SymVuls: シンボリック実行トレースと機械学習を用いたソフトウェアの脆弱性検出
SymVuls: Software vulnerability detection with symbolic execution trace and machine learning

大久保研究室 M2
学籍番号：5593501
加納 永康 (mgs193501@iisec.ac.jp)

Abstract

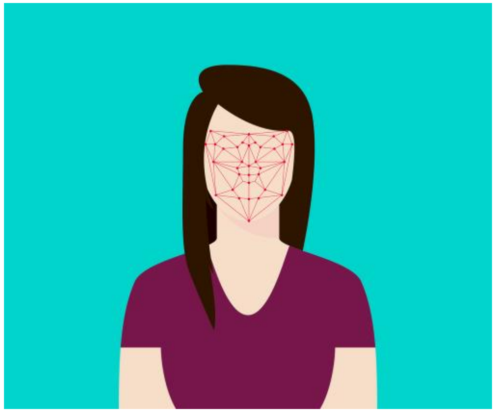
- A proposal of a new methodology to detect software vulnerability with applying machine learning
- Achieved high-performance
  - AUC 0.9953 on the SARD CWE-89 dataset

Background

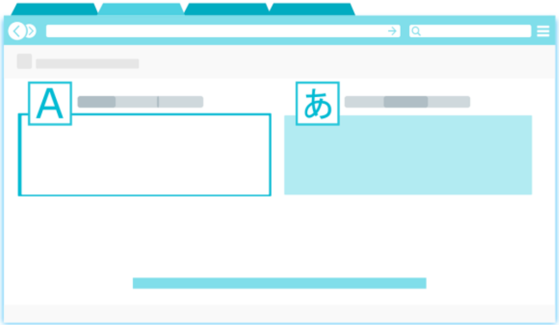
Success of Machine Learning

Machine learning is widely used and successful in many fields.

Face Recognition



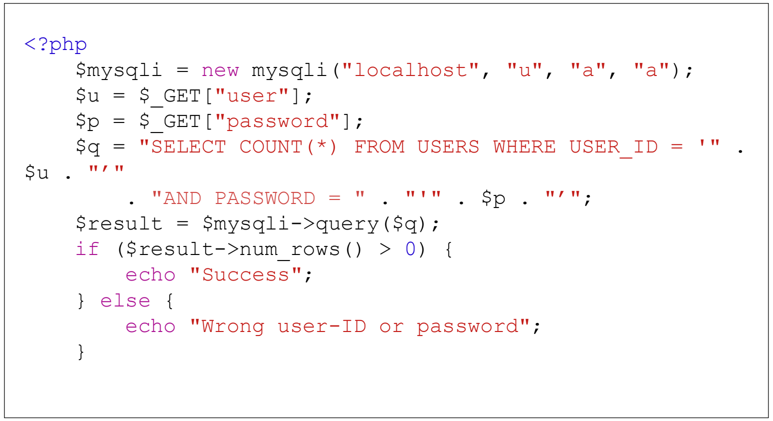
Translation



Text Mining

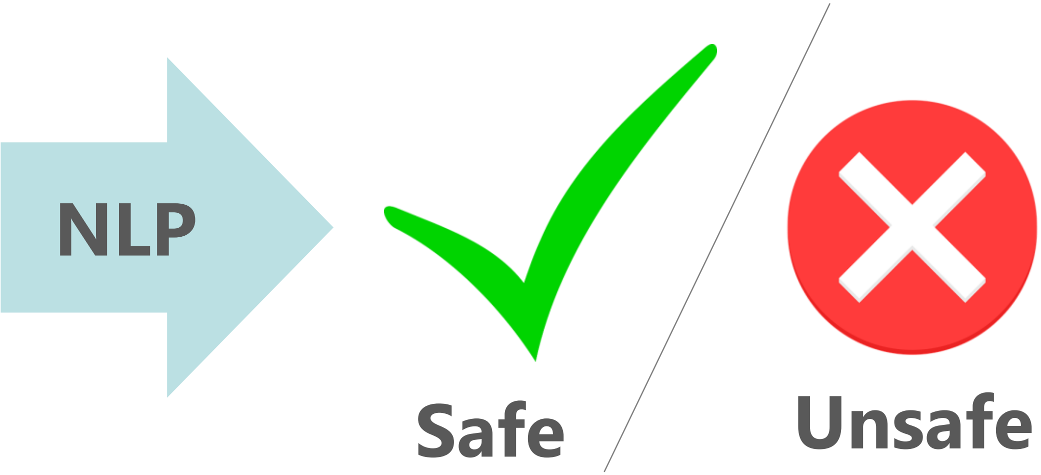
NLP: Natural Language Processing

Understanding Source Code

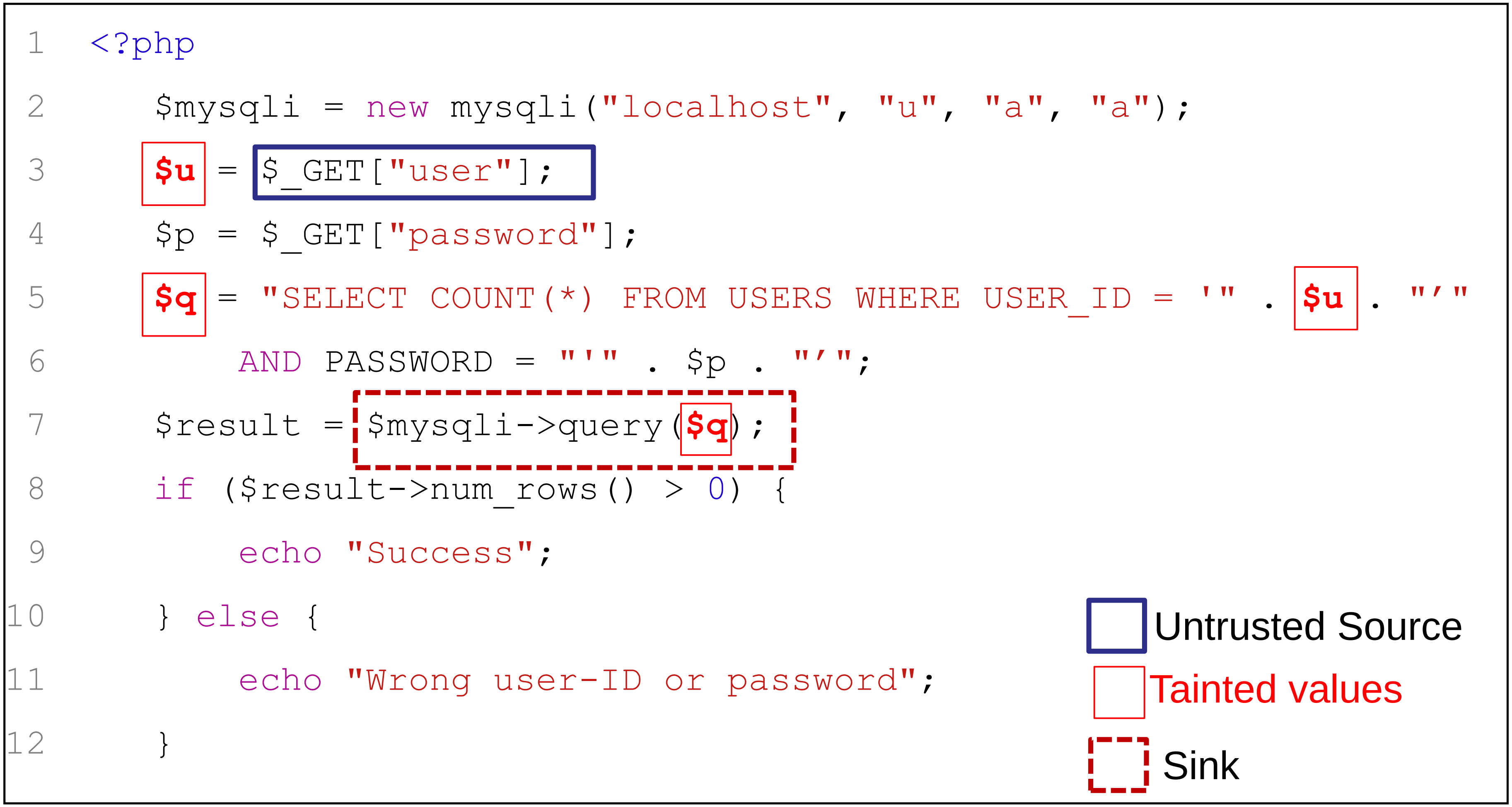


```
<?php
    $mysqli = new mysqli("localhost", "u", "a", "a");
    $u = $_GET["user"];
    $p = $_GET["password"];
    $q = "SELECT COUNT(*) FROM USERS WHERE USER_ID = '" .
$u . "′"
        . "AND PASSWORD = " . "'" . $p . "′";
    $result = $mysqli->query($q);
    if ($result->num_rows() > 0) {
        echo "Success";
    } else {
        echo "Wrong user-ID or password";
    }
```

