

# 软件漏洞分析与防范

## 大作业报告

( 2022 / 2023 学年 第 1 学期 )

题 目：面向源码的漏洞扫描工具使用过程分析

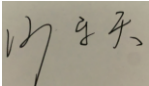
专 业 信 息 安 全

学 生 姓 名 任远哲

班 级 学 号 B190307 班 B19031614

指 导 教 师 沙 乐 天

日 期 2022/12/15

评分内容	具体要求	总分	评分
准备情况	能独立查阅文献资料及从事其他形式的调研，能较好地理解任务并提出实施方案，有分析整理各类信息并从中获取新知识的能力。	10	
功能完成情况	论证、分析、设计、实现、实验正确合理。	30	
报告撰写质量	结构严谨，文字通顺，用语符合技术规范，图表清楚，书写格式规范，符合规定字数要求。	30	
答辩	内容全面，紧扣课题，突出重点，语言简练准确，演示过程顺利。	30	
总评分			
简短评语	<p>任远哲同学在这次大作业中通过查阅文献，了解了目前主流的几种源代码漏洞检测技术，并针对当下新兴的基于深度学习的漏洞检测模型 <i>Automated Vulnerability Detection in Source Code Using Deep Representation Learning</i> 开展研究，在数据集 Draper VDISC Dataset 上复现了实验结果。</p> <p>该模型使用卷积神经网络学习的表征和随机森林算法解决漏洞检测的准确性问题。基本思路是将源代码进行源词法分析从而提取到有用的表征。然后删除源代码的词法重复表示，最后将源代码的神经表征表示与强大的集成分类器随机森林（RF）相结合。实验表明该模型在测试集上有着超过百分之 90 的准确率，因此是被证明有效的。</p> <p>该同学准备充分，功能完成情况优秀，报告内容充实、格式较规范；答辩时思路清晰，演示熟练。</p> <p>综上所述，大作业成绩评定为 xx 分。</p> <p style="text-align: right;">教师签名: </p> <p style="text-align: right;"><u>2022</u> 年 <u>1</u> 月 <u>18</u> 日</p>		
a.讨论课成绩(30%)		b.大作业成绩（70%）	期末成绩（百分制）

## 一. 背景介绍

计算机科学技术的发展在为人类带来便利的同时，也给恶意分子提供了犯罪的工具。尤其是在网络空间安全领域，黑客攻击、数字资产盗窃、用户隐私信息泄露等各种网络安全事件屡屡出现，严重危害了网络的进一步发展和用户对其的信任度。软件系统作为网络空间的核心组件，其本身存在的漏洞是导致这些攻击的根本原因。软件漏洞是指软件在其生命周期（即开发、部署、执行整个过程）中存在的缺陷，而这些缺陷可能会被不法分子利用，绕过系统的访问控制，非法窃取较高的权限从而任意操纵系统，如触发特权命令、访问敏感信息、冒充身份、监听系统运行等。随着现有软件系统愈加复杂庞大，漏洞出现频率不断提升，急需针对漏洞检测领域开展系统的研究工作，以便高效及时地发现软件系统的漏洞，实时修补，提高网络空间的安全等级。

依据分析对象，软件漏洞静态检测可以分为二进制漏洞检测和源代码漏洞检测两类。二进制漏洞检测方法通过直接分析二进制代码检测漏洞，漏洞检测的准确度较高，实用性广泛；但缺乏上层的代码结构信息和类型信息，分析难度较大，因此基于二进制代码的漏洞检测研究工作相对较少。而源代码相对于编译后的二进制代码拥有更丰富的语义信息，因此更利于快捷地找出漏洞，得到了漏洞检测研究人员的广泛关注。

## 二. 相关的研究

虽然存在用于程序分析的现有工具，但这些工具通常仅检测基于预定义规则的可能错误的有限子集。随着最近开放源码库的广泛使用，使用数据驱动的技术来发现漏洞模式已经成为可能。

目前存在各种各样的分析工具，它们试图发现软件中的常见漏洞。静态分析器，如 Clang, Trivy, OpenVAS，无需执行程序即可完成此操作。动态分析器在真实或虚拟处理器上重复执行具有许多测试输入的程序，以识别弱点。静态和动态分析器都是基于规则的工具，因此仅限于它们手工设计的规则，不能保证完全覆盖代码库。基于逻辑推理的漏洞检测方法将源代码进行形式化描述，然后利用数学推理、证明等方法验证形式化描述的一些性质，从而判断程序是否含有某种类型的漏洞。基于

