SySeVR: A Framework for Using Deep Learning to Detect Software Vulnerabilities

Zhen Li, Deqing Zou, Shouhuai Xu, *Senior Member, IEEE*, Hai Jin, *Fellow, IEEE*,
Yawei Zhu, and Zhaoxuan Chen

Abstract—The detection of software vulnerabilities (or vulnerabilities for short) is an important problem that has yet to be tackled, as manifested by the many vulnerabilities reported on a daily basis. This calls for machine learning methods for vulnerability detection. Deep learning is attractive for this purpose because it alleviates the requirement to manually define features. Despite the tremendous success of deep learning in other application domains, its applicability to vulnerability detection is not systematically understood. In order to fill this void, we propose the *first* systematic framework for using deep learning to detect vulnerabilities in C/C++ programs with source code. The framework, dubbed *Syntax-based, Semantics-based, and Vector Representations* (SySeVR), focuses on obtaining program representations that can accommodate syntax and semantic information pertinent to vulnerabilities. Our experiments with four software products demonstrate the usefulness of the framework: we detect 15 vulnerabilities that are not reported in the National Vulnerability Database. Among these 15 vulnerabilities, seven are unknown and have been reported to the vendors, and the other eight have been "silently" patched by the vendors when releasing newer versions of the pertinent software products.

Index Terms—Vulnerability detection, security, deep learning, program analysis, program representation

1 Introduction

Software vulnerabilities (or vulnerabilities for short) are a fundamental reason for the prevalence of cyber attacks. Despite academic and industrial efforts at improving software quality, vulnerabilities remain a big problem. This can be justified by the fact that each year, many vulnerabilities are reported in the *Common Vulnerabilities and Exposures* (CVE) [1].

Given that vulnerabilities are inevitable, it is important to detect them as early as possible. Source code-based static analysis is an important approach to detecting vulnerabilities, including *code similarity-based* methods [2], [3] and *pattern-based* methods [4], [5], [6], [7], [8], [9], [10]. Code similarity-based methods can detect vulnerabilities that are incurred by code cloning, but have high false-negatives when vulnerabilities are not caused by code cloning [11]. Pattern-based methods may require human experts to define vulnerability features for representing vulnerabilities, which makes them error-prone and laborious. Therefore, an ideal method should be able to effectively detect vulnerabilities caused by a wide range of reasons while imposing as little reliance on human experts as possible.

Deep learning — including *Recurrent Neural Networks* (RNNs) [12], [13], [14], [15], *Convolutional Neural Networks* (CNNs) [16], [17], [18], and *Deep Belief Networks* (DBNs) [19], [20] — has been successful in image and natural language processing. While it is tempting to use deep learning to detect vulnerabilities, we observe that there is a "domain gap": deep learning is born to cope with data with natural *vector representations* (e.g., pixels of images); in contrast, software programs do not have such vector representations. Recently, we proposed the *first* deep learning-based vulnerability detection system, dubbed VulDeePecker [11], to detect vulnerabilities at the slice level (i.e., multiple lines of code that are semantically related to each other). While demonstrating the feasibility of using deep learning to detect vulnerabilities, VulDeePecker has four weaknesses: (i) it considers only the vulnerabilities that are related to library/API function calls; (ii) it leverages only the semantic information induced by *data dependency*; (iii) it considers only a particular RNN known as *Bidirectional Long Short-Term Memory* (BLSTM); and (iv) it makes no effort to explain the cause of false-positives and false-negatives.

Our Contributions. In this paper, we propose the *first* systematic framework for using deep learning to detect vulnerabilities in C/C++ programs with source code. The framework is centered at answering the following question: *How can we*

- *Zhen Li is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China, and also with the School of Cyber Security and Computer, Hebei University, Baoding 071002, China. E-mail: lizhenhbu@gmail.com.*
- *Deqing Zou is with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: deqingzou@hust.edu.cn.*
- *Shouhuai Xu is with the Department of Computer Science, University of Colorado Colorado Springs, Colorado Springs, CO 80918 USA. E-mail: sxu@uccs.edu.*
- *Hai Jin, Yawei Zhu, and Zhaoxuan Chen are with the National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Security Engineering Research Center, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {hjin, yokisir, zhaoxaunchen}@hust.edu.cn.*

Manuscript received 18 Sept. 2018; revised 5 Nov. 2019; accepted 10 Jan. 2021. Date of publication 13 Jan. 2021; date of current version 9 July 2022.
(Corresponding author: Deqing Zou.)
Digital Object Identifier no. 10.1109/TDSC.2021.3051525

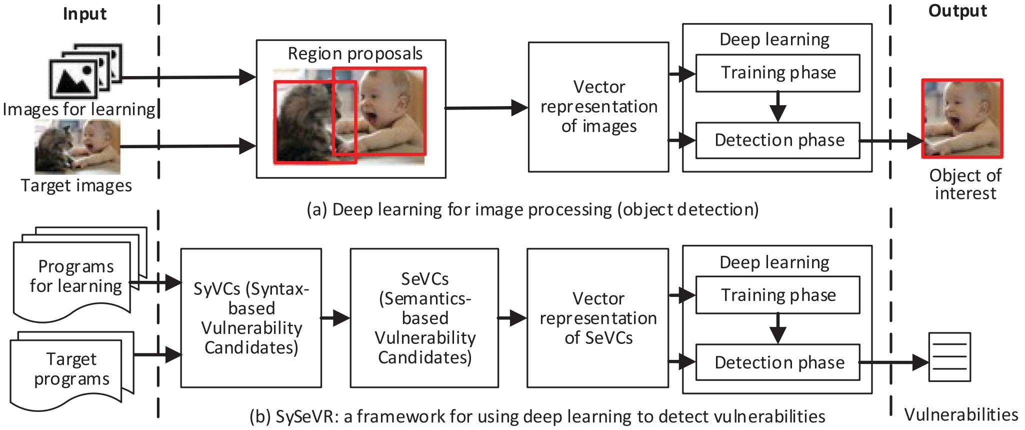

Input
Output
Images for learning
Target images
Region proposals
Vector representation of images
Deep learning
Training phase
Detection phase
Object of interest
(a) Deep learning for image processing (object detection)
Programs for learning
Target programs
SyVCs (Syntax-based Vulnerability Candidates)
SeVCs (Semantics-based Vulnerability Candidates)
Vector representation of SeVCs
Deep learning
Training phase
Detection phase
Vulnerabilities
(b) SySeVR: a framework for using deep learning to detect vulnerabilities

Fig. 1. (a) The notion of *region proposal* in image processing. (b) The SySeVR framework is inspired by the notion of region proposal and is centered on obtaining SyVC, SeVC, and vector representations of programs.

represent programs as vectors that accommodate the syntax and semantic information that is suitable for vulnerability detection? In order to answer this question, we introduce the notions of *Syntax-based Vulnerability Candidates* (SyVCs) and *Semantics-based Vulnerability Candidates* (SeVCs). Intuitively, SyVCs reflect vulnerability syntax characteristics, and SeVCs extend SyVCs to accommodate the semantic information induced by data dependency and control dependency. Moreover, we design algorithms to extract SyVCs and SeVCs automatically. This explains why we call the framework Syntax-based, Semantics-based, and Vector Representations, or SySeVR for short. As we will see, SySeVR overcomes the aforementioned weaknesses (i)-(iv) of VulDeePecker [11].

In order to evaluate the effectiveness of SySeVR, we present a dataset of 126 types of vulnerabilities, which are collected from the *National Vulnerability Database* (NVD) [21] and the *Software Assurance Reference Dataset* (SARD) [22]. This dataset should be of independent value and is made publicly available at https://github.com/SySeVR/SySeVR. It is worth mentioning that the dataset we published earlier in association to VulDeePecker [11] is not sufficient for the purpose of the present paper, simply because the dataset associated to [11] contains only 2 types of vulnerabilities.

Equipped with the new dataset, we show that SySeVR achieves the following.

- SySeVR enables multiple kinds of neural networks to detect various kinds of vulnerabilities. In the SySeVR framework, Bidirectional RNNs, especially *Bidirectional Gated Recurrent Unit* (BGRU), are more effective than unidirectional RNNs and CNNs, which are more effective than DBNs and shallow learning models. Moreover, SySeVR makes deep neural networks (especially BGRU) much more effective than the state-of-the-art vulnerability detection methods.
- The effectiveness of BGRU is substantially affected by the training data. If some syntax elements (e.g., tokens) often appear in vulnerable (versus not vulnerable) pieces of code, then these syntax elements may cause high false-positive rates (correspondingly, false-negative rates). This means that we can explain the cause of false-positives and false-negatives to some extent.
- Accommodating more semantic information (i.e., control dependency and data dependency) can improve the effectiveness of SySeVR-enabled vulnerability detectors. For example, semantic information induced by data dependency and control dependency can reduce the false-negative rate by 30.4 percent on average.
- By applying SySeVR-enabled BGRU to 4 software products (Libav, Seamonkey, Thunderbird, and Xen), we detect 15 vulnerabilities that have not been reported in NVD [21]. Among these 15 vulnerabilities, 7 are unknown to exist in these software products; for ethical reasons, we do *not* release the precise locations of these vulnerabilities, but we have reported them to the respective vendors. The other 8 vulnerabilities have been "silently" patched by the vendors when releasing newer versions of the pertinent software products.

Paper Outline. Section 2 presents the SySeVR framework. Section 3 describes experiments and results. Section 4 discusses limitations of the present study. Section 5 reviews related prior work. Section 6 concludes the paper.

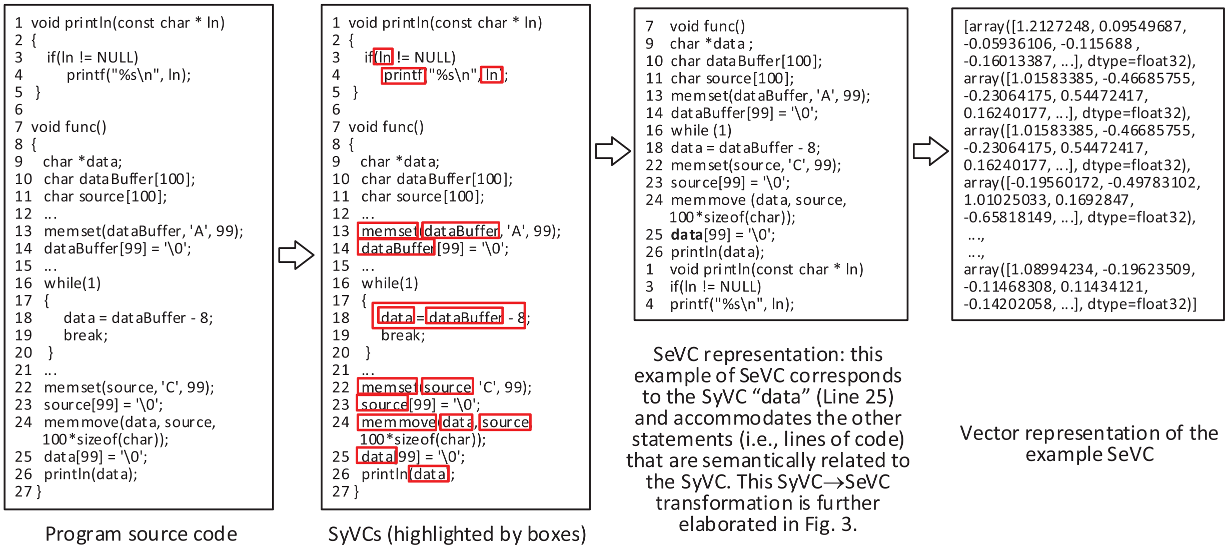
2 The SySeVR Framework

2.1 Basic Idea and Framework Overview

2.1.1 Basic Idea

Deep learning is successful in image processing and other applications. In particular, the notion of *region proposal* [23], [24] in image processing inspires us to adapt it to the context of vulnerability detection. However, the problem vulnerability detection is very different from the problem image processing because the latter has natural structural representations. To see the difference, let us consider an example of using deep learning to detect humans in images. On one hand, as illustrated in Fig. 1a, detecting humans in an image can be achieved by using the notion of *region proposal* and leveraging the structural representation of images (e.g., texture, edge, and color). Multiple region proposals can be extracted from an image, and each region proposal can be treated as a "unit" for training a neural network to detect objects (i.e., humans in this example).

