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]

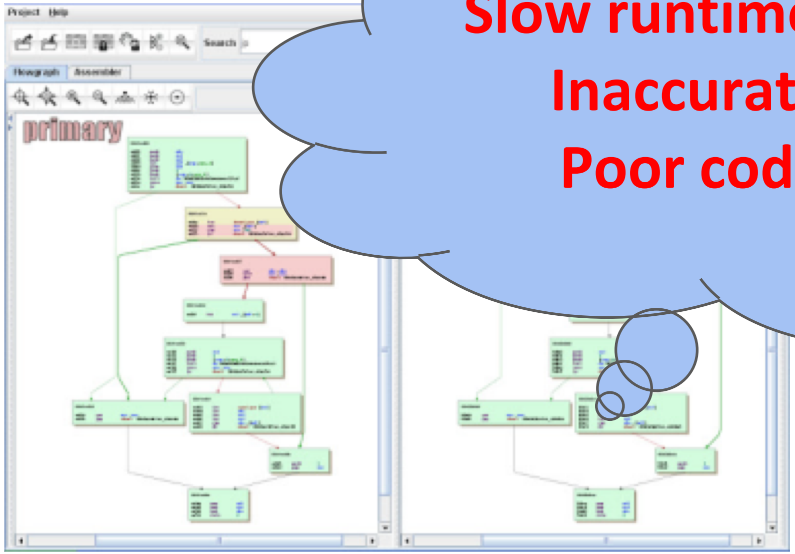
plagiarism detection[FSE'14]

exploit generation [NDSS'11]

Existing Techniques

Static Approaches:

Bindiff, Binslayer [PPREW'13], Tracelet [PLDI'14], CoP [ASE'14], P... discovRE [NDS...



Dynamic Approaches:

iBinHunt [ISC'12]

...USENIX SEC'14]

...'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 diffing at basic block level
  - InnerEye takes 0.6ms to compare one pair of basic blocks
  - millions of basic block comparisons for binary diffing
- 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 p1 = (B1, E1) and p2 = (B2, E2), find the optimal basic block matching that maximizes:

$$SIM(p1, p2) = \max_{m_1, m_2, \ldots, m_k \in M(p_1, p_2)} \sum_{i=1}^{k} sim(m_i), \text{ where:}$$

- $B_1 = \{b_1, b_2, ..., b_n\}$ 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 $\boxed{M(p_1, p_2)}$ represents a matching pair between b_i and $b_j^{'}$;
- $\boxed{sim(m_i)}$ defines the quantitative similarity score between two matching basic blocks.

