Towards Efficient GNN-Based Phishing Detection from HTML Source Code

Warre Hofmans[1], Wei Wei[2], Simon Vanneste[1], and Kevin Mets[1]

[1] Faculty of Applied Engineering, University of Antwerp - imec,
IDLab – Department of Electronics and ICT,
Sint-Pietersvliet 7, 2000 Antwerp, Belgium
[2] Faculty of Science, University of Antwerp - imec,
IDLab – Department of Computer Science,
Sint-Pietersvliet 7, 2000 Antwerp, Belgium
Warre.Hofmans@student.uantwerpen.be
{Wei.Wei,Kevin.Mets,Simon.Vanneste}@uantwerpen.be

Abstract. Phishing websites are a common cyber fraud strategy used to deceive users into disclosing personal or sensitive information by impersonating legitimate websites. Such attacks often have long identification times and are accompanied by high costs. These types of attacks have been a cyberthreat for a long time, but are occurring more frequently, becoming more sophisticated and accessible with the introduction of generative AI tools. Although previous research has achieved great success in detecting phishing websites, most of the earlier techniques are becoming obsolete with the latest advances in the phishing scene, as the pages are increasing in quality. This paper introduces a GNN-based approach to detecting phishing pages by identifying irregularities in their HTML source code, such as poor semantics, the presence of malicious code, or the use of phishing kits. An HTML-reduction algorithm is introduced to a) reduce structural noise and b) lower the computational costs. Using a simple node feature extraction process and a reduction algorithm yields a computationally efficient model, achieving 95.57% F1. The HTML DOM tree-based approach was validated additionally by a) an in-depth dataset analysis, showing a clear difference in benign and phishing source code, and b) traditional machine learning models (Random Forest and XGBoost) achieving up to 96.00% F1 using manually extracted graph features.

Keywords: Phishing detection · GNN.

1 Introduction

Phishing attacks remain a serious cyberthreat for both individuals and enterprises and are performed using a wide variety of techniques [4]. One of the most common strategies is phishing websites, aiming to deceive users into disclosing sensitive information, often impersonating legitimate websites. This technique has been posing a threat for a long time and is becoming more frequent than ever. APWG [6] reports having detected 1,003,924 unique phishing websites in the first quarter of 2025, being the most since Q4 in 2023. Phishing attacks can leave a huge impact on organisations; IBM [27] reports an average annual cost of 4.8M USD and an average identification time of 192 days for 2025.

Attackers are leveraging generative AI tools to increase website quality and significantly lower development times. Begou et al. [12] report an average time of 4 minutes for the generation of a phishing website using ChatGPT. The process outputs a high-quality phishing page with high similarity to the original webpage and requires little technical skill.

Phishing website detection has been extensively researched [48] before but is more relevant than ever, with an increasing amount of phishing pages having higher quality and a shorter development time. Current detection methods are divided into three categories by [33]:

Signature-based methods [49,31,66] rely on identifying common elements or patterns in phishing pages. Pages are compared against predefined signatures based on known phishing websites, phishing kits, or the certificate of the website. These techniques can be effective for widely used phishing attacks but lack protection against new and evolving strategies. These methods require the maintenance of a reference database.

Machine learning-based methods [50,10,51,3] often leverage ML to extract features from the websites to determine their authenticity, removing the need for a reference database. These features are manually selected based on the URL or content of the website. They offer better adaptability against the evolving phishing attacks.

Deep learning-based methods [36,55,24,5,54,30] use automatic extraction of relevant features from the data. These techniques can eliminate the need for manual feature engineering and improve the discriminability of classes by mapping features into a higher-level latent space.

With phishing pages becoming more sophisticated, some of these previous detection methods have become vulnerable. Varshney et al. [56] advise against overly relying on domain or certificate features, as most of the recent phishing is deployed over HTTPS. Techniques based on low-quality visual similarity or textual features become weaker as visual similarity and text quality increase using generative AI tools. With current technical trends, these tools will only improve and therefore also weaponise phishing attacks even more.

This work proposes a phishing detection technique that does not rely on visual, text, or domain features. We show how machine learning (ML) models and graph neural networks (GNNs) can leverage features extracted from the tree-structured source code to detect phishing attacks. GNNs are a class of neural networks designed to operate on graph-structured data. They employ a message-passing mechanism to iteratively aggregate and transform information from neighboring nodes, effectively capturing the complex structure of the graph.

When a phishing page is created, malicious code is either generated by the AI tools used or manually injected by the malicious party. As a result, the HTML source code will always contain traces of phishing websites, unlike the earlier-mentioned features that can be imitated to almost perfection. Source code analysis can detect phishing pages based on the following indicators [33]: poor semantics of HTML code, code with malicious intent (e.g., hidden fields, deceptive redirects), use of phishing kits, or common patterns across phishing pages. A thorough analysis of the used dataset showed a difference in used HTML elements, attributes, and website size, motivating our code-based approach.

Finally, to achieve a fast detection and training time, we limit the computational costs by using only simple node features, and we introduce an HTML-reduction algorithm. For each website, a graph is constructed based on the HTML tree structure, with node features based on the tag name, attributes, and presence of text for each node in the tree. To reduce the size of the graphs, similar nodes and structurally insignificant nodes are pruned from the graph.

Main contributions:

With the proposed method, we make the following main contributions: a) A graph construction algorithm converting the source code of a website into a graph representation fit for machine learning, using lightweight and scalable node features; b) A GraphSAGE model leveraging the semantics of the HTML, avoiding reliance on textual, visual, or domain-level features; c) A HTML reduction algorithm to reduce the size of the graphs by pruning redundant and structurally insignificant nodes, resulting in a 16.25% saving in GPU training cost.

2 Related Work

In recent years, GNN-enabled approaches have shown promising results, achieving up to 99.86% accuracy by Wang et al. [62] on their custom dataset. They are used for their ability to model the complex relationships of websites. A great part of the web is inherently structured as graphs. By leveraging GNNs, these inherent structures can be maintained while capturing relevant information for phishing detection. These can be divided into three categories:

URL-based methods construct graphs using the URLs of websites or those linked to by them. Phishing websites are identified solely by URL-based features, without consideration of the content of the pages. In [61], a graph is created from websites with edges representing similarity scores and using message passing to infer the labels of unknown websites. Despite being proven to be effective with a 94.90% F1 score — a measure that balances precision and recall — on their custom dataset, based on data from PhishTank [44], it lacks zero-day protection and requires the maintenance of a dataset after training. PhishGNN [13] constructs graphs based on the extracted URLs from a website, being the child nodes, and uses message passing to

infer the label of the website, the root node. They report an accuracy of 99.70% on their custom dataset, constructed using URLs from PhishTank [44] and OpenPhish [40]. However, the method could be bypassed by adding a large number of benign URLs to a phishing website.

Content-based methods use the content of the pages, the source code, or a screenshot of the website to create graphs. In [41], the tree structure of HTML is leveraged to create a graph for each website using an RNN to extract local features and a GNN to model the HTML semantics. The method was validated on a benign dominant dataset with examples collected from PhishTank [44], OpenPhish [40], and TrancoTop1M [45], accomplishing 86.34% F1 despite achieving 95.50% accuracy.

In [62], a single text graph is constructed for all websites in the dataset, using phrase nodes, to better capture the syntactic and semantic meaning of source code and document nodes. This technique attained 99.86% F1 but is limited to a transductive setting, whereas real-world application typically involve an inductive setting. The method was validated on a dataset comprising samples from the 2017 China Network Security Technology Challenge [39], Malware Domain List [38], Chinaz [19] and VirusTotal [58]. Instead of the source code, Lindamulage et al. [37] generated graphs based on screenshots of the websites; nodes are local image patches, and edges connect neighboring patches. They report an accuracy of 97.4% on the VisualPhish dataset [2].