On the other hand, when using deep learning to detect vulnerabilities, we need to represent programs in a way that can adequately accommodate the syntax and semantic information related to vulnerabilities. At a first glance, one may



```
1 void println(const char * ln)
2 {
3    if(ln != NULL)
4        printf("%s\n", ln);
5 }
6
7 void func()
8 {
9   char *data;
10  char dataBuffer[100];
11  char source[100];
12  ...
13  memset(dataBuffer, 'A', 99);
14  dataBuffer[99] = '\0';
15  ...
16  while(1)
17  {
18      data = dataBuffer - 8;
19      break;
20  }
21  ...
22  memset(source, 'C', 99);
23  source[99] = '\0';
24  memmove(data, source,
    100*sizeof(char));
25  data[99] = '\0';
26  println(data);
27 }
```

Program source code

```
1 void println(const char * ln)
2 {
3    if(ln != NULL)
4        printf("%s\n", ln);
5 }
6
7 void func()
8 {
9   char *data;
10  char dataBuffer[100];
11  char source[100];
12  ...
13  memset(dataBuffer, 'A', 99);
14  dataBuffer[99] = '\0';
15  ...
16  while(1)
17  {
18      data = dataBuffer - 8;
19      break;
20  }
21  ...
22  memset(source, 'C', 99);
23  source[99] = '\0';
24  memmove(data, source,
    100*sizeof(char));
25  data[99] = '\0';
26  println(data);
27 }
```

SyVCs (highlighted by boxes)

```
7   void func()
9   char *data ;
10  char dataBuffer[100];
11  char source[100];
13  memset(dataBuffer, 'A', 99);
14  dataBuffer[99] = '\0';
16  while (1)
18  data = dataBuffer - 8;
22  memset(source, 'C', 99);
23  source[99] = '\0';
24  memmove (data, source,
    100*sizeof(char));
25  data[99] = '\0';
26  println(data);
1   void println(const char * ln)
3   if(ln != NULL)
4   printf("%s\n", ln);
```

SeVC representation: this example of SeVC corresponds to the SyVC “data” (Line 25) and accommodates the other statements (i.e., lines of code) that are semantically related to the SyVC. This SyVC→SeVC transformation is further elaborated in Fig. 3.

```
[array([1.2127248, 0.09549687,
-0.05936106, -0.115688 ,
-0.16013387, ...], dtype=float32),
array([1.01583385, -0.46685755,
-0.23064175, 0.54472417,
0.16240177, ...], dtype=float32),
array([1.01583385, -0.46685755,
-0.23064175, 0.54472417,
0.16240177, ...], dtype=float32),
array([-0.19560172, -0.49783102,
1.01025033, 0.1692847,
-0.65818149, ...], dtype=float32),
 ...,
 ...,
array([1.08994234, -0.19623509,
-0.11468308, 0.11434121,
-0.14202058, ...], dtype=float32)]
```

Vector representation of the example SeVC

Fig. 2. An example illustrating SyVC, SeVC, and vector representations of program, where SyVCs are highlighted by boxes and one SyVC may be part of another SyVC. The SyVC→SeVC transformation is elaborated in Fig. 3.

suggest treating each *function* in a program as a region proposal in image processing. However, this is too coarse-grained because vulnerability detectors not only need to tell whether a function is vulnerable or not, but also need to pin down locations of vulnerabilities. That is, we need fine-grained representations of programs for vulnerability detection. One may also suggest treating each *line of code* or *statement* (i.e., these two terms will be used interchangeably) as a unit for vulnerability detection. However, this treatment has two drawbacks: (i) most statements in a program do not contain any vulnerability, meaning that few samples are vulnerable; and (ii) multiple statements that are semantically related to each other are not considered as a whole.

The preceding discussion suggests us to divide a program into smaller pieces of code (i.e., a number of statements), which correspond to “region proposals” and exhibit the *syntax* and *semantics* characteristics of vulnerabilities.

2.1.2 Framework Overview

We observe that vulnerabilities exhibit some *syntax characteristics*, such as function call or pointer usage. Therefore, we propose using syntax characteristics to identify SyVCs, which serve as a *starting point* for vulnerability detection (i.e., SyVCs are *not* sufficient for training deep learning models because they accommodate *no* semantic information of vulnerabilities). Fig. 1b highlights the SySeVR framework inspired by the notion of region proposal. Essentially, the framework seeks SyVC, SeVC, and vector representations of programs that are suitable for vulnerability detection.

In order to help understand SySeVR, we use the running example described in Fig. 2 to highlight how SySeVR extracts SyVC, SeVC, and vector representations of programs. At a high level, a SyVC, which is highlighted by a box in Fig. 2, is a code element that matches the syntax characteristics of some vulnerability. A SeVC extends a SyVC to include statements (i.e., lines of code) that are semantically related to the SyVC, where semantic information is induced by control dependency and/or data dependency; this “SyVC to SeVC” (or SyVC→SeVC) transformation is fairly involved and will be elaborated later (in Fig. 3). Finally, each SeVC is encoded into a vector for input to a deep neural network.

2.2 Extracting SyVCs

2.2.1 Extracting Vulnerability Syntax Characteristics

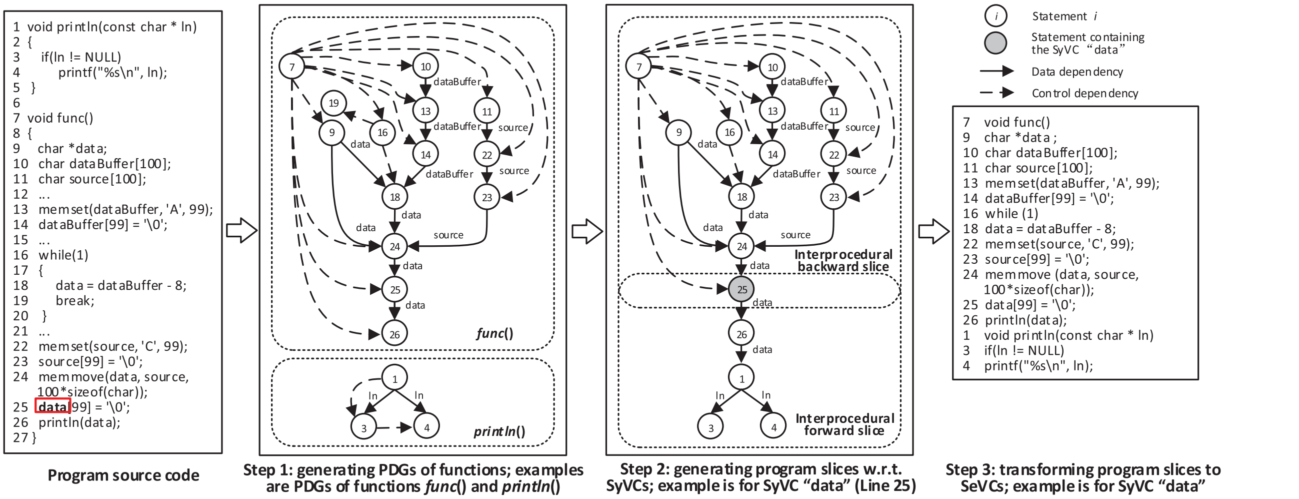
We propose using vulnerability syntax characteristics to identify pieces of code as *initial candidates* for vulnerability detection. For example, vulnerabilities associated to pointer usage would exhibit that the declaration of an identifier contains character ‘$*$’. In Fig. 2, the identifier “data” in Line 18 of the program source code is a pointer usage, and the declaration of “data” in Line 9 contains a character ‘$*$’.

Given that there are many vulnerabilities, we anticipate it to be extremely time-consuming to define and extract their syntax characteristics because this requires to extract the vulnerable lines of code from the vulnerable programs. While this itself is an important research problem, in Section 3.3.1 we will propose a specific method for extracting vulnerability syntax characteristics; we note that this method is far from perfect because it only covers 93.6 percent of the vulnerable programs we collected, but is sufficient for demonstrating the usefulness of SySeVR. In our method, we use the attributes of nodes on the *Abstract Syntax Tree* (AST) of a program to describe vulnerability syntax characteristics.

Regardless of the specific descriptions of vulnerability syntax characteristics, we can use $H = \{h_k\}_{1 \leq k \leq \beta}$ to denote a set of vulnerability syntax characteristics, where h_k represents a vulnerability syntax characteristic and β is the number of vulnerability syntax characteristics. Given H, we need to determine whether a piece of code matches a syntax characteristic h_k or not. Since these matching operations are specific to the representation of vulnerability syntax characteristics, we defer their description to our case study with a specific representation of vulnerability syntax characteristics.

2.2.2 Defining and Extracting SyVCs

We start with the definition of programs, functions, statements and tokens that will be used throughout of the present paper.



```
1  void println(const char * ln)
2  {
3     if(ln != NULL)
4         printf("%s\n", ln);
5  }
6
7  void func()
8  {
9    char *data;
10   char dataBuffer[100];
11   char source[100];
12   ...
13   memset(dataBuffer, 'A', 99);
14   dataBuffer[99] = '\0';
15   ...
16   while(1)
17   {
18       data = dataBuffer - 8;
19       break;
20    }
21   ...
22   memset(source, 'C', 99);
23   source[99] = '\0';
24   memmove(data, source,
     100*sizeof(char));
25   data[99] = '\0';
26   println(data);
27 }
```

Program source code

Step 1: generating PDGs of functions; examples are PDGs of functions *func*() and *println*()

Step 2: generating program slices w.r.t. SyVCs; example is for SyVC “data” (Line 25)

```
7   void func()
9   char *data ;
10  char dataBuffer[100];
11  char source[100];
13  memset(dataBuffer, 'A', 99);
14  dataBuffer[99] = '\0';
16  while (1)
18  data = dataBuffer - 8;
22  memset(source, 'C', 99);
23  source[99] = '\0';
24  memmove (data, source,
    100*sizeof(char));
25  data[99] = '\0';
26  println(data);
1   void println(const char * ln)
3   if(ln != NULL)
4   printf("%s\n", ln);
```

Step 3: transforming program slices to SeVCs; example is for SyVC “data”

Fig. 3. Elaborating the SyVC→SeVC transformation in Algorithm 2 for SyVC “$data$”, where solid arrows (i.e., directed edges) represent data dependency, and dashed arrows represent control dependency. Note that each solid arrow (i.e., data dependency) is annotated with the name of the variable that incurred the data dependency in question.

