

**iTrustSO: An Intelligent System for Automatic Detection of Insecure Code Snippets in Stack Overflow**

Lingwei Chen, Shifu Hou, Yanfang Ye, Thirimachos Bourlai, Shouhuai Xu, Liang Zhao  
\[1\]Department of CDS, Case Western Reserve University, OH, USA  
\[2\]Department of CS, UTSA, TX, USA, \[3\]Department of IST, GMU, VA, USA  
\[1\]yanfang.ye@case.edu, \[2\]shxu@cs.utsa.edu, \[3\]lzhao9@gmu.edu

**Abstract**—Despite the apparent benefits of modern social coding paradigm such as Stack Overflow, its potential security risks have been largely overlooked (e.g., insecure codes could be easily embedded and distributed). To address this imminent issue, in this paper, we bring a significant insight to leverage both social coding properties and code content for automatic detection of insecure code snippets in Stack Overflow. To determine if the given code snippets are insecure, we not only analyze the code content, but also utilize various kinds of relations among users, badges, questions, answers and code snippets in Stack Overflow. To model the rich semantic relationships, we first introduce a structured heterogeneous information network (HIN) for representation and then use meta-path based approach to incorporate higher-level semantics to build up relatedness over code snippets. Later, we propose a novel hierarchical attention-based sequence learning model named CodeHin2Vec to seamlessly integrate node (i.e., code snippet) content with HIN-based relations for representation learning. After that, a classifier is built for insecure code snippet detection. Integrating our proposed method, an intelligent system named iTrustSO is accordingly developed to address the code security issues in modern software coding platforms. Comprehensive experiments on the data collections from Stack Overflow are conducted to validate the effectiveness of our developed system iTrustSO by comparisons with alternative methods.

I. INTRODUCTION

Software plays a vital role in modern society. Almost every corner of our daily lives, such as entertainment, education, and social communication, depends on different reliable software. Unlike conventional approaches, modern software developers heavily engage in a social coding environment to reuse code snippets and projects during the process of software development \[33\]. In particular, Stack Overflow, as the largest online programming discussion platform, has attracted 9.9 million registered developers \[27\]. The active discussions and abundant code snippets make it one of the most important information sources to software developers \[9\]. Despite the apparent benefits of such social coding environment, its potential security risks have been largely overlooked \[11\], \[13\]. According to a recent study \[1\], collected question-answer samples from Stack Overflow contain various security-related issues such as encryption with insecure mode, and insecure Application Programming Interface (API) usage. Those innocent-looking yet insecure code snippets could cause severe damage or even a disaster if not properly handled and directly transplanted to production software. For example, as shown in Figure 1, attackers have injected malicious cryptocurrency mining code such as Coinhive into Stack Overflow; once developers reuse such code snippets to generate the production software, its users’ devices could be compromised (e.g., processing power would be stolen to mine bits of cryptocurrency). To deal with insecure code snippets included in the questions/answers, Stack Overflow has no principled way other than labeling the moderator flag, downvoting those threads or warning in the comments \[3\]. Given the rich structure and information, there is imminent need to develop novel and sound solutions to address the code security issue in Stack Overflow.

To this end, a significant insight brought by this work is to leverage both social coding properties and code content for insecure code snippet detection. As a social coding environment, Stack Overflow is characterized by user communication through questions and answers \[1\] i.e., a rich source of heterogeneous information are available including users, badges, questions, answers, code snippets, and their semantic relationships. To utilize such social coding properties, our previous work \[39\] proposed ICSD over heterogeneous information network (HIN) \[28\], \[30\] for insecure code snippet detection. ICSD encodes code content and social relations to construct the HIN, and then learns node representations from

**Figure 1:** Example of code security attacks in Stack Overflow.
HIN to detect insecure code snippets. In this paper, we’d like to take a different tac to see if we can enhance the representation learning of code snippets in the detection of insecure ones. Different from ICSD, in this work, we propose to utilize HIN to depict relatedness over code snippets to generate code-to-code sequences, based on which sequence to sequence (seq2seq) concept in machine translation is further leveraged to learn representations of code snippets. More specifically, we introduce HIN as an abstract representation, and then use meta-path \( \mathbf{[30]} \) to incorporate higher-level semantic relations to build up relatedness over the code snippets. Afterwards, to take both code content and the social coding properties into account, we propose a novel seq2seq learning model named CodeHin2Vec for representation learning of code snippets. Different from the traditional seq2seq model that uses one long short-term memory (LSTM) to read the input sequence to obtain a fixed-length summary vector from which another LSTM is employed to generate the output sequence \( \mathbf{[31]}, \mathbf{[19]} \), CodeHin2Vec extends this basic encoder-decoder architecture \( \mathbf{[8]} \) by elaborately devising hierarchical attention mechanism to first learn the context between node embeddings in the input sequence, and then learn the alignments and relevances between hidden layer vectors for the output sequence generation. This allows a refined architecture to cope better with sequence modeling and thus fully exploit code content and HIN structure to learn better representations of code snippets. After that, a classifier is built for insecure code snippet detection. We develop a system called \( \mathbf{iTrustSO} \) shown in Figure \( \mathbf{2} \) to integrate our proposed method, which has the following merits:

