24th Pacific-Asia Conference on Knowledge Discovery and Data Mining 2020 (PAKDD 2020), 11-14 May, Singapore

# 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
{tuan.nguyen, trunglm, john.grundy, dinh.phung}@monash.edu
[2] AI Research Lab, Trusting Social, Australia
khanh.nguyen@trustingsocial.com
[3] Defence Science and Technology Group, Australia
{olivier.devel, paul.montague}@dst.defence.gov.au

**Abstract.** Owing to the sharp rise in the severity of the threats imposed by software vulnerabilities, software vulnerability detection has become an important concern in the software industry, such as the embedded systems industry, and in the field of computer security. Software vulnerability detection can be carried out at the source code or binary level. However, the latter is more impactful and practical since when using commercial software, we usually only possess binary software. In this paper, we leverage deep learning and kernel methods to propose the Deep Cost-sensitive Kernel Machine, a method that 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 conduct experiments on two real-world binary datasets. The experimental results have shown a convincing outperformance of our proposed method over the baselines.

## 1 Introduction

Software vulnerabilities are specific flaws or oversights in a piece of software that can potentially allow attackers exploit the code to perform malicious acts including exposing or altering sensitive information, disrupting or destroying a system, or taking control of a computer system or program. Because of the ubiquity of computer software and the growth and the diversity in its development process, a great deal of computer software potentially possesses software vulnerabilities. This makes the problem of software vulnerability detection an important concern in the software industry and in the field of computer security.

Software vulnerability detection consists of source code and binary code vulnerability detection. Due to a large loss of the syntactic and semantic information provided by high-level programming languages during the compilation process, binary code vulnerability detection is significantly more difficult than source code vulnerability detection.

**Acknowledgement**: This research was supported under the Defence Science and Technology Group's Next Generation Technologies Program.

In addition, in practice, binary vulnerability detection is more applicable and impactful than source code vulnerability detection. The reason is that when using a commercial application, we only possess its binary and usually do not possess its source code. The ability to detect the presence or absence of vulnerabilities in binary code, without getting access to source code, is therefore of major importance in the context of computer security. Some work has been proposed to detect vulnerabilities at the binary code level when source code is not available, notably work based on fuzzing, symbolic execution [1], or techniques using handcrafted features extracted from dynamic analysis [4]. Recently, the work of [10] has pioneered learning automatic features for binary software vulnerability detection. In particular, this work was based on a Variational Auto-encoder [7] to work out representations of binary software so that representations of vulnerable and non-vulnerable binaries are encouraged to be maximally different for vulnerability detection purposes, while still preserving crucial information inherent in the original binaries.

By nature, datasets for binary software vulnerability detection are typically imbalanced in the sense that the number of vulnerabilities is very small compared to the volume of non-vulnerable binary software. Another important trait of binary software vulnerability detection is that misclassifying vulnerable code as non-vulnerable is much more severe than many other misclassification decisions. In the literature, kernel methods in conjunction with the max-margin principle have shown their advantages in tackling imbalanced datasets in the context of anomaly and novelty detection [21,13,18]. The underlying idea is *to employ the max-margin principle to learn the domain of normality*, which is decomposed into a set of contours enclosing normal data that helps distinguish normality against abnormality. However, kernel methods are not able to efficiently handle sequential machine instructions in binary software. In contrast, deep recursive networks (e.g., recurrent neural networks or bidirectional recurrent neural networks) are very efficient and effective in tackling and exploiting temporal information in sequential data like sequential machine instructions in binary software.

To cope with the difference in the severity level of the kinds of misclassification, cost-sensitive loss has been leveraged with kernel methods in some previous works, notably [2,12,5]. However, these works either used non-decomposable losses or were solved in the dual form, which makes them less applicable to leverage with deep learning methods in which stochastic gradient descent method is employed to solve the corresponding optimization problem.

To smoothly enable the incorporation of kernel methods, cost-sensitive loss, and deep learning in the context of binary code vulnerability detection, we propose a novel Cost-sensitive Kernel Machine (CKM) which is developed based on the max-margin principle to find two optimal parallel hyperplanes and employs cost sensitive loss to find the best decision hyperplane. In particular, our CKM first aims to learn two parallel hyperplanes that can separate vulnerability and non-vulnerability, while the margin which is defined as the distance between the two parallel hyperplanes is maximized. The optimal decision hyperplane of CKM is sought in the strip formed by the two parallel hyperplanes. To take into account the difference in importance level of two kinds of misclassification, we employ a cost-sensitive loss, where the misclassification of vulnerability as non-vulnerability is assigned a higher cost.

