Towards Security Defect Prediction with AI

In this study:
- We investigate the limits of the current state-of-the-art AI system for detecting buffer overflows and compare it with current static analysis tools.
- We develop a code generator, sa-bAbI, capable of producing an arbitrarily large number of code samples of controlled complexity.

Static analysis tools considered:
- Clang – Based on symbolic execution and, by default, uses unsound heuristics such as loop unrolling to contend with state space explosion.
- Cppcheck – We believe it also uses unsound heuristics, though little has been published about its specific approach.
- Anonymized commercial tool – Well known to be unsound.

sa-bAbI generator
- Modeled after bAbI from Weston et al. 2015, [1]
- Intentionally very simple
  - Valid C code
  - Conditionals
  - Loops
  - Unknown values such as rand()
- Complements existing software assurance datasets for training AI
- Will be included in NIST SARD

A memory network based on Choi et al., 2016 [2]

Input:
- A program code \(X \in \mathbb{N} \times \mathbb{J}\), consisting of \(N\) lines \(X_1, \ldots, X_N\), where each line \(X_i\) is a list of integer tokens \(w_{i1}, \ldots, w_{iD}\)
- A query line \(q \in \mathbb{J}\), equal to one of the lines \(X_i\), encoding a buffer write

Embedding: We fix an embedding dimension \(d\) and establish two learnable embedding matrices \(E_{val}\) and \(E_{addr}\), both of dimension \(V \times d\). Letting \(A\) represent both \(E_{val}\) and \(E_{addr}\), we encode each integer token twice, letting \(Aw_i\) \(\in \mathbb{R}^d\) be the \(w_i\)-th row of \(A\). For \(i = 1, \ldots, N\), define \(m_i \in \mathbb{R}^d\) by

\[
    m_i = \text{Dropout}(W_{1:}(\sum_{j=1}^J I^q_j \cdot Aw_i))
\]

\[
    I^q_j = \frac{1 - j/J}{1 - j/J}
\]

We store the lines \(m_i\) encoded by \(E_{val}\) in a matrix \(M_{val}[N \times d]\), and store the lines encoded by \(E_{addr}\) in a matrix \(M_{addr}\). We embed the query line \(q\) by \(E_{addr}\) and store the result in \(u^s \in \mathbb{R}^d\).

Memory search: For each “hop number” \(h = 1, \ldots, H\) in a fixed number of “hops” \(H\):

\[
    p[N \times 1] = \text{softmax}(M_{addr}u^H)
\]

\[
    o[N \times d] = \sum_{i=1}^N p_i(M_{val})
\]

\[
    (s) r[N \times d] = R_0
\]

\[
    (s) s[N \times d] = \text{Norm}(r)
\]

\[
    u^{N+h}[N \times d] = u^h + s
\]

where \(R_0[d \times d]\) is an internal learnable weight matrix.

Classification:

\[
\hat{y}[2 \times 1] = \text{softmax}(W(u^H)^T)
\]

where \(W[2 \times d]\) is a learnable weight matrix.

The forward pass is effectively an iterative inner-product search matching the current query line \(v^q\) which changes with each processing hop, against each line \(m_i\) of the stored memory, which remains fixed.

We found:
- Static analysis engines have good precision but poor recall on our dataset.
- The state-of-the-art AI system can achieve similar performance to the static analysis engines, but it requires an exhaustive amount of training data to do so.

Our future work:
- Using representations of code that can capture appropriate scope information.
- Using deep learning methods that are able to perform arithmetic operations.

Example Code

```c
#include <stdlib.h>

int main() {
    int entity_4 = 0;
    while(entity_4 < entity_0_1) {
        entity_0_1 = 0;
        entity_1[entity_0_1]  = 's';
        entity_2[entity_0_1] = 's';
        entity_3[entity_0_1] = 's';
        entity_4 = 78;
        entity_5 = 79;
        entity_6 = 79;
        entity_7 = 79;
        entity_7 = 79;
        entity_8 = 79;
        entity_9 = 79;
        entity_0_1 = 0;
        entity_0_1 = 0;
        entity_0_1 = 0;
        entity_4 = 78;
        entity_5 = 79;
    }
    return 0;
}
```