Hybrid methods make use of both the URLs and content of the websites, typically by creating an ensemble model. Ariyadasa et al. [8] introduced PhishDet, consisting of two components, HTMLDet and URLDet, reporting an F1 of 96.42% on the dataset [7]. HTMLDet is a GCN model with graphs maintaining the inherent HTML structure and node embeddings being generated using a doc2vec applied to the HTML elements and their attributes. URLDet is an LRCN model with vectorized URLs as input. HTMLDet and URLDet individually scored F1 of 89.71% and 94.77% on the dataset [7].

Yoon et al. [65] also use an ensemble model, where a transformer combines both character-and word-level models processing the website URL and a GCN model using the HTML. Even though it accomplishes 97.09% F1 with data collected from Common Crawl [20] and Phishtank [44], it might not be appropriate for real-time use cases due to its high computational needs.

Shakir et al. [52] deploy a hybrid approach, using a GNN for feature selection, where graphs are constructed from pairwise distances between features, with the features found most important used to train an SVM model for the final prediction. Their approach achieved an accuracy of 93.52% and 93.78% F1, validated on data collected from OpenPhish [40], PhishTank [44], and other public blacklists, but relies on manually selected features. In [53], a graph per website is created using the HTML source code; domain information is injected by linking its embedding to the root node. Using a GAT model, they obtained up to 91% accuracy using datasets from KnowPhish [34] and Phishpedia [36]. Kavya et al. [29] introduced a Multimodal and Temporal Graph Fusion Framework that leverages textual, visual, and structural features of webpages while ensuring scalability and privacy by incorporating a series of techniques: GNNs, temporal modeling, and contrastive learning. They report achieving accuracy up to 98% and F1 up to 97%, using samples from OpenPhish [40], PhishTank [44], and the dataset from URLNet [32]. Remya et al. [46] combine the text, URL, and metadata of the website using an ensemble model consisting of a GNN, LightGBM, and BERT, reporting an accuracy and F1 of 97.3%, using the public dataset "Phishing Website URLs" sourced from Kaggle [1].

Our proposed method employs a GNN to model the HTML semantics, leveraging the tree structure of the HTML source code to construct the input graphs, similar to prior works [41,8]. However, we introduce a simpler node feature extraction process: Ouyang et al. employ an RNN to extract local features, and Ariyadasa et al. use a domain-specific doc2vec. In contrast, our approach does not rely on a deep learning model and does not require a learning algorithm.

3 Methodology

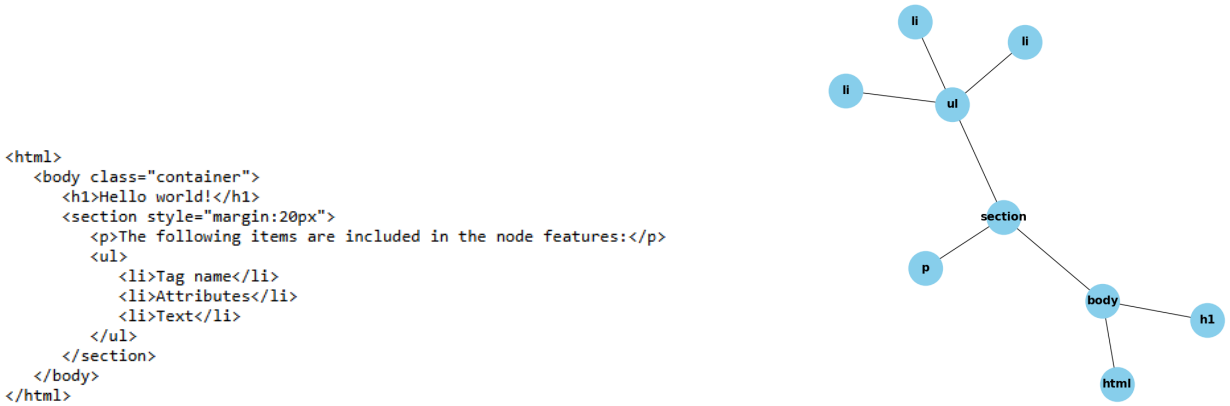
The HTML source code of each website is used to generate graphs fit for machine learning by leveraging the tree structure of HTML. A reduction algorithm is introduced to reduce the size of the graphs and remove structural noise to enhance the model's ability to learn relevant features and lower the computational costs. The effectiveness of the graph generation process is validated by traditional machine learning techniques before introducing the GraphSAGE model.

3.1 Graph Construction from HTML

For each website, a graph fit for machine learning is constructed based on the HTML source code. HTML [63] is structured as a tree, which is a special type of graph. This structure is directly used to construct the graphs; the key part is the encoding of the information of the DOM elements, consisting of a tag name, attributes, and optional text content. Maintaining the semantics of HTML source code is crucial, as its violation can be a clear indicator of phishing pages. Phishing pages often disregard the rules of semantic HTML, prioritising website appearance over underlying code quality. The tag name, attributes, and text content are vectorised into a node feature vector of dimension $\mathbb{R}^{1\times 299}$, capturing key information while minimising computational cost.

- **Tag name**: The name is one-hot encoded into an $\mathbb{R}^{1\times 125}$ vector, as there are currently 125 tag names [60]. Each vector contains zeros at all indexes, except at the one corresponding to the tag name of the node.
- **Attributes**: Attributes occur in key-value-pairs, with 173 possible standard keys [59]. The key *data-** introduces an unlimited amount of variation of custom attribute keys in the source code; therefore, they are all grouped together under one umbrella attribute. The keys are encoded to a vector of dimension $\mathbb{R}^{1\times 173}$. A 1 at an index indicates the presence of an attribute; in the case of multiple occurrences of the same attribute, the value at that index is increased by 1 for each appearance. Absent attributes have a zero at the corresponding index. Attribute values are not taken into account.
- **Text**: As a feature the presence/absence of text is recorded into a vector $\mathbb{R}^{1\times 1}$. By not considering the actual text content, the node vector can be kept minimal while still capturing semantic information, as in semantically correct HTML, text should only appear in leaf nodes.

The final feature vector for the node is the concatenation of the three mentioned vectors. Figure 1 displays the process for an example snippet of HTML.



(a) HTML Code (b) Graph Visualization

Fig. 1: Graph generation process from HTML code to graph-based tensor representation.

Table 1: Overview of all counted HTML elements and attributes.