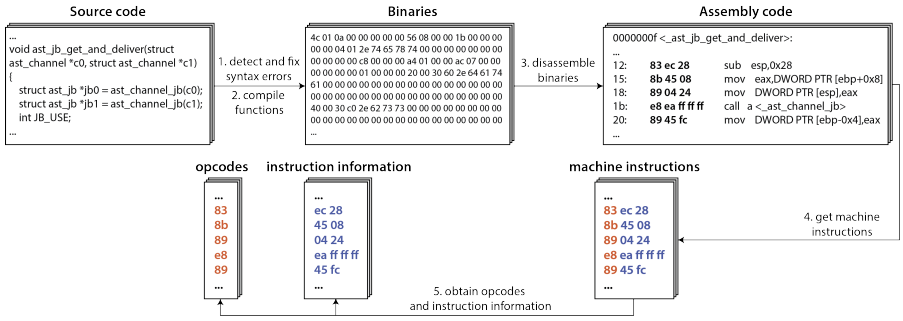


**Fig. 1.** An overview of the *data processing and embedding* process.

We conduct experiments over two datasets, the NDSS18 binary dataset whose source code was collected and compiled to binaries in [15,10] and binaries compiled from 6 open-source projects, which is a new dataset created by us. We strengthen and extend the tool developed in [10] to allow it to be able to handle more errors for compiling the source code in the six open-source projects into binaries. Our experimental results over these two binary datasets show that our proposed DCKM outperforms the baselines by a wide margin.

The major contributions of our work are as follows:

- We upgrade the tool developed in [10] to create a new real-world binary dataset.
- We propose a novel Cost-sensitive Kernel Machine that takes into account the difference in incurred costs of different kinds of misclassification and imbalanced data nature in binary software vulnerability detection. This CKM can be plugged neatly into a deep learning model and be trained using back-propagation.
- We leverage deep learning, kernel methods, and a cost-sensitive based approach to build a novel Deep Cost-sensitive Kernel Machine that outperforms state-of-the-art baselines on our experimental datasets by a wide margin.

## 2 Our Approach: Deep Cost-sensitive Kernel Machine

By incorporating deep learning and kernel methods, we propose a Deep Cost-sensitive Kernel Machine (DCKM) for binary software vulnerability detection. In particular, we use a bidirectional recurrent neural network (BRNN) to summarize a sequence of machine instructions in binary software into a representation vector. This vector is then mapped into a Fourier random feature space via a finite-dimensional random feature map [19,11,17,9,14]. Our proposed Cost-sensitive Kernel Machine (CKM) is invoked in the random feature space to detect vulnerable binary software. Note that the Fourier random feature map which is used in conjunction with our CKM and BRNN enables our DCKM to be trained nicely via back-propagation.

### 2.1 Data Processing and Embedding

Figure 1 presents an overview of the code data processing steps required to obtain the core parts of machine instructions from source code. From the source code repository, we identify the code functions and then fix any syntax errors using our automatic tool. The tool also invokes the *gcc* compiler to compile compilable functions into binaries.

Subsequently, utilizing the *objdump*[4] tool, we disassemble the binaries into assembly code. Each function corresponds to an assembly code file. We then process the assembly code files to obtain a collection of machine instructions and eventually use the *Capstone*[5] framework to extract their core parts. Each core part in a machine instruction consists of two components: the opcode and the operands, called the instruction information (a sequence of bytes in hexadecimal format, i.e., memory location, registers, etc.). The opcode indicates the type of operation, whilst the operands contain the necessary information for the corresponding operation. Since both opcode and operands are important, we embed both the opcode and instruction information into vectors and then concatenate them.

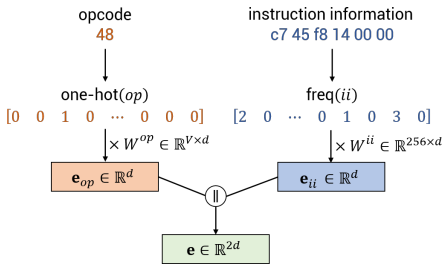


**Fig. 2.** Machine instruction embedding process with examples. The *opcode embedding* $\mathbf{e}_{op}$ is concatenated with *instruction information embedding* $\mathbf{e}_{ii}$ to obtain the *output embedding* $\mathbf{e}$, a $2d$-dimensional vector.

To embed the opcode, we undertake some preliminary analysis and find that there were a few hundred opcodes in our dataset. We then build a vocabulary of the opcodes, and after that embed them using one-hot vectors to obtain the *opcode embedding* $\mathbf{e}_{op}$.

To embed the instruction information, we first compute the frequency vector as follows. We consider the operands as a sequence of hexadecimal bytes (i.e., $00, 01$ to $FF$) and 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* $\mathbf{e}_{ii}$.

