

Automated Vulnerability Detection in Source Code Using Deep Representation Learning

Rebecca L. Russell, Louis Kim , Lei H. Hamilton , Tomo Lazovich,
Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, Marc W.
McConley

Toronto Deep Learning Series, 3rd December, 2018

Presented by:

Alex Hesammohseni, Angshuman Ghosh, Loblaw Digital

Outline

- ▶ Background
- ▶ Existing Solutions
- ▶ Datasets
- ▶ Preprocessing
- ▶ Methods
- ▶ Experiments and Results
- ▶ Discussion Points

Background

Motivation

- ▶ Majority of the computing cycles are spent running C or C++ code
 - ▶ Not memory-safe and extremely weak guarantees offered by type systems
- ▶ Rust/ML and other type-safe system languages still far from replacing millions of lines of C
 - ▶ But there is light at the end of the tunnel

Examples of software vulnerabilities

- ▶ Array out of bounds access errors:
 - ▶ Putting untrusted input in fixed buffers without checking (form validation), Decompression - potentially unbounded output/input size ratio (libtiff, libjpeg, zip bombs), Specialty VM's and stack machine executions (font rendering, TTF hinting)
- ▶ Interaction of undefined behaviour with compiler optimizations:
 - ▶ Arithmetic overflow check or null pointer checking or array bounds checks being optimized out by overzealous compilers
- ▶ Violation of high-level software contracts:
 - ▶ Double-free, use after free, deadlocks, TCP sockets code, race conditions, Unix signals

Existing approaches

Static Analysis

Analyzing source without running it

Caveat:

Rice's theorem (an extension of undecidability of Halting problem) says that checking any non-trivial property of a program (memory safety, etc.) is undecidable (no universal algorithmic solution exists)


*Clang Static Analyzer: <https://clang-analyzer.llvm.org/>

Symbolic Execution

- Replace concrete with symbolic values
- Work backwards from error states using symbolic analysis to find conditions on inputs that leads to undesirable states
- Condition complexity can easily explode

An Illustrative Example

```
int foo(int v) {  
    return 2*v;  
}  
  
void test_me(int x, int y) {  
    int z = foo(y);  
    if (z == x)  
        if (x > y+10)  
            ERROR;  
}
```



Concrete Execution		Symbolic Execution
concrete state	symbolic state	path condition
$x = 2$	$x = x_\theta$	$2*y_\theta == x_\theta$
$y = 1$	$y = y_\theta$	
$z = 2$	$z = 2*y_\theta$	$x_\theta \leq y_\theta + 10$
Solve: $(2*y_\theta == x_\theta)$ and $(x_\theta > y_\theta + 10)$		

Dynamic Analysis

- Analyzing code through running and instrumenting it and looking for correctness/error behaviour
- E.g. Naïve fuzz testing
 - Huge state space and hard to gauge coverage
- Profile-guided fuzzing:
 - AFL - branch instrumentation to cover all CFG basic blocks
- Drawbacks: resource heavy - takes significant processing time and memory.
 - Due to instrumentation overhead and exponential state explosion

- 1 load user-supplied initial test cases into the queue
- 2 take next input file from the queue
- 3 attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program
- 4 repeatedly mutate the file using a balanced and well-researched variety of traditional fuzzing strategies
- 5 if any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue
- 6 go to (2)



Dataset

Common Weakness Enumeration

- ▶ List of common software weaknesses
- ▶ Unified terminology provides common ground for discussions regarding software security
- ▶ Allows software vendors to quantify the security characteristics of their software in a uniform manner

Vulnerabilities

CWE ID	Description	Percentage
120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	29.2%
119	Improper Restriction of Operations within the Bounds of a Memory Buffer	12.7%
469	Use of Pointer Subtraction to Determine Size	11.8%
476	NULL Pointer Dereference	10.7%
805	Buffer Access with Incorrect Length Value	5.4%
362	Concurrent Execution using Shared Resource with Improper Synchronization	3.6%
20, 234, 457, etc	Improper Input Validation, Failure to Handle Missing Parameter, Use of Uninitialized Variable, etc.	26.5%

