AdCPG: Classifying JavaScript Code Property Graphs with Explanations for Ad and Tracker Blocking

Changmin Lee
KAIST

Sooel Son
KAIST

ABSTRACT

Advertising and tracking service (ATS) blocking has been safeguarding the privacy of millions of Internet users from privacy-invasive tracking behaviors. Previous research has proposed using a graph representation that models the structural relationships in loading web resources and then conducting ATS node classification based on this graph representation. However, these context-based ATS classification methods suffer from (1) inconsistent classification due to the varying context in which ATS resources are loaded and (2) a lack of explainability of the classification results, making it difficult to identify the code-level causes for ATS classification.

We propose AdCPG, a graph neural network (GNN) framework tailored for ATS classification. Our approach focuses on classifying JavaScript (JS) content rather than considering the loading context of web resources. Given JS files, AdCPG leverages their code property graphs (CPGs) and conducts graph classification on these CPGs that model the semantic and structural information of these JS files. To provide the explanations for ATS classification, AdCPG highlights the JS code that contributes the most to classifying the JS files into ATS using a GNN explainer. AdCPG achieved an accuracy of 98.75% on the Tranco top-10K websites, demonstrating high performance using only JS content. Upon deployment, AdCPG identified 650 JS files from 500 domains that were not detected by any ATS filter lists and previous ATS classification tools. AdCPG plays a complementary role in identifying ATS resources while providing code-level explanations, which minimizes the engineering effort required to validate ATS classification results.

CCS CONCEPTS

• **Security and privacy → Web application security**.

KEYWORDS

web tracking; web advertising; code property graph; graph neural networks; explainable AI

ACM Reference Format:
Changmin Lee and Sooel Son. 2023. AdCPG: Classifying JavaScript Code Property Graphs with Explanations for Ad and Tracker Blocking. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623084

CCS '23, November 26–30, 2023, Copenhagen, Denmark.

ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623084

1 INTRODUCTION

Every day, millions of Internet users rely on advertising and tracking service (ATS) blocking to prevent privacy-invasive advertising and tracking practices. AdBlock [2], a seminal ATS blocking extension, has been deployed on over 65 million users' browsers. Recent studies have shown that Internet users are aware of the presence of ATS resources and generally have strong intentions to take privacy-protective actions [9, 57].

Researchers have explored various machine learning (ML) algorithms tailored for ATS classification, which range from random forests to deep neural networks (DNNs), to develop high-performing classification models, thereby accurately identifying ATS resources [10, 26, 29, 31, 49, 50, 59, 62]. Recently, two studies, AdGraph [29] and WebGraph [49], proposed leveraging a graph representation that models the hierarchical context of loading HTML elements within a web page. They conducted node classification using a random forest classifier on each node that represented a network request trained upon graph node features.

However, our observation suggests that node classification based on the hierarchical context of network requests often results in inconsistent classification of identical JavaScript (JS) snippets that perform ATS behaviors. For instance, WebGraph classified network nodes fetching a JS snippet of serving Amazon ad content as ATS in 56.23% of the websites embedding this JS snippet. However, it reported the same JS snippet as Non-ATS in the remaining websites (Section 3).

Kaizer *et al.* and Wu *et al.* focused on classifying JS content itself rather than its loading context [31, 59]. They extracted the usages of built-in APIs related to ATS behaviors as features. However, their use of aggregated features makes it difficult to specify the JS code that contributes to classifying a JS file as ATS. This difficulty immediately exacerbates the manual engineering effort required to validate the classification results.

To overcome these limitations of (1) inconsistent classification and (2) a lack of explainability for ATS classification, we propose AdCPG, an ATS classification framework that checks whether a given JS snippet conducts ATS behaviors. The key idea of AdCPG is to leverage the content-rich structural and semantic information encoded in a code property graph (CPG) and to conduct graph classification on this CPG, thus addressing the first limitation. AdCPG also applies GNNExplainer [63], a seminal explainer technique, to highlight important CPG nodes that significantly contribute to ATS classification, thus addressing the second limitation.

AdCPG first converts a given JS file into a CPG and then prunes this CPG to remove unnecessary nodes and edges. AdCPG then leverages a graph neural network (GNN) classifier to classify the pruned CPG as either ATS or Non-ATS. Finally, AdCPG computes a node importance map that highlights the important nodes involved in conducting ATS behaviors.

```
1 (function() {
2   var retargetUrl = window.location.href;
3   if (window.parent && window.location !== window.parent.
        location) {
4     retargetUrl = window.parent.document.referrer;
5   }
6   const ticker = window.setInterval(function () {
7     if (!window.document.body) {
8       return;
9     }
10    clearInterval(ticker);
11    var pixel = document.createElement('img');
12    pixel.setAttribute('style', 'width:1px; height:1px;');
13    pixel.src = 'https://my.rtmark.net/img.gif?rurl=' +
        encodeURIComponent(retargetUrl);
14    window.document.body.appendChild(pixel);
15  }, 500);
16 }());
```

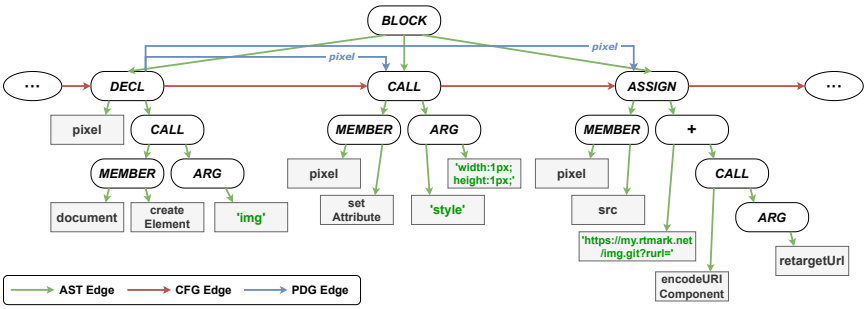


Figure 1: Exemplary ATS JS snippet and its CPG subgraph.

We evaluated the capability of AdCPG in classifying ATS scripts by using various spatial ratios of ATS resources. When evaluating AdCPG trained on the Tranco top-10K websites, AdCPG outperformed all previous approaches with respect to JS classification across all the ATS ratios. In particular, AdCPG achieved an accuracy of 98.75% and an area under the ROC curve (AUC) of 0.9328 when assuming an ATS ratio of 5%.

Moreover, when we deployed AdCPG for real-world ATS identification on random samples of 10K websites from the Tranco top-100K list, AdCPG reported 650 distinct JS files from 500 domains that WebGraph and existing filter lists overlooked. After analyzing the classification explanations for these files, we discovered that the existing filter lists and the previous ML approaches largely missed ATS behaviors that mainly involve JS execution, such as browser fingerprinting. These experimental results with the real-world websites highlight the complementary role of AdCPG in identifying ATS scripts with JS code-level explanations.

In summary, our contributions are as follows:

- We propose a GNN-based ATS detection framework using JS CPGs and tailor a GNN explainer to produce classification explanations that highlight the ATS-related JS code, which significantly eases the manual validation effort.
- We demonstrate the superior performance of AdCPG in JS classification over state-of-the-art ATS classification tools by conducting a deployment study on 10K websites sampled from the Tranco top-100K list.
- We shed light on the important and complementary role in classifying JS content by demonstrating the shortcomings of previous context-based and filter list-based detection approaches. To support reproducible research, we release AdCPG at https://github.com/WSP-LAB/AdCPG.

2 BACKGROUND

2.1 Code Property Graph

Yamaguchi *et al.* [61] proposed a comprehensive representation of source code called a code property graph (CPG). It contains an abstract syntax tree (AST), a control flow graph (CFG), and a program dependence graph (PDG), which are unified into a graph representation. They leveraged this unified graph representation to uncover security vulnerabilities in C applications. They devised graph traversal queries that model the characteristics of vulnerabilities and identified target paths that match these queries.

Recent studies have extended CPGs to cover JS applications. To identify malicious browser extensions, Fass *et al.* [18] extended a CPG to track suspicious data flows between external actors and sensitive APIs in the extensions. Li *et al.* [36] performed a flow- and context-sensitive static analysis on a extended CPG to detect the vulnerabilities in Node.js applications.