More specifically, the *output embedding* is $\mathbf{e} = \mathbf{e}_{op} \parallel \mathbf{e}_{ii}$ where $\mathbf{e}_{op} = \text{one-hot}(op) \times W^{op}$ and $\mathbf{e}_{ii} = \text{freq}\,(ii) \times W^{ii}$ with the opcode $op$, the instruction information $ii$, one-hot vector one-hot$(op)$, frequency vector freq $(ii)$, and the embedding matrices $W^{op} \in \mathbb{R}^{V \times d}$ and $W^{ii} \in \mathbb{R}^{256 \times d}$, where $V$ is the vocabulary size of the opcodes and $d$ is the embedding dimension. The process of embedding machine instructions is presented in Figure 2.

### 2.2 General Framework of Deep Cost-sensitive Kernel Machine

We now present the general framework for our proposed Deep Cost-sensitive Kernel Machine. As shown in Figure 3, given a binary $\boldsymbol{x}$, we first embed its machine instructions into vectors (see Section 2.1); the resulting vectors are then fed to a Bidirectional RNN with the sequence lenght of $L$ to work out the representation $\boldsymbol{h} = \text{concat}\left(\overleftarrow{\boldsymbol{h}}_L, \overrightarrow{\boldsymbol{h}}_L\right)$ for the binary $\boldsymbol{x}$, where $\overleftarrow{\boldsymbol{h}}_L$ and $\overrightarrow{\boldsymbol{h}}_L$ are the left and right $L$-th hidden states (the left and right last hidden states) of the Bidirectional RNN, respectively. Finally, the vector representation $\boldsymbol{h}$ is mapped to a random feature space via a random feature map $\tilde{\Phi}(\cdot)$ [19] where we recruit a cost-sensitive kernel machine (see Section 2.3) to classify vulnerable and non-vulnerable binary software. Note that the formulation for $\tilde{\Phi}$ is as follows:

[4] https://www.gnu.org/software/binutils/
[5] https://www.capstone-engine.org

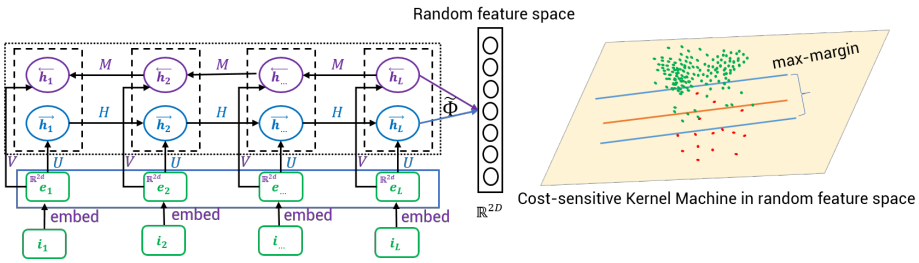


**Fig. 3.** General framework of Deep Cost-sensitive Kernel Machine.

$$\tilde{\Phi}(\boldsymbol{h}) = \left[\frac{1}{\sqrt{D}}\cos\left(\boldsymbol{\omega}_i^{\mathsf{T}}\boldsymbol{h}\right), \frac{1}{\sqrt{D}}\sin\left(\boldsymbol{\omega}_i^{\mathsf{T}}\boldsymbol{h}\right)\right]_{i=1}^{D} \in \mathbb{R}^{2D} \tag{1}$$

where $\boldsymbol{\omega}_1, \ldots, \boldsymbol{\omega}_D$ are the Fourier random elements as in [19] and the dimension of random feature space is hence $2D$.

We note that the use of a random feature map in conjunction with cost-sensitive kernel machine and bi-directional RNN allows us to easily do back-propagation when training our Deep Cost-sensitive Kernel Machine. In addition, let us denote the training set of binaries and their labels by $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{N}$ where $\boldsymbol{x}_i$ is a binary including many machine instructions and $y_i \in \{-1; 1\}$ where the label $-1$ stands for vulnerable binary and the label $1$ stands for non-vulnerable binary. Assume that after feeding the binaries $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$ into the corresponding BRNN as described above, we obtain the representations $\boldsymbol{h}_1, \ldots, \boldsymbol{h}_N$. We then map these representations to the random feature space via the random feature map $\tilde{\Phi}(\cdot)$ as defined in Eq. (1). We finally construct a cost-sensitive kernel machine (see Section 2.3) in the random feature space to help us distinguish vulnerability against non-vulnerability.

