Deep Cost-sensitive Kernel Machine for Binary Software Vulnerability Detection

Tuan Nguyen[1], Trung Le[1], Khanh Nguyen[2], Olivier de Vel[3], Paul Montague[3], John Grundy[1], and Dinh Phung[1]

[1] Monash University, Australia

[2] AI Research Lab, Trusting Social, Australia

[3] Defence Science and Technology Group, Australia

Tuan Nguyen

Master Student, Machine Learning and Data Science

Faculty of Information Technology, Monash University

Email: tuan.ng@monash.edu

AGENDA

Introduction

- Software vulnerability detection
- Binary vulnerability detection

Deep Cost-sensitive Kernel Machine

- Data Processing and Embedding
- Cost-sensitive Kernel Machine

Experiment

- Experimental Results
- Model Behaviors

Conclusion

AGENDA

Introduction

- Software vulnerability detection
- Binary vulnerability detection

Deep Cost-sensitive Kernel Machine

- Data Processing and Embedding
- Cost-sensitive Kernel Machine

Experiment

- Experimental Results
- Model Behaviors

Conclusion

GROUP OF EIGHT AUSTRALIA

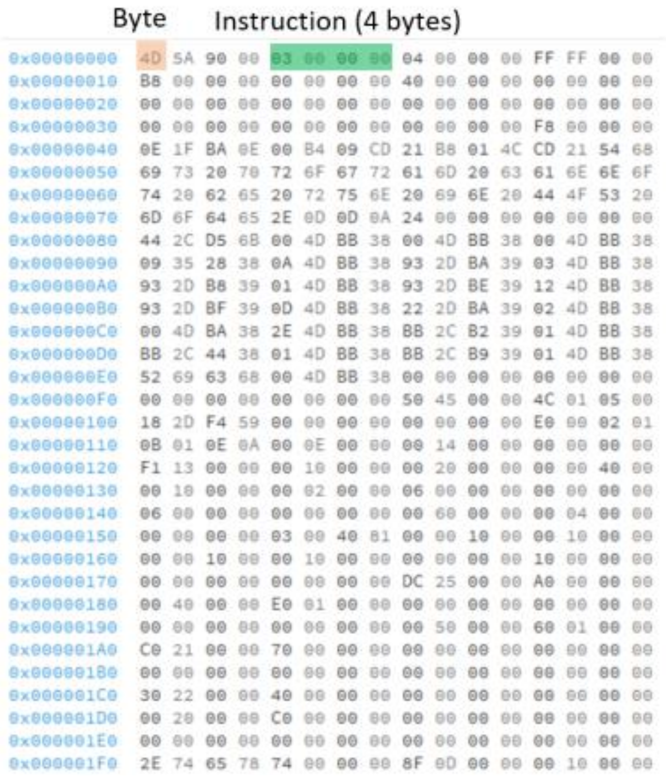
Introduction

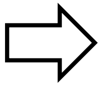
Motivation

Source code vulnerability detection

```c
static int sipsock_read(int *id, int fd, short events, void *ignore)
{
    struct sip_request req;
    struct ast_sockaddr addr;
    int res;
    static char readbuf[65535];

    memset(&req, 0, sizeof(req));
    res = ast_recvfrom(fd, readbuf, sizeof(readbuf) - 1, 0, &addr);
    if (res < 0) {
#if !defined(__FreeBSD__)
        if (errno == EAGAIN)
            ast_log(LOG_NOTICE, "SIP: Received packet with bad UDP checksum\n");
        else
#endif
        if (errno != ECONNREFUSED)
            ast_log(LOG_WARNING, "Recv error: %s\n", strerror(errno));
        return 1;
    }

    readbuf[res] = '\0';

    if (!(req.data = ast_str_create(SIP_MIN_PACKET))) {
        return 1;
    }

    if (ast_str_set(&req.data, 0, "%s", readbuf) == AST_DYNSTR_BUILD_FAILED) {
        return -1;
    }
```

Binary code vulnerability detection





Vulnerable binary code?

