Detecting vulnerabilities in source code with AI

Harald Vranken & Arjen Hommersom

CC BY NC SA

Goal of this talk…

- Give brief overview of our work
- Discover possibility for synergy/cooperation

cc BY NC SA

Introduction

- How to detect/discover vulnerabilities in software?

- DAST: dynamic analysis (execution of binary)
  - vulnerability scanners
  - fuzz testing

- SAST: static analysis (of source code)
  - type checking (for information-flow analysis)
  - static code analysis (Coverity, Fortify, PreFAST, …)

    → rule-based analysis using control/data flow analysis
  - *applying AI*

    → *learning from code properties*

cc BY NC SA

Vulnerability detection with AI

- Analyse meta-information, ie. metrics derived from
  - source code (complexity, size)
  - code repositories (churn, age, comments)
  - developers (developer activity, fault history)

Very modest results
- applicable only for mature software systems
- unable to distinguish vulnerabilities from defects

- Analyse program code (syntax and/or semantics)
  - anomaly detection (look for patterns that do not conform to normal/expected behaviour)
  - vulnerable code pattern recognition (look for patterns that relate to abnormal behaviour)

Eg. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey
Seyed Mohammad Ghaffarian and Hamid Reza Shahriari, ACM Computing Surveys, Vol. 50, No. 4, Article 56, August 2017.

cc BY NC SA

Lots of research opportunities

- Method
  - approach (analyse meta-information, program syntax/semantics, hybrid, combi SAST/DAST, …)
  - AI technique (rule-based, machine learning, deep learning)
  - type of application and programming language (web applications in PHP and JavaScript, embedded software in C/C++, general applications in Java or C#, …)
  - type of vulnerabilities (code injection, buffer overflow, …)

- Questions
  - How good is it for detecting vulnerabilities?
  - How does it compare to other methods?
  - How does it make decisions (explainable)?
  - How specific/general is it?
  - How can it be applied in practice?

CC BY NC SA

Our (ongoing) research

- Team: Arjen Hommersom, Harald Vranken, master students
  - Discovering XSS and SQLi vulnerabilities in PHP code, using machine learning (Jorrit Kronjee, 2018) and deep learning (Bart Elema, 2020)
  - Discovering path traversal and SQLi/XMLi vulnerabilities in C# code, using code2vec (Mathijssen, 2022)
  - Discovering memory corruption vulnerabilities in C++ code, using graph neural networks (De Kraker, 2022) and layerwise relevance propagation (Foeken, 2022)

- Publications
  - Kronjee, Hommersom & Vranken: *Discovering software vulnerabilities using data-flow analysis and machine learning* (ARES 2018)
  - De Kraker, Vranken & Hommersom: *GLICE: Combining Graph Neural Networks and Program Slicing to Improve Software Vulnerability Detection* (DevSecOpsRO 2023)

CC BY NC SA

Our research method

1. Create dataset of code samples (both vulnerable and non-vulnerable samples)
   - challenge: how to obtain samples?

2. Translate code into some abstract representation (graph/model)
   - challenge: how to preserve properties that identify vulnerabilities?

3. Transform graph/model into feature vectors
   - challenge: ML (with feature engineering) or DL?

4. Train a classifier
   - challenge: what ML/DL model?

5. Evaluate trained classifier
   - challenge: how to explain false classifications, and how to improve the model?

Approach
- Apply domain knowledge (ie. security)
- Consider AI methods ‘as is’ (toolbox)

→ primarily security research (applied AI research)

BY NC SA

Case studies

1. Discovering XSS and SQLi vulnerabilities in PHP code using machine learning
   Kronjee, Hommersom &Vranken: *Discovering software vulnerabilities using data-flow analysis and machine learning*
   (ARES 2018)

2. Discovering memory corruption vulnerabilities in C++ code using graph neural networks
   De Kraker, Vranken & Hommersom: *GLICE: Combining Graph Neural Networks and Program Slicing to Improve Software Vulnerability Detection*
   (DevSecOpsRO 2023)

cc BY NC SA

Discovering XSS and SQLi vulnerabilities in PHP code

- Inspired by prior work of Fabian Yamaguchi and Konrad Rieck (and others) in Germany and Lwin Khin Shar and Hee Beng Kuan Tan (and others) in Singapore