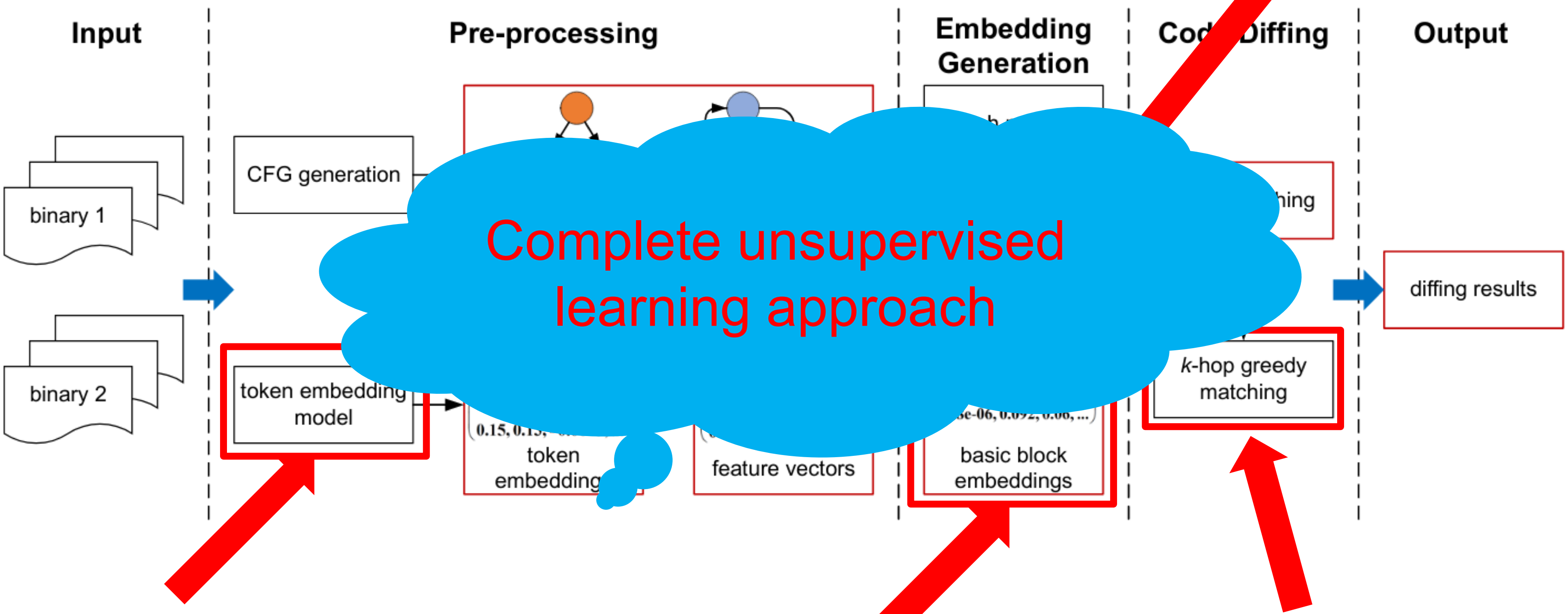
Problem Definition

- **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

program-wide contextual info learning



semantic info learning

calculate *sim(mi)*

efficient matching *M*

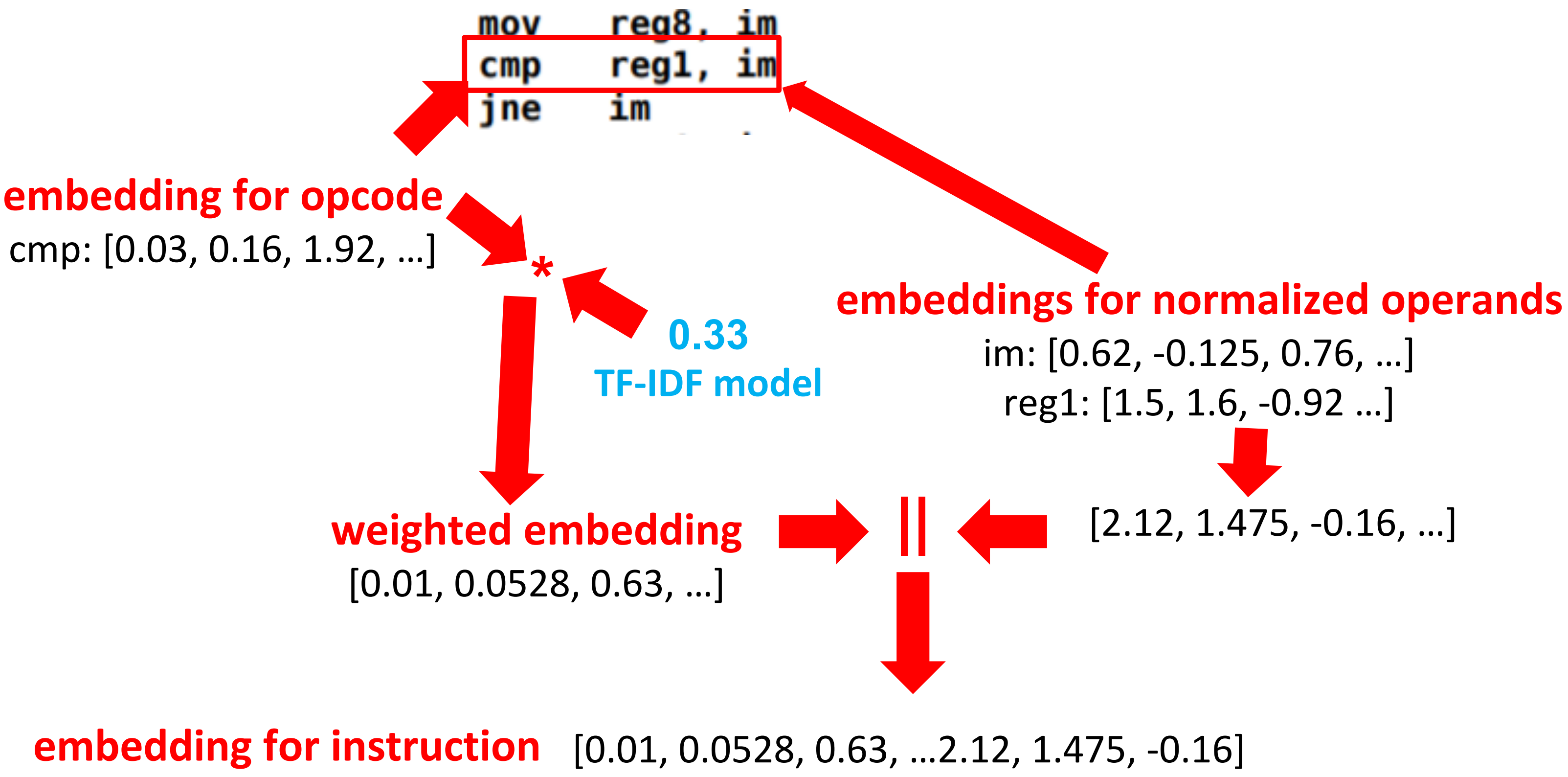