- Software vulnerability detection consists of source code and binary code vulnerability detection.
- In practice, binary vulnerability detection is more applicable and impactful than source code vulnerability detection.

Introduction

- There are some problems of binary software vulnerability detection:
  - The shortage of suitable binary datasets labeled as either vulnerable or non-vulnerable.
  - Misclassifying vulnerable code as non-vulnerable is much more severe than many other misclassification decisions.
- Research question: how to take advantages of deep learning, kernel method and cost-sensitive learning to tackle the problems of binary software vulnerability detection?
- Our contributions:
  - We upgrade the tool to create a new real-world binary dataset.
  - We propose a novel Cost-sensitive Kernel Machine that takes into account different kinds of misclassification and unbalanced data nature in binary software vulnerability detection.

AGENDA

Introduction

- Software vulnerability detection
- Binary vulnerability detection

Deep Cost-sensitive Kernel Machine

- Data Processing and Embedding
- Cost-sensitive Kernel Machine

Experiment

- Experimental Results
- Model Behaviors

Conclusion

Deep Cost-sensitive Kernel Machine

Data Processing and Embedding

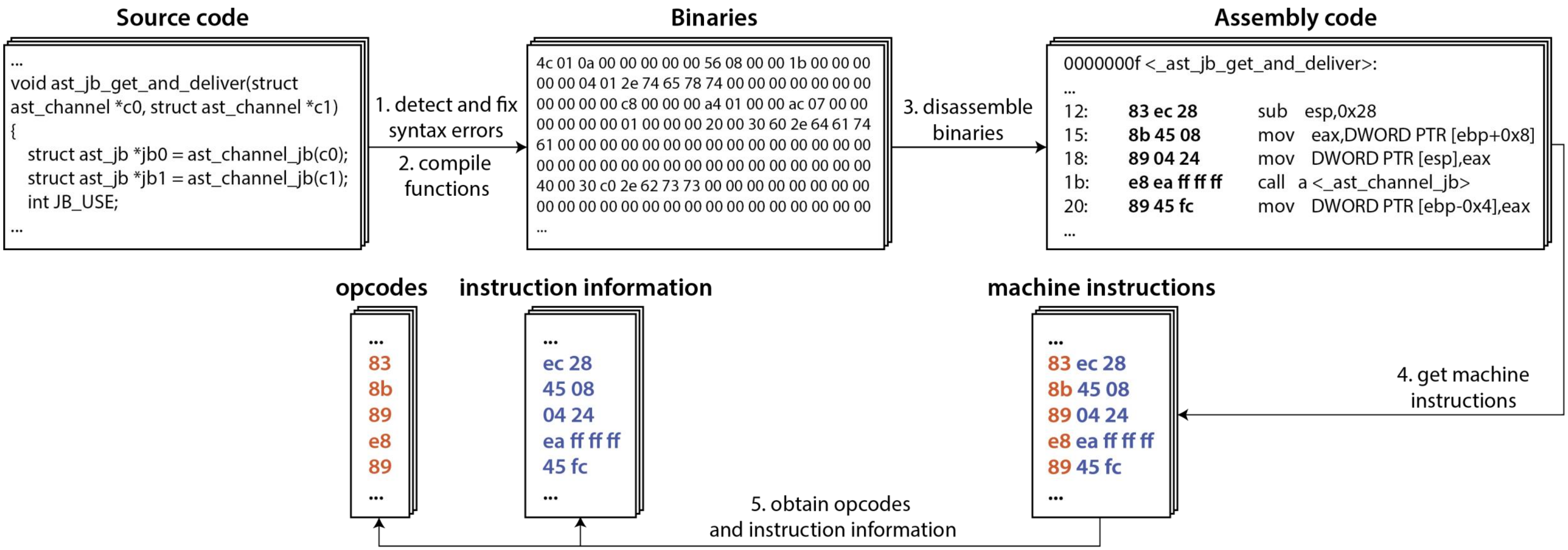


Figure 1. An overview of the data processing and embedding process.