- Approach
  1. Extract features from PHP source code samples using data-flow analysis
  2. Feature selection and supervised machine learning to train various classifiers
  3. Perform experiments to evaluate how good it is

cc BY NC SA

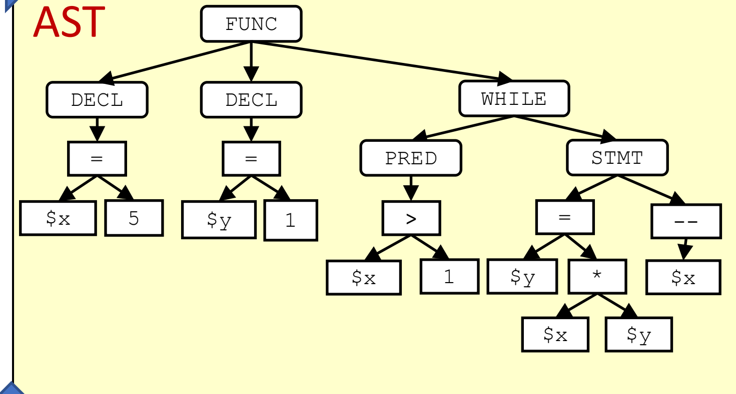
Data-flow analysis

- Create AST from source code
- Derive CFG from AST
- Determine reaching definitions and use-definitions chains
  - Definition (assignment) on line *i* reaches to line *j* if the variable in *i* can reach *j* without intervening definitions
  - UD chain for use of a variable lists all definitions of that variable that can reach that use without any other intervening definitions

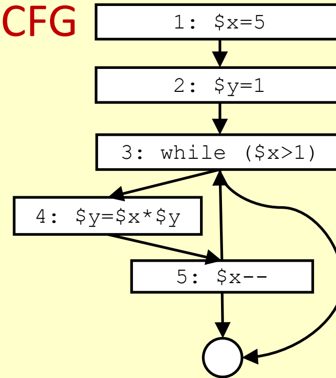
Source code

```
$x = 5;
$y = 1;
while ($x > 1) {
    $y = $x * $y;
    $x--;
}
```

AST



CFG



Reaching definitions

Line	IN	GEN	OUT	KILL
1	∅	x1	x1	x5
2	x1	y2	x1,y2	y4
3	x1,x5,y2,y4	∅	x1,x5,y2,y4	∅
4	x1,x5,y2,y4	y4	x1,x5,y4	y2
5	x1,x5,y4	x5	x5,y4	x1