- It introduces HIN as an abstract representation of Stack Overflow data, and exploits a meta-path based approach to characterize the relatedness over code snippets. The proposed solution provides a natural way of expressing complex relationships in social coding platforms.
- It integrates HIN with seq2seq concept for representation learning. In \( \mathbf{iTrustSO} \), a new model CodeHin2Vec is proposed to seamlessly combine code content and HIN-based relations to learn representations of code snippets, in which code sequences are first generated based on the walk paths guided by different meta-paths; in each code sequence, its elements are represented by the code content feature vectors; then, LSTM using hierarchical attention mechanism is leveraged for code sequence modeling. CodeHin2Vec is a generic framework which can also be applicable for other representation learning task.
- Comprehensive experimental studies demonstrate the performance of our developed system \( \mathbf{iTrustSO} \), which is practical for automatic detection of insecure code snippets.

II. PROPOSED METHOD

In this section, we present the detailed approaches of how we represent and detect the code snippets in Stack Overflow.

A. Feature Extraction

**Code snippets.** Code snippets in Stack Overflow can be first separated from accompanying texts in question and answer threads through pairs of (code) \( \langle / \rangle \) tags; afterwards, each code snippet can be represented by a feature vector \( \mathbf{i.e.}, \mathbf{x}_i \) to denote its code content using word2vec \( \mathbf{[23]}, \mathbf{[22]} \).

**Social coding properties.** To characterize a code snippet in Stack Overflow, we not only consider its code content, but also extract its social coding properties including:

1. \( \mathbf{R1} \): question-have-code relation describes whether a question thread has a code snippet embedded;
2. \( \mathbf{R2} \): answer-include-code relation denotes that an answer thread includes a code snippet;
3. \( \mathbf{R3} \): user-post-question describes the relationship between a user and a question he/she posts;
4. \( \mathbf{R4} \): user-supply-answer relation represents if a user supplies an answer;
5. \( \mathbf{R5} \): answer-echo-question relation denotes if an answer echoes a question;
6. \( \mathbf{R6} \): user-gain-badge relation means a user gains a badge, denoting his/her level (i.e., gold, silver, or bronze) over different contributions (e.g., question, answer, etc.).

B. HIN Construction

This section introduces how to use the extracted entities and social coding properties to represent code snippets in Stack Overflow. We first present the concepts related to HIN:

**Definition 2.1: Heterogeneous information network (HIN)** \( \mathbf{[39]} \). A HIN is defined as a graph \( \mathbf{G} = (\mathbf{V}, \mathbf{E}) \) with an entity type mapping \( \mathbf{\phi} : \mathbf{V} \rightarrow \mathbf{A} \) and a relation type mapping \( \mathbf{\psi} : \mathbf{E} \rightarrow \mathbf{R} \), where \( \mathbf{V} \) denotes the entity set and \( \mathbf{E} \) is the relation set, \( \mathbf{A} \) denotes the entity type set and \( \mathbf{R} \) is the relation type set, and the number of entity types \( |\mathbf{A}| > 1 \) or the number of relation types \( |\mathbf{R}| > 1 \). The network schema \( \mathbf{T}_G = (\mathbf{A}, \mathbf{R}) \) for a HIN \( \mathbf{G} \) is a graph with nodes as entity types from \( \mathbf{A} \) and edges as relation types from \( \mathbf{R} \).
For our case, we have five entity types and six types of relations among them; accordingly, the network schema for HIN in our application is shown in Figure 3.

Figure 3: Network schema for HIN in our application.

The different types of entities and relations motivate us to use a machine-readable representation to enrich the semantics of relatedness among code snippets. To handle this, the concept of meta-path [30] to formulate the higher-order relationships among entities in HIN is extended to our application of insecure code snippet detection.