### 2.3 Cost-sensitive Kernel Machine

**General Idea of Cost-sensitive Kernel Machine** We first find two parallel hyperplanes $\mathcal{H}_{-1}$ and $\mathcal{H}_1$ in such a way that $\mathcal{H}_{-1}$ separates the non-vulnerable and vulnerable classes, $\mathcal{H}_1$ separates 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 $\mathcal{H}_1$ (see Figure 4).

**Formulations of The Hard and Soft Models** Let us denote the equations of $\mathcal{H}_{-1}$ and $\mathcal{H}_1$ by $\mathcal{H}_{-1} : \mathbf{w}^{\mathsf{T}}\tilde{\Phi}(\boldsymbol{h}) - b_{-1} = 0$ and $\mathcal{H}_1 : \mathbf{w}^{\mathsf{T}}\tilde{\Phi}(\boldsymbol{h}) - b_1 = 0$ where $b_1 > b_{-1}$. The margin is hence formulated as $d(\mathcal{H}_{-1}, \mathcal{H}_1) = \frac{|b_1 - b_{-1}|}{\|\mathbf{w}\|} = \frac{b_1 - b_{-1}}{\|\mathbf{w}\|}$. We arrive at the optimization problem:

$$\begin{aligned} &\max_{\mathbf{w}, b_{-1}, b_1} \left(\frac{b_1 - b_{-1}}{\|\mathbf{w}\|}\right) \\ &\text{s.t. } : y_i\left(\mathbf{w}^{\mathsf{T}}\tilde{\Phi}(\boldsymbol{h}_i) - b_{-1}\right) \geq 0, \forall i = 1, \ldots, N \\ &\quad y_i\left(\mathbf{w}^{\mathsf{T}}\tilde{\Phi}(\boldsymbol{h}_i) - b_1\right) \geq 0, \forall i = 1, \ldots, N \end{aligned}$$

It is worth noting that the margin $d(\mathcal{H}_{-1}, \mathcal{H}_1)$ is invariant if we scale $(\mathbf{w}, b_{-1}, b_1)$ by a factor $k > 0$ as $(k\mathbf{w}, kb_{-1}, kb_1)$. Therefore, we can safely assume that $b_1 - b_{-1} = 1$, and hence the following optimization problem:

$$\min_{\mathbf{w},a} \left( \frac{1}{2} \|\mathbf{w}\|^2 \right)$$
$$\text{s.t. } : y_i \left( \mathbf{w}^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_i) - a \right) \geq 0, \forall i = 1, \ldots, N$$
$$y_i \left( \mathbf{w}^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_i) - 1 - a \right) \geq 0, \forall i = 1, \ldots, N$$

where $b_{-1} = a$ and $b_1 = 1 + a$.

Invoking slack variables, we obtain the soft model:

$$\min_{\mathbf{w},a} \left( \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{N} \sum_{i=1}^{N} (\xi_i + \psi_i) \right)$$
$$\text{s.t. } : y_i \left( \mathbf{w}^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_i) - a \right) \geq -\xi_i, \forall i = 1, \ldots, N$$
$$y_i \left( \mathbf{w}^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_i) - 1 - a \right) \geq -\psi_i, \forall i = 1, \ldots, N$$

where $[\xi_i]_{i=1}^N$ and $[\psi_i]_{i=1}^N$ are non-negative slack variables and $\lambda > 0$ is the regularization parameter.

The primal form of the soft model optimization problem is hence of the following form:

$$\min_{\mathbf{w},a} \left( \frac{\lambda}{2} \|\mathbf{w}\|^2 + \frac{1}{N} \sum_{i=1}^{N} \left( \max \left\{ 0, -y_i \left( \mathbf{w}^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_i) - a \right) \right\} + \right. \right.$$
$$\left. \left. \max \left\{ 0, -y_i \left( \mathbf{w}^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_i) - 1 - a \right) \right\} \right) \right) \quad (2)$$

**Finding The Optimal Decision Hyperplane** After solving the optimization problem in Eq. (2), we obtain the optimal solution $\left(\mathbf{w}^*, b^*_{-1}, b^*_1\right)$ where $b^*_{-1} = a^*$ and $b^*_1 = 1 + a^*$ for the two parallel hyperplanes. Let us denote the strip $\mathcal{S}$ formed by the two parallel hyperplanes and the set of training examples $\mathcal{I}$ in this strip as:

$$\mathcal{S} = \left\{ \boldsymbol{v} \mid (\mathbf{w}^*)^\mathsf{T} \boldsymbol{u} - b^*_1 \leq \boldsymbol{v} \leq (\mathbf{w}^*)^\mathsf{T} \boldsymbol{u} - b^*_{-1} \text{ for some } \boldsymbol{u} \right\}$$
$$\mathcal{I} = \left\{ i \mid \tilde{\Phi}(\boldsymbol{h}_i) \in \mathcal{S}, 1 \leq i \leq N \right\}$$

where $\boldsymbol{u}$, $\boldsymbol{v}$ lie in the random feature space $\mathbb{R}^{2D}$.