Learning Token Semantics

- 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

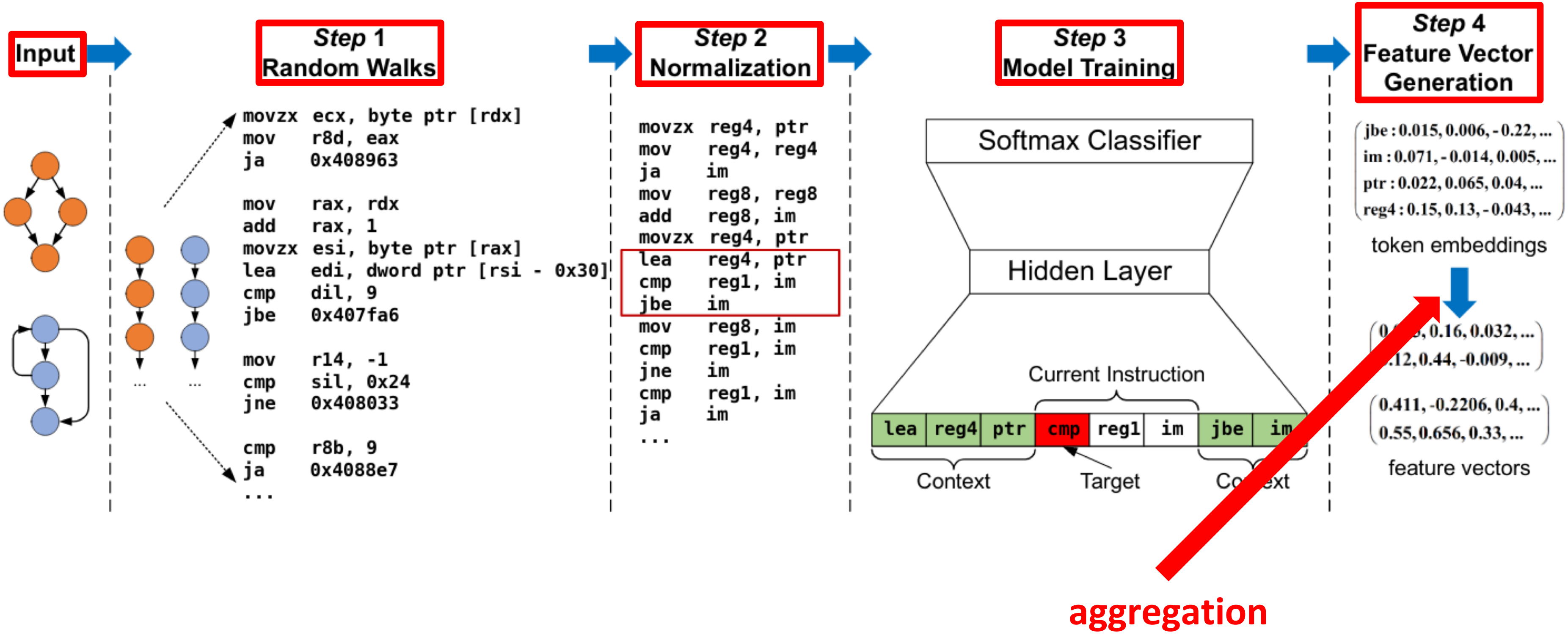
$$FV_b = \sum_{i=1}^{j} \left(\boxed{embed_{p_i}} * \boxed{weight_{p_i}} \boxed{||} \frac{1}{|Set_{t_i}|} * \sum_{n=1}^{k} \boxed{embed_{t_{in}}} \right)$$