Deep Cost-sensitive Kernel Machine

Data Processing and Embedding

- To embed the opcode, we build a vocabulary of the opcodes, and embed them using one-hot vectors to obtain the opcode embedding $\mathbf{e}_{op}$.
- To embed the instruction information, we compute the frequency vector as follows:
  - We count the frequencies of the hexadecimal bytes to obtain a frequency vector with 256 dimensions.
  - The frequency vector is then multiplied by the embedding matrix to obtain the instruction information embedding $\mathbf{e}_{ii}$.
- Output embedding: $\mathbf{e} = \mathbf{e}_{op} \parallel \mathbf{e}_{ii}$

where $\mathbf{e}_{op} = \text{one\_hot}(op) \times W^{op}$ and $\mathbf{e}_{ii} = \text{freq}(ii) \times W^{ii}$

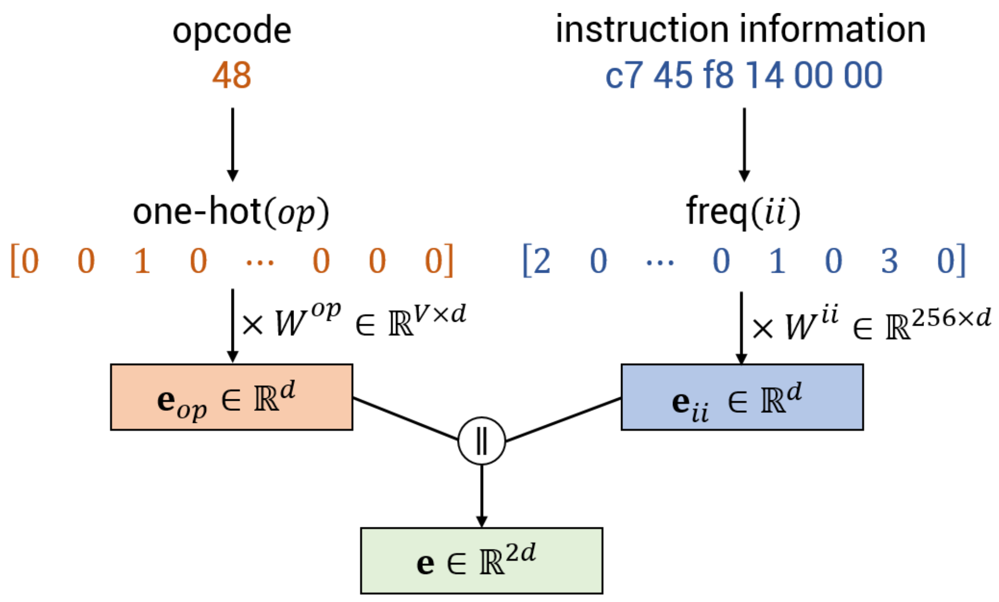


Figure 2. Machine instruction embedding process with examples.

Deep Cost-sensitive Kernel Machine

General Framework

- We fed the machine instruction embedding to a Bidirectional RNN with the sequence length of L to work out the representation $\boldsymbol{h} = \text{concat}(\overrightarrow{\boldsymbol{h}}_L, \overleftarrow{\boldsymbol{h}}_L)$ for the binary, where $\overleftarrow{\boldsymbol{h}}_L$ and $\overrightarrow{\boldsymbol{h}}_L$ are the left and right L-th hidden states of the Bidirectional RNN, respectively.