Figure 1 shows a JS snippet that dynamically embeds an invisible image pixel reporting the current URL of a visitor to an ATS website. Because the visitor's browser attaches a third-party cookie to the web request fetching the image pixel, the ATS website is able to track the web pages that the user has visited. The right-hand side of the figure shows a CPG subgraph that corresponds to the JS statements in Lines 11-13. Each CPG node in the graph represents an AST node. The CPG nodes and the AST edges connecting these nodes form an abstract syntax tree, which represents the syntactic structure of the JS snippet. The CFG edges among the CPG nodes describe the control flow that models the execution order of this JS code. The PDG edges connect the top nodes, each of which represents a JS statement, thus denoting the data dependence between these nodes. For instance, the blue lines in the figure denote that the parameter `pixel` has been declared before its usage in the JS statements corresponding to the connected `CALL` and `ASSIGN` nodes.

2.2 Graph Neural Networks

Graph neural networks (GNNs) have been applied to various classification tasks, including natural language, link prediction, knowledge graph node classification, and graph classification [22, 64, 68, 69]. Numerous GNN architectures, including graph convolutional networks (GCNs) [33] and graph attention networks (GATs) [56], have been proposed, enabling various ways of encoding the semantics and structure of a given graph.

The primary goal of a GNN is to compute the node embeddings that capture the relationships between neighboring nodes in a given graph. By encoding these relationships into a set of node embeddings, the GNN effectively encodes the overall graph structure. The GNN takes in a graph where information is loaded into its nodes and edges. It progressively transforms these embeddings without changing the graph connectivity. Since the GNN does not update the graph connectivity, the output graph is typically described using an adjacency matrix and feature vectors. The GNN updates the embeddings of each node in the graph. These updated node embeddings capture the evolving representation of each node based on its interaction with the neighboring nodes.

Message passing is a widely used technique for updating node embeddings [20]. This process involves iteratively aggregating node embeddings from neighboring nodes, allowing each node to gather the information from its neighbors, which is called messages, and update its own embedding accordingly. The GNN performs an aggregation step in which it combines all the messages from neighboring nodes. This step is typically conducted using a message function that receives the collected messages. By stacking multiple message-passing GNN layers, a node can progressively incorporate the messages from its neighboring nodes, enabling it to learn and reflect the information from the entire graph. The number of message-passing layers determines the extent to which node can learn and propagate the information from its neighbors throughout the graph.

For the graph classification task, a set of node embeddings are transformed into a graph embedding, which is fed into the final classification layer, mostly a multilayer perceptron (MLP). This process is called readout. When training a GNN model, the loss function is cross entropy loss, which aims to minimize the differences between the graph labels and predictions of the GNN model.

GCNs typically aggregate the information from neighbors by summing their embeddings with equal weights. In GATs, each node computes an attention score for each of its neighboring nodes. The attention score represents the importance or relevance of each neighbor in relation to the current node. This attention score is calculated using a self-attention mechanism that takes into account the embeddings of both the current node and its neighboring nodes. Therefore, GNN architectures depend on various factors such as the number of message-passing layers and the types of message and update functions.

Explainable GNN. To provide human-understandable explanations for GNN predictions, various instance-level explainers have been proposed [41, 63, 66, 67]. A typical GNN explainer receives a trained GNN model and an graph instance, and then computes an explanation, which is a small subgraph of the input graph or a subset of node features that play a decisive role in making the prediction of the input graph.

GNNExplainer [63] is a seminal explainer method; it is trained to place soft masks for edges and node features at the positions of important edges and node features. GNNExplainer recasts the problem of placing the soft masks as solving an optimization task that maximizes mutual information between predictions and subgraph candidates. Since examining all possible subgraph candidates is computationally intractable, GNNExplainer applies continuous relaxation to obtain the variational approximation of subgraph distributions. This approach is model-agnostic, thus enabling it to be applied to any GNN classification tasks.

3 MOTIVATION

ATS blocking has protected the privacy of millions of users. The downloads of AdBlock in Chrome Web Store exceeds 65 million, demonstrating the popularity and necessity of ATS blocking [2]. The current trend in blocking ATS resources has depended on human-compiled lists of regular expressions, which non-profit organizations have managed. EasyList [12] is an exemplary filter list that AdBlock has used for identifying ATS resources.

Table 1: Inconsistent classification results of WebGraph.

Label	Script domain	Consistency of WebGraph
ATS	google-analytics.com	97.13%
	facebook.net	96.36%
	licdn.com	94.10%
	...	
	amazon-adsystem.com	56.23%
Non-ATS	Various parties	97.13%
	google.com	87.81%
	jquery.com	96.12%
	...	
	Various parties	37.39%
	Total	82.12%

AdGraph [29] and WebGraph [49] are two recent approaches proposed for ATS classification, which leverage the graph representations of web pages. By instrumenting the Chromium browser or using preloaded JS libraries, they encoded the loading context of various HTML elements in a fetched web page into a graph and performed node classification to determine whether each node involved fetching or rendering ATS web resources, such as JS files or images. Because these context-based ML approaches are trained to capture the commonality of ad-rendering and tracking behaviors, they are able to detect potential false negatives that existing filter lists have not listed. WebGraph reported an accuracy of 94.32% among 10K sites sampled from the Alexa top-100K list, demonstrating the promising efficacy for its real-world deployment [49].

We argue that state-of-the-art ATS detection approaches, including WebGraph, suffer from two limitations: (1) inconsistent classification and (2) an insufficient level of explainability.

Inconsistent classification. Previous context-based approaches use the loading context of HTML/script/storage elements initiating network requests. This context information, including degree, centrality, and information flows regarding shared storage access, is instrumental in classifying a network request node as ATS or Non-ATS. However, the context information regarding diverse ATS nodes varies across websites, often resulting in inconsistent classification, even for the network nodes in different web graphs fetching the identical JS snippets. This inconsistency naturally brings false positives and negatives.

Table 1 shows inconsistent predictions of WebGraph for each identical JS snippet from the different websites embedding the script. Each row corresponds to a JS snippet, and the script domain column specifies the source domain of this snippet. The consistency column describes the portion of the websites that load the corresponding script and share the same classification results specified in the label column. For instance, 56.23% of network requests fetching the ATS script from amazon-adsystem.com were classified as ATS, and the remaining requests were classified into Non-ATS, although they fetched the identical JS content. These experimental results shed light on the direction of classifying JS content rather than leveraging context information varying among different graph nodes.

Insufficient explainability. Previous JS classification [31, 59] and context-based classification [29, 49] approaches rely on aggregated and preprocessed features, such as nodal degree, URL length, and built-in API invocation. Therefore, obtaining the explanation for a

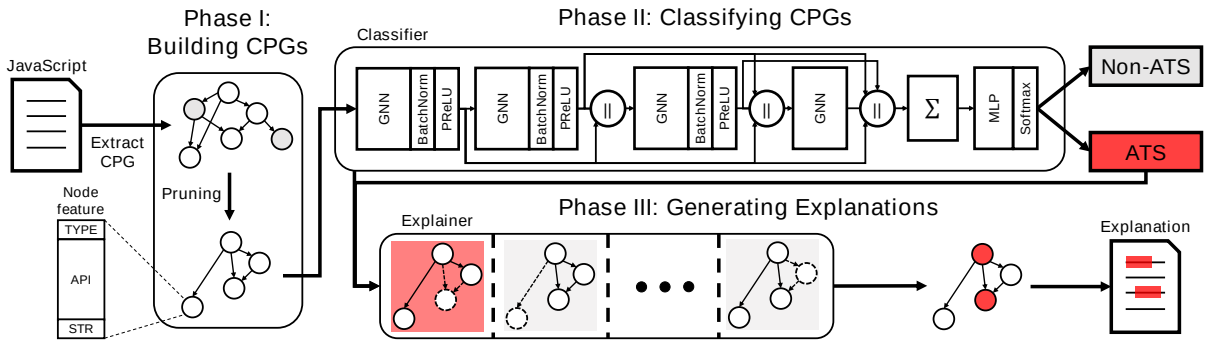


Figure 2: Overview of AdCPG.

prediction result has resorted to computing a feature importance map. Meanwhile, it remains elusive to pinpoint the code-level root causes of the computed prediction. This naturally entails difficulties for domain experts in validating the classification results (i.e., postmortem analysis); they need to further identify the code-level proofs for ATS behaviors.

