DeepFPD: Browser Fingerprinting Detection via Deep Learning with Multi-Modal Learning and Attention

Weizhong Qiang, Kunlun Ren, Yueming Wu, Deqing Zou, and Hai Jin

***Abstract*—Browser fingerprinting is a stateless tracking technique that poses a significant security threat to users' privacy. However, the distinction between fingerprinting and non-fingerprinting scripts is far from well-defined, making the detection of fingerprinting scripts very challenging. Existing methods for detecting browser fingerprinting are based on heuristics or machine learning, and thus either require strictly defined rules or are not able to learn the features of fingerprinting scripts comprehensively, failing to detect a significant fraction of fingerprinting scripts.**

To detect browser fingerprinting more effectively, we propose a deep learning-based detection method, *DeepFPD*, in which multiple script modalities including tokens, abstract syntax trees, and control flow graphs are learned by using different specific neural networks to obtain lexical, syntax, and control flow information of the script code. Moreover, the attention mechanism is introduced to enhance the effectiveness of *DeepFPD*. The experimental results on the training dataset and test dataset constructed based on real-world scripts show that *DeepFPD* outperforms the state-of-the-art work with an F1-measure improvement of 8.3% and 18.7%, respectively.

***Index Terms*—Web privacy, Tracking, Browser fingerprinting detection, Multi-modal learning, Attention mechanism.**

I. Introduction

Browser fingerprinting is a technique of tracking web browsers through the visible configuration and settings of a website to browsers. Since Eckersley et al. [1] came up with the concept of "browser fingerprinting", many researchers have revealed new information for generating the browser fingerprints [2]. It is widely recognized as an abusive practice and a huge threat to browser users' privacy [3], [4], yet its stateless nature makes it difficult to detect. Many studies have measured the deployment of browser fingerprinting on the Web, and different methodologies have been used. For example, Nikiforakis et al. [5] used the fingerprinting codes provided by three companies as the basis for identifying fingerprinting scripts. Acar et al. [6] proposed the FPDetective, a framework for identifying fingerprinting scripts by finding behaviors directly related to fingerprinting activities. Engelhardt et al. [7] proposed a large-scale online measurement of tracking (including fingerprinting) and used some heuristic rules to filter fingerprinting scripts. However, existing studies rely on manual analysis and hard-coded heuristics, which are strictly defined to avoid a high false positive rate but result in a high false negative rate. In addition, with the development of browser fingerprinting techniques, it is difficult for heuristic rules to capture fingerprinting behaviors that evolve.

Even though the process of collecting browser information is relatively simple, analyzing this information to identify browser fingerprints is rather complicated, as the distinction between benign and fingerprinting scripts is far from well-defined. Learning-based methods are very suitable for solving classification issues with less well-defined boundaries, and thus they have also been used for identifying browser fingerprinting. Iqbal et al. [8] proposed a machine learning method based on the syntactic and semantics of the code to detect browser fingerprinting, representing scripts as abstract syntax trees (ASTs) and using a decision tree-based classifier. Their method achieved 99.9% accuracy in classifying fingerprinting scripts, which demonstrates the advantages of machine learning in detecting browser fingerprinting. However, the false negative rate of their method is relatively poor. In their static analysis, only the features extracted from the AST are used, while the deep structural features of the code are ignored. Moreover, they use a relatively simple decision tree-based algorithm. Their methods do not fully utilize the semantics of the code.

The browser fingerprinting detection methods based on static analysis are necessary. They have higher coverage and consume fewer resources compared to methods based on dynamic analysis. However, it is not easy to analyze source code to accurately understand its semantics and thus detect browser fingerprinting. The semantic information of the scripts consists of not only shallow source code information, such as variable names and API sequences, but also deep structural information, such as the control structure of the code, which is ignored by the previous learning-based methods [8].

To fill the gap, we propose *DeepFPD*, a static deep learning-based method for detecting browser fingerprinting, which

Weizhong Qiang and Deqing Zou are with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China; and Jinyinhu Laboratory, Wuhan, 430040, China (e-mail: wzqiang@hust.edu.cn, deqingzou@hust.edu.cn)

Kunlun Ren (corresponding author) is with School of Cyber Science and Engineering, Huazhong University of Science and Technology, Wuhan, 430074, China (kunlunren@hust.edu.cn)

Hai Jin is with National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China (e-mail: hjin@hust.edu.cn)

Yueming Wu is with Nanyang Technological University, School of Computer Science and Engineering 50 Nanyang Avenue, Singapore Singapore, Singapore (email: wuyueming21@gmail.com)

utilizes multi-modal representations of scripts. Deep learning has been proven to have great potential for modeling source code in recent years [9]–[13]. We utilize different neural networks to learn lexical, syntactic, and control flow information in tokens, ASTs, and CFGs (control flow graphs), respectively. Specifically, we use a long short-term memory (LSTM) network to learn the token sequence directly. The AST is first transformed into a sequence of tuples and then learned using another LSTM. A gated graph neural network (GGNN) is used to process the CFG. Furthermore, we are not utilizing these three forms separately but aggregating them. We aggregated the three kinds of information at the network level, using an attention mechanism to assign weights. In this way, *DeepFPD* fully utilizes the information of the code, which leads to a better ability to detect browser fingerprinting.

We train *DeepFPD* and perform 10-fold cross-validation on a dataset of 10,806 scripts, including 1,272 fingerprinting scripts and 9,534 non-fingerprinting scripts. The evaluation results show that *DeepFPD* can achieve 99.4% accuracy, 97.6% F1-measure, 0.1% FPR (false positive rate) and 3.8% FNR (false negative rate) in detecting fingerprinting scripts, significantly outperforming the state-of-the-art method *FP-Inspector* [8], with 8.3% higher F1-measure and 6.4% lower FNR, respectively. In addition, we evaluate *DeepFPD* on 2,082 randomly selected scripts from the huge number of scripts crawled on real-world websites and not in the above dataset. Manual analysis is conducted to label these 2,082 scripts, and the results evaluated on these scripts show that the performance of *DeepFPD* decreases compared to cross-validation, but is still the best compared to *FP-Inspector* and other methods, with an F1-measure of 65.6%, which is 18.7% higher than *FP-Inspector*, and 35.2%-50.7% higher than other methods.

We summarize our main contributions as the following:

1) We propose a deep learning-based method for the detection of browser fingerprinting, *DeepFPD*, in which different specific neural networks are leveraged to learn multiple modalities of scripts, including tokens, ASTs, and CFGs.
2) We aggregate the three kinds of information of scripts at the network level and use an attention mechanism to assign weights to make them an organic whole, leading to a better ability to detect browser fingerprinting.
3) We conduct exhaustive experiments and evaluate the effectiveness of *DeepFPD* on real-world scripts. Through the experimental results, we demonstrate that utilizing multiple modalities and attention mechanisms is indeed beneficial to improve the performance of the detection method.

The remainder of this paper is organized as follows. Section II presents the motivation example for illustrating the proposed multi-modal approach. Section III introduces the design of *DeepFPD* in detail. Section IV presents the experimental evaluation of the performance of *DeepFPD*, as well as the contribution of multi-modal and attention mechanisms to *DeepFPD*. Section V discusses the limitation of *DeepFPD*. Section VI introduces some related works. Section VII concludes this paper.

II. Motivation Example

To better illustrate the key insight of our proposed approach, we present a motivation example here. Script 1 shows a font fingerprinting script that fingerprints the browser by enumerating the fonts supported by the browser. Script 2 is a normal script whose purpose is to configure the browser's settings. The ASTs obtained from these two scripts are very similar. Figure 1 shows the main part of two scripts' ASTs of the two scripts above. The approach of Iqbal et al. [8] to detect fingerprinting scripts is to transform the script into an AST, then extract features from the AST as the features of the script, and finally utilize a decision tree classifier to obtain the result. It is obvious that this approach can hardly be used to distinguish between the two scripts above, since they have almost the same ASTs. However, it can be noticed that the two scripts are very different at the lexical level. Therefore, if the AST and lexical information of the script can be leveraged in combination, it will be easier to distinguish the two scripts.

```
baseFonts = ['monospace','sans-serif','serif'];
fontList = ['...'];
baseFontsSpans = baseFonts.map(createSpan)
fontSpans = {}
for (font in fontList){
  fontSpans[font] = baseFonts.map((baseFont) => createSpanWithFonts(font,
      baseFont))}
defaultWidth = {}
defaultHeigh = {}
for (index = 0; index < baseFonts.length; index++){
  defaultWidth[baseFonts[index]] = baseFontsSpans[index].offsetWidth
  defaultHeigh[baseFonts[index]] = baseFontsSpans[index].offsetHeight}
avalibleFontList = []
for (font in fontList){
  if(baseFonts.some((baseFont, baseFontIndex) =>
    fontSpans[baseFontIndex].offsetWidth !== defaultWidth[baseFont] ||
    fontSpans[baseFontIndex].offsetHeight !== defaultHeigh[baseFont])== true)
  avalibleFontList.append(font)}
```