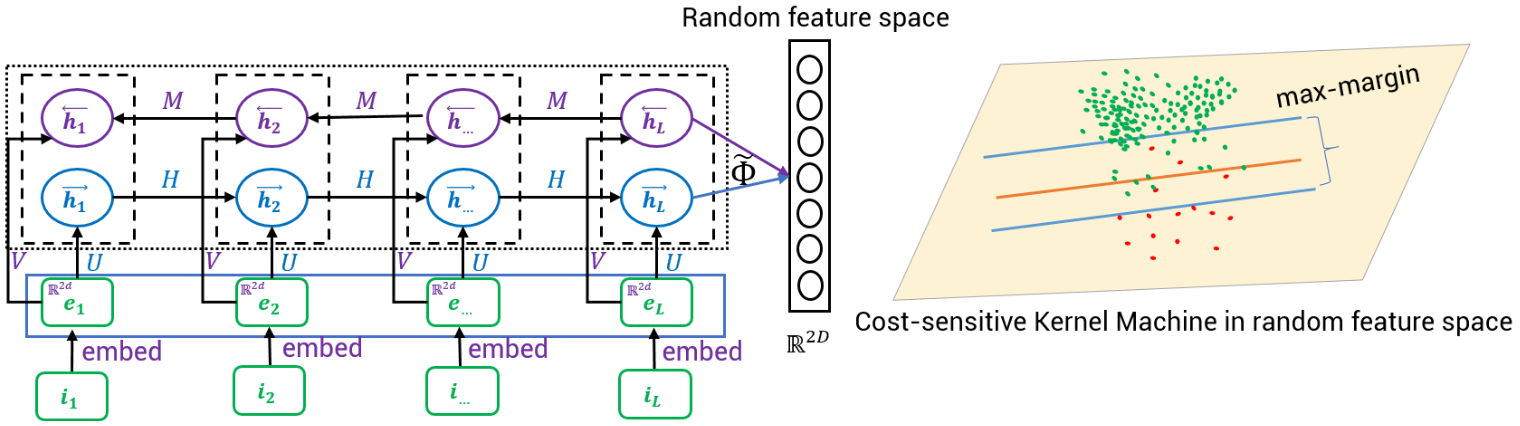


Figure 3. General framework of Deep Cost-sensitive Kernel Machine.

Deep Cost-sensitive Kernel Machine

General Framework

- The vector representation is mapped to a random feature space via a random feature map where we recruit a cost-sensitive kernel machine to classify vulnerable and non-vulnerable binary software.

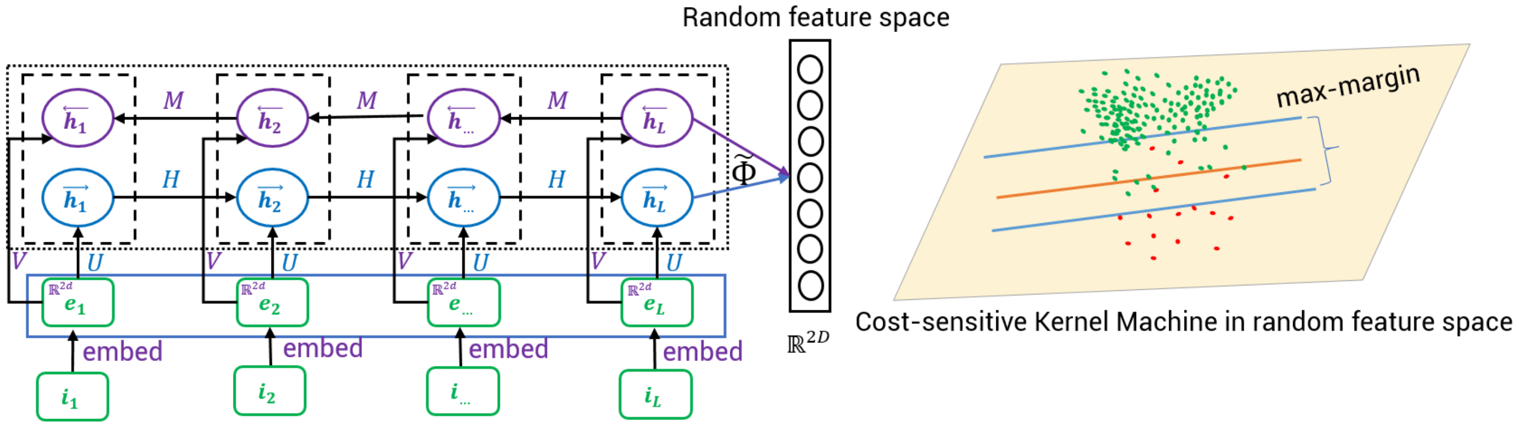


Figure 3. General framework of Deep Cost-sensitive Kernel Machine.