Definition 1 (program, function, statement, token). *A program P is a set of functions $f_1, \ldots, f_\eta$, denoted by $P = \{f_1, \ldots, f_\eta\}$. A function f_i, where $1 \leq i \leq \eta$, is an ordered set of statements $s_{i,1}, \ldots, s_{i,m_i}$, denoted by $f_i = \{s_{i,1}, \ldots, s_{i,m_i}\}$. A statement $s_{i,j}$, where $1 \leq i \leq \eta$ and $1 \leq j \leq m_i$, is an ordered set of tokens $t_{i,j,1}, \ldots, t_{i,j,w_{i,j}}$, denoted by $s_{i,j} = \{t_{i,j,1}, \ldots, t_{i,j,w_{i,j}}\}$. Note that tokens can be identifiers, operators, constants, and keywords, and can be extracted by lexical analysis.*

Given a function f_i, there are standard routines for generating its AST [25]. The root of the AST corresponds to function f_i, a leaf of the AST corresponds to a token $t_{i,j,g}$ ($1 \leq g \leq w_{i,j}$), and an internal node of the AST corresponds to a statement $s_{i,j}$ or multiple consecutive tokens of $s_{i,j}$. Intuitively, a SyVC corresponds to a leaf node of an AST, meaning that it is a token, or corresponds to an internal node of an AST, meaning that it is a statement or consists of multiple consecutive tokens. Formally,

Definition 2 (SyVC). *Consider a program $P = \{f_1, \ldots, f_\eta\}$, where $f_i = \{s_{i,1}, \ldots, s_{i,m_i}\}$ with $s_{i,j} = \{t_{i,j,1}, \ldots, t_{i,j,w_{i,j}}\}$. A* code element *$e_{i,j,z}$ is composed of one or multiple consecutive tokens of $s_{i,j}$, namely $e_{i,j,z} = (t_{i,j,u}, \ldots, t_{i,j,v})$ where $1 \leq u \leq v \leq w_{i,j}$. Given a set of vulnerability syntax characteristics $H = \{h_k\}_{1 \leq k \leq \beta}$, where h_k represents a vulnerability syntax characteristic and β is the number of vulnerability syntax characteristics as mentioned above, a code element $e_{i,j,z}$ that* matches *a vulnerability syntax characteristic h_k is called a SyVC, where the "matching" operation, as discussed above, is related to the specific representation of vulnerability syntax characteristics.*

Algorithm 1 gives a high-level description on the extraction of SyVCs from a given program $P = \{f_1, \ldots, f_\eta\}$ and a set $H = \{h_k\}_{1 \leq k \leq \beta}$ of vulnerability syntax characteristics. Specifically, Algorithm 1 uses a standard routine to generate an AST T_i for each function f_i. Then, Algorithm 1 traverses T_i to identify SyVCs, namely the code elements that "match" some h_k, where the "matching" operation is related to the representation of vulnerability syntax characteristics and therefore will be elaborated when coping with specific vulnerability syntax characteristics (see Section 3.3.1).

Algorithm 1. Extracting SyVCs From a Program

```
Input: A program $P = \{f_1, \ldots, f_\eta\}$; a set $H = \{h_k\}_{1 \leq k \leq \beta}$ of
       vulnerability syntax characteristics
Output: A set $Y$ of SyVCs
1:  $Y \leftarrow \emptyset$;
2:  for each function $f_i \in P$ do
3:    Generate an abstract syntax tree $T_i$ for $f_i$;
4:    for each code element $e_{i,j,z}$ in $T_i$ do
5:      for each $h_k \in H$ do
6:        if $e_{i,j,z}$ matches $h_k$ then
7:          $Y \leftarrow Y \cup \{e_{i,j,z}\}$;
8:        end if
9:      end for
10:   end for
11: end for
12: return $Y$; { the set of SyVCs}
```

In order to help understand the idea, we now consider an example. In the second column of Fig. 2, we use boxes to highlight all of the SyVCs that are extracted from the program source code using the vulnerability syntax characteristics that will be described in Section 3.3.1. We will elaborate how these SyVCs are extracted. It is worth mentioning that one SyVC may be part of another SyVC. For example, there are three SyVCs that are extracted from Line 18 because they are extracted with respect to different vulnerability syntax characteristics.

2.3 Transforming SyVCs to SeVCs

2.3.1 Basic Definitions

In order to detect vulnerabilities, we propose transforming SyVCs to SeVCs (i.e., SyVC→SeVC) to accommodate the statements that are semantically related to the SyVCs in question. For this purpose, we propose leveraging the *program slicing* technique to identify the statements that are semantically related to SyVCs. In order to use the program slicing technique, we need to use *Program Dependency Graph* (PDG). This requires us to use *data dependency* and *control dependency*, which are defined over *Control Flow Graph* (CFG). These concepts are reviewed below.

Definition 3 (CFG [26]). *For a program $P = \{f_1, \ldots, f_\eta\}$, the CFG of function f_i is a graph $G_i = (V_i, E_i)$, where $V_i = \{n_{i,1}, \ldots, n_{i,c_i}\}$ is a set of nodes with each node representing a statement or control predicate, and $E_i = \{\epsilon_{i,1}, \ldots, \epsilon_{i,d_i}\}$ is a set of direct edges with each edge representing the possible flow of control between a pair of nodes.*

Definition 4 (data dependency [26]). *Consider a program $P = \{f_1, \ldots, f_\eta\}$, the CFG $G_i = (V_i, E_i)$ of function f_i, and two nodes $n_{i,j}$ and $n_{i,\ell}$ in G_i where $1 \leq j, \ell \leq c_i$ and $j \neq \ell$. If there is a path from $n_{i,\ell}$ to $n_{i,j}$ in G_i and a value computed at node $n_{i,\ell}$ is used at node $n_{i,j}$, then $n_{i,j}$ is* data-dependent *on $n_{i,\ell}$.*

Definition 5 (control dependency [26]). *Consider a program $P = \{f_1, \ldots, f_\eta\}$, the CFG $G_i = (V_i, E_i)$ of function f_i, and two nodes $n_{i,j}$ and $n_{i,\ell}$ in G_i where $1 \leq j, \ell \leq c_i$ and $j \neq \ell$. It is said that $n_{i,j}$* post-dominates *$n_{i,\ell}$ if all paths from $n_{i,\ell}$ to the end of the program traverse through $n_{i,j}$. If there exists a path starting at $n_{i,\ell}$ and ending at $n_{i,j}$ such that (i) $n_{i,j}$* post-dominates *every node on the path excluding $n_{i,\ell}$ and $n_{i,j}$, and (ii) $n_{i,j}$ does not post-dominate $n_{i,\ell}$, then $n_{i,j}$ is* control-dependent *on $n_{i,\ell}$.*

Based on data dependency and control dependency, PDG can be defined as follows.

Definition 6 (PDG [26]). *For a program $P = \{f_1, \ldots, f_\eta\}$, the PDG of function f_i is denoted by $G'_i = (V_i, E'_i)$, where V_i is the same as in CFG G_i, and $E'_i = \{\epsilon'_{i,1}, \ldots, \epsilon'_{i,d'_i}\}$ is a set of direct edges with each edge representing a data or control dependency between a pair of nodes.*

2.3.2 Defining Program Slices

Given PDGs, we can extract *program slices* from SyVCs. We consider both forward and backward slices because (i) a SyVC may affect some subsequential statements, which may therefore contain a vulnerability; and (ii) the statements affecting a SyVC may render the SyVC vulnerable. Formally,

Definition 7 (forward, backward, and program slices [27] of a SyVC). *Consider a program $P = \{f_1, \ldots, f_\eta\}$, the PDG $G'_i = (V_i, E'_i)$ for each function f_i $(1 \leq i \leq \eta)$, and a SyVC, $e_{i,j,z}$, of statement $s_{i,j}$ in G'_i.*

- *The* forward slice *of SyVC $e_{i,j,z}$ in f_i, denoted by $\mathsf{fs}_{i,j,z}$, is defined as an ordered set of nodes $\{n_{i,x_1}, \ldots, n_{i,x_{\mu_i}}\} \subseteq V_i$, where n_{i,x_p}, $1 \leq x_1 \leq x_p \leq x_{\mu_i} \leq c_i$, is reachable from $e_{i,j,z}$ in G'_i. That is, the nodes in $\mathsf{fs}_{i,j}$ are from all paths in G'_i starting at $e_{i,j,z}$.*
- *The* interprocedural forward slice *of SyVC $e_{i,j,z}$ in program P, denoted by $\mathsf{fs}'_{i,j,z}$, is defined as an ordered set of nodes, where (i) a node belongs to one or multiple PDGs and (ii) each node is reachable starting from $e_{i,j,z}$ via a sequence of function calls. That is, $\mathsf{fs}'_{i,j,z}$ is a forward slice with or without crossing function boundaries (via function calls).*
- *The* backward slice *of SyVC $e_{i,j,z}$ in f_i, denoted by $\mathsf{bs}_{i,j,z}$, is defined as an ordered set of nodes $\{n_{i,y_1}, \ldots, n_{i,y_{\nu_i}}\} \subseteq V_i$, where n_{i,y_p}, $1 \leq y_1 \leq y_p \leq y_{\nu_i} \leq c_i$, from which $e_{i,j,z}$ is reachable in G'_i. That is, the nodes in $\mathsf{bs}_{i,j,z}$ are from all paths in G'_i ending at $e_{i,j,z}$.*
- *The* interprocedural backward slice *of SyVC $e_{i,j,z}$ in program P, denoted by $\mathsf{bs}'_{i,j,z}$, is defined as an ordered set of nodes, where (i) a node belongs to one or multiple PDGs and (ii) each node can reach $e_{i,j,z}$ via a sequence of function calls. That is, $\mathsf{bs}'_{i,j,z}$ is a backward slice with or without crossing function boundaries (via function calls).*
- *Given an interprocedural forward slice $\mathsf{fs}'_{i,j,z}$ and an interprocedural backward slice $\mathsf{bs}'_{i,j,z}$, the (interprocedural)* program slice*of SyVC $e_{i,j,z}$, denoted by $\mathsf{ps}_{i,j,z}$, is defined as an ordered set of nodes (belonging to the PDGs of functions in P) by merging $\mathsf{fs}'_{i,j,z}$ and $\mathsf{bs}'_{i,j,z}$ at the SyVC $e_{i,j,z}$. That is, $\mathsf{ps}_{i,j,z}$ is an ordered set obtained by connecting forward slice $\mathsf{fs}'_{i,j,z}$ and backward slice $\mathsf{bs}'_{i,j,z}$ in an order-preserving fashion while omitting the adjacent repeating nodes (i.e., using one node to replace the multiple adjacent appearances of the same node).*

In Fig. 3, the third column shows the interprocedural forward slice, the interprocedural backward slice, and the program slice of SyVC "data" (Line 25 in the program source code). The interprocedural forward slice of SyVC "data" crosses functions *func* and *println*. The interprocedural backward slice of SyVC "data" is the same as the backward slice of SyVC "data" in function *func*, because there is no other function that calls function *func*. The program slice of SyVC "data" is obtained by connecting the interprocedural forward slice and the interprocedural backward slice while omitting one (of the two) adjacent appearance of the node corresponding to SyVC "data" (Line 25 in the program source code).

Algorithm 2. Transforming SyVCs to SeVCs

```
Input: A program $P = \{f_1, \ldots, f_\eta\}$;
       a set $Y$ of SyVCs generated by Algorithm 1
Output: The set of SeVCs
$C \leftarrow \emptyset$;
for each $f_i \in P$ do
    Generate a PDG $G'_i = (V_i, E'_i)$ for $f_i$;
end for
for each $e_{i,j,z} \in Y$ in $G'_i$ do
    Generate forward slice $\mathsf{fs}_{i,j,z}$ & backward slice $\mathsf{bs}_{i,j,z}$ of $e_{i,j,z}$;
    Generate interprocedural forward slice $\mathsf{fs}'_{i,j,z}$ by interconnecting $\mathsf{fs}_{i,j,z}$ and the forward slices from the functions called by $f_i$;
    Generate interprocedural backward slice $\mathsf{bs}'_{i,j,z}$ by interconnecting $\mathsf{bs}_{i,j,z}$ and the backward slices from both the functions called by $f_i$ and the functions calling $f_i$;
    Generate program slice $\mathsf{ps}_{i,j,z}$ by connecting $\mathsf{fs}'_{i,j,z}$ and $\mathsf{bs}'_{i,j,z}$ at $e_{i,j,z}$;
    for each statement $s_{i,j} \in f_i$ appearing in $\mathsf{ps}_{i,j,z}$ as a node do
        $\delta_{i,j,z} \leftarrow \delta_{i,j,z} \cup \{s_{i,j}\}$, according to the order of the appearance of $s_{i,j}$ in $f_i$;
    end for
    for two statements $s_{i,j} \in f_i$ and $s_{a_p,b_q} \in f_{a_p}$ $(i \neq a_p)$ appearing in $\mathsf{ps}_{i,j,z}$ as nodes do
        if $f_i$ calls $f_{a_p}$ then
            $\delta_{i,j,z} \leftarrow \delta_{i,j,z} \cup \{s_{i,j}, s_{a_p,b_q}\}$, where $s_{i,j} < s_{a_p,b_q}$;
        else
            $\delta_{i,j,z} \leftarrow \delta_{i,j,z} \cup \{s_{i,j}, s_{a_p,b_q}\}$, where $s_{i,j} > s_{a_p,b_q}$;
        end if
    end for
    $C \leftarrow C \cup \{\delta_{i,j,z}\}$;
end for
return $C$; {the set of SeVCs}
```

2.3.3 Defining SeVCs

Having extracted program slices of SyVCs, we can now define SeVCs.

