on a number of factors. Essentially, the practitioner needs to understand the code semantically (e.g., the meanings of statements) and syntactically (e.g., the correct form of the code structure). Viewing the code, the practitioner may check whether there is a missing of bounds check for a variable (i.e., the cause of buffer errors). His/her experience in code inspection and security knowledge helps him/her to think from the perspective of a software tester or an attacker, which may provide a feeling or intuition to pay more attention to one piece of code rather than the other. For instance, whether a source variable (e.g., an input value) can be controlled by an attacker. His/her knowledge and familiarity toward the program also facilitate his/her understanding of code dependences. For instance, a practitioner knows the caller and/or caller of a target function, which allows his/her to perform interprocedural analysis easily.***

code patterns and semantics. At the end of each subsection, we discuss the data sets used by each work, its possible repeatability, and the characteristics of each study shared in the category. We also identify the limitations of studies in each category. In Section IV, we list challenges in this field and share insights into possible future research directions. We conclude this article in Section V.

B. Related Reviews

In the field of software engineering, various techniques have been proposed and investigated for defects/vulnerabilities discovery. There have been several survey articles providing systematic reviews of many software defect/vulnerability detection approaches from various perspectives. Shahriari and Zulkernine [90] reviewed vulnerability detection studies using different code analysis techniques. Malhotra [64] provided reviews of software defect prediction studies which applied ML techniques. A short survey by Liu *et al.* [62] generally reviewed vulnerability detection studies which use both code analysis and ML-based techniques. Ghaffarian and Shahriari [33] presented an extensive review of the studies which focus solely on conventional ML techniques for vulnerability detection. The reviewed studies are primarily based on the features extracted from software metrics, manual-defined vulnerable patterns, and anomaly patterns. Another similar survey [44] also builds on the summary of these studies. An extensive review [7] has provided a systematic study from the perspective of comparing programming languages with natural languages and discussing the motives of model design based on the similarities and differences of code and languages. In this article, we focus on reviewing the work that applies deep-learning-/neural-network-based techniques for vulnerability detection, emphasizing the different feature representations obtained by neural networks for learning the characteristics of vulnerable code snippets for better understanding the code semantics.

II. VULNERABILITY SEMANTIC GAP

A. Software Vulnerability: A Different Perspective

1) Definition: Software security vulnerabilities (short for vulnerabilities), also known as security defects [89], security bugs [108], and software weaknesses [54], are briefly defined as "software bugs that have security implications" [87]. Votipka *et al.* [108] categorize bugs into three categories: 1) performance-related; 2) functionality-related; and 3) security-related, and also suggests that software vulnerabilities can be seen as a subset of software defects or bugs [83]. Once these security-critical bugs are exploited by attackers, it may result in the violation of security policies, leading to security failure [20]. After summarizing different definitions of vulnerabilities from previous studies, Ghaffarian and Shahriari [33] defined vulnerabilities as follows:

> An instance of a flaw, caused by a mistake in the design, development, or configuration of software such that it can be exploited to violate some explicit or implicit security policy.

2) Human Inspector and Detection System, the Semantic Gap: Although the abovementioned definition provides a general understanding of what a vulnerability is,

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

identifying vulnerabilities in software is a challenging task. It requires a code inspector or security expert (collectively, practitioner) to be equipped with sufficient experience and knowledge of the program, as well as the depth of understanding toward the programming language [108], which collectively form a high-level understanding of the code semantics. In this article, we argue that this high-level understanding facilitates the search of a possible vulnerable code snippet in the manual code auditing tasks, as Fig. 1 shows. However, the ML-based detection systems can only attempt to approximate the process of a practitioner judging whether a code snippet contains a possible vulnerability by learning from the feature sets either manually extracted (i.e., the studies based on conventional ML algorithms) or automatically extracted (i.e., the deep learning-based studies) from the code (e.g., source code and assembly). The detection systems are incapable of fully understanding the underlying semantic meanings of the vulnerable code patterns as a practitioner does. Hence, there is a semantic gap between a knowledgeable and experienced practitioner and the ML-based detection systems, which we define as follows.

> The semantic gap is the lack of coincidence between the abstract semantic meanings of a vulnerability that a practitioner can understand and the obtained semantics that an ML algorithm can learn.

Therefore, the goal of the research in this field is to develop ML-based detection systems that can perform vulnerability detection aligning with what experienced and knowledgeable practitioners do, but in a more scalable, efficient, and automated manner.

From a practitioner's perspective, successful detection of vulnerabilities in the process of manual code inspection not only requires the understanding of the code semantics and syntactics but also the profound knowledge of the code base, the practice of secure coding, and the experience of vulnerability detection [109], [121], as Fig. 2 shows. Take the "Heartbleed" vulnerability (CVE-2014-0160) [1] as an example with the vulnerable code snippet shown in the following code snippet:

```
1  /* ssl/d1_both.c */
2  // \ldots
3  int dtls1_process_heartbeat(SSL *s)
4  {
5  unsigned char *p = &s->s3->rrec.data[0], *pl;
6  unsigned short hbtype;
7  unsigned int payload;
8  unsigned int padding = 16; /* Use minimum padding */
9  /* Read type and payload length first */
10 hbtype = *p++;
11 n2s (p, payload);
12 pl = p;
13 if (s->msg_callback)
14 s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
       &s->s3->rrec.data[0], s->s3->rrec.length, s,
       s->msg_callback_arg);
15 if (hbtype == TLS1_HB_REQUEST) {
16 unsigned char *buffer, *bp;
17 int r;
18 // \ldots
19 buffer = OPENSSL_malloc (1 + 2 + payload + padding);
20 bp = buffer;
21 /* Enter response type, length and copy payload */
22 *bp++ = TLS1_HB_RESPONSE;
23 s2n (payload, bp);
24 memcpy (bp, pl, payload);
25 bp += payload;
26 /* Random padding */
27 RAND_pseudo_bytes (bp, padding);
28 r = dtls1_write_bytes (s, TLS1_RT_HEARTBEAT, buffer,
       3 + payload
       + padding);
29 // \ldots
30 }
31 // \ldots
32 return 0;
33 }
```

The Heartbleed vulnerability in OpenSSL.

This vulnerability is caused by missing a bounds check of the variable payload and is related to several intermittent lines of code. First, payload is declared in line 7 and defined by the application program interface (API) function call *n2s* in line 11. In line 19, it is assumed that payload is within bounds, so the function OPENSSL_malloc allocates a memory block with a buffer size of payload length, plus padding length, and plus 3. In line 23, variable payload is passed to API function call s2n and then, library function memcpy is called in line 24, which copies the source buffer pl of payload size to the buffer bp. Eventually, what stores in variable payload is sent out to the network via a function call dtlsl_write_bytes in line 28. A practitioner who inspects this code and finds the vulnerability should be able to correctly identify the flow of variable payload and how it is manipulated in the data flow. Particularly, this requires the practitioner to know the implementation of API calls *n2s* and *s2n* to determine how variable payload changes. Based on this understanding and the experience, the practitioner might feel the necessity to validate the bounds of variable payload before calling memcpy and may investigate further (e.g., performing taint analysis).

From the perspective of designing an ML-based vulnerability detection system, to pass the knowledge (i.e., the familiarity of a program's code) and the understanding of a code base to an ML-based detection system, practitioners have to convert their knowledge and understanding into features optimized for vulnerability discovery so that the ML-based detection systems can learn from these features. However, practitioners can describe the cause of the abovementioned vulnerability and how they find it, converting their description and understanding to numerical feature sets learnable by ML algorithms, which thus far has been a challenging task that many researchers are still exploring. On one hand, translating a human understanding of vulnerabilities into numeric features is an abstraction and quantification process, which would incur information loss and introduce bias. On the other hand, when we apply ML algorithms to train the model based on the extracted features, the model may deviate from our expectations due to various reasons (e.g., underfitting/overfitting, the noise of the data, etc.).

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

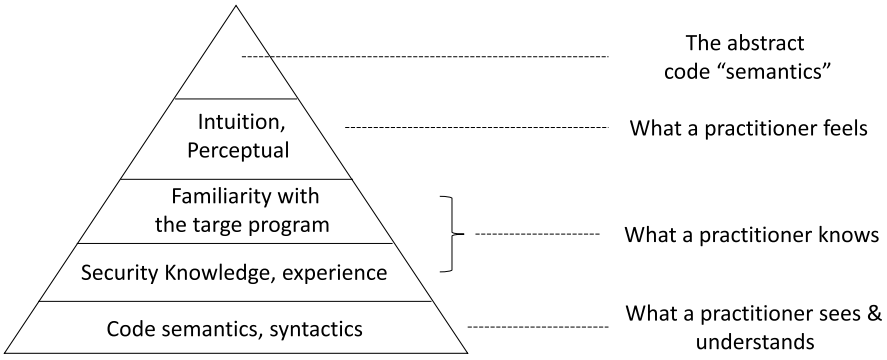


Fig. 2. ***"Code semantics" is the practitioner's high-level understanding toward a piece of code, which may build on: 1) the understanding of actual code semantics and syntactics; 2) the knowledge of secure coding practices and the experience of code inspection/testing; 3) the familiarity with the project; and 4) the objective feeling/intuition. These form the abstract semantics of the code in a practitioner's mind, which facilitates the code auditing.***

B. From Conventional ML to Deep Learning

1) Conventional ML-Based Solutions: Many ML-based solutions have been introduced to bridge the gap by proposing various feature sets for the detection systems to learn from, attempting to approximate the process of how a practitioner searches for vulnerabilities. An extensive survey [33] has reviewed different approaches that applied conventional ML techniques (nonneural network-based) for vulnerability analysis and discovery. The authors categorized these approaches into three major categories: 1) software metrics-based; 2) vulnerable code pattern-based; and 3) anomaly-based. In this survey, from a different perspective, we would like to discuss how the studies categorized in the three groups manage to fill the semantic gap.

The software metrics such as McCabe [65] and Code Churn [70] metrics are numerical indicators of the quality of software products from a different point of view. The McCabe metrics are software complexity metrics. The vulnerability detection models using these metrics build on the assumption that a complex code is difficult for a practitioner to comprehend and consequently hard to maintain and test. The Code Churn metrics used as indicators of vulnerability detection implies that frequently modified code tends to be erroneous, therefore is more likely to be defective and possibly vulnerable, considering that vulnerabilities are a subset of software defects. Although these metrics have a long history in measuring software quality and are used for defect prediction, they are not direct indicators of vulnerabilities. For example, these metrics do not provide measurements for code security analysis [69], and experience shows that not all vulnerable code blocks (e.g., files or functions) contain many lines of code and logically complex. Hence, many studies that exclusively relied on software metrics as features to build ML models reported poor results on vulnerability detection [33]. From the perspective of how a practitioner performs code auditing, factors such as the complexity of the code can be taken into consideration for further checking, but these factors will not be the decisive ones for a practitioner to determine whether a piece of code is vulnerable. Hence, other feature sets that can better depict the characteristics of vulnerable code snippets are needed to train more effective ML classifiers to facilitate vulnerability detection.

There are some studies that applied text-mining techniques for extracting code patterns from the source code context, and the patterns of code are jointly depicted by the frequency of different textual tokens in the source code. For instance, the bag-of-words technique was applied to convert tokens in the source code to vector representations which were then used as feature sets for classification [88]. To consider a sequence of tokens that may interrelate, the N-gram technique was adopted for the mining program's source code [76]. However, some researchers argued that the frequency-based text-mining techniques failed to capture the rich semantic meanings of software code because they "simply smoothed counts of term co-occurrences" and the long-term contextual dependences of code could not be preserved [114]. Other researchers [110] also pointed out that two Java files, which contain exactly the same tokens and the same frequency of each token, can have distinct code semantics.

Compared with the "flattened" feature sets mined from the surface text of source code, many studies applied static code analysis tools to extract "structured" feature sets as code patterns for ML algorithms to learn from. These feature sets were extracted from different forms of program representations generated by static analysis, including the ASTs, CFGs, data-flow graphs (DFGs), program dependence graphs (PDGs), and so on. Compared with software metrics and the frequency-based code features, feature sets derived from the outcomes of code analysis tools and parsers reveal more information about the code because each form of the program representation provides a view on the source code from different aspects. For instance, the ASTs represents code structure in a tree view. The CFG, DFG, and PDG help to analyze how a variable flows from source to sink and can be used for building variable

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

dependences and understanding the program structure. Particularly, an approach provided by Yamaguchi *et al.* [119] combined AST, CFG, and PDG into a joint representation called the code property graph (CPG) to provide a more comprehensive graph structure to depict the properties of code. In addition to the feature sets derived from structured program representations, the program execution trace obtained from the dynamic analysis was also used as features [91]. Combined with the static feature sets extracted from the CFG and DFG, their model yielded better performance compared with their previous study [92]. Therefore, the extraction of these feature sets based on the high-level source code structures can reflect how a practitioner finds vulnerabilities to a large extent.