Deep Cost-sensitive Kernel Machine

Cost-sensitive Kernel Machine

- General idea: We first find two parallel hyperplanes $\mathcal{H}_{-1}$ and $\mathcal{H}_1$ in such a way that $\mathcal{H}_{-1}$ and $\mathcal{H}_1$ can separate the vulnerable and non-vulnerable classes, and the margin, which is the distance between the two parallel hyperplanes $\mathcal{H}_{-1}$ and $\mathcal{H}_1$, is maximized.
- We then find the optimal decision hyperplane $\mathcal{H}_d$ by searching in the strip formed by $\mathcal{H}_{-1}$ and $\mathcal{H}_1$.

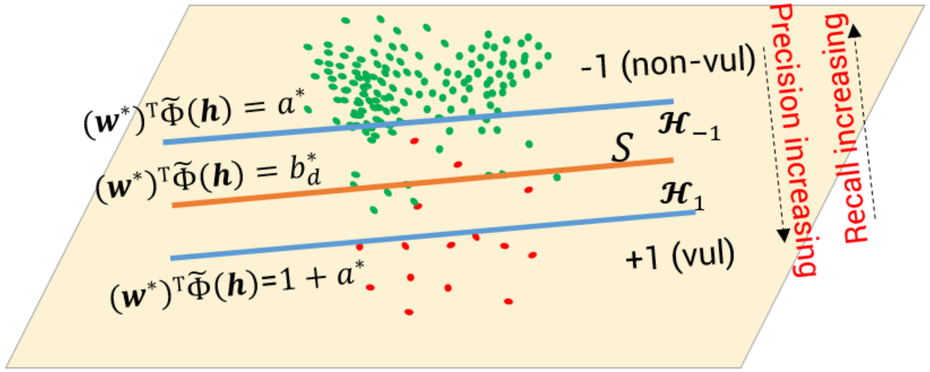


Figure 4. Cost-sensitive kernel machine in the random feature space.

Deep Cost-sensitive Kernel Machine

Cost-sensitive Kernel Machine

- Finding the optimal decision hyperplane:

We define the cost-sensitive loss and the optimal decision hyperplane $(\boldsymbol{w}^*)^T\widetilde{\Phi}(\boldsymbol{h}) - b_d^* = 0$ as:

$$l(\boldsymbol{w}^*, b_d^m) = \theta \sum_{y_{i_k}=1} \mathbb{I}_{(\boldsymbol{w}^*)^T\widetilde{\Phi}\left(\boldsymbol{h}_{i_k}\right)-b_d^m<0} + \sum_{y_{i_k}=-1} \mathbb{I}_{(\boldsymbol{w}^*)^T\widetilde{\Phi}\left(\boldsymbol{h}_{i_k}\right)-b_d^m>0}$$

$m^* = \underset{1\le m\le M+1}{\text{argmin}}\ l(\boldsymbol{w}^*, b_d^m)$ and $b_d^* = b_d^{m^*}$, where $\theta = \#\text{non} - \text{vul}: \#\text{vul} >> 1.$

Steps to find the optimal decision hyperplane:

① search the hyperplane in the strip.

② compute the cost-sensitive loss.

③ obtain the optimal decision hyperplane which is corresponding to the minimal loss value.