. Script 1: A fingerprinting script example

```
default_set = {radio: 0, checkbox: 0, file: 0,
           password: 0, image: []}
show_image = []
remote_set = {
    radio: !0, checkbox: !0, file: !0, password: !0, image: [...]}
opt_list = []
for (b in remote_set){
    opt_set[b] = remote_set[b]}
for (b = 0; b < remote_set['image'].length;b++){
    if(b < 10) show_image[b] = opt_set['image'][b]}
for (b in default_set){
    if(opt_set[b])
        opt_set[b] = na(b)}
```

. Script 2: A normal script example

```
baseFonts = ['monospace','sans-serif','serif'];
fontList = ['...'];
baseFontsSpans = baseFonts.map(createSpan)
fontsSpans = {}
font = fontList[0]
fontsSpans[font] = baseFonts.map((baseFont) => createSpanWithFonts(font,
      baseFont))
defaultWidth = {}
defaultHeigh = {}
for (index = 0; index < baseFonts.length; index++){
  defaultWidth[baseFonts[index]] = baseFontsSpans[index].offsetWidth
  defaultHeigh[baseFonts[index]] = baseFontsSpans[index].offsetHeight}
avalibleFontList = false
fontSpans = fontsSpans[font]
if(baseFonts.some((baseFont, baseFontIndex) =>
fontSpans[baseFontIndex].offsetWidth !== defaultWidth[baseFont] ||
fontSpans[baseFontIndex].offsetHeight !== defaultHeigh[baseFont])){
  avalibleFontList = true}
```

. Script 3: A slightly modified version of Script 1

Moreover, without considering the full semantics of the script, it will not be sufficient to distinguish scripts. For example, Script 3 is a slightly modified script of Script 1, and its purpose is to determine whether a font is available in the browser. It can be seen that the token sequences of the two scripts are very similar, but the semantics of them are

considerably different. The reason for this huge difference in semantics is that the control structure of the script is modified, and this difference can be seen in the CFGs.

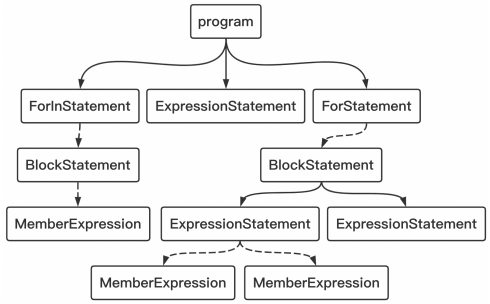

Fig. 1. The main part of Script 1's and Script 2's ASTs

In summary, using only one form to represent scripts is not enough, so it is necessary to use multiple forms of scripts, including token, AST, and CFG, to more accurately identify scripts.

III. System Design

A. Overview

In this section, we introduce *DeepFPD*, a deep learning-based approach that can learn multi-modal representations of JavaScript code to detect browser fingerprinting. *DeepFPD* consists of three main components, namely, *dataset collection*, *script conversion*, and *classifier based on multi-modal neural network*.

B. Dataset Collection

DeepFPD is based on deep learning and therefore requires a large amount of labeled data for training, but so far there is no available dataset. Therefore, we can only crawl scripts from real-world websites and then label them based on a combination of heuristic rules and manual analysis.

First, the scripts are crawled from large-scale websites. Then, possible fingerprinting scripts are selected from them by using the heuristic rules in [7] and [8]. Englehardt et al. [7] define the detection rules for several types of fingerprinting, which include *canvas fingerprinting*, *canvas font fingerprinting*, *webRTC-based fingerprinting*, and *AudioContext fingerprinting*. Iqbal et al. [8] make some slight modifications to these rules to reduce the false positives. We use the version of Iqbal et al. directly, details of which can be found in their paper. Additionally, if there is a collection of attributes or screen APIs, it is also selected as a suspicious script for further analysis. Next, these selected scripts are manually analyzed. When performing manual analysis, the scripts are analyzed by two reviewers independently, and only if both are sure that a script is a fingerprinting script, it is labeled as a positive sample.

Since fingerprinting scripts represent only a very small fraction of real-world scripts, the number of positive samples we can obtain is relatively small. Therefore, the data published by Iqbal et al. [8], which contains fingerprinting scripts, i.e. positive samples, are used to extend the positive samples. Specifically, we remove the duplicated samples and combine the two sets of fingerprinting scripts to form the positive samples in our dataset.

```
canvas.width = 122
canvas.height = 110
context.globalCompositeOperation = 'multiply'
arrayList = [['#f2f', 40, 40],['#2ff', 80, 40],
            ['#ff2', 60, 80]]
for (const i in arrayList) {
  context.fillStyle = arrayList[i][0]
  context.beginPath()
  context.arc(arrayList[i][1], arrayList[i][2], 40, 0, Math.PI * 2, true)
  context.closePath()
  context.fill()}
context.fillStyle = '#f9c'
context.arc(60, 60, 0, Math.PI * 2, true)
context.arc(60, 60, 20, 0, Math.PI * 2, true)
context.fill('evenodd')
canvas.toDataURL()
```

. Script 4: A canvas fingerprinting script example

C. Script Conversion

As mentioned above, it is not enough to just utilize tokens to represent code. We simultaneously utilize different representations of the code to obtain the lexical, syntactic, and control flow information of the code. Specifically, we get the lexical information from tokens of the script. The abstract syntax tree (AST) is a tree-like data structure specialized for representing the syntactic structure of program code. We exploit it to get the syntactic information. The control flow graph (CFG) is a graphical data structure used to represent the control flow of code. We utilize it to get the control flow information. Overall, we convert the scripts into tokens, ASTs, and CFG, and we obtain richer information about the script by utilizing these three forms simultaneously.

1) Tokens: In 2012, Hindle et al. [14] first pointed out that programs are human-written languages with a repetitive nature, with several statistical characteristics that can theoretically be captured by language models. Based on this theory, deep learning methods can be used to learn the probability distribution among code tokens, thereby modeling the probability of code token sequences.

To collect tokens of the script, we first tokenize the source code. Since the script code includes many custom words, a large number of words will be obtained after tokenization, many of which may interfere rather than benefit the model. These custom words are low-frequency words, which are usually the noise for code classification. Therefore, a word is only preserved if its frequency exceeds the threshold, or if it is part of JavaScript APIs. For the determination of the frequency, we rely on the empirical "Pareto principle", that is to say, the corpus obtained by this frequency can reduce the length of the script's token sequence to about 20% of its original length, and the final empirical value used is 50.

2) ASTs: Unlike natural languages, the syntactic structure of programming languages is very standardized. We apply AST, which is an abstract representation of the syntactic structure of the source code, to reveal information about the syntactic of the script. An AST represents the syntactic structure in the form of a tree, and each node in the tree represents a structure in the code.

ASTs have been used in previous studies to construct features for learning-based methods [8], [9], [15]–[19]. In *DeepFPD*, the AST of the script is traversed to form a

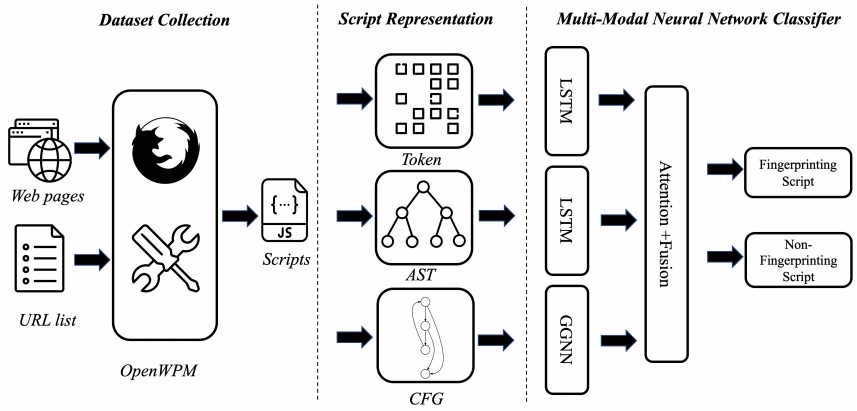

Fig. 2. System overview of *DeepFPD*

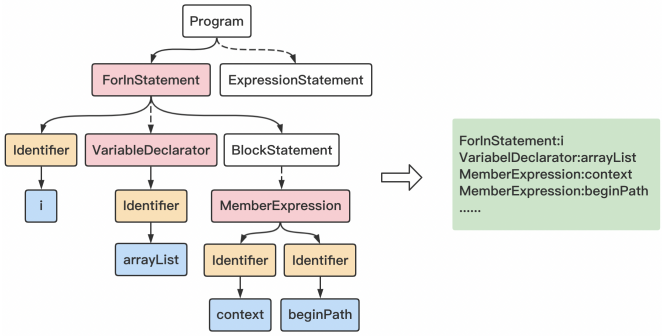

Fig. 3. A schematic of converting AST of Script 4 into tuples

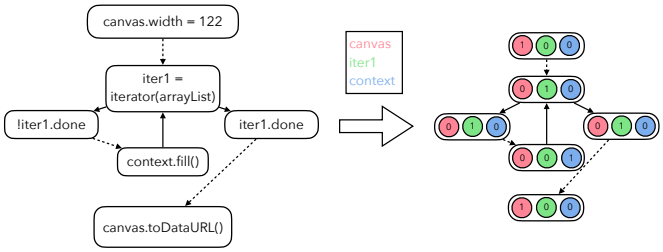

Fig. 4. A schematic of converting CFG of Script 4 into vectors

sequence, and then it is modeled using a sequence-based model.