Source Code



Taint Analysis



```php
<?php
    $mysqli = new mysqli("localhost", "u", "a", "a");
    $u = $_GET["user"];
    $p = $_GET["password"];
    $q = "SELECT COUNT(*) FROM USERS WHERE USER_ID = '" . $u . "'"
        AND PASSWORD = "'" . $p . "'";
    $result = $mysqli->query($q);
    if ($result->num_rows() > 0) {
        echo "Success";
    } else {
        echo "Wrong user-ID or password";
    }
```

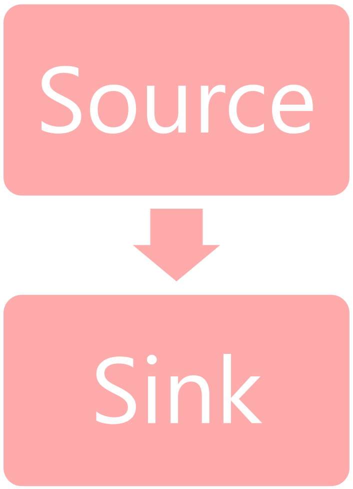
□ Untrusted Source

□ Tainted values

⬚ Sink

Taint Analysis Specifications

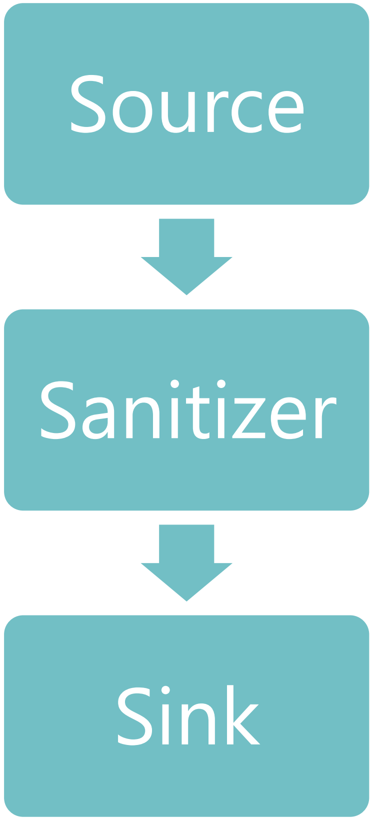
✗ Vulnerable



Source

Sink

✓ Secure



Source

Sanitizer

Sink

Taint Analysis Specifications

Type	XSS	SQLi
Source	_GET _POST _ENV	_GET _POST _ENV
Sanitizer	htmlspecialchars	mysqli_escape_string

Sink	echo
	print
		query()
		execute()

Different for each vulnerability and programing language

Machine Learning for SVP

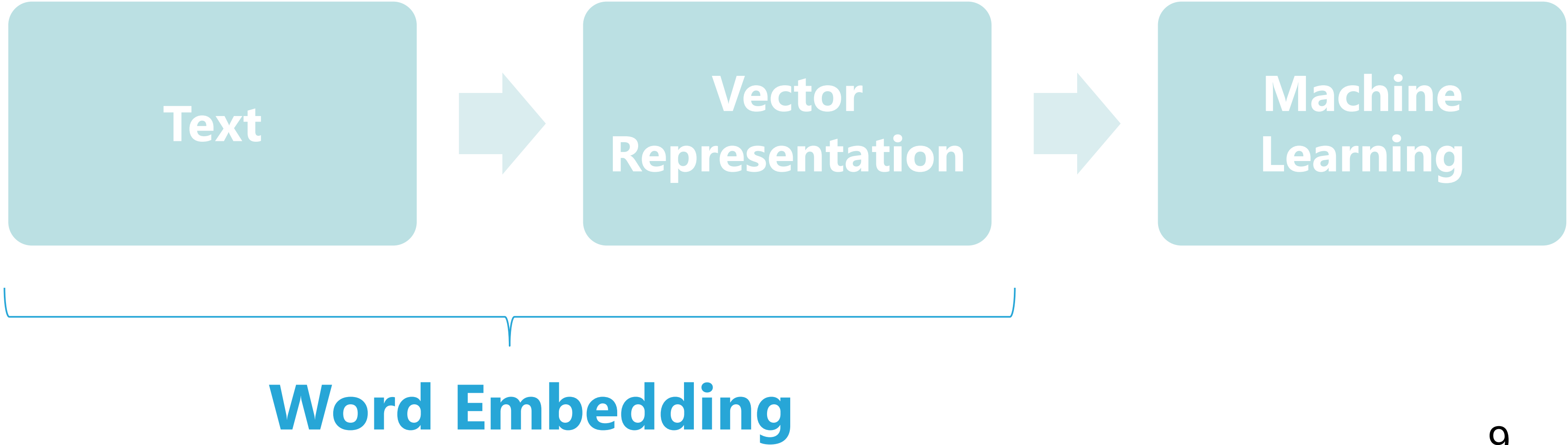
SVP (Software Vulnerability Prediction)

1. Software Metrics
2. **Text Mining Features**
3. Graph Features
4. Taint Analysis Features
5. Others

ZhanJun Li, Yan Shao: *A Survey of Feature Selection for Vulnerability Prediction Using Feature-based Machine Learning*, ICMLC '19: Proceedings of the 2019 11th International Conference on Machine Learning and Computing, P.36–42, 2019

Text Mining Features

Analyze source code as text, and convert it to vector representation to feed it into the machine learning algorithm.