embedding for opcode **TF-IDF model** **embeddings for operands**

Learning Token Semantics



Learning Semantics Info

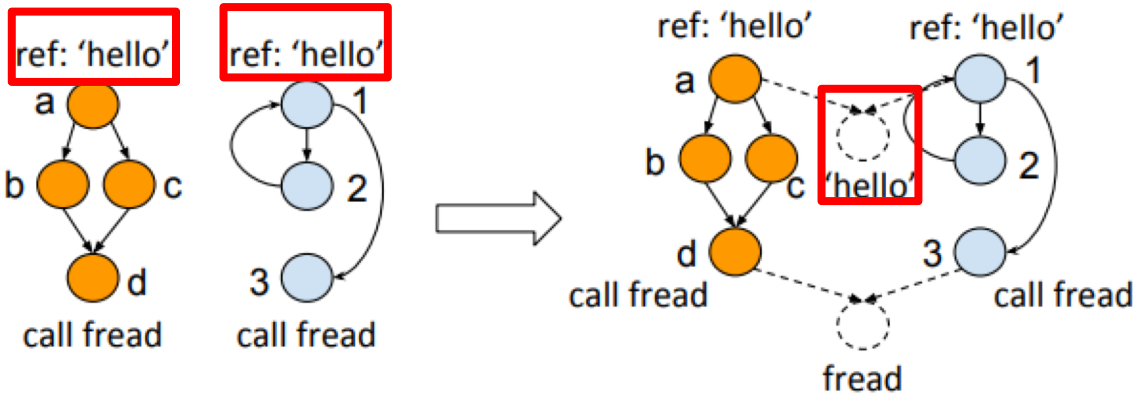


Learning Program-wide Contextual Info

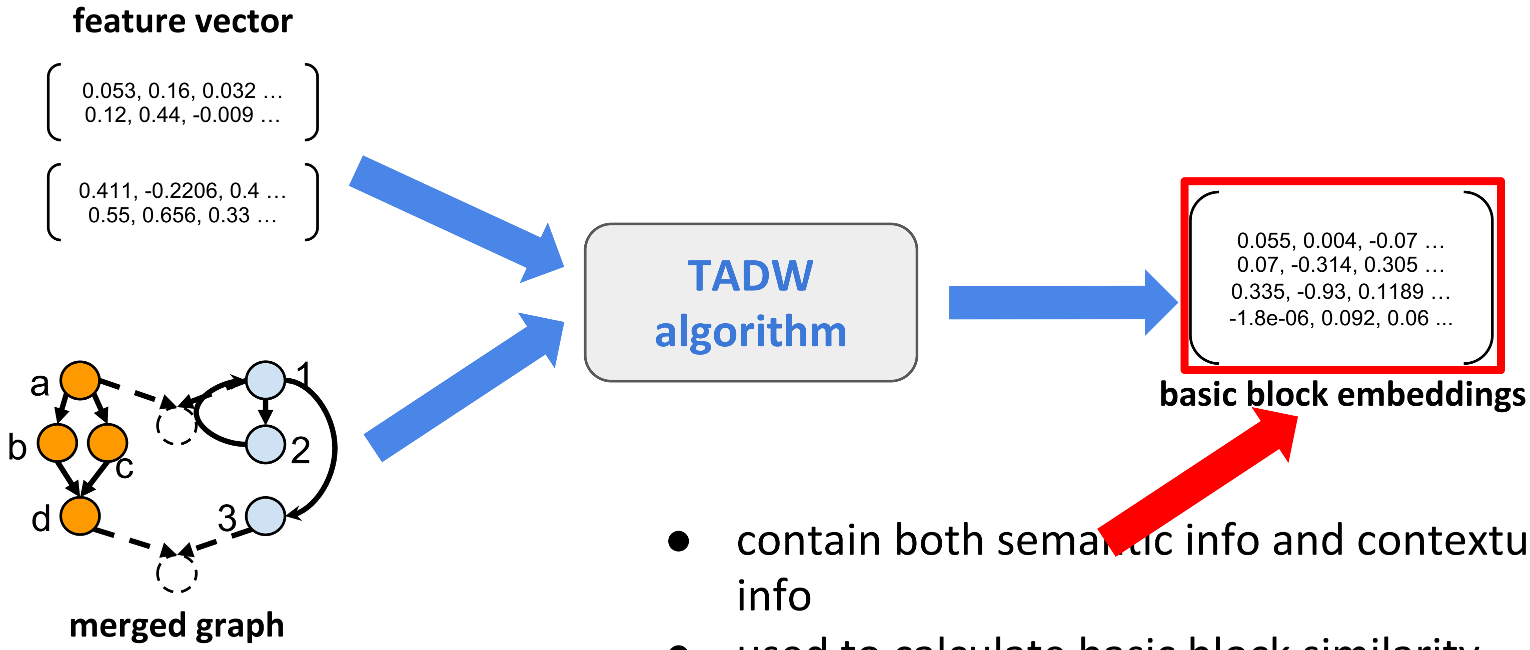
- Program-wide contextual info
  - useful for differentiating similar basic blocks in different contexts
  - learned from inter-procedural CFG
  - leverage Text-associated DeepWalk algorithm (TADW)