To retain more syntactic information, sequences of tuples are extracted from the AST instead of directly converting to sequences of nodes by traversing the tree. Specifically, the leaf nodes of an AST represent the most basic syntactic units of the source code, such as identifiers, literals, operators, keywords, and so on. The parent nodes of the leaf nodes represent the content or syntactic structure in which these fundamental syntax elements are located, such as if statements, while statements, variable declarations, and so on. We use *Esprima* [20], a standard-compliant ECMAScript parser, to represent scripts as ASTs. Here only three types of leaf nodes are kept, which are identifier, literal, and operator. Then we traverse the AST to get all the leaf nodes and the corresponding parent nodes. Leaf nodes and corresponding parent nodes are kept in pairs. We save leaf nodes not by their type, but by specific values such as variable names, strings, addition symbols, etc. The AST is then converted into a sequence of tuples of the form "*content:value*", where *content* is the syntax unit, such as *IfStatement*, *value* represents the specific value under the specific value, such as "*i*". In this way, we preserve the correspondence between basic units in the source code and the syntactic structures to which they belong. If we just perform such a naive conversion, we will end up with a very large number of tuples. To avoid this, we only consider tuples that contain at least one keyword that matches a word in one of the JavaScript APIs. That is, the obtained tuples whose *value* is not a word in one of JavaScript APIs are removed.

Script 4 shows a part of a canvas fingerprinting script. The purpose of this script is to render the canvas and read the rendered image data. Figure 3 shows how the AST of Script 4 is converted into tuples. In this way, we can achieve serialization of the AST, and thus we can represent the AST using a similar model that represents tokens.

3) CFGs: As mentioned above, code has structural information, which cannot be represented by using only tokens and ASTs. The CFG is a graphical representation of all possible paths a program may traverse during execution, which enables access to information about the program structure.

Specifically, we obtain the CFG of each script through an open source tool named *styx* [21], where each node represents a code statement and edges indicate the flow of code execution. There are a large number of nodes in some CFGs, not all of which are necessary, and may also introduce noise. Also, too many nodes can introduce a huge overhead on graph learning. So we remove some unnecessary nodes. Only the node is preserved when it corresponds to a statement including at least one word matching the name of one of the script APIs or one word with a sufficiently large frequency. Empirically, we sort the words of all the files by word frequency and then take the top 20%. To further facilitate subsequent graph learning, each node of the CFG is represented by a 256-dimensional vector. TF-IDF (term frequency–inverse document frequency) is applied to rank the words in the script and the top 256 words are selected. Each dimension in the node vector represents whether the corresponding word is in the statement represented by this node. If the statement contains one of these words,

the value representing that sub-dimension will be set to 1, otherwise it will be set to 0. Then the nodes whose vector values are all zero are removed. After we remove the node, we connect the nodes pointing to this removed node to the nodes pointed to by this removed node. This maximizes the retention of control information and enables information aggregation. Figure 4 shows the CFG generated by Script 4 and how the nodes in the CFG are converted to vectors.

D. Classifier Based on Multi-Modal Neural Network

We present a hybrid neural network for classifying fingerprinting scripts. Since different representations of the code contain different information, we use three kinds of networks for code embedding. We apply two LSTM-based networks to embed tokens and features extracted from ASTs, a GGNN-based network to embed CFGs. Then, we utilize an attention mechanism to fuse three code embeddings.

The network to embed the tokens. Similar to natural language, after the code is converted into tokens, the semantic information of the code is hidden in the sequence features in it. As recurrent neural networks (RNNs) have been successful in natural language processing, and long short-term memory (LSTM) improved from recurrent neural networks has also been shown to be effective in learning sequence features [22], [23], we use LSTM to learn the sequence features of the code. The main difference between LSTM and RNN is that LSTM places more emphasis on memory blocks, thus adding memory cells to hold long-term states. An LSTM network can remember past information and associate it with current data. Its internal structure includes input gates, forget gates, and output gates, enabling it to better handle long-term dependencies.

In our network setup, tokens are passed sequentially into the LSTM as timing sequences x_i, and each result is passed into the next layer as the input h_i for the hidden layer. Thus, tokens are embedded as follows:

$$h_i^{Token} = LSTM(h_{i-1}^{Token}, w(x_i)) \tag{1}$$

where $i = 1, ..., |x|$, w is the embedding layer that maps tokens into vectors. h_{i-1}^{Token} is the hidden state of step $i-1$, and h_i^{Token} is the hidden state of step i. The final hidden state $h_{|x|}^{Token}$ is the embedding of tokens of a script, notated as u^{Token}. The embedding is a vector of fixed length that can be considered as encoding the lexical information of the script in a continuous vector space.

The network to embed the ASTs. As mentioned above, the sequence of tuples is extracted from the AST as the features. In this way, the tree structure is converted into a sequence, and the AST can be learned using the model that learns sequence features.

We also use LSTM as the model for learning the AST. Similar to the steps for processing tokens, tuples we obtain from traversing the AST are passed sequentially into the LSTM as timing sequences x_i. The ASTs are embedded as follows:

$$h_i^{AST} = LSTM(h_{i-1}^{AST}, w(x_i)) \tag{2}$$

where $i = 1, ..., |x|$, w is the embedding layer that maps tuples into vectors. However, since the AST contains different information than the tokens, the model will be different in parameters and structure from the model learning the tokens, making the mode better embed the AST. The embedding layer here is different from the one that embeds the tokens, with different input and output dimensions. The final hidden state $h_{|x|}^{AST}$ is the embedding of tuples of a script's AST, notated as u^{CFG}. It can be considered as encoding the syntactical information of the script.

The network to embed the CFGs. Different from the tokens and ASTs, the CFGs represent the code in the form of graphs, and the structure of the graph can well represent the structural information of the code. Deep learning has made considerable progress on graph data, resulting in the deep graph neural network, a graph-based deep learning method [24], [25].

We use the gated graph neural network (GGNN) [26], which has the advantage of learning directed graphs, to embed the CFG. We define the CFG as $G = (V, E)$, where V is the set of nodes (v, l_v) of the CFG, E is the set of edges (e, l_e) of the control flow, l_v and l_e are the labels of nodes and edges. Then we use GGNN to learn the embedding of node v in multiple iterations. For each round t, each node $v \in V$ receives the message $m_{v,t}$ aggregated from its neighbours,

$$m_{v,t} = \sum_{v' \in N(v)} W_{l_e} h_{v',t-1} \tag{3}$$

where $N(v)$ is the neighbours of vertex v, W_{l_e} maps the message from the hidden state h of each neighbour. For each node $v \in V$, the GGNN updates its hidden state h with Gate Recurrent Unit (GRU).

$$h_{v,t}^{CFG} = GRU(h_{v,t}^{CFG}, m_{v,t}) \tag{4}$$

The node vector x_v is a 256-dimensional vector initialized as mentioned above. Then the node vectors are passed into the GRU as the initial hidden layer state. After T rounds of iterations, the final state is used as a vector representation of the node. A graph-level representation is generated by averaging the representations of all nodes, which is the embedding of the script's CFG, notated as u^{CFG}.

Attention fusion. Next, we integrate the multi-modal embeddings of a script into one embedding. When fusing different code embeddings together to accomplish subsequent classification tasks, different parts may have different importance, so the method of assigning different weights to them is critical. The human *Attention Mechanism* derived from intuition, which devotes more attention resources to the focused target area to obtain more detailed information about the target, thus suppressing other useless information. The attention mechanism has been widely applied to different types of deep learning tasks such as image classification and natural language processing [27]–[30]. The effectiveness of these methods has been significantly improved with attention mechanism.

There are a variety of attention methods. In our network, we apply self-attention [31] to fuse three distinct embeddings. Self-attention is a mechanism for modeling relationships

between elements within sequence data. Self-attention is a foundational component of the Transformer model architecture, which has achieved state-of-the-art results in numerous natural language processing tasks. It is inherently adaptable, automatically assigning varying attention weights to different elements.

Specifically, to compare the effect of the methods of assigning weights at different levels, we implement two attention mechanisms. One is to concatenate the three types of embedding and apply self-attention to them as a whole. We refer to this attention mechanism as *global attention*. The other is to utilize multi-head attention, applying self-attention to each type of embedding before concatenating them together. We just refer to this attention mechanism as *multi-head attention*. Note that the *global attention* and *multi-head attenion* here refer simply to the two attention mechanisms, and the same below.

Global attention allows the model to capture a global representation directly on all three types of embedding.