Text

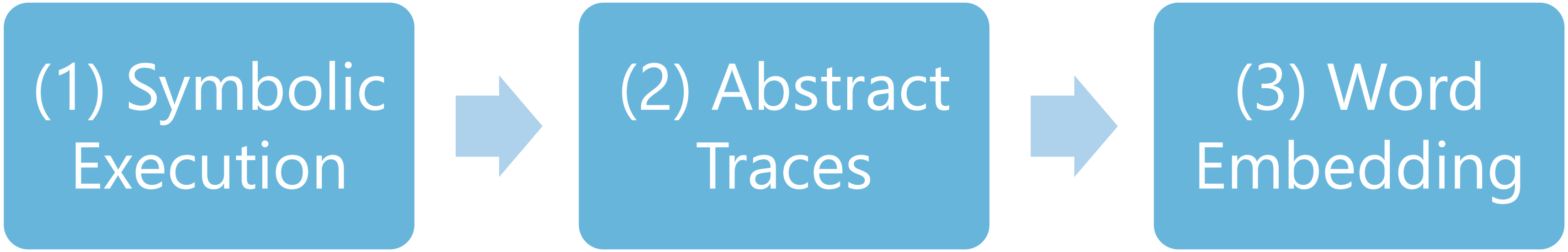
Vector Representation

Machine Learning

Word Embedding

Word Embedding for Programming Language

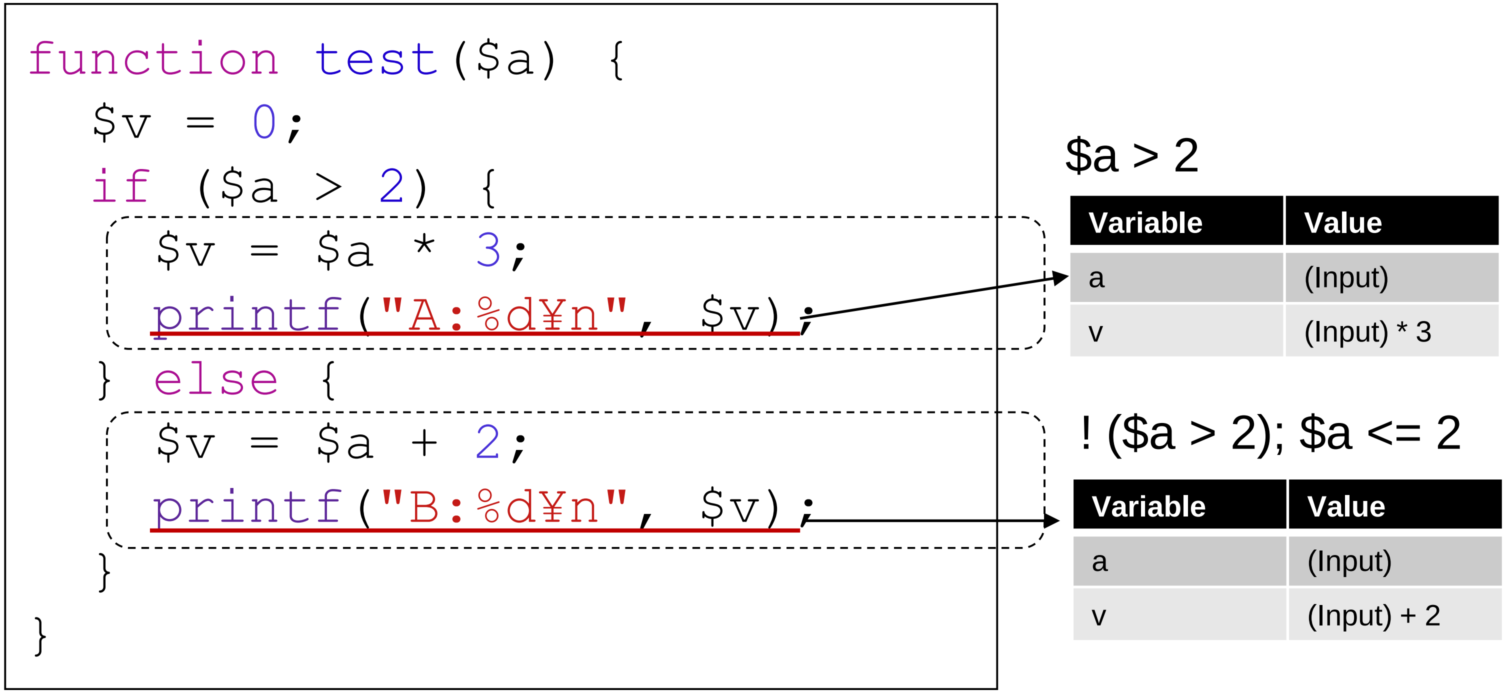
Jordan Henkel et al. proposed a new methodology that uses **Symbolic Execution** Traces to perform high-quality word embedding



Jordan Henkel et al., *CodeVectors: Understanding Programs Through Embedded Abstracted Symbolic Traces* (2018)

Symbolic Execution

"In computer science, symbolic execution (also symbolic evaluation or symbex) is a means of analyzing a program to determine what inputs cause each part of a program to execute."



```
function test($a) {
   $v = 0;
   if ($a > 2) {
      $v = $a * 3;
      printf("A:%d¥n", $v);
   } else {
      $v = $a + 2;
      printf("B:%d¥n", $v);
   }
}
```

$a > 2

Variable	Value
a	(Input)
v	(Input) * 3

! ($a > 2); $a <= 2

Variable	Value
a	(Input)
v	(Input) + 2

Often referred with SMT (Satisfiability Modulo Theories), mainly for test automation, test cases generation.

