NDSS Symposium 2018

VulDeePecker : A Deep Learning-Based System for Vulnerability Detection

Zhen Li[1], Deqing Zou[1], Shouhuai Xu[2], Xinyu Ou[1], Hai Jin[1], Sujuan Wang[1], Zhijun Deng[1] , Yuyi Zhong[1]

[1]Huazhong University of Science and Technology (HUST), Wuhan, China

[2]University of Texas at San Antonio (UTSA), San Antonio, USA

Automatic Software Vulnerability Detection

- Automatic detection of software vulnerabilities is an important research problem
- Static vulnerability detection tools and studies

Flawfinder	RATS	CHECKMARX	
COVERITY	FORTIFY	QoBOT	
ReDeBug	VUDDY (SP'17)	VulDeePecker (ACSAC'16)	…

Drawbacks of Existing Approaches

- First, imposing intense labor of **human experts**
  - ✓ Define features
- Second, incurring **high false negative rates**
  - ✓ Two most recent vulnerability detection systems
    - VUDDY (SP’17): false negative rate = 18.2% for Apache HTTPD 2.4.23
    - VulPecker (ACSAC’16): false negative rate = 38% with respect to 455 vulnerability samples

Research Problem

- ✧ Given the source code of a target program, how can we determine whether or not the target program is vulnerable and if so, where are the vulnerabilities?

Without asking human experts to manually define features

Without incurring a high false negative rate or false positive rate

Our Main Contribution

Vulnerability Deep Pecker **(VulDeePecker):**

A deep learning-based system for automatically detecting vulnerabilities in programs (source code)

Outline

- ✧ Guiding Principles
- ✧ Design of VulDeePecker
- ✧ Experiments and Results
- ✧ Limitations
- ✧ Conclusion

Outline

- ✧ **Guiding Principles**
- ✧ Design of VulDeePecker
- ✧ Experiments and Results
- ✧ Limitations
- ✧ Conclusion

Guiding Principles: three questions

Q1: How to represent software programs for deep learning-based vulnerability detection?

Q2: What is the appropriate granularity for deep learning-based vulnerability detection?

Q3: How to select a specific neural network for vulnerability detection?

Guiding Principles

Q1: How to represent software programs for deep learning-based vulnerability detection?

Preserve the semantic relationships between the programs' elements (e.g., data-flow and control-flow information).

Guiding Principles

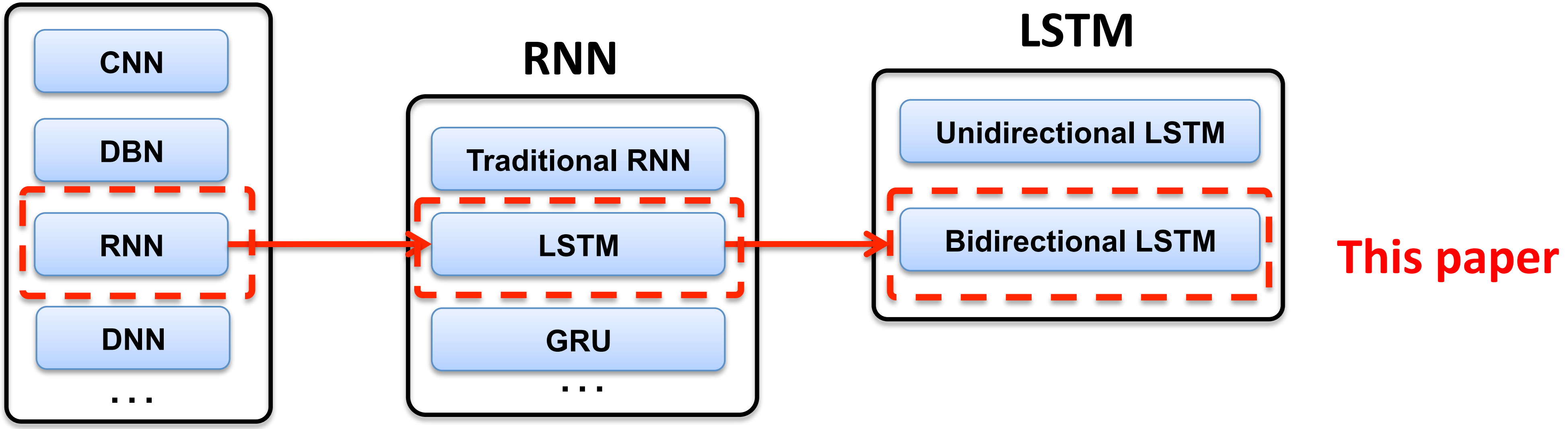
Q2: What is the appropriate granularity for deep learning-based vulnerability detection?