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



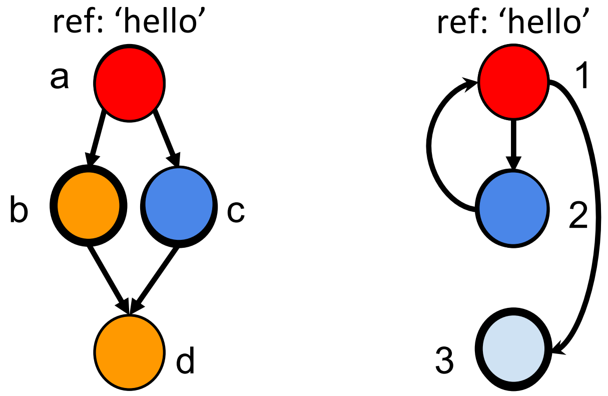
- contain both semantic info and contextual info
- used to calculate basic block similarity
- solve ***sim(mi)***

Code Diffing: *k*-hop greedy matching

- Goal: Given two input binaries p1 and p2, find optimal matching ***M(p1,p2)***.

Initially, matching_set = {(a, 1)}

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



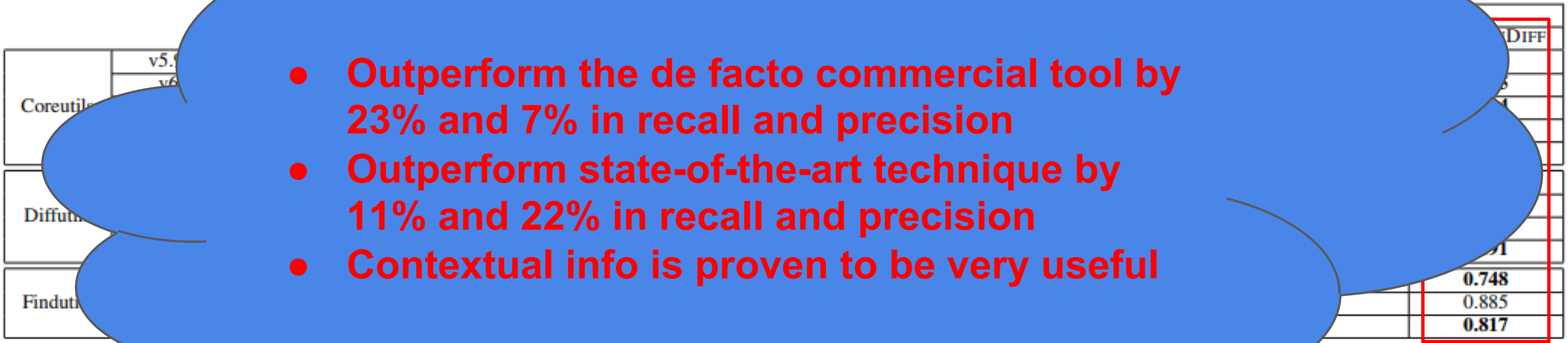
Evaluation

- 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

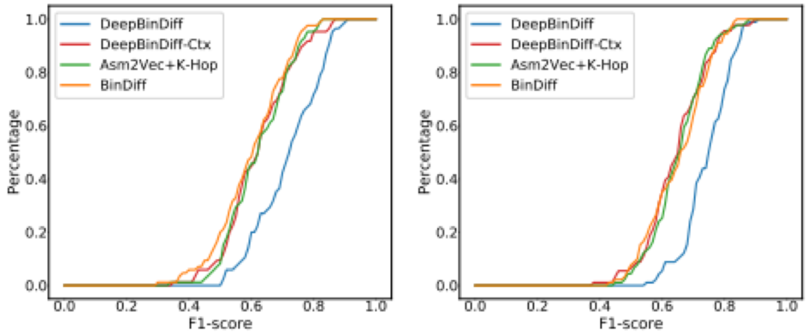
- Baseline techniques
  - De-facto commercial tool
    - BinDiff
  - State-of-the-art techniques
    - Asm2Vec + *k*-hop
    - InnerEye + *k*-hop
      - only used to evaluate a subset of binaries
  - Our tool without contextual info
    - DeepBinDiff-ctx

Evaluation - Cross-version diffing

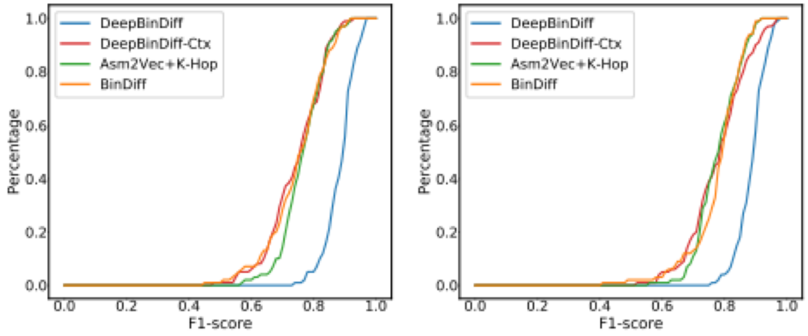


- **Outperform the de facto commercial tool by 23% and 7% in recall and precision**
- **Outperform state-of-the-art technique by 11% and 22% in recall and precision**
- **Contextual info is proven to be very useful**