There is a line of studies that considered vulnerable code patterns as the anomalous patterns which deviate from the correct and normal code patterns. For these studies, instead of considering how to find a vulnerability, the authors emphasized on what may cause a vulnerability. Early studies such as [29], [57], [124], and [111] applied rules derived from the experience of knowledgeable individuals as "templates" for detecting potential buggy/vulnerable code that did not conform to programming guidelines or best practices. Later studies applied features extracted from abnormal API usage patterns [6], imports, and function calls [73], and API symbols missing checks [122] for vulnerability detection. However, these feature sets were usually linked to a small set of vulnerabilities or a particular type of them. For example, the features extracted from imports and function calls can only train a classifier which detects vulnerabilities introduced by some vulnerable header files or libraries, and the API symbols associated with missing checks can only be used for discovering vulnerabilities caused by the lack of validation or boundary check. These rules or feature sets generated by anomalous code patterns can either be general such as "calling to a lock function indicates that an unlock function should be called afterward" or task/project-specific, for example, the features generated from import or function calls may not be applicable for every software projects and solutions for missing check detection are designed for mature software projects. Hence, solutions proposed by these studies may result in many false positives or only be suitable for task-specific applications.

2) Emerging Deep Learning Techniques: The application of conventional ML models for vulnerability detection tasks generally relies on handcrafted features extracted from static and/or dynamic code analysis to guide the model to learn the vulnerable code patterns. The breakthrough application of deep neural networks (DNNs) to speech recognition [25], image recognition [49], machine translation [9], [102], and many other areas has enabled DNNs to become the foundation for many state-of-the-art artificial intelligence (AI) applications [104]. To date, DNNs have been applied for automated vulnerability discovery and the detection performance is promising. Compared with the conventional ML techniques, in the applications of vulnerability detection, deep learning techniques:

1) enable the learning of high-level features or representations with more complexity and abstraction [104], compared with conventional ML algorithms;
2) are capable of automatically learning latent features or representations which are more generalizable, thus relieving practitioners from labor-intensive, subjective, and error-prone feature engineering tasks [56], [61];
3) provide flexibility to allow network structures to be customized for different application scenarios. For instance, combining the long-short term memory (LSTM) network with dense layers for learning function-level representations as high-level features [56], [61], implementing attention layers for learning feature importance [8], and adding external memory "slots" for capturing long-range code dependences [18], [89].

The abovementioned characteristics of deep learning techniques allow the researchers to build detection systems that can capture code semantics, understand code contextual dependences, and automatically learn high-level feature sets which are more generalizable. With these capabilities, the systems built can better "understand" the semantics and the context of code, and, therefore, would further reduce the semantic gap.

III. FEATURE REPRESENTATIONS FOR BRIDGING THE SEMANTIC GAP

The rapid growth of the application of deep learning techniques in various fields has proved their effectiveness in learning complex, latent, and abstract patterns. The recent breakthrough in the field of natural language processing (NLP), such as the machine translation [9], [102] and language understanding [26], has also demonstrated the capability of neural network models in understanding the underlying rich "semantics" of text/language data. In addition, early researchers discovered that natural language models were also effective for processing software source code [39], [40], which motivates many researchers to apply deep learning techniques for code analysis tasks.

A. Neural Models Facilitating Representation Learning

In this article, we specifically focus on reviewing the literature that applied DNNs for software vulnerability detection. Among the reviewed studies, different types of network structures are used for extracting the abstract features from various types of inputs, which we call the feature representations, for identifying the semantic characteristics of vulnerable code snippets.

1) *The fully connected network (FCN):* The FCN which is also called multilayer perceptron (MLP) was chosen by many pioneer researchers as the model for vulnerability detection primarily based on handcrafted

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

numeric features. This category of works treated the network as a highly nonlinear classifier for learning hidden and possibly complex vulnerable patterns. Compared with conventional ML algorithms, such as random forest, support vector machine (SVM), and C4.5, the FCN can fit the patterns that are highly nonlinear and abstract. This is supported by the universal approximation theorem [23] that an FCN with one single hidden layer with a finite number of neurons can approximate any continuous functions [22]. Thus, the FCN has the potential of learning more richer models than conventional ML algorithms given a large data set. This potential has motivated researchers to use it for modeling the vulnerable code patterns which are latent and complex. Another advantage of the FCN is that it is "input structure-agnostic," which means that the network can take many forms of input data (e.g., images or sequences) [84]. This also offers researchers the flexibility to handcraft various types of features for the network to learn from.

2) *Convolutional neural network (CNN):* The CNN was designed to learn structured spatial data [84]. In image processing, the filtering operations in the first layer of the CNN are capable of learning features from nearby pixels that are semantically similar. Then, the succeeding filters[1] will learn higher level features which will be used by the subsequent layers (i.e., the dense layers) for classification. The capability of CNN being able to learn features from nearby pixels can also facilitate NLP tasks. For instance, in the task of text classification, the filters of CNN which are applied on a context window (i.e., containing a few word embeddings) can project words within the context window to local contextual feature vectors in the contextual feature space where the vectors of semantically similar words are in close proximity [123]. Hence, CNN can capture the contextual meanings of words, which motivates researchers to apply CNN for learning vulnerable code semantics that is context-aware.
3) *Recurrent neural network:* Compared with the feed-forward networks, such as the FCN or CNN, the RNN is naturally designed for processing sequential data (e.g., text). Therefore, a large body of studies applied variants of RNN for learning semantic meanings of vulnerabilities. Particularly, the bidirectional form of RNNs is capable of capturing the long-term dependences of a sequence. For this reason, many studies utilized the bidirectional LSTM (Bi-LSTM) [41] and gated recurrent unit (GRU) [17] structures to learn code contextual dependences which are crucial for understanding the semantics of many types of vulnerabilities (e.g., buffer overrun vulnerabilities). These vulnerabilities are associated with code context containing multiple consecutive or intermittent lines of code that form a vulnerable context.
4) *The miscellaneous types:* There are some vulnerability detection studies applied to other network structures which do not fit in the aforementioned types such as the deep belief network (DBN) [110] and variational autoencoders (VAEs) [52]. Another promising characteristic of deep learning techniques is that the network structure can be customized to cater for different application scenarios. For instance, the researchers [18], [89] applied the memory network [100], [112] which equips with external memory "slots" for storing previously introduced information for future access [18]. Compared with the LSTM structure, this type of network is capable of capturing the sequence dependence of longer range; therefore, it has the enhanced capability for capturing of the longer range of code sequences that are key for recognizing buffer overrun vulnerabilities being context-dependent.

B. Previous Work Categorization

We categorize the reviewed studies into four categories of feature representations, which are summarized as follows.

1) *The graph-based feature representations:* A large body of studies apply DNNs for learning feature representations from different types of graph-based program representations, including the AST,[2] the CFG, the PDG, and a combination of them.
2) *The sequence-based feature representations:* Studies in this category utilize DNNs for extracting feature representations from sequential code entities, such as the execution trace, the function call sequences, and the variable flow/sequence.
3) *The text-based feature representations:* For this category of works, the feature representations are directly learned from the surface text of the code.
4) *The mixed feature representations:* This category includes recent studies that combined the abovementioned three types of feature representations.

The rationale behind the proposed categorization is twofold. On one hand, the contributions of these studies lie on how to process the software code to generate feature representations that facilitate the DNN's understandings of the code semantics and capture the patterns as indicators of potentially vulnerable code snippets. On the other hand, a DNN model works as a classifier with built-in representation learning capability. The existing studies which are based on different types of feature inputs allow the DNNs to obtain high-level representations revealing different semantic information. For example, the studies which use ASTs as the input of the neural model enable the DNNs

[1]The filter is also known as the convolutional kernel.

[2]In this article, we consider the tree as a special case of a graph.

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

to capture the code patterns and semantics through the hierarchical structure of the program. The studies applying function call sequences allow DNNs to learn abstract features associated with the function call patterns.

In Sections III-C–III-F, we will review each category of work that applied different feature representations to facilitate the detection of vulnerabilities. We also discuss the data sets used by each work, their possible repeatability, the characteristics shared by each category, and/or limitations.

C. Graph-Based Feature Representations

In this section, we review studies that use various types of graph-based program representations, including the AST, CFG, PDGs, and data dependence graphs (DDGs) as inputs to the DNNs for learning the deep feature representations.

1) Summary of Recent Works: An approach for detecting SQL injection (SQLI) and cross-site scripting (XSS) vulnerabilities of PHP-based web applications was developed in [93]. It was a fine-grained solution targeting statement-level detection. The method was motivated by the observation that the implementation of input sanitization code was crucial for the prevention of SQLI and XSS vulnerabilities. Based on the observation, they proposed a set of static code attributes which characterize the sanitization code patterns from the CFGs and DDGs. Specifically, the authors identified the inputs and their corresponding sinks using CFGs to generate the input attributes, and the DDGs were used to locate the functions that performed input sanitization. Based on the identified sanitization functions, the authors classify some sanitization attributes. These static code attributes formed 20-D feature vectors to depict the sanitization patterns. Then, the authors applied C4.5, Naive Bayes (NB), and FCN as classifiers for training and classification. Their empirical study showed that an averaged result of 93% recall and 11% false alarm rate were achieved in detecting SQLI vulnerabilities, which was slightly better than the prediction result achieved for XSS vulnerabilities being 78% recall and 6% false alarm rate. The authors also reported that the FCN outperformed the other two conventional ML classifiers on their data sets.

The first study utilized ASTs for learning neural representations for detecting defects and vulnerabilities from Java source code was proposed in [110]. The assumption was that the ASTs represented the code syntax containing the programming patterns, and the code semantics which was hidden in the source code can be revealed by analyzing ASTs. When extracting ASTs from the source code, the authors preserved three types of nodes: 1) nodes of function invocation and class instance creations; 2) nodes of declarations; and 3) control-flow nodes. Then, the nodes were converted to token sequences. In the scenario of cross-project detection, project-specific tokens were replaced by general names such as method declaration/invocation. In particular, the authors applied a distance similarity computation algorithm [72] to remove noise from their data. The algorithm was able to calculate distances among the token sequences based on the tokens and the order among the tokens. After this, the k-nearest neighbor algorithm was used to the calculated distances. If a certain instance had opposite labels to its neighbors, the instance would be fagged as noise and removed. Finally, the remaining token sequences were tokenized by using a one-to-one mapping table to map each token to an integer so that the sequences can be used as inputs to the DBN. Similar to existing studies such as [38], [61], and [86], the authors fed the samples to the trained DBN for automatically generating high-level feature representations which were then used to train conventional ML algorithms such as ADTree, NB, and logistic regression (LR).

For the evaluation of within-project detection, the authors chose the Java projects from the PROMISE[3] defect repository and compared their proposed approach with two different feature sets: 1) the software metrics and 2) the ASTs. The results showed that the AST-based features generated by DBN achieved the best performance on all the chosen projects. For the cross-project scenario, the authors chose the TCA+ [71] as the baseline system. The performance evaluation showed that in 17 out of 22 cross-project scenarios, the DBN-based method outperformed the TCA+ in terms of the F1 scores.

Another AST-based approach proposed in [60] applied the Bi-LSTM network to learn feature representations for cross-project vulnerability discovery. Considering that obtaining manual labels of vulnerable functions can be expensive in practice, the authors leveraged software complexity metrics which could be automatically generated as a proxy to substitute the actual labels for generating high-level representations at function-level. Their approach was built on the assumption which was shared by many studies, i.e., to use software code metrics (e.g., the code complexity metrics) as features for vulnerability detection because a complex code is difficult to test and maintain; therefore, it is more likely to be defective and vulnerable. Specifically, the authors extracted raw features from the ASTs of source code functions. They used "CodeSensor" [120], which allowed them to obtain ASTs without a working build environment for AST extraction. Then, the obtained ASTs were traversed using the depth-first traversal (DFT) to convert them to sequences. To perform cross-project detection, the authors blurred the project-specific AST tokens so that the left tokens were project-agnostic. Subsequently, each token within the AST sequences was converted to integers and was mapped to a 64-dimensional embedding vector which was fed to the proposed Bi-LSTM network and fine-tuned during the network training. The network consisted of two Bi-LSTM layers for learning the contextual information from the embedded AST sequences. Followed by the Bi-LSTM layers, there were two fully connected layers

[3]http://promise.site.uottawa.ca/SERepository/index.html

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

to form a complete network. During the training phase, the authors chose "essential complexity" as the substitutes of actual labels to train the network. When training was completed, the authors fed the AST sequences to the network and obtained the output of the second last layer as the learned high-level representations which were then used as features for training a random forest classifier. The authors used the functions collected from two open-source projects: FFmpeg and LibTIFF as the training set, and open-source project LibPNG as the test set for experiments. Different from previous studies, the authors used the ranked results which were based on the prediction probabilities of each function being vulnerable, and they found that when retrieving 200 most probably vulnerable functions, 47% of them were actually vulnerable. Considering that the representations were obtained with no label information provided, this article suggested a possible new method for vulnerability feature extraction using code complexity metrics as a substitute for actual labels.