Represented at a finer granularity than treating a program or a function as a unit.

Guiding Principles

Q3: How to select a specific neural network for vulnerability detection?

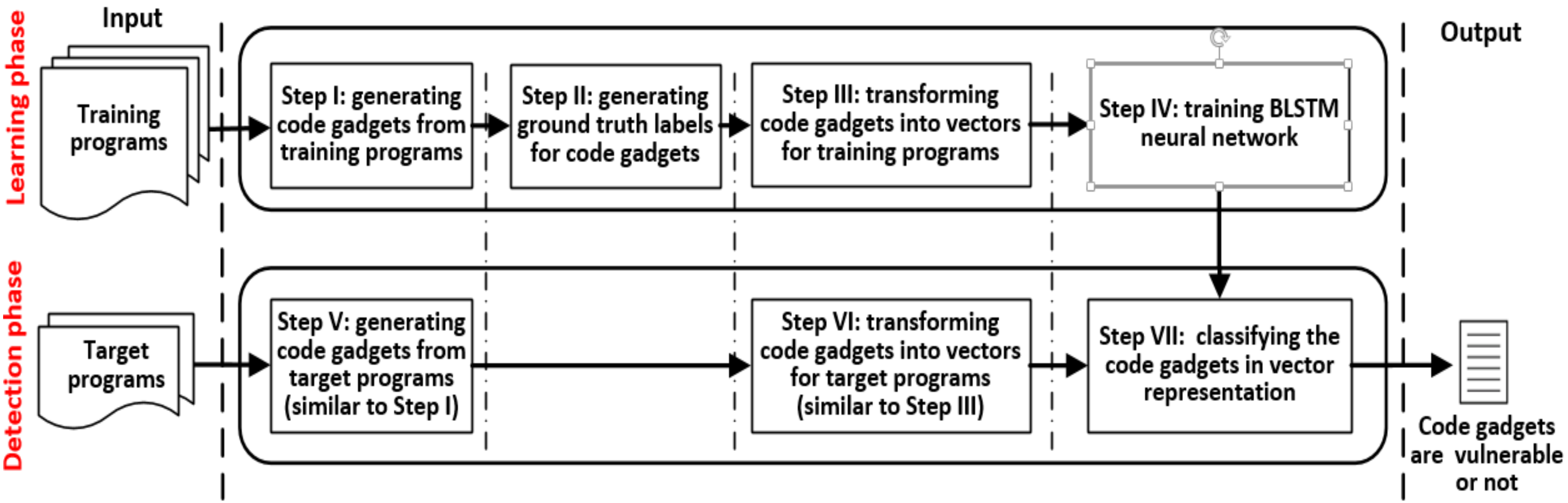
Neural networks that can cope with contexts may be suitable for vulnerability detection.



Outline

- ✧ Guiding Principles
- ✧ **Design of VulDeePecker**
- ✧ Experiments and Results
- ✧ Limitations
- ✧ Conclusion

Overview of VulDeePecker



The Concept of Code Gadget

- ✧ A unit for vulnerability detection
- ✧ A number of program statements that are semantically related to each other in terms of data dependency or control dependency
- ✧ Example: vulnerabilities related to library/API function calls