First, the three types of embedding are concatenated together.

$$u = concat(u^{Token}, u^{AST}, u^{CFG}), \tag{5}$$

where u^{Token}, u^{AST}, and u^{CFG} are the three embeddings previously obtained, and u is the vector concatenated from these embeddings. Then, the self-attention computation is performed with u as the input vector.

$$Q = uW_Q, K = uW_K, V = uW_V, \tag{6}$$

where u is linearly transformed into three vectors: Query Q, Key K, and Value V. W_Q, W_K, and W_V are weight matrices for the linear transformations. Q and K are used to calculate the attention scores.

$$A = softmax(\frac{QK^T}{\sqrt{d_k}}), \tag{7}$$

where d_k is the dimensionality of the Key vectors, and A is the attention scores, that is, the weights corresponding to the elements in u. So we can get the final representation x:

$$x = AV. \tag{8}$$

Multi-head attention allows the model to consider multiple perspectives including token, AST, and CFG when capturing dependencies within the input. We perform self-attention computation for each type of embedding.

$$Q_i = u_i W^{Q_i}, K_i = u_i W^{K_i}, V_i = u_i W^{V_i}, \tag{9}$$

$$x_i = softmax(\frac{Q_i {K_i}^T}{\sqrt{d_{ki}}})V_i, \tag{10}$$

where i denotes one of tokens, AST, and CFG. The final representation x is obtained by concatenating the three attention heads:

$$x = concat(x_{Token}, x_{AST}, x_{CFG}). \tag{11}$$

Finally, the output of the attention mechanism is passed through a final linear layer and a cross-entropy loss function to obtain the loss. We train this model with the labels of scripts.

$$Loss = cross - entropy(xW, y), \tag{12}$$

where W is the weight matrix of the last layer, and y is the label of the script, which is binary here.

IV. Evaluation

In this section, we describe the experimental setup in detail and evaluate the performance of *DeepFPD* in detecting fingerprinting scripts and the contribution of different parts of *DeepFPD* to its performance.

A. Dataset

We first use OpenWPM [32] to crawl scripts from Alexa Top-10k websites and obtain 65,203 scripts. It is worth mentioning that some scripts that are too short or too long need to be removed because scripts that are too short contribute little to the deep learning task, while scripts that are too long greatly increase memory consumption during training. Then, for the convenience of manual analysis and without loss of generality, we randomly select 10,000 scripts from the remaining scripts and use the method mentioned in Section III-B to label the positive samples and negative samples. As a result, 229 positive samples and 9,534 negative samples are obtained, and the remaining samples very difficult to analyze are excluded.

Then, we remove duplicate URLs from the list reported by Iqbal et.al [8], download those fingerprinting scripts, and combine these 1,043 positive samples with the positive samples obtained from our analysis. Finally, 1,272 positive samples and 9,534 negative samples make up our *training dataset*. Note that the experiments on the *training dataset* are conducted by performing 10-fold cross-validations.

In addition, due to the lack of ground truth, to evaluate our method more convincingly, we construct a *test dataset* based on manual analysis. Similarly, to reduce the overhead of manual analysis, but without loss of generality, we randomly select 2,082 samples from the remaining crawled scripts for manual analysis. In total, 62 fingerprinting scripts and 2,020 non-fingerprinting scripts are manually labeled from these samples.

B. Evaluation Metrics

The experiments are conducted on the above dataset to test the effectiveness of *DeepFPD*. The detection of fingerprinting scripts is a kind of classification task, that is, for each input sequence, the model outputs a classification result, which represents the type of the sequence. Therefore, we measure the performance of *DeepFPD* using widely used metrics, namely, accuracy, precision, recall, F_1-measure, false positive rate, and false negative rate, which are defined in Table I.

C. Implementation Details

To conduct the experiments, we use Pytorch (version 1.0.1) to implement *DeepFPD*, and scikit-learn (version 0.24.1) to implement *FP-Inspector* [8] and other machine learning based approaches. All experiments are performed on Ubuntu 18.04.1 with Quadro RTX 5000 and 32GB physical memory. For the base models, we set dropout rate $P_{dropout}$ to 0.1 and the

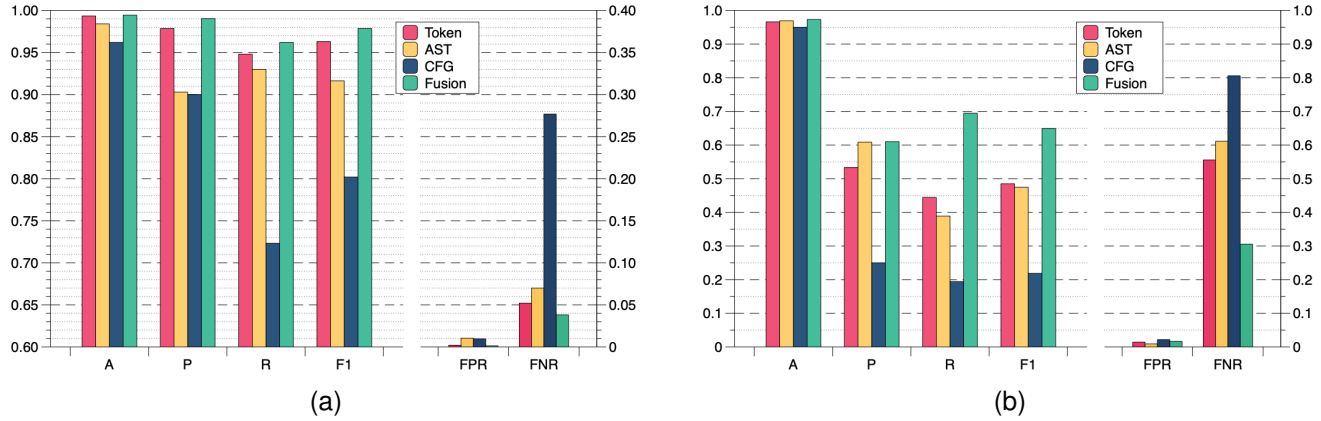

Fig. 5. Comparison of tokens network, ASTs network, CFGs network, and fusion network. (a) On the *training dataset*. (b) On the *test dataset*

TABLE I
METRICS USED IN THE EXPERIMENTS

Metrics	Abbreviation	Definition
Accuracy	A	(TP + TN)/(TP + FP + TN + FN)
Precision	P	TP/(TP + FP)
Recall	R	TP/(TP + FN)
F1-measure	F1	$2 \cdot P \cdot R/(P+R)$
False Positive Rate	FPR	FP/(FP + TN)
False Negative Rate	FNR	FN/(FN + TP)

model dimension d_{model} to 64 to conduct the experiments. Meanwhile, the number of LSTM layers is set to 2, and the number of GRU propagation steps is set to 5.

D. Performance

To evaluate the effectiveness of *DeepFPD* in detecting fingerprinting scripts, we perform 10-fold cross-validation on the *training dataset* and also test the trained model on the *test dataset*. In addition, to validate the superiority of *DeepFPD* over machine learning methods, we also implement *Logistic Regression* (LR), *K Nearest Neighbors* (KNN), *Support Vector Machine* (SVM), and *Random Forest* (RF) for comparison. Similar to the previous method [8], these machine learning methods utilize ASTs for classification.

TABLE II
COMPARISON OF *DeepFPD* AND OTHER MACHINE LEARNING METHODS ON THE TRAINING DATASET

Methods	A(%)	P(%)	R(%)	F1(%)	FPR(%)	FNR(%)
LR	97.8	96.7	83.5	89.6	0.38	16.5
KNN	97.2	92.7	83.4	87.5	0.96	16.6
SVM	97.7	95.7	84.3	89.6	0.50	15.7
RF	98.5	98.0	88.5	93.0	0.24	11.5
DeepFPD	**99.4**	**99.0**	**96.2**	**97.6**	**0.13**	**3.81**

The experimental results on the *training dataset* are presented in Table II. Compared with other methods, *DeepFPD* performs the best on all metrics, achieving 99.4% accuracy and 97.6% F1-measure. *DeepFPD* also achieves the best results in terms of false positive rate and false negative rate, reaching 0.13% and 3.81%, respectively. The machine learning-based methods perform poorly on FNR, 7.7%-12.8% higher than *DeepFPD*, which suggests their inadequate ability to learn fingerprinting script features. In terms of F1, a comprehensive metric, *DeepFPD* is 4.6%-10.1% higher than them. Overall, *DeepFPD* performs well on all metrics and outperforms machine learning-based methods on the *training dataset*.