For instance, assume that we validate an ATS prediction that WebGraph made on the network node fetching the script in Figure 1. Since WebGraph leverages a random forest classifier, we can compute a feature importance map to highlight the important features using TreeInterpreter [54]. Previous research showed that *average degree connectivity* is a key feature for ATS classification [29]. However, the value of this feature provides little information to validate the ATS classification. It is difficult to explicitly derive the code-level ATS behaviors from this feature attribution.

To overcome these limitations, we propose conducting GNN-based graph classification on a CPG that represents a given JS snippet and leveraging a GNN explainer to pinpoint the CPG nodes that contribute to classifying the CPG into ATS. By classifying a CPG that reflects a given script rather than its loading context, we address the first limitation, rendering AdCPG as a complementary method for detecting ATS resources. Moreover, classifying a CPG using the GNN model and leveraging the GNN explainer enables one to pinpoint the specific JS code that conducts ATS behaviors, thus addressing the second limitation.

3.1 Technical Challenges

Building a performant and explainable CPG classification framework entails two technical challenges: (1) CPG feature engineering and (2) highlighting the important CPG nodes that contribute the most to ATS classification.

CPG feature engineering. Although a CPG is an effective tool for capturing syntactic and semantic information for bug finding, it was not originally designed to be fed into a GNN model. Therefore, CPGs should be tailored for graph classification, thus extracting the necessary features for accurate and efficient classification.

Specifically, a CPG classification framework using a GNN should prune unnecessary CPG nodes and edges while preserving their underlying semantics. By design, CPGs contain a variety of node and edge types that may not be relevant for ATS classification. Even worse, such unnecessary components can adversely affect performance by increasing the distances among important CPG nodes that contain ATS-related JS code. It is known that GNNs perform poorly when prediction tasks depend on long-range interaction between distant graph nodes because prolonged node distances impede information propagation along the paths between graph nodes [3]. Furthermore, reducing CPG sizes boosts shortening the training and testing time by cutting down the computational resources in obtaining node embeddings and filling out more graph instances in a single batch.

To address these challenges, we suggest a CPG pruning algorithm that removes function subgraphs that invoke no built-in APIs related to ATS behaviors and prunes intermediate nodes while preserving the semantics of CPGs (Section 4.2).

Explainable classification. Given a CPG, it is not straightforward to pinpoint the important nodes that contribute the most to classifying the CPG into ATS. Extracting arbitrary features from the CPG may break the correspondence between source code and classification features. To preserve this mapping nature, we leverage a GNN designed to learn CPG node embeddings. In addition, by leveraging both the GNN and the GNN explainer, it becomes straightforward to locate the JS code that contributes to ATS classification. This enables one to examine the code areas where web tracking and advertising behaviors are highly likely to occur (Section 5.3).

4 DESIGN

4.1 Overview

We recast the ATS classification problem for a given JS snippet as a graph classification task. Figure 2 describes the overall workflow of AdCPG. Given a JS snippet, AdCPG conducts three steps to compute a prediction and its corresponding explanation, highlighting the important CPG nodes contributing to the ATS classification.

AdCPG begins with a given JS file, typically fetched from the Web. The first phase involves converting the JS file into a CPG and then processing the CPG by pruning unnecessary nodes and edges for precise and efficient classification. In the second phase, this pruned CPG is fed into a GNN classifier, which emits a prediction vector. The last phase of AdCPG involves leveraging a GNN explainer to highlight the CPG nodes that are highly attributed to the prediction.

4.2 Phase I: Building CPGs

Given a list of websites, AdCPG visits each website and stores each fetched JS script into a JS file. AdCPG then converts each JS file

Table 2: CPG node and edge types that AdCPG uses for computing CPGs.

	Type name	*w/o* Pruning	*w/* Pruning
Node	BLOCK	●	·
	CALL	●	●
	CONTROL_STRUCTURE	●	·
	FIELD_IDENTIFIER	●	·
	IDENTIFIER	●	●
	LITERAL	●	●
	LOCAL	●	●
	METHOD	●	●
	METHOD_PARAMETER_IN	●	●
	METHOD_REF	●	·
	RETURN	●	●
Edge	AST	●	●
	CFG	●	●
	PDG	●	●
	REF	●	·

into a CPG using Joern [30]. We revised Joern to consider the list of chosen node and edge types in Table 2. The table shows the CPG node and edge types that AdCPG uses to compose CPGs before (i.e., *w/o*) and after (i.e., *w/*) CPG pruning. AdCPG considers control flow (e.g., CFG), program dependence (e.g., PDG), and reference (e.g., REF) edges, but not call edges, thus converting each JS file into a set of subgraphs; each subgraph contains the intra-procedural semantic and syntactic information of a JS function block. We designed AdCPG to capture the structural and semantic information of a given JS file, thus preserving the semantics of the JS functionalities implemented in this file.

CPG features. We leveraged Pytorch Geometric [19] to convert a given CPG into a matrix representation. AdCPG encodes the CPG into a node matrix, each row of which represents a CPG node. The size of each row (i.e., the dimension of a node representation) is 333, which is a concatenation of three vector types: TYPE, API, and STR. Therefore, each node has a corresponding vector that encodes three kinds of CPG node information, as shown in Table 3.

A TYPE vector represents a CPG node type labeled by Joern. Specifically, given a CPG node, the TYPE vector is a one-hot vector that encodes its type among seven pruned CPG node types.

An API vector indicates whether a set of built-in objects or interfaces is referenced. The dimension of the API vector is 322, each element of which is 0 or 1, indicating whether the corresponding properties and methods appear in the CPG node. To identify a set of APIs that AdCPG uses to compute API vectors, we investigated MDN Web Docs [11], the ECMAScript 2023 Language Specification [52], and the DOM Standard [53]. We selected 861 methods, properties, and built-in objects listed in these specifications. Encoding these APIs as a one-hot vector is ineffective, considering that several APIs exhibit similar functionalities. Therefore, we clustered APIs into 322 groups according to their functionalities. We excluded some APIs with common names (e.g., `length`) and transferred APIs to other groups if necessary (e.g., `Location.toString` is more suitable for the `String` group than for the `Location` group).

Lastly, an STR vector is a four-dimensional vector that indicates the usage of HTML tag names, cookie attributes, JS events, and

Table 3: Node features used in AdCPG.

Feature	Component	Size	Type	Description
TYPE	Node type	7	All	CALL, IDENTIFIER, LITERAL, LOCAL, METHOD, METHOD_PARAMETER_IN, RETURN
API	Identifier	16	LOCAL	Objects used as identifiers
	Global	7	CALL	Global object
	Fundamental	2		Object, Function objects
	Utility	5		Date, String, RegExp, Array, JSON objects
	Event	1		Event, EventTarget interfaces
	Node	187		Node, Document, Element interfaces
	Window	88		Window object
	Navigator	1		Navigator object
	Screen	1		Screen interface
	Storage	1		Storage interface
	URL	3		Location, History, URL interfaces
	Request	1		XMLHttpRequest interface
	Performance	1		Performance, PerformanceTiming interfaces
	Observer	1		MutationObserver, IntersectionObserver interfaces
	HTML	7		HTMLElement interfaces
STR	Tag	1	LITERAL	HTML element tag names
	Cookie	1		Cookie attribute names
	Event	1		Event type names
	Resource	1		webRequest.ResourceType
	Total	333		

resources in the CPG node. When a node contains a string constant with one of nine tag names, eight property names in cookie attributes, 34 event type names, or 20 resource type names specifying `webRequest.ResourceType`, AdCPG sets 1 for the corresponding element in the STR vector; otherwise, AdCPG sets 0.

CPG pruning. AdCPG reconstructs a preprocessed CPG by removing unnecessary nodes and edges. For this, AdCPG intentionally deletes intermediate nodes to decrease the number of hops between nodes while preserving the semantics of JS files.

We deployed two deletion strategies: (1) pruning all CPG nodes in a JS function that does not invoke any built-in APIs used for computing API vectors and (2) deleting intermediate nodes that do not change the semantics of the CPG. These deletion strategies enabled AdCPG to reduce the number of nodes and edges by 47% and 30% on average, respectively, while boosting its performance. The detailed effects of CPG pruning is described are Section 5.4.