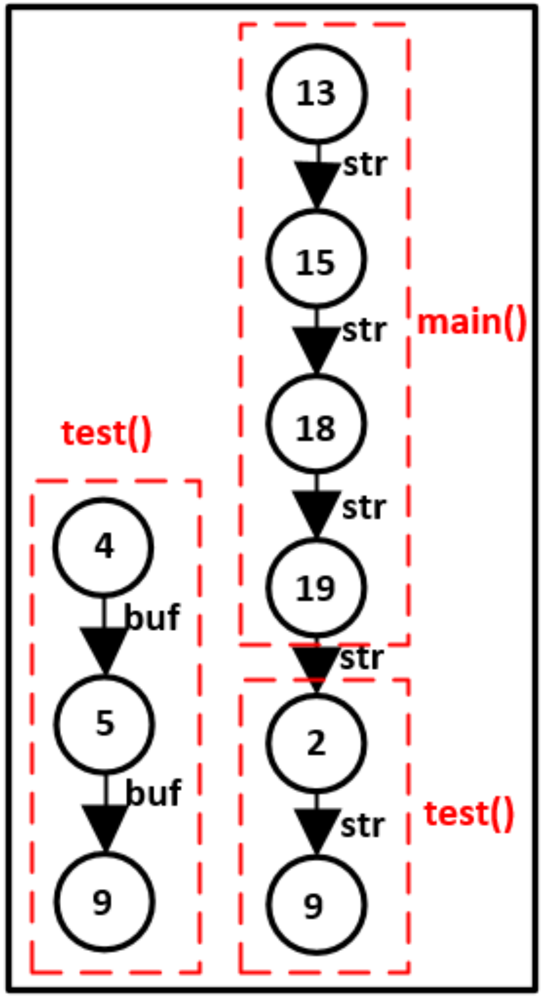
Step I: Generating Code Gadgets

```
1   void
2   test(char *str)
3   {
4     int MAXSIZE=40;
5     char buf[MAXSIZE];
6
7     if(!buf)
8         return;
9     strcpy(buf, str); /*string copy*/
10  }
11
12  int
13  main(int argc, char **argv)
14  {
15    char *userstr;
16
17    if(argc > 1) {
18        userstr = argv[1];
19        test(userstr);
20    }
21    return 0;
22  }
```

Program source code

```
9  strcpy(buf, str)
```

(1) Extract library/API function calls



(2) Generate slices for arguments of library/API function calls

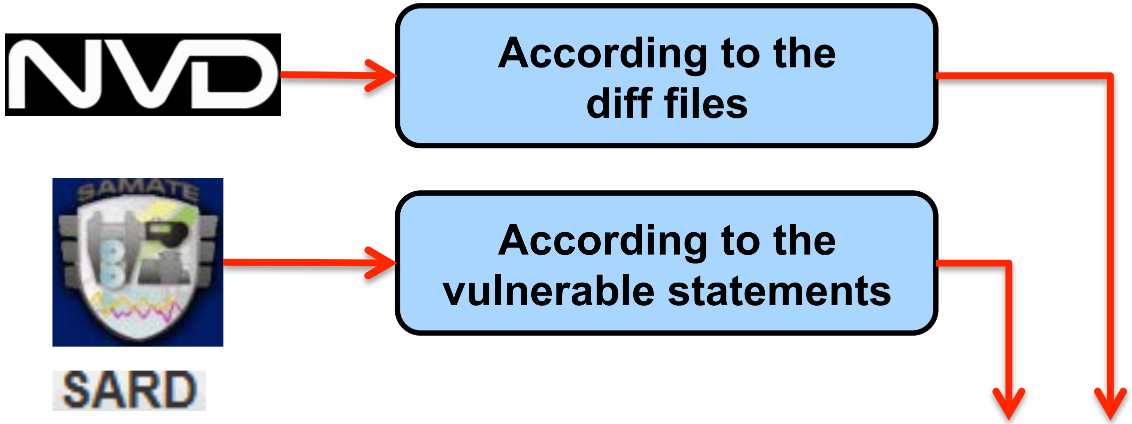
A code gadget corresponding to strcpy()

```
13  main(int argc, char **argv)
15  char *userstr;
18  userstr = argv[1];                 main()
19  test(userstr);
2   test(char *str)
4   int MAXSIZE=40;                    test()
5   char buf[MAXSIZE];
9   strcpy(buf, str); /*string copy*/
```

