DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing

Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin
Motivation

Binary Code Differential Analysis

- quantitatively measure the similarity between two given binaries
- produce the fine-grained basic block level matching
Motivation

vulnerability analysis [ICSE’17]
plagiarism detection [FSE’14]
exploit generation [NDSS’11]
Existing Techniques

**Static Approaches:**
- Bindiff, Binslayer [PPREW’13], Tracelet [PLDI’14], CoP [ASE’14], Pewny et al. [SP’15], discovRE [NDSS’16], Esh [PLDI’16]

**Dynamic Approaches:**
- iBinHunt [ISC’12], Blanket Execution [USENIX SEC’14], BinSim [USENIX SEC’17]

Slow runtime performance
Inaccurate matching
Poor code coverage
Existing Techniques

Learning-based Approaches:

- **Genius [CCS’16]**
  - traditional machine learning
  - function matching
- **Gemini [CCS’17]**
  - deep learning based approach
  - manually crafted features
  - function matching
- **InnerEye [NDSS’19]**
  - basic block comparison
  - instruction semantics by NLP
- **Asm2vec [SP’19]**
  - token and function semantic info by NLP
  - function matching
Existing Techniques

Limitations of Learning-based Approaches:

- No efficient binary differencing at basic block level
  - InnerEye takes 0.6ms to compare one pair of basic blocks
  - Millions of basic block comparisons for binary differencing
- No program-wide dependency information
  - What if the two binaries contain multiple similar basic blocks
- Heavily rely on labeled training data
  - Extreme diversity of binaries
  - Overfitting problem
Problem Definition

Given two binaries $p_1 = (B_1, E_1)$ and $p_2 = (B_2, E_2)$, find the optimal basic block matching that maximizes:

$$SIM(p_1, p_2) = \max_{m_1, m_2, ..., m_k \in M(p_1, p_2)} \sum_{i=1}^{k} sim(m_i),$$

where:

- $B_1 = \{b_1, b_2, ..., b_n\}$ and $B_2 = \{b'_1, b'_2, ..., b'_m\}$ are two sets containing all the basic blocks in $p_1$ and $p_2$;
- Each element $e$ in $E \subseteq B \times B$ corresponds to control flow dependency between two basic blocks;
- Each element $m_i$ in $M(p_1, p_2)$ represents a matching pair between $b_i$ and $b_j$;
- $sim(m_i)$ defines the quantitative similarity score between two matching basic blocks.
Problem Definition

- **Our goal**: Solve the binary diffing problem
  - *a. sim(mi)*: leveraging both the token (opcode and operand) semantics and program-wide contextual info to calculate similarity
  - *b. M(p1,p2)*: efficient basic block matching

- **Assumptions**
  - only stripped binaries
  - compiler optimization techniques applied
  - same architecture
Our solution: DeepBinDiff

- DeepBinDiff program-wide contextual info learning
- Complete unsupervised learning approach
- Calculate $sim(mi)$
- Efficient matching $M$
- Semantic info learning
- Program-wide contextual info learning

Diagram:
- Input: binary 1, binary 2
- Pre-processing: CFG generation, token embedding model
- Embedding Generation: token embedding, feature vectors
- Code Differing: k-hop greedy matching
- Output: differencing results

Cloud: Complete unsupervised learning approach
Learning Token Semantics

- Token semantic info
  - each instruction: **opcode** + potentially multiple **operands**
  - represented as token embeddings, learned by leveraging NLP technique
  - aggregated to generate feature vector for each basic block

\[
FV_b = \sum_{i=1}^{j} (\text{embed}_{p_i}^{*} \text{weight}_{p_i}^{\|} \frac{1}{|\text{Set}_{t_i}|} \sum_{n=1}^{k} \text{embed}_{t_{i,n}}^{})
\]

- **embedding for opcode**
- **TF-IDF model**
- **embeddings for operands**
Learning Token Semantics

embedding for opcode
cmp: [0.03, 0.16, 1.92, ...]

embeddings for normalized operands
im: [0.62, -0.125, 0.76, ...]
reg1: [1.5, 1.6, -0.92, ...]

TF-IDF model

weighted embedding
[0.01, 0.0528, 0.63, ...]

embedding for instruction
[0.01, 0.0528, 0.63, ...2.12, 1.475, -0.16]
Learning Semantics Info

**Step 1: Random Walks**

```
movzx ecx, byte ptr [rdx]
mov r8d, eax
ja 0x408963
mov rax, rdx
add rax, 1
movzx esi, byte ptr [rax]
lea edi, dword ptr [rsi - 0x30]
cmp dil, 9
jbe 0x407fa6
mov r14, -1
cmp sil, 0x24
jne 0x408033
cmp r8b, 9
ja 0x4088e7
...```

**Step 2: Normalization**

```
mov reg4, ptr
mov reg4, reg4
ja im
mov reg8, reg8
add reg8, im
mov reg4, ptr
lea reg4, ptr
cmp reg1, im
jbe im
mov reg8, im
cmp reg1, im
jne im
cmp reg1, im
ja im
...```