Evaluation - Cross-version diffing



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



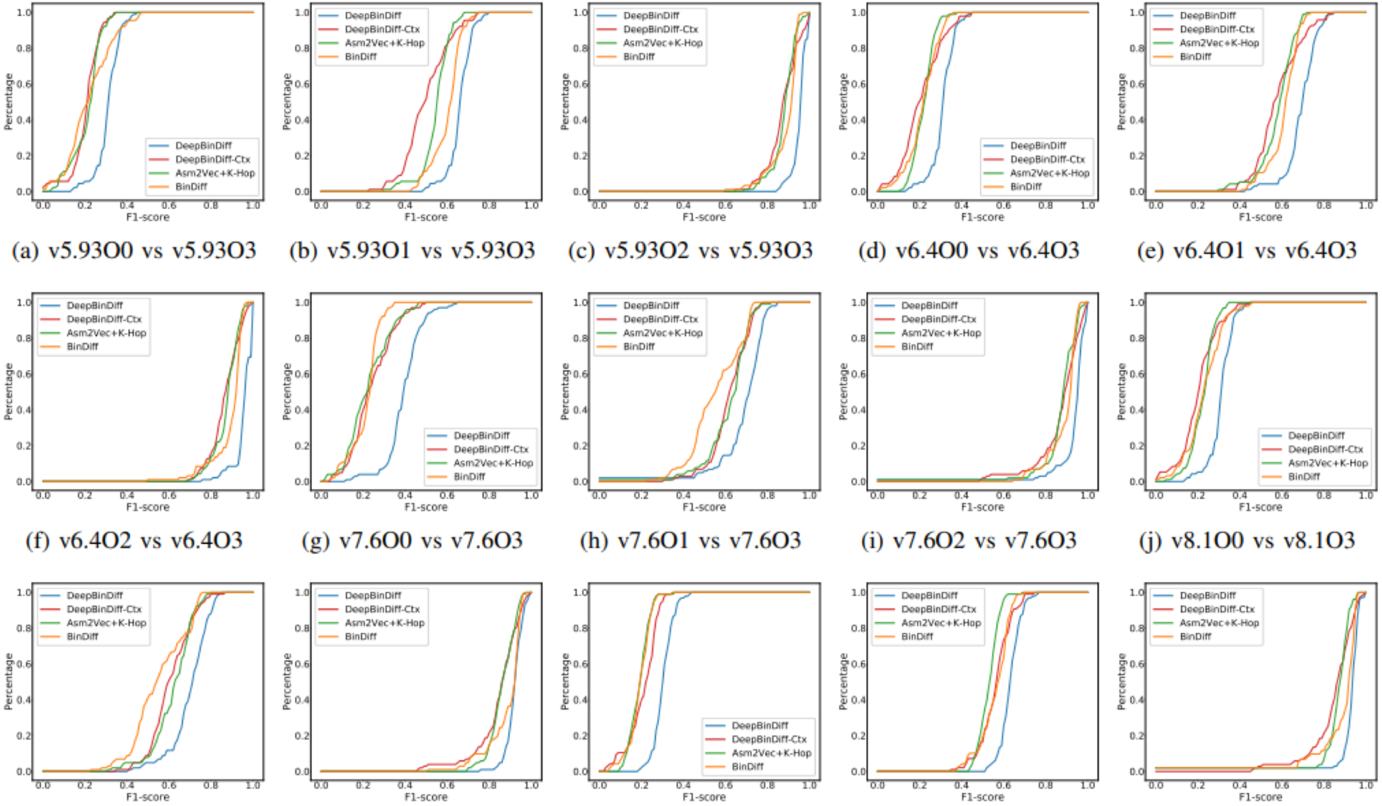
(c) v7.6 compared with v8.30 (d) v8.1 compared with v8.30