Algorithm 1 describes the graph reconstruction process. Lines 1-6 describe the pruning process of function subgraphs that do not invoke any APIs in Table 3. A JS function is represented as a subtree of which the root has the node type of METHOD. If the function calls no predefined built-in APIs, AdCPG removes all nodes in the subtree except the root node. Given target nodes and a CPG, *RemoveNodes* deletes the target nodes and all the edges connected to those nodes in the CPG. Lines 7-17 illustrate the process of deleting unnecessary intermediate nodes. *TransferEdges* takes a target node and its child or parent node, along with a CPG, as inputs. It then reconstructs the CPG by transferring the incoming and outgoing edges of the target node to its child or parent node, depending on the target node type. After transferring the edges, the target node and the self-loops of the child or parent node are removed from the CPG.

In short, AdCPG prunes intermediate nodes that have neighboring nodes corresponding to the same code tokens or that involve no predefined built-in API calls.

Specifically, we designed AdCPG to prune CPG nodes of which types are among BLOCK, CONTROL_STRUCTURE, FIELD_IDENTIFIER, IDENTIFIER, and METHOD_REF. For BLOCK and CONTROL_STRUCTURE nodes, which only indicate the presence of code blocks, AdCPG removes those nodes and uses their parent nodes connected through AST edges. AdCPG also replaces FIELD_IDENTIFIER nodes with their parents connected through AST edges, which have the node type of CALL. For IDENTIFIER nodes, AdCPG prunes these nodes and uses their parents, which have the node type of either LOCAL or METHOD_PARAMETER_IN, connected through REF edges. Similarly, METHOD_REF nodes are substituted by METHOD nodes that are linked to these nodes through REF edges.

We note that FIELD_IDENTIFIER, IDENTIFIER, and METHOD_REF nodes are replaced with their corresponding child and parent nodes. When performing this pruning step, AdCPG also makes the each replaced CPG node to indicate a code-level JS token to indicate a literal or an identifier that represents a function name, function parameter, field accessor, or a local variable. This pruning procedure with mapping code-level identifiers and literals facilitates the explanation generation in Section 4.4.

4.3 Phase II: Classifying CPGs

Given a pruned CPG from the previous phase (Section 4.2), AdCPG classifies this CPG as ATS or Non-ATS. AdCPG consists of three DNNs: node embedding networks, readout networks, and classification networks.

The node embedding networks consist of four layers; each layer starts with a graph convolutional layer, which is followed by a batch normalization layer and an activation layer except for the last layer. We used GATv2 [7] for the convolutional layer, and applied PReLU [23] for the activation layer. The dimension of hidden states is 16, which exhibited the best performance.

We designed AdCPG to densely connect all the layers by concatenating the output embeddings of all previous layers. This architecture enables each layer to access the features of all previous layers, offering feature reuses and lowering the number of model parameters, which enhances the model capacity to generalize to unknown data. It also mitigates the vanishing gradient issue by allowing gradients to flow across the entire networks [25].

For the readout networks, AdCPG conducts sum aggregation to convert node embeddings into a graph embedding. AdCPG receives node embeddings from the node embedding networks as inputs and sums all node features across the node dimension. The readout networks emit a 64-dimensional vector, which goes into the classification networks.

The classification networks produce a final classification result. We used a two-layer MLP followed by the softmax layer, which produces a prediction vector that shows the probabilities for ATS and Non-ATS, respectively.

4.4 Phase III: Generating Explanations

To generate an explanation, AdCPG takes a given CPG and the GNN classifier as inputs. It then computes a node importance map

Algorithm 1 Pruning CPGs.

Input: CPG $G = (V, E, X, T)$; V in DFS reverse pre-ordering of AST root; API feature vector $x_v \in X$; Node type $t_v \in T$;
Output: Pruned CPG G

for $v \in V$ **do**
  if $t_v \in$ {METHOD} **and** $Max(x_s) = 0, \forall s \in Subtree(v)$ **then**
    ▷ Remove a given set of nodes
    $G \leftarrow RemoveNodes(G, Subtree(v) - \{v\})$
  end if
end for
for $v \in V$ **do**
  if $t_v \in$ {IDENTIFIER, METHOD_REF} **then**
    ▷ Transfer the edges from a node v to its child connected through the REF edge
    $G \leftarrow TransferEdges(G, v, Child(v,\text{REF}))$
    $G \leftarrow RemoveNodes(G, \{v\})$
  else if $t_v \in$ {BLOCK, CONTROL_STRUCTURE, FIELD_IDENTIFIER} **then**
    ▷ Transfer the edges from a node v to its parent connected through the AST edge
    $G \leftarrow TransferEdges(G, v, Parent(v,\text{AST}))$
    $G \leftarrow RemoveNodes(G, \{v\})$
  end if
end for

in which CPG nodes are sorted according to their importance in computing the prediction of the CPG via a GNN explainer. Given a threshold for the proportion of important nodes to highlight, AdCPG produces a CPG subgraph that consists of nodes within the corresponding proportion. For visualization, AdCPG focuses on CPG nodes corresponding to the AST node types of identifier and literal. Each highly influential CPG node holds its location in the JS snippet and a JS code token that maps to a literal or an identifier. Therefore, the identified important CPG nodes can be highlighted in the script, computing an intuitive source code-level explanation.

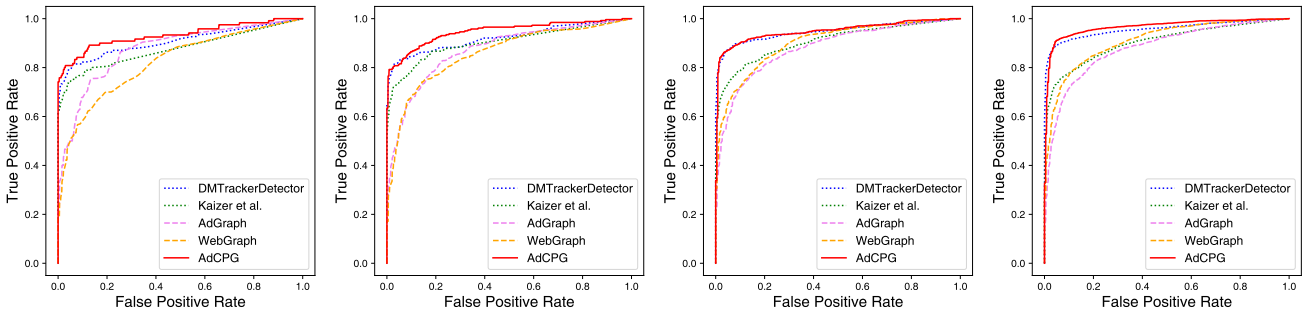
We applied GNNExplainer [63], which is a relatively lightweight explainer, to generate explanations for the computed predictions. Note that a typical JS CPG consists of thousands of nodes and edges. Computing explanations thus requires a huge amount of computational resources. SubgraphX [67], a seminal GNN explainer, requires a long time to compute its explanation because it explores a large number of subgraphs with the Monte Carlo tree search algorithm. A large CPG size exponentially hinders its practical use because of the large search space [66]. PGExplainer [41] can be an alternative explainer, which requires training on multiple instances to compute explanations. To avoid such an additional training stage, we selected GNNExplainer, which does not require an auxiliary dataset.

4.5 Implementation

We implemented AdCPG in Python. To build a crawler, we leveraged Selenium Wire [58]. AdCPG uses Joern [30] to extract CPGs from JS files. To convert CPGs into graphs and build a GNN classifier as well as its explainer, AdCPG leverages Pytorch [47] and Pytorch Geometric [19].

Table 4: Evaluation of AdCPG against previous ATS methods in JS classification. The best results are marked in bold.