Definition 8 (SeVC). *Given a program $P = \{f_1, \ldots, f_\eta\}$ and a SyVC $e_{i,j,z}$ in statement $s_{i,j}$ of function f_i, the SeVC corresponding to SyVC $e_{i,j,z}$, denoted by $\delta_{i,j,z}$, is defined as an ordered subset of statements in P, denoted by $\delta_{i,j,z} = \{s_{a_1,b_1}, \ldots, s_{a_{v_{i,j,z}},b_{v_{i,j,z}}}\}$, where a data dependency or control dependency exists between statement s_{a_p,b_q} $(1 \leq p, q \leq v_{i,j,z})$ and SyVC $e_{i,j,z}$. In other words, a SeVC $\delta_{i,j,z}$ is an ordered set of statements that correspond to the nodes of (interprocedural) program slice $\mathsf{ps}_{i,j,z}$.*

2.3.4 Computing SeVCs

Algorithm 2 summarizes the preceding discussion in three steps: generating PDGs; generating program slices of the SyVCs output by Algorithm 1; and transforming program slices to SeVCs. In what follows we elaborate these steps and use Fig. 3 to illustrate a running example. Specifically, Fig. 3 elaborates the SyVC→SeVC transformation of SyVC "$data$" (related to pointer usage) while accommodating semantic information induced by data dependency and control dependency.

Step 1 (Lines 2-4 in Algorithm 2). This step generates a PDG for each function. For this purpose, there are standard algorithms (e.g., [26]). As a running example, the second column of Fig. 3 shows the PDGs respectively corresponding to functions $func$ and $println$, where each number represents the line number of a statement.

Step 2 (Lines 6-9 in Algorithm 2). This step generates the program slice $\mathsf{ps}_{i,j,z}$ for each SyVC $e_{i,j,z}$. The interprocedural forward slice $\mathsf{fs}'_{i,j,z}$ is obtained by merging $\mathsf{fs}_{i,j,z}$ and the forward slices from the functions called by f_i. The interprocedural backward slice $\mathsf{bs}'_{i,j,z}$ is obtained by merging $\mathsf{bs}_{i,j,z}$ and the backward slices from both the functions called by f_i and the functions calling f_i. Finally, $\mathsf{fs}'_{i,j,z}$ and $\mathsf{bs}'_{i,j,z}$ are merged into a program slice $\mathsf{ps}_{i,j,z}$.

As a running example, the third column in Fig. 3 shows the program slice of SyVC "$data$", where the backward slice corresponds to function $func$ and the forward slice corresponds to functions $func$ and $println$. It is worth mentioning that for obtaining the forward slice of a SyVC, we leverage only data dependency for two reasons: (i) statements affected by a SyVC via control dependency would not be vulnerable in most cases and (ii) utilizing statements that have a control dependency on a SyVC would involve many statements that have little to do with vulnerabilities. Consider for example a pointer variable SyVC in the condition expression of a "while" loop. If the pointer variable is not referred to in the body of the "while" loop, the statements in the body of the "while" loop are affected by the SyVC only via control dependency, meaning that the SyVC would not cause any vulnerability in the body of the "while" loop. If the forward slice of the pointer variable related SyVC mentioned above involves a control dependency, all of the statements in the body of the "while" loop, which are control-dependent on the SyVC, would be contained in the SeVC despite that they have little to do with vulnerabilities. On the other hand, for obtaining the backward slice of a SyVC, we leverage both data dependency and control dependency.

Step 3 (Lines 10-19 in Algorithm 2). This step transforms program slices to SeVCs as follows. First, the algorithm transforms the statements belonging to function f_i and appearing in $\mathsf{ps}_{i,j,z}$ as nodes to a SeVC, while preserving the order of these statements in f_i. As a running example shown in Fig. 3, 13 statements belong to function $func$, and 3 statements belong to function $println$. According to the order of these statements in the two functions, we obtain two ordered sets of statements: Lines {7, 9, 10, 11, 12, 14, 16, 18, 22, 23, 24, 25, 26} and Lines {1, 3, 4}.

Second, the algorithm transforms the statements belonging to different functions to a SeVC. For statements $s_{i,j} \in f_i$ and $s_{a_p,b_q} \in f_{a_p}$ $(i \neq a_p)$ appearing in $\mathsf{ps}_{i,j,z}$ as nodes, if f_i calls f_{a_p}, then $s_{i,j}$ and s_{a_p,b_q} are in the same order of function call, that is, $s_{i,j} < s_{a_p,b_q}$; otherwise, $s_{i,j} > s_{a_p,b_q}$. As a running example shown in Fig. 3, the SeVC is Lines {7, 9, 10, 11, 13, 14, 16, 18, 22, 23, 24, 25, 26, 1, 3, 4}, in which the statements in function $func$ appear before the statements in function $println$ because $func$ calls $println$. The fourth column in Fig. 3 shows the SeVC corresponding to SyVC "$data$", namely an order set of statements that are semantically related to SyVC "$data$".

2.4 Encoding SeVCs into Vectors

Algorithm 3 encodes SeVCs into vectors in three steps.

Step 1 (Lines 2-6 in Algorithm 3). In order to make SeVCs independent of user-defined variables and function names while capturing program semantic information, each SeVC $\delta_{i,j,z}$ is transformed to a *symbolic representation*. For this purpose, we propose removing non-ASCII characters and comments, then map user-defined variable names to symbolic names (e.g., "V1", "V2") in a one-to-one fashion, and finally map user-defined function names to symbolic names (e.g., "F1", "F2") in a one-to-one fashion. Note that different SeVCs may have the same symbolic representation. Please refer to [11] for more details about the mapping process.

Step 2 (Lines 8-13 in Algorithm 3). This step is to encode the symbolic representations into vectors. For this purpose, we propose dividing the symbolic representation of a SeVC $\delta_{i,j,z}$ (e.g., "V1 = V2 - 8;") into a sequence of symbols via a lexical analysis (e.g., "V1", "=", "V2", "-", "8", and ";"). We transform a symbol to a fixed-length vector. By concatenating the vectors, we obtain a vector $R_{i,j,z}$ for each SeVC.

Step 3 (Lines 14-22 in Algorithm 3). Because (i) the number of symbols (i.e., the vectors representing SeVCs) may be different and (ii) neural networks take vectors of the same length as input, we use a threshold θ as the length of vectors for the input to neural network. When a vector is shorter than θ, zeroes are padded to the end of the vector. When a vector is longer than θ, there are three scenarios but the basic idea is to make the SyVC appear in the middle of the resulting vector. (i) The sub-vector up to the SyVC is shorter than $\theta/2$. In this case, we delete the rightmost portion of $R_{i,j,z}$ to make the resulting vector have length θ. (ii) The sub-vector next to the SyVC is shorter than $\theta/2$. In this case, we delete the leftmost portion of $R_{i,j,z}$ to make the resulting vector have length θ. (iii) Otherwise, we keep the sub-vector of length $\lfloor(\theta-1)/2\rfloor$ immediately left to the SyVC and the sub-vector of length $\lceil(\theta-1)/2\rceil$ immediately right to the

SyVC. Together with the SyVC, we obtain a vector of length θ. For example, suppose $\theta = 15,000$ and the length of each symbol is 30, meaning that each SeVC has 500 symbols. Suppose the number of symbols in a SeVC is 510 (and thus needs to be reduced to 500) and the SyVC is at the position of the 255th symbol (among the 510 symbols), then we retain 249 consecutive symbols immediately left to the SyVC and 250 symbols immediately right to the SyVC. Together with the SyVC, we obtain a vector of 500 = 249 + 1 + 250 symbols. We stress that the preceding operations are well defined because each SyVC is transformed to a SeVC and appears exactly once in the SeVC.

Algorithm 3. Encoding SeVCs into Vectors

```
Input: A set $Y$ of SyVCs generated by Algorithm 1;
       a set $C$ of SeVCs corresponding to $Y$ and generated by Algorithm 2;
       a threshold $\theta$
Output: The set of vectors corresponding to SeVCs
$R \leftarrow \emptyset$;
for each $\delta_{i,j,z} \in C$ (corresponding to $e_{i,j,z} \in Y$) do
    Remove non-ASCII characters in $\delta_{i,j,z}$;
    Map variable names in $\delta_{i,j,z}$ to symbolic names;
    Map function names in $\delta_{i,j,z}$ to symbolic names;
end for
for each $\delta_{i,j,z} \in C$ (corresponding to $e_{i,j,z} \in Y$) do
    $R_{i,j,z} \leftarrow \emptyset$;
    Divide $\delta_{i,j,z}$ into a set of symbols $S$;
    for each $\alpha \in S$ in order do
        Transform $\alpha$ to a fixed-length vector $v(\alpha)$;
        $R_{i,j,z} \leftarrow R_{i,j,z}||v(\alpha)$, where $||$ means concatenation;
    end for
    if $R_{i,j,z}$ is shorter than $\theta$ then
        Zeroes are padded to the end of $R_{i,j,z}$;
    else if the sub-vector (of $\delta_{i,j,z}$) up to the position of the SyVC $e_{i,j,z}$ is shorter than $\theta/2$ then
        Delete the rightmost portion of $R_{i,j,z}$ to make the resulting vector of length $\theta$;
    else if the sub-vector (of $\delta_{i,j,z}$) next to the the position of the SyVC $e_{i,j,z}$ is shorter than $\theta/2$ then
        Delete the leftmost portion of $R_{i,j,z}$ to make the resulting vector of length $\theta$;
    else
        Keep the sub-vector (in $\delta_{i,j,z}$) immediately left to the position of the SyVC of length $\lfloor(\theta-1)/2\rfloor$, the sub-vector corresponding to the SyVC, and the sub-vector immediately right to the position of the SyVC of length $\lceil(\theta-1)/2\rceil$ {the resulting vector has length $\theta$;}
    end if
    $R \leftarrow R \cup R_{i,j,z}$;
end for
return $R$; {the set of vectors corresponding to SeVCs}
```

2.5 Labeling SeVCs and Corresponding Vectors

In order to learn a deep neural network, we label the vectors (i.e., the SeVCs they represent) as vulnerable or not as follows: A SeVC (i.e., the vector representing it) containing a known vulnerability is labeled as “1” (i.e., vulnerable), and “0” otherwise (i.e., not vulnerable). A learned deep neural network encodes vulnerability patterns and can detect whether given SeVCs are vulnerable or not.

3 Experiments and Results

3.1 Research Questions and Dataset

Research Questions. Our experiments are geared towards answering the following Research Questions (RQs):

- RQ1: Can SySeVR make BLSTM detect multiple kinds (versus single kind) of vulnerabilities?
- RQ2: Can SySeVR make multiple kinds of neural networks to detect multiple kinds of vulnerabilities? Can we explain their (in)effectiveness?
- RQ3: Can accommodating control-dependency make SySeVR more effective, and by how much?
- RQ4: How more effective are SySeVR-based methods when compared with the state-of-the-art methods?

In order to answer these questions, we implement the deep neural networks in Python using Tensorflow [28]. The computer running experiments has a NVIDIA GeForce GTX 1080 GPU and an Intel Xeon E5-1620 CPU running at 3.50 GHz.

Vulnerability Dataset. We produce a vulnerability dataset from two sources: NVD [21] and SARD [22]. NVD contains vulnerabilities in software products (i.e., software systems) and possibly `diff` files describing the difference between a vulnerable piece of code and its patched version. SARD contains production, synthetic and academic programs (also known as *test cases*), which are categorized as “good” (i.e., having no vulnerabilities), “bad” (i.e., having vulnerabilities), and “mixed” (i.e., having vulnerabilities whose patched versions are also available). Note that a program in NVD consists of one or several files (e.g., .c or .cpp files) which contain some vulnerability (corresponding to a CVE ID) or its patched version, and that a program in SARD is a test case.