Elements	a, div, li, span, p, td, img, br, ul, i, tr, strong, svg, h3, h2, b, sup, style, form, input, textarea, select, option, button, label
Attributes	class, href, id, data-*, title, style, rel, type, value, src, name, alt, content, target, onclick, required, http-equiv, method, action, autocomplete, placeholder

3.2 Baselines: Random Forest and XGBoost

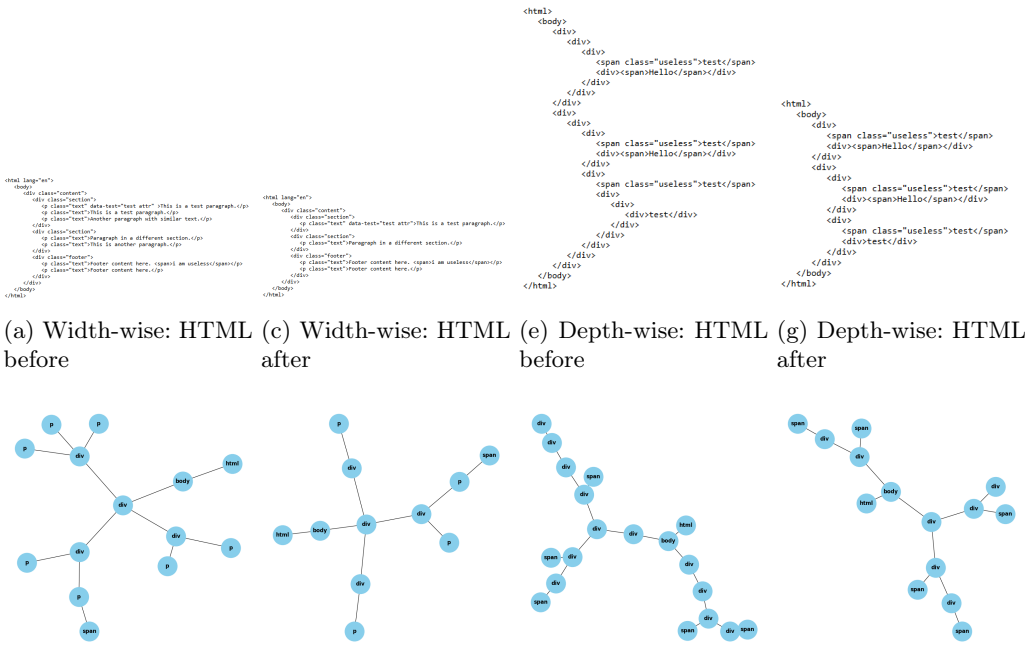
As a baseline, machine learning models – Random Forest [14] and XGBoost [17] – were chosen because they have proven to be strong learners while being significantly less complex than deep learning-based approaches. Features were manually extracted from the generated graphs, resulting in a tabular dataset with graph feature vectors of dimension $\mathbb{R}^{1\times 53}$. The selected features are divided into two categories:

- **Structural features**[3]: node count, degree, closeness, betweenness, diameter, spectral radius.
- **Content features**: normalised tag and attribute counts. For each graph, a selection of HTML elements and attributes was counted and normalised. This selection consists of the most frequently occurring tag names, attributes, and manually chosen entries. Table 1 shows the full list of tracked items.

3.3 GraphSAGE

A GraphSAGE [25] model is used to capture the local patterns and eliminate manual graph-feature engineering, it's chosen for its ability to generalise to unseen data and its scalability.

Each convolutional layer is followed by a GraphNorm [15], ReLU activation, and dropout layer, forming a basic block of the network. These blocks are connected in residual fashion to preserve information flow when going deeper and introduce flexibility. Graph representations are obtained using attentional aggregation [35], aggregating all node embeddings into a single graph embedding. The gate network learns scores for each node, allowing them to contribute differently to the final embedding, modelling the varying importance of HTML elements. For instance, a `<form>` element on a potential phishing website may be more insightful than a `<p>` element. The gate network consists of a linear layer followed by a ReLU activation, dropout, and a final linear layer. The graph representation is passed through a dropout layer before being passed through the classification head: a linear, ReLu, dropout, and final linear layer producing a single logit. The final binary prediction is obtained by applying a sigmoid function to the logit and comparing the output against a set threshold.



(a) Width-wise: HTML before (c) Width-wise: HTML after (e) Depth-wise: HTML before (g) Depth-wise: HTML after

(b) Width-wise: Graph before (d) Width-wise: Graph after (f) Depth-wise: Graph before (h) Depth-wise: Graph after

Fig. 2: Graph reduction: depth-wise vs width-wise.

[3] The structural features are inspired on [28]

3.4 Graph Reduction

A reduction algorithm is introduced to simplify the graphs. The graphs can be pruned in two ways:

- **Width-wise**: Similar leaf nodes are removed. Leaf nodes are deemed similar when representing the same HTML element and having a high cosine similarity, calculated on the attribute vectors. A threshold of 0.7 was used.
- **Depth-wise**: Nodes that add depth to the DOM tree while contributing little semantic information are pruned. A node is classified as *meaningless depth* if it satisfies all the following conditions:
  - It is a `<div>` or `<span>` element;
  - It contains fewer than 4 attributes;
  - It has only a single non-text child node.

To minimise information loss, the node feature vector is extended with two values, width-and depth-reduction-factor, to keep track of pruned nodes, resulting in a vector of dimension $\mathbb{R}^{1\times 301}$. Figure 2 shows an example for both width- and depth-wise reduction.

[64] and [18] defined the time complexity of the GraphSAGE training algorithm as:

$$\mathcal{O}(r^K \cdot n \cdot d^2) \tag{1}$$

where n = number of nodes, K = number of GNN layers, r = number of neighbors sampled per node and d = dimension of node hidden features (assumed constant for simplicity).

Theorem 1. *Let $T(n)$ be the training cost of the GraphSAGE algorithm given by $T(n) = O(r^K \cdot n \cdot d^2)$, If the number of nodes per graph is reduced by a factor R ($0 < R < 1$), then the training cost is reduced proportionally by the same factor, i.e., $T'(n) = O((1-R) \cdot r^K \cdot n \cdot d^2)$.*

Proof. By the definition of $T(n)$, replacing n by $(1-R) \cdot n$ yields $T'(n) = O(r^K \cdot (1-R) \cdot n \cdot d^2) = O((1-R) \cdot r^K \cdot n \cdot d^2)$. Hence, the cost decreases by a factor R.

Corollary 1. *In the absence of neighborhood sampling ($r = n$), the training cost is reduced with factor $(1-R)^{1+K}$. i.e., the savings grow exponentially with the number of layers K.*

Figure 3 demonstrates the practical impact of Theorem 1.

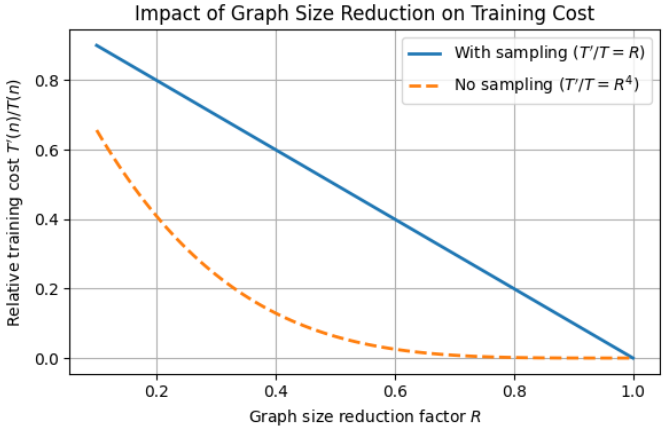


Fig. 3: Relative training cost $T'(n)/T(n)$ as a function of graph size reduction factor R for $K = 3$ layers, $d = 128$, and $r = 10$. The dashed curve shows the case without neighborhood sampling ($r = n$), where the reduction grows exponentially with K. The solid line shows the proportional case with sampling.

The graph reduction algorithms are directly integrated into the graph generation algorithm, increasing its time complexity from $\mathcal{O}(n)$ to $\mathcal{O}(n \cdot (m + d_e))$ (appendix B), with m the average node degree and d_e the depth of the DOM tree. However, the reduction still results in significant computational savings because:

- The graphs are only generated once, whereas the training algorithm is executed many times.
- Even for a single training epoch, it holds that $R \cdot r^K \cdot d^2 \gg m + d_e$, assuming typical values for r, K, and d. The used dataset has an average degree of 1.95 and depth of 13.41, assuming a small embedding size of 32 and knowing $r^K \geq 1$: $0.2152 \cdot 32^2 \cdot r^K \gg 15.36$, meaning the reduction yields a great performance boost.