Definition 2.2: Meta-path [30]. A meta-path \( \mathcal{P} \) is a path defined on the graph of network schema \( \mathcal{T}_0 = (\mathcal{A}, \mathcal{R}) \), and is denoted in the form of \( A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \ldots \xrightarrow{R_l} A_{L+1} \), which defines a composite relation \( R = R_1 \circ R_2 \circ \ldots \circ R_l \), between types \( A_1 \) and \( A_{L+1} \), where \( \cdot \) denotes relation composition operator, and \( L \) is the length of \( \mathcal{P} \).

Table I: Meta-paths built for insecure code snippet detection

<table>
<thead>
<tr>
<th>ID</th>
<th>Meta-paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID1</td>
<td>( c \xrightarrow{I} a \xrightarrow{S} u \xrightarrow{S} a \xrightarrow{I} c )</td>
</tr>
<tr>
<td>PID2</td>
<td>( c \xrightarrow{H} q \xrightarrow{P} u \xrightarrow{P} q \xrightarrow{H} c )</td>
</tr>
<tr>
<td>PID3</td>
<td>( c \xrightarrow{H} q \xrightarrow{P} u \xrightarrow{P} q \xrightarrow{E} a \xrightarrow{I} c )</td>
</tr>
<tr>
<td>PID4</td>
<td>( c \xrightarrow{H} q \xrightarrow{P} u \xrightarrow{P} q \xrightarrow{E} a \xrightarrow{I} c )</td>
</tr>
<tr>
<td>PID5</td>
<td>( c \xrightarrow{H} q \xrightarrow{P} u \xrightarrow{P} q \xrightarrow{H} c )</td>
</tr>
<tr>
<td>PID6</td>
<td>( c \xrightarrow{H} q \xrightarrow{P} u \xrightarrow{P} q \xrightarrow{H} c )</td>
</tr>
<tr>
<td>PID7</td>
<td>( c \xrightarrow{H} q \xrightarrow{P} u \xrightarrow{P} q \xrightarrow{H} c )</td>
</tr>
<tr>
<td>PID8</td>
<td>( c \xrightarrow{H} q \xrightarrow{P} u \xrightarrow{P} q \xrightarrow{H} c )</td>
</tr>
</tbody>
</table>

Given a network schema with different types of entities and relations, we can enumerate a lot of meta-paths. In our application, based on the collected data, resting on the six different kinds of relationships, we design eight meaningful meta-paths for characterizing relatedness over code snippets, i.e., PID1-PID8 shown in Table I (symbols are the abbreviations shown in Figure 3). Different meta-paths depict the relatedness between two code snippets at different views. For example, a typical one to formulate the relatedness over code snippets in Stack Overflow is PID1: \( c \xrightarrow{I} a \xrightarrow{S} u \xrightarrow{S} a \xrightarrow{I} c \) which means that two code snippets can be connected as they are included in the answers supplied by the same user.

C. CodeHin2Vec

To devise a comprehensive solution to combine both node (i.e., code snippet) content and HIN-based relations for insecure code snippet detection, we observe from our previous work [39] that the HIN-based neighborhood relationships among code snippets can be represented by the code sequences (denoted as CodeSeq) based on different meta-paths. In this way, the generated CodeSeqs can preserve both semantic and structure information of HIN. To further couple CodeSeqs with code content, a straightforward yet novel way is to use the content feature vector \( \mathbf{x}_c \) to represent each code snippet in the CodeSeq. To this end, the representation learning of code snippets can be viewed as a sequence modeling task. As LSTM has shown significant improvement in language modeling [4], we leverage its power to seamlessly integrate code content and HIN structure into hidden layer vectors that can be used as the representations of code snippets [19].

Although it is promising to comprehensively utilize LSTM to learn the mapping from the code content sequence to code identity sequence, it still faces the following two challenges: (1) word2vec assigns each code snippet a static embedding vector based on code content which is not context-aware to different sequences it interacts with. For example, as illustrated in Figure 4, guided by the designed meta-paths, we may generate CodeSeq-A and CodeSeq-B. With function fileProcess defined, Code-1 in CodeSeq-A performs as file encryption for Ransomware while Code-3 in CodeSeq-B implements the regular file reading and writing; in this respect, even though Code-2 listed in both sequences calls the same function fileProcess, its embedding vector should be significantly different which may demonstrate insecure potential when interacting with Code-1 and normal aspect when related to Code-3. LSTM is known to learn the sequential dependencies [25], but strict alignment of the positions of the input sequence; therefore, contextualized code content embeddings may help to refine the hidden-layer information in the early stage. (2) Since LSTM needs to read the whole input sequence to further generate the output sequence, its performance using a basic encoder-decoder architecture may degrade as the length of an input sequence increases [17], [4] which may in turn degenerate the representations learned from hidden layers, especially in our case that code sequences are much longer than the sentences.