TABLE III
COMPARISON OF *DeepFPD* AND OTHER MACHINE LEARNING METHODS ON THE TEST DATASET

Methods	A(%)	P(%)	R(%)	F1(%)	FPR(%)	FNR(%)
LR	97.6	59.7	59.7	59.7	1.2	40.3
KNN	97.4	59.5	40.3	48.1	0.8	59.7
SVM	97.0	49.4	62.9	55.3	2.0	37.1
RF	**98.3**	**96.6**	45.2	61.5	**0.1**	54.8
DeepFPD	98.0	66.7	**64.5**	**65.6**	1.0	**35.5**

Table III shows the results of each method on the *test dataset*, from which it can be seen that there is a significant degradation in performance for all methods, and the results show similar features to those in the *training dataset*, namely, low FPR and high FNR. Although very low FPRs of 0.8% and 0.1% are achieved in *KNN* and *RF*, their FNRs are excessively high, reaching 59.7% and 54.8% respectively. Combining the FPR and FNR, *SVM*, and *LR* perform relatively better, but all are worse than *DeepFPD*, with FPR and FNR 0.2%-2.0% and 1.6%-4.8% higher, respectively. In terms of F1, *DeepFPD* performs the best, with 4.1%-17.5% higher than other methods.

In summary, *DeepFPD* performs best both on the *training dataset* and *test dataset*. However, the performance on the *test dataset* decreases compared to that on the *training dataset*. The reason could be that the number of positive samples in the *training dataset* is still too small, which does not contain certain types of fingerprinting, causing some scripts in the *test dataset* to be undetectable.

E. Comparison with Other Methods

In this subsection, we compare *DeepFPD* with three state-of-the-art browser fingerprinting detection methods:

- *Heuristic*: The detection rules are first defined by Englehardt et al. [7] and then made some modifications by Iqbal et al. [8]. We implement the rules ourselves as described in the paper by Iqbal et al.
- Method proposed by *Zalingen et al.* [33]: The method detects browser fingerprinting through static code analysis and a support vector machine classifier. We directly use their implementation available on GitHub [34].
- *FP-Inspector* [8]: Iqbal et al. propose *FP-Inspector*, which combines static and dynamic analysis and uses a decision tree classifier to detect browser fingerprinting. Since *DeepFPD* is a static method, we use *FP-Inpector*'s static part for comparison. The method is not open source, so we replicate its static part in our experiments.

Table IV presents the results of *DeepFPD* and three comparative methods for fingerprinting detection on the *training dataset* and *test dataset*. Since *Heuristic* does not require training, to be consistent with the other methods, the results are obtained by randomly sampling one-tenth of the data in the *training dataset*.

From the results on the *training dataset* we can see that *DeepFPD* performs best on all metrics. In terms of the most comprehensive metric, F1, *DeepFPD* outperforms *Heuristic*, *Zalingen et al.*, and *FP-Inspector* by 77%, 63.8%, and 8.3%, respectively. *Heuristic* performs worst, with an F1 of only 20.6%. The inflexibility of these human-set rules makes it easy for fingerprinting scripts to escape their detection. *Zalingen et al.* performs just slightly better than *Heuristic*, with an F1 of 33.8%. The feature extraction of JavaScript code in the method is very simple and unable to cope with a variety of fingerprinting scripts. *FP-Inpector* performs better and can achieve 89.3% F1, but it is still significantly inferior to our method. It only utilizes the ASTs of scripts and doesn't have sufficient insight into the scripts.

The performance of all three comparative methods on the *test dataset* also decreases, and *DeepFPD* still performs significantly better than them. The only exception is that *DeepFPD* has a higher FPR than *Zalingen et al.* However, the recall of *Zalingen et al.* is very low. This suggests that despite having a low FPR, it comes at the cost of only being able to detect a very small fraction of fingerprinting scripts. In terms of F1, *DeepFPD* is 18.7%-50.7% higher than the three methods.

Furthermore, we compare the fingerprinting scripts detected by each method through manual analysis. Fingerprinting scripts detected by *Heuristic* are all detectable by other methods. There is only one script that *Zalingen et al.* detects that *DeepFPD* does not. This script is slightly obfuscated by putting all readable information such as variable names, function names, and so on into an array, and then replacing the use of these names with access to the array. The feature of *Zalingen et al.* utilizing the suspicious JS call happens to make it possible to detect this script. 11 scripts are detected by *FP-Inspector* exclusively. These scripts are characterized by the fact that the fingerprinting behavior is relatively obscure. Manual analysis also requires great scrutiny to obtain its semantics. The performance of learning-based methods on such scripts is difficult to explain. On these scripts, *DeepFPD* may fail compared to other learning-based methods. In practice, a better option is to combine several different learning-based methods to detect browser fingerprinting.

F. The Contribution of Fusion and Attention

1) Fusion: We compare three separate networks with the fusion network to verify the effectiveness of the multi-modal fusion. The experiments are conducted on the fusion network, as well as separate networks that embed tokens, ASTs, and CFGs, respectively.

Figure 5 shows the results on the *training dataset* and *test dataset*. As shown in Figure 5a, three separate networks achieve relatively high performance results, while *DeepFPD*, which combines the three networks through a fusion layer, outperforms any single network on all metrics. In particular, for the F1-measure metric, the value of fusion network is 1.3%, 6.0% and 17.4% higher than that of the tokens network, the ASTs network, and the CFGs network, respectively. For the FNR metric, the value of fusion network is 1.4%, 3.2%, and 23.9% lower than that of the three networks, respectively.

For the results on the *test dataset*, as shown in Figure 5b, a similar pattern can be seen, i.e., *DeepFPD* also outperforms any single network. The F1-measure of the fusion network is 16.5%, 17.5%, and 43.0% higher and the FNR is 25.0%, 30.6%, and 50.0% lower than those of the tokens network, the ASTs network and the CFGs network, respectively.

Therefore, it can be concluded that *DeepFPD* can improve the effectiveness in identifying fingerprinting scripts through a multi-modal deep neural network that can learn by combining lexical, syntactic, and structural information.

2) Attention: To find out the contribution of the attention mechanism to the performance, we set up experiments on three networks with and without attention, as well as experiments on the fusion networks under the simple *linear* layer, the *global attention* layer, and the *multi-head attention* layer, respectively.

Figure 6 shows the results of these networks on the *training dataset* and *test dataset*. For the results on the *training dataset*, it can be seen that the networks with attention mechanism, whether used for embedding tokens, ASTs, or CFGs, show more or less improvement in most metrics compared to the networks without attention mechanism. The improvement is particularly significant for the tokens network, with the one with attention being 4.0% higher in F1 and 7.6% lower in FNR than the one without attention. The performance of the network embedding CFGs with attention is improved compared to that without attention, with a 2.8% higher F1, but the performance of the network embedding ASTs with attention is decreased slightly, with a 1.8% lower F1. For the fusion network, we can see that the network with global attention is also significantly more effective than the network simply fused with a linear layer, with 2.0% higher F1-measure and 2.9% lower FNR respectively. However, the multi-head attention fusion network performs slightly worse than the global attention network, which could be caused by conflicts resulting from the use of multiple layers of attention mechanisms.

The results on the *test dataset* also show a similar pattern. The networks with the attention layer are higher in F1 compared to the ones without the attention layer. Compared

TABLE IV
COMPARISON OF *DeepFPD* AND OTHER BROWSER FINGERPRINTING DETECTION METHODS

Methods	Dataset	A(%)	P(%)	R(%)	F1(%)	FPR(%)	FNR(%)
Heuristic	Training	94.8	21.9	19.4	20.6	2.5	80.6
	Test	96.2	21.9	11.3	14.9	1.22	88.7
Zalingen et al.	Training	95.7	35.1	33.9	33.8	2.0	66.1
	Test	97.0	46.7	22.6	30.4	**0.8**	77.4
FP-Inspector	Training	97.5	89.9	90.0	89.3	1.5	10.2
	Test	95.9	38.0	61.3	46.9	3.0	38.7
DeepFPD	Training	**99.4**	**99.0**	**96.2**	**97.6**	**0.1**	**3.8**
	Test	**98.0**	**66.7**	**64.5**	**65.6**	1.0	**35.5**