A detailed description of the used reduction algorithms can be found in appendix B.

4 Experimental setting

The traditional machine learning models were trained on multiple subsets of the dataset with and without the reduction algorithm to showcase the effectiveness of the graph generation and reduction process. Different GraphSAGE models where trained to demonstrates the trade-off between complexity and performance, and the training cost savings by using the reduction algorithm.

All code was implemented in Python, including the used figures, using the following libraries: XGBoost [16], Scikit-learn [43], Pytorch [42], PyTorch Geometric [23] , Matplotlib [26], SciPy [57], igraph [21] and Beautiful Soup [47].

4.1 Dataset

The publicly available dataset [7] was used. The dataset was chosen for two main reasons: a) as it's used by a similar approach, HTMLDet [8], it enables fair comparison, and b) it contains both the URL and the HTML content of the website, removing the need to crawl the URLs. Benign examples were constructed based on Google keyword searches and another publicly available dataset [11]. The phishing websites originate from PhishTank [44], OpenPhish [40] and PhishRepo [9], collected between December 2020 and October 2021. Table 2 shows the distribution of the dataset after removing duplicate and invalid examples.

Table 2: Distribution of used dataset.

Class	Count	Fraction
Phishing	29,995	37.57%
Benign	49,852	62.43%
Overall	79,847	100%

As the tag names and attributes are primarily used for constructing the graphs, their distributions within the dataset were studied and some key observations can be made:

- Phishing pages contain more form-related elements and attributes.
- Phishing pages used attributes like *required* or *autocomplete* to entice website visitors into disclosing personal information.
- The attribute *http-equiv* appears more frequently on phishing websites, as it is often abused for malicious purposes.
- Phishing pages are significantly smaller than the benign websites.

Figures showing the most occurring HTML tag and attribute in the dataset and the size distributions of the graphs can be found in appendix A.

During the experiments, the dataset was split into a train and test set with a ratio of 4:1. The test set is exclusively used to evaluate the performance of the models. When training the GraphSAGE model, the train set was further split into training data and validation data with a ratio of 4:1, where the validation data is used for model selection.

4.2 Baselines: Random Forest and XGBoost

The models were trained on three subsets of the features: structural features, content features, and all features, both with and without applying the reduction algorithm. In all cases, a grid search was performed to find the optimal hyperparameters and the best threshold. The results can be found in table 3.

Without applying the reduction algorithm, using solely content features yielded the best results. However, these implicitly contain structural information as:

- Element counts are divided by the number of nodes in the graph.
- Attribute counts are divided by the total number of attributes on the website.

Thus, in both cases, larger websites will have smaller normalised element counts. Structural features added noise and reduced the model's performance, as they capture global information but fail to detect small malicious code pieces that are present in local information.

The performance on the structural features was notably improved by using the reduction algorithm, suggesting structural noise decreased by pruning structurally insignificant and redundant nodes. However, the models trained on the content features dropped in performance due to a loss of information. When decreasing the noise introduced by the structural features, it improves the model, as all features performed the best.

Table 3: Performance of Random Forest (RF) and XGBoost (XGB) on structural, content, and combined features with and without the graph reduction algorithm.

Feature Type	**Model**	**Not Reduced**				**Reduced**			
		Precision	Recall	F1	Acc	Precision	Recall	F1	Acc
Structural	RF	0.90	0.90	0.90	0.91	0.92	0.92	0.92	0.93
	XGB	0.87	0.87	0.87	0.88	0.86	0.88	0.87	0.88
Content	RF	0.96	0.95	0.96	0.96	0.90	0.90	0.90	0.91
	XGB	0.95	0.95	0.95	0.95	0.90	0.91	0.90	0.91
All	RF	0.96	0.95	0.95	0.96	0.93	0.93	0.93	0.93
	XGB	0.95	0.95	0.95	0.95	0.90	0.92	0.91	0.91

4.3 GraphSAGE

As shown in appendix A, the dataset contains very large graphs with examples containing over 50,000 nodes. As GNNs are computationally expensive, the reduction algorithm from section 3.4 is used to decrease the size of graphs.

Table 4 shows the results of the reduction algorithm. The average nodes per graph decreased by 21.25% and the coefficient of variation (CV) by 10.85%, making the graphs not only smaller but also more uniform in size.

Table 4: Nodes statistics before and after reduction

Metric	**Mean**	**Median**	**Std. Dev.**	**CV**
No reduction	553.12	255	1427.81	2.58
Reduction	441.13	186	1013.66	2.30
Factor	21.25%	27.06%	29.00%	10.85%

The model was trained using no neighbourhood sampling, as the introduced reduction algorithm decreased the training computational needs within reasonable bounds. It prevents further information loss and missing crucial nodes in the HTML graphs; a lot of nodes will not be relevant.

To increase generalisation of the model, graphs were injected with noise during training. Arbitrary HTML elements with attributes are added to the graphs, as it won't change the underlying goal of the website. A phishing website with an extra text paragraph or one more form element is still a phishing website. Noise was injected proportionally to the size of the graphs; the number of added nodes is for each graph equal to 0.2 x total number of nodes.

As optimizer Adam with learning rate 10^-3 was used in combination with class weighted cross entropy loss function, as the dataset is slightly imbalanced. It was found that the model tends to be overconfident; to counteract this, label smoothing was implemented. The reduction features in the node feature vector were normalised.

Graphs containing more than 50,000 or fewer than 10 nodes after reduction were dropped, leading to a more uniform size distribution. This helped the model to learn more complex patterns rather than a size difference in the webpages, as found in section 4.1. This subset contained 69,738 samples.

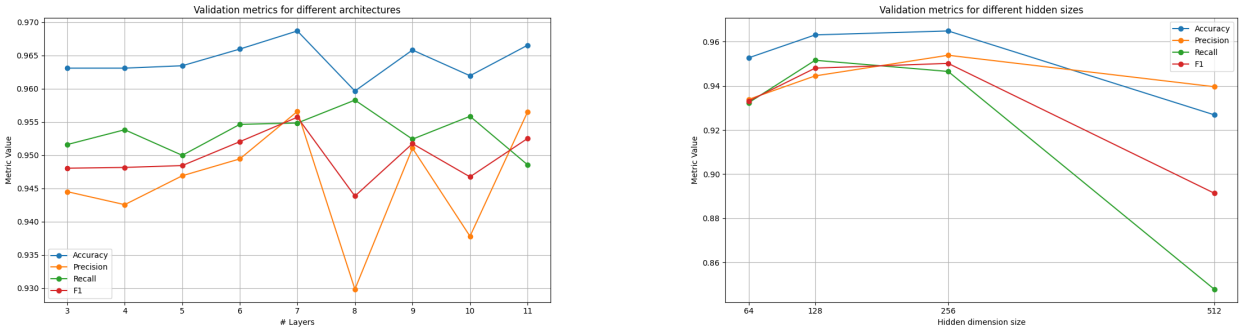
5 Results

Different versions of the model were trained, experimenting both with the depth and width of the network. Figure 4 demonstrates the trade-off between the complexity of the model and its performance. Experiments in figure 4a were executed with hidden dimension 128 and showed that smaller models achieved great results but were outperformed by more complex models. It was found that the performance stopped increasing after 7 layers.

Figure 4b shows the performance of models with different hidden dimensions; a three-layer model was used. It again shows that great results could be achieved with smaller models, but the best results were obtained using larger hidden dimensions. The performance stopped increasing for hidden dimensions larger than 256.

A seven-layer model with hidden dimension 128 was found to be the best model, achieving an F1 of 95.57%, while three-layer models performed slightly worse with an F1 up to 95.02% but had significantly lower computational costs. All results are found in table 5.