https://en.wikipedia.org/wiki/Symbolic_execution

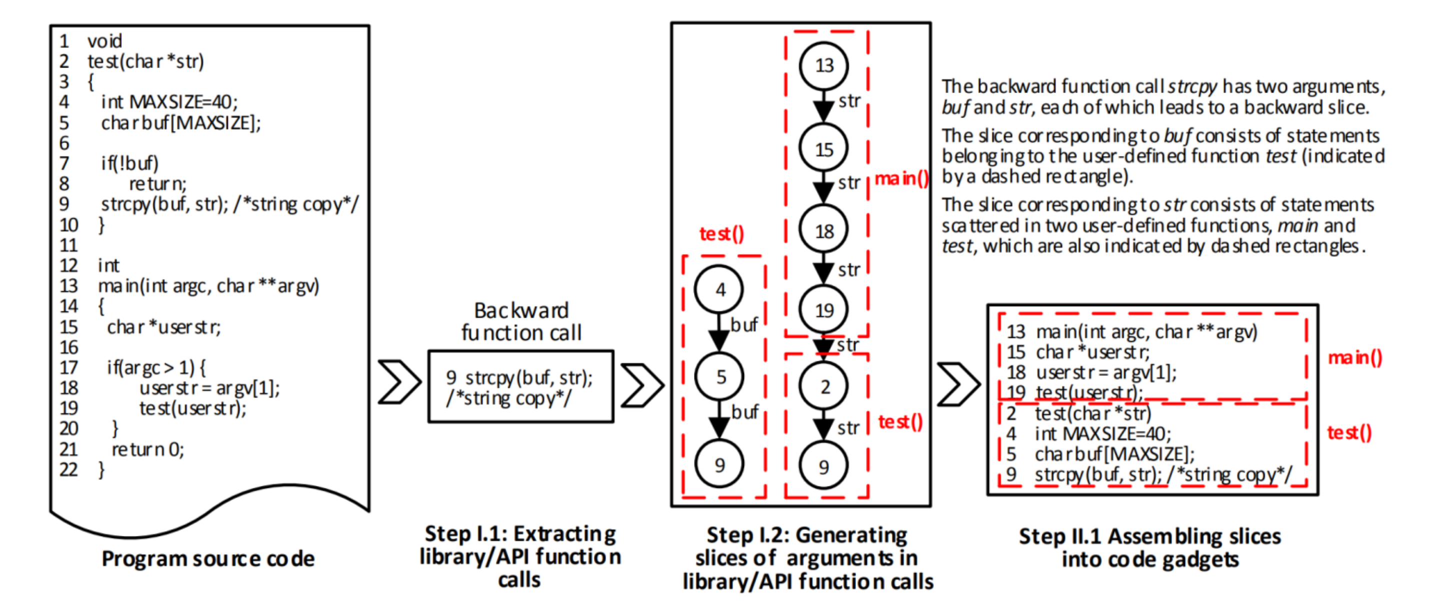
Research Questions

RQ1 Would vulnerability detection result be improved if Symbolic Execution traces are used for Word Embedding?

RQ2 Does the trained model have enough versatility to detect vulnerabilities in other test source code?

Related Work

VulDeePecker (2018)



```
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

Backward function call

```
9 strcpy(buf, str);
/*string copy*/
```

Step I.1: Extracting library/API function calls

Step I.2: Generating slices of arguments in library/API function calls

The backward function call *strcpy* has two arguments, *buf* and *str*, each of which leads to a backward slice.

The slice corresponding to *buf* consists of statements belonging to the user-defined function *test* (indicated by a dashed rectangle).

The slice corresponding to *str* consists of statements scattered in two user-defined functions, *main* and *test*, which are also indicated by dashed rectangles.

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

Step II.1 Assembling slices into code gadgets

Figure 3. Illustrating the extraction of library/API function calls (Step I.1) from a (training) program, which contains a backward function call (i.e., *strcpy*) that is also used as an example to demonstrate the extraction of program slices (Step I.2) and the assembly of program slices into code gadgets (Step II.1).

Word2Vec & BLSTM learning

Zhen Li, Deqing Zou, Shouhuai Xux, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng and Yuyi Zhong
VulDeePecker: A Deep Learning-Based System for Vulnerability Detection (NDSS 2018, 2018)

- **Using Program Slicing to extract vulnerable code**
- **Need to specify a “Sink” or “Source” to get the slice**

TAP (2019)

AUC	0.9941

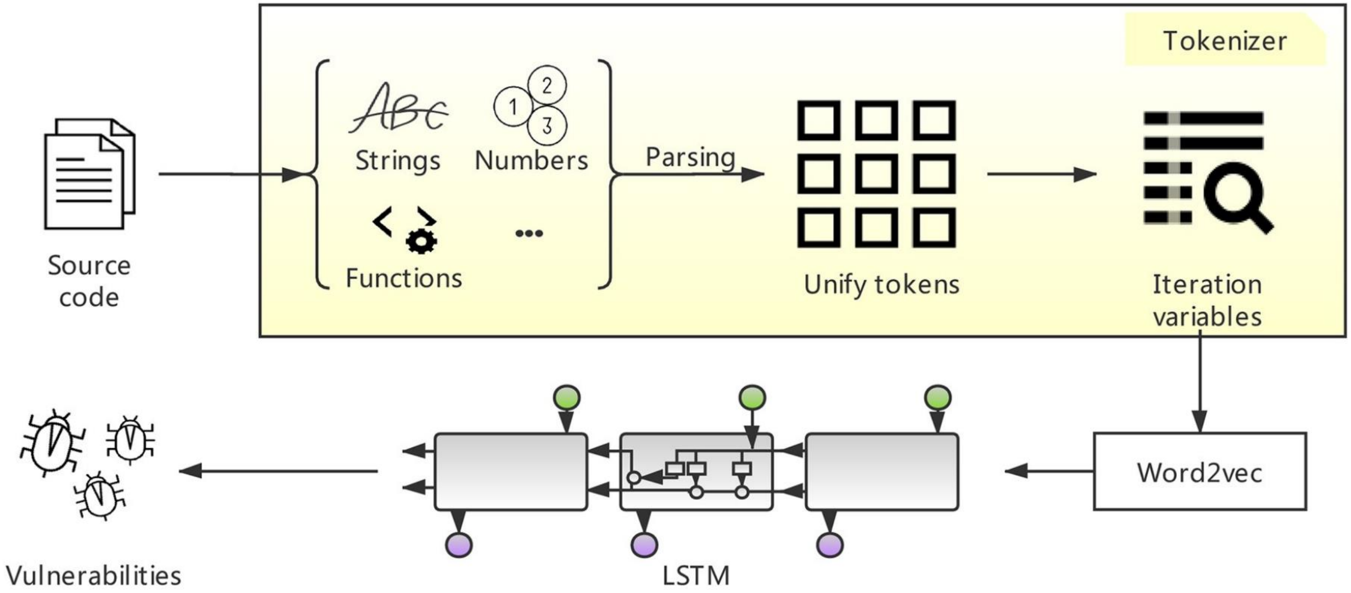


Fig 1. Overview of TAP.

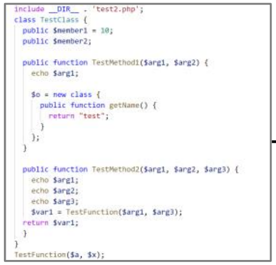
Yong Fang, Shengjun Han, Cheng HuangID, Runpu Wu
TAP: A static analysis model for PHP vulnerabilities based on token and deep learning technology (2019)

- **Grouping of token is done in advance**
- **Highly dependent on PHP processor, Versatility is in doubt**

New Methodology

Proposed Methodology

Source Code



