CS 250 Software Security

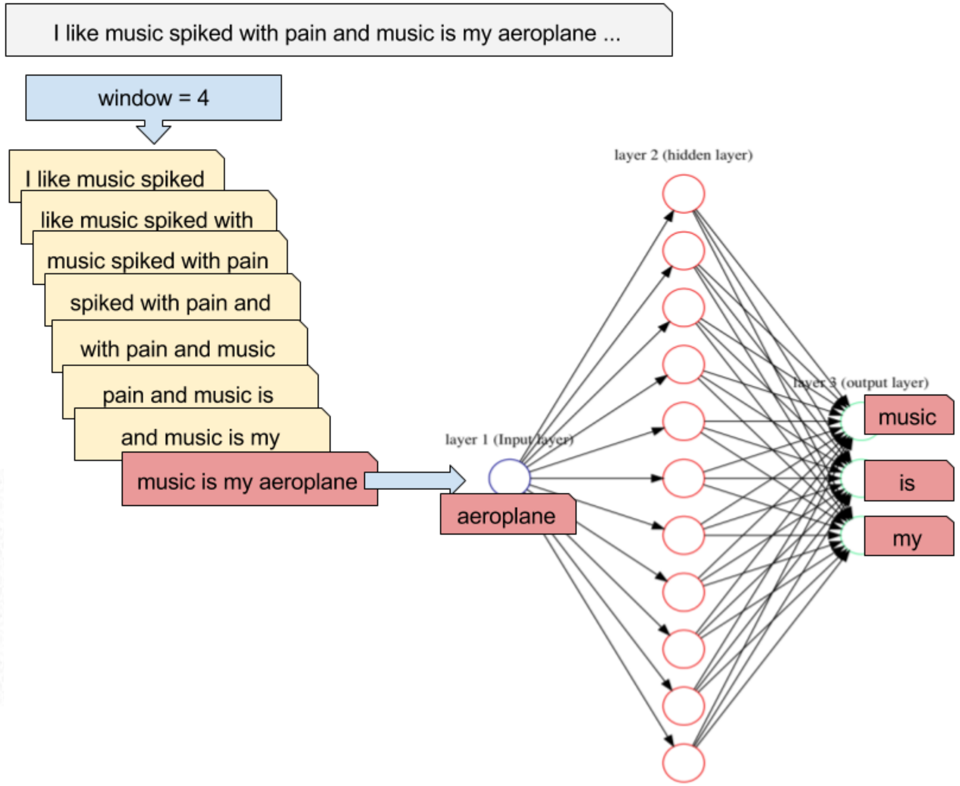
Binary Code Embedding

When Binary Analysis Meets Deep Learning

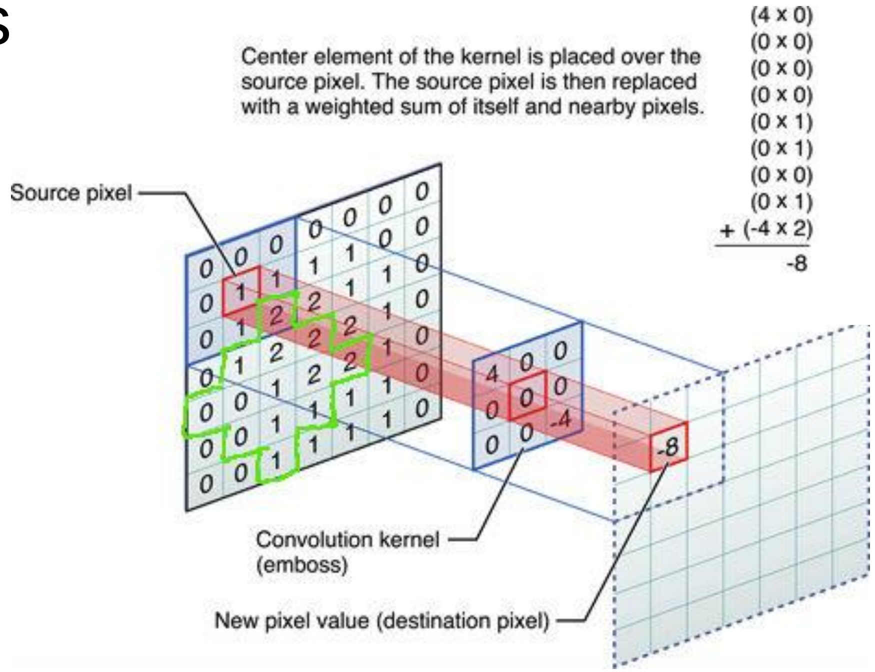
- Input Types (Must be numeric vector)
  - Raw bytes
  - Manually Selected Features
  - Automatically learned features (representation)
- Problems
  - Malware Classification
  - Vulnerability Detection
  - Function Argument Inference
  - Type Inference
  - Value-Set Analysis
  - …

Representation Learning in Other Domains

- Natural Language
  - Word Embeddings (word2vec)



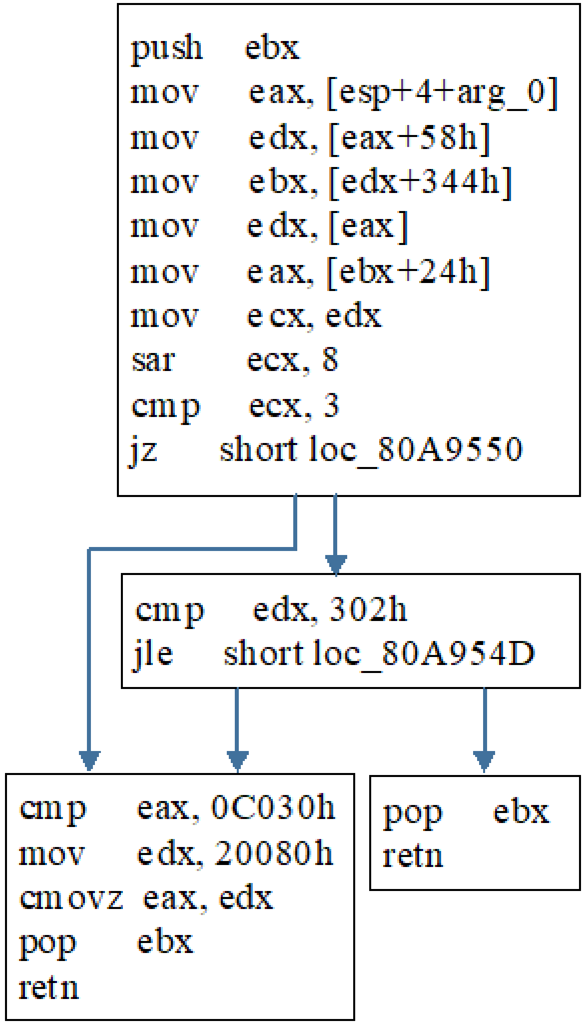
- Image Processing
  - Kernels/Filters



Binary Code Representation (Embedding)

- Disassembly
  - DeepDi [USENIX Sec 2022]: A deep learning-based fast disassembler
- Instruction
  - Word2Vec
  - Asm2Vec [S&P 2019]: A variant of Doc2Vec (PM-DM)
  - PalmTree [CCS 2021]: An assembly language model based on BERT
- Basic Block
  - Sum up instruction embeddings
  - InnerEye [NDSS 2019]
  - DeepBinDiff [NDSS 2020]
- Function
  - Genius [CCS 2016]: Codebook, vector quantization
  - Gemini [CCS 2017]: Graph Neural Network

Function Embedding



Numeric Vector (or Embedding)

Such that two functions that have the same semantics (same I/O behaviors) will have similar embeddings (distance is short), and two functions with different semantics will have dissimilar embeddings (distance is long).

Binary Code Clone Search

- Given a piece of binary code (e.g., a binary function)
- Quickly return a set of candidates
  - Semantically equivalent or similar
  - May come from different architectures
  - May be generated by different compilers and options

- Applications
  - Vulnerability Search
  - Plagiarism Detection
  - Malware Provenance

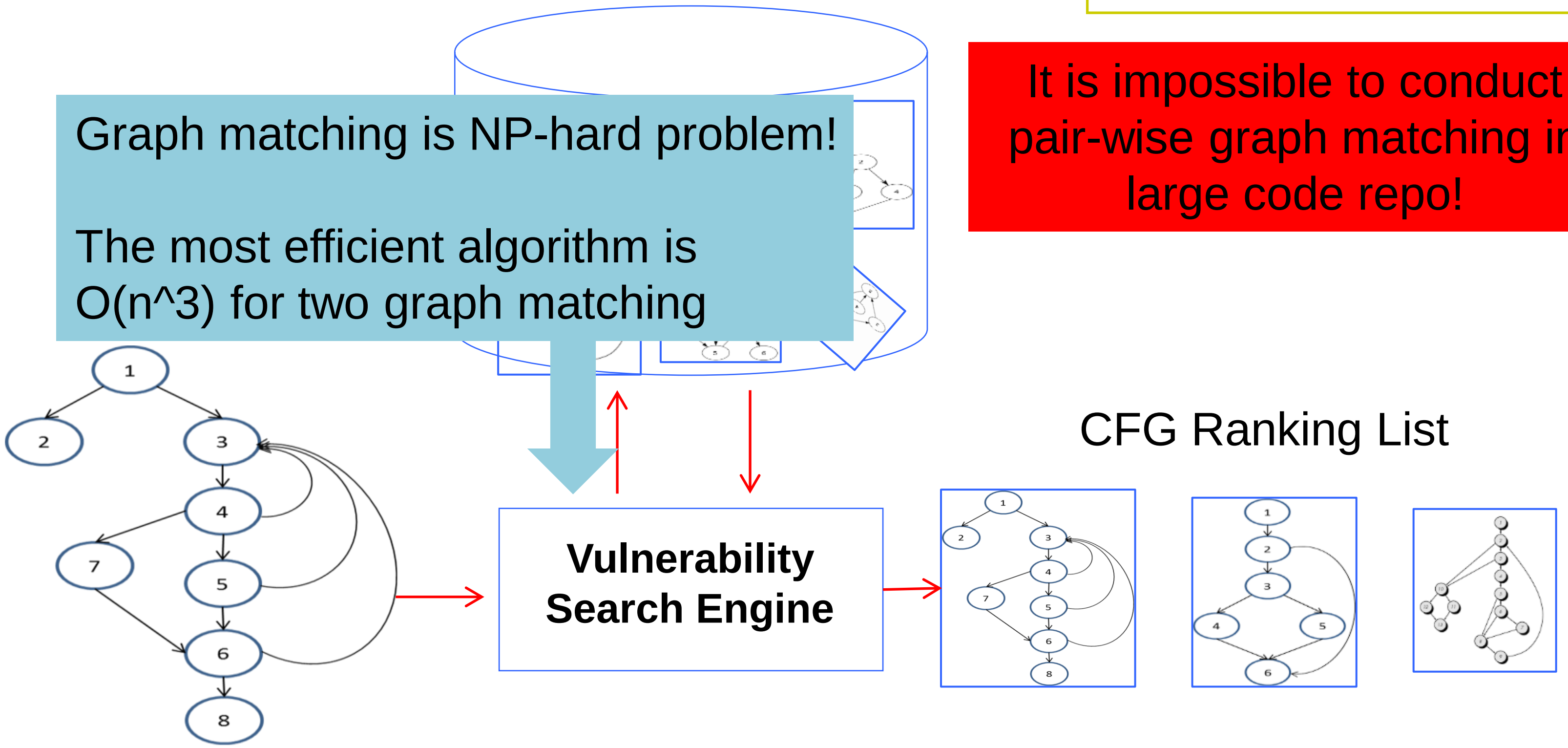
Prior Work based on Graph Matching

“Multi-MH & Multi-k-MH”*[Pewny et al. Oakland’15]*
“DiscovRe” *[Eschweiler et al. NDSS’16]*

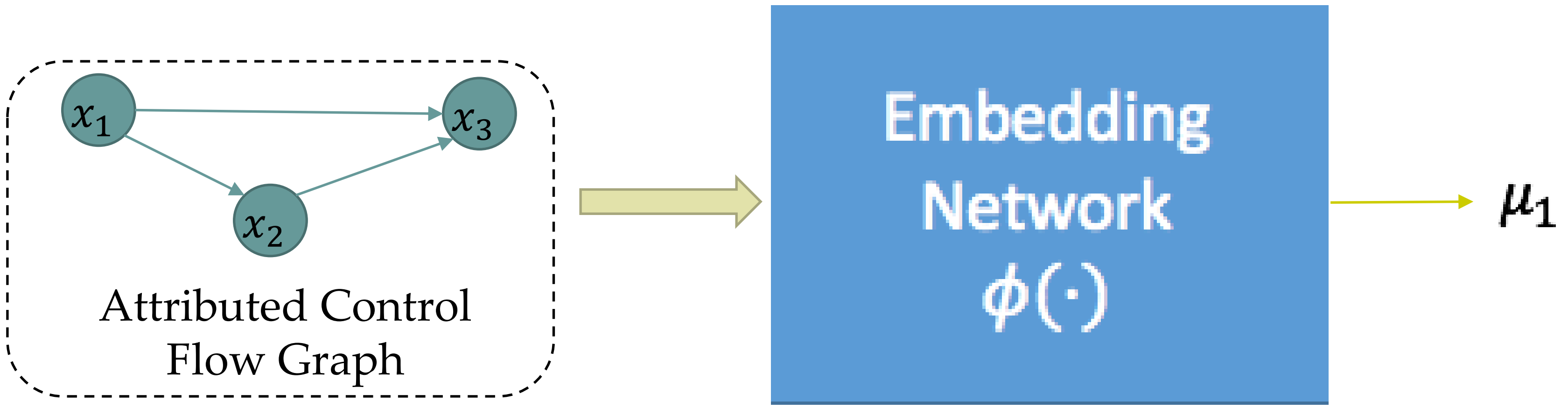
It is impossible to conduct pair-wise graph matching in large code repo!

Graph matching is NP-hard problem!

The most efficient algorithm is O(n^3) for two graph matching



Our Idea: DNN-based Function Presentation Learning



Dai, et al. Discriminative Embeddings of Latent Variable Models for Structured Data. ICML 2016.