To study the impact of the reduction algorithm, a GraphSAGE model with three layers with a hidden dimension of 128 was trained with and without applying the reduction algorithm. For 10 epochs, the CPU and GPU times were captured, found in Table 6. Because the number of graphs (batches) remains constant, operations involving input/output and memory transfers remain largely unaffected by the size reduction. Additionally, the input size of both models differ (299 vs 301), while in section 3.4, it was assumed to be constant. As a result, there is a disparity between the theoretical saving of 40% according to Theorem 1 and the found saving of 16.25% in GPU training usage. The savings will become more significant as the models are scaled.



(a) Depth (b) Width

Fig. 4: Performance of different architectures.

Table 5: Comparison of GraphSAGE models with different depths and hidden sizes

Layers	Hidden Size	Precision	Recall	F1	Accuracy
3	128	0.9445	0.9515	0.9480	0.9630
3	256	0.9539	0.9465	0.9502	0.9649
7	128	**0.9566**	**0.9548**	**0.9557**	**0.9687**
7	256	0.9513	0.9471	0.9492	0.9641

Table 6: Training times: captured with PyTorch Profiler on GPU Quadro RTX 4000

	CPU	GPU
No reduction	394.65s	252.53s
Reduction	389.57s	211.50s
Savings	1.29%	16.25%

6 Discussion & Future work

Ariyadasa et al. [8] used the same dataset [7] but selected a subset of 50,000 examples and made it balanced. Our work used 69,738 examples with a small imbalance. While only using HTML, our model (95.57% F1) achieves similar performance to PhishDet (96.42% F1) and significantly outperforms the similar HTMLDet (89.71% F1) model.

To further increase the computational efficiency of the model, neighbourhood sampling could be introduced. By considering a fixed number of neighbours for each node, the computational costs can be significantly lowered [25]. This work focused on saving training cost by reducing GPU load, but the savings were overshadowed by the constant overhead. CPU optimisation should be done to decrease overall training costs, making the GPU savings more significant.

The used dataset is slightly outdated, with the oldest entries dating from 2020. The dataset was used because of the lack of benchmark datasets and the costs of constructing a dataset. The proposed method should be validated against a newer dataset, including the latest phishing trends. The recent phishing data should also be used to validate the zero-day protection of the model, as this has not been done yet. The model's behaviour during adversarial attacks should also be tested.

A comparison of the proposed GNN model with similar prior works should be conducted to evaluate its performance and robustness relative to existing methods. Furthermore, the impact of the reduction algorithm on model performance should be investigated by performing the same experiments without the reduction algorithm and by applying it to existing methods to compare the results.

7 Conclusion

This work introduced a simple yet effective algorithm for generating graphs fit for machine learning from HTML source code. The HTML-based approach was validated by baseline models, achieving up to 96.00% F1 and a GNN model attaining up to 95.57% F1. Raw HTML code introduced structural noise to those models; it was found that a) not all nodes are as relevant, and b) local structure is more important than the global structure.

An HTML reduction algorithm pruned both similar leaf nodes and nodes that added meaningless depth to the tree. The reduction resulted in a) up to 16.25% savings in GPU training cost and b) better utilisation of structural information by the models.

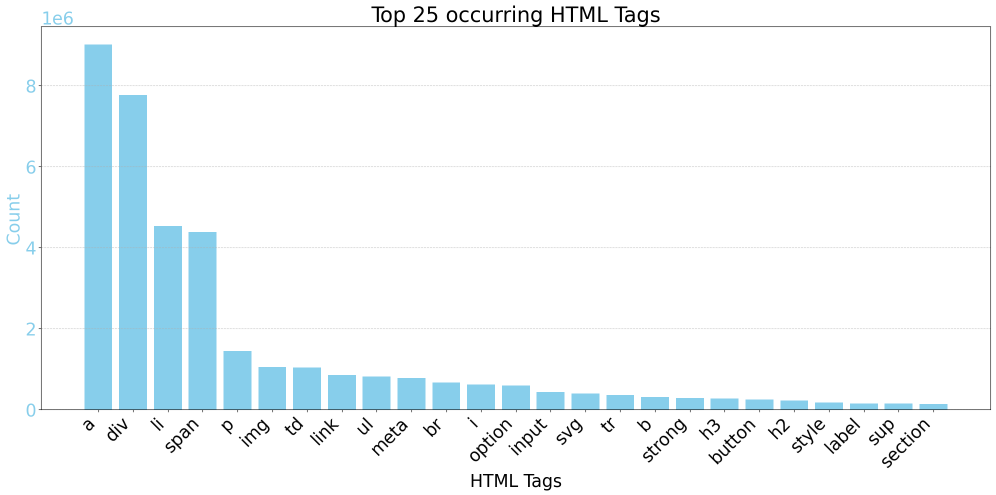
Random Forest and XGBoost were less complex and faster to train, but they require an additional preprocessing step to create graph-level features. The GraphSAGE model eliminates this step by directly leveraging the HTML structure. Therefore, it removes the need for manual feature engineering and captures local structure more effectively. Moreover, the learned representations can be used for a range of downstream tasks. Finally, the GraphSAGE model could be extended with continual learning methods [67,22] to stay up to date with evolving phishing trends.

8 Acknowledgments

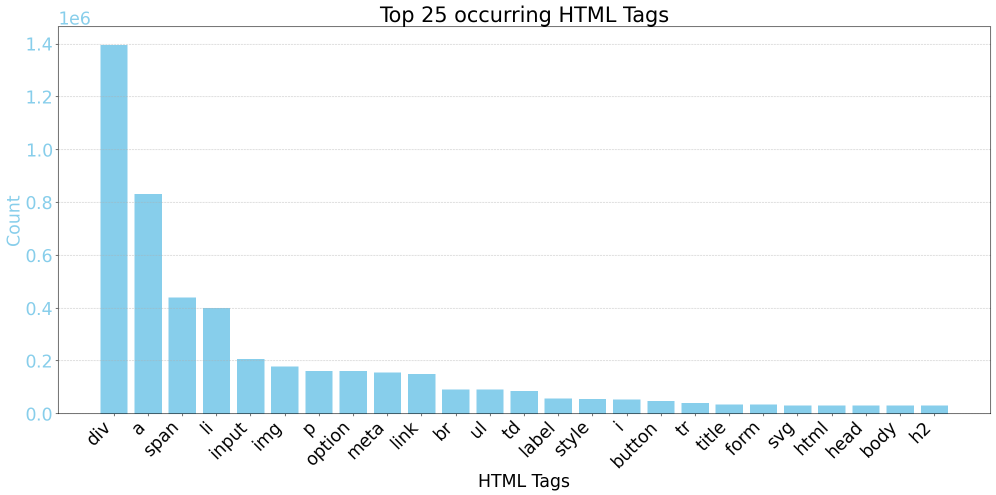
This work was completed as part of the Honours Programme at the Faculty of Applied Engineering, University of Antwerp.

A Appendix 1

This appendix contains the figures to motivate the claims in section 4.1. Figure 5 shows the most frequently occurring HTML elements in the dataset, and figure 6 highlights the most used attributes. Figure 7 shows a size comparison of the graphs generated from the websites in the dataset in terms of the number of nodes.