```
include __DIR__ . 'test2.php';
class TestClass {
  public $member1 = 10;
  public $member2;

  public function TestMethod1($arg1, $arg2) {
    echo $arg1;

    $o = new class {
      public function getName() {
        return "test";
      }
    };
  }

  public function TestMethod2($arg1, $arg2, $arg3) {
    echo $arg1;
    echo $arg2;
    echo $arg3;
    $var1 = TestFunction($arg1, $arg3);
  return $var1;
  }
}
TestFunction($a, $x);
```

Symbolic Execution & Simplification

Function Call Traces

```
sprintf ( #str# ' %s ' ,  )
mysql_connect ( #str# , #str# , #str# )
mysql_select_db ( #str# )
echo ( #str# . sprintf ( #str# ' %s ' , _GET [ #str# ] ) . #str# )
echo ( #str# . sprintf ( #str# ' %s ' ,  ) . #str# )
mysql_query ( sprintf ( #str# ' %s ' , _GET [ #str# ] ) )
mysql_query ( sprintf ( #str# ' %s ' ,  ) )
echo ( #str# )
echo ( #str# )
echo ( #str# )
echo ( #str# )
…
…
```

Word Embedding

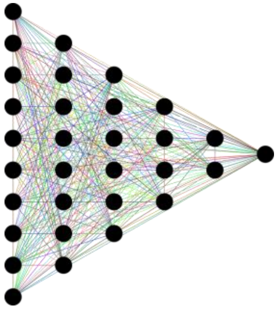
Vectors

```
echo = [-0.5185164 , -0.8347919 ,  1.171482  ,  0.6900932 ,  0.64918447,
      -0.08244713,  0.5991925 ,  0.10949674,  -0.47637585,   0.9477746 ,
      -0.8089947 , -0.7869316 ,  0.42289555,   0.49892744,  -0.3618514 ,
      -1.4450327 , -0.43445647,   0.81112355,  -1.177568  ,  0.94679874,
       0.7710409 , -0.09612622,  -0.85266286,  -0.46487492,   1.0379548 ,
       1.0902969 ,  0.21682492,  -0.1771169 ,  1.2194921 ,  1.8131593 ,
        ...]
sprintf = [-0.5185164 , -0.8347919 ,  1.171482  ,  0.6900932 ,  0.64918447,
      -0.08244713,  0.5991925 ,  0.10949674,  -0.47637585,   0.9477746 ,
      -0.8089947 , -0.7869316 ,  0.42289555,   0.49892744,  -0.3618514 ,
      -1.4450327 , -0.43445647,   0.81112355,  -1.177568  ,  0.94679874,
       0.7710409 , -0.09612622,  -0.85266286,  -0.46487492,   1.0379548 ,
       1.0902969 ,  0.21682492,  -0.1771169 ,  1.2194921 ,  1.8131593 ,
        ...]
mysql_query = [-0.5185164 , -0.8347919 ,  1.171482  ,  0.6900932 ,  0.64918447,
      -0.08244713,  0.5991925 ,  0.10949674,  -0.47637585,   0.9477746 ,
      -0.8089947 , -0.7869316 ,  0.42289555,   0.49892744,  -0.3618514 ,
      -1.4450327 , -0.43445647,   0.81112355,  -1.177568  ,  0.94679874,
       0.7710409 , -0.09612622,  -0.85266286,  -0.46487492,   1.0379548 ,
       1.0902969 ,  0.21682492,  -0.1771169 ,  1.2194921 ,  1.8131593 ,
        ...]
...
```

Filtering

```
mysql_query = [-0.5185164 , -0.8347919 ,  1.171482  ,  0.6900932 ,  0.64918447,
      -0.08244713,  0.5991925 ,  0.10949674,  -0.47637585,   0.9477746 ,
      -0.8089947 , -0.7869316 ,  0.42289555,   0.49892744,  -0.3618514 ,
      -1.4450327 , -0.43445647,   0.81112355,  -1.177568  ,  0.94679874,
       0.7710409 , -0.09612622,  -0.85266286,  -0.46487492,   1.0379548 ,
       1.0902969 ,  0.21682492,  -0.1771169 ,  1.2194921 ,  1.8131593 ,
        ...]
...
```

Learning



Model

Language: PHP

Target Vulnerability: CWE-89 SQL Injection

Function Call Traces

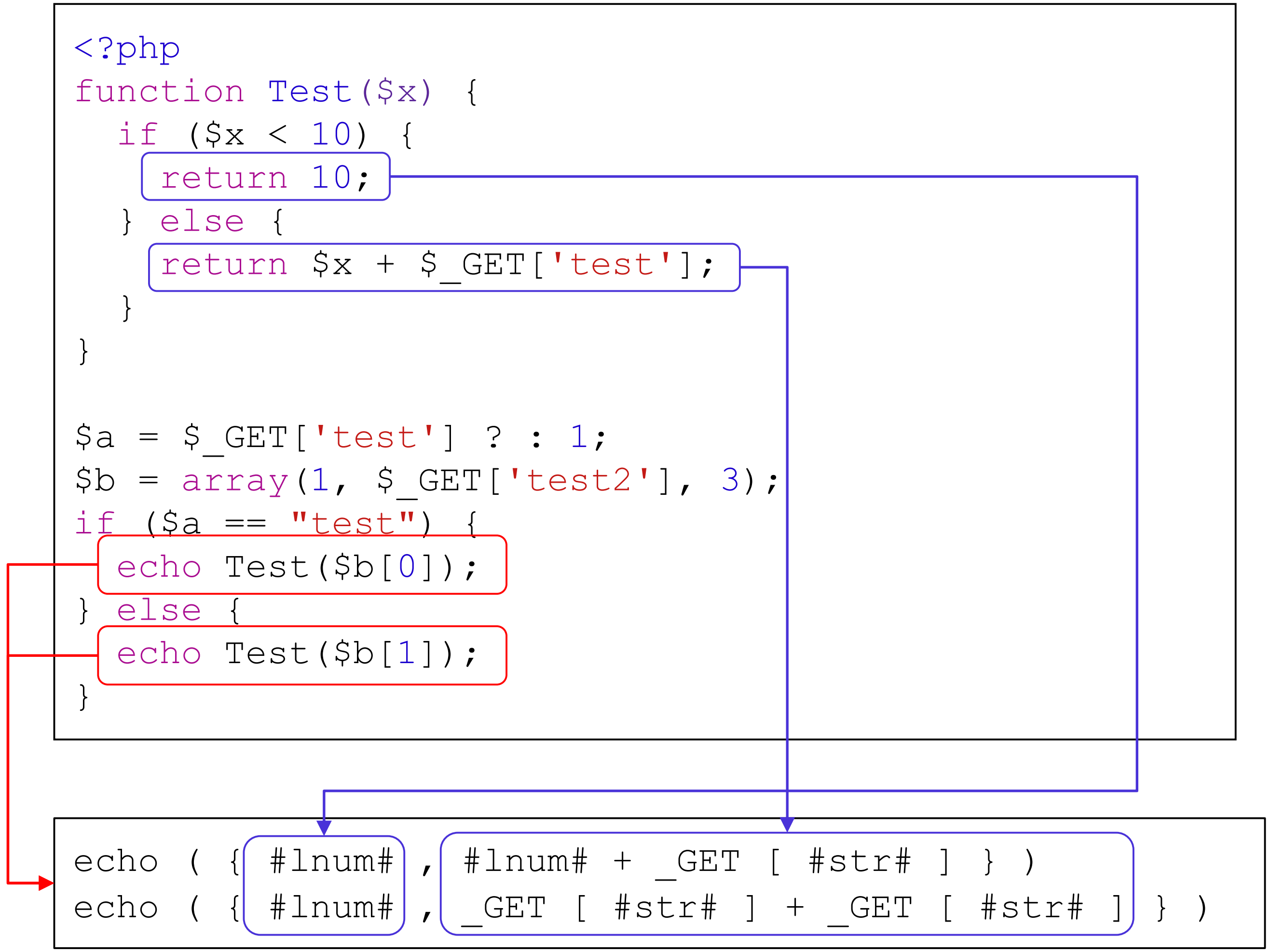
```
<?php