Input

- Attributed Control Flow Graph

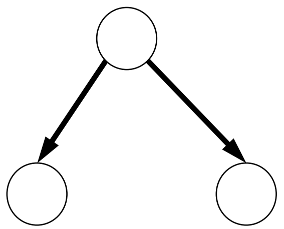
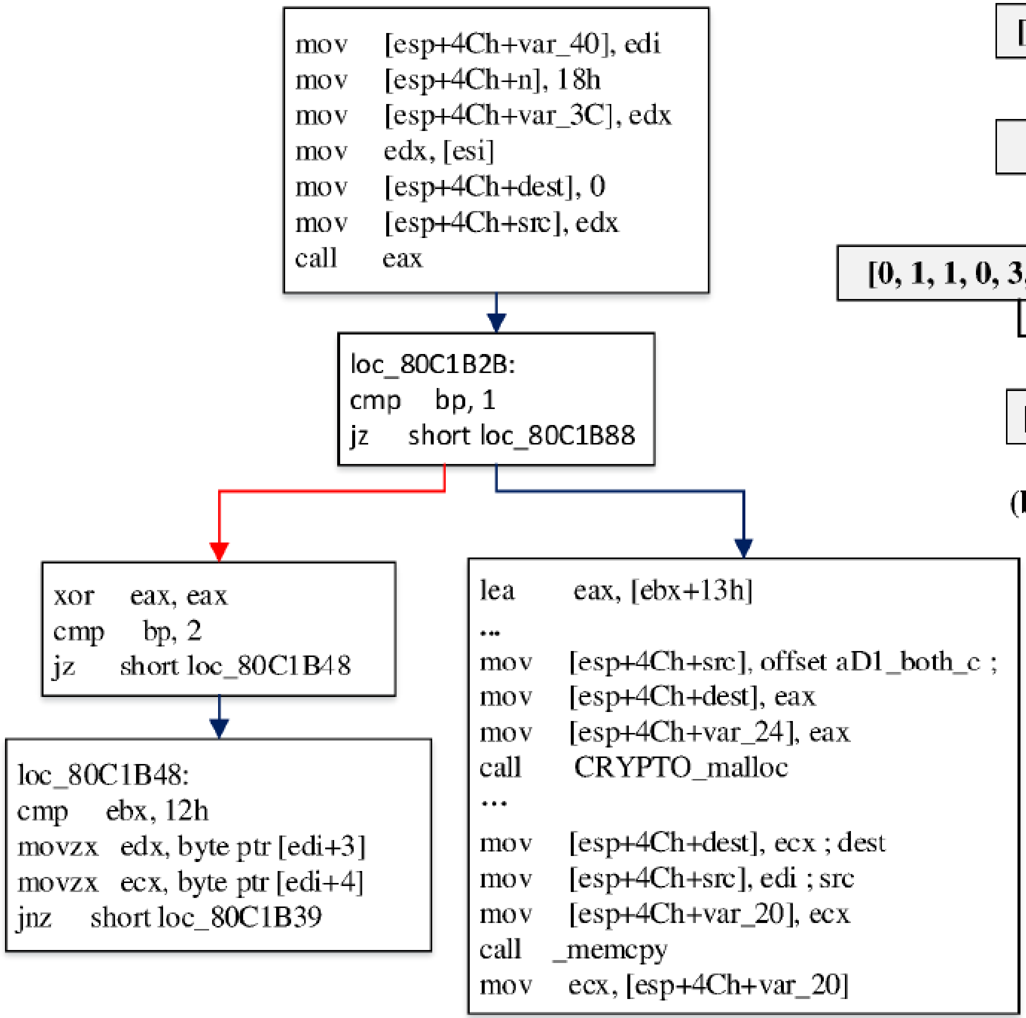


Table 1: Basic-block level features.

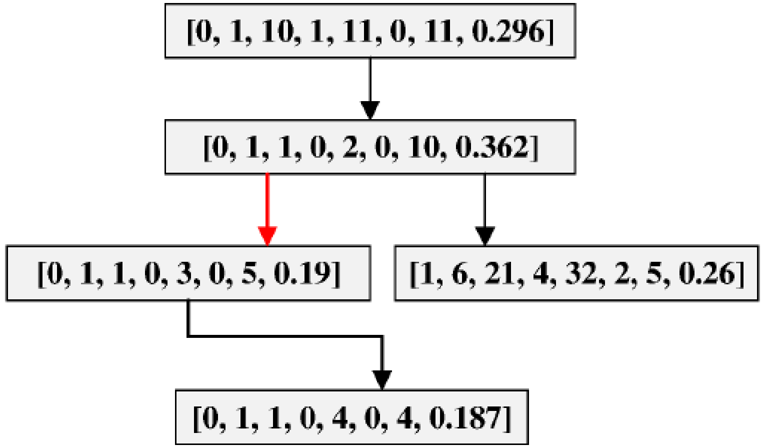
Type	Feature Name
Statistical Features	String Constants
	Numeric Constants
	No. of Transfer Instructions
	No. of Calls
	No. of Instructions
	No. of Arithmetic Instructions
Structural Features	No. of offspring
	Betweeness

Manually selected features for now to support cross-architecture search

An example of ACFG

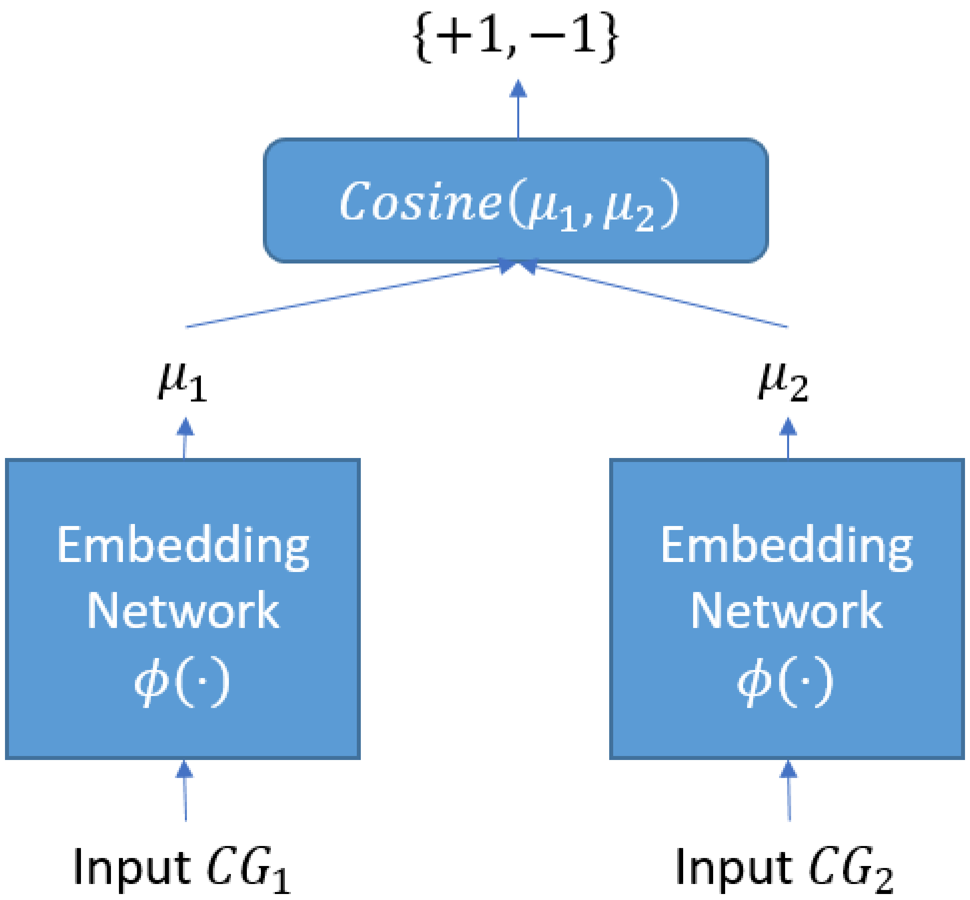


(a) Partial control flow graph of dtls1_process_heartbeat



(b) The corresponding ACFG

Training: Siamese



1. Application-independent pretraining
   - Compile given source code into different platforms using different compilers and different optimization-levels
   - A pair of binary functions compiled from the same source code is labeled with +1
   - Otherwise, -1

2. Application-dependent retraining
   - Human can label similar and dissimilar pairs of binary functions
   - This additional training data can be used in a retraining process

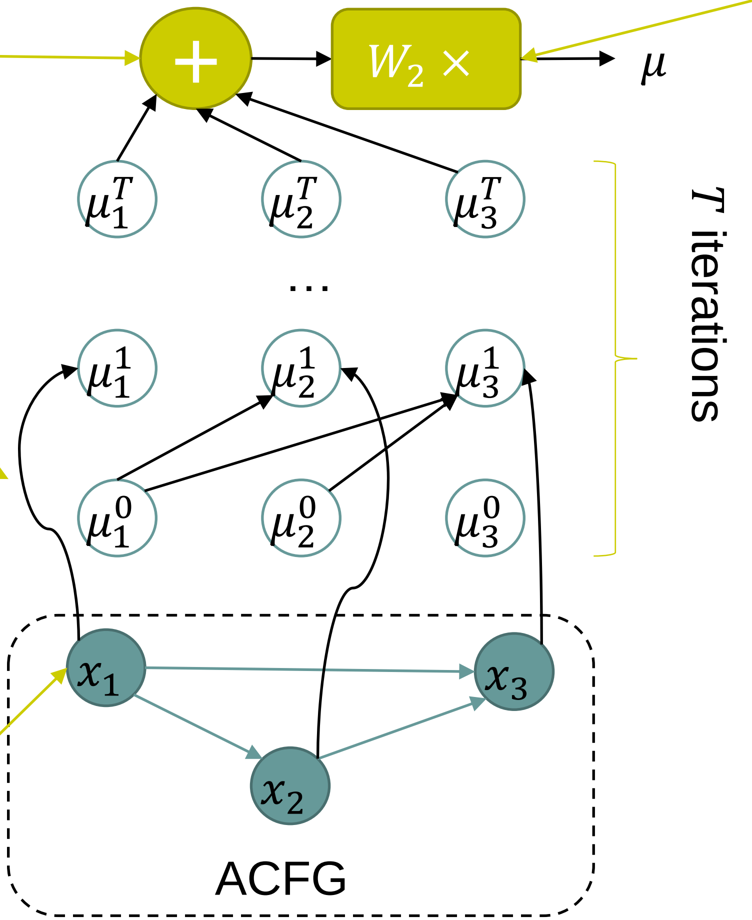
Take a closer look at the embedding network

3. After the last iteration, the embeddings on all vertices are aggregated together

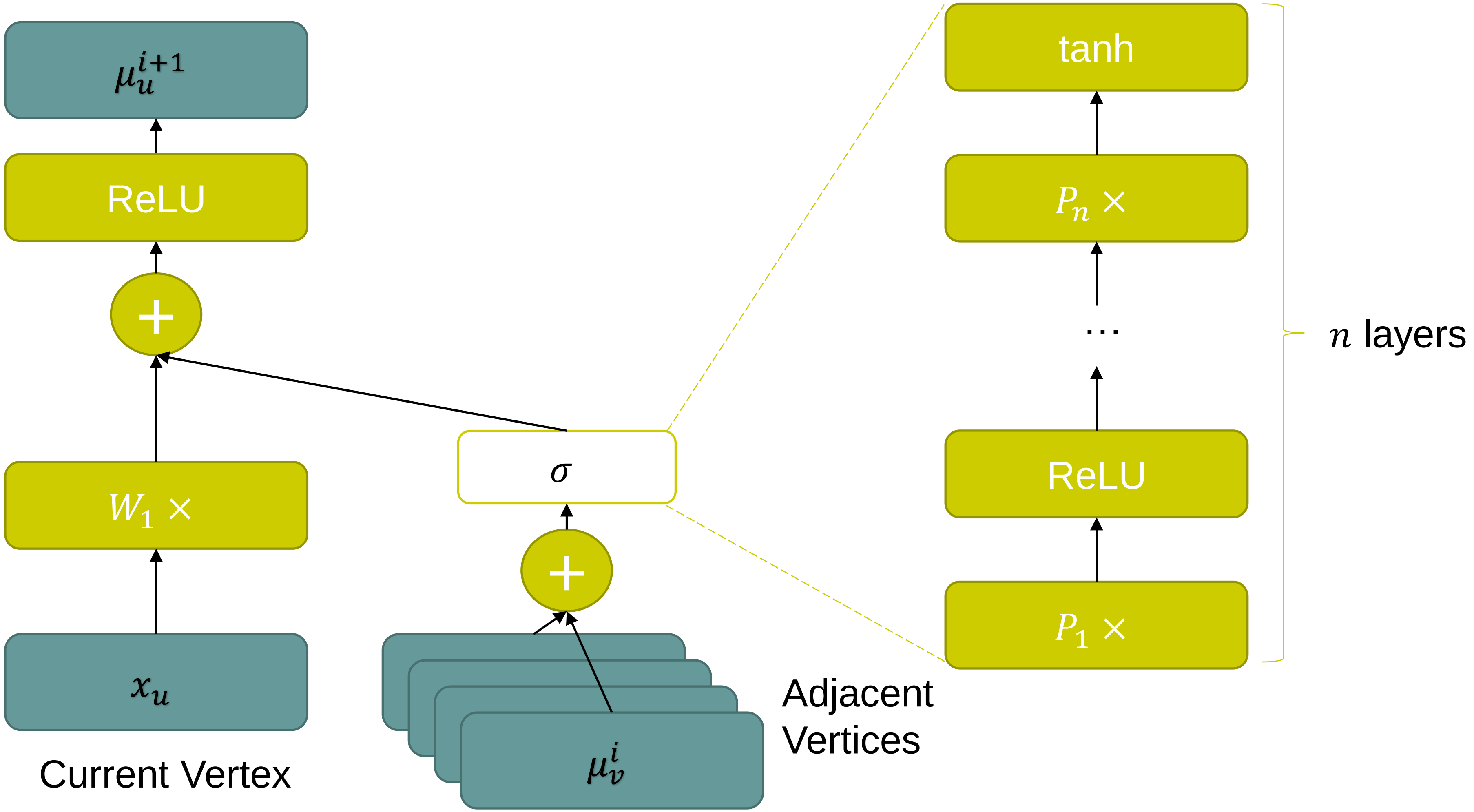
4. An affine transformation is applied in the end to compute the embedding for the graph