Later, a method was proposed by learning transferable representations from vulnerability data of historical software projects for detecting vulnerabilities on a target project which had insufficient labeled data for training a robust ML model [61]. Similarly, the authors used functions' ASTs as the raw features, and ASTs were converted to sequences of tokens based on the DFT to partially preserve the structural information. Then, they applied Word2vec [67] with the continuous bag-of-words (CBOW) model to transfer each element of the sequences to embeddings of 100 dimensions for restoring the code semantic information. Then, the authors applied the Bi-LSTM network to facilitate the learning of the project-independent representations that were indicative of vulnerability detection from both the structural and semantic information preserved in the ASTs. Compared with the network used in [60], the network they proposed consisted of only one Bi-LSTM layer for learning contextual information of the AST sequences and followed by a global max-pooling layer and two fully connected layers. The functionality of the global max-pooling layer was similar to the max-pooling layer in the CNN structure for capturing the most important signals related to the potential vulnerabilities. The authors manually labeled 457 vulnerable functions from six open-source projects based on the vulnerability databases provided by the National Vulnerability Database (NVD)[4] and the Common Vulnerability and Exposures (CVE),[5] and they also collected 14 648 nonvulnerable functions to from the experimental data sets. To simulate the scenarios, using the historical data for transfer learning, the authors divided their data sets by using five of the projects as the source projects, and one as the target project. Then, they further divided the target project into training and test sets within a 1-to-3 ratio to simulate the scenario where there usually was insufficient labeled data for vulnerability discovery for new open-source projects. During the training phase, the data of the source project were used to train the proposed network. When the training was completed, the authors fed the training and test sets of the target project to the trained network and obtained the output of the second last layer as the learned deep AST transfer representations. Finally, the obtained representations were used for training a random forest classifier. The empirical study showed that compared with using the 23 software code metrics as features, the learned representations from the proposed network were more effective.

An AST-based study was proposed in [24] for utilizing a sequence to sequence (seq2seq) LSTM network for file-level vulnerability detection on Java source code. The authors claimed that the network was able to automatically learn the semantic and syntactic features from ASTs. The ASTs were extracted from source files and converted to sequences of tokens so that each sequence corresponded to a function. To transfer tokens to embeddings acceptable to the LSTM network, the authors constructed a lookup table to map each token to a fixed-length vector. Then, code sequences could be represented as vector representations to be fed to the seq2seq LSTM network. The seq2seq network took one sequence at a time, and each token was input into an LSTM unit. Given an input token x_t within an input sequence $[x_1, x_2, \ldots, x_n]$, the LSTM unit computed the output state vector s_t, and the state vectors were used to predict the next tokens. Given the output state vector of code token w_t, the network was trained to generate the next code token w_{t+1} from a context of earlier code tokens $w_{1:t}$. Thus, during the training process, the network was able to learn the relationships of token sequences. To generate the syntactic features that represented the sequential structure of the source code, the authors fed the embeddings of code token sequences to the trained network and applied mean pooling to the output of the network. By doing this, all the token states in the same sequence were aggregated and stored in the vector representation generated by the mean pooling layer, and each function-level vector was then assembled to form a single vector for a file, which could be used as features for the within-project vulnerability detection.

Due to the resulting feature vectors being project-specific, the authors constructed a codebook which summarized all token states (i.e., the semantic space) for performing cross-project detection. The codebook contained tokens across all projects. The states of the tokens were obtained using k-means to cluster all state vectors in the training set. Given a new file, the authors computed the distances between all the state vectors within the file and the closest centroids of each cluster to generate a feature vector for this file, which could be used as the feature for cross-project prediction. The authors collected 18 open-source Android projects for evaluating

[4]https://nvd.nist.gov/

[5]https://cve.miter.org/

[6]https://f-droid.org/

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Table 1 Reviewed Studies Which Applied Graph-Based Feature Representations for Vulnerability Detection

Paper	Data	Labels Obtained from & Provided by	Features Representations	Detection Granularity
Shar and Tan [93]	Source code of open source PHP projects	Manually labeled & provided by security advisories	Hancrafted features from CFGs & DDGs	Statement-level
Wang et al. [110]	The PROMISE defect data sets	The PROMISE	ASTs at function-level & network generated semantic representations at file-level	File-level
Lin et al. [60]	Three open source projects from Github written in C language	Manual effort	ASTs at function-level in serialized format	Function-level
Lin et al. [61]	Six open source projects from Github written in C language	Manual effort	ASTs at function-level in serialized format & High-level features generated by Bi-LSTM network	Function-level
Dam et al. [24]	Open source Android projects from F-driod [6]	F-driod	AST-based Syntactic & semantic features generated by seq2seq network, mean pooling & "codebook"	File-level
Li et al. [55]	C/C++ test cases from the SARD project & open source projects from Github	Automated & manual effort	Syntax characteristics derived from ASTs & semantic relationships from CFGs & PDGs	Intra-procedural & inter-procedural

the effectiveness of their approach. For comparisons, the authors selected different feature extraction methods including software metrics and bag-of-word technique, and a state-of-the-art method which implemented DBN for vulnerability detection [110]. In the within-project detection scenario, the proposed approach based on syntactic features achieved the best detection performance which was 93% in F-measure. In a cross-project scenario, the best performance (87% in F-measure) was accomplished by the proposed approach using the semantic features and the combination of using both syntactic and semantic features.

A systematic framework provided in [55] leveraged different types of RNNs (Bi-LSTM and Bi-GRU) for targeting the detection of multiple types of vulnerabilities. This framework aimed to explore methods which could better represent programs as vectors so that the code syntax and semantic information which reflected the characteristics of vulnerabilities could be preserved to facilitate vulnerability detection. The authors observed that most vulnerabilities exhibited some syntax characteristics which could be retrieved from programs' ASTs. With the Syntax-Based Vulnerability Candidates (SyVCs), the authors needed to transform them into Semantics-Based Vulnerability Candidates (SeVCs) which accommodated the statements that were semantically related to the SyVCs. The transformation of the SyVCs to SeVCs required the analysis of programs' CFGs. Then, the authors used program slicing techniques to derive the DFGs. Based on the CFGs and DFGs, the PDGs can be constructed. Next, the authors developed an algorithm to encode the statements which depicted the semantic relationship of the SyVC to vector representations and these representations were further converted to fixed-length embeddings using Word2vec [67], [68].

To investigate the performance of the proposed framework, the authors conducted a series of experiments based on the data sets collected from the NVD and the Software Assurance Reference Data set (SARD) project. The experiment on the 420 627 SyVCs, which corresponded to the four types of syntax characteristics, yielded an average of 86% of F1-measure using the Bi-LSTM network, which outperformed their previous work [56] by 5%. The authors also carried out experiments on five other types of neural networks: CNN, DBN, LSTM, GRU, and Bi-GRU for systematic performance comparison. The best performance was achieved on Bi-GRU which was 85.9% of F1-measure, and the worst performance was on DBN, being 17.8%. Based on the network of Bi-GRU, the authors compared the proposed framework with other state-of-the-art vulnerability detection systems, including Flawfinder [113], RATS [4], Checkmarx, VUDDY [45], and VulDeePecker [56], and their framework achieved the best detection performance.

2) Discussion: The previous section has reviewed six existing studies which applied graph-based feature representations for vulnerability detection. A summary of the reviewed studies is presented in Table 1, where we have highlighted the key difference of each work.

Only half of the studies made their data sets publicly available (refer to Table 5). Unfortunately, due to the expensive processes of data labeling, it is impossible to reproduce and evaluate the proposed methods without access to the data. Among the literature which provides data sets access, two studies [60], [61] have made their code published on Github, which enables easy replication of their works. However, the data sets made available by [60], [61] are relatively small in terms of the number of vulnerable samples (less than 500 source code functions).

Among the six studies reviewed earlier, five adopted ASTs as inputs to the DNNs for learning feature representations for vulnerability detection. According to [119], ASTs preserve a hierarchical structure of how statements and expressions are organized, containing relatively more

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

code semantics and syntactic in contrast to CFGs [61]. Therefore, ASTs can be a useful source for different types of neural models to learn feature representations relevant to potentially vulnerable patterns. However, to the best of our knowledge, there are no studies which compare ASTs with other forms of graph-based program representations such as CFGs, PDGs, or DDGs. Moreover, there are no studies providing evaluations on which program representation can enable a neural model to learn more effective code semantics yielding the best vulnerability detection performance.

Furthermore, the ASTs, CFGs, PDGs, and DDGs are graph-based program representations. Nevertheless, instead of using their original tree/graph form for processing, the aforementioned studies "flattened" them before feeding them to the deep network. The graph embedding techniques and the graph-based neural networks can be alternative and maybe more effective solutions for processing the abovementioned graph-based program representations for vulnerability detection. We will discuss this point in-depth in Section IV-C.

D. Sequence-Based Feature Representations

In this section, we review studies that adopt sequential code entities for the DNNs to learn the feature representations, which include system execution trace, function call sequences, sequences of statements forming the data flows, and so on.

1) Summary of Recent Works: An approach proposed in [35] utilized lightweight features extracted from both static and dynamic analysis for predicting memory corruption vulnerabilities in the operating system (OS) scale programs at the binary level. The assumption is that the call sequences/traces from both the static and dynamic analysis can reveal the usage patterns of the C library functions which exhibit the memory corruption vulnerabilities. The authors extracted static features from a set of call sequences associated with the standard C library functions, which requires the authors to disassemble the binaries. Obtaining the dynamic features required the execution of the program for a limited time period. During the execution, the authors monitored the events of the program and collected the call sequences. However, the obtained dynamic call sequences contained a large number of arguments of function calls which are low-level computational values. The authors needed to categorize them into subtypes to reduce the diversity of the argument types. Subsequently, the authors applied the N-gram language model and Word2vec [67], [68] for converting text sequences to meaningful vector representations which were then fed to three classifiers: LR, FCN, and random forest for training, validation, and testing. The authors reported that their approach could correctly detect 55% of the vulnerable programs and outperformed a simple fuzzer in terms of detection accuracy and efficiency.

An evaluation study for C program vulnerability detection was carried out in [116] to compare the performance of different types of DNNs on feature sets extracted from the dynamic function call sequences. Four types of network structures were chosen for evaluation: the CNN network proposed in [46], an LSTM network that contains only one LSTM layer, a CNN-LSTM network which has one convolution layer and one LSTM layer, respectively, and an FCN having two hidden layers. The authors collected 9872 32-bit Linux programs in binary format and applied the method introduced in [35] for obtaining C standard library function call sequences by allowing the programs to execute in a limited time period. Then, the authors followed the same method adopted in [35] to reduce the diversity of the argument types by using the subtypes of the arguments of the extracted function. Next, the authors used the tokenization tool provided by Keras [19], which is a deep learning framework, for converting the function call sequences to numeric ones by replacing each token with a unique integer. The padding and truncation were also applied to convert the sequences of various length to a fixed length of 25 tokens. Finally, the authors divided the 9872 program data set into training, validation, and test sets with a ratio of 8:1:1. To feed the input sequences to the networks, an embedding layer was added as the first layer of the three networks to convert each token in the input numeric call sequence to vectors of fixed length. The evaluation result showed that the LSTM network achieved the least false positive rate (19%) and the CNN-LSTM network outperformed the other networks in terms of the F-Score with 83.3%.

An approach proposed in [56] utilized a program representation called "code gadget" for detecting buffer errors and resource management error vulnerabilities. The code gadget is a number of consecutive or intermittent lines of code which are semantically related to each other, forming a sequence of statements depicting variable flows and data dependences. Specifically, according to this article, the semantic relationship of the code was defined as data dependence or control dependence. Therefore, the code gadget defined a number of lines of code which hinted the existence of vulnerability and could be captured by data flow or control flow dependences. To extract the library/API calls and the corresponding program slices that are related to the data flow or control flow, the authors used a commercial tool called Checkmarx.[8] Then, the extracted program slices were assembled to form a code gadget that represented a complete flow of data and/or API call sequence. Subsequently, the authors converted the program slices in a code gadget to a token's sequence and applied Word2vec [67], [68] to transfer them to vector representations of fixed length. A Bi-LSTM network was applied as a classifier for vulnerability detection based on the code gadget features.