逻辑推理的漏洞检测方法由于以数学推理为基础，因此分析严格，结果可靠。但对于较大规模的程序，将代码进行形式化表示本身是一件非常困难的事情。

除了这些传统的工具外，最近在使用机器学习进行程序分析方面也进行了大量的工作。大量开源代码的可获得性为直接从挖掘的数据中了解软件漏洞的模式提供了机会。

基于传统机器学习的漏洞检测方法依赖于专家手工定义特征属性，采用机器学习模型自动对漏洞代码和无漏洞代码进行分类。但由于输入机器学习模型的代码粒度通常较粗，无法确定漏洞行的确切位置。

基于深度学习的漏洞检测方法不需要专家手工定义特征，可以自动生成漏洞模式，有望改变软件源代码漏洞检测方法，使面向各种类型漏洞的漏洞模式从依赖专家手工定义向自动生成转变，并且显著提高漏洞检测的有效性。然而目前该方法的相关研究刚刚起步，在漏洞定位、数据集构建、深度学习模型解释等方面有待深入研究。

在本次大作业中，我主要研究了当下新兴的基于深度学习的源代码漏洞检测方法：*Automated Vulnerability Detection in Source Code Using Deep Representation Learning*。它利用大量 C/C++ 开源代码，对源代码进行词法分析后应用机器学习的方法，开发了一个大规模漏洞检测系统。

## 三. 算法原理

### 3.1 数据预处理

#### 3.1.1 源代码的词法分析

为了从每个函数的原始代码中生成有用的特征，可以设计了一个 C/C++ 词法分析器，旨在捕捉关键的单词的含义，同时要保证表示形式的通用性并最小化总体上单词数量。将不同来源的代码词法分析为标准化的表示形式，这样可以使得我们在整个数据集上做迁移学习，同样也可以防止过拟合。

通过词法分析器将表示 C/C++ 代码的总体单词数量降到 156 个单词(包括所有基础的 C/C++ 关键字，操作符，分隔符等等)。删除了不影响编译的代码(例如注释)。

字符串、字符、浮点数与所有标识符一样，都被词法分析为特定类型的占位符。整数是逐位标记的，因为这些值通常与漏洞相关。来自常见库的函数调用（可能与漏洞相关）都被映射到了通用的版本。例如，u32, uint32\_t, UINT32, uint32 和 DWORD 都被词法分析为相同的单词来表示 32 位无符号数。

### 3.1.2 去重

对于神经网络的训练，其中非常重要的一步就是删除重复数据，因为其对于训练毫无帮助并且影响训练速度，所以要删除训练集中潜在的重复函数。对于如何删除这些重复数据，也有相应的方法：删除任何源代码的词法重复表示或编译级特征向量重复的函数。

## 3.2 模型结构-用于源代码漏洞检测的表征学习方法

输入源代码被转换成一个可变长度的令牌序列（设长度为  $l$ ），嵌入到一个  $1 \times k$  表征中。通过  $n$  个大小为  $m \times k$  的卷积进行滤波，并沿着序列长度最大化为固定大小的特征向量。嵌入和卷积滤波器通过来自完全连接的分类层的加权交叉熵损失来学习。学习的  $n$  维特征向量被用作随机森林分类器的输入，与单独的神经网络分类器相比，这提高了性能。