$conn = mysql_connect('localhost', 'mysql_user', 'mysql_password');
mysql_select_db('dbname') ;

$tainted = "";
if ($_GET['input'] == "safe") {
  $tainted = mysql_real_escape_string($_GET['test']);
} else {
  $tainted = $_GET['test'];
}
$query = "SELECT * FROM users where username = '" . $tainted . "'";
$res = mysql_query($query); //execution

mysql_close($conn);

?>
```

Extract the called **function name and the parameters** in the symbolic execution

```
mysql_connect ( #str# , #str# , #str# )
mysql_select_db ( #str# )
mysql_real_escape_string ( _GET [ #str# ] )
mysql_query ( #str# ' . mysql_real_escape_string ( _GET [ #str# ] ) . ' )
mysql_query ( #str# ' . _GET [ #str# ] . ' )
mysql_close ( mysql_connect ( #str# , #str# , #str# ) )
```

Classes / User Defined Functions

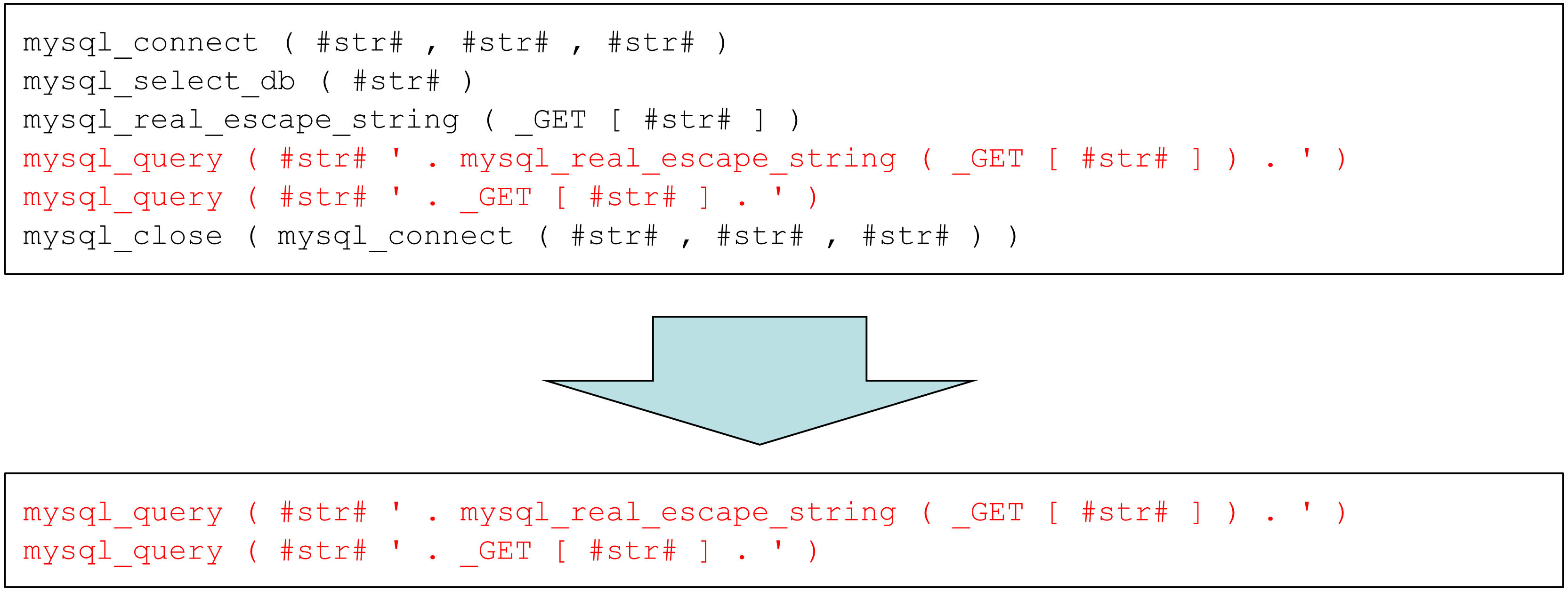
Expand the function call like making a Program Slice



```
<?php
function Test($x) {
  if ($x < 10) {
    return 10;
  } else {
    return $x + $_GET['test'];
  }
}

$a = $_GET['test'] ? : 1;
$b = array(1, $_GET['test2'], 3);
if ($a == "test") {
  echo Test($b[0]);
} else {
  echo Test($b[1]);
}
```