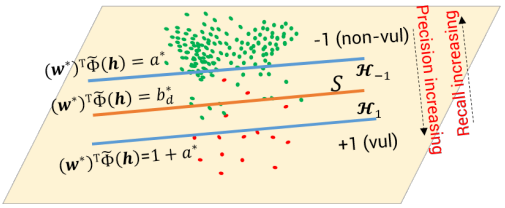


**Fig. 4.** Cost-sensitive kernel machine in the random feature space. We first find two optimal parallel hyperplanes $\mathcal{H}_1$ and $\mathcal{H}_{-1}$ with maximal margin and then search for the optimal decision hyperplane in the strip $\mathcal{S}$ formed by $\mathcal{H}_1$ and $\mathcal{H}_{-1}$ to balance between precision and recall for minimizing the cost-sensitive loss and obtaining a good F1 score.

As shown in Figure 4, when sliding a hyperplane from $\mathcal{H}_1$ to $\mathcal{H}_{-1}$, the recall is increased, but the precision is decreased. In contrast, when sliding a hyperplane from $\mathcal{H}_{-1}$ to $\mathcal{H}_1$, the precision is increased, but the recall is decreased. We hence desire to find out the optimal decision hyperplane to balance between precision and recall for minimizing the cost-sensitive loss and obtaining good F1 scores. We also conduct intensive experiments on real datasets to empirically demonstrate this intuition in Section 3.4.

Inspired by this observation, we seek the optimal decision hyperplane $\mathcal{H}_d$ by minimizing the cost-sensitive loss for the training examples inside the strip $\mathcal{S}$, where we treat the two kinds of misclassification unequally. In particular, the cost of misclassifying a non-vulnerability as a vulnerability is 1, while misclassifying a vulnerability as a non-vulnerability is $\theta$. The value of $\theta$, the relative cost between two kinds of misclassification, is set depending on specific applications. In this application, we set $\theta = \#\text{non-vul} : \#\text{vul} >> 1$, which makes sense because, in binary software vulnerability detection, the cost suffered by classifying vulnerable binary code as non-vulnerable is, in general, much more severe than the converse.

Let $|\mathcal{I}| = M$ where $|\cdot|$ specifies the cardinality of a set and arrange the elements of $\mathcal{I}$ according to their distances to $\mathcal{H}_{-1}$ as $\mathcal{I} = \{i_1, i_2, \ldots, i_M\}$ where $(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_1}) \leq (\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_2}) \leq \cdots \leq (\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_M})$. We now define the cost-sensitive loss for a given decision hyperplane: $(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}) - b_d^m = 0$ in which we denote

$$
\begin{aligned}
b_d^1 &= \frac{b_{-1}^* + (\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_1})}{2}, \\
b_d^m &= \frac{(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_{m-1}}) + (\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_m})}{2}, \; 2 \leq m \leq M, \\
b_d^{M+1} &= \frac{(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_M}) + b_1^*}{2}
\end{aligned}
$$

and the optimal decision hyperplane $(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}) - b_d^* = 0$ as:

$$
\begin{aligned}
l(\mathbf{w}^*, b_d^m) &= \theta \sum_{y_{i_k}=1} \mathbb{I}_{(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_k}) - b_d^m < 0} + \sum_{y_{i_k}=-1} \mathbb{I}_{(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_k}) - b_d^m > 0} \\
m^* &= \underset{1 \leq m \leq M+1}{\operatorname{argmin}} \; l(\mathbf{w}^*, b_d^m) \text{ and } b_d^* = b_d^{m^*}
\end{aligned}
$$

where the indicator function $\mathbb{I}_S$ returns 1 if $S$ is true and 0 if otherwise.

It is worth noting if #non-vul $\approx$ #vul (i.e., $\theta \approx 1$), we obtain a Support Vector Machine [3] and if #non-vul $>>$ #vul (i.e., $\theta \approx 0$), we obtain a One-class Support Vector Machine [21]. We present Algorithm 1 to efficiently find the optimal decision hyperplane. The general idea is to sequentially process the $M+1$ possible hyperplanes: $(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}) - b_d^m = 0, \forall i = 1, \ldots, M+1$ and compute the cost-sensitive loss cumulatively. The computational cost of this algorithm includes: i) the cost to determine $\mathcal{S}$, which is $\mathcal{O}(2DN)$, ii) the cost to sort the elements in $\mathcal{S}$ according to their distances to $\mathcal{H}_{-1}$, which is $\mathcal{O}(M \log M)$, and iii) the cost to process the possible hyperplanes, which is $\mathcal{O}(M+1)$.