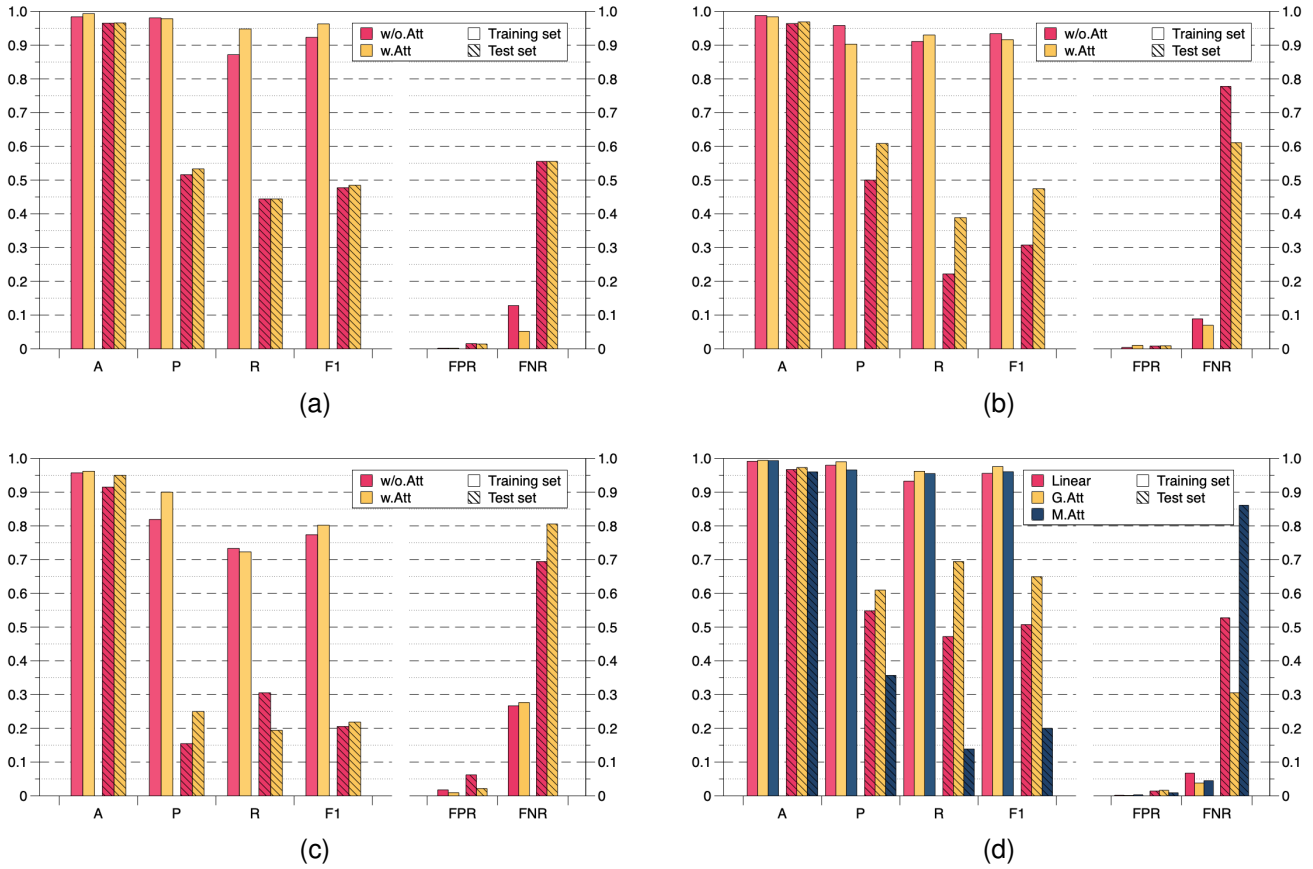

Fig. 6. Comparison of tokens network, ASTs network, CFGs network without attention (w/o.Att) and with attention (w.Att), and the fusion network with linear layer (Linear), global attention (G.Att), multi-head attention (M.Att). (a) The network for tokens with/without attention. (b) The network for ASTs with/without attention. (c) The network for CFGs with/without attention. (d) The fusion network with linear layer, global attention, and multi-head attention.

to networks without the attention layer, the networks for embedding tokens and CFGs with the attention layer are 0.7%, and 1.3% higher in F1. The network for embedding ASTs with the attention layer shows a significant improvement, which is 16.7% higher in F1 and 16.7% lower in FNR. For the fusion network, global attention also performs the best, with 14.2% and 44.9% higher in F1 and 22.2% and 55.7% lower in FNR than linear fusion and multi-head attention.

As a result, it can be seen that the attention mechanism helps to improve the performance of detecting fingerprinting scripts. However, a simple linear layer, such as the linear fusion layer described above, can be counterproductive and even less effective than a separate network, while too many fusion layers do not substantially improve the fusion effect, but rather have a negative impact.

G. Runtime Overhead

In this part, we discuss the time overhead of *DeepFPD*. Table V presents the average time consumed and standard deviation of each period of *DeepFPD* on **one** file. The model is trained for 50 epochs. The experimental results are obtained by running *DeepFPD* three times on our *training dataset* and then averaging the time. Note that the standard deviation of training and classification is not included, as the optimization of

deep learning libraries for large-scale data makes the standard deviation here not so meaningful.

In script conversion, the time overhead of converting scripts to tokens is small, averaging 2 ms per file. Whereas operations involving graphs incur higher overhead, the time overhead of converting scripts to CFGs is an order of magnitude higher than converting scripts to tokens. The part of AST has the highest overhead, an order of magnitude higher than CFG. This is because we perform operations of traversing the AST to fetch tuples, which is very computationally expensive. All three parts of time overhead have relatively large standard deviations. The reason for this is that their runtime is related to file size, which varies greatly from file to file. The training phase is the most time-consuming, more than two orders of magnitude higher than the other parts. The classification phase, on the other hand, is much faster, taking only about 0.1 s per file on average.

Of all the parts, the training phase is the most time-consuming. It may take days to train the model on large-scale data. However, the training phase only needs to be run once. In the practical case of detection, only the script conversion and classification are required. That means that a file only takes about 0.3s on average. The time is acceptable for large-scale detection.

TABLE V
DeepFPD'S RUNTIME OVERHEAD

Period		Average time consumed per file (s)	Standard Deviation (s)
Script Conversion	Token	0.002	0.003
	AST	0.161	0.150
	CFG	0.038	0.057
Training		11.520	-
Classification		0.111	-

V. DISCUSSION

Based on deep learning, our method performs well on all metrics for identifying fingerprinting scripts, unlike previous methods with low FPR and high FNR. This proves the advantage of deep learning in terms of abstract information such as semantics. In particular, a single modality does not contain enough semantic information about the code. To learn the semantics of the script code more comprehensively, inputting the three modalities of the code into the corresponding specific network for training, and finally performing the fusion, will indeed improve the effectiveness. Furthermore, the attention mechanism allows the deep network to focus more on where the semantics of the code are determined, which is consistent with the intuition of understanding code semantics. Given this, we believe that *DeepFPD* learns the semantics of the code more comprehensively, and therefore, there is a reason why it performs better at identifying fingerprinting scripts. We also believe that *DeepFPD* can be extended to other tasks related to code semantics, such as identifying malicious scripts.

However, there are still some limitations of *DeepFPD*.

Dataset size. A sufficient number of samples is beneficial to the effect of deep learning. However, collecting unique fingerprinting scripts is challenging because websites usually replicate fingerprinting scripts from each other, so only 1,272 fingerprinting scripts are included in the experiments, which is comparable to the number reported in the previous study [8]. This may prevent *DeepFPD* from learning all the features of fingerprinting scripts. To alleviate this limitation, it is necessary to combine *DeepFPD* prediction and manual analysis on a large scale on more sites to increase the number of positive samples.

Obfuscated scripts. Some fingerprinting scripts are obfuscated to avoid detection. In our experiments, obfuscated scripts have not been deobfuscated, which should have an impact on their performance and needs to be further evaluated and resolved.

VI. RELATED WORK

A. Browser Fingerprinting

Browser fingerprinting can be utilized for various purposes. Currently, it is mainly used to identify browsers that have no stateful identifiers, for example, the identifier in a cookie is cleared. A browser fingerprint consists of a set of properties of the browser. Executing a fingerprinting script in the browser allows the tracker to access sensitive data such as browser settings and even operating system and hardware information. Since no information is stored on the browser side, it is entirely stateless and difficult to detect and block.

Mayer [35] first investigated whether remote servers could exploit differences in browser environments to identify users, and discovered that browsers could reveal the unique combination of operating systems, hardware, and browser configuration. Later, Eckerley [1] conducted the *Panopticlick* experiment, the first large-scale demonstration that "browser fingerprinting" is practical, with very strong privacy implications.

Moreover, as browsers become more feature-rich, the information covered by browser fingerprints continues to expand, and many research efforts have been devoted to adding new information to browser fingerprints. Researchers found that canvas [36], WebGL [37], the audio API [7], fonts [38], the battery API [39], CSS properties [40], [41] and browser extensions [42]–[46] can be used to generate browser fingerprints. Even some studies have uncovered information about devices by benchmarking their CPU and GPU capabilities through JavaScript [47]–[51].

Browser technology is developing all the time, and browser fingerprinting is also evolving with it, and sometimes new browser technology is even helpful in enhancing the ability of browser fingerprinting to identify browsers. For example, new browser features designed to enhance the user experience, such as canvas [52] and WebGL [53], have proven to be used to generate more aggressive fingerprints, making it more difficult to detect fingerprinting.

B. Browser Fingerprinting Detection

In 2013, Nikiforakis et al. [5] examined scripts from Alexa top 10,000 sites and revealed the vulnerability of the browser

ecosystem against fingerprinting. Not long after that Acar et al. [6] carried out a large crawl, finding 549 of the 1 million sites performing fingerprinting. Since then, several studies have been conducted to measure fingerprinting adoption on the Web [7], [54], [55], which use different methods to identify fingerprinting scripts. While collecting information in the browser is straightforward, how to identify fingerprinting scripts is much more complicated. The difference between benign and fingerprinting scripts is unclear. Different fingerprinting detection techniques can lead to very different fingerprint counts, which fuels the demand for learning-based solutions.