**Step 3: Model Training**

```
Softmax Classifier

Hidden Layer

Current Instruction

Context | Target | Context

lea reg4, ptr  cmp reg1, im  jbe im```

**Step 4: Feature Vector Generation**

```
jbe : 0.015, 0.006, -0.22, ...
im : 0.071, -0.14, 0.005, ...
ptr : 0.022, 0.065, 0.04, ...
reg4 : 0.15, 0.13, -0.043, ...
token embeddings

(0.035, 0.16, 0.032, ...)
(0.12, 0.04, -0.009, ...)
(0.411, -0.2206, 0.4, ...)
(0.55, 0.656, 0.33, ...)```

aggregation
Learning Program-wide Contextual Info

- Program-wide contextual info
  - useful for differentiating similar basic blocks in different contexts
  - learned from inter-procedural CFG
  - leverage Text-associated DeepWalk algorithm (TADW)
Learning Program-wide Contextual Info

- Now that we have two ICFGs
  - merge two ICFGs into one
  - learning algorithm runs only once
  - embeddings can be comparable
  - boost the similarity
  - graph structure stays unchanged
Learning Program-wide Contextual Info

- contain both semantic info and contextual info
- used to calculate basic block similarity
- solve \( \text{sim(mi)} \)

feature vector

\[
\begin{align*}
0.053, 0.16, 0.032 \ldots \\
0.12, 0.44, -0.009 \ldots \\
0.411, -0.2206, 0.4 \ldots \\
0.55, 0.656, 0.33 \ldots 
\end{align*}
\]

merged graph

TADW algorithm

basic block embeddings

\[
\begin{align*}
0.055, 0.004, -0.07 \ldots \\
0.07, -0.314, 0.305 \ldots \\
0.335, -0.93, 0.1189 \ldots \\
-1.8e-06, 0.092, 0.06 \ldots 
\end{align*}
\]
Code Diffing: $k$-hop greedy matching

- Goal: Given two input binaries $p_1$ and $p_2$, find optimal matching $M(p_1,p_2)$.

Initially, $\text{matching\_set} = \{(a, 1)\}$

- find $k$-hop neighbors of a matching pair
  - $1\text{hn}(a) = \{b,c\}$
  - $1\text{hn}(1) = \{2,3\}$
- use basic block embeddings to calculate similarities among $1\text{hn}(a)$ and $1\text{hn}(1)$
- find most similar pair (must be above a threshold), put it into $\text{matching\_set}$
- run the process iteratively
- use linear assignment algorithm for unmatched ones
Evaluation

● Dataset
  ○ C binaries:
    ■ Coreutils, Diffutils, Findutils
    ■ Multiple versions (5 for Coreutils, 4 for Diffutils, and 3 for Findutils)
    ■ 4 different compiler optimization levels (O0, O1, O2 and O3)
  ○ C++ binaries:
    ■ 2 popular open-source projects (10 binaries)
    ■ contain plenty of virtual functions
    ■ 3 versions for each project, compile with default optimization levels
  ○ Case study
    ■ 2 real-world vulnerabilities in OpenSSL

● The most comprehensive evaluation for cross-version and cross-optimization-level binary diffing.
Evaluation

● Baseline techniques
  ○ De-facto commercial tool
    ■ BinDiff
  ○ State-of-the-art techniques
    ■ Asm2Vec + k-hop
    ■ InnerEye + k-hop
      • only used to evaluate a subset of binaries
  ○ Our tool without contextual info
    ■ DeepBinDiff-ctx
Evaluation - Cross-version differencing

- Outperform the de facto commercial tool by 23% and 7% in recall and precision
- Outperform state-of-the-art technique by 11% and 22% in recall and precision
- Contextual info is proven to be very useful
Evaluation - Cross-version differencing

(a) v5.93 compared with v8.30  (b) v6.4 compared with v8.30

(c) v7.6 compared with v8.30  (d) v8.1 compared with v8.30
Evaluation - Cross-optimization level diffing

- Outperform the de facto commercial tool by 28% and 5% in recall and precision
- Outperform state-of-the-art technique by 18% and 19% in recall and precision
Evaluation - Cross-optimization level diffing
Evaluation - Case study

handle function inlining
Evaluation - Case study

handle basic block insertion/deletion
Discussion - Compiler Optimizations

- **Instruction scheduling**
  - choose not to use sequential info

- **Instruction replacement**
  - NLP technique to distill semantic info

- **block reordering**
  - treat ICFG as undirected graph when matching

- **function inlining**
  - generate random walks across function boundaries
  - avoid function level matching
  - $k$-hop matching is done upon ICFG rather than CFG

- **register allocation**
  - register name normalization
Summary

- A novel unsupervised program-wide code representation learning technique
- \(k\)-hop greedy matching algorithm for efficient matching
- Comprehensive evaluation against state-of-the-art techniques and the de facto commercial tool
Summary

Open source project: https://github.com/deepbindiff/DeepBinDiff

THANK YOU!