Attention mechanism has shown remarkable effectiveness in various sequence modeling tasks, allowing models to learn alignments between different modalities [34], [4], [21], [17]. In this work, to address the challenges above, we propose CodeHin2Vec to elaborate a hierarchical attention mechanism into LSTM to fully exploit code content and HIN structure to learn effective representations of code snippets, which first generates CodeSeqs based on the walk paths guided by different meta-paths; and then leverages LSTM with hierarchical attention mechanism for CodeSeq modeling.

Figure 4: Different contexts among code snippets.
**CodeSeq generation guided by different meta-paths.** Given a source node $v_j$ in a homogeneous network, the traditional random walk is a stochastic process with random variables $v_j^1, v_j^2, \ldots, v_j^i$ such that $v_j^{i+1}$ is a node chosen at random from the neighbors of node $v_j$. The transition probability $p(v_j^{i+1} | v_j^i)$ at step $i$ is the normalized probability distributed over the neighbors of $v_j^i$ by ignoring their node types. However, this mechanism is unable to capture the semantic and structural correlations among different types of nodes in a HIN. In our application, given a HIN $G = (V,E)$ with schema $T_G = (A,R)$, and a set of different meta-paths $\mathcal{P} = \{P_j\}_{j=1}^n$, each of which is in the form of $A_1 \rightarrow \ldots \rightarrow A_l \rightarrow A_{l+1} \rightarrow \ldots \rightarrow A_1$, we put a random walker to traverse the HIN. The random walker first randomly chooses a meta-path $P_k$ from $\mathcal{P}$ and the transition probabilities at step $i$ are defined as follows:

$$p(v_j^{i+1} | v_j^i, P_k) = \begin{cases} \frac{1}{N_{A_{i+1}}(v_j^i)} & \text{if } (v_j^{i+1}, v_j^i) \in E, \phi(v_j^i) = A_c, \phi(v_j^{i+1}) = A_{i+1} \\ 0 & \text{otherwise} \end{cases}$$

(1)

where $\phi$ is the node type mapping function, $N_{A_{i+1}}(v_j^i)$ denotes the $A_{i+1}$-type neighborhood of node $v_j^i$, $A_c$ is entity type of Code, and $\lambda$ is the number of meta-paths starting with $A_c \rightarrow A_{i+1}$. For each walk path, the nodes whose entity types are not Code will be removed; then the remaining ones form a CodeSeq, whose representation is consisted of the content feature vector $x_c$. In such way, given walk path length $l$, a CodeSeq is presented as $(x_{c_1}, x_{c_2}, \ldots, x_{c_l})$.

**CodeSeq modeling with LSTM.** LSTM learns a mapping from an input sequence to an output sequence. As intermediate states, a hidden vector is generated for each timestep; we use an encoder-decoder LSTM architecture [3] for CodeSeq modeling in which two attention layers are elaborately added to improve the quality of representation learning (as illustrated in Figure 5).

**Encoder attention:** Resting on all the content vectors in the input sequence, the encoder attention layer computes the contextualized embedding for each code snippet as a weighted sum where the weight, also called context score, assigned to each content vector is computed by a dot product of the corresponding pair of content vectors [34]. Specifically, given an input CodeSeq $(x_{c_1}, x_{c_2}, \ldots, x_{c_l})$, for any two code snippets $c_i$ and $c_l$, the context score can be calculated as

$$S(x_{c_i}, x_{c_l}) = x_{c_i}^T x_{c_l},$$

(2)

where $\top$ denotes the dot product, and thus the contextualized embedding for code snippet $c_i$ can be computed as

$$\hat{x}_{c_i} = \sum_{i=1}^{l} \frac{\exp(S(x_{c_i}, x_{c_l}))}{\sum_{l=1}^{l} \exp(S(x_{c_i}, x_{c_l}))} x_{c_l}.$$  

(3)

![Figure 5: Architecture of LSTM using hierarchical attention.](image)

In this sense, a CodeSeq can be refined as $(\hat{x}_{c_1}, \hat{x}_{c_2}, \ldots, \hat{x}_{c_l})$, which will be used as the actual input sequence.