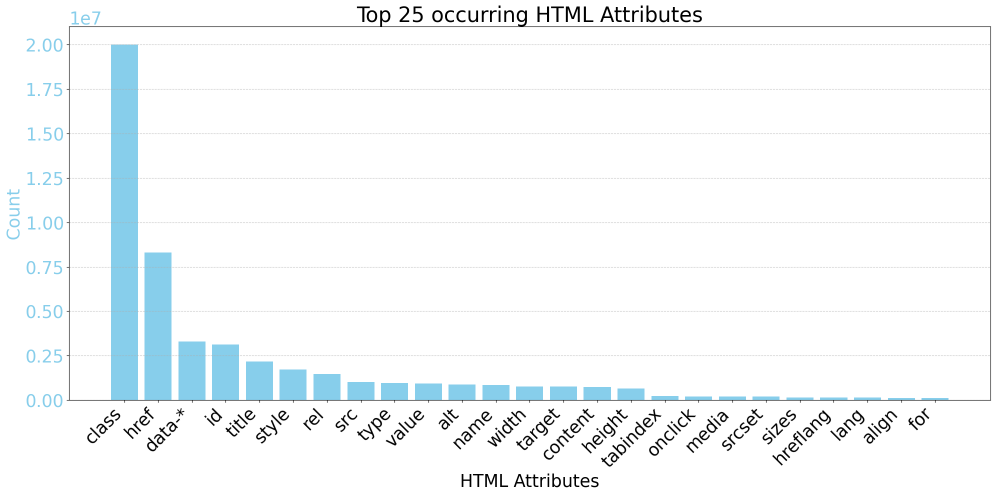


(a) Benign

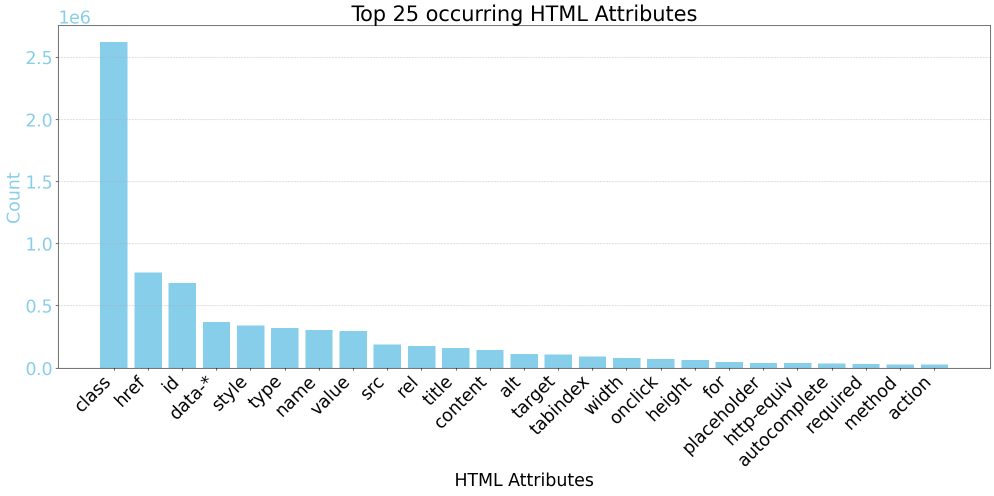


(b) Phishing

Fig. 5: HTML tag distribution within the dataset

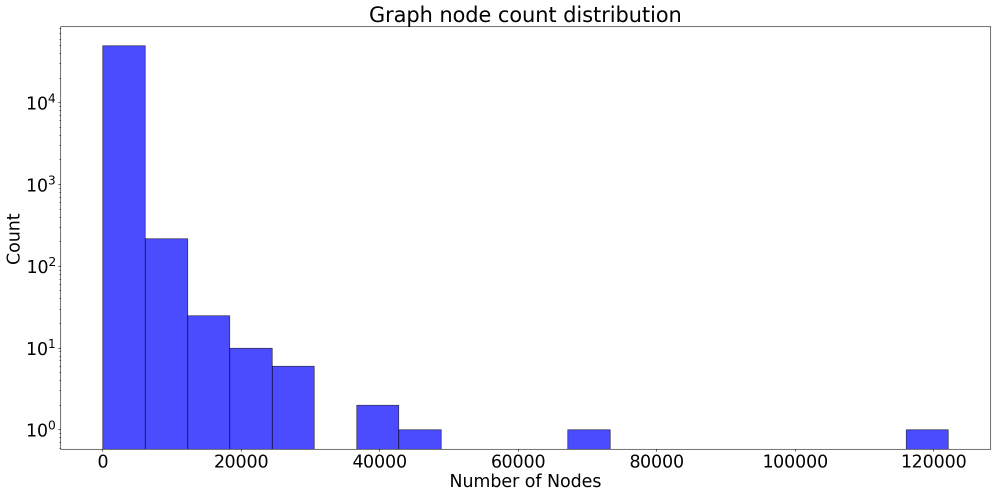


(a) Benign

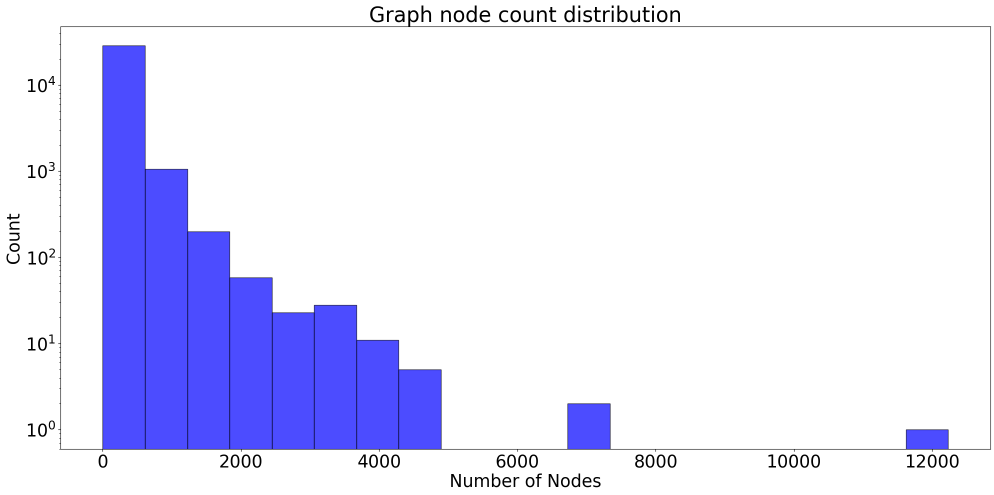


(b) Phishing

Fig. 6: HTM attribute distribution within the dataset



(a) Benign



(b) Phishing

Fig. 7: Size distribution within the dataset

B Appendix 2

This appendix showcases the used reduction and graph generation algorithms and discusses their time complexity.

1 shows the depth-wise reduction algorithm having time complexity $\mathcal{O}(d)$ with d = depth of the HTML tree. 2 shows the width-wise reduction algorithm having time complexity $\mathcal{O}(m \cdot (1 + 2a))$ with m =degree of a node and a = number of attributes of a node, which can be simplified to $\mathcal{O}(m)$ as $1 \gg 2a$, having found an average of 0.00031 attributes per node within the used dataset. Finally, 3 showcases the used graph generation algorithm with the reduction algorithm integrated, having a simplified time complexity $\mathcal{O}(n \cdot (m + d))$. 7 displays an overview of all algorithms and their simplified time complexities.

Table 7: Simplified Time Complexities of Preprocessing and Graph Construction Algorithms

Algorithm	**Time Complexity**
`ReduceHtmlDepth`	$\mathcal{O}(d)$
`ReduceHtmlWidth`	$\mathcal{O}(m)$
`GenerateGraph` (no reduction)	$\mathcal{O}(n)$
`GenerateGraph` (with reduction)	$\mathcal{O}(n \cdot (m + d))$

n: total number nodes, m: node degree, d: depth of the HTML tree.