Classifier	ATS 5%				ATS 10%				ATS 25%				ATS 50%			
	Accuracy	Precision	Recall	AUC	Accuracy	Precision	Recall	AUC	Accuracy	Precision	Recall	AUC	Accuracy	Precision	Recall	AUC
DMTrackerDetector	98.04%	93.27%	66.09%	0.9066	96.81%	95.93%	71.08%	0.9221	94.12%	94.15%	81.65%	0.9481	91.95%	95.41%	88.15%	0.9569
Kaizer *et al.*	97.72%	89.17%	63.08%	0.8766	95.41%	86.80%	64.63%	0.9036	89.75%	83.51%	73.67%	0.9075	83.39%	85.70%	80.23%	0.9039
AdGraph	96.01%	64.17%	26.67%	0.8735	92.24%	74.73%	38.57%	0.8714	86.19%	77.18%	63.81%	0.8814	80.87%	81.35%	80.21%	0.8793
WebGraph	95.91%	62.50%	13.33%	0.8207	92.19%	81.09%	29.52%	0.9186	87.34%	82.60%	62.70%	0.9021	83.25%	83.62%	82.75%	0.9092
AdCPG	**98.75%**	**98.18%**	**75.83%**	**0.9328**	**97.56%**	**96.53%**	**78.46%**	**0.9476**	**95.10%**	**95.14%**	**84.62%**	**0.9495**	**93.57%**	**96.34%**	**90.63%**	**0.9649**



(a) ATS 5% **(b) ATS 10%** **(c) ATS 25%** **(d) ATS 50%**

Figure 3: ROC curves for five classifiers: AdCPG, DMTrackerDetector, Kaizer *et al.* , AdGraph, and WebGraph.

5 EVALUATION

We evaluated the classification performance of AdCPG (Section 5.2) and its explainability (Section 5.3). We further conducted an ablation study to demonstrate the efficacy of each technical component with respect to precise classification, and examined the robustness of AdCPG against JS obfuscation (Section 5.5).

5.1 Experimental Setup

We crawled websites from the Tranco top-10K list, compiling the evaluation dataset. Each website was loaded with a timeout of 60 seconds, which involved fetching JS snippets and storing each snippet in a separate JS file. We collected a total of 101,278 JS files, each of which has the source URL from which it was fetched.

We then performed deduplication on these files, reducing the number of crawled JS files to 48,307 distinct files. To accommodate minor differences among JS files, we leveraged the AST representations of the files and grouped identical JS files by comparing the AST structures via Esprima [17] and Escodegen [16].

To label the JS files, we used four widely-used filter lists: EasyList [12], EasyPrivacy [13], Fanboy's Annoyance List [38], and Peter Lowe's List [39]. These filter lists contain regular expressions that can be matched against the source URLs of each JS file. We labeled a JS file as ATS when the source URL of the file matched the regular expressions in at least two filter lists. When there was no matching in any of the filter lists, we labeled the file as Non-ATS. Note that public filter lists can contain incorrect ATS filter rules because of crowd-sourced reports [28, 51]. Ill-intent advertisers have even registered fake advertising domains to undermine the integrity and performance of filter lists [37]. To establish trustworthy ground truth, we only considered only ATS scripts that matched regular expressions in at least two filter lists. Table 5 summarizes the statistics of the dataset on which AdCPG was evaluated.

Table 5: Ground truth data for training and testing AdCPG.

Label	# Distinct JS files	# Crawled JS files	Duplication ratio
ATS	2,726	20,539	7.53
Non-ATS	45,581	80,739	1.77
Total	48,307	101,278	2.10

To ensure unbiased training and testing, we evaluated AdCPG using various spatial ratios of ATS resources, as Pendlebury *et al.* suggested [45]. We assumed the ratio of ATS resources on the Web is either 5%, 10%, 25%, or 50%. For each spatial ratio, we made each data fold to reflect this ratio when conducting cross validation.

We conducted stratified 10-fold cross validation and reported the average value for accuracy, precision, recall, and AUC metrics. When training the GNN classifier, we used the AdamW optimizer [40] with a learning rate of 0.001, a weight decay of 0.01, and a batch size of 16. We adopted an early stopping with a patience of 50 epochs. For the GNN explainer, we computed a node feature map for 500 epochs with a learning rate of 0.001.

5.2 Comparison to Previous Works

We compared the performance of AdCPG against two previous JS classification methods, namely DMTrackerDetector [59] and Kaizer *et al.* [31]. We also included two previous approaches using web graphs, AdGraph [29] and WebGraph [49].

DMTrackerDetector is an ATS detection tool that classifies a given JS snippet using the presence of built-in JS API calls. Kaizer *et al.* developed an ML classifier trained upon features extracted from HTTP headers and a limited set of JS properties. AdGraph and WebGraph are designed to identify various types of ATS elements within a web page, including network requests that fetch JS snippets. We trained and tested these models on network request nodes

Table 6: Categories of ATS behaviors.

Category	Description
Cookie/Storage	Read or write data into browser storage (e.g., cookie, localStorage, sessionStorage, indexedDB)
Web beacons	Insert invisible or hidden HTML elements (e.g., image, iframe)
Fingerprinting	Perform browser fingerprinting
URL redirection	Redirect web pages
User activity tracking	Record user activities using non built-in functions
Resource requests	Send requests for ATS resources or load cross-domain ATS resources
Displaying ads	Load visible images, iframes, or style sheets

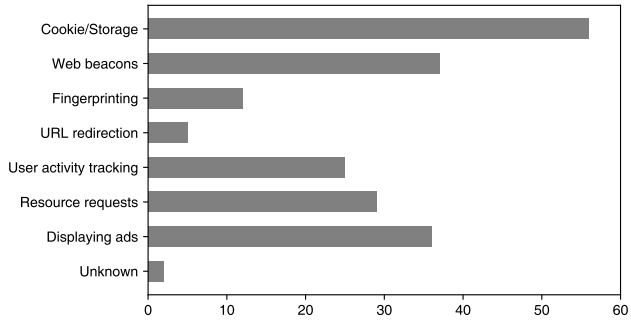


Figure 4: Frequency of observed explanations for each ATS category.

fetching JS snippets, which enabled us to assess the capability of identifying JS files originating from ATS hosts.

Table 4 and Figure 3 provide a comprehensive overview of the performance of AdCPG and other previous classification methods. The experimental results clearly indicate that AdCPG outperformed the other methods across all ATS spatial ratios. Assuming an ATS ratio of 5% in the underlying distribution, AdCPG attained an accuracy of 98.75% and an AUC of 0.9328, demonstrating high performance compared with all previous methods. We emphasize that AdCPG outperformed the other JS classification works, which also highlights the unique capability of AdCPG in leveraging content-rich CPG information rather than relying solely on API access and HTTP header features.

Note that the performance of the previous methods may differ from those metrics outlined in their papers. We conducted a thorough deduplication step and explored various spatial ratios to minimize any bias in classifying JS snippets (Section 5.1), which may affect the classification performance. In particular, AdGraph and WebGraph exhibited the performance drops in JS classification. When following the original setups of WebGraph [49], where the model is trained on network requests of all resource types regardless of ATS ratios and deduplication, WebGraph exhibited an accuracy of 94.78%, a precision of 90.67%, and a recall of 84.59%. However, when considering only the JS classification results, its performance decreased to an accuracy of 91.84%, a precision of 87.70%, and a recall of 73.82%. This implies that the performance of WebGraph can be aided by the classification results of other resource types besides JavaScript.

We observed that JS classification methods, including AdCPG, tended to perform well, especially when the ATS ratios were relatively low. In particular, AdCPG demonstrated its robustness in recall and AUC metrics when the ATS ratio was low. However, it becomes challenging to extract distinguishable patterns from the loading context of ATS scripts when an ATS ratio is low, which makes context-based classification methods perform less effectively. In contrast, JS snippets provide information-rich features even when the ATS ratio is relatively low, contributing to increasing the performance of the JS classification methods.

These experimental results necessitate the classification of JS snippets, which play a complementary role in filling in loopholes that the context-based classification approaches, including AdGraph and WebGraph, may introduce.

Error analysis. To examine false positives (FPs) and false negatives (FNs) of AdCPG, we established ATS categories, as shown in Table 6. We then manually analyzed the JS files of those FPs and FNs, checking whether they matched any of the categories in the table. If so, we labeled them ATS. We randomly sampled 50 files from each of the FPs and the FNs, analyzing a total of 100 JS snippets.