# 3 Experiments

## 3.1 Experimental Datasets

Creating labeled binary datasets for binary code vulnerability detection is one of the main contributions of our work. We first collected the source code from two datasets on GitHub: NDSS18[6] and six open-source projects[7] collected in [16] and then processed to create 2 labeled binary datasets.

[6] https://github.com/CGCL-codes/VulDeePecker

[7] https://github.com/DanielLin1986/TransferRepresentationLearning

**Algorithm 1** Pseudo-code for seeking the optimal decision hyperplane.

**Input:** $\mathcal{D} = \{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_N, y_N)\}$, $\mathbf{w}^*$, $b^*_{-1}$, $b^*_1$
**Output:** $m^*$, $b^*_d$

Determine $\mathcal{S}$ and $\mathcal{I}$
Sort the elements in $\mathcal{I}$ as $(\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_1}) \leq (\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_2}) \leq \cdots \leq (\mathbf{w}^*)^\mathsf{T} \tilde{\Phi}(\boldsymbol{h}_{i_M})$
$loss = |\{i \in \mathcal{I} \mid y_i = -1\}|$; //all in $S$ are classified as +
$m^* = 1$; $b^*_d = b^1_d$; $minLoss = loss$; $t = 1$;
**repeat**
  $(y_{i_t} == 1)?loss = loss + \theta : loss = loss - 1$;
  **if** $loss < minLoss$ **then**
    $minLoss = loss$; $m^* = t$;
  **end if**
  $t = t + 1$;
**until** $t > M + 1$

The NDSS18 binary dataset was created in previous work [10] – the functions were extracted from the original source code and then compiled successfully to obtain $8,991$ binaries using an automated tool. However, the source code in the NDSS18 dataset involves the code weaknesses CWE119 and CWE399, resulting in short source code chunks used to demonstrate the vulnerable examples, hence not perfectly reflecting real-world source code, while the source code files collected from the six open-source projects, namely FFmpeg, LibTIFF, LibPNG, VLC, Pidgin and Asterisk are all real-world examples. The statistics of our binary datasets are given in Table 1.

**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** | $52,281$ | $618$ | $52,899$ |

### 3.2 Baselines

We compared our proposed DCKM with various baselines:

- **BRNN-C, BRNN-D:** A vanilla Bidirectional RNN with a linear classifier and two dense layers on the top.
- **Para2Vec:** The paragraph-to-vector distributional similarity model proposed in [8] which allows us to embed paragraphs into a vector space which are further classified using a neural network.
- **VDiscover:** An approach proposed in [4] that utilizes lightweight static features to "approximate" a code structure to seek similarities between program slices.
- **VulDeePecker:** An approach proposed in [15] for source code vulnerability detection.
- **BRNN-SVM:** The Support Vector Machine using linear kernel, but leveraging our proposed feature extraction method.
- **Att-BGRU:** An approach developed by [22] for sequence classification using the attention mechanism.
- **Text CNN:** An approach proposed in [6] using a Convolutional Neural Network (CNN) to classify text.

– **MDSAE:** A method called Maximal Divergence Sequential Auto-Encoder in [10] for binary software vulnerability detection.
– **OC-DeepSVDD:** The One-class Deep Support Vector Data Description method proposed in [20].

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

**Table 2.** The experimental results (%) except for the column CS of the proposed method compared with the baselines on the NDSS18 *binary* dataset. 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** |

**Table 3.** The experimental results (%) except for the column CS of the proposed method compared with the baselines on the *binary* dataset from the six open-source projects. 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 | 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** |

### 3.3 Parameter Setting

For our datasets, we split the data into 80% for training, 10% for validation, and the remaining 10% for testing. For the NDSS18 binary dataset, since it is used for the purpose of demonstrating the presence of vulnerabilities, each vulnerable source code

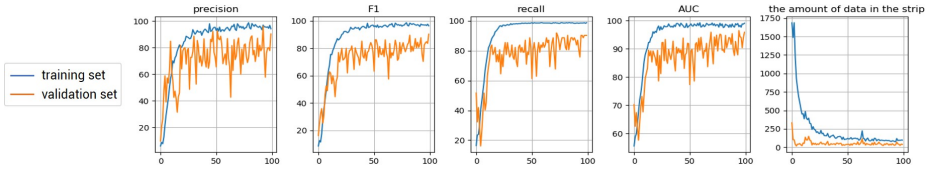


**Fig. 5.** Predictive scores and the number of data examples in the $\mathcal{S}$ strip after 100 epochs.

is associated with its fixed version, hence this dataset is quite balanced. To mimic a real-world scenario, we made this dataset imbalanced by randomly removing vulnerable source code to keep the ratio #vul : #non-vul $= 1 : 50$. For the dataset from six open-source projects, we did not modify the datasets since they are real-world datasets.