Algorithm 1 Reduce HTML Depth

```
1: function ReduceHtmlDepth(parent, counter = 0)
2:     if MeaninglessDepth(parent) then
3:         Unwrap(child)
4:         return ReduceHtmlDepth(parent, counter + 1)        ▷ Counter keeps tracked of pruned nodes
5:     else
6:         return (parent, counter)
7:     end if
8: end function
```

Algorithm 2 Reduce HTML Width

```
function ReduceHtmlWidth(parent)
  Phase 1: Collect leaf nodes
    leafNodes ← empty list
    countMap ← empty map
    for all child in children of parent do
      countMap[child] ← 0
      if IsHardSkippable(child) then
        Decompose(child)
        continue
      end if
      if IsLeaf(child) then
        Append child to leafNodes
      end if
    end for
  Phase 2: Remove similar leaf nodes
    ptr ← 0
    while ptr < length(leafNodes) − 1 do
      if leafNodes[ptr].tag = leafNodes[ptr+1].tag then
        node1 ← TokenizeAttributes(leafNodes[ptr].attrs)
        node2 ← TokenizeAttributes(leafNodes[ptr+1].attrs)
        if CosineSimilarity(node1, node2) > 0.7 then
          countMap[leafNodes[ptr]] ← countMap[leafNodes[ptr]] + 1
          Decompose(leafNodes[ptr+1])
          Remove leafNodes[ptr+1] from list
        else
          ptr ← ptr + 1
        end if
      else
        ptr ← ptr + 1
      end if
    end while
    return (parent, countMap)
end function
```

Algorithm 3 Graph Construction with Optional Reductions

```
function GenerateGraph(parent, nodeFeatures, edgeSrc, edgeDst, parentId, depth = 0)
    nodeId ← parentId
Phase 1: width reduction
    if reduction is enabled then
        parent, widthMap ← ReduceHtmlWidth(parent)
    end if
    for all child in children of parent do
Phase 2: feature vector generation
        tagVec ← EncodeTag(child.tag)
        attrVec ← EncodeAttributes(child.attrs)
        hasText ← 1 if child contains text else 0
        if reduction is enabled then
            feature ← tagVec + attrVec + hasText + depth + widthMap[child]
        else
            feature ← tagVec + attrVec + hasText
        end if
        Append feature to nodeFeatures
        nodeId ← nodeId + 1
        Add edge from parentId to nodeId
Phase 3: depth reduction
        if reduction is enabled then
            child, newDepth ← ReduceHtmlDepth(child)
        else
            newDepth ← 0
        end if
        nodeId, nodeFeatures, edgeSrc, edgeDst ← GenerateGraph(child, nodeFeatures, edgeSrc, edgeDst,
nodeId, newDepth)
    end for
    return nodeId, nodeFeatures, edgeSrc, edgeDst
end function
```

References

1. Kaggle. https://www.kaggle.com
2. Abdelnabi, S., Krombholz, K., Fritz, M.: Visualphishnet: Zero-day phishing website detection by visual similarity. In: Proceedings of the 2020 ACM SIGSAC conference on computer and communications security. pp. 1681–1698 (2020)
3. Adap, V.A.V., Castillo, G.A., Delos Reyes, E.J.M., Ronquillo, E.B., Vea, L.A.: Do not feed the phish: Phishing website detection using url-based features. In: Proceedings of the 2023 5th World Symposium on Software Engineering. pp. 135–141 (2023)
4. Aleroud, A., Zhou, L.: Phishing environments, techniques, and countermeasures: A survey. Computers & Security **68**, 160–196 (2017)
5. Aljofey, A., Jiang, Q., Rasool, A., Chen, H., Liu, W., Qu, Q., Wang, Y.: An effective detection approach for phishing websites using url and html features. Scientific Reports **12**(1), 8842 (2022)
6. Anti-Phishing Working Group: Phishing activity trends report, 1st quarter 2025. Tech. rep., APWG (July 2025), https://docs.apwg.org/reports/apwg_trends_report_q1_2025.pdf, accessed: 2025-08-07
7. Ariyadasa, S., Fernando, S., Fernando, S.: Phishing websites dataset (2021). https://doi.org/10.17632/n96ncsr5g4.1, https://doi.org/10.17632/n96ncsr5g4.1
8. Ariyadasa, S., Fernando, S., Fernando, S.: Combining long-term recurrent convolutional and graph convolutional networks to detect phishing sites using url and html. IEEE Access **10**, 82355–82375 (2022)
9. Ariyadasa, S.N., Fernando, S., Fernando, S.: Phishrepo dataset (2022), https://data.mendeley.com/datasets/ttmmtsgbs8/4
10. Bahaghighat, M., Ghasemi, M., Ozen, F.: A high-accuracy phishing website detection method based on machine learning. Journal of Information Security and Applications **77**, 103553 (2023)
11. Bastem, E.: Phishing dataset (2017), https://github.com/ebubekirbbr/pdd/tree/master/input
12. Begou, N., Vinoy, J., Duda, A., Korczyński, M.: Exploring the dark side of ai: Advanced phishing attack design and deployment using chatgpt. In: 2023 IEEE Conference on Communications and Network Security (CNS). pp. 1–6. IEEE (2023)
13. Bilot, T., Geis, G., Hammi, B.: Phishgnn: A phishing website detection framework using graph neural networks. In: 19th International Conference on Security and Cryptography. pp. 428–435. SCITEPRESS-Science and Technology Publications (2022)
14. Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
15. Cai, T., Luo, S., Xu, K., He, D., Liu, T.Y., Wang, L.: Graphnorm: A principled approach to accelerating graph neural network training (2021), https://arxiv.org/abs/2009.03294
16. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. p. 785–794. KDD '16, ACM (Aug 2016). https://doi.org/10.1145/2939672.2939785, http://dx.doi.org/10.1145/2939672.2939785
17. Chen, T., He, T., Benesty, M., Khotilovich, V., Tang, Y., Cho, H., Chen, K., Mitchell, R., Cano, I., Zhou, T., et al.: Xgboost: extreme gradient boosting. R package version 0.4-2 **1**(4), 1–4 (2015)
18. Chiang, W.L., Liu, X., Si, S., Li, Y., Bengio, S., Hsieh, C.J.: Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In: Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining. pp. 257–266 (2019)
19. Chinaz: Top websites ranking. https://top.chinaz.com (2021)
20. Common Crawl: Common crawl—open repository of web crawl data. [Online]. Available: https://commoncrawl.org/
21. Csardi, G., Nepusz, T.: The igraph software package for complex network research. InterJournal **Complex Systems**, 1695 (2006), https://igraph.org
22. Febrinanto, F.G., Xia, F., Moore, K., Thapa, C., Aggarwal, C.: Graph lifelong learning: A survey (2022), https://arxiv.org/abs/2202.10688
23. Fey, M., Lenssen, J.E.: Fast graph representation learning with pytorch geometric (2019), https://arxiv.org/abs/1903.02428
24. Gopali, S., Namin, A.S., Abri, F., Jones, K.S.: The performance of sequential deep learning models in detecting phishing websites using contextual features of urls. In: Proceedings of the 39th ACM/SIGAPP Symposium on Applied Computing. pp. 1064–1066 (2024)
25. Hamilton, W., Ying, Z., Leskovec, J.: Inductive representation learning on large graphs. Advances in neural information processing systems **30** (2017)
26. Hunter, J.D.: Matplotlib: A 2d graphics environment. Computing in Science & Engineering **9**(3), 90–95 (2007). https://doi.org/10.1109/MCSE.2007.55
27. IBM Security and Ponemon Institute: Cost of a data breach report 2025. Tech. rep., IBM (July 2025), https://www.ibm.com/reports/data-breach, accessed: 2025-08-07