[7] http://caca.zoy.org/wiki/zzuf
[8] https://www.checkmarx.com/

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Table 2 Reviewed Studies Which Applied Sequence-Based Feature Representations for Vulnerability Detection

Paper	Data	Labels Obtained from & Provided by	Features Representations	Detection Granularity
Grieco et al. [35]	Debian program binaries	Debian Bug Tracker & a simple fuzzer	Static & dynamic call sequences of C library functions	Program-level
Wu et al. [116]	32-bit Linux programs	A fuzzer named zzuf [7]	Dynamic call sequences of C library functions	Program-level
Li et al. [56]	C/C++ test cases from the SARD project & open source projects from Github	Automated & manual effort	The "code gadget" which depicts data flow & control flow	Intra-procedural & inter-procedural
Zou et al. [127]	C/C++ test cases from the SARD project & open source projects from Github	Automated & manual effort	The refined "code gadget" which depicts relations of data and control dependencies & "code attention" revealing specific libraries or API calls	Intra-procedural & inter-procedural

To evaluate the proposed approach, the authors collected 61 638 code gadgets (17 725 labeled as vulnerable and 43 913 nonvulnerable) from the code samples derived from the NVD and the SARD project [11]. Among the vulnerable samples, there were 10 440 buffer error vulnerabilities and 7285 resource management error vulnerabilities. Their empirical study showed that the proposed approach could achieve 95.0% F1-measure on the data set containing the resource management error vulnerabilities and the nonvulnerable ones. For the data set that contained only buffer error vulnerabilities, their result was 86.6% in terms of the F1-measure. For the data set containing both types of vulnerabilities, their approach achieved 85.4% of F1-measure. Notably, their study was the first to claim that they discovered some 0-day vulnerabilities among the reviewed studies.

A recent study [127] further extended [56] by incorporating the "code gadget" to contain not only the code sequences depicting the relations of data dependences but also the code ones revealing the relations of control dependences. Thus, the code sequences form contexts that capture the "global" semantics related to possible vulnerabilities. In order to detect specific vulnerability types, they also proposed the so-called "code attention" to focus on "localized" information within a statement such as "the arguments in a specific library or an API call." To facilitate the learning of the "global" semantics-based features and the "localized" semantics-based features, they proposed two deep Bi-LSTM networks with the same settings but on different scales. Eventually, they used a merge layer to fuse the global and localized features. Then, the fused feature representations were passed to another Bi-LSTM layer, followed by a softmax layer for classification.

For evaluation, the authors also extended the data sets published by Li *et al.* [56]. In terms of vulnerability types, they collected 116 different types of vulnerabilities from 323 programs written in C/C++ and 33 086 test cases from the SARD project. Their empirical studies proved that the proposed approach significantly outperformed the previous work in terms of detecting vulnerabilities of a variety of types, achieving 94.22% of F1-measure. Their method also discovered two real-world vulnerabilities in project Xen, which is proof of the effectiveness of the proposed approach.

2) Discussion: The previous section has reviewed four existing studies which applied sequential code entities for the DNNs to learn the feature representations for vulnerability detection. A summary of the reviewed studies is presented in Table 2, where we have highlighted the key difference of each work.

In this category, two studies [56], [127] have published their data sets on Github (see Table 5), Their published data can form a large data set containing more than 33 000 artificial test cases from the SARD project and more than 138 000 processed "code gadgets" from the real-world software projects. However, this data set failed to be an ideal baseline for performance comparison because it cannot be used by methods which are not based on "code gadgets." It would be more helpful if the authors could release the original data sets before the processing of "code gadgets."

Notably, the approach proposed in [56] is an important milestone. The program representation they defined, which is called "code gadget," accurately captures the semantic patterns and characteristics of buffer errors (CWE-119) and resource management errors (CWE-399) vulnerabilities, achieving the detection performance of 95% F1-measure on their data sets, which is the best performance among the reviewed studies. Inspired by their work, researchers further improved their idea by extending "code gadget" to include control dependences [127] or to accommodate more syntax- and semantic-based information [55] for detecting other types of vulnerabilities. However, their work is somewhat difficult to replicate and follow because the "code gadget" can only be extracted using a commercial tool.

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Compared with the graph-based inputs, the sequence-based inputs such as the call sequences and the data flows depicted by "code gadget" are completely "flattened," but they allow the neural models to capture the patterns and high-level features which are flow-based. Due to different studies being based on self-collected/-gathered data sets and focusing on different detection granularities, it is difficult to directly compare which form of input (i.e., the ASTs or the "code gadgets") can better facilitate the DNNs to learn more effective feature representations, resulting in better vulnerability detection performance. There is a need for a benchmarking data set offering a large number of label and code pairs for evaluating the proposed solutions.

E. Text-Based Feature Representations

In this section, we review studies which use a software code text for DNNs to learn the feature representations. The code text refers to the surface text of source code, the assembly instructions, and the source code processed by the code lexer.

1) Summary of Recent Works: A recent approach proposed in [78] converted Java source files to lists of textual tokens and applied the N-gram model for transferring the tokens to vectors. They assumed that the vulnerable patterns could be identified by mining the frequencies of the source code tokens. To handle the high dimensionality of the resulting feature vectors when using the N-gram model with $N > 2$, they performed feature selection by applying Wilcoxon rank-sum [115] for filtering irrelevant features to reduce the dimensionality of feature vectors. The authors fed the feature vectors to a deep FCN and the best performance achieved was 93% of detection accuracy on average with 97.6% of precision and 89.26% recall on their data sets.

Another CNN-based method introduced by [54] for detecting C program vulnerabilities at the assembly level aimed at capturing vulnerable code patterns directly from the assembly instructions. To transfer assembly instructions to vector representations, the authors developed Instruction2vec, which is based on Word2vec [67], [68] implementation. Instruction2vec generates a lookup table for assembly code, by mapping four types of assembly codes—opcode, register, pointer value, and library functions—to corresponding dense vectors of fixed length. Since each instruction has a fixed length of nine values (one opcode with two operands each of which contains four values), the authors can represent each instruction as a $9 \times m$-dimensional vector, (where m is the embedding dimension determined by the Word2vec model). With a piece of assembly code consisting of n instructions, Instruction2vec will convert the code to a $9 \times m \times n$ size matrix which can be directly fed to a CNN for processing. To evaluate the effectiveness of the proposed method, the authors set up four groups of experiments on the data set provided by the Juliet test suite [12]. They collected 3474 stack-based buffer overflow test cases (with training and test ratio of 4 to 1) and compiled them to 32-bit binaries. Then, the binaries were disassembled to generate assembly code files and were used to train the Word2vec and Instruction2vec, respectively. They also selected two types of CNNs: one was the CNN structure proposed in [46] which contained one convolution layer with nine different types of filters, and the other consisted of two convolution layers with 32 3×3 filters and 64 3×3 filters. The empirical study showed that the experiment using Instruction2ve and the CNN structure proposed in [46] achieved the best result which was 96.1% detection accuracy.

A similar approach that applied the CNN for function-level vulnerability detection was developed in [86]. The authors collected 12 million source code function samples from the Juliet Test Suite [12], Debian Linux distribution, and GitHub. The authors implemented a custom C/C++ lexer for parsing the source code to convert a function source code to a token sequence. To capture the meaning of key tokens and minimize the token vocabulary size, the lexer used only 156 tokens to represent source code. This means that, in addition to all C/C++ keywords, operators, and separators, the code that did not affect compilation was stripped out. Then, the authors applied trainable embedding for converting each token in the function sequence to a fixed k-dimensional representation, so that the embedding of each token can be fine-tuned via backpropagation during the network training phase. The authors applied two networks for automated feature extraction: 1) the CNN structure proposed in [46] having one convolution layer followed by a max-pooling layer and a two-layer GRU network followed by a max-pooling layer. During the network training phase, the whole network was trained. When the training processes completed, the authors fed the trained networks with the training and test data and obtained the output of the max-pooling layer of the two networks as the learned features which were called neural representations. Then, the neural representations were used as inputs to a random forest classifier for further training and test. The experiments indicated that compared with directly using the two networks as classifiers, using the networks as feature generators plus the random forest classifier yielded better detection performance.

A recent study proposed a maximal divergence sequential autoencoder (MDSAE) which builds on the VAE [47] for automated learning of representations from sequences of machine instructions for detecting vulnerabilities at the binary level [52]. The authors applied two learnable (nonfixed) Gaussian priors for each class (the vulnerable and the nonvulnerable class), and the codes from the latent space are fit into the data prior to distribution. Then, the divergence [e.g., Wasserstein (WS) distance or Kullback–Leibler (KL) divergence] between these two priors is maximized to separate the representations of vulnerable and nonvulnerable classes. In particular, the proposed MDSAE network can act as a feature extractor to generate

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

representations as features for another independent classifier (e.g., SVM or random forest) for classification tasks. Alternatively, the network can be incorporated with a shallow FCN and trained simultaneously as a classifier.

The data set for empirical study contains 32 281 binary functions which were obtained by compiling the data set provided in [56]. The resulting data set includes 17 977 binaries for Windows and 14 304 binaries for Linux. The Capstone binary disassembly framework[9] is used to convert binaries to assembly code which is more semantically meaningful. Then, the machine instructions are further processed to remove redundant prefixes and the opcode and the instruction information (e.g., memory location, registers) are preserved. The authors build two vocabularies to embed the opcode and the instruction information, respectively. To convert the opcode to a vector representation, the authors convert them to one-hot vectors and multiply the one-hot vector of the opcode with the corresponding embedding matrix. To convert the instruction information to vectors, the authors treat the instruction information as a sequence of hex-bytes based on which a frequency vector is constructed. The vector representation of the instruction information can be obtained by the multiplication of the frequency vector and the corresponding embedding matrix. Finally, the vector representation of instruction can be generated by concatenating the vectors of opcode and instruction information.

To evaluate the effectiveness of the proposed method, the authors set up five baseline systems: an RNN for learning representations and a linear classifier which is trained on the resulting representations for classifying vulnerable and nonvulnerable functions; an RNN with a linear classifier placed on the top of the last hidden unit; a paragraph-to-vector distributional similarity model [51]; a sequential VAE; and the method developed in [56]. The empirical study showed that the proposed method outperformed the baselines in all chosen performance measures [accuracy, recall, precision, F1-score, and the area under the receiver operating characteristic curve (AUC)].

The reviewed studies have shown that layered structures of a DBN and FCN are capable of learning high-level representations. They also proved that the CNN and variants of RNN (e.g., LSTM network) were able to capture the contextual patterns or structures from text corpora such as source codes or AST sequences. However, no comprehensive study has demonstrated that the conventional structures of DNN are able to understand the transition of value changes in a given code context until Choi *et al.* [18] filled the gap. They applied memory network for tracking changes made to the values of different variables across a number statements, and their empirical study illustrated that the memory network was able to judge whether a buffer overrun could be triggered based on the context where the values of different variables were changed by multiple assignment statements.

[9]www.capstone-engine.org

According to [18], monitoring value changes made to multiple variables not only requires the network to understand the structure of the code, but the network should also have the capability of remembering variables and their corresponding values as well as the changes being made to them. The conventional RNN structures (i.e., LSTM and GRU) which have "memory" through the hidden states or using the attention mechanism fail to accurately memorize all the contents expressed in a long sequence (e.g., a paragraph or a function body). Thus, networks equipped with extra-built memory blocks (e.g., the Neural Turing Machine [34] and the memory network [100], [112]) were developed to retain the extra-long range sequences for NLP tasks such as question–answering.

A memory network was implemented in [18] for performing an end-to-end detection of buffer overrun (also known as buffer overflow) vulnerabilities on synthetic data sets. The proposed network [18] directly took a program's source code as an input, and output whether the program contained a vulnerability or not. The program code is taken line by line. Each code token in a line was encoded by a V-dimensional one-hot vector, where V is the vocabulary size of unique tokens in all programs in the data set. The programs having less than N lines were padded by zeros, where N is the max number of lines a memory slot can store, for each line, zero padding was also used to fill the rest of each line. Then, the authors computed the vector representation of each line using its words. Specifically, a d-dimensional vector was used to represent each token and each line was encoded using an embedding matrix ($E - \text{val}$). To enable the model to differentiate the roles of identical tokens appearing in the different positions of a single line, the authors applied the position encoding provided in [99] to encode the position of tokens within each line. Next, the encoded lines were allocated to memory slots. In particular, the memory network has two types of memory slots: the memory value blocks for storing the content of the code lines and memory address blocks which stored the location information for accessing the code lines. Hence, each line was needed to be encoded using different embedding matrices: the E_{val} for the content of each line and the E_{addr} for the location information.

To check whether a line contained a buffer overrun vulnerability, this line was then treated as a query encoded into a vector representation using E_{addr} which was the address-related encoding. The attention mechanism was applied by computing the inner products between the query embedding and each slot of the memory address block. Subsequently, a softmax function was used to the resulting vector to produce an attention vector which suggested how related each line was to the query. The obtained attention vector was then multiplied with each slot of the memory value block to generate a response vector. The underlying idea of these operations is that if one of the lines within the search range would contain relevant information (e.g., variable names appeared in

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

the query), the resulting attention would be higher, thus contributing more significantly to a response vector. Finally, the response vector was either be applied to a weight matrix to generate an output or was added to the previous query embeddings which would be used for computing attention vectors and eventually the response vectors. The model iteratively generated response vectors through multiple hops and produced output ranging from 0 (unsafe) to 1 (safe).