Dataset

	SATE IV	Debian	GitHub
Total	121,353	2,806,469	9,532,081
Passing curation	12,001	380,381	955,683
Not vulnerable	6,559 (55%)	364,306 (96%)	907,186 (95%)
Vulnerable	5,442 (45%)	16,075 (4%)	48,497 (5%)

- Individual functions from public repositories and vulnerability testing datasets
- SATE IV: Generated dataset for comparison of static analysis tools
- Debian & GitHub: Labelled using existing static analysis tools
- Pre-processing: Custom lexical analysis followed by deduplication ('curation') at function level

Labelling

- ▶ SATE IV functions are labeled
- ▶ For GitHub & Debian, functions are labeled using existing static analysis tools
- ▶ Static analyzers used: [CLANG](#), [CppCheck](#), [FlawFinder](#)
- ▶ Researchers mapped outputs from analyzers to CWE issues
- ▶ Prevalence of security vulnerabilities: 5.1%

Preprocessing

Lexical Analysis

- ▶ Non-Code entities (whitespace, comments, etc.) are stripped
- ▶ C/C++ keywords retained as is
- ▶ String/char/float literals replaced using placeholders. Integers tokenized at digit level.
- ▶ Data types are homogenized - e.g. u32, uint32, etc replaced using one lexicon
- ▶ Vocabulary - 156 unique tokens.
- ▶ Drawbacks -
 - ▶ No ambiguity resolution (e.g. Most Vexing Parse)
 - ▶ Loss of information from literals, datatypes

Data Curation

► Intuition:

- Functions that create the exact same tokens after lexical analysis/program compilation likely to cause over-fitting if they exist both in training and test sets

► Solution:

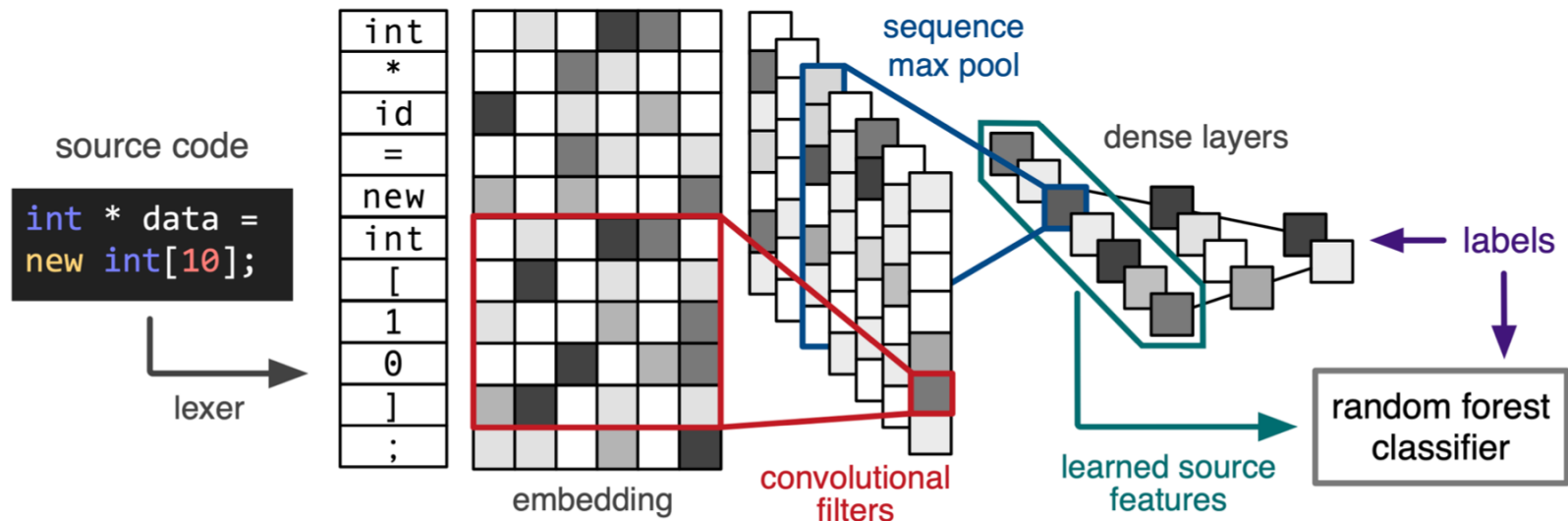
- Remove functions that are duplicates after custom lexical analysis
- Remove functions that have the same CFG shape and basic-block operator-vector composition after compilation