(3) Assemble slices into code gadgets

Step II: Generating Ground Truth Labels

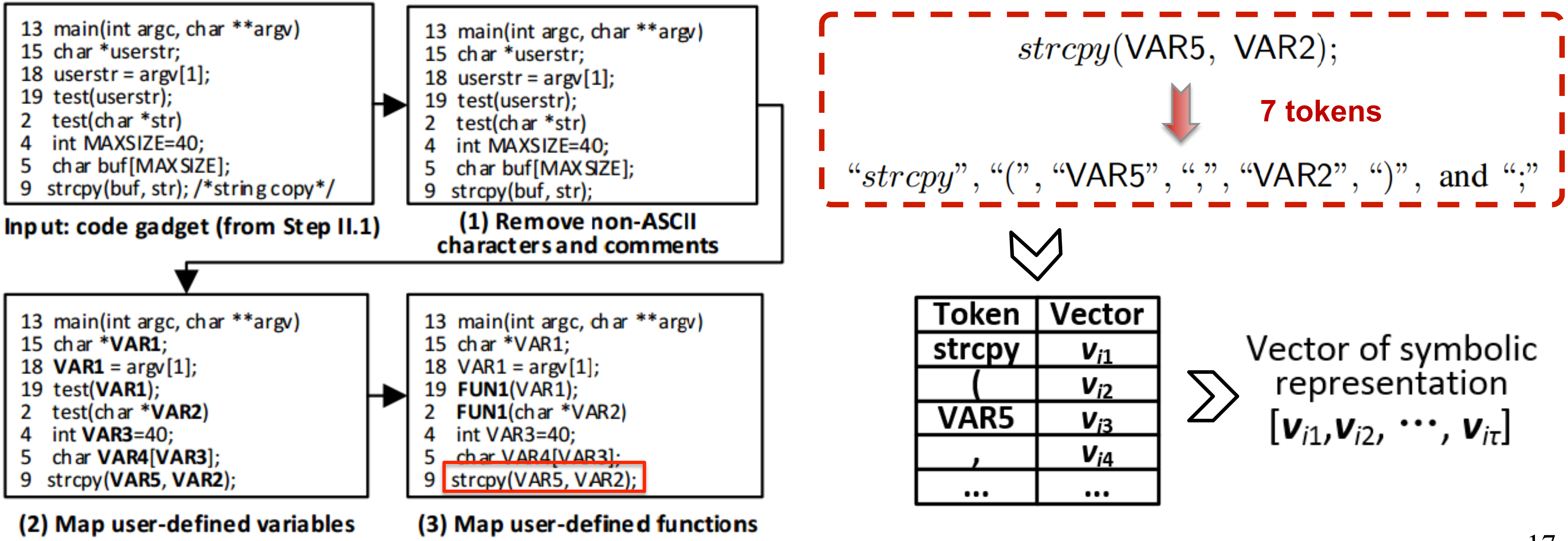
✧ **Each code gadget is labeled as "1" (i.e., vulnerable) or "0" (i.e., not vulnerable).**



Code gadget	Label
Code gadget 1	1
Code gadget 2	0
Code gadget 3	1
Code gadget 4	0
Code gadget 5	0
...	...