**Encoder:** The encoder reads $(\hat{x}_{c_1}, \hat{x}_{c_2}, ..., \hat{x}_{c_l})$ through the hidden layer function $H$ so that each hidden layer vector $h_t^e$ at timestep $t$ can be denoted as

$$h_t^e = H(\hat{x}_{c_1}, h_{t-1}^e),$$

(4)

where $H$ is implemented using memory cells to store information, which can be formulated as the following composite functions [14]:

$$i_t = \sigma(W_{x_i} \hat{x}_{c_t} + W_{h_i} h_{t-1}^e + W_{c_i} c_{t-1} + b_i)$$

(5)

$$f_t = \sigma(W_{x_f} \hat{x}_{c_t} + W_{h_f} h_{t-1}^e + W_{c_f} c_{t-1} + b_f)$$

(6)

$$c_t = f_t \circ c_{t-1} + i_t \circ \tanh(W_{x_c} \hat{x}_{c_t} + W_{h_c} h_{t-1}^e + b_c)$$

(7)

$$o_t = \sigma(W_{x_o} \hat{x}_{c_t} + W_{h_o} h_{t-1}^e + W_{c_o} c_{t-1} + b_o)$$

(8)

$$h_t^e = o_t \circ \tanh(c_t)$$

(9)

where $\sigma$ is the logistic sigmoid function, $i_t, f_t, o_t, c_t$ are the input gate, forget gate, output gate, and cell activation vectors respectively, $W$s are the weight matrices, $b$s are the bias vectors, and $\circ$ is the point-wise product between two vectors. Since the input sequence has no direction, in order to learn both the forward and backward sequential dependency information, we utilize bidirectional encoder so that hidden layer vector $h_t^e$ at timestep $t$ can be concatenated as $h_t^e = [h_t^e_1, h_t^e_2]$. After forward and backward reading CodeSeq, the concatenation of the last two hidden states $[h_{t-1}^e_1, h_{t-1}^e_2]$ is used as the summary vector $s$ of the whole input sequence.

**Decoder attention:** The decoder attention layer exploits all the hidden states of the encoder to compute the aligned and joint information as the context vector [41, 42], which is integrated with the summary vector $s$ to extract the target code identity. Similar to the encoder attention, the alignment scores need to be first defined to formulate such context vector as a weighted sum. Note that, unlike the dot product attention, decoder attention should allow the gradient of the cost function to be backpropagated through [41]. We accordingly use a simple feed-forward neural network to compute the alignment score

$$\alpha_t = W_{\alpha}^2 \text{ReLU}(W_{\alpha}^1 h_t^e + b_{\alpha}^1) + b_{\alpha}^2,$$

(10)

where $W_{\alpha}s$ and $b_{\alpha}s$ denote the weight matrices and the bias
vectors, and the alignment score vector $\alpha_t$, trained by all the other hidden states of the encoder reflects the importance of $h_t^c$ in generating $y_t$. The context vector for $h_t^c$ can thus be

$$h_t^c = \sum_{i=1}^{l} \frac{\exp(\alpha_t^{c,i})}{\sum_{j=1}^{l} \exp(\alpha_t^{j,i})} h_t^i.$$  \hspace{1cm} (11)

**Decoder:** The decoder takes the summary vector $s$ as input (i.e., $h_0^d = s$) and generates a sequence of target hidden states; each hidden state $h_t^d$ at timestep $t$ can be calculated as

$$h_t^d = h((0, h_{t-1}^c),$$  \hspace{1cm} (12)

where $0$ is an all-zero vector. Given the target hidden state $h_t^d$ and the context vector $h_t^c$, we concatenate them to form an attentional hidden state $h_t^d = [h_t^c; h_t^d]$. Accordingly, the output vector $y_t \in \mathbb{R}^{|V|}$ can be generated as follows

$$y_t = \sigma(W_{hy}h_t^d + b_y).$$  \hspace{1cm} (13)

$y_t$ is capable to predict the real code snippet $c_t$ through a softmax layer. The sequence loss $L$ is adopted to measure the correctness of decoding, which is computed as

$$L = - \sum_{i=1}^{l} \log p(c_i|y_i) = - \sum_{i=1}^{l} \log \frac{\exp(y_i^{c_i})}{\sum_{c \in V} \exp(y_i^{c})}. $$  \hspace{1cm} (14)

The weights can be efficiently calculated with backpropagation through time [36], [14], and the LSTM model can then be trained using Adam optimization algorithm.

For the generated CodeSeqs guided by different meta-paths, each code snippet may appear in multiple CodeSeqs. Suppose that code snippet $c_t$ exists in $|c_t|$ CodeSeqs, by doing avg pooling over all $h_t^c$’s for code snippet $c_t$, $\forall t = 1, ..., |c_t|$, we obtain an embedding $h$ for each code snippet

$$h = \text{avgPooling}([h_t^c : i = 1, ..., |c_t|]).$$  \hspace{1cm} (15)

Using CodeHin2Vec, the mapped feature vectors of code snippets, encoding the information of code content and HIN-based relations, can be fed to a classifier to train the classification model, based on which the unlabeled code snippets can be predicted if they are insecure or not.