Prior studies have explored learning-based tracking detection. Ikram et al. [56] extracted syntactic and semantic features from JavaScript source code, which are leveraged to train a one-class classifier to detect tracking scripts. Wu et al. [57] extracted features from execution trace of scripts through Web API method calls. Iqbal et al. [58] presented a graph-based machine-learning method to detect advertisements and trackers. Amjad et al. [59] proposed *TrackerSift*, which analyses domain, hostname, script, and method to reveal tracking under the cover of functional web resources. Iqbal et al. [60] proposed *KHALEESI*, which detects advertising and tracking request chains leveraging the essential sequential context.

Some approaches were proposed specifically for fingerprinting detection. Rizzo et al. [61] detected fingerprinting scripts based on features extracted from specific APIs and the use of machine learning classifiers. Iqbal et al. [8] utilized features extracted through the AST and execution trace of scripts to train a machine learning classifier. Bird et al. [62] proposed a semi-supervised machine learning approach with the central idea that fingerprinting scripts have similar API access patterns when generating their fingerprints. Sjösten et al. [63] looked for the essence of fingerprinting by analyzing the pattern of gathering information from browser APIs and communicating the information to the network. Rizzo et al. [64] designed an approach to detect fingerprinting script providers based on static or dynamic analysis of JavaScript code and machine learning. Durey et al. [65] proposed a technique that relies on both automatic and manual decisions to identify browser fingerprinting scripts. Ngan et al. [66] investigated the robustness of a fingerprinting detection approach, *FP-Inspector*, against obfuscated fingerprinting scripts.

Different from these approaches, *DeepFPD* utilizes token, AST, and CFG of the code simultaneously, leveraging attention mechanism at the network level to aggregate them. This allows our approach to fully utilize the information of the code and thus detect browser fingerprinting more effectively.

C. Deep Learning for Code Representation

Many studies have been proposed to utilize deep learning for better code representation. Nguyen et al. [67] proposed the deep neural network model *Dnn4C* to map lexical sequences, syntactic symbol sequences, and type conversion sequences into hidden vectors and fuse them into a representation of the source code. Mou et al. [17] designed tree-structured convolutional neural networks to capture features from ASTs of programs for source code processing. Zhang et al. [15] and Ben-Nun et al. [67] used the AST of the code and the intermediate language to perform the statement-level vector representation of the code. Alon et al. proposed models such as *Code2Vec* [9], [68] and *Code2Seq* [16] based on encoder-decoder to partition and represent ASTs of code as vectors. Allamanis et al. [69] applied graph neural networks to capture syntactic and semantic features from the AST of the source code. Sui et al. [70], [71] proposed to preserve interprocedural program dependence for better code embedding.

Utilizing deep learning for code representation has also made great strides in different tasks related to code. White et al. [12] used RNNs to extract the lexical and structural information of the code, which greatly improved the performance of traditional methods in the tasks of code recommendation and code clone detection. Dam et al. [72] proposed *DeepSoft* based on LSTM for learning long-term dependencies in software modeling for code recommendation. Zhao et al. [19] designed a deep learning model to measure code similarity. They encoded code control flow and data flow information and leveraged deep learning models to learn patterns of these hidden representations. Wan et al. [13] presented a multi-modal attention network to represent tokens, AST, and CFG of the code for semantic source code retrieval. Cheng et al. [73] proposed a novel approach by utilizing a graph convolutional network to embed code fragments into a representaion that preserves control-flow information for vulnerability detection. Further, some methods [74], [75] are presented to embed feasible value-flow paths to detect vulnerabilities.

VII. Conclusion

In this paper, we propose *DeepFPD*, a deep learning-based detection method for browser fingerprinting. In *DeepFPD*, three different code representations, including tokens, ASTs, and CFGs, are fused to obtain comprehensive code semantics. In addition, the attention mechanism is introduced to enhance the effectiveness of *DeepFPD*. To evaluate *DeepFPD*, a training dataset is constructed with 1,272 fingerprinting scripts and 9,534 non-fingerprinting scripts, and then *DeepFPD* is experimentally evaluated on the training dataset and a test dataset of 2,082 samples constructed based on manual analysis. The experimental results show that *DeepFPD* is effective in detecting browser fingerprinting and outperforms the state-of-the-art work, with an 8.3% increase in F1 and a 6.4% decrease in FNR on our *training dataset*, and a 21.3% increase in F1 and an 8.3% decrease in FNR on our *test dataset*, respectively, and also significantly outperforms other machine learning-based methods.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the National Natural Science Foundation of China (Grant No. 62272181).

References

[1] P. Eckersley, "How unique is your web browser?" in *Proceedings of the 10th International Symposium on Privacy Enhancing Technologies (PETS 2010)*, 2010, pp. 1–18.

[2] P. Laperdrix, N. Bielova, B. Baudry, and G. Avoine, "Browser fingerprinting: A survey," *ACM Transactions on the Web*, vol. 14, no. 2, pp. 1–33, 2020.

[3] N. J. Rubenking, "You tossed your cookies but they're still tracking you; here's how to hide your browser fingerprint," https://www.pcmag.com/how-to/you-tossed-your-cookies-but-theyre-still-tracking-you-heres-how-to-hide, 2022.

[4] W3C, "Mitigating browser fingerprinting in web specifications," https://www.w3.org/TR/fingerprinting-guidance, 2019.

[5] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna, "Cookieless monster: Exploring the ecosystem of web-based device fingerprinting," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP 2013)*, 2013, pp. 541–555.

[6] G. Acar, C. Eubank, S. Englehardt, M. Juarez, A. Narayanan, and C. Diaz, "The web never forgets: Persistent tracking mechanisms in the wild," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS 2014)*, 2014, pp. 674–689.

[7] S. Englehardt and A. Narayanan, "Online tracking: A 1-million-site measurement and analysis," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS 2016)*, 2016, pp. 1388–1401.

[8] U. Iqbal, S. Englehardt, and Z. Shafiq, "Fingerprinting the fingerprinters: Learning to detect browser fingerprinting behaviors," in *Proceedings of the 42nd IEEE Symposium on Security and Privacy (SP 2021)*, 2021, pp. 1143–1161.

[9] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," in *Proceedings of the 46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2019)*, 2019, pp. 1–29.

[10] S. V. Nguyen, T. N. Nguyen, Y. Li, and S. Wang, "Combining program analysis and statistical language model for code statement completion," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, 2019, pp. 710–721.

[11] O. Karnalim and Simon, "Syntax trees and information retrieval to improve code similarity detection," in *Proceedings of the Twenty-Second Australasian Computing Education Conference (ACE 2020)*, 2020, pp. 48–55.

[12] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2019)*, 2019, pp. 479–490.

[13] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. S. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, 2019, pp. 13–25.

[14] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. T. Devanbu, "On the naturalness of software," in *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, 2012, pp. 837–847.

[15] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering (ICSE 2019)*, 2019, pp. 783–794.

[16] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.

[17] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth Conference on Artificial Intelligence (AAAI 2016)*, 2016, pp. 1287–1293.

[18] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. S. Yu, "Multi-modal attention network learning for semantic source code retrieval," in *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*, 2019, pp. 13–25.

[19] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/SIGSOFT 2018) (FSE 2018)*, 2018, pp. 141–151.

[20] jQuery, "esprima," https://github.com/jquery/esprima, 2022.

[21] mariusschulz, "styx," https://github.com/mariusschulz/styx, 2022.

[22] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *CoRR*, 2015.

[23] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, 1997.

[24] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[25] W. Cao, C. Zheng, Z. Yan, and W. Xie, "Geometric deep learning: progress, applications and challenges," *Science China Information Sciences*, vol. 65, no. 2, pp. 1–3, 2022.

[26] Y. Li, D. Tarlow, M. Brockschmidt, and R. S. Zemel, "Gated graph sequence neural networks," in *Proceedings of the 4th International Conference on Learning Representations (ICLR 2016)*, 2016.

[27] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proceedings of the 3rd International Conference on Learning Representations (ICLR 2015)*, 2015.

[28] M. Guo, T. Xu, J. Liu, Z. Liu, P. Jiang, T. Mu, S. Zhang, R. R. Martin, M. Cheng, and S. Hu, "Attention mechanisms in computer vision: A survey," *CoRR*, 2021.

[29] D. Hu, "An introductory survey on attention mechanisms in NLP problems," in *Proceedings of the 2019 Intelligent Systems Conference (IntelliSys 2019)*, 2019, pp. 432–448.

[30] J. B. Lee, R. A. Rossi, S. Kim, N. K. Ahmed, and E. Koh, "Attention models in graphs: A survey," *ACM Transactions on Knowledge Discovery from Data*, vol. 13, no. 6, 2019.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 5998–6008. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

[32] openwpm, "Openwpm," https://github.com/openwpm/OpenWPM, 2022.