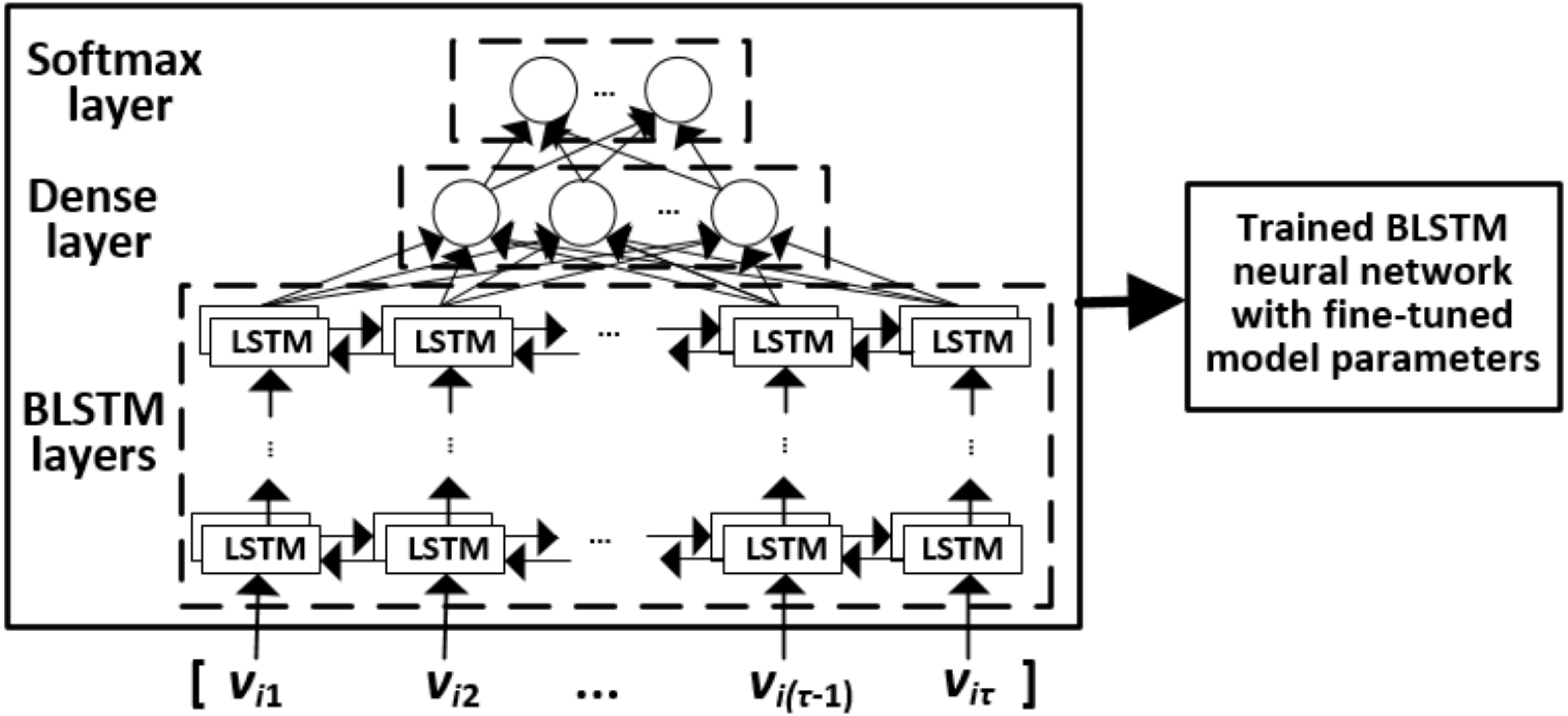
Step III: Transforming Code Gadgets into Vectors

- ✧ Transform code gadgets into their symbolic representations
- ✧ Encode the symbolic representations into vectors

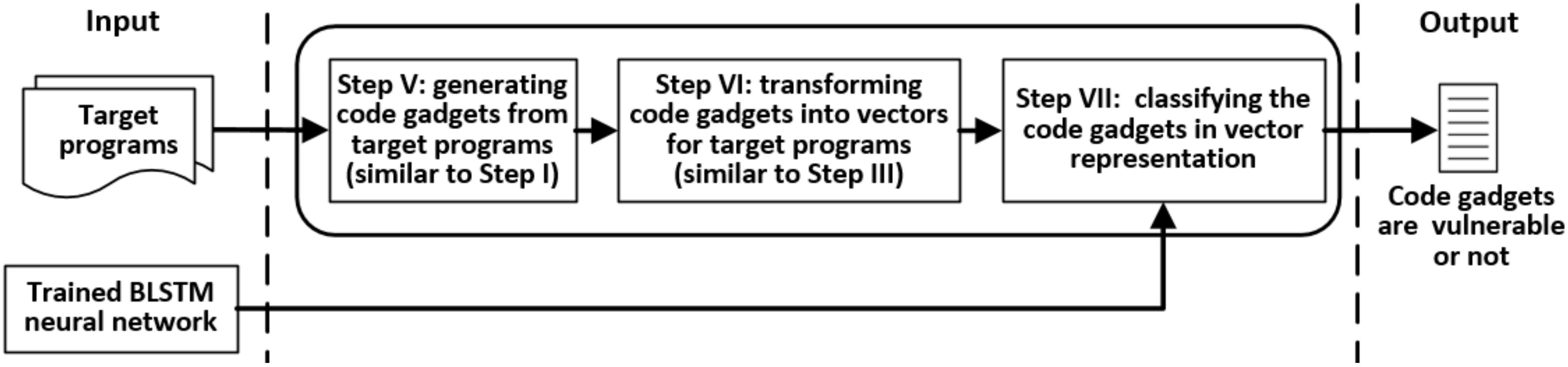


Step IV: Training the BLSTM Neural Network

- Training process for learning the BLSTM neural network is standard



Steps V-VII: Detection Phase



Outline

- ✧ Guiding Principles
- ✧ Design of VulDeePecker
- ✧ **Experiments and Results**
- ✧ Limitations
- ✧ Conclusion

Research Questions

RQ1: Can VulDeePecker deal with multiple types of vulnerabilities at the same time?