Use-definition chains

Use	UD chain
y4	y2,y4
x5	x1,x5

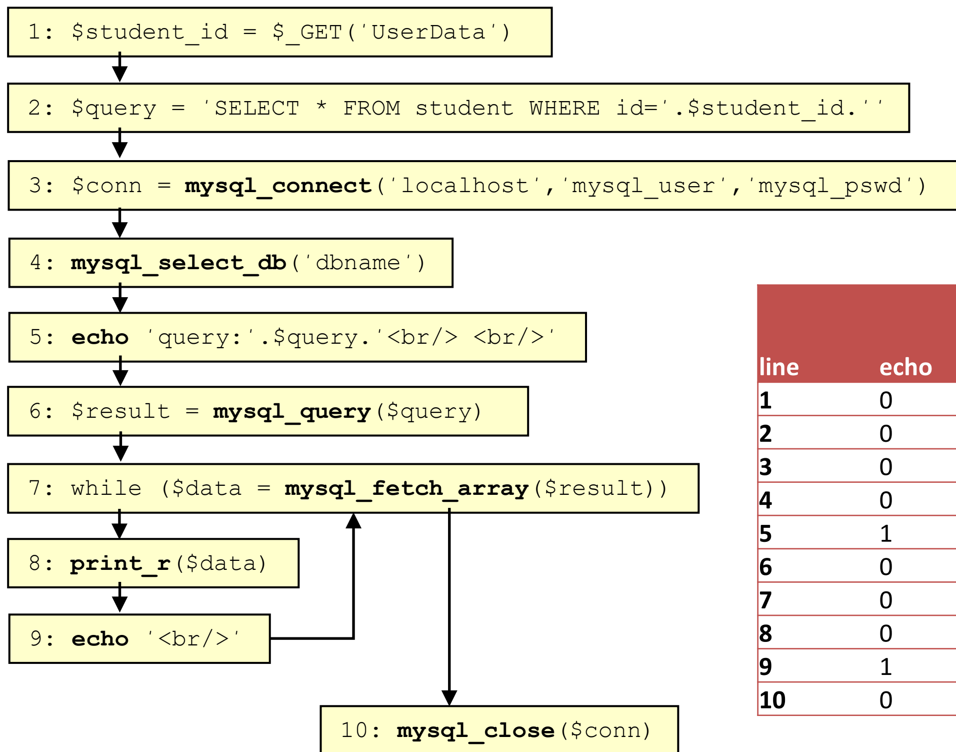
cc BY NC SA

Extracting features

- General pattern with XSS and SQLi
  - tainted data enters application
  - application does not sufficiently validate and sanitise the data
  - application uses the data in a vulnerable function
- Tainted data
  - assume all variables are tainted (except variables of type *float*, *int*, *double, bool*)
  - feature: consider line of code as tainted if at least one variable on the line is tainted
- Sanitisation
  - PHP function `filter_var()` uses constant as second parameter to specify type of filtering
  - features: use of these constants (like `FILTER_SANITIZE_STRING`)
- Potentially vulnerable functions
  - features: function usage (also consider UD chain for data used in function)

cc BY NC SA

Example



use	UD chain
...	...
$result7	$result6
$data8	$data7
...	...

line	echo	mysql_ close	mysql_ connect	mysql_ fetch_ array	mysql_ query	...	tainted	vulnerable
1	0	0	0	0	0	...	1	0
2	0	0	0	0	0	...	1	0
3	0	0	1	0	0	...	0	0
4	0	0	0	0	0	...	0	0
5	1	0	0	0	0	...	1	0
6	0	0	0	0	1	...	1	1
7	0	0	0	1	1	...	1	1
8	0	0	0	1	0	...	1	1
9	1	0	0	0	0	...	0	0
10	0	1	1	0	0	...	0	0

cc BY NC SA

Results

- Created dataset of PHP code samples (from NIST NVD and SAMATE) with XSS or SQLi vulnerabilities
- Trained 5 ML classifiers (Random forest, Decision tree, …) separately for SQLi and XSS
- Evaluated classifiers
  - standard metrics
  - comparison against other open-source tools
  - try on PHP code repositories (identified SQLi in Piwigo, CVE-2018-6883)

SQLi

	AUC-PR
Decision Tree	**0.88**
Logistic Regression	0.87
Random Forest	0.85
TAN	0.75
Naive Bayes	0.64
Dummy	0.51

	Precision	Recall	F_1-score
Our tool	**0.94**	**0.94**	**0.94**
Pixy	0.86	0.61	0.69
RIPS	0.83	0.80	0.82
WAP	0.83	0.84	0.83
Yasca	0.01	0.10	0.02

XSS

	AUC-PR
Decision Tree	**0.82**
Random Forest	**0.82**
TAN	0.81
Logistic Regression	0.79
Naive Bayes	0.69
Dummy	0.51

	Precision	Recall	F_1-score
Our tool	**0.79**	**0.71**	**0.71**
Pixy	0.61	0.61	0.61
RIPS	0.37	0.61	0.46
WAP	0.51	0.58	0.51
Yasca	0.24	0.25	0.24

cc BY NC SA

Case studies

1. Discovering XSS and SQLi vulnerabilities in PHP code using machine learning
   Kronjee, Hommersom &Vranken: *Discovering software vulnerabilities using data-flow analysis and machine learning*
   (ARES 2018)

2. Discovering memory corruption vulnerabilities in C++ code using graph neural networks
   De Kraker, Vranken & Hommersom: *GLICE: Combining Graph Neural Networks and Program Slicing to Improve Software Vulnerability Detection*
   (DevSecOpsRO 2023)

cc BY NC SA

Discovering buffer overflows in C++ code

- Inspired by prior work of Chinese researchers at Huazhong University of Science and Technology (SySeVR, VulDeepecker) and Northwest University (FUNDED)