Out of the 50 FPs, 18 (i.e., 36%) were actually true positives. These cases were mislabeled because of FNs in the filter lists that constituted the ground truth of AdCPG. The remaining 32 reported FPs were genuine FPs; they exhibited abnormal behaviors, such as manipulating cookies while invoking numerous JS built-in functions involving `RegExp` as well as referencing properties like `Element.clientWidth` and `Element.clientHeight`, which are often observed in ATS scripts.

Among the 50 sampled FNs, seven (i.e., 14%) were actually Non-ATS scripts; we were unable to locate ATS-related behaviors in these files. The remaining ones were genuine FNs. The dominant root causes for the FNs were bootstrapping JS snippets that are responsible for preparing ATS behaviors without explicitly invoking built-in APIs. AdCPG struggles to effectively capture the subtle indicators associated with these snippets.

5.3 Explainability

To showcase the explainability of AdCPG, we conducted a manual analysis of AdCPG explanations for 100 randomly selected JS files classified as ATS. AdCPG highlights the CPG nodes that represent JS tokens, specifically those that significantly contribute to the ATS classification. We examined the highlighted code and categorized their behaviors into seven distinct categories, as outlined in Table 6. In cases where we were unable to determine the exact semantics of JS scripts, we categorized them as *Unknown*.

Figure 4 shows the analysis results of the sampled 100 JS files, representing a sample distribution of ATS behaviors. 56 of them were found to exhibit ATS behaviors associated with *Cookie/Storage* operations, such as setting cookie identifiers or cookie syncing. For example, AdCPG identified JS snippets that set the value of the `SameSite` cookie attribute to `None`, which attaches cookies to cross-origin requests [32]. The next most common category of ATS behaviors was the placement of *Web beacons*. Web beacons are widely used tracking methods that insert invisible HTML elements, reporting the current URL of visitors to ATS hosts.

```
(function() {
  var retargetUrl = window.location.href;
  if (window.parent && window.location !== window.parent.
     location) {
    retargetUrl = window.parent.document.referrer;
  }
  const ticker = window.setInterval(function () {
    if (!window.document.body) {
      return;
    }
    clearInterval(ticker);
    var pixel = document.createElement('img');
    pixel.setAttribute('style', 'width:1px; height:1px;');
    pixel.src = 'https://my.rtmark.net/img.gif?rurl=' +
     encodeURIComponent(retargetUrl);
    window.document.body.appendChild(pixel);
  }, 500);
}());
```

Listing 1: Explanation on the JS snippet categorized as *Web beacons.*

The source code-level explanations that AdCPG produces facilitate the analysis and validation of ATS behaviors in JS files, as Figure 4 shows, which is the core strength of AdCPG. Note that WebGraph and DMTrackerDetector produce no code-level explanations for their ATS classifications. Siby *et al.* [49] used TreeInterpreter [54], which can be applied to decision tree-based classifiers, to identify the most contributing features for a given prediction. They computed a feature importance map for each prediction, which can be considered as an explanation of WebGraph. However, identifying important features and their values has a limited impact on validating predictions, as mentioned in Section 3. DMTrackerDetector is not different because it uses aggregated statistical features that cannot be mapped to JS code.

In contrast, AdCPG explicitly highlights important JS code areas in a given JS file. We present case studies of AdCPG explanations in the following. The red boxes indicate the JS code corresponding to CPG nodes that are listed in the top 20% of important nodes. The orange ones correspond to nodes listed between the top 20% and 40% of important nodes in the ATS classification.

Case study 1: *Web beacons.* A web beacon is commonly used to track user accesses and interactions with web content [6]. An exemplary web beacon is shown in Listing 1, accompanied by an explanation generated by AdCPG. AdCPG highlights specific JS code that involves appending an invisible image pixel with its source attribute pointing to a third-party domain. The execution of this code triggers a web request to rtmark.net containing the current URL stored in the parameter `retargetUrl`. Note that rtmark.net is reported as a suspicious domain that performs malvertising [34, 60].

Since WebGraph supports computing a feature importance map, we computed explanations for network request nodes fetching this JS snippet. The explanations pointed out that *average degree connectivity* is one of the important features in ATS classification. However, this feature value itself provide little information about ATS behaviors. One needs to know the distributions of this feature for ATS and Non-ATS. A previous study revealed that ATS nodes tend to have lower *average degree connectivity* than Non-ATS nodes; ATS nodes stand apart from other nodes in a website because they only interact with ATS content which is a small portion of the entire website [29]. However, it is still not straightforward to validate this prediction solely using this feature value.

```
function Fingerprint2(a) {
  this.options = this.extend(a, {
    detectScreenOrientation: !0
  });
  ...
};
...
Fingerprint2.prototype = {
  getUserAgent: function () {
    return navigator.userAgent;
  },
  getScreenResolution: function () {
    var c = this.options.detectScreenOrientation ? [screen.
     height, screen.width] : null;
    ...
  },
  hasSessionStorage: function () {
    return !!window.sessionStorage;
  },
  ...
};
```

Listing 2: Explanation on the JS snippet categorized as *Fingerprinting.*

In the case of DMTrackerDetector, we revealed that the API usages of `Document.referrer` and `Node.appendChild` were the important features in classifying this JS snippet into ATS. Nevertheless, knowing the existence of these APIs does not always indicate the presence of web beacons because such built-in APIs are often used in Non-ATS scripts. On the other hand, from the explanation produced by AdCPG, one can confirm that the return value of `Document.referrer`, which is stored in the parameter `retargetUrl`, flows into `Node.appendChild` through the parameter `pixel`, aiding in validating the existence of a web beacon.

Case study 2: *Fingerprinting.* Fingerprinting refers to a technique used to collect users' device or browser information to compute a distinctive fingerprint for each individual [5, 14, 27, 35]. Listing 2 presents a JS snippet that performs browser fingerprinting, which AdCPG identified. It collects various browser information such as user agent, screen resolution, and the presence of session storage using APIs including `Navigator.userAgent`, `Screen.height`, and `Window.sessionStorage`. AdCPG highlights the function identifier `Fingerprint2`, which is also a strong indication of ATS. Note that AdCPG does not leverage any explicit ATS-related keywords, such as `ads`, `tracking`, `fingerprint`, or `banner`, in the node features. This indirectly shows that AdCPG learns the structural characteristics of the ATS JS code, and that the explainer uncovers what the model learns.

We emphasize that previous context-based approaches are unable to provide explicit code-level explanations, especially in the case of fingerprinting. AdGraph and WebGraph cannot identify fingerprinting-related behaviors because such ATS behaviors do not appear in a graph representation that captures the loading context of web resources, not internal interactions of JS API calls and their data flows.

In summary, AdCPG provides source code-level explanations of ATS classification compared to previous methods. We argue that this explanation capability highly alleviates the manual engineering effort of validating ATS classification results, especially for security operators who manage ATS filter lists, such as EasyList and Peter Lowe's List.

Table 7: CPG composition and classification performance differences according to CPG pruning.

	Metric	*w/o* Pruning	*w/* Pruning	Differences
Composition	# Nodes	36,911	19,715	-46.59%
	# Edges	139,465	97,685	-29.96%
	AST height	26	10	-61.54%
Performance	Accuracy	90.74%	93.57%	+2.83%p
	Precision	95.07%	96.34%	+1.27%p
	Recall	85.92%	90.63%	+4.71%p
	AUC	0.9604	0.9649	+0.0045

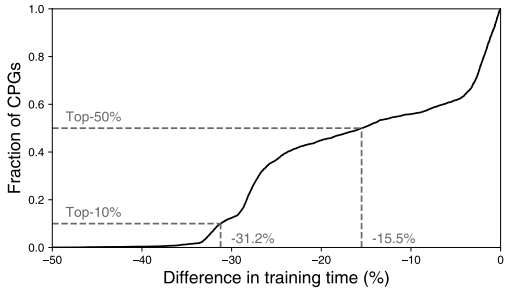


Figure 5: CDF for ratios of training time decrease according to CPG pruning.

5.4 Ablation Study

We conducted an ablation study to measure the efficacy of each technical component with respect to the performance of AdCPG.

CPG pruning. To test the impact of CPG pruning, we measured the classification performance and training time differences according to CPG pruning. Table 7 shows the comparisons of CPG composition and classification performance of AdCPG between without and with CPG pruning. The CPG pruning step significantly decreased the overall graph size; the number of nodes and edges decreased by 46.59% and 29.96%, respectively.