The authors designed four levels of code samples, and a higher level indicating a more complex condition (e.g., level-1 samples refer to direct buffer access, which considers only one variable operation, whereas level-four samples are associated with multiple function invocation and type reallocation). The authors trained their network on 10 000 sample programs and tested on four test sets (each level per set) with 1000 samples each. Compared with CNN proposed in [46], and an LSTM network, the proposed memory network achieved the best results on all four levels of tests. Notably, the CNN under-performed on all levels of samples. The LSTM network achieved comparable performance on level 1 samples, but its performance decreased dramatically on higher level samples while the performance of memory network performance relatively stable.

However, the approach proposed in [18] has several weaknesses: the synthetic code used by them is syntactically invalid, so the code cannot be compiled, and the synthetic code does not have any control flow, which does not reflect the real world scenario. Hence, a recent study [89] made following improvements to address the weaknesses: first, they produced a more realistic code data set (named s-bAbI) containing syntactically accurate C programs with nontrivial control flow structures. The code samples contain six different types of labels and the labels are provided at line-level. Second, the authors added line number to each line of code to explicitly inform the memory network the order of line sequences when processing the code. Third, the authors applied Dropout [31], [98] with the value of 0.3 to each line of code tokens to prevent overfitting. For the network structure and the embedding method, the authors inherit the configurations and settings applied in [18]. To solve the class imbalance issue, the authors applied a random sampling technique to ensure the number of query lines with each label are equal in the generated batches.

For evaluation, the authors used the s-bAbI data set containing 76 549 samples (67% of them are sound) and the data set published in [18]. For comparison, the authors selected three open-source and one commercial static code analysis tools. The authors reimplemented the memory network proposed in [18] to test on their data set, and their network achieved 71.3% in terms of the F1-score. However, due to the data set provided in [18] being syntactically incorrect, the static analysis tools failed to perform. On the s-bAbI data set, the memory network achieved 91.7% and 90.6% of the F1-score on s-bAbI and the sound one, respectively, which outperformed the open-source tools, but the commercial tool achieved 93% and 91.9% in F1-score on the same data sets, respectively. For the sound data set, a code analysis tool called Frama-c accomplished 98.6% in terms of the F1-score. The authors concluded that the static analysis tools could achieve high performance due to the generated code being simple in structure (e.g., no complex function calls and only containing three control flow nodes). However, they also emphasized that the memory network could accomplish more than 90% of F1-score because the training and the test data set shared similar distribution. On the buffer overflow test cases of the Juliet Test Suite data set, the network failed to converge.

2) Discussion: The previous section has reviewed six existing studies which applied software code text for DNNs to learn the feature representations for vulnerability detection. A summary of the reviewed studies is presented in Table 3, where we have highlighted the key difference of each work.

Among the above-reviewed literature, three out of six studies [18], [52], [89] have made their data publicly accessible (see Table 5). The data sets published in [52] are based on the ones proposed in [56] but in an assembly/binary format. The data sets provided in [18] and [89] are artificially constructed and not compilable, which fail to realistically reflect the real vulnerable patterns in the production software.

Different from the studies of other categories, the works in this category do not apply any code analysis methods. They directly use code text as inputs for learning feature representations and expect the neural models to be expressive and capable of learning potentially vulnerable code semantics hidden in the surface text of the code. The study [78] applied the FCN which is a generic network structure for learning source code frequency-based features. Later studies applied the CNN for context learning. Due to the vulnerable code semantics often containing multiple lines of code forming a context, the CNN structure can be a suitable option for detecting vulnerabilities that are related to vulnerable code context. The latest studies [18] and [89] proposed the memory network which is a complex structure incorporating additional memory slots for storing long-range code context, allowing the network to directly accepts source code as inputs. With the network structure being more complex and expressive, the efforts required for code processing decrease, which can be a new trend. Section IV-B provides an in-depth discussion in this regard.

Comparing and evaluating the proposed studies in this category is challenging. For example, among the studies which published their data sets, Nguyen *et al.* [52] focus on assembly-level detection. Two studies [18] and [89] require the labels to be provided at the statement-level. Performing comparative studies of these approaches demands a baseline data set that meets the requirement would incur tremendous laboring.

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Table 3 Reviewed Studies Which Applied Text-Based Feature Representations for Vulnerability Detection

Paper	Data	Labels Obtained from & Provided by	Features Representations	Detection Granularity
Peng et al. [78]	Source code of Android apps	Provided by F-Driod	Source code frequency features based on N-gram	File/class-level
Lee et al. [54]	The stack-based buffer overflow test cases from the Juliet Test Suit	The Juliet Test Suit	Assembly code-based features	Function-level
Russell et al. [86]	The Juliet Test Suit, Debian programs & open source projects from Github	Combining the Clang, Cppcheck & Flawfinder with manual evalution	Lexed source code with reduced vocabulary size	Function-level
Le et al. [52]	The binary functions compiled from the data set provided by Li et al. [56]	Provided by Li et al. [56]	Surface text of the assembly code	Function-level
Choi et al. [18]	The synthetic data set (CJOC-bAbI)	Automatically generated	Generated from source code	Statement-level
Sestili et al. [89]	CJOC-bAbI & s-bAbI & the buffer overflow test cases from Juliet Test Suite	Automated & the Juliet Test Suite	Generated from source code	Statement-level

F. Mixed Feature Representations

In this section, we review studies based on the input types which combine any of the aforementioned categories.

1) Summary of Recent Works: An approach for detecting vulnerabilities in Android binary executables was proposed in [27]. Specifically, the authors decompiled the Android APK files and obtained the smali files which contain bags of dalvik instructions. Two types of features were extracted from the smali files: 1) the token features represented by the frequency of dalvik instructions which show the token properties and 2) the semantic features which were generated by traversing the ASTs of the smali files. To extract the token features, the authors categorized the dalvik instructions of smali files into eight categories and built a mapping table. The token features were obtained by recording the frequency of the instructions in the mapping table. The semantic features were extracted from the ASTs which were parsed from the smali files using antlr [77]. The authors applied depth-first search (DFS) traversal to convert ASTs to sequences. Then, the resulting AST sequences were converted to integer sequences based on the previously built dalvik instructions mapping table. Lastly, the feature vectors were constructed by combining the token feature and the semantic feature vectors and fed to a deep FCN for training and test. The authors also performed a comparison between the deep FCN and other conventional ML algorithms including SVM, NB, C4.5, and LR. Their empirical results showed that the deep FCN outperformed the other algorithms and achieved an AUC of 85.98%.

Another approach which was proposed in [38] was based on two sets of features for vulnerability detection. The first set was the source-based features extracted directly from the source code and the second one was the build-based features derived from the CFGs generated by Clang and low level virtual machine (LLVM) [50]. Specifically, for the source-based features, the authors implemented a custom C/C++ lexer to convert source code functions to token sequences followed by the procedure of [36]. Particularly, variable names were mapped to the same generic identifier, but each unique variable name within a single function was assigned a separate index to keep track of variable reappearance. Then, Mikolov *et al.* [67], [68] used the bag-of-word model and the Word2vec model to convert the resulting token sequences into two types of vector representations. For the build-based features, the authors compiled the programs and extracted features from the function-level CFGs and basic blocks which are at the instruction-level. The feature sets consisted of the operations happening in each basic block and the definition and use of variables. For instance, the authors constructed a use-def matrix to record the definition and use of variables. If a variable is defined at instruction i and used in instruction j, then both the (j, i) entry of the use-def matrix are set to 1. The resulting features sets contained the adjacency matrices and op-vec/use-def vectors. Then, the authors defined a hand-crafted fixed-size vector to accommodate the adjacency matrix of the CFG and the op-vec/use-def vector by performing average operations.

For the performance evaluation, the authors compared the CNN proposed in [46] with conventional models which are random forest and extremely randomized trees [32]. The empirical study indicated that the models based on source-based features achieved better performance. Compared with using the CNN as the classification model,

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Table 4 Reviewed Studies Which Applied Mixed Feature Representations for Vulnerability Detection

Paper	Data	Labels	Features Representations	Detection Granularity
Dong et al. [27]	Android apk files.	Provided by Checkmarx CxEnterprise (A commercial static code analysis tool)	AST-based semantic features & tokens frequency features	Statement-level
Harer et al. [38]	Debian programs & open source projects from Github	The Clang static analyzer (SA)	Build-based: CFGs & basic blocks; Source-based: lexed source code with reduced vocabulary size	Function-level
Lin et al. [59]	Six open source projects from Github written in C language & C/C++ test cases from the SARD project	Automated & manual effort	ASTs at function-level in serialized format & high-level features generated from source code	Function-level

the combining features learned by the CNN (without the dense layers) and classified by the tree-based models yielded the best performance.

A framework proposed in [59] utilized two Bi-LSTM networks for obtaining feature representations from two different types of data sources to compensate for the shortage of labeled data. This framework allowed each network to be trained independently and to be used as a feature extractor for learning the latent representations of the vulnerable patterns from a combination of two vulnerability-relevant data sources which are the SARD project [11] containing synthetic vulnerable test cases and the real-world vulnerability data source. To bridge the difference between the real-world samples and the synthetic ones from the SARD project, the authors applied two different types of feature representations: the ASTs extracted from the real-world samples and the source code of the synthetic samples. Then, one network was fed with the ASTs and the other took the source code as input during the training phase. They assumed that the two trained networks were able to capture more "vulnerability knowledge" from both vulnerability-relevant data sources compared with using one isolated network trained with a single data source. After the training phase, they used the available labeled data from the target software project and fed them to both trained Bi-LSTM networks to obtain two groups of high-level representations, respectively. Subsequently, the obtained representations were concatenated to form an aggregated feature set which could be used to train a conventional ML classifier such as a random forest. Their empirical studies proved that the framework utilizing two vulnerability-relevant data sources was more effective than using only one data source. The framework outperformed their previous work [61] on projects FFmpeg, LibTIFF, and LibPNG.

2) Discussion: The previous section has reviewed three existing studies which applied a combination of different inputs for learning feature representations for vulnerability detection. A summary of the reviewed studies is presented in Table 4, where we have highlighted the key difference of each work.

Among the studies in this category, only one work [59] makes the data sets and code publicly available (see Table 5). Their approach is also the first one that utilizes two networks for learning two types of feature representations from two groups of inputs—the ASTs and the source code. The other two studies in this category suggest an automated solution for data labeling, which is to use static code analysis tools to provide labels. However, static tools tend to generate many false positives and manual effort is needed for a further check.

Notably, all three studies reported that combining different types of feature representations resulted in improved detection performance compared with using a single type. This may suggest a new direction that using neural networks to learn different types of feature representations may obtain more vulnerability-relevant knowledge, which contributes to more effective learning of vulnerable code semantics, leading to better detection results. The representation learning capability of neural networks can bridge the differences among different types of data sets (e.g., either synthetic or real-world) by learning high-level and abstract representations containing the generic vulnerability-relevant information. In other words, the data sets which are fed to the neural networks may be different in terms of data formats and methods of processing, but the feature representations learned by the network are generic. This may provide insights into comparing and evaluating the proposed solutions which are based on different data sets.

IV. CHALLENGES AND FUTURE DIRECTIONS

Despite the reviewed studies have proposed various types of feature sets and applied different network structures to reduce the semantic gap, there is still a long way to go. In this section, we will discuss the challenges and future works, and share some insights.