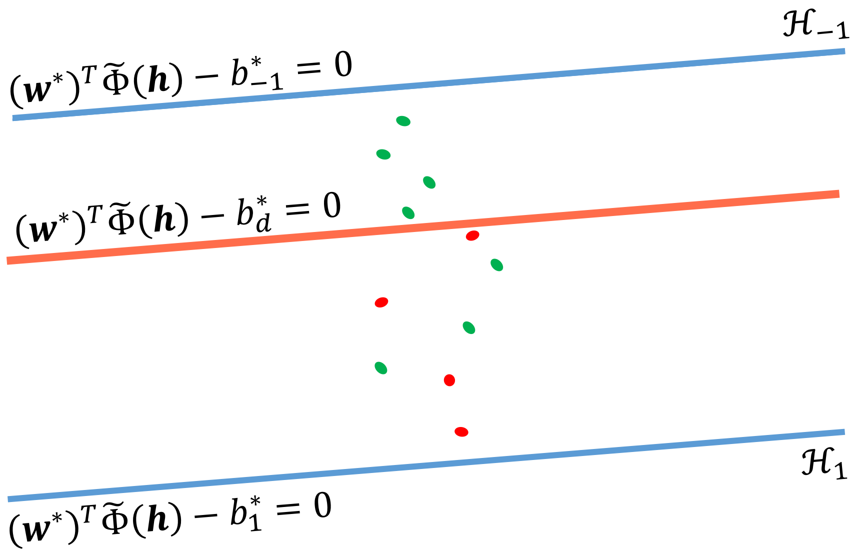


Figure 5. Finding the Optimal Decision Hyperplane.

AGENDA

Introduction

- Software vulnerability detection
- Binary vulnerability detection

Deep Cost-sensitive Kernel Machine

- Data Processing and Embedding
- Cost-sensitive Kernel Machine

Experiment

- Experimental Results
- Model Behaviors

Conclusion

GROUP OF EIGHT AUSTRALIA

Experiment

Datasets

- We collected the source code from two datasets on GitHub: NDSS18[1] and six open-source projects[2] then processed to create 2 labeled binary datasets.
- We split the data into 80% for training, 10% for validation, and the remaining 10% for testing.
- We ran our experiments on a computer with an Intel Xeon Processor E5-1660 which had 8 cores at 3.0 GHz and 128 GB of RAM.

Table 1. The statistics of the two binary datasets.

		#Non-vul	#Vul	#Binaries
NDSS18	Windows	8,999	8,978	17,977
	Linux	6,955	7,349	14,304
	Whole	15,954	16,327	32,281
6 open-source	Windows	26,621	328	26,949
	Linux	25,660	290	25,950
	Whole	52,281	618	52,899

- The implementation of our model and the binary datasets for reproducing the experimental results can be found online at https://github.com/tuanrpt/DCKM.

[1] https://github.com/CGCL-codes/VulDeePecker
[2] https://github.com/DanielLin1986/TransferRepresentationLearning

Experiment

Experimental Results

- The experimental results (%) except for the column CS of the proposed method compared with the baselines on NDSS18 binary dataset (left) and the binary dataset from the six open-source projects (right). Pre, Rec, and CS are shorthand for the performance measures precision, recall, and cost-sensitive loss, respectively.