- Approach
  1. Apply program slicing
     - program slice contains all statements on which arguments of a function call depend
     - similar to SySeVR, but consider call tree of multiple functions instead of single function
  2. Transform program slide into Graph Neural Network (GNN)
     - similar to FUNDED, but with support for more language constructs and bug fixes
  3. Train classifier (GLICE)
  4. Perform experiments to evaluate how good it is

CC BY NC SA

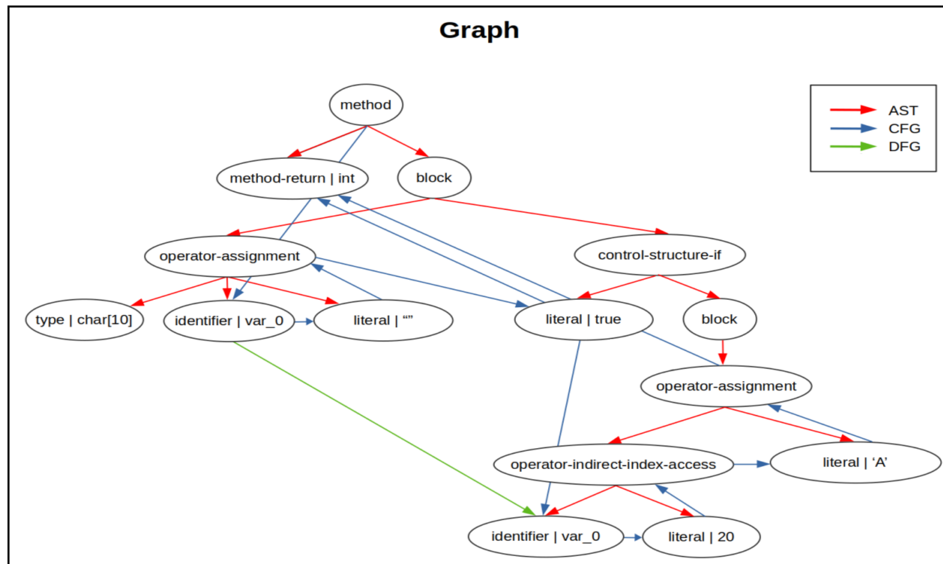
Example

Code sample with vulnerability at line 4:

```
1    void Print_Prefix(char* source)
2    {
3        char dest[5]="A";
4        strcat(dest, source);
5        printf("%s\n", dest);
6    }
7
8    int main()}
9    {
10       char source[10]="";
11       memset(source, 'B', 9);
12       source[9]="\0";
13       Print_Prefix(source);
14   }
```

Program slice for criterion (strcat(), {dest, source}):

```
10   char source[10]="";
11   memset(source, 'B', 9);
12   source[9]="\0";
 3   char dest[5]="A";
 4   strcat(dest, source);
```



cc BY NC SA

Results

- Created dataset of C++ code samples (from NIST NVD and SARD) with buffer overflow vulnerabilities
- Derived program slices from code samples (for set of potentially vulnerable functions, eg. strcpy)
- Trained GLICE model
- Experimental results
  - comparison against original FUNDED model
  - evaluate trade-off call-tree depth vs. detection performance vs. resource usage

	FUNDED Original	FUNDED Improved	GLICE
Precision	0.7857	0.8133	0.9991
Recall	0.9961	0.9928	0.9980
F1-score	0.8784	0.8941	0.9986
Accuracy	0.8621	0.8824	0.9986

- Detection accuracy of GLICE improves up to 13% when compared to FUNDED, while training time for GLICE model is about 9 times smaller (target depth 4)

Target depth	0	1	2	3	4
Precision	0.8128	0.9508	0.9664	0.9822	0.9991
Recall	0.9943	0.9922	0.9930	0.9937	0.9980
F1-score	0.8944	0.9709	0.9795	0.9879	0.9986
Accuracy	0.8826	0.9703	0.9792	0.9878	0.9986

cc BY NC SA

Thanks!

Any questions?

Contact:

✉ harald.vranken@ou.nl

🕸 www.open.ou.nl/hvr

cc BY NC SA