2. In each iteration, the embedding on each vertex is propagated to its neighbors

1. Initially, each vertex has an embedding vector computed from each code block



Take a closer look at propagation

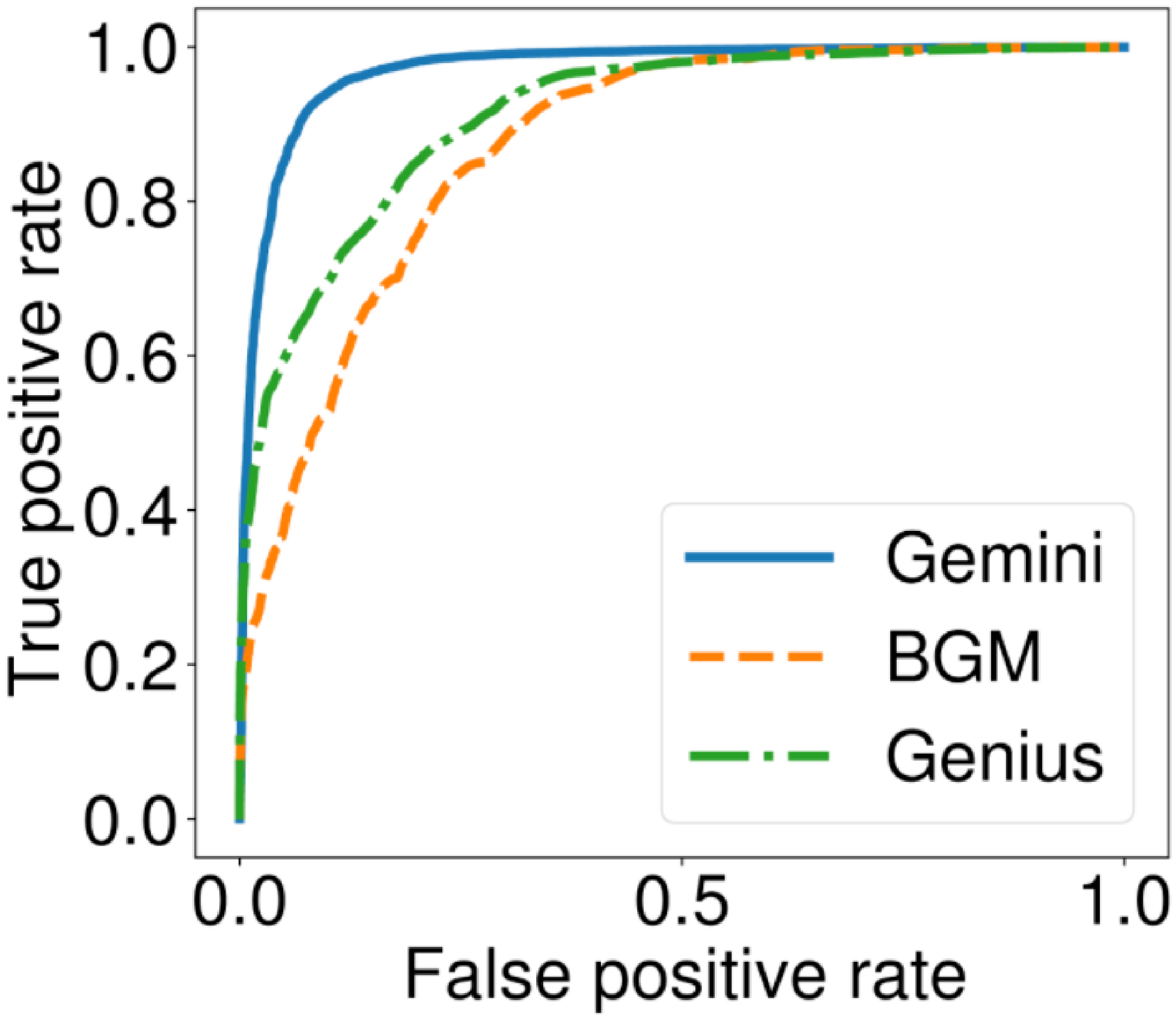


Why Deep Neural Network?

- High efficiency
  - Vector and Matrix Computations can be accelerated by SIMD and GPU

- High accuracy
  - The model is trained end-to-end
  - No graph matching algorithm!

Accuracy: ROC curve on test data



Serving time (per function processing time)

Genius: a few secs to a few mins

Now: a few milliseconds

2500 × to 16000 × faster!

Training time

Genius: > 1 week

Now: < 30 mins

Identified Vulnerabilities in Large Scale Dataset

Function Name	Vendor	Firmware	Binary File	Similarity
ssl3_get_new_session_ticket	D-Link	DAP-1562_FIRMWARE_1.00	wpa_supplicant.acfgs	0.962374508
port_check_v6	D-Link	DES-1210-28_REVB_FIRMWARE_3.12.015	in.ftpd.acfgs	0.955408692
sub_42EE7C	TP-Link	TD-W8970B_V1_140624	racoon.acfgs	0.954742193
sub_42EE7C	TP-Link	TD-W8970_V1_130828	racoon.acfgs	0.954742193
prsa_parse_file	TP-Link	Archer_D5_V1_140804	racoon.acfgs	0.949814439
sub_432B8C	TP-Link	TD-W8970B_V1_140624	racoon.acfgs	0.949583828
sub_432B8C	TP-Link	TD-W8970_V1_130828	racoon.acfgs	0.949583828
ssl3_get_new_session_ticket	DD-wrt	dd-wrt.v24-23838_NEWD-2_K3.x_mega-WNR3500v2_VC	openvpn.acfgs	0.94668287
ucSetUsbipServer	TP-Link	WDR4900_V2_130115	httpd.acfgs	0.946312308
ssl3_get_new_session_ticket	Netgear	tomato-Cisco-M10v2-NVRAM32K-1.28.RT-N5x-MIPSR2-110-PL-Mini	libssl.so.1.0.0.acfgs	0.945933044
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-K26-1.28.RT-MIPSR1-109-Mini	libssl.so.1.0.0.acfgs	0.945933044
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-K26USB-1.28.RT-N5x-MIPSR2-110-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E4200USB-NVRAM60K-1.28.RT-MIPSR2-110-PL-BT	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E3000USB-NVRAM60K-1.28.RT-MIPSR2-110-BT-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-K26USB-1.28.RT-MIPSR1-109-AIO	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-Netgear-3500Lv2-K26USB-1.28.RT-N5x--109-AIO	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E4200USB-NVRAM60K-1.28.RT-MIPSR2-109-AIO	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E1550USB-NVRAM60K-1.28.RT-N5x-MIPSR2-110-Nocat-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-K26USB-1.28.RT-N5x-MIPSR2-115-PL-L600N	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E1550USB-NVRAM60K-1.28.RT-N5x-MIPSR2-110-BT-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E3000USB-NVRAM60K-1.28.RT-MIPSR2-108-PL-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E1550USB-NVRAM60K-1.28.RT-N5x-MIPSR2-110-Mega-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E1200v2-NVRAM64K-1.28.RT-N5x-MIPSR2-108-PL-Max	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-K26USB-1.28.RT-MIPSR1-109-Mega-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E3000USB-NVRAM60K-1.28.RT-MIPSR2-109-Big-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E4200USB-NVRAM60K-1.28.RT-MIPSR2-108-PL-Nocat-VPN	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-Netgear-3500Lv2-K26USB-1.28.RT-N5x--110-ND-AIO	libssl.so.1.0.0.acfgs	0.945932984
ssl3_get_new_session_ticket	Tomato_by_Shibby	tomato-E4200USB-NVRAM60K-1.28.RT-MIPSR2-109-Nocat-VPN	libssl.so.1.0.0.acfgs	0.945932984

Among top 50: **42** out of **50** are confirmed vulnerabilities

Previous work: **10**/**50**

More Embedding Schemes Coming!

UCR

- Gemini published in CCS, October 2017
  - 422 citations as of 08/10/2022

- Asm2Vec, Oakland 2019
- InnerEye, NDSS 2019
- FunctionSimSearch, Google Project Zero Team
- CodeCMR, NeurIPS 2020
- StateFormer, FSE 2021
- jTrans, ISSTA 2022
- How ML is solving binary similarity problems, USENIX Security 2022

Project Zero

News and updates from the Project Zero team at Google

Tuesday, December 18, 2018

Searching statically-linked vulnerable library functions in executable code

Helping researchers find 0ld days

Posted by Thomas Dullien, Project Zero

On the side of academic research, several interesting papers (CCS '16, CCS '17) have proposed sophisticated machine-learning-based methods to combine code embeddings with approximate nearest neighbor searches. They calculate a representation of code in R^n, and then search for nearby points to identify good candidates. While these approaches look powerful and sophisticated, public implementations do not exist, and adoption among practitioners has not happened. On the practical side, real-world use has been derived from CFG-focused algorithms such as MACHOC - but with the downside of being not tolerant to structural changes and not allowing for any "learning" of distances. Recently at (SSTIC '18) a neural-network based approach has been presented, with an announcement of making the code available in the next months.

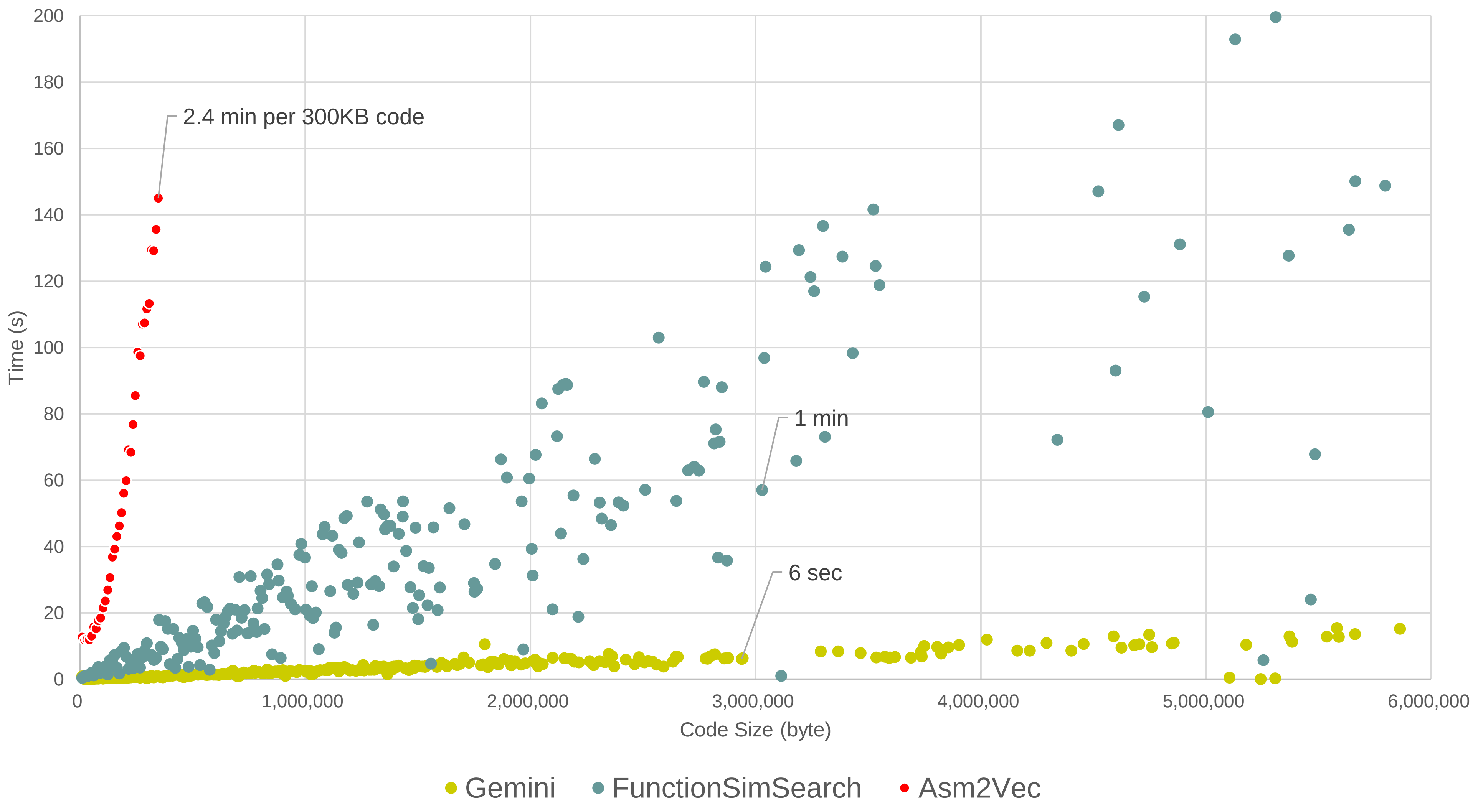
This file describes FunctionSimSearch - an Apache-licensed C++ toolkit with Python bindings which provides three things:

Table 4: Comparison of fuzzy hashing and machine-learning models on Dataset-2

Model name	Description	AUC			MRR10			Recall@1			Testing time (s)		
		XO	XA	XA+XO	XO	XA	XA+XO	XO	XA	XA+XO	Feat	Inf	Tot 100
[67] Zeek (direct comparison)	Strands	0.92	0.94	0.91	0.42	0.45	0.36	0.28	0.31	0.21	7225.41	67.00	9.92
[40] GMN (direct comparison)	CFG + BoW opc 200	**0.97**	**0.98**	**0.96**	**0.75**	**0.84**	**0.71**	**0.66**	**0.77**	**0.61**	1093.68	1005.00	1.83
[40] GMN (direct comparison)	CFG + No features	0.93	0.97	0.95	0.61	0.76	0.67	0.51	0.68	0.59	978.15	876.00	1.63
[40] GNN	CFG + BoW opc 200	0.95	0.97	0.95	0.67	0.79	0.67	0.57	0.73	0.57	1093.68	116.52	1.66
[40] GNN	CFG + No features	0.91	0.96	0.93	0.54	0.71	0.59	0.44	0.62	0.49	978.15	100.34	1.48
[76] GNN (s2v)	CFG + BoW opc 200	0.94	0.95	0.93	0.58	0.57	0.58	0.48	0.42	0.47	1093.68	118.59	1.66
[76] GNN (s2v)	CFG + Gemini	0.93	0.96	0.93	0.57	0.74	0.57	0.47	0.64	0.49	5139.91	98.40	7.18
[76] GNN (s2v)	CFG + No features	0.75	0.79	0.77	0.18	0.20	0.23	0.12	0.13	0.16	978.15	40.87	1.40
[45] w2v + AVG + GNN (s2v)	CFG + N. asm 150	0.90	0.88	0.87	0.46	0.31	0.42	0.38	0.18	0.33	1070.01	258.95	1.82
[45] w2v + wAVG + GNN (s2v)	CFG + N. asm 150	0.87	0.87	0.85	0.37	0.29	0.36	0.29	0.17	0.27	1070.01	253.72	1.81
[45] w2v + RNN + GNN (s2v)	CFG + N. asm 150	0.88	0.90	0.88	0.32	0.35	0.35	0.19	0.18	0.23	1070.01	685.50	2.41
[49] w2v + SAFE	N. asm 150	0.88	0.90	0.88	0.27	0.30	0.31	0.14	0.18	0.20	1031.23	33.33	1.46
[49] w2v + SAFE	N. asm 250	0.86	0.88	0.87	0.28	0.32	0.28	0.16	0.19	0.19	1031.23	33.33	1.46
[49] w2v + SAFE + trainable	N. asm 150	0.91	0.93	0.91	0.40	0.43	0.37	0.26	0.25	0.23	1031.23	33.57	1.46
[49] rand + SAFE + trainable	N. asm 150	0.90	0.91	0.90	0.28	0.33	0.31	0.14	0.17	0.21	1031.23	33.81	1.46
[14] Asm2Vec	Rand walks asm	0.94	0.69	0.75	0.60	0.07	0.22	0.49	0.02	0.18	978.15	5235.00	8.51
[38] PV-DM	Rand walks asm	0.94	0.66	0.72	0.64	0.08	0.23	0.51	0.05	0.19	978.15	5239.00	8.52
[38] PV-DBOW	Rand walks asm	0.94	0.66	0.72	0.63	0.07	0.23	0.50	0.03	0.20	978.15	3004.00	5.46
[60] Trex	512 Tokens	0.94	0.94	0.94	0.61	0.50	0.53	0.50	0.38	0.46	1493.58	1365.89	3.92
[74] Catalog_1	size 16	0.72	0.50	0.55	0.43	0.06	0.14	0.38	0.06	0.14	654.70	0.00	0.90
[74] Catalog_1	size 128	0.86	0.48	0.57	0.50	0.07	0.17	0.42	0.06	0.14	823.47	0.00	1.13
[18] FSS	G	0.77	0.81	0.77	0.26	0.35	0.32	0.18	0.26	0.26	1903.46	466.07	3.25
[18] FSS	G + M	0.79	0.68	0.69	0.29	0.15	0.21	0.23	0.09	0.15	1903.46	466.07	3.25
[18] FSS	G + M + I	0.80	0.68	0.69	0.30	0.16	0.20	0.23	0.10	0.14	1903.46	466.07	3.25
[18] FSS	w(G + M + I)	0.83	0.80	0.78	0.43	0.30	0.36	0.36	0.23	0.29	1903.46	466.07	3.25

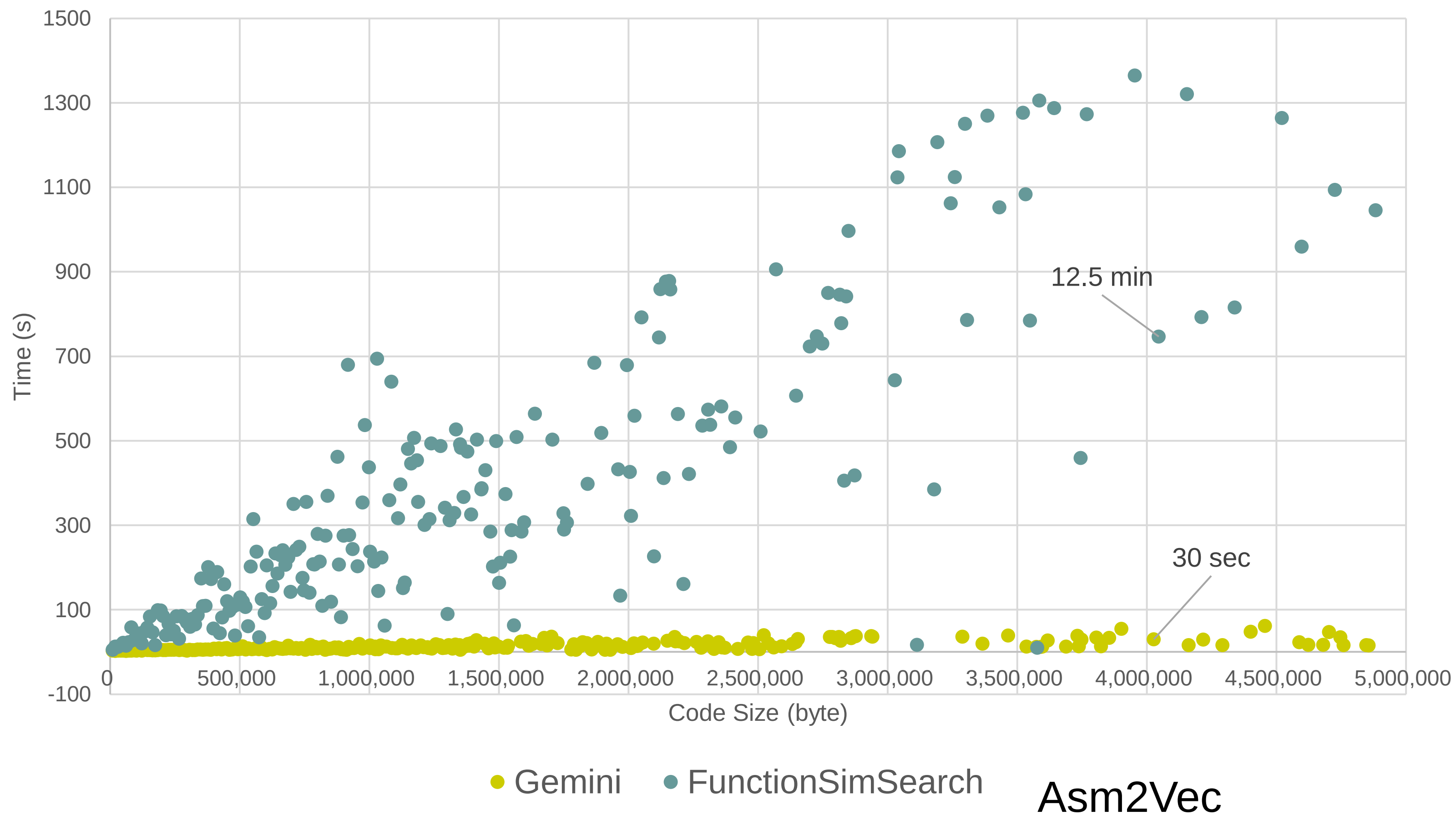
How ML is solving binary similarity problems, USENIX Security 2022

Efficiency – Embedding Generation



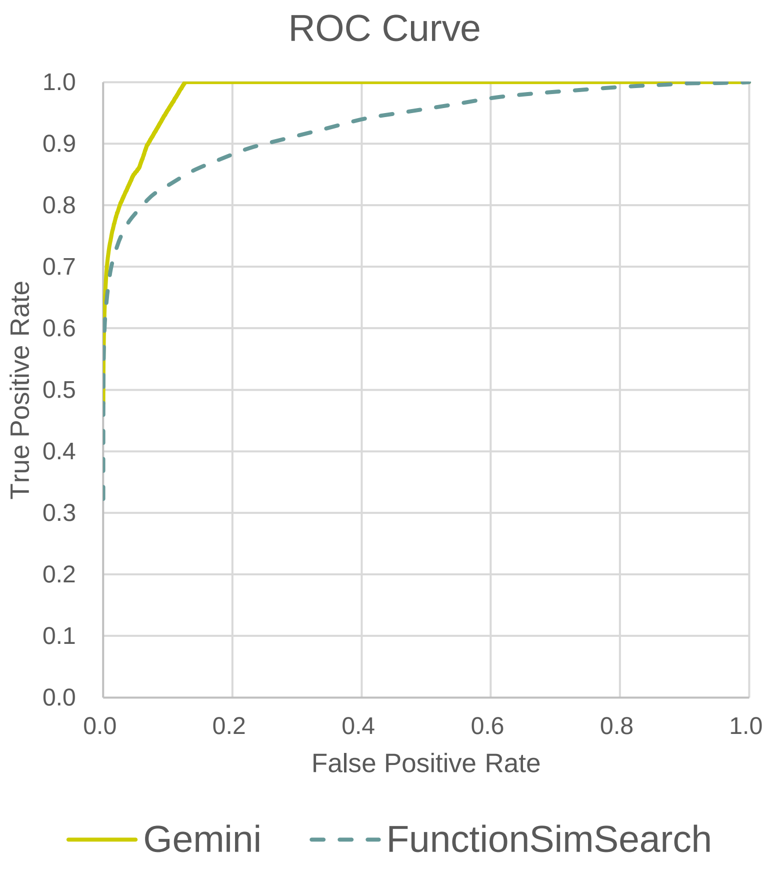
Efficiency - Matching

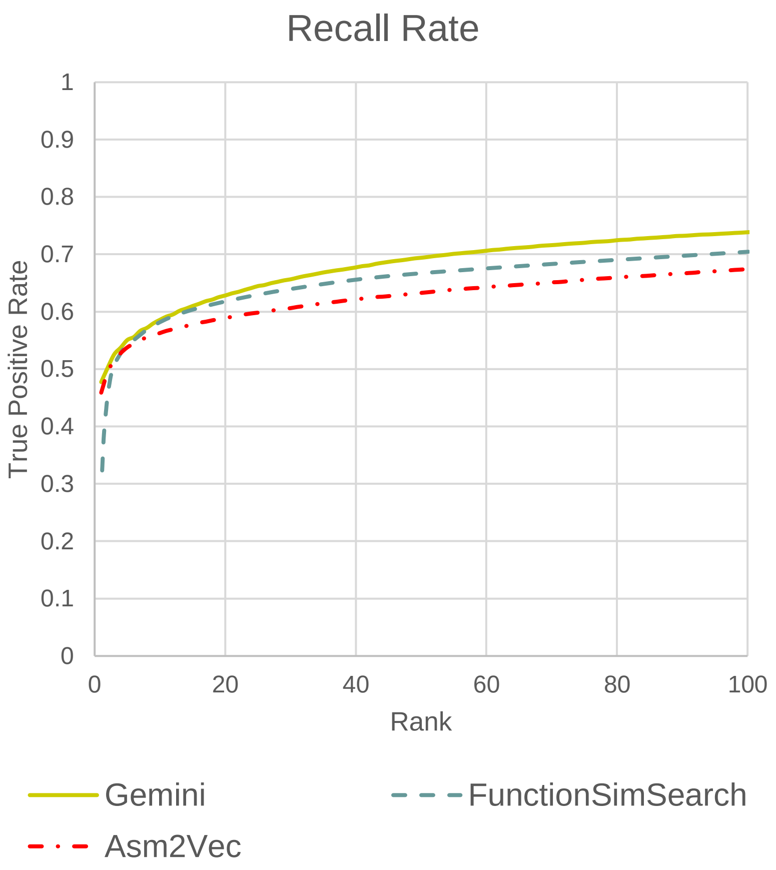
Binary x 1.5M Functions



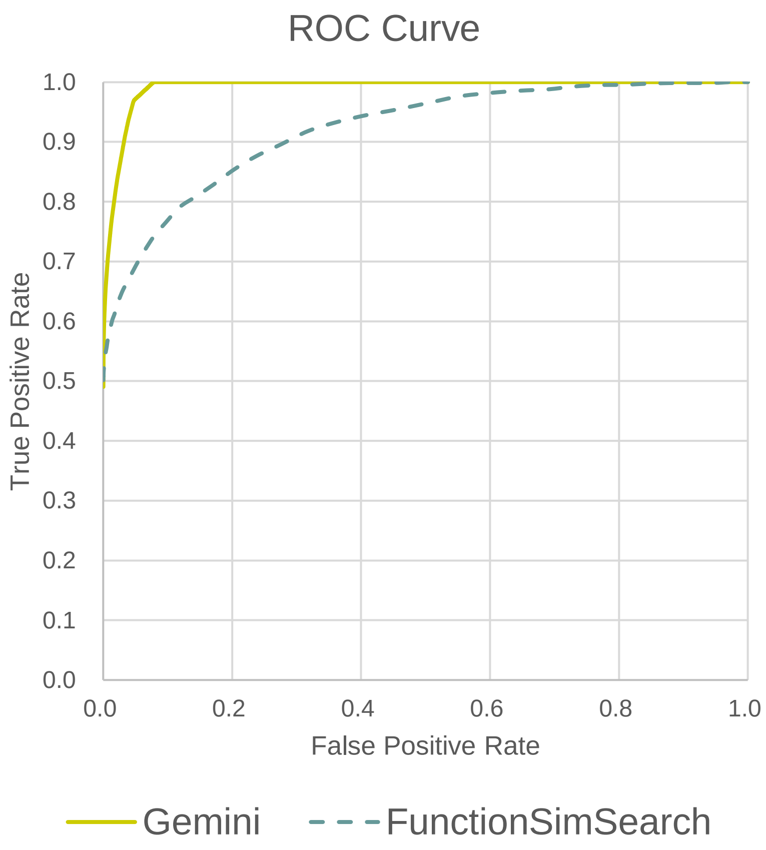
Asm2Vec
500KB code x 60K Functions
20 sec