A. Big Ground-Truth Data Sets

The data sets are the major barrier that hinders the development of this field. At the current stage, the proposed neural network-based vulnerability detection techniques were all evaluated on self-constructed data sets. Due to different approaches targeting various detection

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Table 5 Publicly Available Software Vulnerability Data Sets Collected by the Reviewed Studies at the Time of Writing

Paper	Data Type & Ganularity	Accessible Link to The Data Set
Lin et al. [60]	Real-world C code samples labeled at function-level	https://github.com/DanielLin1986/function_representation_learning
Lin et al. [61]	Real-world C code samples labeled at function-level	https://github.com/DanielLin1986/TransferRepresentationLearning
Lin et al. [59]	Real-world and synthetic C code samples labeled at function-level	https://github.com/DanielLin1986/RepresentationsLearningFromMulti_domain
Lin et al. [58]	Real-world C code samples labeled at file-level and function-level	https://github.com/DanielLin1986/Function-level-Vulnerability-Detection
Li et al. [56]	Real-world and synthetic C code samples labeled at "code gadget" level	https://github.com/CGCL-codes/VulDeePecker
Li et al. [55]	Real-world and synthetic C code samples labeled at "SeVCs" level	https://github.com/SySeVR/SySeVR
Zou et al. [127]	Real-world and synthetic C code samples labeled at "code gadgets" level, composing of multiple program statements, having direct or indirect data-/ control-dependence relationships with the library/API function calls.	https://github.com/muVulDeePecker/muVulDeePecker
Choi et al. [18]	Synthetic C code samples labeled at line-of-code level	https://github.com/mjc92/buffer_overrun_memory_networks
Sestili et al. [89]	Synthetic C code samples labeled at line-of-code level	https://github.com/cmu-sei/sa-bAbI
Dong et al. [27]	Open source Java code samples labeled at line-of-code level	https://github.com/breezedong/DNN-based-software-defect-prediction
Le et al. [52]	Compiled binary code samples based on Li et al. [56] , labeled at function-level	https://github.com/dascimal-org/MDSeqVAE

granularity, the ground truth data sets constructed by existing studies are labeled at different levels (e.g., labeled at the line-of-code level or at function level). In addition, the data sets gathered by existing studies are either synthetic or collected from real-world programs. Hence, there is an urgent need for a standard benchmarking data set to act as a unified metric for evaluating and comparing the effectiveness of proposed approaches. To fill this gap, the work [58] is the first to propose a benchmarking data set containing vulnerable source code collected from nine open-source software projects written in C programming language. Their data set offers labels at two levels of granularity i.e., the function-level and the file-level. However, their proposed data set is still at the preliminary stage since it only consists of around 1400 vulnerable functions and 1300 vulnerable files. Ideally, a standard benchmarking data set should be carefully tailored for approaches focusing on multiple levels of detection granularity and should contain a large number of both synthetic and real-world code samples for evaluation and comparison qualitatively and quantitatively.

There are a number of data repositories providing synthetic vulnerability samples. The Software Assurance Metrics And Tool Evaluation (SAMATE) project [10] provided by the National Institute of Standards and Technology (NIST) is established for the evaluation of automated tools to assure software quality. According to [89], the SAMATE project consists of several data sets, including The Juliet Test Suite [12], the SARD [11], The Securely Taking On New Executable Software of Uncertain Provenance (STONESOUP) project by the Intelligence Advanced Research Projects Activity (IARPA) [2], and the Static Analysis Tool Exposition (SATE) data sets for the evaluation of static analysis tools [75]. The Juliet Test Suite and the SARD project were adopted by several reviewed studies such as [55], [56], and [89], and contain mainly artificially constructed code samples which are compilable. However, Choi *et al.* [18] questioned that code patterns of the test cases in Juliet Test Suite was not diverse enough. Many code samples follow a similar coding format and a large portion of the code remains identical among many samples.

More importantly, the aforementioned data repositories lack the labeled code samples from real-world software projects. The performance evaluation of the real-world test cases can realistically reflect how the proposed vulnerability detection approaches perform in the real-world environment. To address this issue, many reviewed studies constructed real-world data sets for evaluation, but only a few studies made their self-constructed data sets publicly available. Table 5 lists the publicly accessible links to the data sets collected by the reviewed studies at the time of writing. However, the data sets available for use are generally small in size, providing insufficiently labeled vulnerability data [58]–[61] or offering synthetic samples which cannot be compiled [18], [89]. Studies provide data sets in a relatively large size, which may require the usage of a commercial tool [55], [56], [127]. Thus, we appeal to researchers in this field to keep contributing their data and code to the public.

B. Code Analysis and Neural Learning

From the reviewed literature, a trend can be seen that the network models applied for vulnerability detection is becoming increasingly complex and more expressive for better learning code semantics indicative of vulnerable code snippets. From the early study which applied the MLP [92] to the recent studies using the CNN [38], [86] or LSTM [56], [61] until the very recent studies adopting memory network [18], [89], the evolving network structure has shown the research effort that has been put into exploring the potential of neural networks for reasoning about the code semantics and rich patterns for facilitating vulnerability discovery. Researchers from the ML and NLP communities have been motivated to adopt the state-of-the-art tools/approaches for code analysis for vulnerability detection [7].

Another trend can also be revealed that with the network becoming more complex, the effort required

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

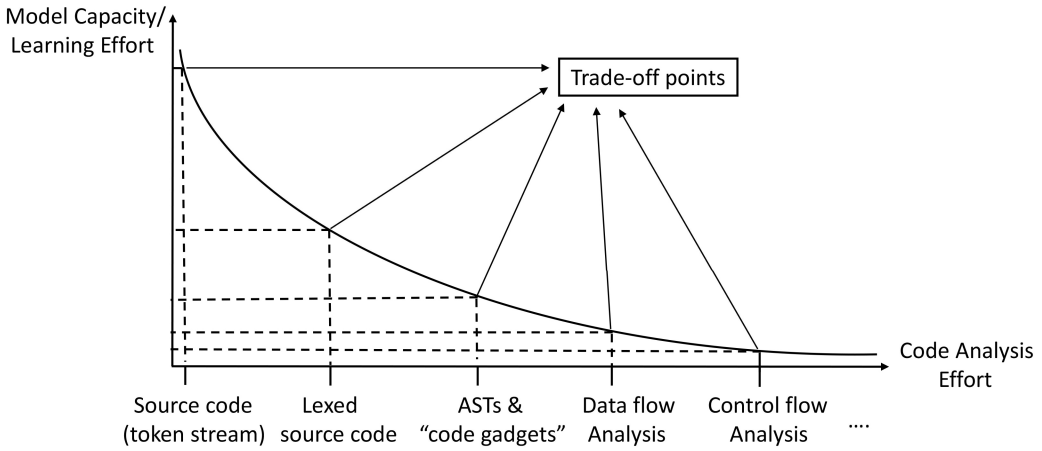


Fig. 3. ***Tradeoff between the neural model learning/design effort and the code analysis effort. There are tradeoff points that one needs to consider between the effort of designing a capable/expressive model and the effort of code analysis for generating effective features [8].***

for code analysis efforts decreased. The study applied MLP [92] which uses the handcrafted features from both CFGs and DDGs for learning the input validation and sanitization code patterns. Extracting CFGs and DDGs require significant code analysis effort as well as expertise for deriving features. Later studies, which used the CNN [38], [86], can directly learn features from lexed source code that required less code analysis effort while the very recent studies applying the memory network takes source code as input without the need for any code analysis. Nonetheless, expertise and knowledge from software engineering and cybersecurity fields are still in need of defining representative and effective features to guide the design of the network architecture for better discovering vulnerable semantics and patterns which are latent and complex.

Fig. 3, which is based on [8], illustrates the trends that with more expressive models applied for learning vulnerable semantics, less effort is needed for code analysis. In addition, Fig. 3 also shows the tradeoff points made by some of the reviewed studies to balance the model learning/design effort and code analysis effort in real-world scenarios. In practice, ML-based tasks demand the consideration of a trade-off between the effort of designing/training a complex but capable model and the effort of defining high-quality and representative features. Loosely defined features require an expressive model to learn from scratch, whereas developing/training an expressive model fit for the task of interest is also challenging [114].

C. Neural Model With Semantic Reservation

A key point in advancing the field of applying neural networks for vulnerability detection is to fill the semantic gap by enabling a neural model to better reason with the semantics of programming languages [7], [33].

At the algorithm level, to develop more expressive and capable models for learning code semantics and syntactic structures can contribute to resolving the semantic gap. Despite the trend that the network applied by the reviewed studies is becoming increasingly complex, the state-of-the-art implementation, such as memory network, was proved to be well-performed (achieved 91.7% of F1-score) only on synthetic data set containing simplified code samples instead of on the real-world code samples. However, this does not stop researchers from further exploring the potential of complex neural models. The very recent study which applied generative adversarial network (GAN) [37] for repairing vulnerabilities on C/C++ synthetic code examples has demonstrated the capability of the neural model in understanding code logic and semantics.

In the field of NLP, the recent advances in sequence modeling and natural language understanding have been encouraging. For instance, the Transformer neural network architecture [107], which is based on a self-attention mechanism, has outperformed state-of-the-art approaches in natural language translation tasks and demonstrated strong capability in natural language understanding. The fruitful research outcomes from the field of NLP has been increasingly adopted in software engineering research [28], e.g., the adoption of Word2vec and Bi-LSTM for code semantics learning [56], [61]. It would be exciting to apply models like Transformer for better understanding programming language semantics for code analysis tasks such as the vulnerability discovery.

Another gap to be filled is to apply tree-/graph-based neural networks for vulnerability detection. Many program representations are in the form of trees or graphs. For example, the AST depicts code components in a hierarchical structure of a tree. The CFG can be a directed graph revealing a program's control flows and the PDG uses graph notation to explicitly represent data dependences and control dependences. Existing studies utilize these program representations but in their "flattened" form and process them in a "sequential" manner using FCN, CNN, or RNNs, which may inevitably lose the hierarchical and/or structural information containing possible vulnerable code semantics. Therefore, to better process and preserve the hierarchical and/or structural information hidden in the tree-/graph-based program representations,

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

tree-/graph-based neural networks can be ideal options. The tree-structured LSTM (TS-LSTM) network proposed in [105] and the recursive neural network in [97] have demonstrated their effectiveness in processing syntactic properties of natural languages which are organized in a tree structure. These networks can be applicable and may be effective in handling ASTs for learning the tree-structured code semantics.

There is also an abundance of graph-based neural networks (also known as the graph neural networks (GNNs) [118]) which can operate directly on graph-based program representations (e.g., the CGFs and PDGs) for learning high-lever feature representations indicative of potentially vulnerable code. For example, the graph convolutional networks (GCNs) [48] and the graph autoencoders (GAEs) [14] can facilitate the learning of program representations presented in the graph structures and the interdependency between nodes and edges which depict the relationships of code entities.

Another possible solution to assist the model for learning code semantics can be providing other sources of information that are relevant to code to support the model learning. The open-source repositories, such as GitHub, offer abundant textual footprint of software projects, which can be used as relevant sources of information to facilitate the model to learn code semantics, for instance, the commit messages to the code repositories [79], the comments describing/explaining to a piece of code, and the code review comments.

D. Code Representation Learning

From the perspective of data processing, the semantic gap can be narrowed by designing better feature engineering techniques to incorporate rich semantic and syntactic code features to boost model learning capacity. On one hand, defining high-quality features that are representative of vulnerable code characteristics facilitates the model learning, which can significantly contribute to the success of ML-based detection systems [33]. On the other hand, high-quality features rely less on complex and expressive models which may require a large amount of data for training.

In the field of vulnerability detection, due to vulnerable code patterns being diverse and statements, which form a vulnerable code context being either within function boundary (intraprocedural) or spanning multiple functions (interprocedural), defining feature sets that universally characterize all types of vulnerabilities is nearly infeasible and impossible. Hence, defining feature sets reflect the characteristics of certain types of vulnerabilities can be an eclectic choice, but developing detection systems targeting a particular type of vulnerabilities has achieved promising results [56], [121].

As we desire better features to represent/describe the characteristics of vulnerabilities, we also call for more effective embedding techniques. When transferring code or textual features to vector representations, we aim to preserve more semantic meanings of textual tokens so that the down-stream models can learn richer and more expressive semantics. A large body of the reviewed work relied on frequency-based (e.g., the n-gram model) and prediction-based (e.g., the Word2vec model [67], [68]) word embedding solutions. The frequency-based embedding techniques produce word embeddings based on word frequencies that appeared in a given context, which captures limited word semantic meanings. The prediction-based models, particularly the Word2vec, can learn word semantics from a relatively small context window but is unable to capture the meaning that is interdependent between the target words and their context as a whole [80]. Thus, a more effective embedding technique providing a learning system with richer semantic information to learn from is desirable.

E. Model Interpretability for Human Understanding

ML models, particularly the neural network models, are black boxes [85], meaning that the reasons behind how a model makes predictions/classifications are unknown to practitioners. A large body of the reviewed studies made no effort to interpret the model's behavior. In the field of vulnerability detection, the inability to understand how a model predicts a piece of code to be vulnerable/nonvulnerable may cause the utility of the model to be questionable. One may raise the following questions: is the model trustworthy? Or is this (individual) prediction/classification reliable? Unable to understand a model's behavior can be one of the obstacles hindering the application of neural network-based models for vulnerability detection in practice.

Unlike linear models, the neural networks introduce nonlinearity and their layered structures bring difficulties for model interpretability. In particular, when applying neural models for automated learning of high-level representations as features, the obtained features are abstract and humanly unreadable. To address this problem, LIME[10] [85] was proposed to offer interpretability for any model by learning a simple and interpretable model (e.g., a linear or a decision tree model) locally around the prediction samples. However, LIME is not applicable when neural network models are used as feature extractors whose extracted features are then used to train a classifier.