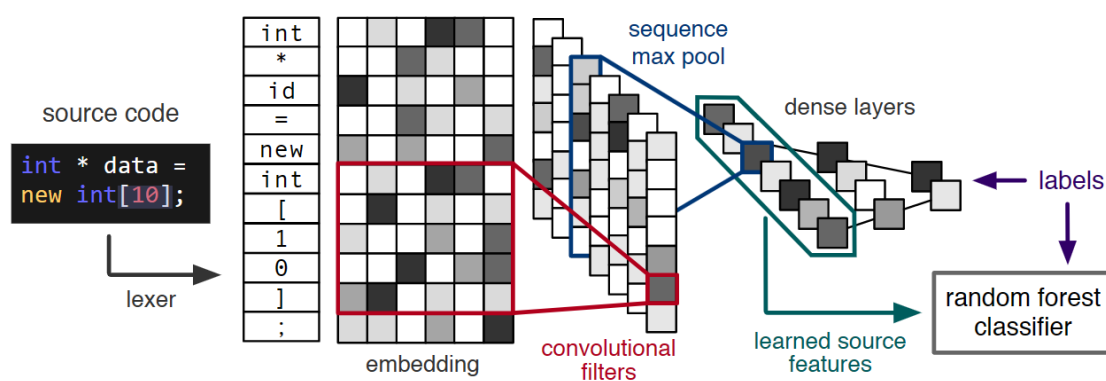


图 1：模型结构

### 3.2.1 Embedding:

将词法分析后的函数 embedding 成维度为  $k$ ，数值在  $[-1, 1]$  之间的向量， $(1 \times k)$

可以采用 word2vec 等方法初始化 embedding 的值(随机初始化结果相差不大),之后通过训练更新 embedding 的值。由于单词量很小,所以我们向量维度也要小一些,经实验发现  $K=13$  时效果最佳,既包含足够信息,又不会过拟合。添加少量的随机高斯噪声会减轻过拟合现象,且相比其他正则化技术更有效

### 3.2.2 特征提取

可以采用两种特征提取方法:

卷积特征提取:使用  $n$  个形状为  $m \times k$  的卷积核, $m$  决定了多少个单词将会被放在一起考虑。经实验得实验最佳结果为:  $m=9, n=512$ ,且需增加 BN 和 RELU 层。

循环特征提取:使用循环神经网络捕获远距离单词之间的依赖特征,将 embedding 输入到双层 GRU/RNN 网络中,隐层维度  $n=256$ ,将每一步得到的长度为 1 的序列 concatenate 到一起。

### 3.2.3 池化

由于源代码中不同函数的长度不同,使用 max\_pooling 来进行池化操作,生成固定长度的表示。

### 3.2.4 Dense 层

经过卷积层(或 RNN)、池化层进行特征提取后,经过一个全连接层(包含两个隐层,分别有 64 和 16 个神经元,输入层取 50%的 dropout),最后进过 softmax 层得到输出结果。

### 3.2.5 训练

为了方便批处理,我们只训练长度在 10-500 个单词之间的函数, batch 取 128, 优化器选用 Adam, loss 使用交叉熵。

根据最高的有效马修斯相关系数(MCC)调整 and 选择模型。MCC 是一个描述实际分类与预测分类之间的相关系数,它的取值范围为  $\{-1, 0, 1\}$ 。取值为 1 时表示对

被测试对象的完美预测。取值为 0 时表示预测的结果不如随机预测的结果。取值为 -1 时表示预测分类和实际分类完全不一致。

因为 MCC 同时考虑了真阳性、真阴性和假阳性和假阴性，所以通常认为该指标是一个比较均衡的指标，即使是在两类别的样本含量差别很大时，也可以应用它。基于 MCC 的这个特性，所以非常适合于测评本文所提出的模型。MCC 计算公式如下所示

$$MCC = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

### 3.2.6 集成学习

将前序神经网络的输入输出作为 RF (等强大的集成分类器) 的输入，有助于防止过拟合，可以产生更好的效果。

## 四. 实验过程

本次实验借助 Colab 平台完成，选用数据集 Draper VDISC Dataset - Vulnerability Detection in Source Code。该数据集包含从开源软件中挖掘的 127 万个函数的源代码，并通过静态分析标记潜在漏洞。有关数据集和基准测试结果的详细信息可看 <https://arxiv.org/abs/1807.04320>

数据集分为三个 HDF5 文件，对应 80:10:10 训练/验证/测试划分，数据集中每个函数的原始源代码（从函数名开始）存储为可变长度的 UTF-8 字符串。为每个函数提供了五个二进制“漏洞”标签，对应于我们数据中最常见的四个 CWE 以及所有其他 CWE。

CWE-120 (3.7% of functions)

CWE-119 (1.9% of functions)

CWE-469 (0.95% of functions)

CWE-476 (0.21% of functions)

CWE-other (2.7% of functions)

4.1 下载数据集

```
[ ] from google.colab import drive
    drive.mount('/content/drive')

[ ] !wget https://files.osf.io/v1/resources/d45bw/providers/osfstorage/?zip= --output-document osfstorage-archive.zip

[ ] !unzip 'osfstorage-archive.zip'

Archive: osfstorage-archive.zip
  inflating: VDISC_validate.hdf5
  inflating: VDISC_test.hdf5
  inflating: VDISC_train.hdf5
  inflating: README
```