Methods

Learning Embeddings

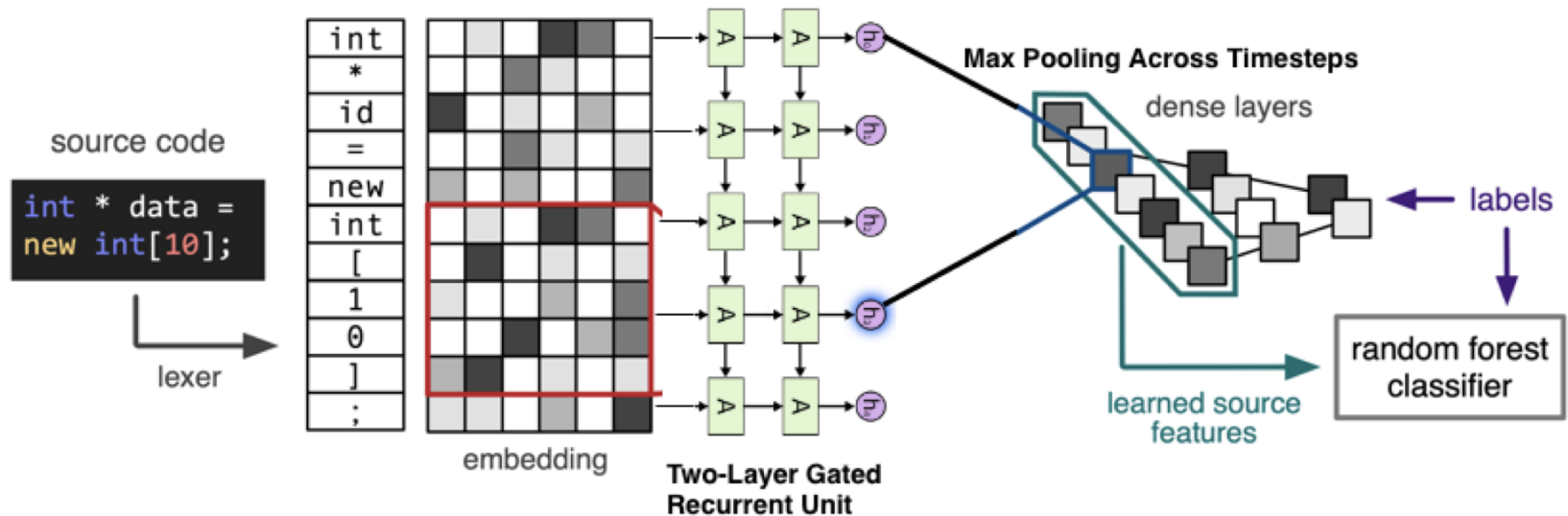
- ▶ **Learn** k-dimensional embeddings on output of lexer
- ▶ Seeded randomly
- ▶ Overfitting prevented by adding Gaussian Noise
$$\mathcal{N}(\mu = 0, \sigma^2 = 0.01)$$
- ▶ Value of k experimentally set to 13
 - ▶ Low value for k works because there are only 156 tokens in total
- ▶ Learned during training of network

Convolutional Feature Extraction



- Filters: $m \times k$ filters used
- k = dimension of embedding, m = width of receptive field
- Number of filters = 512, $m = 9$ (experimentally determined)
- Batch Normalization and activation using ReLU