For NVD, we focus on 19 popular C/C++ open source products (same as in [11]) and their vulnerabilities that are accompanied by `diff` files, which are needed for extracting vulnerable pieces of code. As a result, we collect 1,591 open source C/C++ programs, of which 874 are vulnerable. For SARD, we collect 14,000 C/C++ programs, of which 13,906 programs are vulnerable (i.e., “bad” or “mixed”). Note that a large number of these vulnerable programs belong to the “mixed” category and come with both the vulnerable functions and their patched versions. The average length of these programs is 573.5 lines of code. In total, we collect 15,591 programs, of which 14,780 are vulnerable; these vulnerable programs contain 126 *types* of vulnerabilities, where each type is uniquely identified by a *Common Weakness Enumeration IDentifier* (CWE ID) [29]. The 126 CWE IDs are published with our dataset.

3.2 Evaluation Metrics

The effectiveness of vulnerability detectors can be evaluated by the following widely-used metrics [30]: *false-positive rate* (*FPR*), *false-negative rate* (*FNR*), *accuracy* (*A*), *precision* (*P*), *F1-measure* (*F*1), and *Matthews Correlation Coefficient* (MCC) [31]. Let TP denote the number of vulnerable samples that are detected as vulnerable, FP denote the number of samples are not vulnerable but are detected as vulnerable, TN denote the number of samples that are not vulnerable (dubbed *non-vulnerable*) and are detected as not vulnerable, and FN denote the number of vulnerable samples that are detected as not vulnerable. The metric $FPR = \frac{\mathsf{FP}}{\mathsf{FP}+\mathsf{TN}}$ measures the proportion of

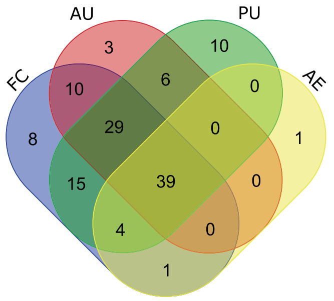


Fig. 4. Venn diagram of the FC, AU, PU, and AE in terms of the CWE IDs they cover, where $|\mathrm{FC}| = 106$, $|\mathrm{AU}| = 87$, $|\mathrm{PU}| = 103$, $|\mathrm{AE}| = 45$, and $|\mathrm{FC} \cup \mathrm{AU} \cup \mathrm{PU} \cup \mathrm{AE}| = 126$.

false-positive samples among the samples that are not vulnerable. The metric $FNR = \frac{\text{FN}}{\text{TP+FN}}$ measures the proportion of false-negative samples among the vulnerable samples. The metric $A = \frac{\text{TP+TN}}{\text{TP+FP+TN+FN}}$ measures the proportion of correctly detected samples among all samples. The metric $P = \frac{\text{TP}}{\text{TP+FP}}$ measures the proportion of truly vulnerable samples among the detected (or claimed) vulnerable samples. The metric $F1 = \frac{2 \cdot P \cdot (1-FNR)}{P+(1-FNR)}$ measures the overall effectiveness by considering both precision and false-negative rate. The metric $MCC = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP+FP})(\text{TP+FN})(\text{TN+FP})(\text{TN+FN})}}$ measures the degree to which model predictions match ground-truth labels; this metric is useful especially when dealing with imbalanced data, which is the case of the present paper because we have many more non-vulnerable samples than vulnerable ones.

3.3 Experiments

The experiments follow the SySeVR framework, with elaborations when necessary.

3.3.1 Extracting SyVCs

In what follows we will elaborate the two components in Algorithm 1 that are specific to different kinds of vulnerabilities: the extraction of vulnerability syntax characteristics and how to match them.

Extracting Vulnerability Syntax Characteristics. In order to extract syntax characteristics of known vulnerabilities, it would be natural to extract the vulnerable lines of code from the vulnerable programs mentioned above, and analyze their syntax characteristics. However, this is an extremely time-consuming task, which prompts us to leverage the C/C++ vulnerability rules of a state-of-the-art commercial tool, Checkmarx [6], to analyze vulnerability syntax characteristics. As we will see, this alternate method is effective because it covers 93.6 percent of the vulnerable programs collected from SARD. It is worth mentioning that we choose Checkmarx over open-source tools (e.g., Flawfinder [4] and RATS [5]) because the latter have simple parsers and imperfect rules [32].

Our *manual* examination of Checkmarx rules leads to the following 4 kinds of vulnerability syntax characteristics (each accommodating many vulnerabilities).

- *Library/API Function Call* (FC for short): This kind of syntax characteristic covers 811 library/API function calls, which are published with our dataset. These 811 function calls correspond to 106 CWE IDs.
- *Array Usage* (AU for short): This kind of syntax characteristic covers 87 CWE IDs related to arrays (e.g., issues related to array element access, array address arithmetic).
- *Pointer Usage* (PU for short): This kind of syntax characteristic covers 103 CWE IDs related to pointers (e.g., improper use in pointer arithmetic, reference, address transfer as a function parameter).
- *Arithmetic Expression* (AE for short): This kind of syntax characteristic covers 45 CWE IDs related to improper arithmetic expressions (e.g., integer overflow).

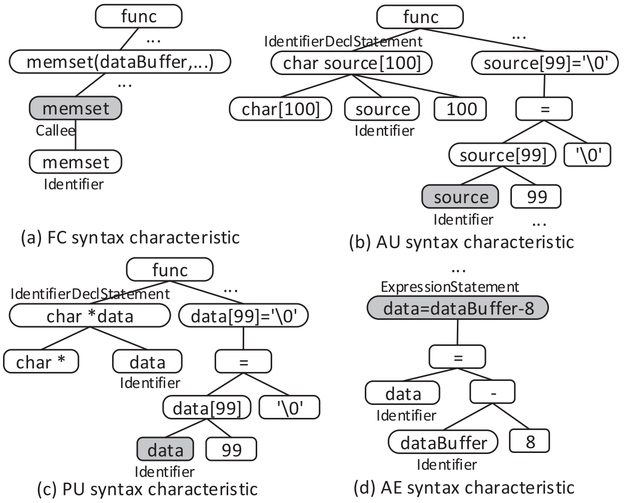


Fig. 5. Examples for illustrating the matching of syntax characteristics, where a highlighted node matches some vulnerability syntax characteristic and therefore is a SyVC.

Fig. 4 shows that these 4 kinds of syntax characteristics overlap with each other in terms of the CWE IDs they cover. These 4 kinds of syntax characteristics are generated from programs corresponding to 126 CWE IDs. Note that one kind of syntax characteristics may cover multiple CWE IDs and that one CWE ID may be covered by one or multiple kinds of syntax characteristics. For example, Fig. 4 shows that the vulnerabilities corresponding to 10 CWE IDs are covered by the PU-kind syntax characteristics but not others, and the vulnerabilities corresponding to 39 CWE IDs are covered by all of the 4 kinds of syntax characteristics (i.e., FC, AU, PU, and AE).

Matching Syntax Characteristics. In order to use Algorithm 1 to extract SyVCs, we need to determine whether or not a code element $e_{i,j,z}$, which is on the abstract syntax tree T_i of function f_i in program P, matches a vulnerability syntax characteristic. Note that T_i can be generated by using *Joern* [33]. The following method, as illustrated in Fig. 5 via the example program shown in Fig. 2, can automatically decide whether or not code element $e_{i,j,z}$ matches a syntax characteristic.

- As illustrated in Fig. 5a, we say code element $e_{i,j,z}$ (i.e., "memset") matches the FC syntax characteristic if (i) $e_{i,j,z}$ on T_i is a "callee" (i.e., the function is called), and (ii) $e_{i,j,z}$ is one of the 811 function calls mentioned above.
- As illustrated in Fig. 5b, we say code element $e_{i,j,z}$ (i.e., "source") matches the AU syntax characteristic if (i) $e_{i,j,z}$ is an identifier declared in an identifier declaration statement (i.e., IdentifierDeclStatement) node and (ii) the IdentifierDeclStatement node contains characters '[' and ']'.

- As illustrated in Fig. 5c, we say code element $e_{i,j,z}$ (i.e., "data") matches the PU syntax characteristic if (i) $e_{i,j,z}$ is an identifier declared in an IdentifierDeclStatement node and (ii) the IdentifierDeclStatement node contains character '$*$'.
- As illustrated in Fig. 5d, we say code element $e_{i,j,z}$ ("data=dataBuffer-8") matches the AE syntax characteristic if (i) $e_{i,j,z}$ is an expression statement (ExpressionStatement) node and (ii) $e_{i,j,z}$ contains a character '=' and has one or more identifiers on the right-hand side of '='.

Extracting SyVCs. Now we can use Algorithm 1 to extract SyVCs from the 15,591 programs. Corresponding to the 4 kinds of syntax characteristics, we extract 4 kinds of SyVCs:

- FC-kind SyVCs: We extract 6,356 from NVD and 58,047 from SARD, or 64,403 in total.
- AU-kind SyVCs: We extract 9,812 from NVD and 32,417 from SARD, or 42,229 in total.
- PU-kind SyVCs: We extract 73,890 from NVD and 217,951 from SARD, or 291,841 in total.
- AE-kind SyVCs: We extract 5,295 from NVD and 16,859 from SARD, or 22,154 in total.

Putting them together, we extract 420,627 SyVCs, which cover 13,016 (out of the 13,906, or 93.6 percent) vulnerable programs collected from SARD; this coverage validates our idea of using Checkmarx rules to derive vulnerability syntax characteristics. Note that we can compute the coverage 93.6 percent because SARD gives the precise location of each vulnerability; in contrast, we cannot compute the coverage with respect to NVD because it does not give precise locations of vulnerabilities. The average time for extracting a SyVC is 270 milliseconds.

3.3.2 Transforming SyVCs to SeVCs

When using Algorithm 2 to transform SyVCs to SeVCs, we use *Joern* [33] to extract PDGs. Corresponding to the 420,627 SyVCs extracted from Algorithm 1, Algorithm 2 generates 420,627 SeVCs (while recalling that one SyVC is transformed to one SeVC). In order to see the effect of semantic information, we actually use Algorithm 2 to generate two sets of SeVCs: one set accommodating semantic information induced by data dependency only, and the other set accommodating semantic information induced by both data dependency and control dependency. In either case, the second column of Table 1 summarizes the numbers of SeVCs categorized by the kinds of SyVCs from which they are transformed. In terms of the efficiency of the SyVC→SeVC transformation, on average it takes 331 milliseconds to generate a SeVC accommodating data dependency and 362 milliseconds to generate a SeVC accommodating data dependency and control dependency.

3.3.3 Encoding SeVCs into Vector Representation

We use Algorithm 3 to encode SeVCs into vectors. For this purpose, we adopt *word2vec* [34] to encode the symbols of the SeVCs (extracted from the 15,591 programs) into fixed-length vectors. The main hyper-parameters include: the dimensionality of word vectors is 30, the window size is 5, the training algorithm is skip-gram, and the threshold for configuring which higher-frequency words are randomly downsampled is 0.001. Then, each SeVC is represented by the concatenation of the vectors representing its symbols. We set each SeVC to have 500 symbols (padding or truncating if necessary, as discussed in Algorithm 3) and the length of each symbol is 30, meaning $\theta = 15,000$.

TABLE 1
The Number of SeVCs, Vulnerable SeVCs, and Non-Vulnerable SeVCs From the 15, 591 Programs

Kind of SyVCs	#SeVCs	#Vul. SeVCs	#Non-vul. SeVCs
FC-kind	64,403	13,603	50,800
AU-kind	42,229	10,926	31,303
PU-kind	291,841	28,391	263,450
AE-kind	22,154	3,475	18,679
Total	420,627	56,395	364,232

3.3.4 Generating Ground-Truth Labels of SeVCs

We generate ground-truth labels for the SeVCs in two steps. First, we generate *preliminary* labels automatically. For SeVCs extracted from NVD, we examine the vulnerabilities whose `diff` files contain *line deletion*, while noting that we do not consider the `diff` files that only contain *line addition* because NVD does not give the vulnerable statements in such cases. For a `diff` file containing *line deletion*, we parse it to mark and distinguish (i) the lines (i.e., statements) that are prefixed with "-" and are deleted/modified from (ii) the lines that are prefixed with "-" and are moved (i.e., deleted at one place and added at another place). If a SeVC contains at least one deleted/modified statement that is prefixed with "-", it is labeled as "1" (i.e., vulnerable); if a SeVC contains at least one moved statement prefixed with "-" and the detected file contains a known vulnerability, it is labeled as "1"; otherwise, it is labeled as "0" (i.e., not vulnerable). For SeVCs extracted from SARD, a SeVC extracted from a "good" program is labeled as "0" (i.e., not vulnerable); a SeVC extracted from a "bad" or "mixed" program is labeled as "1" (i.e., vulnerable) if the SeVC contains at least one vulnerable statement; otherwise, it is labelled as "0".