The main purpose of CPG pruning is to reduce the distances between nodes so that node embedding information efficiently propagates through close neighbors via message-passing layers. After pruning, the AST height, which represents the maximum node distance in the CPG, decreased by 61.54%. At the same time, we observed an overall boost in performance, especially in the recall that increased by 4.71%p with CPG pruning.

Moreover, CPG pruning reduced the training time of AdCPG. Figure 5 shows a cumulative distribution function (CDF) for training time decrease ratios after CPG pruning. The training time was measured when the batch size was 1, where each batch corresponded to a single CPG. The training time declined by 15.5% on average when CPG pruning was applied to CPGs, which can be attributed to the reduced time for computing node embeddings of each CPG. In addition, considering that the number of nodes decreased by 46.59%, the batch size can be nearly doubled, which further decreased the training time.

Node feature. Figure 6 shows the performance differences after redacting specific feature types in each CPG node. We measured the differences assuming that the underlying ATS ratio is 50%. Note that the overall performance metrics decreased after each node feature type was redacted. These experimental results demonstrate that each feature type contributed to boosting overall performance. We observed that removing API features in CPG nodes significantly degraded the performance, decreasing the recall by 5.36%p.

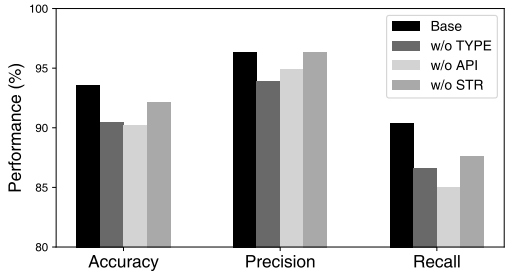


Figure 6: Performance differences according to the exclusion of each node feature type.

Table 8: Robustness of AdCPG against JS obfuscation.

Obfuscation technique	Recall	Semantics modification
Minification	100.00%	✗
Variable renaming	100.00%	✗
Control flow flattening	35.91%	✓
Dead code injection	30.42%	✓

5.5 Robustness

We evaluated the robustness of AdCPG in classifying obfuscated JS files. We classified obfuscated versions of 2,186 true positives from Section 5.2, varying the levels of obfuscation using JavaScript Obfuscator [44]. Table 8 shows the classification results of the obfuscated ATS scripts. When applying code minification and variable renaming as obfuscation techniques, AdCPG classified all scripts as ATS, achieving a recall of 100%. Because AdCPG leverages a CPG that has AST representations, it is robust to code structure changes. However, when faced with control flow flattening and dead code injection, AdCPG attained recall rates of 35.91% and 30.42%, respectively. These obfuscation techniques significantly alter the content of a given JS file, increasing the distances between CPG nodes by inserting meaningless intermediate nodes, degrading the performance of the GNN classifier.

It is worth noting that we did not observed heavily obfuscated ATS JS snippets detected by AdCPG or WebGraph. Heavily obfuscated JS files bring additional latencies and overheads in loading web pages, resulting in an adoption rate for heavy obfuscation of below 1% [43]. Furthermore, deploying heavily obfuscated ATS scripts provides a strong indicator for blocking such obfuscated third-party scripts, which a separate JS classifier can effectively detect.

AdCPG is robust to adversarial attacks [49, 70] that manipulate web page structures or URL components on a webpage. This resilience stems from AdCPG 's focus on classifying JS content rather than its hierarchical loading context. However, achieving robustness against adversarial attacks that perturb a JS snippet while preserving its semantics requires further research. We leave this for future work.

Table 9: Top-20 JS snippets consistently classified as ATS by AdCPG but not by WebGraph.

Script domain	Organization	Industry sector	Explanation							# JS files	Block rate of WebGraph
			Cookie/Storage	*Web beacons*	*Fingerprinting*	*URL redirection*	*User activity tracking*	*Resource requests*	*Displaying ads*		
twitter.com	Twitter	Social media	·	●	·	·	·	●	●	65	32.31%
twitter.com	Twitter	Social media	·	·	·	·	●	·	·	60	36.67%
Various parties	-	-	·	·	·	·	●	·	·	47	6.38%
google.com	Google	Advertising	·	●	·	·	·	·	·	45	2.22%
hsappstatic.net	HubSpot	Marketing	·	·	●	·	●	●	·	27	0.00%
google.com	Google	Advertising	·	·	·	·	●	·	·	24	20.83%
Various parties	-	-	●	●	●	·	·	·	·	21	9.52%
usercentrics.eu	Usercentrics	Software	●	·	·	·	·	·	·	19	42.11%
Various parties	-	-	·	·	·	·	●	·	·	15	20.00%
concert.io	Vox Media	Advertising	·	·	·	·	●	·	●	13	0.00%
yastatic.net	Yandex	Search engine	·	·	·	·	●	●	●	11	27.27%
speedcurve.com	SpeedCurve	Software	·	●	·	·	·	·	·	11	36.36%
wistia.com	Wistia	Marketing	·	●	●	·	·	·	●	10	20.00%
tildacdn.com	Tilda	Software	·	●	·	·	·	·	·	9	11.11%
Various parties	-	-	·	·	·	·	●	·	●	9	0.00%
cookielaw.org	OneTrust	Software	●	●	·	·	·	·	·	7	42.86%
Various parties	-	-	·	●	·	·	●	·	·	7	14.29%
paypalobjects.com	PayPal	Business	·	·	·	·	●	·	●	7	14.29%
paypalobjects.com	PayPal	Business	·	·	·	·	●	·	·	7	0.00%
baidu.com	Baidu	Internet	·	●	·	·	·	·	●	5	20.00%

6 DEPLOYMENT

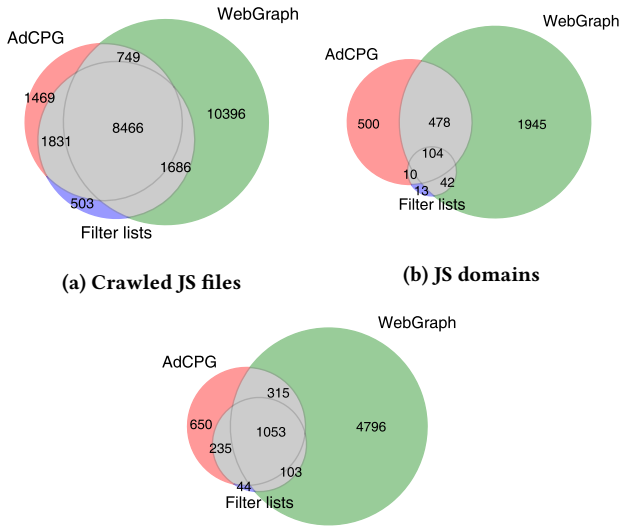
We conducted a deployment study to assess the effectiveness of AdCPG in identifying real-world JS snippets that engage in ATS behaviors. In addition, we deployed WebGraph [49] to contrast the advantages that AdCPG offers.

Deployment models. To test the classifiers on random samples of 10K websites from the Tranco top-100K list, we deployed the following models trained on the Tranco top-10K websites. For AdCPG, we used the model from Section 5.2, which had the highest AUC among the ten cross validation models assuming an ATS ratio of 50%. Specifically, we deployed AdCPG with an accuracy of 93.64%, a precision of 97.69%, a recall of 89.41%, and an AUC of 0.9752.

For WebGraph, we used the random forest classifier that followed the experimental settings outlined in the paper [49]. This involved training WebGraph on the dataset of all resource types but without taking into account deduplication and ATS ratios while utilizing the same filter lists from the paper. Similar to AdCPG, we deployed the model with the highest AUC among the cross validation models, which exhibited an accuracy of 91.92%, a precision of 91.60%, a recall of 88.96%, and an AUC of 0.9715.

Discovered ATS JS files. Figure 7 depicts Venn diagrams that illustrate the number of crawled JS files, crawled JS file domains, and distinct JS files, which are identified by AdCPG, WebGraph, and filter lists from Section 5.1. For the filter list-based approach, we considered a JS snippet as ATS if its URL matched the rules in at least two filter lists. Note that AdCPG identified 1,469 JS files that were not detected by WebGraph or the filter lists. This indicates that AdCPG uncovers ATS resources that would have been missed by the other detection methods.