Evaluation - Cross-optimization level diffing

		Recall				Precision			
		BinDiff	ASM2VEC+k-HOP	DEEPBINDIFF-CTX	DEEPBINDIFF	BinDiff	ASM2VEC+k-HOP	DEEPBINDIFF-CTX	DEEPBINDIFF
Coreutils	v5.93 O0 - O3	0.176	0.155	0.163	**0.311**	0.291	0.211	0.235	**0.315**
	v5.93 O1 - O3	0.571	0.545	0.497	**0.666**	0.638	0.544	0.515	**0.681**
	v5.93 O2 - O3	0.837	0.911	0.912	**0.975**	[illegible]	[illegible]	[illegible]	[illegible]
	v6.4 O0 - O3	0.166	0.201	[illegible]	**0.391**	[illegible]	[illegible]	[illegible]	[illegible]
	v6.4 O1 - O3	0.576	0.579	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	0.893	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
Findutils	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	**0.985**
	v4.41 O0 - O3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.145	**0.242**
	v4.41 O1 - O3	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	0.678	**0.885**
	v4.41 O2 - O3	0.839	0.917	[illegible]	[illegible]	[illegible]	0.952	0.961	**0.962**
	v4.6 O0 - O3	0.075	0.151	0.139	[illegible]	[illegible]	0.172	0.185	**0.315**
	v4.6 O1 - O3	0.563	0.645	0.627	**0.761**	0.633	0.727	0.705	**0.806**
	v4.6 O2 - O3	0.958	0.935	0.923	**0.957**	0.932	0.914	0.921	**0.957**
	Average	0.545	0.592	0.587	**0.685**	0.609	0.596	0.598	**0.688**

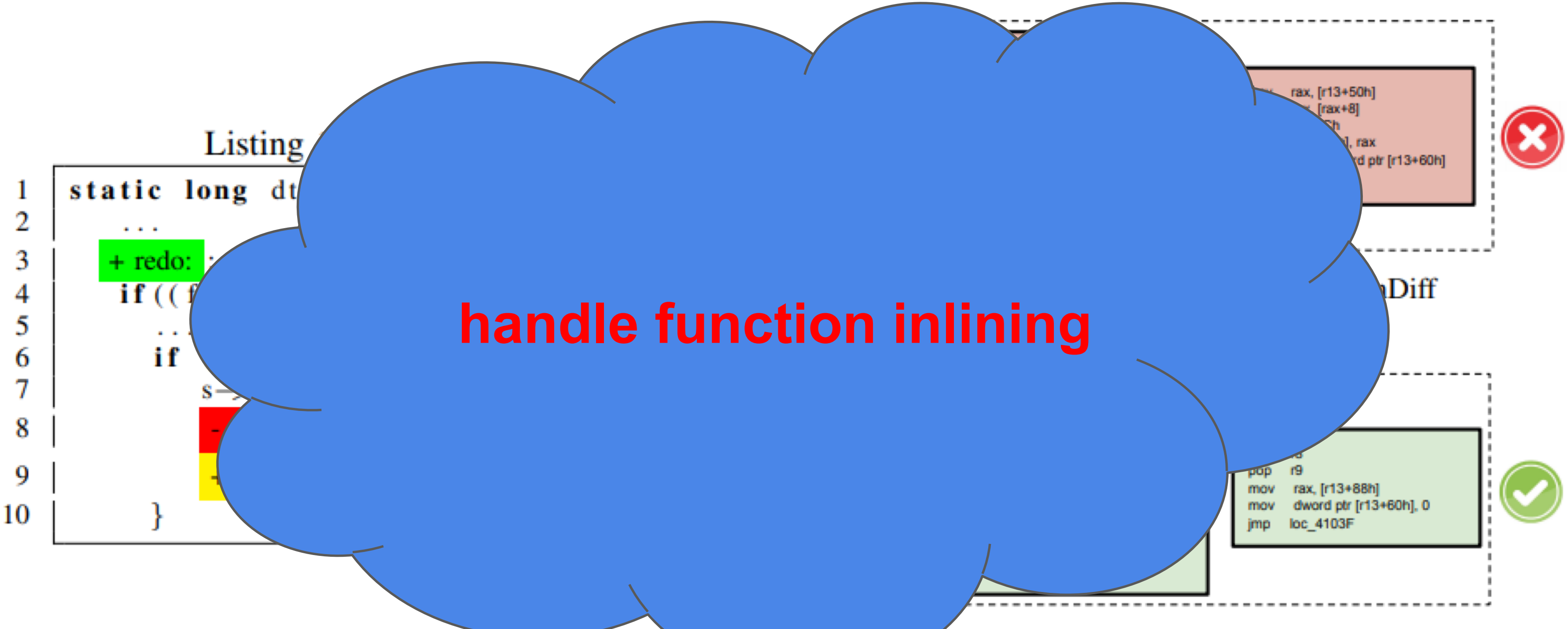
- **Outperform the de facto commercial tool by 28% and 5% in recall and precision**
- **Outperform state-of-the-art technique by 18% and 19% in recall and precision**