Second, in order to improve the quality of the preliminary labels mentioned above, we use stratified k-fold (k=5) cross validation to identify the vulnerable SeVCs that may have been mislabeled in the previous step (while noting that a true vulnerable sample is never mislabelled as "0") and check them manually, as follows. (i) The dataset is divided into 5 subsets. (ii) One subset is used as the validation set and the other 4 subsets are put together as the training set. (iii) The samples in the validation set are classified by the trained neural network. The false-negatives (i.e., the vulnerable samples that are not detected as vulnerable) are considered as the samples that may have been mislabeled. Then, we manually check these samples and correct the mislabeled samples. Steps (ii) and (iii) are repeated 5 times such that each subset is used as the validation set once. In total, we manually check the 2,605 samples that may have been mislabeled (i.e., 0.6 percent of all 420,627 samples). Among these 2,605 samples, we manually corrected 1,641 false-negatives (while noting that there are no false-positives because these 2,605 samples are all vulnerable).

TABLE 2
Effectiveness of VulDeePecker [11] Versus SySeVR-Enabled BLSTM (or SySeVR-BLSTM) for Detecting Vulnerabilities Related to Various Kinds of SyVCs (Metrics Unit: %)

Method	Kind of SyVC	FPR	FNR	A	P	F1	MCC
VulDee-Pecker	FC-kind	5.5	22.5	90.8	79.1	78.3	72.5
SySeVR-BLSTM	FC-kind	2.1	17.5	94.7	91.5	86.8	83.6
	AU-kind	3.8	17.1	92.7	88.3	85.5	80.7
	PU-kind	1.3	19.7	96.9	87.3	83.7	82.1
	AE-kind	1.5	18.3	96.6	87.9	84.7	82.9
	All-kinds	1.7	19.0	96.0	88.0	84.4	82.2

In total, 56,395 SeVCs are labeled as "1" and 364,232 SeVCs are labeled as "0". The third and fourth columns of Table 1 summarize the number of vulnerable versus not vulnerable SeVCs corresponding to each kind of SyVCs. The ground-truth label of the vector corresponding to a SeVC is the same as the ground-truth label of the SeVC.

3.4 Experimental Results

For the programs collected from NVD and SARD, we randomly select 80 percent of them as the training set (i.e., for training and validation) and the rest 20 percent of programs as the test set (i.e., for testing), respectively.

3.4.1 Experiments for Answering RQ1

In this experiment, we use BLSTM as in [11] and the SeVCs accommodating semantic information induced by data and control dependencies. We randomly choose 30,000 SeVCs extracted from the training programs as the training set and 7,500 SeVCs extracted from the test programs as the test set. Both sets contain SeVCs corresponding to the 4 kinds of SyVCs, proportional to the ratio of vulnerable versus non-vulnerable SeVCs in each kind of SyVCs. For fair comparison with VulDeePecker [11], we also randomly choose 30,000 SeVCs corresponding to the FC-kind SyVCs extracted from the training programs as the training set, and 7,500 SeVCs corresponding to the FC-kind SyVCs extracted from the test programs as the test set (also proportional to the ratio of vulnerable versus non-vulnerable SeVCs in the entire set of FC-kind SyVCs). Note that these SeVCs only accommodate semantic information induced by data dependency (as in [11]). We use the stratified 5-fold cross-validation to train deep neural networks, and choose the values of hyper-parameters that lead to the highest F1-measure (i.e., the overall vulnerability detection effectiveness). The main hyper-parameters we use to learn BLSTM are described as follows. The dropout is 0.2; the batch size is 16; the number of epochs is 20; the output dimension is 256; the minibatch stochastic gradient descent together with ADAMAX [35] is used for training with a default learning rate of 0.002; the dimension of hidden vectors is 500; and the number of hidden layers is 2.

Table 2 summarizes the results. We observe that SySeVR-BLSTM can detect vulnerabilities of the AU-kind with the lowest FNR (17.1 percent), but with a higher FPR than the other three kinds of vulnerabilities. It detects vulnerabilities of the FC-kind with the highest F1-measure (86.8 percent) and MCC (83.6 percent). The other three kinds of vulnerabilities lead to, on average, a FPR of 1.6 percent and a FNR of 18.5 percent. Overall, SySeVR-BLSTM achieves a 3.4 percent lower FPR and a 5.0 percent lower FNR than VulDeePecker when applied to detect vulnerabilities of the same kind (i.e., the FC-kind). This can be explained by the fact that SySeVR-BLSTM accommodates more semantic information (e.g., control dependency) via SeVCs. This leads to:

TABLE 3
Effectiveness of SySeVR-Enabled Different Kinds of Models in Detecting the Four Kinds of Vulnerabilities (Metrics Unit: %)

Model	FPR	FNR	A	P	F1	MCC
LR	2.0	45.5	92.1	80.8	65.1	62.5
MLP	2.0	37.3	93.1	82.1	71.1	68.1
DBN	2.0	44.0	91.6	82.1	66.6	63.5
CNN	2.0	17.9	95.7	85.6	83.8	81.4
LSTM	2.0	21.7	95.2	85.2	81.6	79.0
GRU	2.0	17.6	95.7	85.7	84.0	81.7
BLSTM	2.0	15.7	96.0	86.2	84.3	83.0
BGRU	2.0	14.7	96.0	86.4	85.8	83.7

Insight 1. *SySeVR-BLSTM can detect vulnerabilities related to function calls, array usage, pointer usage and arithmetic expressions, and can achieve a 3.4 percent lower FPR and a 5.0 percent lower FNR when compared with VulDeePecker in detecting vulnerabilities related to library/API function calls.*

3.4.2 Experiments for Answering RQ2

In order to answer RQ2, we use the stratified 5-fold cross-validation to train 8 standard models: a linear *Logistic Regression* (LR) classifier, a neural network with one hidden layer *Multi-Layer Perception* (MLP), a DBN [36], a CNN [37], and four RNNs (i.e., *Long Short-Term Memory* (LSTM), *Gated Recurrent Unit* (GRU), BLSTM, and BGRU [38], [39], [40]), using the same dataset (of 4 kinds of SyVCs) as in Section 3.4.1. In each case, we choose the hyper-parameter value that leads to the highest F1-measure.

Table 3 summarizes the results by setting FPR to 2.0 percent in each model, which is chosen because it is the FPR of the model that achieves the highest F1-measure. We observe that when compared with unidirectional RNNs (i.e., LSTM and GRU), bidirectional RNNs (i.e., BLSTM and BGRU) can respectively improve the FNR by 4.5 percent and the F1-measure by 2.3 percent on average. This improvement might be caused by the following: Bidirectional RNNs can accommodate more information about the statements that appear before and after the statement in question. We further observe that bidirectional RNNs (especially BGRU) are more effective than CNN, which in turn is more effective than DBN and shallow learning models (i.e., LR and MLP). Moreover, these models achieve a similar effectiveness in both MCC and F1-measure, meaning that the issue of data imbalance is not significant. In summary,

Insight 2. *SySeVR-enabled bidirectional RNNs (especially BGRU) are more effective than SySeVR-enabled unidirectional RNNs and CNN, which are more effective than SySeVR-enabled DBN and shallow learning models (i.e., LR and MLP). Still, FNRs of all these models are consistently much higher than their FPRs.*

TABLE 4
Effectiveness of SySeVR-Enabled Models Using Vectors Derived From Bag-of-Words (Metrics Unit: %)

Model	FPR	FNR	A	P	F1	MCC
LR	2.0	34.4	93.5	83.4	73.4	70.5
MLP	2.0	29.7	94.1	84.2	76.6	73.7
CNN	2.0	55.1	90.6	76.8	56.7	54.2
BGRU	2.0	63.3	89.5	72.8	48.8	46.9

SySeVR-enabled models mentioned above adopt word2vec [34] to generate vectors. In order to see if word2vec can be replaced by a simpler vector representation, say *token frequency*, we use bag-of-words [41] to encode SeVCs into fixed-length vectors. With this vector representation, we conduct experiments using two shallow models (i.e., LR and MLP) and two deep neural networks (i.e., CNN and BGRU). Table 4 reports the experimental results. We observe that the best result for word2vec (BGRU achieving an F1 of 85.8 percent and a MCC of 83.7 percent as shown in Table 3) is much better than the best result for bag-of-words (MLP achieving an F1 of 76.6 percent and a MCC of 73.7 percent). For bag-of-words, we observe that shallow models are more effective than deep neural networks; for word2vec, deep neural networks are more effective than shallow models. In particular, BGRU, which is the most effective for word2vec (F1 of 85.8 percent and MCC of 83.7 percent), is the least effective for bag-of-words (F1 of 48.8 percent and MCC of 46.9 percent). This can be explained by the fact that there is no context information for the vectors generated by bag-of-words, causing BGRU not to be able to capture the context and achieve a low effectiveness. This leads to:

Insight 3. *Using a distributed representation, such as word2vec, to capture context information is important to SySeVR. In particular, a representation centered at token frequency is not sufficient.*

Because of this, we always use word2vec to generate vectors for the experiments that will be discussed in the rest of the paper.

Towards Explaining the Effectiveness of BGRU in Vulnerability Detection. It is important, but an outstanding open problem, to explain the effectiveness of deep neural networks. Now we report our initial effort along this direction. In what follows we focus on BGRU because it is more effective than the others.

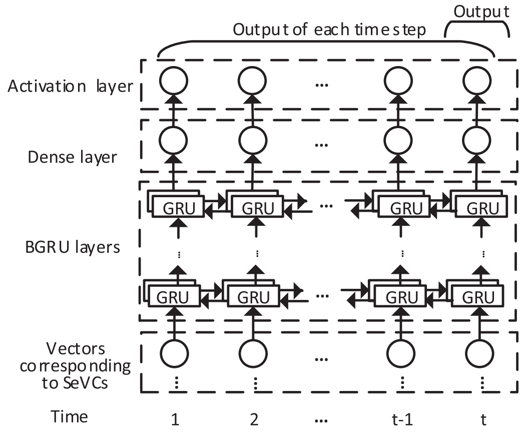


Fig. 6. The structure of BGRU.

In order to explain the effectiveness of BGRU, we review its structure in Fig. 6. For each SeVC and each time step, there is an output (belonging to [0, 1]) at the activation layer. The output of BGRU is the output of the last time step at the activation layer; the closer this output is to 1, the more likely the SeVC is classified as vulnerable. For the classification of a SeVC, we identify the tokens (i.e., the symbols representing them) that play a critical role in determining its classification. This can be achieved by looking at all pairs of tokens at time steps $(t', t' + 1)$. We find that if the activation-layer output corresponding to the token at time step $t' + 1$ is substantially (e.g., 0.6) greater (versus smaller) than the activation-layer output corresponding to the token at time step t', then the token at time step $t' + 1$ plays a critical role in classifying the SeVC as vulnerable (correspondingly, not vulnerable). Moreover, we find that some false-negatives are caused by the token "`if`" or the tokens following it, because these tokens frequently appear in SeVCs that are not vulnerable. We also find that some false-positives are caused by the tokens related to library/API function calls and their arguments, because these tokens frequently appear in SeVCs that are vulnerable. In summary,

Insight 4. *If a syntax element (e.g., token) appears in vulnerable (resp. non-vulnerable) SeVCs much more frequently than appearing in non-vulnerable (resp. vulnerable) ones, the syntax element may cause false-positives (resp. false-negatives); this means that appearance frequency of syntax elements matters.*

3.4.3 Experiments for Answering RQ3

We use experiments to compare the effectiveness of (i) the 8 models learned from the SeVCs that accommodate semantic information induced by data dependency and (ii) the 8 models learned from the SeVCs that accommodate semantic information induced by data dependency and control dependency. In either case, we randomly choose 30,000 SeVCs extracted from the training programs as the training set and 7,500 SeVCs extracted from the test programs as the test set. All of these training and test sets correspond to the 4 kinds of SyVCs, proportional to the amount of vulnerable versus non-vulnerable SeVCs for each kind of SyVCs.