### III. Experimental Results and Analysis

In this section, we fully evaluate the performance of iTrustSO in insecure code snippet detection. We consider Java programming language for Android app as a case study. Based on our prior work ICSD [39], in this paper, we further expand our data collection and annotation from Stack Overflow: (1) using our developed crawlers, we collect 505,548 question threads and 719,430 answer threads posted by 229,394 users including 821,792 code snippets, through March 2010 to October 2018; (2) we also expand our annotated data in [39] to finally obtain 21,989 labeled code snippets (10,013 are insecure while 11,976 are secure) as the ground truth to evaluate different detection methods. To quantitatively validate the effectiveness of different methods, we use accuracy (ACC) and F1 measure (F1) as the performance measures.

#### A. Evaluation of Different Meta-paths

In this set of experiments, given a specific meta-path scheme, we use a basic LSTM to learn the latent representations of code snippets in HIN, which is then fed to SVM for detection. Here we perform 10-fold cross validations for evaluation. The experimental results are shown in Table II from which we can see that different meta-paths indeed show different performances: (1) PID1, PID3, PID5, and PID7 perform better than PID2, PID4, PID6, and PID8; the reason behind this is that the code snippets posted in the answer threads are more likely to be reused by the developers than the ones posted in question threads, and thus they have closer connections. (2) PID3 outperforms the others, which indicates that its semantics reflecting the insecure code snippet detection problem is better than the others. (3) PID9 using different meta-paths is more expressive than individuals in depicting the code snippets and thus achieve better performance.

<table>
<thead>
<tr>
<th>ID Meta-paths included</th>
<th>Recall</th>
<th>Precision</th>
<th>ACC</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID1 –</td>
<td>0.8481</td>
<td>0.7956</td>
<td>0.8316</td>
<td>0.8210</td>
</tr>
<tr>
<td>PID2 –</td>
<td>0.8098</td>
<td>0.7491</td>
<td>0.7899</td>
<td>0.7783</td>
</tr>
<tr>
<td>PID3 –</td>
<td>0.8596</td>
<td>0.8119</td>
<td>0.8454</td>
<td>0.8351</td>
</tr>
<tr>
<td>PID4 –</td>
<td>0.8344</td>
<td>0.7769</td>
<td>0.8155</td>
<td>0.8046</td>
</tr>
<tr>
<td>PID5 –</td>
<td>0.8605</td>
<td>0.8086</td>
<td>0.8437</td>
<td>0.8337</td>
</tr>
<tr>
<td>PID6 –</td>
<td>0.8140</td>
<td>0.7588</td>
<td>0.7975</td>
<td>0.7854</td>
</tr>
<tr>
<td>PID7 –</td>
<td>0.8042</td>
<td>0.7444</td>
<td>0.7851</td>
<td>0.7731</td>
</tr>
<tr>
<td>PID8 –</td>
<td>0.7843</td>
<td>0.7203</td>
<td>0.7631</td>
<td>0.7509</td>
</tr>
<tr>
<td>PID9 – (PID1,..., PID8)</td>
<td>0.8785</td>
<td>0.8415</td>
<td>0.8693</td>
<td>0.8596</td>
</tr>
</tbody>
</table>

#### B. Evaluation of Attention

In this set of experiments, we’d like to assess whether the hierarchical attention mechanism devised in our model is meaningful for representation learning. To this end, we explore the performances of basic LSTM without attention (LSTM-b), LSTM with encoder attention (LSTM-e), LSTM with decoder attention (LSTM-d), and CodeHin2Vec. The better detection result implies that the learn representations take better advantage of the corresponding sequence learning architecture. From the results illustrated in Figure 6 we have the following observations: (1) LSTM-e and LSTM-d with single attention layer both outperform LSTM-b without attention; (2) CodeHin2Vec achieves the most promising performance for fully utilizing the contextualized input embeddings and the aligned information from the hidden states of the encoder. In other words, CodeHin2Vec has potential to let LSTM learn better sequential dependencies and code better with the sequence.
Table III: Comparisons of CodeHin2Vec with other network representation learning methods in insecure code snippet detection