图 2：代码展示

4.2 载入数据集

```
data = h5py.File("VDISC_train.hdf5", 'r')
data2 = h5py.File("VDISC_validate.hdf5", 'r')
data3 = h5py.File("VDISC_test.hdf5", 'r')

[ ] mydf = pd.DataFrame(list(data['functionSource']))
    mydf2 = pd.DataFrame(list(data2['functionSource']))
    mydf3 = pd.DataFrame(list(data3['functionSource']))
```

图 3：代码展示

4.3 数据集中源代码内容如下（节选）

0	b'clear_area(int startx, int starty, int xsize...
1	b'ReconstructDuList(Statement* head)\n{\n S...
2	b'free_speaker(void)\n{\n if(Lenghts)\n ...
3	b'mlx4_register_device(struct mlx4_dev *dev)\n...
4	b'Parse_Env_Var(void)\n{\n char *p = getenv("...

图 4：数据集展示



#### 4.4 定义超参数

```
myrand=71926
np.random.seed(myrand)
tf.random.set_seed(myrand)
print("Random seed is:",myrand)
```

Random seed is: 71926

```
[ ] # Set the global value
WORDS_SIZE=10000
INPUT_SIZE=500
NUM_CLASSES=2
MODEL_NUM=0
EPOCHS=10
```

图 5: 代码展示

#### 4.5 给数据集中的源代码分词（词法分析）

```
[ ] # Tokenizer with word-level
tokenizer = tf.keras.preprocessing.text.Tokenizer(char_level=False)
#[x.decode('utf-8') for x in list(x_all)]
tokenizer.fit_on_texts([x.decode('utf-8') for x in list(x_all)])
del(x_all)
print('Number of tokens: ',len(tokenizer.word_counts))
```

Number of tokens: 1094129

```
[ ] # Reducing to top N words
tokenizer.num_words = WORDS_SIZE
```

图 6: 代码展示

#### 4.6 分词结果：出现频率最高的 10 个单词

```
sorted(tokenizer.word_counts.items(), key=lambda x:x[1], reverse=True)[0:10]
```

```
↳ [('if', 3126441),
    ('0', 2106459),
    ('return', 1745333),
    ('i', 1375259),
    ('1', 1186857),
    ('int', 1016932),
    ('null', 975347),
    ('the', 791897),
    ('t', 733766),
    ('n', 716010)]
```

图 7: 数据集分析内容

#### 4.7 将训练集中的源代码转化成 token 序列

```
list_tokenized_train = tokenizer.texts_to_sequences([x.decode('utf-8') for x in train['functionSource']])
x_train = tf.keras.preprocessing.sequence.pad_sequences(list_tokenized_train,
                                                         maxlen=INPUT_SIZE,
                                                         padding='post')

x_train = x_train.astype(np.int64)
```

图 8：代码展示

#### 4.8 将训练集中的标签转化成 one-hot 向量

```
y_train=[]
y_test=[]
y_validate=[]

for col in range(1,6):
    y_train.append(tf.keras.utils.to_categorical(train.iloc[:,col], num_classes=NUM_CLASSES).astype(np.int64))
    #y_test.append(tf.keras.utils.to_categorical(test.iloc[:,col], num_classes=NUM_CLASSES).astype(np.int64))
    #y_validate.append(tf.keras.utils.to_categorical(validate.iloc[:,col], num_classes=NUM_CLASSES).astype(np.int64))

y_validate=y_train
```