The attention mechanism can serve as a means for explaining the model. With the attention mechanism, a CNN/RNN can selectively attend to or focus on one part of an input (e.g., an image or a text sequence) at a time [53]. By plotting the attended parts (e.g., an attention weight matrix), a degree of interpretability for the model can be provided. For instance, an attention network architecture was applied for learning function-level representations from the paths of a function's AST [8]. To combine a bag of path-context vectors to form a representation,

[10]https://github.com/marcotcr/lime

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

the attention network can learn how much attention should be paid to each path-context. By obtaining the attention vectors, the authors can have a quantitative measurement of how much attention the network gave for each path-context within an AST, suggesting the various degrees of importance each path-context being given during the representation generation process. Most recently, the attention network has been applied for defect prediction [30], indicating a promising research direction for offering explainable deep learning-based code analysis and vulnerability/defect prediction.

V. CONCLUSION

The application of neural networks for vulnerability detection is an immature research field with many problems unsolved. As the ML field is fast-growing, major progress in ML and deep learning will add value to the toolkit for vulnerability detection. We reviewed existing studies which applied neural networks for vulnerability detection, showing an evolving trend of network structures being adopted for better learning the characteristics of vulnerable code snippets, aiming to bridge a gap between how a practitioner understands the vulnerability and the obtained semantics that a neural network model can learn from. The representation learning capability of the neural models and their customizable structure offer exciting potential for automated learning of complex vulnerable code patterns, which will motivate and attract more researchers to contribute to this promising field. ■

REFERENCES

[1] *The Heartbleed Vulnerability*. Accessed: Jul. 18, 2017. [Online]. Available: http://heartbleed.com/

[2] (Sep. 2009). *Securely Taking on New Executable Software of Uncertain Provenance (Stonesoup)*. Accessed: May 8, 2019. [Online]. Available: https://www.iarpa.gov/index.php/research-programs/stonesoup/baa

[3] (Sep. 2014). *Shellshock: All You Need to Know About the Bash Bug Vulnerability*. Accessed: Jul. 18, 2017. [Online]. Available: https://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability

[4] (2016). *Rough Audit Tool for Security*. Accessed: May 26, 2018. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[5] (Sep. 2017). *Equifax Had Patch 2 Months Before Hack and Didn't Install It, Security Group Says*. Accessed: May 3, 2019. [Online]. Available: https://www.usatoday.com/story/money/2017/09/14/equifax-identity-theft-hackers-apache-struts/665100001/

[6] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 25–34.

[7] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, p. 81, 2018.

[8] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proc. ACM Program. Lang. POPL*, vol. 3, 2019, p. 40.

[9] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*. [Online]. Available: http://arxiv.org/abs/1409.0473

[10] P. E. Black, "Samate's contribution to information assurance," *NIST Special Publication*, vol. 500, no. 264, p. 2, 2006.

[11] P. E. Black, "A software assurance reference dataset: Thousands of programs with known bugs," *J. Res. Nat. Inst. Standards Technol.*, vol. 123, pp. 1–3, Apr. 2018.

[12] P. E. Black and P. E. Black, *Juliet 1.3 Test Suite: Changes From 1.2*, U.S. Department of Commerce, National Institute of Standards and Technology, 2018.

[13] C. Cadar *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, vol. 8, pp. 209–224, 2008.

[14] S. Cao, W. Lu, and Q. Xu, "Deep neural networks for learning graph representations," in *Proc. 13th AAAI Conf. Artif. Intell.*, 2016, pp. 1145–1152.

[15] X. Chen *et al.*, "Android HIV: A study of repackaging malware for evading machine-learning detection," *IEEE Trans. Inf. Forensics Security*, vol. 15, pp. 987–1001, Jul. 2020.

[16] B. Chess and M. Gerschefske, *Rough Auditing Tool for Security*. Accessed: Mar. 18, 2019. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[17] K. Cho *et al.*, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*. [Online]. Available: http://arxiv.org/abs/1406.1078

[18] M.-J. Choi, S. Jeong, H. Oh, and J. Choo, "End-to-end prediction of buffer overruns from raw source code via neural memory networks," 2017, *arXiv:1703.02458*. [Online]. Available: http://arxiv.org/abs/1703.02458

[19] F. Chollet *et al.* (2015). *Keras*. [Online]. Available: https://github.com/fchollet/keras

[20] I. Chowdhury and M. Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, Mar. 2011.

[21] R. Coulter, Q.-L. Han, L. Pan, J. Zhang, and Y. Xiang, "Data-driven cyber security in perspective–intelligent traffic analysis," *IEEE Trans. Cybern.*, early access, Oct. 15, 2019, doi: 10.1109/TCYB.2019.2940940.

[22] B. C. Csáji, "Approximation with artificial neural networks," Dept. Sci., Eötvös Loránd Univ., Budapest, Hungary, Tech. Rep. 48, 2001, vol. 24, p. 7.

[23] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Math. Control, Signals Syst.*, vol. 2, no. 4, pp. 303–314, 1989.

[24] H. K. Dam, T. Tran, T. Pham, S. W. Ng, J. Grundy, and A. Ghose, "Automatic feature learning for vulnerability prediction," 2017, *arXiv:1708.02368*. [Online]. Available: http://arxiv.org/abs/1708.02368

[25] L. Deng *et al.*, "Recent advances in deep learning for speech research at Microsoft," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, May 2013, pp. 8604–8608.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*. [Online]. Available: http://arxiv.org/abs/1810.04805

[27] F. Dong, J. Wang, Q. Li, G. Xu, and S. Zhang, "Defect prediction in Android binary executables using deep neural network," *Wireless Pers. Commun.*, vol. 102, no. 3, pp. 2261–2285, Oct. 2018.

[28] V. Efstathiou, C. Chatzilenas, and D. Spinellis, "Word embeddings for the software engineering domain," in *Proc. 15th Int. Conf. Mining Softw. Repositories (MSR)*, 2018, pp. 38–41.

[29] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: A general approach to inferring errors in systems code," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 57–72, 2001.

[30] G. Fan, X. Diao, H. Yu, K. Yang, and L. Chen, "Software defect prediction via attention-based recurrent neural network," *Sci. Program.*, vol. 2019, pp. 1–14, Apr. 2019.

[31] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian approximation: Representing model uncertainty in deep learning," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1050–1059.

[32] P. Geurts, D. Ernst, and L. Wehenkel, "Extremely randomized trees," *Mach. Learn.*, vol. 63, no. 1, pp. 3–42, Apr. 2006.

[33] S. M. Ghaffarian and H. R. Shahriari, "Software vulnerability analysis and discovery using machine-learning and data-mining techniques: A survey," *ACM Comput. Surv.*, vol. 50, no. 4, pp. 1–36, Nov. 2017.

[34] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," 2014, *arXiv:1410.5401*. [Online]. Available: http://arxiv.org/abs/1410.5401

[35] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, "Toward large-scale vulnerability discovery using machine learning," in *Proc. 6th ACM Conf. Data Appl. Secur. Privacy (CODASPY)*, 2016, pp. 85–96.

[36] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common C language errors by deep learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1345–1351.

[37] J. Harer *et al.*, "Learning to repair software vulnerabilities with generative adversarial networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 7933–7943.

[38] J. A. Harer *et al.*, "Automated software vulnerability detection with machine learning," 2018, *arXiv:1803.04497*. [Online]. Available: http://arxiv.org/abs/1803.04497

[39] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 122–131.

[40] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng. (ICSE)*, Jun. 2012, pp. 837–847.

[41] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

[42] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 48–62.

[43] J. Jiang, S. Wen, S. Yu, Y. Xiang, and W. Zhou, "Identifying propagation sources in networks: State-of-the-art and comparative studies," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 1,

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

pp. 465–481, 1st Quart., 2017.
[44] G. Jie, K. Xiao-Hui, and L. Qiang, "Survey on software vulnerability analysis method based on machine learning," in *Proc. IEEE 1st Int. Conf. Data Sci. Cyberspace (DSC)*, Jun. 2016, pp. 642–647.
[45] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 595–614.
[46] Y. Kim, "Convolutional neural networks for sentence classification," 2014, *arXiv:1408.5882*. [Online]. Available: http://arxiv.org/abs/1408.5882
[47] D. P. Kingma and M. Welling, "Auto-encoding variational Bayes," 2013, *arXiv:1312.6114*. [Online]. Available: http://arxiv.org/abs/1312.6114
[48] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," 2016, *arXiv:1609.02907*. [Online]. Available: http://arxiv.org/abs/1609.02907
[49] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.
[50] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Gener. Optim., Feedback-Directed Runtime Optim.* Washington, DC, USA: IEEE Computer Society, Mar. 2004, pp. 75–86.
[51] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.
[52] T. Le *et al.*, "Maximal divergence sequential autoencoder for binary software vulnerability detection," in *Proc. 7th Int. Conf. Learn. Represent. (ICLR)*, 2018.
[53] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.
[54] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *Proc. KSII 9th Int. Conf. Internet Symp. (ICONI)*, 2017.
[55] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," 2018, *arXiv:1807.06756*. [Online]. Available: http://arxiv.org/abs/1807.06756
[56] Z. Li *et al.*, "Vuldeepecker: A deep learning-based system for vulnerability detection," in *Proc. NDSS*, 2018, pp. 1–15.
[57] Z. Li and Y. Zhou, "PR-miner: Automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 306–315, Sep. 2005.
[58] G. Lin, W. Xiao, J. Zhang, and Y. Xiang, "Deep learning-based vulnerable function detection: A benchmark," in *Proc. Int. Conf. Inf. Commun. Secur.* Cham, Switzerland: Springer, 2019, pp. 219–232.
[59] G. Lin *et al.*, "Software vulnerability discovery via learning multi-domain knowledge bases," *IEEE Trans. Dependable Secure Comput.*, early access, Nov. 19, 2019, doi: 10.1109/TDSC.2019.2954088.
[60] G. Lin, J. Zhang, W. Luo, L. Pan, and Y. Xiang, "POSTER: Vulnerability discovery with function representation learning from unlabeled projects," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Oct. 2017, pp. 2539–2541.
[61] G. Lin *et al.*, "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Trans. Ind. Informat.*, vol. 14, no. 7, pp. 3289–3297, Jul. 2018.
[62] B. Liu, L. Shi, Z. Cai, and M. Li, "Software vulnerability discovery techniques: A survey," in *Proc. 4th Int. Conf. Multimedia Inf. Netw. Secur.*, Nov. 2012, pp. 152–156.
[63] L. Liu, O. De Vel, Q.-L. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: A survey," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 2, pp. 1397–1417, 2nd Quart., 2018.
[64] R. Malhotra, "A systematic review of machine learning techniques for software fault prediction," *Appl. Soft Comput.*, vol. 27, pp. 504–518, Feb. 2015.
[65] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. SE-2, no. 4, pp. 308–320, Dec. 1976.
[66] A. Meneely and L. Williams, "Secure open source collaboration: An empirical study of Linus' law," in *Proc. 16th ACM Conf. Comput. Commun. Secur. (CCS)*, 2009, pp. 453–462.
[67] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*. [Online]. Available: http://arxiv.org/abs/1301.3781
[68] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.
[69] P. Morrison, K. Herzig, B. Murphy, and L. Williams, "Challenges with applying vulnerability prediction models," in *Proc. Symp. Bootcamp Sci. Secur.*, 2015, p. 4.
[70] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, 2005, pp. 284–292.
[71] J. Nam, S. J. Pan, and S. Kim, "Transfer defect learning," in *Proc. Int. Conf. Softw. Eng. (ICSE)*, May 2013, pp. 382–391.
[72] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, Mar. 2001.
[73] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proc. 14th Conf. CCS*, 2007, pp. 529–540.
[74] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proc. NDSS*, 2005, pp. 3–4.
[75] V. Okun, A. Delaitre, and P. E. Black, "Report on the static analysis tool exposition (sate) IV," *NIST Special Publication*, vol. 500, p. 297, Jan. 2013.
[76] Y. Pang, X. Xue, and A. S. Namin, "Predicting vulnerable software components through N-Gram analysis and statistical feature selection," in *Proc. IEEE 14th Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2015, pp. 543–548.
[77] T. Parr, *The Definitive ANTLR 4 Reference*. Raleigh, NC, USA: Pragmatic Bookshelf, 2013.
[78] H. Peng, L. Mou, G. Li, Y. Liu, L. Zhang, and Z. Jin, "Building program vector representations for deep learning," in *Proc. Int. Conf. Knowl. Sci., Eng. Manage.* Cham, Switzerland: Springer, 2015, pp. 547–553.
[79] H. Perl *et al.*, "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proc. 22nd SIGSAC Conf. CCS*, 2015, pp. 426–437.
[80] M. E. Peters *et al.*, "Deep contextualized word representations," in *Proc. NAACL*, 2018, pp. 1–15.
[81] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, "Leveraging semantic signatures for bug search in binary programs," in *Proc. 30th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2014, pp. 406–415.
[82] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: An emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation," *ACM SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, pp. 15–27, Oct. 2006.
[83] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proc. USENIX Secur. Symp.*, 2015, pp. 49–64.
[84] B. Ramsundar and R. B. Zadeh, *TensorFlow for Deep Learning: From Linear Regression to Reinforcement Learning*. Newton, MA, USA: O'Reilly Media, 2018.
[85] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why should I trust you?': Explaining the predictions of any classifier," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, Aug. 2016, pp. 1135–1144.
[86] R. Russell *et al.*, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.
[87] C. Sabottke, O. Suciu, and T. Dumitras, "Vulnerability disclosure in the age of social media: Exploiting Twitter for predicting real-world exploits," in *Proc. USENIX Secur. Symp.*, 2015, pp. 1041–1056.
[88] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, "Predicting vulnerable software components via text mining," *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 993–1006, Oct. 2014.
[89] C. D. Sestili, W. S. Snavely, and N. M. VanHoudnos, "Towards security defect prediction with AI," 2018, *arXiv:1808.09897*. [Online]. Available: http://arxiv.org/abs/1808.09897
[90] H. Shahriar and M. Zulkernine, "Mitigating program security vulnerabilities: Approaches and challenges," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 1–46, Jun. 2012.
[91] L. K. Shar, L. C. Briand, and H. B. K. Tan, "Web application vulnerability prediction using hybrid program analysis and machine learning," *IEEE Trans. Dependable Secure Comput.*, vol. 12, no. 6, pp. 688–707, Nov. 2015.
[92] L. K. Shar and H. B. K. Tan, "Predicting common Web application vulnerabilities from input validation and sanitization code patterns," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, Sep. 2012, pp. 310–313.
[93] L. K. Shar and H. B. K. Tan, "Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1767–1780, Oct. 2013.
[94] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, Nov. 2011.
[95] Y. Shin and L. Williams, "An empirical model to predict security vulnerabilities using code complexity metrics," in *Proc. 2nd ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, Oct. 2008, pp. 315–317.
[96] Y. Shin and L. Williams, "Can traditional fault prediction models be used for vulnerability prediction?" *Empirical Softw. Eng.*, vol. 18, no. 1, pp. 25–59, Feb. 2013.
[97] R. Socher, C. C. Lin, C. Manning, and A. Y. Ng, "Parsing natural scenes and natural language with recursive neural networks," in *Proc. 28th Int. Conf. Mach. Learn. (ICML)*, 2011, pp. 129–136.
[98] J. Xiong, K. Zhang, and H. Zhang, "A vibrating mechanism to prevent neural networks from overfitting," in *Proc. 15th Int. Wireless Commun. Mobile Comput. Conf. (IWCMC)*, Jun. 2019, pp. 1929–1958.
[99] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, "Weakly supervised memory networks," 2015, *arXiv:1503.08895*. [Online]. Available: https://arxiv.org/abs/1503.08895
[100] S. Sukhbaatar *et al.*, "End-to-end memory networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 2440–2448.
[101] N. Sun, J. Zhang, P. Rimba, S. Gao, L. Y. Zhang, and Y. Xiang, "Data-driven cybersecurity incident prediction: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1744–1772, 2nd Quart., 2019.
[102] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[103] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. London, U.K.: Pearson Education, 2007.
[104] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proc. IEEE*, vol. 105, no. 12, pp. 2295–2329, Dec. 2017.
[105] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