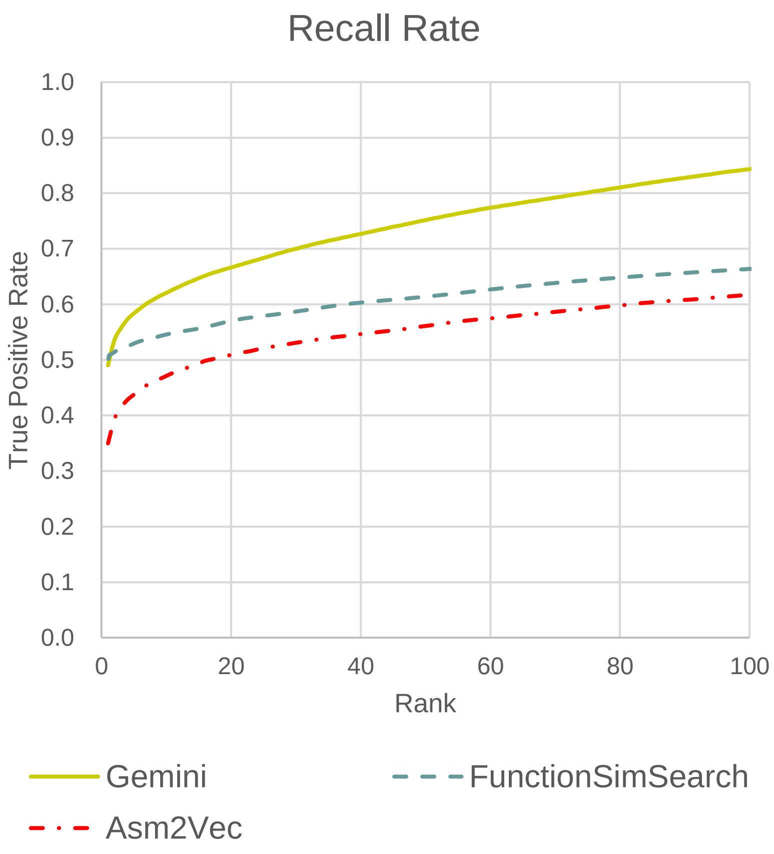
Accuracy - Cross Optimization Level





Accuracy - x86 vs x64

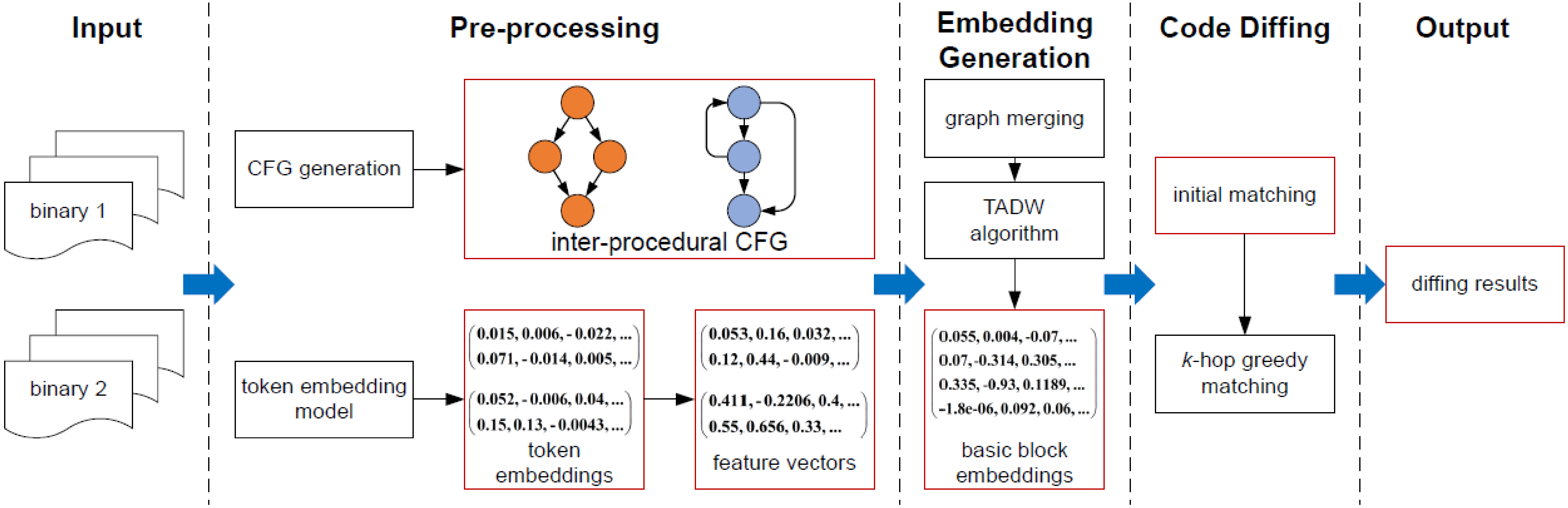




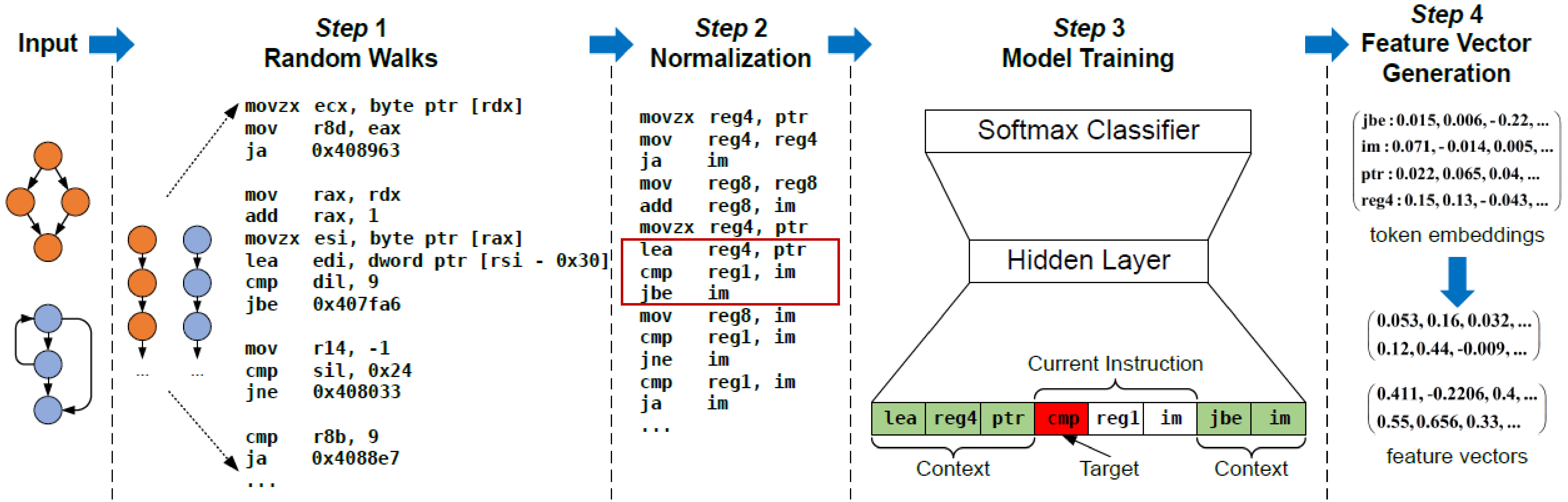
Binary Code Diffing

- We all know BinDiff, right?
  - Graph isomorphic matching with lots of heuristics
- We developed DeepBinDiff
  - Learn an embedding for each **basic block**
  - Capture both **block-level features** and **topological features** in CFG via DeepWalk