图 9：代码展示

#### 4.9 定义模型

```
inp_layer = tf.keras.layers.Input(shape=(INPUT_SIZE,))
mid_layers = tf.keras.layers.Embedding(input_dim = WORDS_SIZE,
                                       output_dim = 13,
                                       weights=[random_weights],
                                       input_length = INPUT_SIZE)(inp_layer)

mid_layers = tf.keras.layers.Convolution1D(filters=512, kernel_size=(9), padding='same', activation='relu')(mid_layers)
mid_layers = tf.keras.layers.MaxPool1D(pool_size=5)(mid_layers)
mid_layers = tf.keras.layers.Dropout(0.5)(mid_layers)
mid_layers = tf.keras.layers.Flatten()(mid_layers)
mid_layers = tf.keras.layers.Dense(64, activation='relu')(mid_layers)
mid_layers = tf.keras.layers.Dense(16, activation='relu')(mid_layers)

output1 = tf.keras.layers.Dense(2, activation='softmax')(mid_layers)
output2 = tf.keras.layers.Dense(2, activation='softmax')(mid_layers)
output3 = tf.keras.layers.Dense(2, activation='softmax')(mid_layers)
output4 = tf.keras.layers.Dense(2, activation='softmax')(mid_layers)
output5 = tf.keras.layers.Dense(2, activation='softmax')(mid_layers)
model = tf.keras.Model(inp_layer, [output1, output2, output3, output4, output5])

# Define custom optimizers
adam = tf.keras.optimizers.Adam(lr=0.005, beta_1=0.9, beta_2=0.999, epsilon=1, decay=0.0, amsgrad=False)

## Compile model with metrics
model.compile(optimizer=adam, loss='categorical_crossentropy', metrics=['accuracy'])
print("CNN model built: ")
model.summary()
```

图 10：代码展示

#### 4.10 训练模型

```
class_weights = [{0: 1., 1: 5.}, {0: 1., 1: 5.}, {0: 1., 1: 5.}, {0: 1., 1: 5.}, {0: 1., 1: 5.}]

history = model.fit(x = x_train,
                    y = [y_train[0], y_train[1], y_train[2], y_train[3], y_train[4]],
                    validation_data = (x_validate, [y_validate[0], y_validate[1], y_validate[2], y_validate[3], y_validate[4]]),
                    epochs = 40,
                    batch_size = 128,
                    verbose =2,
                    class_weight= class_weights,
                    callbacks=[mcp, tbCallback])

with open('history/History-ALL-40EP-CNN', 'wb') as file_pi:
    pickle.dump(history.history, file_pi)
```

图 11：代码展示

```

Train on 1019471 samples, validate on 127476 samples
Epoch 1/40

Epoch 00001: val_loss improved from inf to 1.22285, saving model to model/model-ALL-01.hdf5
1019471/1019471 - 207s - loss: 1.4301 - dense_17_loss: 0.2974 - dense_18_loss: 0.4755 - dense_19_loss: 0.0630 - dense_20_loss: 0.0630
Epoch 2/40

Epoch 00002: val_loss improved from 1.22285 to 1.13063, saving model to model/model-ALL-02.hdf5
1019471/1019471 - 203s - loss: 1.1590 - dense_17_loss: 0.2164 - dense_18_loss: 0.3676 - dense_19_loss: 0.0443 - dense_20_loss: 0.0443
Epoch 3/40

```

图 12: 训练过程

#### 4.11 在测试集上验证

```

[[121949  3018]
 [   779 1673]]

TP: 1673
FP: 3018
TN: 121949
FN: 779

Accuracy: 0.9702006765082131
Precision: 0.3566403751865274
Recall: 0.682300163132137
F-measure: 0.46843063138737223
Precision-Recall AUC: 0.3864869549314679
AUC: 0.9452323096821215
MCC: 0.4801512959877963

```

图 13: 评估结果

#### 4.12 每一轮训练中准确率的变化趋势

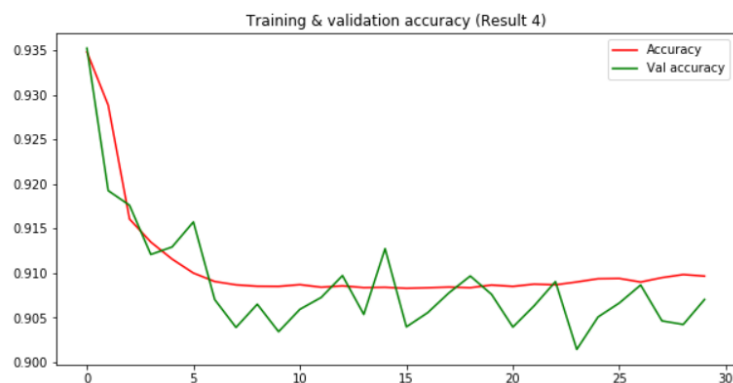


图 14: 训练过程

## 五. 实验结果分析

总体而言，无论是作为独立的分类器还是特征生成器，CNN 模型都比 RNN 模型表现得更好。此外，CNN 的训练速度更快，需要的参数要少得多。在自然函数数据

集上，对于 CNN 和 RNN 特征，基于神经特征表示训练的 RF 分类器都比独立的网络表现得更好。同样，基于神经网络表示训练的 RF 分类器比基准弓形分类器表现得更好。