Datasets	Windows					Linux					Whole				
Methods	**Pre**	**F1**	**Rec**	**AUC**	**CS**	**Pre**	**F1**	**Rec**	**AUC**	**CS**	**Pre**	**F1**	**Rec**	**AUC**	**CS**
Para2Vec	17.5	24.1	38.9	67.6	0.98	36.4	44.4	57.1	77.6	0.83	28.6	26.7	25.0	61.9	0.96
Vdiscover	58.8	57.1	55.6	77.4	0.90	52.9	58.1	64.3	81.6	0.68	48.4	47.6	46.9	72.9	0.93
BRNN-C	80.0	84.2	88.9	94.2	0.89	76.9	74.1	71.4	85.5	0.65	84.6	75.9	68.7	84.2	0.87
BRNN-D	77.8	77.8	77.8	88.7	0.92	92.3	88.9	85.7	92.8	0.68	85.2	78.0	71.9	85.8	0.81
VulDeePecker	70.0	73.7	77.8	88.6	0.98	80.0	82.8	85.7	92.6	0.70	85.2	78.0	71.9	85.8	0.84
BRNN-SVM	79.0	81.1	83.3	91.4	0.98	92.3	88.9	85.7	92.8	0.68	85.7	80.0	75.0	87.4	0.84
Att-BGRU	**92.3**	77.4	66.7	83.3	0.97	92.3	88.9	85.7	92.8	0.68	86.5	79.3	71.9	85.8	0.82
Text CNN	**92.3**	77.4	66.7	83.3	0.99	91.7	84.6	78.6	89.2	0.74	84.6	75.9	68.7	84.2	0.85
MDSAE	77.7	86.4	**97.2**	84.4	0.11	80.6	88.3	**97.7**	86.8	0.05	78.4	**87.1**	**98.1**	85.2	0.72
OC-DeepSVDD	91.7	73.3	61.1	80.5	0.19	**100**	83.3	71.4	85.7	0.14	85.5	78.1	71.9	83.1	0.84
DCKM	84.2	**86.5**	88.9	**94.3**	**0.06**	92.9	**92.9**	92.9	**96.4**	**0.03**	**87.1**	85.7	84.4	**92.1**	**0.58**

Datasets	Windows					Linux					Whole				
Methods	**Pre**	**F1**	**Rec**	**AUC**	**CS**	**Pre**	**F1**	**Rec**	**AUC**	**CS**	**Pre**	**F1**	**Rec**	**AUC**	**CS**
Para2Vec	28.9	31.0	33.3	66.2	0.96	19.2	24.0	32.1	65.3	0.98	28.1	26.9	25.8	62.5	0.97
Vdiscover	23.3	22.2	21.2	60.2	0.98	42.1	34.0	28.6	64.1	0.92	18.0	13.9	11.3	55.3	0.98
BRNN-C	42.9	25.5	18.2	59.0	0.97	53.9	34.2	25.0	62.4	0.93	43.2	32.3	25.8	62.7	0.95
BRNN-D	30.8	27.1	24.2	61.8	0.96	46.2	29.3	21.4	60.6	0.96	36.7	25.3	19.4	59.5	0.98
VulDeePecker	31.6	23.1	18.2	58.9	0.97	53.9	34.2	25.0	62.4	0.94	65.5	41.8	30.7	65.2	0.93
BRNN-SVM	73.9	60.7	51.5	75.6	0.98	87.5	63.6	50.0	75.0	0.99	65.6	65.0	64.5	82.1	0.91
Att-BGRU	70.8	59.7	51.5	75.6	0.92	**100**	56.4	39.3	69.7	0.93	85.1	73.4	64.5	82.2	0.91
Text CNN	**100**	70.6	54.6	77.3	0.90	81.8	72.0	**64.3**	82.0	0.89	**100**	74.8	59.7	79.8	0.91
MDSAE	88.2	60.0	45.5	72.7	0.91	60.0	41.9	32.1	66.0	0.93	82.4	74.3	67.7	83.8	0.90
OC-DeepSVDD	**100**	77.8	63.6	81.8	0.83	88.9	69.6	57.1	78.5	0.90	**100**	70.8	54.8	77.4	0.89
DCKM	79.4	**80.6**	**81.8**	**90.8**	**0.78**	90.0	**75.0**	**64.3**	**82.1**	**0.85**	90.3	**90.3**	**90.3**	**95.1**	**0.56**

Experiment

Model Behaviors

- Our optimal decision hyperplane marked with the red stars can achieve the minimal cost-sensitive loss, while maintaining comparable F1 and AUC scores compared with the optimal-F1 hyperplane marked with the purple stars.