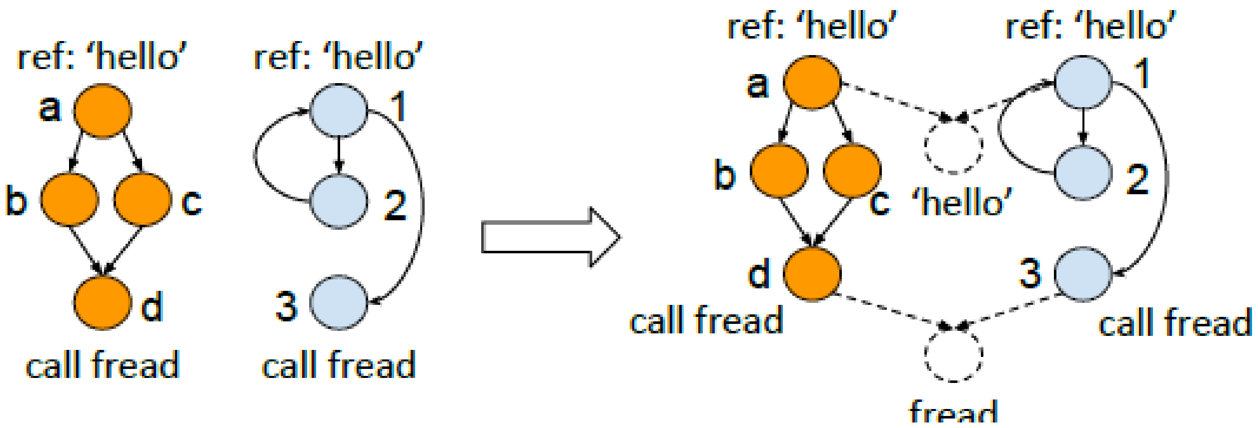
DeepBinDiff Overview

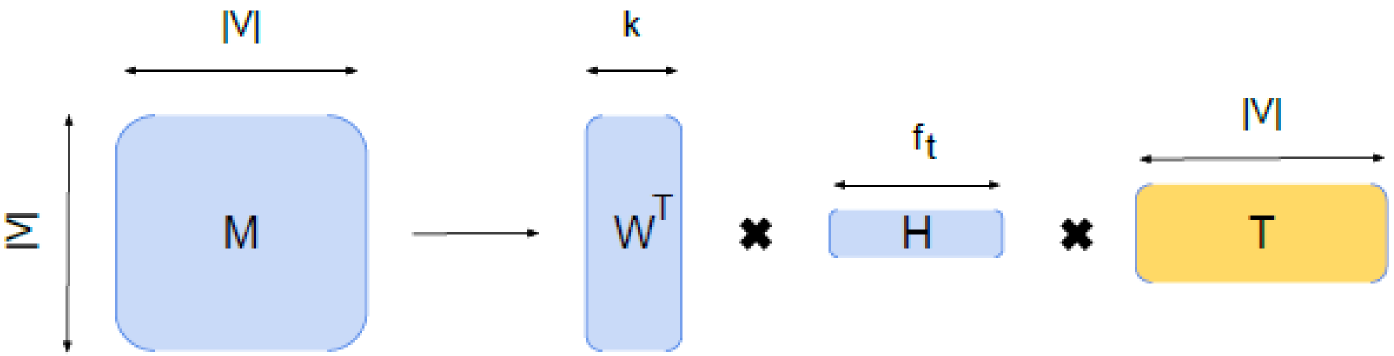


Basic Block Embedding Generation



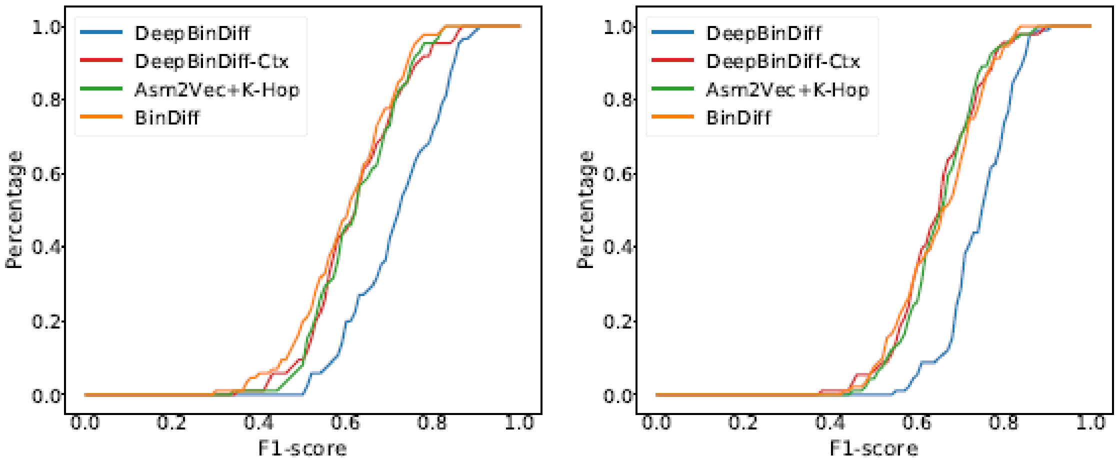
Graph Merge and TADW



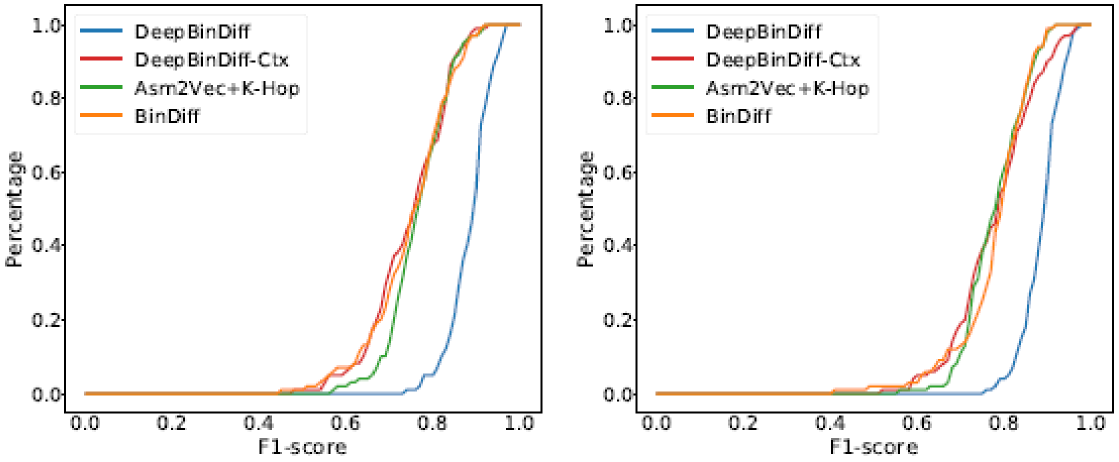


$$\min_{W,H} ||M - W^T HT||_F^2 + \frac{\lambda^2}{2}(||W||_F^2 + ||H||_F^2)$$

Cross-Version Diffing F1-Score CDF on Coreutils

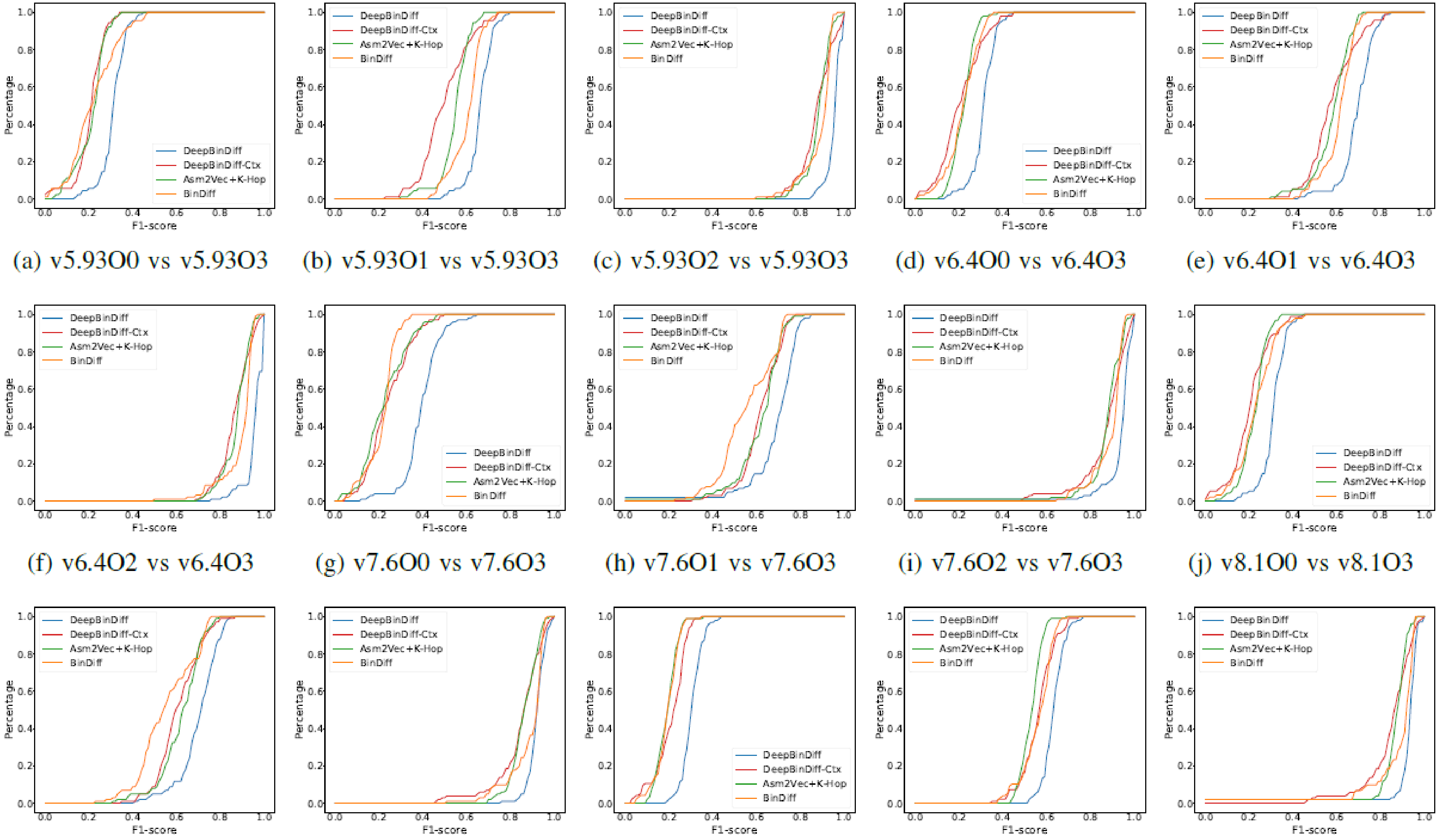


(a) v5.93 compared with v8.30 (b) v6.4 compared with v8.30



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

Cross-Optimization 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

PalmTree: Learning an Assembly Language Model for Instruction Embedding

ACM CCS 2021

Currently available at:
https://github.com/palmtreemodel/PalmTree

Background – Input Choices

1. **Raw-byte Encoding**
   - Feed Raw-byte directly: αDiff (ASE'18)
   - One-hot encoding: converts each byte into a 256-dimensional vector. Shin et al. (USENIX'15), MalConv (AAAI Workshop'18), DeepVSA (USENIX'19)
   - *It does not provide any semantic level information.*

2. **Manual Encoding of Disassembled Instructions**
   - Instruction2Vec (ICONI'17), Gemini (CCS'17)
   - Expert knowledge.

Background – Input Choices

3. **Learning-based Encoding**

- Word2vec: instruction – word, function – document
  - Code similarity detection: SAFE (DIMVA'19), InnerEye (NDSS'19)
  - Function prototype inference: EKLAVYA (USENIX'17)
- Doc2vec (PV-DM):
  - Asm2Vec (S&P'19) – treat instruction as one opcode and two operands
- *Can carry higher-level semantic information.* ***However:***

Background – Challenges

1. **Instructions are complex and diverse**

A constant or a string symbol

Small Constant

CPU registers

Memory operand: **base**+**index***scale***+**displacement**

```
; memory operand with complex expression
mov [ebp+eax*4-0x2c], edx
; three explicit operands, eflags as implicit operand
imul [edx], ebx, 100
; prefix, two implicit memory operands
rep movsb
; eflags as implicit input
jne 0x403a98
```

Conditional Jump takes **EFLAGS** as an implicit input

Background – Challenges

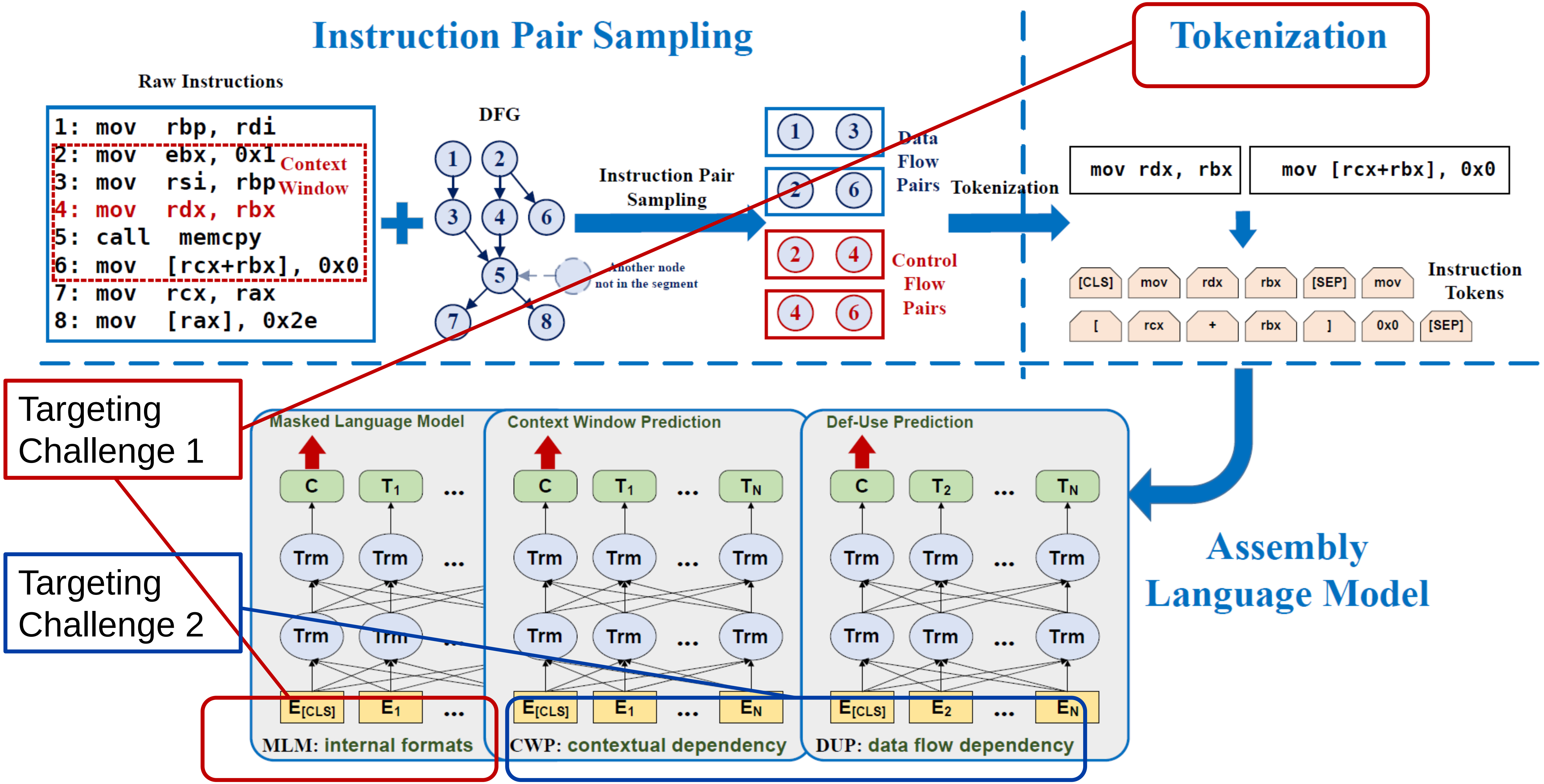
2. **Instructions can be reordered**

Data Dependency

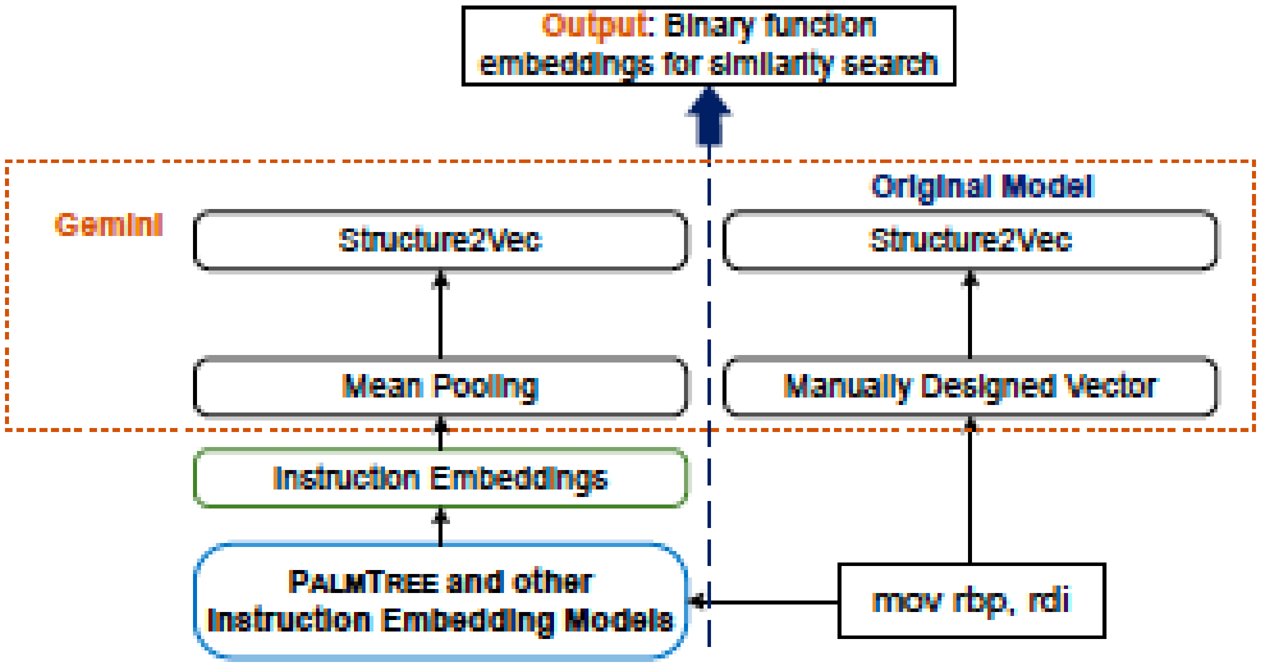
```
; prepare the third argument for function call
mov rdx, rbx
; prepare the first argument for function call
mov rsi, rbp
; prepare the second argument for function call
mov rdi, rax
; call memcpy() function
call memcpy
; test rbx register (this instruction is reordered)
test rbx, rbx
; store the return value of memcpy() into rcx register
mov rcx, rax
; conditional jump based on EFLAGS from test instruction
je  0x40adf0
```