```
echo ( { #lnum# , #lnum# + _GET [ #str# ] } )
echo ( { #lnum# , _GET [ #str# ] + _GET [ #str# ] } )
```

Filtering

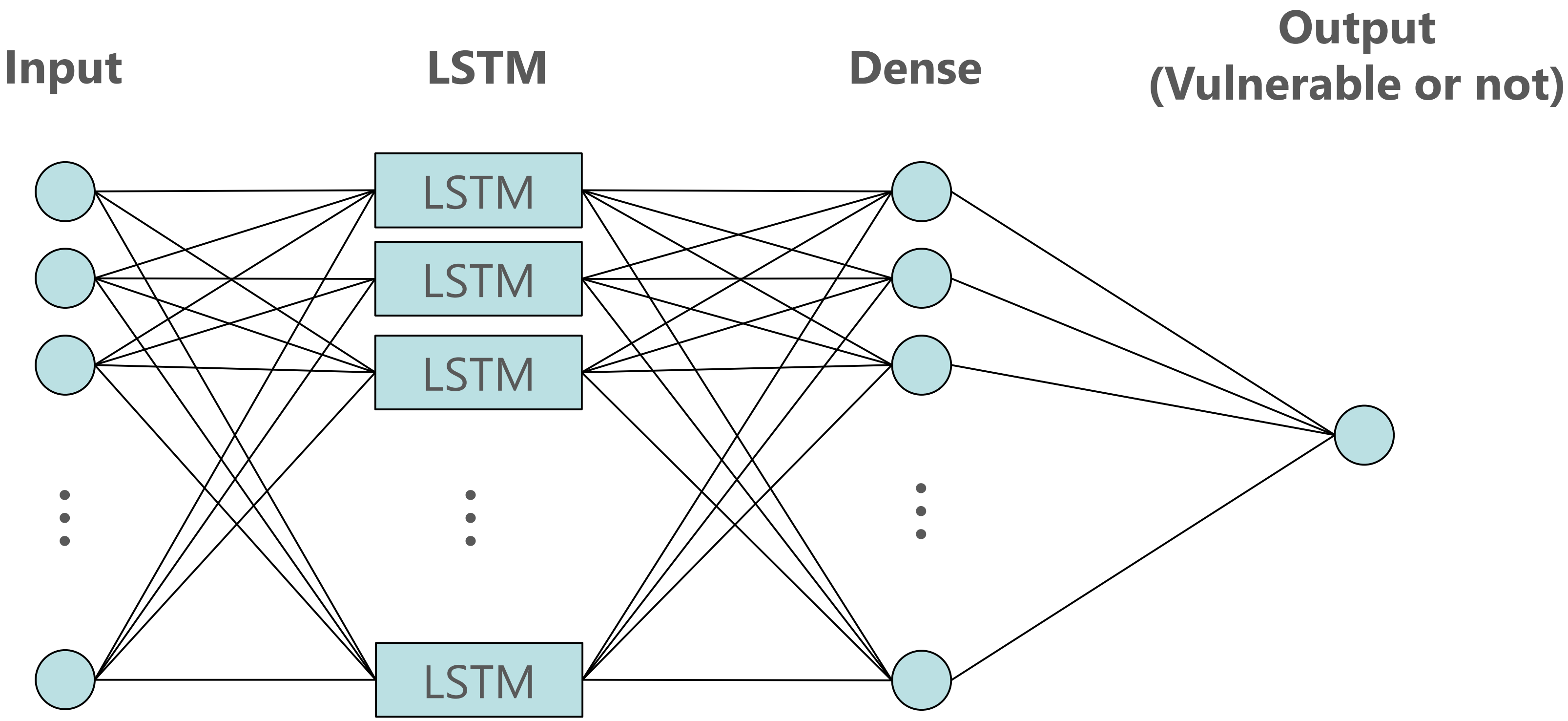


```
mysql_connect ( #str# , #str# , #str# )
mysql_select_db ( #str# )
mysql_real_escape_string ( _GET [ #str# ] )
mysql_query ( #str# ' . mysql_real_escape_string ( _GET [ #str# ] ) . ' )
mysql_query ( #str# ' . _GET [ #str# ] . ' )
mysql_close ( mysql_connect ( #str# , #str# , #str# ) )
```