Table 5 summarizes the results by setting FPR to 2.0 percent in each model because 2.0 percent is the FPR of the model achieving the highest F1-measure (i.e., BGRU using data dependency and control dependency). For models learned from the datasets that accommodate data dependency, we observe that CNN and bidirectional RNNs (i.e., BLSTM and BGRU) are much more effective than DBN and shallow learning models (i.e., LR and MLP). When compared with the models that are learned from the datasets that accommodate data dependency only, we observe that the models learned from the datasets that accommodate both data dependency and control dependency can improve FNR by 30.4 percent and F1-measure by 24.0 percent on average. This can be explained by the fact that control dependency accommodates extra information useful to vulnerability detection.

Insight 5. *A model which accommodates more semantic information (i.e., control dependency and data dependency) achieves a higher vulnerability detection capability.*

TABLE 5
Effectiveness of Semantic Information Induced by Data Dependency ("DD" for Short) Versus Induced by Data Dependency and Control Dependency ("DDCD" for Short) (Metrics Unit: %)

Model	Kind of SeVC	FPR	FNR	A	P	F1	MCC
LR	DD	2.0	69.7	88.6	69.7	42.2	41.0
	DDCD	2.0	45.5	92.1	80.8	65.1	62.5
MLP	DD	2.0	66.9	89.0	72.0	45.4	44.0
	DDCD	2.0	37.3	93.1	82.1	71.1	68.1
DBN	DD	2.0	78.5	87.4	63.0	32.0	31.7
	DDCD	2.0	44.0	91.6	82.1	66.6	63.5
CNN	DD	2.0	42.9	92.3	81.3	67.0	64.0
	DDCD	2.0	17.9	95.7	85.6	83.8	81.4
LSTM	DD	2.0	68.6	88.7	70.0	43.4	41.9
	DDCD	2.0	21.7	95.2	85.2	81.6	79.0
GRU	DD	2.0	42.8	92.3	81.7	67.3	64.4
	DDCD	2.0	17.6	95.7	85.7	84.0	81.7
BLSTM	DD	2.0	45.7	92.1	82.1	65.3	62.7
	DDCD	2.0	15.7	96.0	86.2	84.3	83.0
BGRU	DD	2.0	42.3	92.5	82.3	67.8	65.0
	DDCD	2.0	14.7	96.0	86.4	85.8	83.7

3.4.4 *Experiments for Answering RQ4*

We consider BGRU learned from the 341,536 SeVCs corresponding to the 4 kinds of SyVCs extracted from the training programs and the 79,091 SeVCs extracted from the test programs, while accommodating semantic information induced by data dependency and control dependency. We compare our most effective model BGRU with the commercial static vulnerability detection tool Checkmarx [6] and open-source static analysis tools Flawfinder [4] and RATS [5], because (i) these tools arguably represent the state-of-the-art static analysis for vulnerability detection; (ii) they are widely used for detecting vulnerabilities in C/C++ source code; (iii) they directly operate on the source code (i.e., no need to compile the source code); and (iv) they are available to us. We also consider the state-of-the-art system VUDDY [2], which is particularly suitable for detecting vulnerabilities incurred by code cloning. We further consider VulDeePecker [11], and we consider all 4 kinds of SyVCs and data as well as control dependency for SySeVR.

Table 6 summarizes the experimental results. We observe that SySeVR-enabled BGRU substantially outperforms the state-of-the-art vulnerability detection methods. The open-source Flawfinder and RATS have high FPRs and FNRs. Checkmarx is better than Flawfinder and RATS, but still has high FPRs and FNRs. VUDDY is known to trade a high FNR for a low FPR, because it can only detect vulnerabilities that are nearly identical to the vulnerabilities in the training programs. SySeVR-enabled BGRU is much more effective than VulDeePecker because VulDeePecker cannot cope with other kinds of SyVCs (than FC) and cannot accommodate semantic information induced by control dependency. Moreover, BGRU learned from a larger training set (i.e., 341,536 SeVCs) is more effective than BGRU learned from a smaller training set (30,000 SeVCs; see Table 3), especially reducing FNR by 9.1 percent. In summary,

Insight 6. *SySeVR-enabled BGRU is much more effective than the state-of-the-art vulnerability detection methods.*

TABLE 6
Comparing BGRU in the SySeVR Framework and State-of-the-Art Vulnerability Detectors (Metrics Unit: %)

Method	FPR	FNR	A	P	F1	MCC
Flawfinder	21.6	70.4	69.8	22.8	25.7	22.1
RATS	21.5	85.3	67.2	12.8	13.7	12.6
Checkmarx	20.8	56.8	72.9	30.9	36.1	33.0
VUDDY	4.3	90.1	71.2	47.7	16.4	15.2
VulDeePecker	2.5	41.8	92.2	78.0	66.6	64.9
SySeVR-BGRU	1.4	5.6	98.0	90.8	92.6	90.5

3.4.5 *Applying BGRU to Detect Vulnerabilities in Software Products*

In order to show the usefulness of SySeVR in detecting software vulnerabilities in real-world software products, we apply SySeVR-BGRU trained in Section 3.4.4 to detect vulnerabilities in 4 software products: Libav, Seamonkey, Thunderbird, and Xen. Each of these products contains multiple targets programs, from which we extract their SyVCs, SeVCs, and vectors. For each product, we apply SySeVR-enabled BGRU to its 20 versions so that we can tell whether some vulnerabilities have been "silently" patched by the vendors when releasing a newer version.

As highlighted in Table 7, we detect 15 vulnerabilities that are *not* reported in NVD. Among them, 7 are unknown (i.e., their presence in these products are not known until now) and are indeed similar (upon our manual examination) to the *CVE IDentifiers* (CVE IDs) mentioned in Table 7. We do not give the full details of these vulnerabilities for ethical considerations, but we have reported these 7 vulnerabilities to the vendors. The other 8 vulnerabilities have been "silently" patched by the vendors when releasing newer versions of the products in question. Checkmarx, which we use to extract vulnerability syntax characteristics, missed all of these vulnerabilities except the two in Seamonkey 2.35 and Thunderbird 38.0.1, demonstrating its ineffectiveness.

4 Limitations

The present study has several limitations. *First*, we focus on detecting vulnerabilities in C/C++ program source code, meaning that the framework may need to be adapted to cope with other programming languages and/or executables. *Second*, our experiments focus 4 kinds of vulnerability syntax characteristics, which cover 93.6 percent of the vulnerable programs collected from SARD. This coverage is not perfect while noting that the SARD data may not be representative of real-world software products. Future research needs to identify more complete vulnerability syntax characteristics. *Third*, the algorithms for generating SyVCs and SeVCs could be improved to accommodate more syntactic/semantic information for vulnerability detection. *Fourth*, our experiments use a single model to detect multiple kinds of vulnerabilities. Future research should investigate which of the following is more effective: using multiple models that are respectively tailored to detect multiple kinds of vulnerabilities versus using a single model to detect multiple kinds of vulnerabilities. *Fifth*, we detect vulnerabilities at the slice level (i.e., multiple lines of code that are semantically related to each other), which could

TABLE 7
The 15 Vulnerabilities, Which are Detected by BGRU but not Reported in the NVD, Include 7 Unknown Vulnerabilities and 8 Vulnerabilities That Have Been "Silently" Patched

Target product	CVE ID	Vulnerable product reported	Vulnerability release date	Vulnerable file in the target product	Kind of SyVC	1st patched version of target product
Libav 10.3	CVE-2013-7020	FFmpeg	12/09/2013	libavcodec/ffv1dec.c	PU-kind	Libav 10.4
Libav 10.3, Libav 12.3	CVE-2013-****	FFmpeg	**/**/2013	libavcodec/**.c	AU-kind	–
	CVE-2013-****	FFmpeg	**/**/2013	libavcodec/**.c	PU-kind	–
	CVE-2014-****	FFmpeg	**/**/2015	libavcodec/**.c	PU-kind	–
	CVE-2014-****	FFmpeg	**/**/2014	libavcodec/**.c	PU-kind	–
Libav 9.10	CVE-2014-9676	FFmpeg	02/27/2015	libavformat/segment.c	PU-kind	Libav 10.0
Seamonkey 2.32	CVE-2015-4511	Firefox	09/24/2015	.../src/nestegg.c	AU-kind	Seamonkey 2.38
Seamonkey 2.35	CVE-2015-****	Firefox	**/**/2015	.../gonk/**.cpp	FC-kind	–
Thunderbird 38.0.1	CVE-2015-4511	Firefox	09/24/2015	.../src/nestegg.c	AU-kind	Thunderbird 43.0b1
	CVE-2015-****	Firefox	**/**/2015	.../gonk/**.cpp	FC-kind	–
Xen 4.4.2	CVE-2013-4149	Qemu	11/04/2014	.../net/virtio-net.c	PU-kind	Xen 4.4.3
	CVE-2015-1779	Qemu	01/12/2016	ui/vnc-ws.c	PU-kind	Xen 4.5.5
	CVE-2015-3456	Qemu	05/13/2015	.../block/fdc.c	PU-kind	Xen 4.5.1
Xen 4.7.4	CVE-2016-4453	Qemu	06/01/2016	.../display/vmware_vga.c	AE-kind	Xen 4.8.0
Xen 4.8.2, Xen 4.12.0	CVE-2016-****	Qemu	**/**/2016	.../net/**.c	PU-kind	–

be improved to more precisely pin down the *line* of code that contains a vulnerability. *Sixth*, we generate ground-truth labels by manually checking 0.6 percent of all samples, which may have been mislabeled by the automatic method we use (owing to the lack of ground-truth dataset). Future research should investigate more effective automatic labeling methods; for this purpose, one may leverage the idea of co-training [42]. *Seventh*, our experiments show some deep neural networks are more effective than the state-of-the-art vulnerability detection methods. Although we have gained some insights into explaining the "why" part, more investigations are needed to explain the success of deep learning in this context and beyond.

5 Related Work

Prior Studies Related to Vulnerability Detection. There are two methods for source code-based static vulnerability detection: *code similarity-based* versus *pattern-based*. Since code similarity-based detectors can only detect vulnerabilities incurred by code cloning and SySeVR is a pattern-based method, we only review prior studies in pattern-based methods, which can be further divided into *rule-based* and *machine learning-based* methods.

Rule-based methods use vulnerability patterns to detect vulnerabilities, where patterns are manually generated by human experts (e.g., Flawfinder [4], RATS [5], Checkmarx [6]). These tools often incur high false-positive rates and/or high false-negative rates [32], as also confirmed by our experiments (Section 3.4.4). Vulnerability patterns can be defined using, for example, *code property graphs* [33]. In contrast, SySeVR uses vulnerability patterns that are learned automatically and represented by deep neural networks.

Machine learning-based methods, as discussed elsewhere [43], can be further divided into the following three sub-categories. (i) Vulnerability prediction methods based on software metrics: These methods are built on top of software metrics (e.g., imports and function calls [8], complexity [44], [45], code churn and developer activity [45], [46]), but predict vulnerabilities at a coarse granularity (e.g., component-level [8] or file-level [45]), meaning that they cannot pin down the locations of vulnerabilities. (ii) Anomaly detection methods: These methods find vulnerabilities via abnormal patterns in (for example) API usage [47] or missing checks [9], [48], but cannot cope with rarely-used but normal patterns. (iii) Vulnerable code pattern recognition methods: These methods extract vulnerability patterns related to (for example) ASTs [10], code property graphs [49] or system calls [7], and use these patterns to detect vulnerabilities. These methods demand human experts to define features and use the traditional machine learning models (e.g., support vector machine and k-nearest neighbor) to detect vulnerabilities. Recently, deep learning has been leveraged for vulnerability detection, while alleviating the problem of manual feature definition. Lin *et al.* [15] presented a method for automatically learning high-level representations of functions (i.e., coarse-grained). VulDeePecker [11] is the first system showing the feasibility of using deep learning to detect vulnerabilities at the slice level, which is much finer than the function level. A more recent development is μVulDeePecker [50], which extends VulDeePecker to detect *multiclass* vulnerabilities. SySeVR overcomes the weaknesses of VulDeePecker discussed in Section 1, and is the first systematic framework for using deep learning to detect vulnerabilities.