RQ2: Can human intelligence (other than defining features) improve the effectiveness of VulDeePecker?

RQ3: How effective is VulDeePecker when compared with other approaches?

- Metrics for evaluation
  - ✓ False positive rate (FPR), false negative rate (FNR), recall, precision, F-measure

Preparing Input to VulDeePecker

- ✧ Programs collection for answering the RQs
  - ✓ Two sources of vulnerability data
    - 19 C/C++ open source products which vulnerabilities are described in NVD, and C/C++ test cases in SARD
  - ✓ Collect 520 open source software program files and 8,122 test cases for the buffer error vulnerability (i.e., CWE-119) , and 320 open source software program files and 1,729 test cases for the resource management error vulnerability (i.e., CWE-399)
- ✧ Training programs vs. target programs
  - ✓ Randomly choose 80% of the programs we collect as training programs and the rest 20% as target programs

Learning BLSTM Neural Networks

✧ Datasets for answering the RQs

✓ Code Gadget Database (CGD): 61,638 code gadgets

✓ Six datasets of CGD

Dataset	#Code gadgets	#Vulnerable code gadgets	#Not vulnerable code gadgets
BE-ALL	39,753	10,440	29,313
RM-ALL	21,885	7,285	14,600
HY-ALL	61,638	17,725	43,913
BE-SEL	26,720	8,119	18,601
RM-SEL	16,198	6,573	9,625
HY-SEL	42,918	14,692	28,226

Table I. DATASETS FOR ANSWERING THE RQS

BE: Buffer error vulnerabilities
RM: Resource management vulnerabilities
HY: Hybrid of the above two types of vulnerabilities

ALL: All library/API function calls
SEL: Manually selected library/API function calls

RQ1

RQ1: Can VulDeePecker deal with multiple types of vulnerabilities at the same time?

✧ **Insight**: VulDeePecker can detect multiple types of vulnerabilities, but the effectiveness is sensitive to the amount of data (which is common to deep learning).

Dataset	FPR(%)	FNR(%)	TPR(%)	P(%)	F1(%)
BE-ALL	2.9	18.0	82.0	91.7	86.6
RM-ALL	2.8	4.7	95.3	94.6	95.0
HY-ALL	5.1	16.1	83.9	86.9	85.4

RM: 16 function calls related to vulnerabilities

BE: 124 function calls related to vulnerabilities

RQ2

RQ2: Can human intelligence (other than defining features) improve the effectiveness of VulDeePecker?

- **Insight**: Human expertise can be used to select function calls to improve the effectiveness of VulDeePecker.

Dataset	FPR(%)	FNR(%)	TPR(%)	P(%)	F1(%)
HY-ALL	5.1	16.1	83.9	86.9	85.4
HY-SEL	4.9	6.1	93.9	91.9	92.9

RQ3: VulDeePecker vs. Static Analysis Tools

RQ3: How effective is VulDeePecker when compared with other approaches?

✧ **Insight**: A deep learning-based vulnerability detection system can be more effective by taking advantage of the data-flow information.

System	Dataset	FPR (%)	FNR (%)	TPR (%)	P (%)	F1 (%)
VulDeePecker vs. Other pattern-based vulnerability detection systems						
Flawfinder	BE-SEL	44.7	69.0	31.0	25.0	27.7
RATS	BE-SEL	42.2	78.9	21.1	19.4	20.2
Checkmarx	BE-SEL	43.1	41.1	58.9	39.6	47.3
VulDeePecker	BE-SEL	**5.7**	**7.0**	93.0	88.1	90.5
VulDeePecker vs. Code similarity-based vulnerability detection systems						
VUDDY	BE-SEL-NVD	0	95.1	4.9	100	9.3
VulPecker	BE-SEL-NVD	1.9	89.8	10.2	84.3	18.2
VulDeePecker	BE-SEL-NVD	**22.9**	**16.9**	83.1	78.6	80.8
VUDDY	BE-SEL-SARD	N/C	N/C	N/C	N/C	N/C
VulPecker	BE-SEL-SARD	N/C	N/C	N/C	N/C	N/C
VulDeePecker	BE-SEL-SARD	**3.4**	**5.1**	94.9	92.0	93.4

RQ3: VulDeePecker vs. Code Similarity-Based Approaches

RQ3: How effective is VulDeePecker when compared with other approaches?