28. Islam, S., Hasan, M.N., Khanra, P.: A structural feature-based approach for comprehensive graph classification. Journal of Computational Science p. 102679 (2025)
29. Kavya, S., Sumathi, D.: Multimodal and temporal graph fusion framework for advanced phishing website detection. IEEE Access (2025)
30. Koide, T., Fukushi, N., Nakano, H., Chiba, D.: Detecting phishing sites using chatgpt. arXiv preprint arXiv:2306.05816 (2023)
31. Kondracki, B., Azad, B.A., Starov, O., Nikiforakis, N.: Catching transparent phish: Analyzing and detecting mitm phishing toolkits. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 36–50 (2021)
32. Le, H., Pham, Q., Sahoo, D., Hoi, S.: Urlnet: Learning a url representation with deep learning for malicious url detection. arxiv 2018. arXiv preprint arXiv:1802.03162 (2018)
33. Li, W., Manickam, S., Chong, Y.W., Leng, W., Nanda, P.: A state-of-the-art review on phishing website detection techniques. IEEE Access (2024)
34. Li, Y., Huang, C., Deng, S., Lock, M.L., Cao, T., Oo, N., Lim, H.W., Hooi, B.: {KnowPhish}: Large language models meet multimodal knowledge graphs for enhancing {Reference-Based} phishing detection. In: 33rd USENIX Security Symposium (USENIX Security 24). pp. 793–810 (2024)
35. Li, Y., Gu, C., Dullien, T., Vinyals, O., Kohli, P.: Graph matching networks for learning the similarity of graph structured objects (2019), https://arxiv.org/abs/1904.12787
36. Lin, Y., Liu, R., Divakaran, D.M., Ng, J.Y., Chan, Q.Z., Lu, Y., Si, Y., Zhang, F., Dong, J.S.: Phishpedia: A hybrid deep learning based approach to visually identify phishing webpages. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3793–3810 (2021)
37. Lindamulage, J., MandiraPabasari, L., Yapa, S., Perera, I., Krishara, J.: Vision gnn based phishing website detection. In: 2023 International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems (ICSES). pp. 1–7. IEEE (2023)
38. Malware Domain List: Mdl – malware domain list. http://www.malwaredomainlist.com/mdl.php (2021)
39. National Computer Network Emergency Response Technical Team/Coordination Center of China (CNCERT/CC): 2017 china network security technology challenge. http://www.cert.org.cn/ (2017), organized by CNCERT/CC
40. OpenPhish: Openphish dataset. https://www.openphish.com/phishing_database.html (nd)
41. Ouyang, L., Zhang, Y.: Phishing web page detection with html-level graph neural network. In: 2021 IEEE 20th international conference on trust, security and privacy in computing and communications (trustCom). pp. 952–958. IEEE (2021)
42. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library (2019), https://arxiv.org/abs/1912.01703
43. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
44. PhishTank: Phishtank dataset. https://phishtank.org/ (nd)
45. Pochat, V.L., Van Goethem, T., Tajalizadehkhoob, S., Korczyński, M., Joosen, W.: Tranco: A research-oriented top sites ranking hardened against manipulation. arXiv preprint arXiv:1806.01156 (2018)
46. Remya, S., Pillai, M.J., Aparna, B., Subbareddy, S.R., Cho, Y.Y.: Bgl-phishnet: Phishing website detection using hybrid model-bert, gnn, and lightgbm. IEEE Access **13**, 47552–47569 (2025)
47. Richardson, L.: Beautiful soup documentation. April (2007)
48. Safi, A., Singh, S.: A systematic literature review on phishing website detection techniques. Journal of King Saud University-Computer and Information Sciences **35**(2), 590–611 (2023)
49. Sakurai, Y., Watanabe, T., Okuda, T., Akiyama, M., Mori, T.: Discovering httpsified phishing websites using the tls certificates footprints. In: 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). pp. 522–531. IEEE (2020)
50. Sameen, M., Han, K., Hwang, S.O.: Phishhaven—an efficient real-time ai phishing urls detection system. Ieee Access **8**, 83425–83443 (2020)
51. Sánchez-Paniagua, M., Fidalgo, E., Alegre, E., Alaiz-Rodríguez, R.: Phishing websites detection using a novel multipurpose dataset and web technologies features. Expert Systems with Applications **207**, 118010 (2022)
52. Shakir, S.S., Mohammad Khanli, L., Emami, H.: Convolutional graph network-based feature extraction to detect phishing attacks. Future Internet **17**(8), 331 (2025)
53. Song, T., Casas, P., Meo, M.: Do we really need reference-based phishing detection? unleashing the power of gnn. In: 2025 9th Network Traffic Measurement and Analysis Conference (TMA). pp. 1–4. IEEE (2025)
54. Su, M.Y., Su, K.L.: Bert-based approaches to identifying malicious urls. Sensors **23**(20), 8499 (2023)

55. Trinh, N.B., Phan, T.D., Pham, V.H.: Leveraging deep learning image classifiers for visual similarity-based phishing website detection. In: Proceedings of the 11th International Symposium on Information and Communication Technology. pp. 134–141 (2022)
56. Varshney, G., Kumawat, R., Varadharajan, V., Tupakula, U., Gupta, C.: Anti-phishing: A comprehensive perspective. Expert Systems with Applications **238**, 122199 (2024)
57. Virtanen, P., Gommers, R., Oliphant, T.E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S.J., Brett, M., Wilson, J., Millman, K.J., Mayorov, N., Nelson, A.R.J., Jones, E., Kern, R., Larson, E., Carey, C.J., Polat, İ., Feng, Y., Moore, E.W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E.A., Harris, C.R., Archibald, A.M., Ribeiro, A.H., Pedregosa, F., van Mulbregt, P., SciPy 1.0 Contributors: SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. Nature Methods **17**, 261–272 (2020). https://doi.org/10.1038/s41592-019-0686-2
58. VirusTotal: Virustotal. https://www.virustotal.com (2021)
59. W3Schools: Html attribute reference. https://www.w3schools.com/tags/ref_attributes.asp (nd), accessed: 2025-08-03
60. W3Schools: Html tag reference. https://www.w3schools.com/TAGS/default.asp (nd), accessed: 2025-08-03
61. Wang, C., Liu, Z., Zeng, Y.: Detecting phishing urls with gnn-based network inference. In: 2024 International Conference on Networking and Network Applications (NaNA). pp. 504–509. IEEE (2024)
62. Wang, Y., Xue, S., Song, J.: A malicious webpage detection method based on graph convolutional network. Mathematics **10**(19), 3496 (2022)
63. WHATWG: Html living standard. https://html.spec.whatwg.org/, accessed: 2025-08-02
64. Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., Yu, P.S.: A comprehensive survey on graph neural networks. IEEE transactions on neural networks and learning systems **32**(1), 4–24 (2020)
65. Yoon, J.H., Buu, S.J., Kim, H.J.: Phishing webpage detection via multi-modal integration of html dom graphs and url features based on graph convolutional and transformer networks. Electronics **13**(16), 3344 (2024)
66. Zhang, P., Sun, Z., Kyung, S., Behrens, H.W., Basque, Z.L., Cho, H., Oest, A., Wang, R., Bao, T., Shoshitaishvili, Y., et al.: I'm spartacus, no, i'm spartacus: proactively protecting users from phishing by intentionally triggering cloaking behavior. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 3165–3179 (2022)
67. Zhang, X., Song, D., Tao, D.: Continual learning on graphs: Challenges, solutions, and opportunities (2024), https://arxiv.org/abs/2402.11565