Recurrent Feature Extraction



- Gated Recurrent Units used for sequence modelling
- Two layers of GRUs are stacked
- Hidden state size is 256
- Output at each step concatenated

Max Pooling and Dense Layers

- ▶ Convolutional Feature Extraction:
 - ▶ Sequence max pooling: Take the highest activation for each convolution filter along the sequence dimension
- ▶ Recurrent Feature Extraction:
 - ▶ Max Pooling Across Time: Store activations for each time-step, take max
- ▶ Dense Layer
 - ▶ Dropout: 50% dropout used for connection between pooling and dense layers
 - ▶ Two dense layers 64 and 16 neurons each used
 - ▶ Softmax Output Layer used for prediction

Training

- ▶ Functions with token length between 10 and 500 used, zero-padded to 500
- ▶ Training batch size: 128
- ▶ Optimization: ADAM
 - ▶ Convnet learning rate: 5×10^{-4}
 - ▶ GRU learning rate: 1×10^{-4}
- ▶ Loss: Cross Entropy
- ▶ Loss weightage - due to class imbalance, cross-validated loss weightage used
- ▶ Dataset Segmentation: 80 Train; 10 Validate; 10 Test

Experiments and Results

Experiments

► Methods Compared:

- Baseline Bag-of-words with Random Forest
- Lexed Bag-of-words with Random Forest
- RNN
- CNN
- RNN features with Random Forest
- CNN features with Random Forest

► Metrics

- Area under ROC curve
- Area under Precision Recall Curve (higher the better) (max 1)
- Matthews Correlation Coefficient

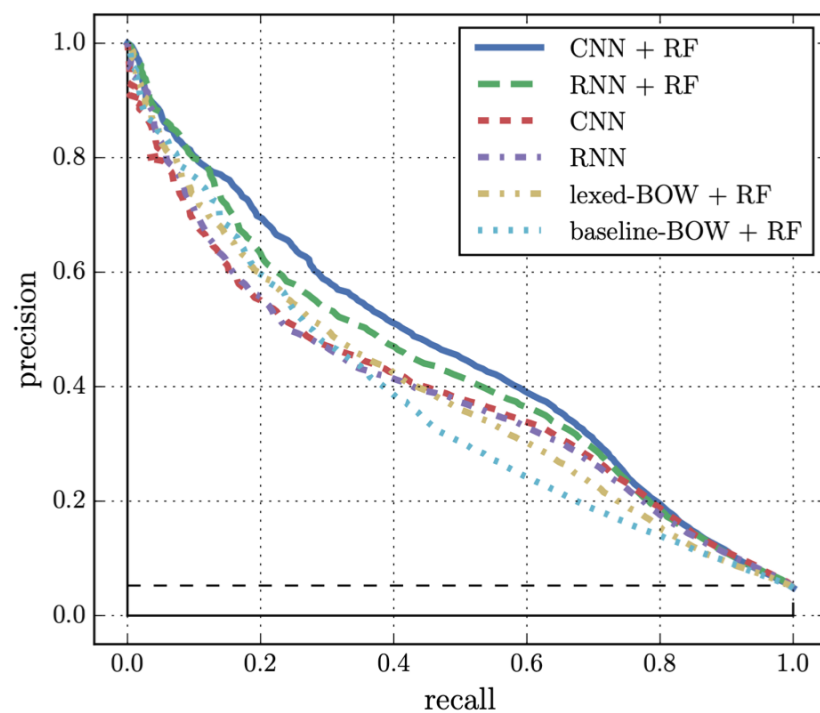
$$\text{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

► $-1 \leq MCC \leq 1$

- F_1 score

	PR AUC	ROC AUC	MCC	F_1
baseline-BOW + RF	0.368	0.853	0.352	0.365
lexed-BOW + RF	0.389	0.865	0.391	0.419
RNN	0.389	0.880	0.407	0.429
CNN	0.391	0.884	0.413	0.435
RNN + RF	0.431	0.885	0.431	0.456
CNN + RF	0.456	0.888	0.451	0.477