networks," 2015, *arXiv:1503.00075*. [Online]. Available: http://arxiv.org/abs/1503.00075

[106] H. A. Thanassis, C. S. Kil, and B. David, "AEG: Automatic exploit generation," in *Proc. Ser. Netw. Distrib. Syst. Secur. Symp.*, 2011.

[107] A. Vaswani *et al.*, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 5998–6008.

[108] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, "Hackers vs. Testers: A comparison of software vulnerability discovery processes," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 374–391.

[109] D. Votipka, R. Stevens, E. Redmiles, J. Hu, and M. Mazurek, "Hackers vs. Testers: A comparison of software vulnerability discovery processes," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2018, pp. 134–151.

[110] S. Wang, T. Liu, and L. Tan, "Automatically learning semantic features for defect prediction," in *Proc. 38th Int. Conf. Softw. Eng. (ICSE)*, 2016, pp. 297–308.

[111] S. Wen, M. S. Haghighi, C. Chen, Y. Xiang, W. Zhou, and W. Jia, "A sword with two edges: Propagation studies on both positive and negative information in online social networks," *IEEE Trans. Comput.*, vol. 64, no. 3, pp. 640–653, Mar. 2015.

[112] J. Weston, S. Chopra, and A. Bordes, "Memory networks," 2014, *arXiv:1410.3916*. [Online]. Available: http://arxiv.org/abs/1410.3916

[113] D. A. Wheeler. (2016). *Flawfinder*. Accessed: May 20, 2018. [Online]. Available: https://www.dwheeler.com/flawfinder/

[114] M. White, C. Vendome, M. Linares-Vasquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, May 2015, pp. 334–345.

[115] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol. 1, no. 6, pp. 80–83, Dec. 1945.

[116] F. Wu, J. Wang, J. Liu, and W. Wang, "Vulnerability detection with deep learning," in *Proc. 3rd IEEE Int. Conf. Comput. Commun. (ICCC)*, Dec. 2017, pp. 1298–1302.

[117] T. Wu, S. Wen, Y. Xiang, and W. Zhou, "Twitter spam detection: Survey of new approaches and comparative study," *Comput. Secur.*, vol. 76, pp. 265–284, Jul. 2018.

[118] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," 2019, *arXiv:1901.00596*. [Online]. Available: http://arxiv.org/abs/1901.00596

[119] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proc. IEEE Symp. Secur. Privacy*, May 2014, pp. 590–604.

[120] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proc. 28th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2012, pp. 359–368.

[121] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, "Automatic inference of search patterns for taint-style vulnerabilities," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 797–812.

[122] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. SIGSAC ACM CCS*, 2013, pp. 499–510.

[123] W.-T. Yih, X. He, and C. Meek, "Semantic parsing for single-relation question answering," in *Proc. 52nd Annu. Meeting Assoc. Comput. Linguistics*, vol. 2, 2014, pp. 643–648.

[124] J. Zhang, Y. Xiang, Y. Wang, W. Zhou, Y. Xiang, and Y. Guan, "Network traffic classification using correlation information," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 1, pp. 104–117, Jan. 2013.

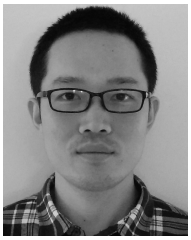
[125] T. Zhu, G. Li, W. Zhou, and P. S. Yu, "Differentially private data publishing and analysis: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 8, pp. 1619–1638, Aug. 2017.

[126] T. Zhu, P. Xiong, G. Li, W. Zhou, and S. Y. Philip, "Differentially private model publishing in cyber physical systems," *Future Gener. Comput. Syst.*, vol. 108, pp. 1297–1306, Jul. 2020.

[127] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Trans. Dependable Secure Comput.*, early access, Sep. 23, 2019, doi: 10.1109/TDSC.2019.2942930.

[128] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Aspects Comput.*, vol. 27, no. 3, pp. 573–609, 2015.

ABOUT THE AUTHORS



Guanjun Lin received the Ph.D. degree from the Swinburne University of Technology, Melbourne, VIC, Australia, in 2019.

He is currently working full-time as a Lecturer with Sanming University, Sanming, Fujian, China. His research interest is the application of deep-learning techniques for software vulnerability detection.

Sheng Wen (Member, IEEE) received the Ph.D. degree from Deakin University, Melbourne, VIC, Australia, in October 2014.

He has been working full time as a Senior Lecturer with the Swinburne University of Technology, Melbourne. Before this, he has been a Research Fellow and then a Lecturer in computer science with the School of Information Technology, Deakin University, since 2015. He manages several research projects in the last three years. Since late 2014, he has received over 3 million Australian dollars funding from both academia and industries. He is also leading a Medium-Size Research Team (around 15 members) in cybersecurity area. His research interests include IP network resilience and security and software security.

Qing-Long Han (Fellow, IEEE) received the B.Sc. degree in mathematics from Shandong Normal University, Jinan, China, in 1983, and the M.Sc. and Ph.D. degrees in control engineering and electrical engineering from the East China University of Science and Technology, Shanghai, China, in 1992 and 1997, respectively.

From September 1997 to December 1998, he was a Postdoctoral Research Fellow with the Laboratoire d'Automatique et d'Informatique Industielle (currently, Laboratoire d'Informatique etd'Automatique pour les Systemes), Ecole Sup erieure d'Ingenieurs de Poitiers (currently, Ecole Nationale Superieure d'Ingenieurs de Poitiers), Universite de Poitiers, Poitiers, France. From January 1999 to August 2001, he was a Research Assistant Professor with the Department of Mechanical and Industrial Engineering, Southern Illinois University at Edwardsville, Edwardsville, IL, USA. From September 2001 to December 2014, he was a Laureate Professor and an Associate Dean (Research and Innovation) with the Higher Education Division and the Founding Director of the Centre for Intelligent and Networked Systems, Central Queensland University, Rockhampton, QLD, Australia. From December 2014 to May 2016, he was the Deputy Dean (Research) with the Griffith Sciences and a Professor with the Griffith School of Engineering, Griffith University, Mount Gravatt, QLD, Australia. In May 2016, he joined the Swinburne University of Technology, Melbourne, VIC, Australia, where he is currently the Pro Vice-Chancellor (Research Quality) and a Distinguished Professor. His research interests include networked control systems, multiagent systems, time-delay systems, complex dynamical systems, and neural networks.

Dr. Han is a Fellow of The Institution of Engineers Australia. He is also an Associate Editor of several international journals, including the IEEE TRANSACTIONS ON CYBERNETICS, the IEEE TRANSACTIONS ON INDUSTRIAL ELECTRONICS, the IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS, the *IEEE Industrial Electronics Magazine*, the IEEE/CAA JOURNAL OF AUTOMATICA SINICA, *Control Engineering Practice*, and *Information Sciences*. He is a Highly Cited Researcher according to Clarivate Analytics (formerly Thomson Reuters).

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

Jun Zhang (Senior Member, IEEE) received the Ph.D. degree in computer science from the University of Wollongong, Wollongong, NSW, Australia, in 2011.

He is currently a Co-Founder and the Director of the Cybersecurity Lab, Swinburne University of Technology, Melbourne, VIC, Australia. He is also an Associate Professor. In particular, he is also leading his team to develop intelligent defense systems against sophisticated cyber attacks. He is also the chief investigator of several projects in cybersecurity, funded by the Australian Research Council (ARC). He has published more than 100 research papers in many international journals and conferences, such as the IEEE COMMUNICATIONS SURVEYS AND TUTORIALS, the IEEE TRANSACTIONS ON INFORMATION FORENSICS AND SECURITY, and the ACM Conference on Computer and Communications Security. Two of his articles were selected as the featured articles in the July/August 2014 issue of the IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING and the March/April 2016 issue of *IEEE IT Professional*. His research has been widely cited in the area of cybersecurity. He has been internationally recognized as an active researcher in cybersecurity, evidenced by his chairing of 15 international conferences and presenting invited keynote addresses in six conferences, and an Invited Lecturer in the IEEE SMC Victorian Chapter. His research interests include cybersecurity and applied machine learning.

Yang Xiang (Fellow, IEEE) received the Ph.D. degree in computer science from Deakin University, Melbourne, VIC, Australia, in 2007.

He is currently a Full Professor and the Dean of the Digital Research and Innovation Capability Platform, Swinburne University of Technology, Melbourne. He has published more than 200 research papers in many international journals and conferences, such as the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, the IEEE TRANSACTIONS ON INFORMATION SECURITY AND FORENSICS, and the IEEE JOURNAL ON SELECTED AREAS IN COMMUNICATIONS. He has published two books, *Software Similarity and Classification* and *Dynamic and Advanced Data Mining for Progressing Technological Development*. His research interests include cybersecurity, which covers networks and system security, data analytics, distributed systems, and networking.

Dr. Xiang has been a PC Member for more than 80 international conferences in distributed systems, networking, and security. He has served as the Program/General Chair for many international conferences, such as SocialSec 15, the IEEE DASC 2015/2014, the IEEE UbiSafe 2015/2014, the IEEE TrustCom 13, and so on. He served as an Associate Editor for the IEEE TRANSACTIONS ON COMPUTERS, the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, and *Security and Communication Networks* (Wiley). He also served as the Editor for the *Journal of Network and Computer Applications*. He is a Co-Ordinator, Asia, of the IEEE Computer Society Technical Committee on Distributed Processing (TCDP).

Authorized licensed use limited to: University of Exeter. Downloaded on June 09,2020 at 15:30:51 UTC from IEEE Xplore. Restrictions apply.