<table>
<thead>
<tr>
<th>Metric</th>
<th>Method</th>
<th>Feature</th>
<th>10%</th>
<th>20%</th>
<th>30%</th>
<th>40%</th>
<th>50%</th>
<th>60%</th>
<th>70%</th>
<th>80%</th>
<th>90%</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>word2vec</td>
<td>Content</td>
<td>0.6554</td>
<td>0.6757</td>
<td>0.6989</td>
<td>0.7105</td>
<td>0.7379</td>
<td>0.7590</td>
<td>0.7725</td>
<td>0.7730</td>
<td>0.7753</td>
</tr>
<tr>
<td></td>
<td>DeepWalk</td>
<td>Relation</td>
<td>0.6263</td>
<td>0.6632</td>
<td>0.6678</td>
<td>0.6809</td>
<td>0.7087</td>
<td>0.7132</td>
<td>0.7349</td>
<td>0.7403</td>
<td>0.7430</td>
</tr>
<tr>
<td></td>
<td>TADW</td>
<td>Relation</td>
<td>0.7241</td>
<td>0.7415</td>
<td>0.7562</td>
<td>0.7747</td>
<td>0.7898</td>
<td>0.8065</td>
<td>0.8035</td>
<td>0.8204</td>
<td>0.8312</td>
</tr>
<tr>
<td></td>
<td>ICSD</td>
<td>Content&amp;Relation</td>
<td>0.7659</td>
<td>0.7761</td>
<td>0.7902</td>
<td>0.8029</td>
<td>0.8144</td>
<td>0.8312</td>
<td>0.8394</td>
<td>0.8478</td>
<td>0.8537</td>
</tr>
<tr>
<td></td>
<td>CodeHin2Vec</td>
<td>Content&amp;Relation</td>
<td>0.8026</td>
<td>0.8234</td>
<td>0.8487</td>
<td>0.8588</td>
<td>0.8783</td>
<td>0.8884</td>
<td>0.8968</td>
<td>0.9035</td>
<td>0.9107</td>
</tr>
</tbody>
</table>

ACC

|        | word2vec       | Content                        | 0.6292 | 0.6507 | 0.6756 | 0.6875 | 0.7166 | 0.7379 | 0.7519 | 0.7530 | 0.7560 |
|        | DeepWalk       | Relation                       | 0.6023 | 0.6415 | 0.6439 | 0.6551 | 0.6871 | 0.6911 | 0.7139 | 0.7203 | 0.7233 |
|        | TADW           | Relation                       | 0.7005 | 0.7199 | 0.7356 | 0.7552 | 0.7711 | 0.7884 | 0.7853 | 0.8038 | 0.8147 |
|        | ICSD           | Content&Relation               | 0.7446 | 0.7565 | 0.7717 | 0.7855 | 0.7977 | 0.8147 | 0.8239 | 0.8330 | 0.8390 |
|        | CodeHin2Vec    | Content&Relation               | 0.7855 | 0.8083 | 0.8338 | 0.8454 | 0.8662 | 0.8769 | 0.8866 | 0.8933 | 0.9015 |

F1

C. Evaluation of CodeHin2Vec

Here, CodeHin2Vec is evaluated by comparisons with several representation learning methods: (1) word2vec [22] is a baseline using code content information; (2) DeepWalk [24] is a homogeneous network embedding method leveraging relation information; (3) metapath2vec [12] is a HIN embedding model utilizing HIN-based relations; (4) TADW [37] considers both content and relation information for homogeneous network representation learning; (5) ICSD [39] takes content and relation into account in HIN. For DeepWalk and TADW, we ignore the heterogeneous property of HIN and directly feed the HIN for embedding; in metapath2vec, a walk path is generated based on a single meta-path scheme; in ICSD, code content is extracted as keywords to be devised to HIN. The parameter settings used for CodeHin2Vec are in line with typical values used for the baselines: content dimension $c = 300$, vector dimension $d = 200$, walks per node $r = 10$, walk length $l = 80$ (TADW: walk steps are set to 2), and window size $w = 10$. To facilitate the comparisons, we randomly select a portion of labeled code snippets (ranging from 10% to 90%) for training and the remaining ones for testing. SVM is used as the classification model for all the methods. Table III illustrates the detection results: CodeHin2Vec outperforms all baselines in terms of ACC and F1 in most cases. That is to say, CodeHin2Vec learns significantly better code snippet representation than current state-of-the-art methods. The success of CodeHin2Vec lies in the seamless integration of code content with HIN-based relations for representation learning, which leverages the advantage of (1) CodeSeq generation based on the different meta-paths and (2) the CodeSeq modeling power of LSTM using hierarchical attentions.