We employed a dynamic BRNN to tackle the variation in the number of machine instructions of the functions. For the BRNN baselines and our models, the size of the hidden unit was set to $128$ for *the six open-source projects*'s binary dataset and $256$ for the NDSS18 dataset. For our model, we used Fourier random kernel with the number of random features $2D = 512$ to approximate the RBF kernel, defined as $K(\boldsymbol{x}, \boldsymbol{x}') = \exp\left\{-\gamma \left\|\boldsymbol{x} - \boldsymbol{x}'\right\|^2\right\}$, wherein the width of the kernel $\gamma$ was searched in $\{2^{-15}, 2^{-3}\}$ for the dataset from 6 open-source projects and NDSS18 dataset, respectively. The regularization parameter $\lambda$ was $0.01$. We set the relative cost $\theta =$ #non-vul : #vul. We used the Adam optimizer with an initial learning rate equal to $0.0005$. The minibatch size was set to $64$ and results became promising after $100$ training epochs. We implemented our proposed method in Python using Tensorflow, an open-source software library for Machine Intelligence developed by the Google Brain Team. 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. For each dataset and method, we ran the experiment five times and reported the average predictive performance.

### 3.4 Experimental Results

**Experimental Results on the Binary Datasets** We conducted a variety of experiments on our two binary datasets. We split each dataset into three parts: the subset of Windows binaries, the subset of Linux binaries, and the whole set of binaries to compare our methods with the baselines.

In the field of computer security, besides the AUC and F1 score which takes into account both precision and recall, the cost-sensitive loss, wherein we consider the fact that the misclassification of a vulnerability as a non-vulnerability is more severe than the converse, is also very important. The experimental results on the two datasets are shown in Table 2 and 3. It can be seen that our proposed method outperforms the baselines in all performance measures of interest including the cost-sensitive loss, F1 score, and AUC. Especially, our method significantly surpasses the baselines on the AUC score, one of the most important measures of success for anomaly detection. In addition, although our proposed DCKM aims to directly minimize the cost-sensitive loss, it can balance between precision and recall to maintain very good F1 and AUC scores. In what follows, we further explain this claim.

**Inspection of Model Behaviors**

*Discovering the trend of scores and number of data points in the strip during the training process* Figure 5 shows the predictive scores and the number of data examples

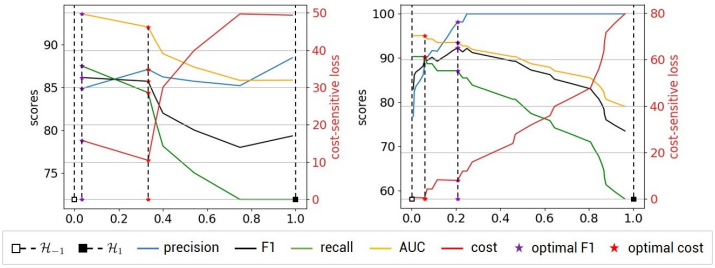


**Fig. 6.** The variation of predictive scores when sliding the hyperplane in the strip formed by $\mathcal{H}_{-1}$ and $\mathcal{H}_1$ on the NDSS18 (left) and the dataset from six open-source projects (right). The red line illustrates the tendency of the cost-sensitive loss, while the purple star and the red star represent the optimal F1 and the optimal cost-sensitive loss values, respectively.

in the parallel strip on training and valid sets for the binary dataset from six open-source projects across the training process. It can be observed that the model gradually improves during the training process with an increase in the predictive scores, and a reduction in the amount of data in the strip from around 1,700 to 50.

*The tendency of predictive scores when sliding the decision hyperplane in the strip formed by $\mathcal{H}_{-1}$ and $\mathcal{H}_1$* By minimizing the cost-sensitive loss, we aim to find the optimal hyperplane which balances precision and recall, while at the same time maintaining good F1 and AUC scores. Figure 6 shows the tendency of scores and cost-sensitive loss when sliding the decision hyperplane in the strip formed by $\mathcal{H}_{-1}$ and $\mathcal{H}_1$. We especially focus on four milestone hyperplanes, namely $\mathcal{H}_{-1}$, $\mathcal{H}_1$, the hyperplane that leads to the optimal F1 score, and the hyperplane that leads to the optimal cost-sensitive loss (i.e., our optimal decision hyperplane). As shown in Figure 6, 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.