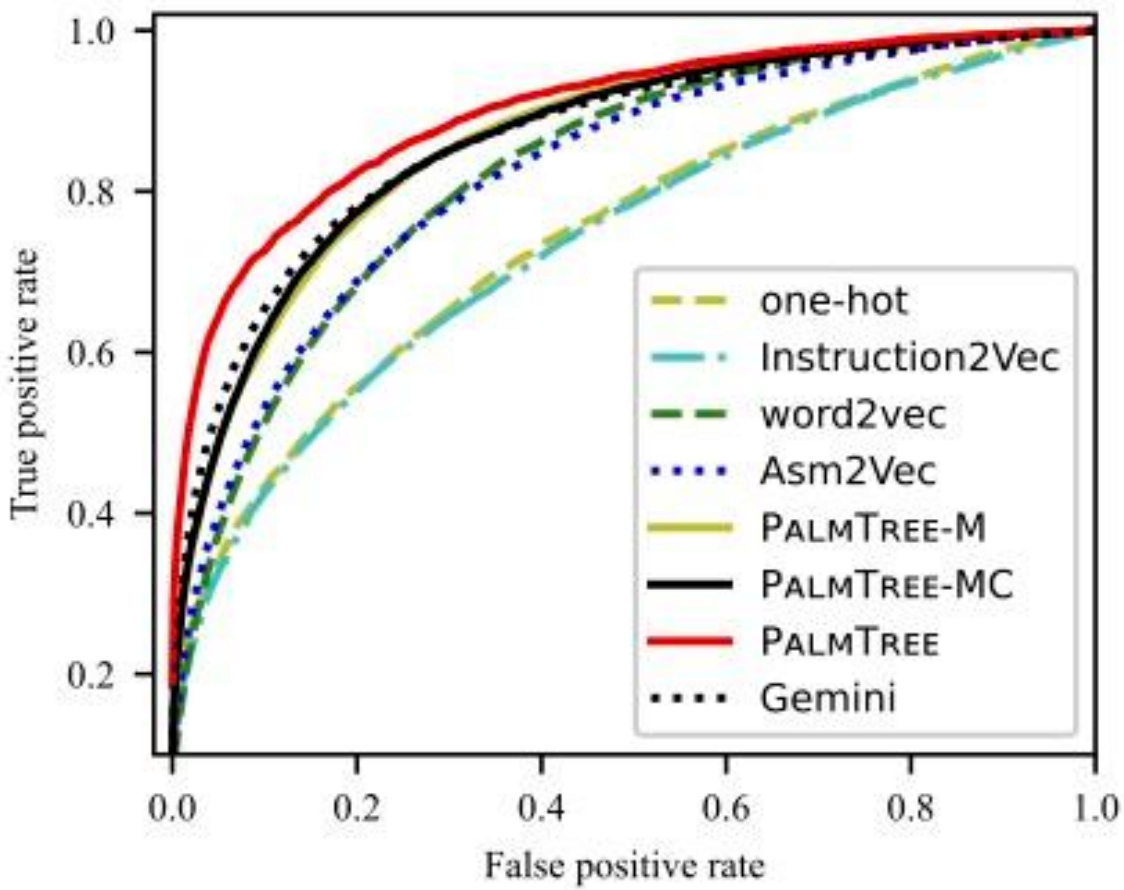
Data Dependency

PalmTree: a pre-trained assembly language model

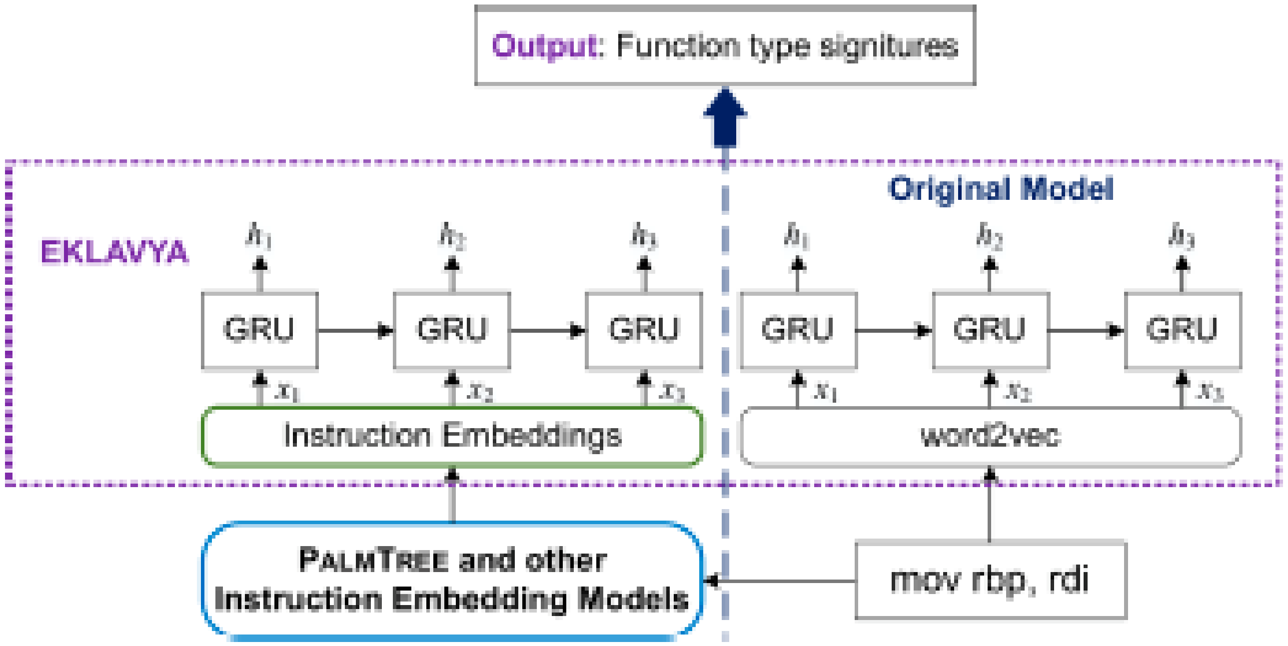


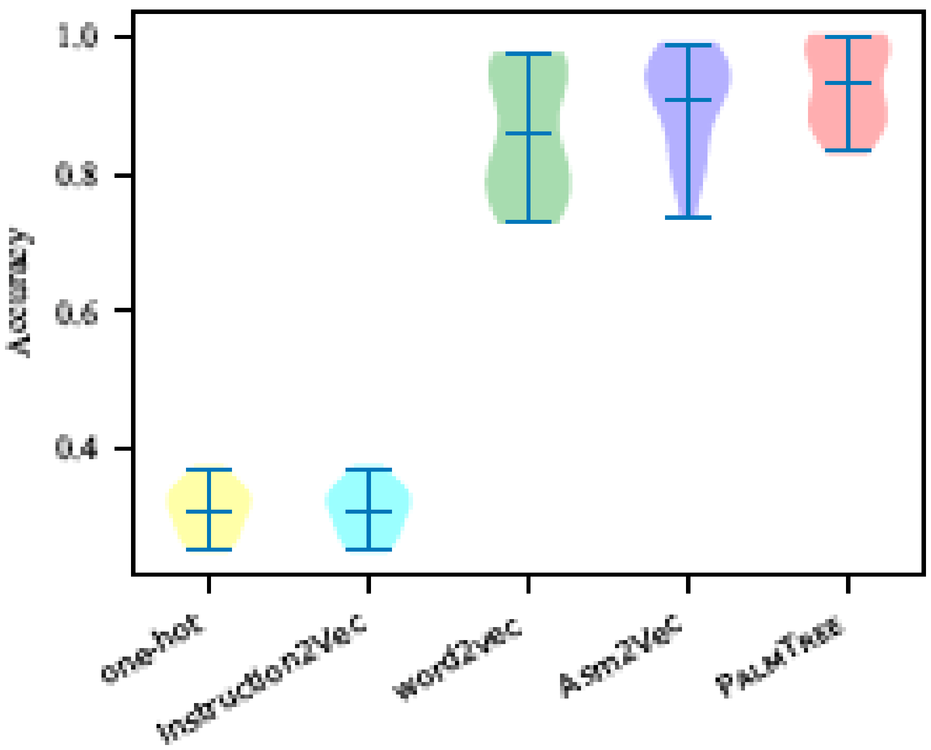
PalmTree's Impact on Gemini



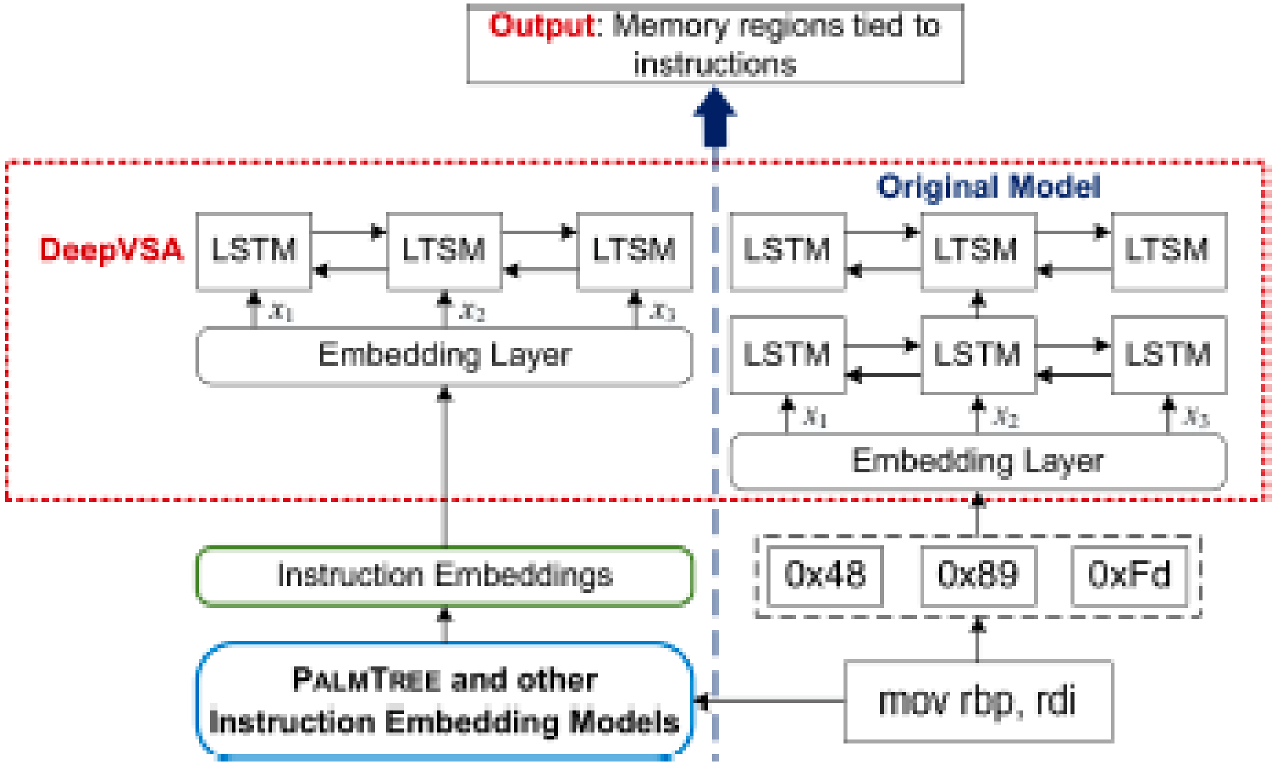


PalmTree's Impact on EKALAVYA





PalmTree's Impact on DeepVSA



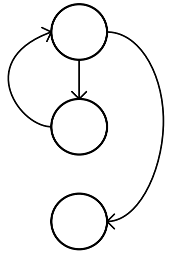
Embeddings	Global			Heap			Stack			Other		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
one-hot	0.453	0.670	0.540	0.507	**0.716**	0.594	0.959	0.866	0.910	0.953	0.965	0.959
Instruction2Vec	0.595	0.726	0.654	0.512	0.633	0.566	0.932	0.898	0.914	0.948	0.946	0.947
word2vec	0.147	0.535	0.230	0.435	0.595	0.503	0.802	0.420	0.776	0.889	0.863	0.876
Asm2Vec	0.482	0.557	0.517	0.410	0.320	0.359	0.928	0.894	0.911	0.933	0.964	0.948
DeepVSA	**0.961**	0.738	0.835	0.589	0.580	0.584	0.974	0.917	0.944	0.943	0.976	0.959
PALMTREE-M	0.845	0.732	0.784	0.572	0.625	0.597	0.963	0.909	0.935	0.956	0.969	0.962
PALMTREE-MC	0.910	0.755	0.825	**0.758**	0.675	0.714	0.965	0.897	0.929	0.958	**0.988**	**0.972**
PALMTREE	0.912	**0.805**	**0.855**	0.755	0.678	**0.714**	**0.974**	**0.929**	**0.950**	**0.959**	0.983	0.971

DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly

USENIX Security 2022

Cornerstone in Binary Analysis

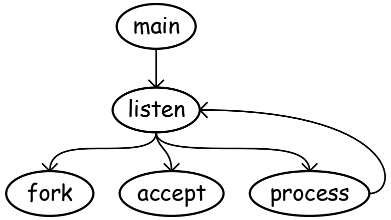
CFG



IR

%eq = icmp eq i32 %0, 0
br il %eq, label %then, label %else
%sub = sub i32 %0, 1

Call Graph



…

Gemini (CCS'17)
rev.ng (CC'17)
Karta
B2SMatcher (ASE'19)
Nucleus (EuroS&P'17)
EKLAVYA (USENIX Security'17)
…

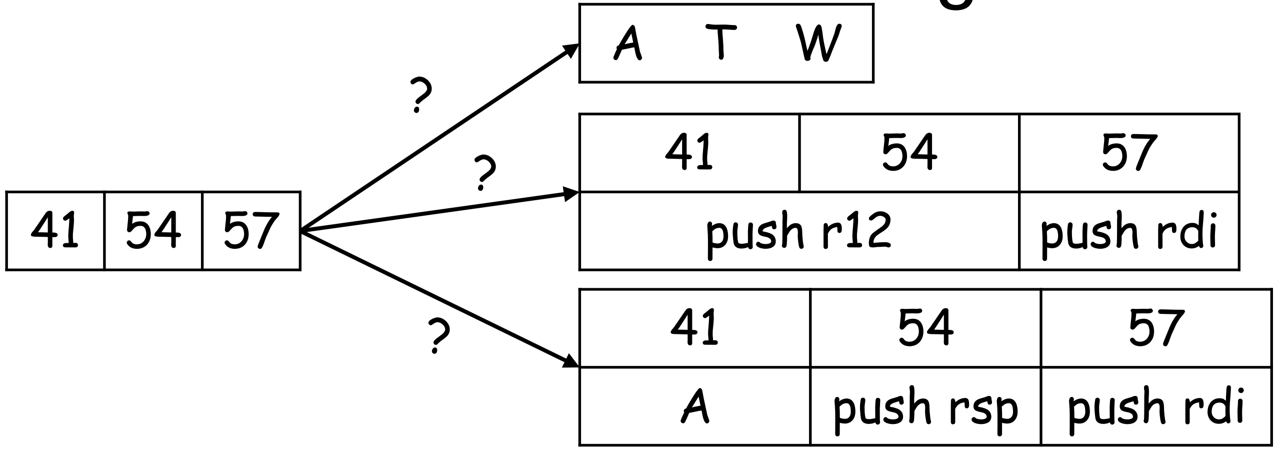
Existing Approaches

Method	Pros	Cons	Efficiency*	
			CPU	GPU
Recursive Disassembly	Low false positive rate	Low coverage, slow, vulnerable to obfuscation	10 – 200 KB/s	N/A
Superset Disassembly (NDSS'18)	Fast, no false negative	85% false positive rate	4 – 5 MB/s	1+ GB/s
Probabilistic Disassembly (ICSE'19)	No false negative	3% false positive rate, slow	4 KB/s	N/A
Shingled Graph Disassembly (PAKDD'14)	Accurate, 2X faster than IDA Pro	Small evaluation dataset, closed source	70 – 200 KB/s	N/A
Datalog Disassembly (USENIX Security'20)	Close to 100% accuracy	Slow, limited format support	4 – 50 KB/s	N/A
XDA (NDSS'21)	Close to 100% accuracy	Slow	140 B/s	47 KB/s