与传统的静态分析工具相比，基于 ML（机器学习）的方法具有一些额外的优势。ML 模型可以快速消化和评分大型存储库和源代码，而无需编译代码。此外，由于 ML 方法都是输出概率，因此可以调整阈值以达到所需的精度和召回率。另一方面，静态分析器会返回固定数量的发现，这些发现对于庞大的代码库来说可能非常大，对于关键应用程序来说可能太小。

虽然静态分析器能够更好地定位他们发现的漏洞，但我们可以使用可视化技术，如下图所示的特征激活图，帮助理解我们的算法做出决策的原因。

```
wchar_t * data;
unionType myUnion;
data = new wchar_t[100];
wmemset(data, L'A', 100-1);
data[100-1] = L'\0';
myUnion.unionFirst = data;
{
    wchar_t * data = myUnion.unionSecond;
    {
        wchar_t dest[50] = L"";
        memcpy(dest, data, wcslen(data)*sizeof(wchar_t));
        dest[50-1] = L'\0';
        printWLine(data);
        delete [] data;
    }
}
```

图 15：特征激活图

## 六. 总结

通过长达接近 20 天的自主学习，查阅资料，我较为完整地完成了课题所确定的目标。在以上这篇文章中体现了近来的学习成果。

首先通过查阅文献，我了解到目前的源代码漏洞检测技术主要分为基于静态程序分析技术，基于中间表示的漏洞检测技术和基于逻辑推理的漏洞检测技术。

基于静态程序分析技术检测源代码中的漏洞，具有代码覆盖率高、漏报低的优点，但对已知漏洞的依赖性较大，误报较高；基于中间表示的漏洞检测方法首先将源代码转换为有利于漏洞检测的中间表示，然后对中间表示进行分析，检查是否匹配预定义的某个漏洞规则，从而判断源程序中是否含有对应漏洞规则相关的漏洞。

依据对中间表示的分析技术，漏洞检测方法可以分为 4 类：基于代码相似性的漏洞检测、基于符号执行的漏洞检测、基于规则的漏洞检测以及基于机器学习的漏洞检测。其中，第一类方法主要针对由于代码复制导致的相同漏洞进行检测；后三类方法基于漏洞模式，针对各种原因导致的漏洞进行检测；基于逻辑推理的漏洞检测方法将源代码进行形式化描述，然后利用数学推理、证明等方法验证形式化描述的一些性质，从而判断程序是否含有某种类型的漏洞。基于逻辑推理的漏洞检测方法由于以数学推理为基础，因此分析严格，结果可靠。但对于较大规模的程序，将代码进行形式化表示本身是一件非常困难的事情。基于中间表示的漏洞检测方法没有上述局限性，适用于分析较大规模程序，因此得到了更为广泛的应用。

而在本次大作业中，我主要研究了 *Automated Vulnerability Detection in Source Code Using Deep Representation Learning*。这属于对中间表示的分析技术中基于机器学习的漏洞检测方法。我在谷歌提供的云平台 Colab 上部署了实验，首先下载好数据集 Draper VDISC Dataset 并将其载入内存，接着使用 tokenizer 分词器对数据集中的源代码进行词法分析，并将对应的 label 转化成 one-hot 向量形式，最后定义模型，训练，并在测试集上进行评估。实验表明该模型在测试集上有着超过百分之九十的准确率，因此是被证明有效的。

通过本次实验，我加强了对于安全编程技术的理解，并思考了信息安全问题与当下流行的人工智能技术的结合性。虽然软件漏洞检测只是信息安全中的一小部分。但它的发展历程却能以小博大地反映信息安全的其他领域如恶意代码审计，日志检测的发展过程，并指出信息安全发展的新方向：与机器学习结合。信息安全领域如此，其他学科领域亦如此。机器学习的智能算法，尤其是深度学习，在各行各业都开辟了新天地。我们要善于把握时代趋势，主动学习一些新技术，而不能固步自封，抱残守缺。未来的时代学科与学科之间的壁垒会越来越小，我们学生需要培养的是知识交叉的能力，即用别的专业的知识来改进自己专业的东西。