Prior Studies Related to Deep Learning. Deep learning has been used for program analysis. CNN has been used for software defect prediction [16] and locating buggy source code [51]; DBN has been used for software defect prediction [19], [20]; RNN has been used for vulnerability detection [11], [15], [50], software traceability [12], code clone detection [13], and recognizing functions in binaries [14]. The present study offers the first framework for using deep learning to detect vulnerabilities.

6 Conclusion

We presented the SySeVR framework for using deep learning to detect vulnerabilities. Based on a large dataset of vulnerability we collected, we drew a number of insights, including an explanation on the effectiveness of deep learning in vulnerability detection. Moreover, we detected 15 vulnerabilities that were *not* reported in the NVD. Among these 15 vulnerabilities, 7 are unknown and have been reported to the vendors, and the other 8 have been "silently" patched by the vendors when releasing newer versions. There are many open problems for future research. In addition to addressing the limitations discussed in Section 4, it is important to investigate the impact of *code duplication* [52] on SySeVR-enabled models.

Acknowledgment

The authors would like to thank the reviewers for their constructive comments, which have guided them in improving the article. They would also like to thank Sujuan Wang and Jialai Wang for collecting the vulnerable programs from NVD and SARD. The authors from Huazhong University of Science and Technology and Hebei University were supported in part by the National Natural Science Foundation of China under Grant U1936211 and Grant 61802106 and in part by the Natural Science Foundation of Hebei Province under Grant F2020201016. The work of Shouhuai Xu was supported in part by ARO under Grant W911NF-17-1-0566 as well as NSF under Grant 1814825 and Grant 1736209. Any opinions, findings, conclusions, or recommendations expressed in this work are those of the authors and do not reflect the views of the funding agencies in any sense.

References

[1] CVE, 2018. [Online]. Available: http://cve.mitre.org/

[2] S. Kim, S. Woo, H. Lee, and H. Oh, “VUDDY: A scalable approach for vulnerable code clone discovery,” in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 595–614.

[3] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, “VulPecker: An automated vulnerability detection system based on code similarity analysis,” in *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, 2016, pp. 201–213.

[4] FlawFinder, 2018. [Online]. Available: http://www.dwheeler.com/flawfinder

[5] Rough Audit Tool for Security, 2014. [Online]. Available: https://code.google.com/archive/p/rough-auditing-tool-for-security/

[6] Checkmarx, 2018. [Online]. Available: https://www.checkmarx.com/

[7] G. Grieco, G. L. Grinblat, L. C. 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*, 2016, pp. 85–96.

[8] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2007, pp. 529–540.

[9] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: Exposing missing checks in source code for vulnerability discovery,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2013, pp. 499–510.

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

[11] Z. Li *et al.*, “VulDeePecker: A deep learning-based system for vulnerability detection,” in *Proc. 25th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2018, pp. 1–15.

[12] J. Guo, J. Cheng, and J. Cleland-Huang, “Semantically enhanced software traceability using deep learning techniques,” in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 3–14.

[13] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, “Deep learning code fragments for code clone detection,” in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 87–98.

[14] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 611–626.

[15] 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.*, 2017, pp. 2539–2541.

[16] J. Li, P. He, J. Zhu, and M. R. Lyu, “Software defect prediction via convolutional neural network,” in *Proc. IEEE Int. Conf. Softw. Qual. Rel. Secur.*, 2017, pp. 318–328.

[17] Q. Geng, Z. Zhou, and X. Cao, “Survey of recent progress in semantic image segmentation with CNNs,” *Sci. China-Inf. Sci.*, vol. 61, no. 5, pp. 051 101:1–051 101:18, 2018.

[18] J. Wang, J. Sun, H. Lin, H. Dong, and S. Zhang, “Convolutional neural networks for expert recommendation in community question answering,” *Sci. China-Inf. Sci.*, vol. 60, no. 11, pp. 110 102:1–110 102:9, 2017.

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

[20] X. Yang, D. Lo, X. Xia, Y. Zhang, and J. Sun, “Deep learning for just-in-time defect prediction,” in *Proc. IEEE Int. Conf. Softw. Qual. Rel. Secur.*, 2015, pp. 17–26.

[21] NVD, 2018. [Online]. Available: https://nvd.nist.gov/

[22] Software assurance reference dataset, 2018. [Online]. Available: https://samate.nist.gov/SRD/index.php

[23] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards real-time object detection with region proposal networks,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017.

[24] A. Shrivastava, A. Gupta, and R. Girshick, “Training region-based object detectors with online hard example mining,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 761–769.

[25] R. E. Noonan, “An algorithm for generating abstract syntax trees,” *Comput. Lang.*, vol. 10, no. 3/4, pp. 225–236, 1985.

[26] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The program dependence graph and its use in optimization,” *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, 1987.

[27] F. Tip, “A survey of program slicing techniques,” *J. Program. Lang.*, vol. 3, no. 3, pp. 121–189, 1995.

[28] M. Abadi *et al.*, “TensorFlow: A system for large-scale machine learning,” in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 265–283.

[29] Common weakness enumeration, 2018. [Online]. Available: http://cwe.mitre.org/

[30] M. Pendleton, R. Garcia-Lebron, J. Cho, and S. Xu, “A survey on systems security metrics,” *ACM Comput. Surv.*, vol. 49, no. 4, pp. 62:1–62:35, 2017.

[31] B. Matthews, “Comparison of the predicted and observed secondary structure of T4 phage lysozyme,” *Biochimica et Biophysica Acta (BBA) - Protein Struct.*, vol. 405, no. 2, pp. 442–451, 1975.

[32] F. Yamaguchi, “Pattern-based vulnerability discovery,” PhD dissertation, Gottingen: Univ. Göttingen, 2015. [Online]. Available: http://ediss.uni-goettingen.de/handle/11858/00-1735-0000-0023-9682-0

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

[34] Word2Vec, 2018. [Online]. Available: http://radimrehurek.com/gensim/models/word2vec.html

[35] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *Proc. 3rd Int. Conf. Learn. Representations*, 2015, pp. 1–15.

[36] G. E. Hinton, “Deep belief networks,” *Scholarpedia*, vol. 4, no. 5, 2009, Art. no. 5947.

[37] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, “Face recognition: A convolutional neural-network approach,” *IEEE Trans. Neural Netw.*, vol. 8, no. 1, pp. 98–113, Jan. 1997.

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

[39] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” in *Proc. 8th Workshop Syntax Semantics Struct. Statist. Transl.*, 2014, pp. 103–111.

[40] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional LSTM and other neural network architectures,” *Neural Netw.*, vol. 18, no. 5/6, pp. 602–610, 2005.

[41] Y. Zhang, R. Jin, and Z. Zhou, “Understanding bag-of-words model: A statistical framework,” *Int. J. Mach. Learn. Cybern.*, vol. 1, no. 1–4, pp. 43–52, 2010.

[42] W. L. Caldas, J. P. P. Gomes, and D. P. P. Mesquita, “Fast co-MLM: An efficient semi-supervised co-training method based on the minimal learning machine,” *New Gener. Comput.*, vol. 36, no. 1, pp. 41–58, 2018.

[43] 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. 56:1–56:36, 2017.

[44] J. Walden, J. Stuckman, and R. Scandariato, “Predicting vulnerable components: Software metrics vs text mining,” in *Proc. 25th IEEE Int. Symp. Softw. Rel. Eng.*, 2014, pp. 23–33.

[45] 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./Dec. 2011.

[46] A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates, "When a patch goes bad: Exploring the properties of vulnerability-contributing commits," in *Proc. ACM / IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2013, pp. 65–74.

[47] N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 projects: Lightweight cross-project anomaly detection," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 119–130.

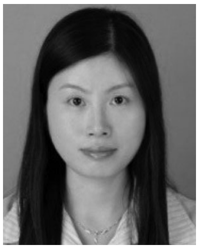
[48] R.-Y. Chang, A. Podgurski, and J. Yang, "Discovering neglected conditions in software by mining dependence graphs," *IEEE Trans. Softw. Eng.*, vol. 34, no. 5, pp. 579–596, Sep./Oct. 2008.

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

[50] 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.*, to be published, doi: 10.1109/TDSC.2019.2942930.

[51] X. Huo, M. Li, and Z. Zhou, "Learning unified features from natural and programming languages for locating buggy source code," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 1606–1612.

[52] M. Allamanis, "The adverse effects of code duplication in machine learning models of code," in *Proc. ACM SIGPLAN Int. Symp. New Ideas New Paradigms Reflections Program. Softw.*, 2019, pp. 143–153.

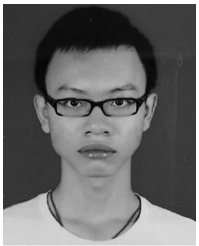


Zhen Li received the BE and ME degrees in computer science and technology from Hebei University, Baoding, China, in 2003 and 2006, respectively, and is currently working toward the PhD degree in cyberspace security at the Huazhong University of Science and Technology, Wuhan, China. Her research interests mainly include vulnerability detection and software security.

Deqing Zou received the PhD degree from the Huazhong University of Science and Technology, Wuhan, China, in 2004. He is currently a professor of cyberspace security with the Huazhong University of Science and Technology, Wuhan, China. His main research interests include system security, trusted computing, virtualization, and cloud security. He has served as a reviewer for several prestigious journals, such as the *IEEE Transactions on Dependable and Secure Computing*, *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, and *IEEE Transactions on Cloud Computing*. He is on the editorial boards of four international journals, and has served as PC chair/PC member of more than 40 international conferences.

Shouhuai Xu (Senior Member, IEEE) received the PhD degree in computer science from Fudan University, Shanghai, China, in 2000. He is the Gallogly chair professor with the Department of Computer Science, University of Colorado Colorado Springs (UCCS). Prior to joining UCCS, he has been with the University of Texas at San Antonio. He pioneered the cybersecurity dynamics approach as foundation for the emerging science of cybersecurity, with three pillars: First-principle cybersecurity modeling and analysis (the x-axis); cybersecurity data analytics (the y-axis, to which the present paper belongs); and cybersecurity metrics (the z-axis). He co-initiated the International Conference on Science of Cyber Security and is serving as its Steering Committee chair. He is/was an associate editor of the *IEEE Transactions on Dependable and Secure Computing* (TDSC), *IEEE Transactions on Information Forensics and Security* (T-IFS), and *IEEE Transactions on Network Science and Engineering* (TNSE).

Hai Jin (Fellow, IEEE) received the PhD degree in computer engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994. He is a Cheung Kung Scholars chair professor of computer science and engineering with HUST in China. He was awarded Excellent Youth Award from the National Science Foundation of China, in 2001. He is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of the CCF and a member of the ACM. He has co-authored 22 books and published more than 700 research papers. His research interests include computer architecture, virtualization technology, cluster computing and cloud computing, peer-to-peer computing, network storage, and network security.



Yawei Zhu received the BE degree in information security from the University of Electronic Science and Technology of China, Chengdu, China, in 2017, and is currently working toward the ME degree in cyberspace security at the Huazhong University of Science and Technology, Wuhan, China. His primary research interests include vulnerability detection and deep learning.

Zhaoxuan Chen received the BE degree in computer science and technology from Guangxi University, Nanning, China, in 2013, and is currently working toward the ME degree in computer science and technology at the Huazhong University of Science and Technology, Wuhan, China. His primary research interest includes vulnerability detection.