✧ **Insight**: VulDeePecker is more effective than code similarity-based approaches

System	Dataset	FPR (%)	FNR (%)	TPR (%)	P (%)	F1 (%)
VulDeePecker vs. Other pattern-based vulnerability detection systems						
Flawfinder	BE-SEL	44.7	69.0	31.0	25.0	27.7
RATS	BE-SEL	42.2	78.9	21.1	19.4	20.2
Checkmarx	BE-SEL	43.1	41.1	58.9	39.6	47.3
VulDeePecker	BE-SEL	**5.7**	**7.0**	93.0	88.1	90.5
VulDeePecker vs. Code similarity-based vulnerability detection systems						
VUDDY	BE-SEL-NVD	0	95.1	4.9	100	9.3
VulPecker	BE-SEL-NVD	1.9	89.8	10.2	84.3	18.2
VulDeePecker	BE-SEL-NVD	**22.9**	**16.9**	83.1	78.6	80.8
VUDDY	BE-SEL-SARD	N/C	N/C	N/C	N/C	N/C
VulPecker	BE-SEL-SARD	N/C	N/C	N/C	N/C	N/C
VulDeePecker	BE-SEL-SARD	**3.4**	**5.1**	94.9	92.0	93.4

Using VulDeePecker in Practice

- ✧ VulDeePecker detected 4 vulnerabilities, which were not reported in the NVD, but were "silently" patched by the vendors.
- ✧ These vulnerabilities are missed by most of the other vulnerability detection systems mentioned above

Target product	CVE ID	Vulnerable product published in the NVD	Vulnerability publish time	Vulnerable file in target product	Library/API function call	1st patched version of target product
Libav 10.1	CVE-2013-0851	FFmpeg	12/07/2013	libavcodec/eamad.c	memset	Libav 10.3
Seamonkey 2.31	CVE-2015-4517	Firefox	09/24/2015	.../dom/system/gonk/NetworkUtils.cpp	snprintf	Seamonkey 2.38
	CVE-2015-4513	Firefox	11/05/2015	.../netwerk/protocol/http/Http2Stream.cpp	memset	Seamonkey 2.39
Xen 4.6.0	CVE-2016-9104	Qemu	12/09/2016	tools/qemu-xen/hw/9pfs/virtio-9p.c	memcpy	Xen 4.9.0

Outline

- ✧ Guiding Principles
- ✧ Design of VulDeePecker
- ✧ Experiments and Results
- ✧ **Limitations**
- ✧ Conclusion

Limitations and Open Problems

- ✧ **Present design**
  - ✓ Assuming source code is available
  - ✓ Only dealing with C/C++ programs
  - ✓ Only dealing with vulnerabilities related to library/API function calls
  - ✓ Only accommodating data-flow information, but not control-flow information
  - ✓ Using some heuristics
- ✧ **Present implementation**
  - ✓ Limit to the BLSTM neural network
- ✧ **Present evaluation**
  - ✓ The dataset only contains vulnerabilities about buffer errors and resource management errors

Outline

✧ Guiding Principles

✧ Design of VulDeePecker

✧ Experiments and Results

✧ Limitations

✧ **Conclusion**

Conclusion

- ✧We **initiate the study** of using deep learning for vulnerability detection, and discuss some preliminary guiding principles
- ✧We present **VulDeePecker**, and evaluate it from 3 perspectives
- ✧We present the **first dataset** for evaluating deep learning-based vulnerability detection systems
  - ✧ https://github.com/CGCL-codes/VulDeePecker

New Results (after finishing the paper; in submission)

- ✧Cope with **all kinds of vulnerabilities** (including library/API function calls related ones)
- ✧Accommodate both **data dependency and control dependency**
- ✧**Detect 7 (potential) 0-day vulnerabilities and 8 silently patched vulnerabilities from 4 software products**
- ✧Some deep neural networks are more powerful than others

Takeaways

- ✧ The first deep learning-based vulnerability detection system using a finer-granularity unit code gadget
- ✧Guiding principles for deep learning-based vulnerability detection
- ✧The first dataset for evaluating deep learning-based vulnerability detection systems

Thanks!

lizhen_hust@hust.edu.cn

Data available at:

https://github.com/CGCL-codes/VulDeePecker