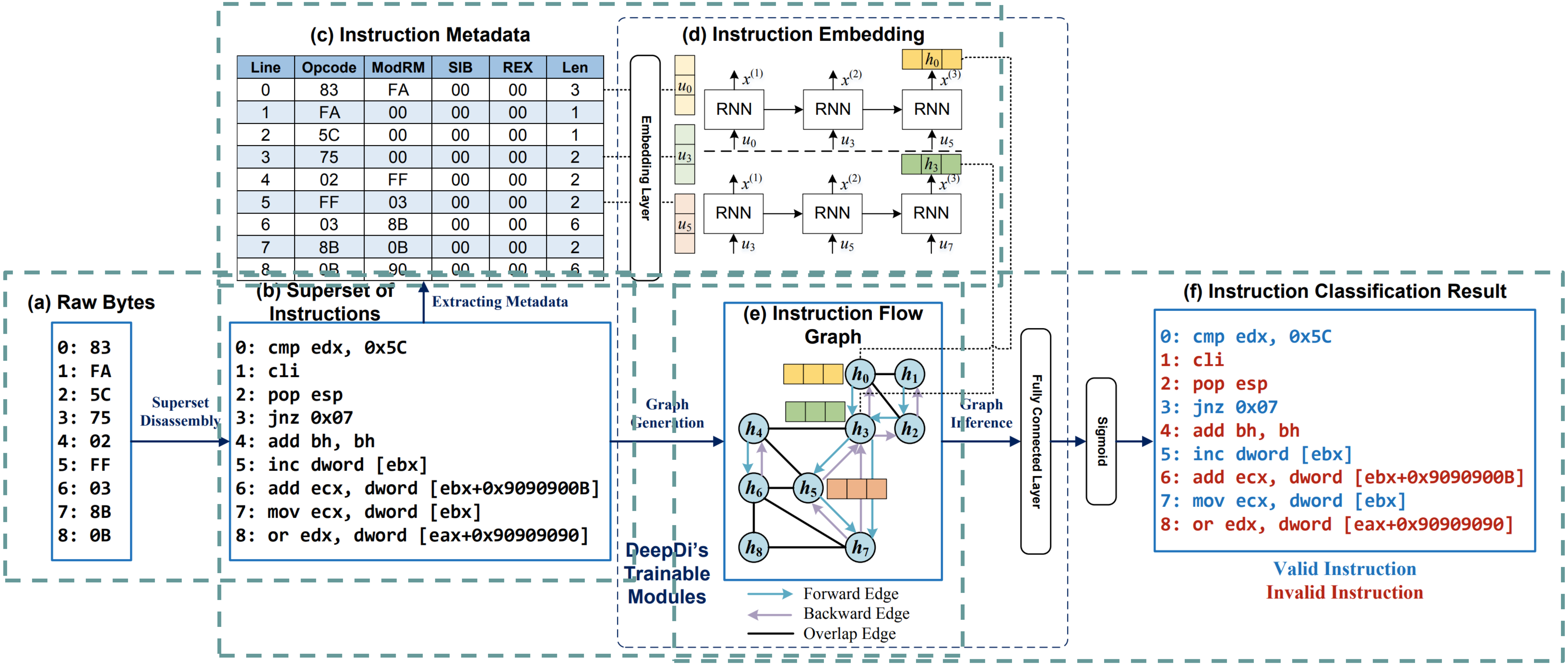
* CPU efficiency is tested on single core, GPU efficiency is tested on GTX 2080 Ti

x86 Instruction Decoding Challenges

- Code and data interleaving
  - String
  - Jump table
- Dense encoding
  - Decoding will almost always succeed
- Instructions are variable-length



Our Approach



Superset Disassembly

(c) Instruction Metadata

Line	Opcode	ModRM	SIB	REX	Len
0	83	FA	00	00	3
1	FA	00	00	00	1
2	5C	00	00	00	1
3	75	00	00	00	2
4	02	FF	00	00	2
5	FF	03	00	00	2
6	03	8B	00	00	6
7	8B	0B	00	00	2
8	0B	90	00	00	6

Extracting Metadata

(a) Raw Bytes

```
0: 83
1: FA
2: 5C
3: 75
4: 02
5: FF
6: 03
7: 8B
8: 0B
```

Superset Disassembly

(b) Superset of Instructions

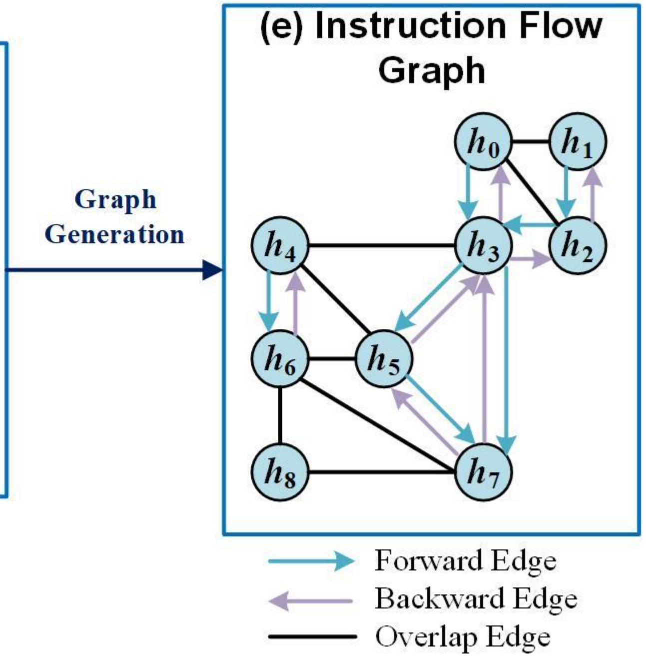
```
0: cmp edx, 0x5C
1: cli
2: pop esp
3: jnz 0x07
4: add bh, bh
5: inc dword [ebx]
6: add ecx, dword [ebx+0x9090900B]
7: mov ecx, dword [ebx]
8: or edx, dword [eax+0x90909090]
```

- Deterministic
- In GPU
  - Modified decoder to leverage GPU parallelism
- No false negative

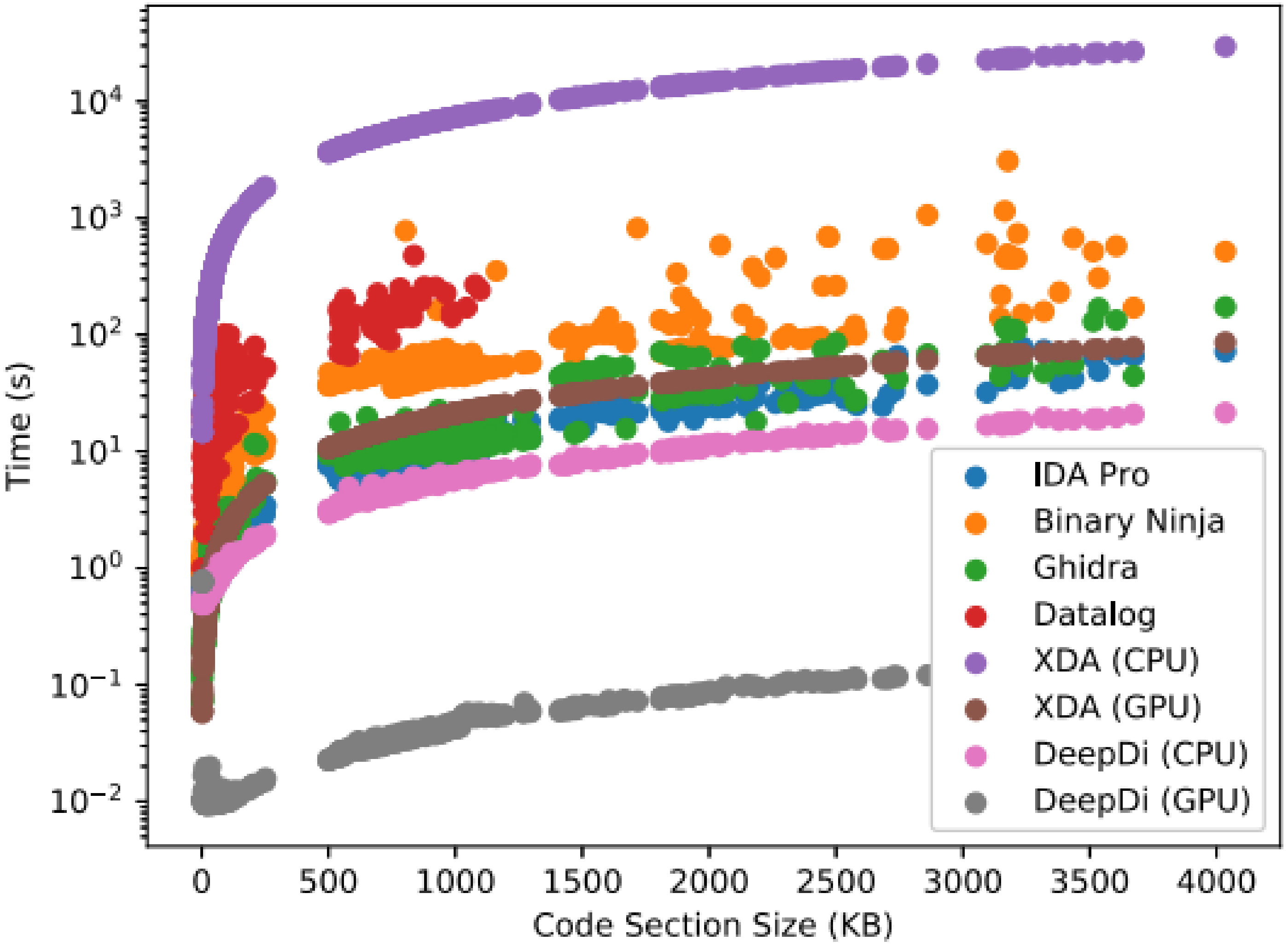
Instruction Flow Graph

(b) Superset of Instructions

```
0: cmp edx, 0x5C
1: cli
2: pop esp
3: jnz 0x07
4: add bh, bh
5: inc dword [ebx]
6: add ecx, dword [ebx+0x9090900B]
7: mov ecx, dword [ebx]
8: or edx, dword [eax+0x90909090]
```



Efficiency Evaluation



Generalizability Evaluation

Table 3: Precision and Recall on Unseen Binaries from an Unseen Compiler

Model	Train \ Test	Instruction								Function							
		Od		O1		O2		Ox		Od		O1		O2		Ox	
		P	R	P	R	P	R	P	R	P	R	P	R	P	R	P	R
DeepDi	O0	98.6	**99.1**	**98.1**	**97.6**	**98.0**	**97.6**	**98.2**	**97.7**	**94.5**	**42.3**	**95.9**	**38.4**	**74.8**	**26.2**	**73.1**	**26.0**
	O1	98.6	**98.9**	**97.2**	**96.6**	97.9	**97.1**	98.0	**97.1**	**94.9**	**60.5**	**93.3**	**76.8**	**72.2**	**72.1**	**69.5**	**71.9**
	O2	98.9	**99.7**	**98.3**	**98.6**	**98.3**	**98.5**	98.2	**98.6**	**89.4**	**47.3**	**86.7**	**61.6**	**82.6**	**55.0**	**83.1**	**53.7**
	O3	98.2	**99.0**	**97.7**	**96.9**	**98.1**	**97.3**	98.1	**97.4**	**80.4**	**21.0**	**78.7**	**39.5**	**72.9**	**30.9**	**74.3**	**32.5**
XDA	O0	**98.7**	38.9	96.1	43.9	97.1	42.1	97.5	42.6	56.9	0.1	77.6	0.7	5.3	0.03	45.5	0.6
	O1	**99.0**	37.5	**97.2**	44.2	**98.1**	42.5	**98.4**	43.0	2.6	0.4	8.9	1.2	2.3	0.9	3.6	1.4
	O2	**99.1**	38.7	97.2	46.5	98.2	44.2	**98.5**	44.6	16.8	0.5	57.6	3.8	29.5	2.9	34.1	3.9
	O3	**98.9**	39.8	97.3	47.6	**98.1**	44.8	**98.4**	45.1	8.7	0.2	40.4	1.4	7.6	0.4	20.5	1.4

Obfuscation Evaluation

Binaries Obfuscated by Linn and Debray’s tool

Disassembler	Precision	Recall	Time
DeepDi (GPU)	**84.1**	**95.2**	**1.2s**
XDA (GPU)	80.2	95.1	282s
IDA Pro	75.8	44.8	262s
Ghidra	69.1	47.0	10,240s

Impact on Malware Classification

Model	Training Accuracy	Testing Loss	Time (GPU)
Gemini	$96.52\% \pm 0.595$	0.134974 ± 0.036	7m
MalConv	$97.81\% \pm 0.659$	0.159165 ± 0.048	48.6s

Gemini: function embedding + min/max pooling + MLP

Model	Training Accuracy	Testing Loss	Time
EMBER	$99.13\% \pm 0.1747$	0.041541 ± 0.0022	21m
EMBER w/ code	$99.40\% \pm 0.2465$	0.024391 ± 0.0018	24m

Summary

- ML/DL is powerful tool for binary analysis

- We have built a pipeline
  - Fast and Accurate Disassembly
  - Pretrained Instruction Embedding
  - Context-aware Basic Block Embedding
  - Function Embedding