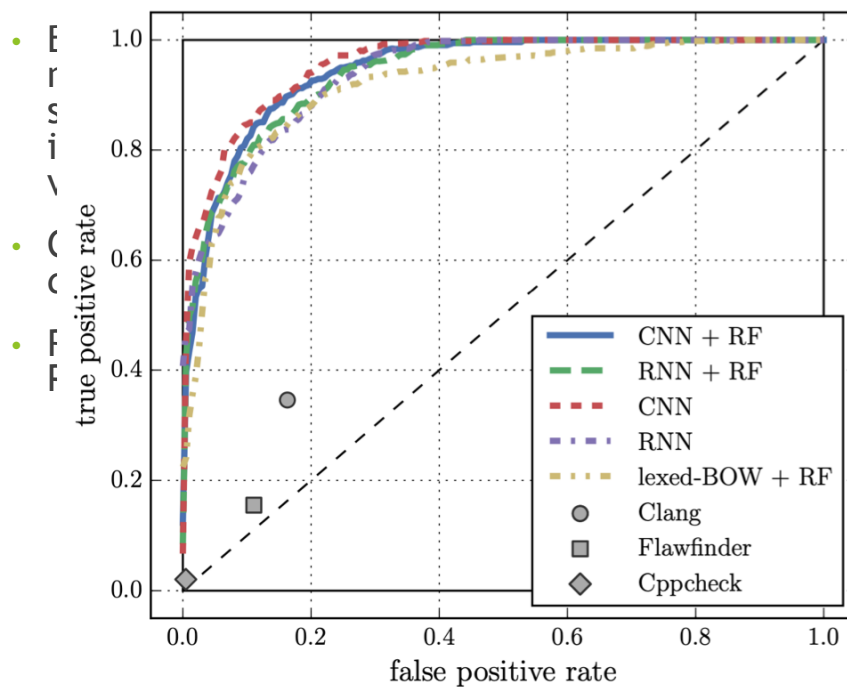
Evaluation on packages from Debian and GitHub



	PR AUC	ROC AUC	MCC	F_1
Clang	—	—	0.211	0.448
Flawfinder	—	—	0.065	0.241
Cppcheck	—	—	0.073	0.040
lexed-BOW + RF	0.905	0.918	0.579	0.785
RNN	0.926	0.937	0.647	0.815
CNN	0.950	0.958	0.696	0.838
RNN + RF	0.932	0.941	0.671	0.827
CNN + RF	0.933	0.945	0.682	0.831

- For SATE IV, since labels are available, comparison is

Evaluation on SATE IV Juliet Test Suite



Interactive Output

```
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;
    }
}
```

- Activations from Convolutions used to highlight parts of code which is likely to have vulnerabilities. This increases usability of tool for developers.

Discussion Points

Discussion Points

Lack of analysis of custom lexer

- ▶ No concrete analysis is provided to support claims about the lexer used in the paper.
- ▶ Each contribution should be tested against a hypothesis: authors make assumptions to build lexer without any validation backed by data

Discussion Points

Lack of verification for labels

- ▶ For GitHub, Debian existing static analyzers were used to generate labels
- ▶ Study on SATE IV show low precision/recall for these analyzers at vulnerability detection
- ▶ Thus, first experiment basically shows how well the CNN can approximate the result of the combined output of the open source static analyzers

Discussion Points

Lack of qualitative analysis of results

- ▶ Typically, positive results from the methods which do not appear in the golden set are manually inspected to ensure correct labeling
- ▶ Neither humans, nor OS static analyzers have perfect accuracy
- ▶ Whether or not a method can lead to discovery of new as yet unknown vulnerabilities is a good test of the quality of analyzers.
- ▶ In the absence of such an analysis, it is really hard to know what the true performance is