```
mysql_query ( #str# ' . mysql_real_escape_string ( _GET [ #str# ] ) . ' )
mysql_query ( #str# ' . _GET [ #str# ] . ' )
```

Filter out traces other than *mysql_query*

Machine Learning Model



Evaluation

Experiment (1)

Objective

Confirm the impact of Word Embedding quality

#	Preprocess	Word Embedding Algorithm	Machine Learning Model (Same for all)	Filtering
1	**Symbolic Execution Trace**	**Word2Vec**	**LSTM + Dense + Sigmoid**	**Yes**
2	Symbolic Execution Trace	Word2Vec	LSTM + Dense + Sigmoid	No
3	Symbolic Execution Trace	Bag of Words	LSTM + Dense + Sigmoid	No
4	token_get_all	Word2Vec	LSTM + Dense + Sigmoid	No
5	token_get_all	Bag of Words	LSTM + Dense + Sigmoid	No

Dataset

SARD* CWE-89 PHP Dataset

Safe samples	Unsafe samples	Total
8,640	912	9,552

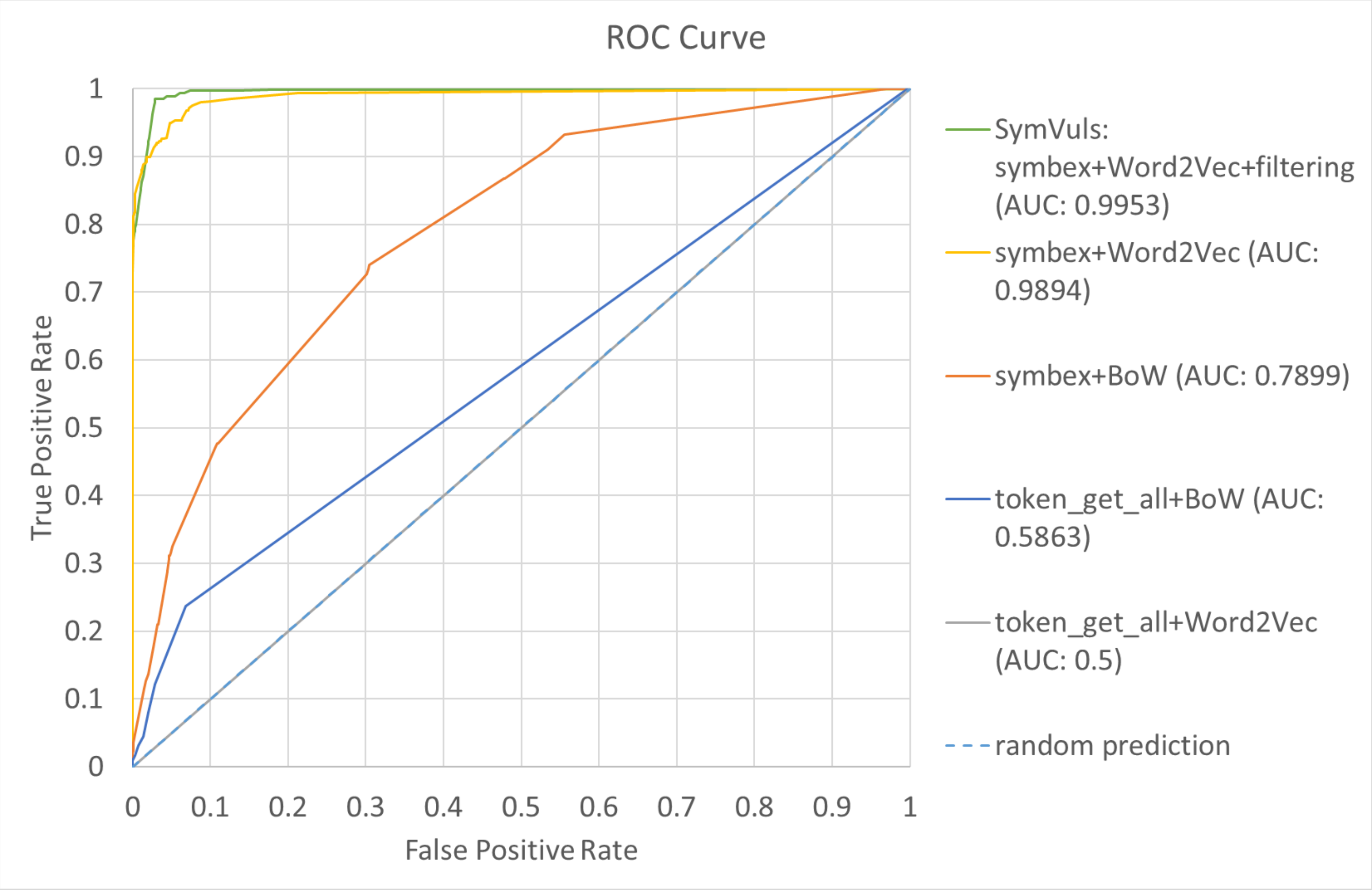
SARD CWE-89 Dataset had an error in samples that use filter_var with FILTER_VALIDATE_EMAIL. The error was corrected.

* National Institute of Standards and Technology
"The NIST Software Assurance Reference Dataset Project"
https://samate.nist.gov/SARD/

Indicators

- Receiver Operating Characteristic Curve (ROC Curve)
  - A graph that shows the characteristic of sensitivity of the model to the target vulnerability
- Area Under the Curve (AUC)
  - Accumulated area under the curve

Result (1)


ROC Curve
True Positive Rate
False Positive Rate
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
SymVuls: symbex+Word2Vec+filtering (AUC: 0.9953)
symbex+Word2Vec (AUC: 0.9894)
symbex+BoW (AUC: 0.7899)
token_get_all+BoW (AUC: 0.5863)
token_get_all+Word2Vec (AUC: 0.5)
random prediction

Comparison with TAP

Model	Safe Samples			Unsafe Samples			Accuracy	AUC
	Precision	Recall	F1	Precision	Recall	F1		
SymVuls	0.9778	0.9983	0.9879	0.9795	0.7851	0.8716	0.9779	0.9953
TAP	0.9773	0.9988	0.9880	0.9874	0.7970	0.8820	0.9782	0.9941

Experiment (2)

Objective

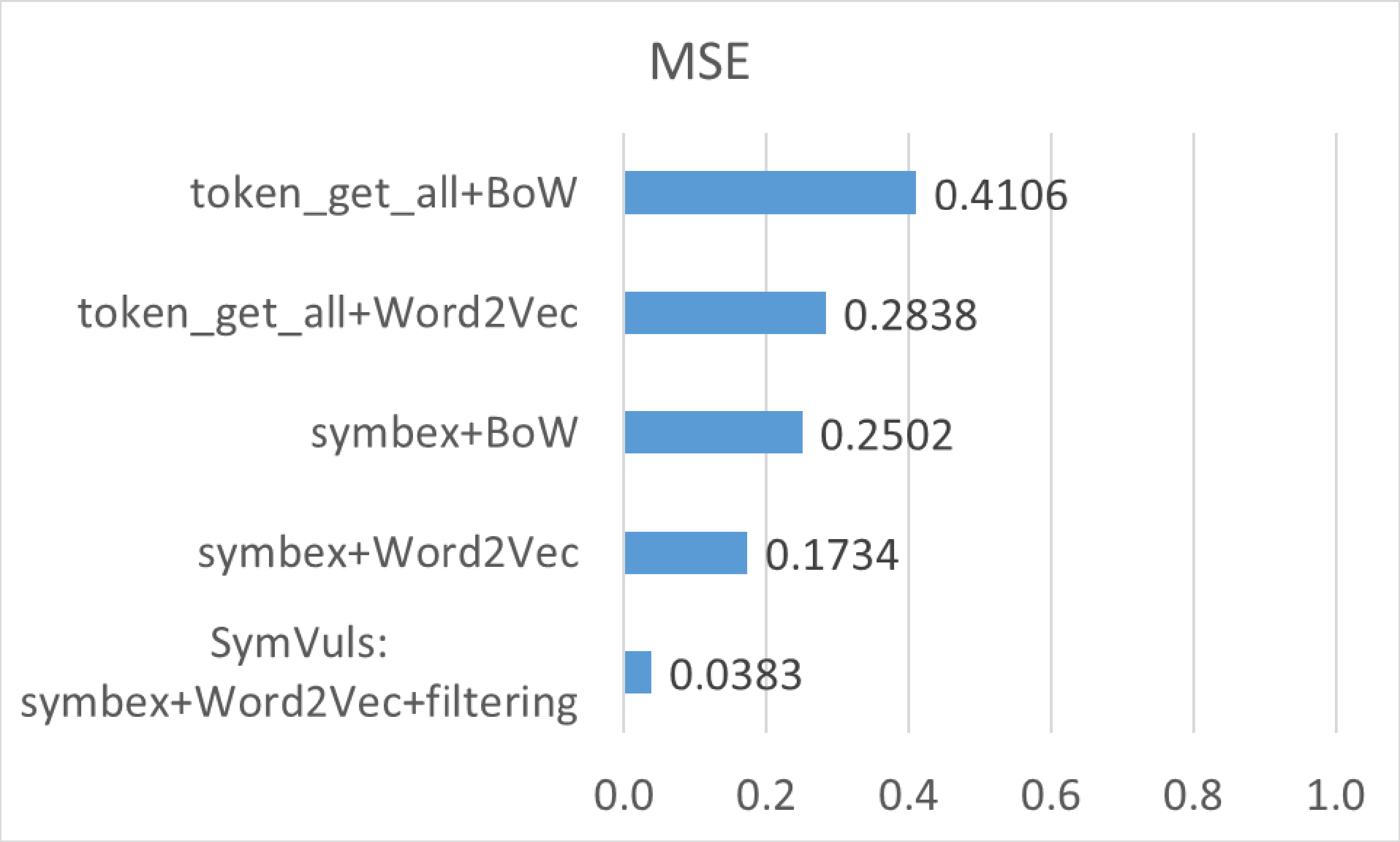
Confirm the versatility of the trained model

- Prepared a set of very simple vulnerable PHP code
- Compare the loss

#	Content
1	Very simple code
2	filter_var + sprintf formatting
3	Long code
4	sprintf formatting

Result (2)

Evaluated with MSE (Mean Squared Error) - Lower is better



Summary

RQ1

RQ1 Would vulnerability detection result be improved if Symbolic Execution traces are used for Word Embedding?

- Yes, it does improve
- Symbolic Execution helps the feature extraction
- Word Embedding quality is also an important factor to keep the features

RQ2

RQ2 Does the trained model have enough versatility to detect vulnerabilities in other test source code?

- Low versatility if no filtering applied
- The API that causes the vulnerability needs to be specified in one way or another

End

Thank you for listening

Any questions?