[33] S. Haanen, "Detection of browser fingerprinting by static javascript code classification," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:44184792

[34] "static-javascript-fingerprint-classification," [n.d,]. [Online]. Available: https://github.com/Timvanz/static-javascript-fingerprint-classification

[35] J. R. Mayer, ""any person... a pamphleteer": Internet anonymity in the age of web 2.0," *Undergraduate Senior Thesis, Princeton University*, 2009.

[36] K. Mowery and H. Shacham, "Pixel perfect: Fingerprinting canvas in html5," in *Proceedings of W2SP (W2SP 2012)*, 2012.

[37] Y. Cao, S. Li, and E. Wijmans, "(cross-)browser fingerprinting via OS and hardware level features," in *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS 2017)*, 2017.

[38] D. Fifield and S. Egelman, "Fingerprinting web users through font metrics," in *Proceedings of the 19th International Conference on Financial Cryptography and Data Security (FC 2015)*, 2015, pp. 107–124.

[39] L. Olejnik, G. Acar, C. Castelluccia, and C. Díaz, "The leaking battery - A privacy analysis of the HTML5 battery status API," in *Proceedings of Data Privacy Management, and Security Assurance - 10th International Workshop (DPM 2015), and 4th International Workshop (QASA 2015)*, 2015, pp. 254–263.

[40] T. Unger, M. Mulazzani, D. Fruhwirt, M. Huber, S. Schrittwieser, and E. R. Weippl, "SHPF: enhancing HTTP(S) session security with browser fingerprinting," in *Proceedings of the 2013 International Conference on Availability, Reliability and Security (ARES 2013)*, 2013, pp. 255–261.

[41] N. Takei, T. Saito, K. Takasu, and T. Yamada, "Web browser fingerprinting using only cascading style sheets," in *Proceedings of 10th International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2015*, 2015, pp. 57–63.

[42] A. Sjösten, S. Van Acker, and A. Sabelfeld, "Discovering browser extensions via web accessible resources," in *Proceedings of the Seventh ACM Conference on Data and Application Security and Privacy (CODASPY 2017)*, 2017, pp. 329–336.

[43] O. Starov and N. Nikiforakis, "XHOUND: quantifying the fingerprintability of browser extensions," in *Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP 2017)*, 2017, pp. 941–956.

[44] I. Sánchez-Rola, I. Santos, and D. Balzarotti, "Extension breakdown: Security analysis of browsers extension resources control policies," in *Proceedings of the 26th USENIX Security Symposium, (USENIX Security 2017)*, 2017, pp. 679–694.

[45] G. G. Gulyás, D. F. Somé, N. Bielova, and C. Castelluccia, "To extend or not to extend: On the uniqueness of browser extensions and web logins," in *Proceedings of the 2018 Workshop on Privacy in the Electronic Society (WPES@CCS 2018)*, 2018, pp. 14–27.

[46] P. Laperdrix, O. Starov, Q. Chen, A. Kapravelos, and N. Nikiforakis, "Fingerprinting in style: Detecting browser extensions via injected

style sheets," in *Proceedings of the 30th USENIX Security Symposium (USENIX Security 2021)*, 2021, pp. 2507–2524.

[47] K. Mowery, D. Bogenreif, S. Yilek, and H. Shacham, "Fingerprinting information in javascript implementations," in *Proceedings of W2SP (W2SP 2011)*, 2011.

[48] G. Nakibly, G. Shelef, and S. Yudilevich, "Hardware fingerprinting using HTML5," *CoRR*, 2015.

[49] T. Saito, K. Yasuda, T. Ishikawa, R. Hosoi, K. Takahashi, Y. Chen, and M. Zalasinski, "Estimating CPU features by browser fingerprinting," in *Proceedings of 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2016)*, 2016, pp. 587–592.

[50] T. Saito, K. Yasuda, K. Tanabe, and K. Takahashi, "Web browser tampering: Inspecting CPU features from side-channel information," in *Proceedings of the 12th International Conference on Broad-Band Wireless Computing, Communication and Applications (BWCCA 2017)*, 2017, pp. 392–403.

[51] I. Sánchez-Rola, I. Santos, and D. Balzarotti, "Clock around the clock: Time-based device fingerprinting," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS 2018)*, 2018, pp. 1502–1514.

[52] M. W. Docs, "Canvas api," https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API, 2022.

[53] ——, "Webgl," https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API, 2022.

[54] G. Acar, M. Juárez, N. Nikiforakis, C. Díaz, S. F. Gürses, F. Piessens, and B. Preneel, "Fpdetective: dusting the web for fingerprinters," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS 2013)*, 2013, pp. 1129–1140.

[55] N. M. Al-Fannah, W. Li, and C. J. Mitchell, "Beyond cookie monster amnesia: Real world persistent online tracking," in *Proceedings of Information Security - 21st International Conference (ISC 2018)*, 2018, pp. 481–501.

[56] M. Ikram, H. J. Asghar, M. A. Kâafar, A. Mahanti, and B. Krishnamurthy, "Towards seamless tracking-free web: Improved detection of trackers via one-class learning," in *Proceedings of the 17th Privacy Enhancing Technologies Symposium (PETS 2017)*, 2017, pp. 79–99.

[57] Q. Wu, Q. Liu, Y. Zhang, P. Liu, and G. Wen, "A machine learning approach for detecting third-party trackers on the web," in *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016)*, 2016, pp. 238–258.

[58] U. Iqbal, P. Snyder, S. Zhu, B. Livshits, Z. Qian, and Z. Shafiq, "Adgraph: A graph-based approach to ad and tracker blocking," in *Proceedings of the 2020 IEEE Symposium on Security and Privacy (SP 2020)*, 2020, pp. 763–776.

[59] A. H. Amjad, D. Saleem, M. A. Gulzar, Z. Shafiq, and F. Zaffar, "Trackersift: Untangling mixed tracking and functional web resources," in *Proceedings of the 21st ACM Internet Measurement Conference*, 2021, pp. 569–576.

[60] U. Iqbal, C. Wolfe, C. Nguyen, S. Englehardt, and Z. Shafiq, "Khaleesi: Breaker of advertising and tracking request chains," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2911–2928.

[61] V. Rizzo, "Machine learning approaches for automatic detection of web fingerprinting," Ph.D. dissertation, Politecnico di Torino, 2018.

[62] S. Bird, V. Mishra, S. Englehardt, R. Willoughby, D. Zeber, W. Rudametkin, and M. Lopatka, "Actions speak louder than words: Semi-supervised learning for browser fingerprinting detection," *arXiv preprint arXiv:2003.04463*, 2020.

[63] A. Sjösten, D. Hedin, and A. Sabelfeld, "Essentialfp: Exposing the essence of browser fingerprinting," in *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2021, pp. 32–48.

[64] V. Rizzo, S. Traverso, and M. Mellia, "Unveiling web fingerprinting in the wild via code mining and machine learning." *Proc. Priv. Enhancing Technol.*, vol. 2021, no. 1, pp. 43–63, 2021.

[65] A. Durey, P. Laperdrix, W. Rudametkin, and R. Rouvoy, "An iterative technique to identify browser fingerprinting scripts," *arXiv preprint arXiv:2103.00590*, 2021.

[66] R. Ngan, S. Konkimalla, and Z. Shafiq, "Nowhere to hide: Detecting obfuscated fingerprinting scripts," *arXiv preprint arXiv:2206.13599*, 2022.

[67] A. T. Nguyen, T. D. Nguyen, H. D. Phan, and T. N. Nguyen, "A deep neural network language model with contexts for source code," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)*, 2018, pp. 323–334.

[68] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018.

[69] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *Proceedings of the 6th International Conference on Learning Representations (ICLR 2018)*, 2018.

[70] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: value-flow-based precise code embedding," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, pp. 233:1–233:27, 2020. [Online]. Available: https://doi.org/10.1145/3428301

[71] Y. Sui and J. Xue, "Value-flow-based demand-driven pointer analysis for C and C++," *IEEE Trans. Software Eng.*, vol. 46, no. 8, pp. 812–835, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2869336

[72] H. K. Dam, T. Tran, J. C. Grundy, and A. K. Ghose, "Deepsoft: a vision for a deep model of software," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*, 2016, pp. 944–947.

[73] X. Cheng, H. Wang, J. Hua, M. Zhang, G. Xu, L. Yi, and Y. Sui, "Static detection of control-flow-related vulnerabilities using graph embedding," in *24th International Conference on Engineering of Complex Computer Systems, ICECCS 2019, Guangzhou, China, November 10-13, 2019*, J. Pang and J. Sun, Eds. IEEE, 2019, pp. 41–50. [Online]. Available: https://doi.org/10.1109/ICECCS.2019.00012

[74] X. Cheng, G. Zhang, H. Wang, and Y. Sui, "Path-sensitive code embedding via contrastive learning for software vulnerability detection," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 519–531. [Online]. Available: https://doi.org/10.1145/3533767.3534371

[75] X. Cheng, X. Nie, N. Li, H. Wang, Z. Zheng, and Y. Sui, "How about bug-triggering paths? - understanding and characterizing learning-based vulnerability detectors," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–18, 2022.