## 4 Conclusion

Binary software vulnerability detection has emerged as an important and crucial problem in the software industry, such as the embedded systems industry, and in the field of computer security. In this paper, 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 conducted experiments on two binary datasets. The experimental results have shown a convincing outperformance of our proposed method compared to the state-of-the-art baselines.

## References

1. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Communications of the ACM **56**(2), 82–90 (2013)
2. Cao, P., Zhao, D., Zaïane, O.R.: An optimized cost-sensitive SVM for imbalanced data learning. In: PAKDD (2). Lecture Notes in Computer Science, vol. 7819, pp. 280–292. Springer (2013)

3. Cortes, C., Vapnik, V.: Support-vector networks. Machine Learning **20**(3), 273–297 (Sep 1995)
4. Grieco, G., Grinblat, G.L., Uzal, L., Rawat, S., Feist, J., Mounier, L.: Toward large-scale vulnerability discovery using machine learning. In: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy. pp. 85–96. CODASPY '16 (2016)
5. Katsumata, S., Takeda, A.: Robust cost sensitive support vector machine. In: AISTATS. JMLR Workshop and Conference Proceedings, vol. 38. JMLR.org (2015)
6. Kim, Y.: Convolutional neural networks for sentence classification. In: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL. pp. 1746–1751 (2014)
7. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. arXiv preprint arXiv:1312.6114 (2013)
8. Le, Q.V., Mikolov, T.: Distributed representations of sentences and documents. In: International on Machine Learning 2014. JMLR Workshop and Conference Proceedings, vol. 32, pp. 1188–1196. JMLR.org (2014)
9. Le, T., Nguyen, K., Nguyen, V., Nguyen, T.D., Phung, D.: Gogp: Fast online regression with gaussian processes. In: In International Conference on Data Mining (2017)
10. Le, T., Nguyen, T., Le, T., Phung, D., Montague, P., De Vel, O., Qu, L.: Maximal divergence sequential autoencoder for binary software vulnerability detection. In: International Conference on Learning Representations (2019), https://openreview.net/forum?id=ByloIiCqYQ
11. Le, T., Nguyen, T.D., Nguyen, V., Phung, D.: Dual space gradient descent for online learning. In: In Advances in Neural Information Processing (2016)
12. Le, T., Tran, D., Ma, W., Pham, T., Duong, P., Nguyen, M.: Robust support vector machine. In: International Joint Conference on Neural Networks (2014)
13. Le, T., Tran, D., Ma, W., Sharma, D.: A unified model for support vector machine and support vector data description. In: IJCNN. pp. 1–8 (2012)
14. Le, T., Nguyen, K., Nguyen, V., Nguyen, T., Phung, D.: Gogp: Scalable geometric-based gaussian process for online regression. Knowledge and Information Systems(KAIS) journal (2018)
15. Li, Z., Zou, D., Xu, S., Ou, X., Jin, H., Wang, S., Deng, Z., Zhong, Y.: VulDeePecker: A deep learning-based system for vulnerability detection. CoRR **abs/1801.01681** (2018)
16. Lin, G., Zhang, J., Luo, W., Pan, L., Xiang, Y., De Vel, O., Montague, P.: Cross-project transfer representation learning for vulnerable function discovery. In: IEEE Transactions on Industrial Informatics (2018)
17. Nguyen, T.D., Le, T., Bui, H., Phung, D.: Large-scale online kernel learning with random feature reparameterization. In: In Proceedings of the 26th International Joint Conference on Artificial Intelligence (2017)
18. Nguyen, V., Le, T., Pham, T., Dinh, M., Le, T.H.: Kernel-based semi-supervised learning for novelty detection. In: 2014 International Joint Conference on Neural Networks (IJCNN). pp. 4129–4136 (July 2014)
19. Rahimi, A., Recht, B.: Random features for large-scale kernel machines. In: Advances in neural information processing systems. pp. 1177–1184 (2008)
20. Ruff, L., Vandermeulen, R., Goernitz, N., Deecke, L., Siddiqui, S.A., Binder, A., Müller, E., Kloft, M.: Deep one-class classification. In: Dy, J., Krause, A. (eds.) Proceedings of the 35th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 80, pp. 4393–4402. Stockholmsmï¿œssan, Stockholm Sweden (10–15 Jul 2018)
21. Schölkopf, B., Platt, J.C., Shawe-Taylor, J., Smola, A.J., Williamson, R.C.: Estimating the support of a high-dimensional distribution. Neural computation **13**(7), 1443–1471 (2001)
22. Zhou, P., Shi, W., Tian, J., Qi, Z., Li, B., Hao, H., Xu, B.: Attention-based bidirectional long short-term memory networks for relation classification. In: ACL (2016)