Evaluation - Cross-optimization level diffing



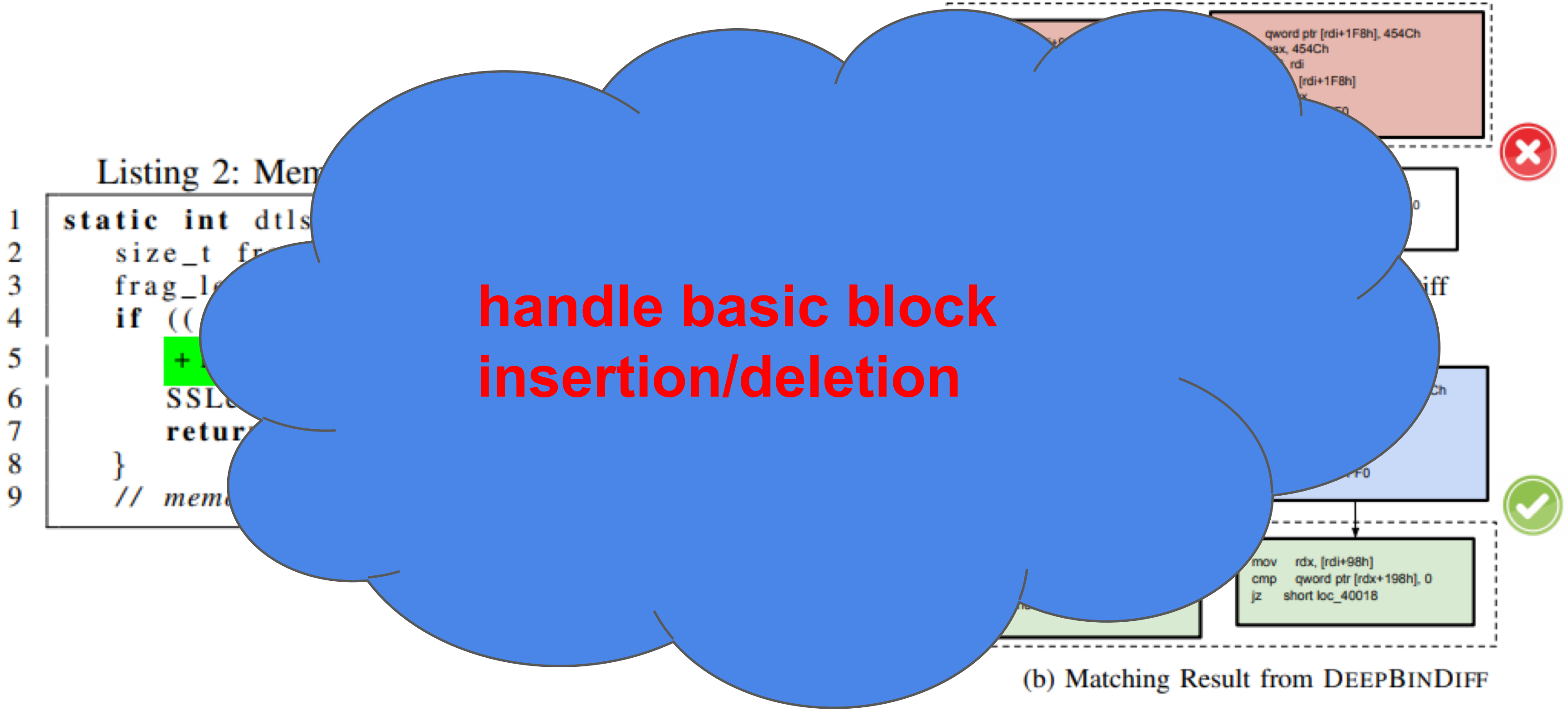
(a) v5.93O0 vs v5.93O3
(b) v5.93O1 vs v5.93O3
(c) v5.93O2 vs v5.93O3
(d) v6.4O0 vs v6.4O3
(e) v6.4O1 vs v6.4O3
(f) v6.4O2 vs v6.4O3
(g) v7.6O0 vs v7.6O3
(h) v7.6O1 vs v7.6O3
(i) v7.6O2 vs v7.6O3
(j) v8.1O0 vs v8.1O3
(k) v8.1O1 vs v8.1O3
(l) v8.1O2 vs v8.1O3
(m) v8.30O0 vs v8.30O3
(n) v8.30O1 vs v8.30O3
(o) v8.30O2 vs v8.30O3

Evaluation - Case study



(b) Matching Result from DEEPBINDIFF

Evaluation - Case study



(b) Matching Result from DEEPBINDIFF

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!