D. Evaluation of Parameters

In this set of experiments, we first conduct the sensitivity analysis of how different choices of parameters will affect the performance of CodeHin2Vec. From the results shown in Figure 7(a) and 7(b), we can observe that the balance between computational cost (number of walks per node $r$ and walk length $l$ in $x$-axis) and efficacy (F1 in $y$-axis) can be achieved when $r ≥ 10$ and $l ≥ 80$. As shown in Figure 7(c), we can see that the performance tends to be stable once content vector dimension $c$ reaches around 200 to 300; similarly, from Figure 7(d) we can find that the performance inclines to be stable when vector dimensions $d$ increases to around 200 to 400. Overall, CodeHin2Vec is not strictly sensitive to these parameters, and is able to reach high performance under a cost-effective parameter choice. We then further evaluate the scalability of CodeHin2Vec which can be parallelized for optimization. We run the experiments using the default parameters with different number of threads (i.e., 1, 4, 8, 12, 16), each of which utilizes one CPU core. Figure 7(e) shows the speed-up of CodeHin2Vec deploying multiple threads over the single-threaded case, which reveals that the model achieves acceptable sub-linear speed-ups as the line is close to the optimal line; while Figure 7(f) shows that the performance remains stable when using multiple threads for model updating. Overall, the proposed system are efficient and scalable for large-scale HIN with large numbers of nodes. For stability evaluation, Figure 7(g) shows the ROC curves of CodeHin2Vec based on the 10-fold cross validations; it achieves an average 0.9043 TPR at the 0.1221 FPR for detection.
E. Comparisons with Traditional Machine Learning Methods

In this set of experiments, iTrustSO is compared with other traditional machine learning methods. For these methods, we construct three types of features: \( f-1 \): content-based features (i.e., \( x_i \)); \( f-2 \): two original relation-based features (i.e., \( R1 \) and \( R2 \)); \( f-3 \): augmented features of content-based features and \( R1–R2 \). Based on these features, we consider two typical classification models, i.e., Naive Bayes (NB) and SVM. The experimental results shown in Table IV illustrates that feature engineering \( f-3 \) helps the performance of machine learning, but iTrustSO leveraging the knowledge represented as HIN and the long-range influence among code snippets learned from LSTM with attentions significantly outperforms other baselines. This again demonstrates that, to detect the insecure code snippets, iTrustSO using CodeHin2Vec to seamlessly integrate node content with HIN relations is able to build the higher-level semantic and structural connection between code snippets with a more expressive and comprehensive view and thus achieves better detection performance.

Table IV: Comparisons of other machine learning methods

<table>
<thead>
<tr>
<th>Metric</th>
<th>NB</th>
<th>SVM</th>
<th>iTrustSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACC</td>
<td>0.7493</td>
<td>0.6854</td>
<td>0.7753</td>
</tr>
<tr>
<td>F1</td>
<td>0.7284</td>
<td>0.6613</td>
<td>0.7834</td>
</tr>
</tbody>
</table>

F. Case Studies

To gain deeper insights into the security-related risks of modern social coding platform of Stack Overflow, in this section, based on our developed system iTrustSO, we further analyze 8,105 detected insecure code snippets in Stack Overflow. We categorize the security risks or vulnerabilities resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%). From these categories, we can observe that the most prevalent insecure code infiltration resulted from these insecure code snippets into six types: (1) Android Manifest configuration (28.43%), (2) WebView component (03.20%), (3) data security (22.62%), (4) file directory traversal (15.15%), (5) implicit intents (09.06%), and (6) security checking (21.55%).

Figure 8: Insecure codes with manifest vulnerabilities.
V. CONCLUSION

To address the code security issue in modern social coding platforms, in this paper, we bring an important new insight to exploit social coding properties in addition to code content for automatic detection of insecure code snippets in Stack Overflow. To depict the code snippets, we not only analyze the code content, but also utilize various kinds of relations among users, badges, questions, answers and code snippets in Stack Overflow. To model the rich semantic relationships, we first introduce a structured HIN for representation and then use meta-path based approach to incorporate higher-level semantics to build up relatedness over code snippets. Later, we propose a novel hierarchical attention-based sequence learning model named CodeHin2Vec to seamlessly embed code content with HIN-based relations for representation learning. After that, a classifier is built for insecure code snippet detection. Though it’s proposed for code security analysis, the embedding model CodeHin2Vec is a general framework which is able to learn desirable node representation in HIN and thus can be further applied to various network mining tasks, such as node classification, clustering and similarity search. The experimental results based on the data collections from Stack Overflow demonstrate that the developed system iTrustSO integrating our proposed method outperforms alternative approaches in insecure code snippet detection.

ACKNOWLEDGEMENT

This work is partially supported by the NSF under grants CNS-1618629, CNS-1814825, CNS-1845138 and OAC-1839909, the DoJ/NJ under grant NJI 2018-75-CX-0032, the WV HEPC Grant (HEPC.dsr.18.5), and the WVU RSA grant (R-844).

REFERENCES