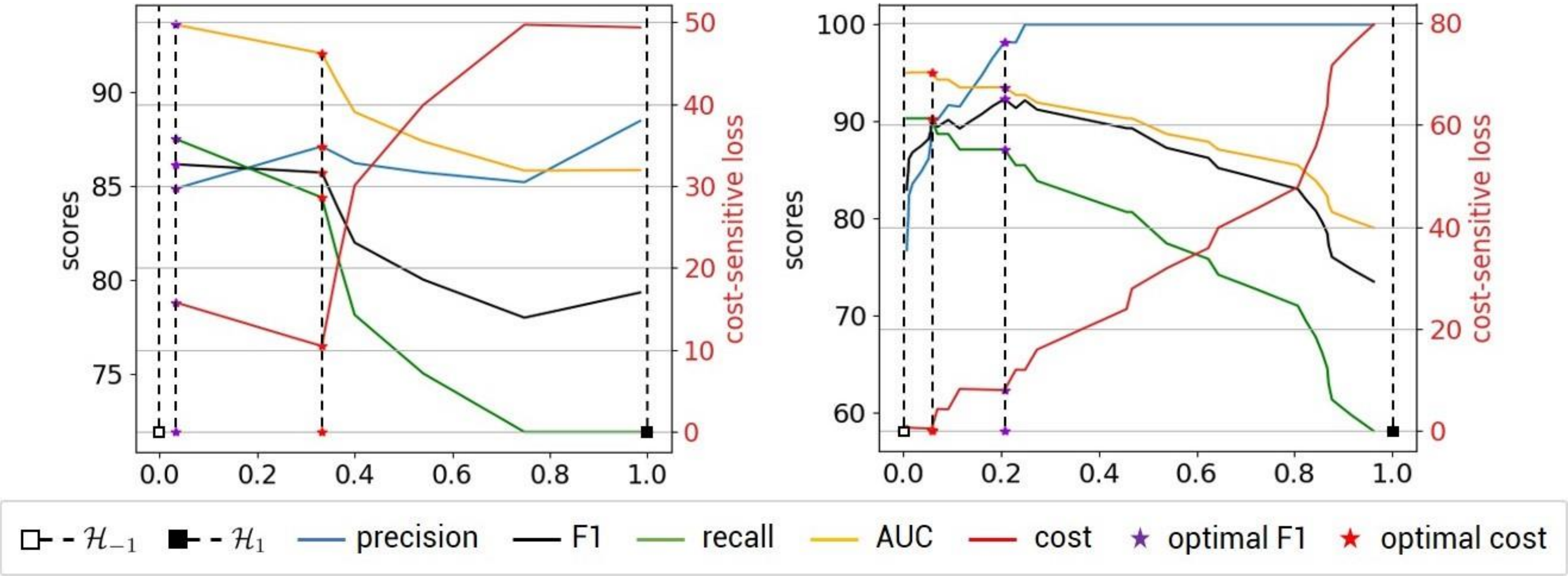


Figure 7. The variation of predictive scores when sliding the hyperplane in the strip formed by $\mathcal{H}_{-1}$ and $\mathcal{H}_1$: on the NDSS18 (left) and the dataset from six open-source projects (right).

AGENDA

Introduction

- Software vulnerability detection
- Binary vulnerability detection

Deep Cost-sensitive Kernel Machine

- Data Processing and Embedding
- Cost-sensitive Kernel Machine

Experiment

- Experimental Results
- Model Behaviors

Conclusion

Conclusion

- In this work, we have leveraged deep learning and kernel methods to propose the Deep Cost-sensitive Kernel Machine for tackling binary software vulnerability detection.
- Our proposed method inherits the advantages of deep learning methods in efficiently tackling structural data and kernel methods in learning the characteristic of vulnerable binary examples with high generalization capacity.
- We upgrade the tool to create a new real-world binary dataset.
- The experimental results have shown a convincing outperformance of our proposed method compared to the state-of-the-art baselines.

Thanks for your attention

Q&A