In Figure 7a, the Venn diagram represents the inclusion relationships between the crawled JS snippets. Both AdCPG and WebGraph have unique areas of ATS JS files that other classifiers are



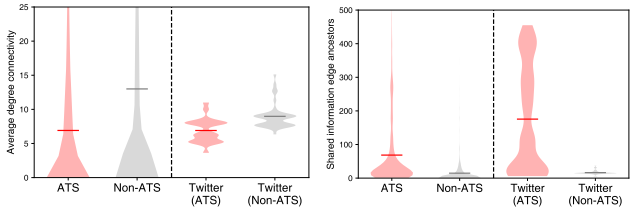
(a) Crawled JS files

(b) JS domains

(c) Distinct JS files

Figure 7: Comparisons of ATS JS files and their domains discovered by AdCPG, WebGraph, and filter lists.

unable to detect. This observation highlights the complementary role of AdCPG in finding ATS resources. Furthermore, we observed that WebGraph identified a larger number of ATS scripts compared to the other detection approaches. This is primarily due to the fact that we labeled JS snippets as ATS when their URLs matched at least one ATS filter list, which aligns with the original experimental settings of WebGraph. This broader definition of ATS classification led to a higher number of identified ATS JS files. At the same time,



(a) Average degree connectivity **(b) Shared information edge ancestors**

Figure 8: Distributions of important features.

this introduced a high number of FPs in WebGraph; 528 network nodes requesting the jQuery library were misclassified, particularly when the *average degree connectivity* was abnormally low. Figure 7b shows the inclusion relationships between the script domains of the crawled JS files in Figure 7a. It shows that AdCPG identified 500 script domains that were not identified by the other approaches.

The Venn diagram in Figure 7c shows the inclusion relationships between the identified JS snippets, taking into account that different URLs can serve the same JS snippet. We counted distinct JS snippets based on their AST structures. For WebGraph, we labeled a unique JS snippet as ATS when it consistently classified at least 50% of the network requests fetching the identical JS snippet. AdCPG reported 650 distinct JS files as ATS JS snippets that the other approaches were unable to identify or consistently classify.

From the 650 distinct JS files that only AdCPG classified as ATS, we sampled the most popular 20 JS files that were fetched or embedded across various websites. We examined each JS snippet using AdCPG explanations and summarized their behaviors in Table 9. The last column in the table indicates that WebGraph exhibited inconsistent classification with variations ranging from 0% to 42.86%. We also examined the domains from which these JS files were fetched and their industry sectors. As the table shows, AdCPG correctly identified prevalent JS files that engage in ATS behaviors in the advertising and social engineering sectors, where WebGraph exhibited inconsistent classification.

Postmortem analysis. We further analyzed the possible causes of the observed inconsistent classification by WebGraph. We analyzed the distributions of important features in the network request nodes fetching the JS files in Table 9. Figure 8 shows the distributions of the *average degree connectivity* and *shared information edge ancestors* features, which reported the highest information gains in ATS classification [29, 49]. In each figure, the left-hand side plots represent the distributions of the features from ATS and Non-ATS instances, and the right-hand side plots show those of the ATS and Non-ATS network requests that fetch the Twitter JS script conducting ATS behaviors. The width of each plot represents the frequency of the feature values. The solid line in each plot represents an average of the feature values. The left-hand side plots confirm that the feature distributions of ATS and Non-ATS are clearly different. However, the feature values of network nodes that fetch the identical ATS scripts from Twitter vary across different websites, as the right-hand plots show. Such a variance in the feature values of *shared information edge ancestors* is even more apparent, as shown in Figure 8b.

We observed that a lot of JS files solely classified by AdCPG engaged in *Fingerprinting* and *User Activity Tracking*. In particular, WebGraph did not identify any network requests that fetched a JS file from hsappstatic.net. This script is designed to conduct browser fingerprinting to extract various types of information using `Navigator` and non built-in APIs. As discussed in Section 5.3, WebGraph cannot detect ATS behaviors that involve the internal interactions of JS APIs. These observations demonstrate the necessity of JS classification based on CPGs, which addresses the FNs that existing state-of-the-art classification methods produce.

7 LIMITATIONS AND DISCUSSION

Coverage. AdCPG is specifically designed to classify JS snippets based on their CPGs. It does not cover other types of ATS resources, such as images and CSS files. However, we emphasize that ATS JS snippets serve as the foundation for loading other ATS web resources, including advertising images and iframes. Furthermore, JS code plays a crucial role in enabling various tracking behaviors, including fingerprinting. Therefore, AdCPG indirectly covers other ATS resources that are loaded by ATS scripts.

Due to a limited capability of the crawler, AdCPG is unable to capture inline scripts and scripts composed via dynamic code generation. This limitation can be addressed by a more sophisticated crawler using browser instrumentation. However, we emphasize that most ad-serving inline scripts involve fetching third-party JS snippets via network requests, which AdCPG is able to cover.

CPG feature engineering. AdCPG builds CPGs tailored for graph classification to identify ATS resources. Because extracting necessary features from CPGs involves manual investigation based on domain knowledge, additional feature engineering can further improve the performance. We attempted to encode API features by adding various combinations of fingerprinting-related interfaces from previous studies [5, 14, 27, 35], such as `BatteryManager` and `CanvasRenderingContext2D`. However, when augmenting those fingerprinting-related APIs to AdCPG, the accuracy fluctuated by less than 1%.

Deployment. AdCPG is well-suited for batch classification of JS files. Its ability to efficiently process multiple JS files makes it a valuable tool for compiling a filter list in the backend or assisting the maintainers of such a list. Given that extracting CPGs can be a time-consuming process, AdCPG can be utilized to automate the generation of a filter list based on ATS classification results.

Filter lists have limitations in terms of scalability, robustness, and coverage. Manually curating filter lists is a challenging task, as it cannot keep up with the rapidly changing web environment, and the coverage is often limited to popular websites [28, 51]. Moreover, filter lists relying on static, predefined rules are susceptible to being bypassed [4, 24]. ML-based approaches have been introduced to address these issues. However, they still have FPs and FNs, making it problematic to directly create filter rules based on their predictions. Therefore, manual verification is necessary to validate JS files classified as ATS. One of the challenges in manual verification is the lack of intuitive explanations provided by context-based approaches. An effective explanation should pinpoint the specific code responsible for ATS behaviors. Additionally, when dealing with lengthy JS files, manual verification can be time-consuming and labor-intensive.

In these cases, AdCPG can be particularly helpful. It assists in manual verification by pinpointing the JS code areas that are responsible for conducting ATS behaviors, which aids in categorizing ATS scripts and narrowing down the area of investigation. By providing the detailed explanations and highlighting the specific JS code related to ATS behaviors, AdCPG streamlines the manual verification process and facilitates the identification and validation of ATS resources.

8 RELATED WORK

Previous research has demonstrated that the presence of ATS resources is prevailing, threatening the privacy of users [1, 15]. Web advertising and tracking services have not only expanded their influence in various platforms, including desktop, mobile, and OTT devices, but also abused the security vulnerabilities of these platforms [1, 15, 42, 48].

URL-based detection. Adblock [2], Adblock Plus [46], and uBlock Origin [55] are popular URL-based ATS blocking tools that depend on crowd-sourced filter lists of matching ATS URL components against web resource URLs. Gugelmann *et al.* [21] extracted supplementary information from network requests and applied ML techniques to augment filter lists. Yu *et al.* [65] proposed a method for identifying tracking domains by detecting third-party domains that receive similar unique tokens across multiple first-party websites. However, previous studies have pointed out the limitations of URL-based detection in terms of scalability and robustness [4, 24, 28, 37, 51].

JavaScript-based detection. Motivated by the fact that ATS JS files invoke different APIs from Non-ATS JS files, Wu *et al.* [59] proposed DMTrackerDetector, an ML classifier that uses dynamic JS API calls as features. Kaizer *et al.* [31] focused on monitoring a limited number of APIs compared to DMTrackerDetector but additionally considered HTTP headers and metadata of JS snippets. These methods achieved high performance in classifying ATS scripts. However, they are unable to pinpoint which part of the JS code is responsible for the classifier, making final decisions. On the other hand, AdCPG provides a fine-grained explanation for each classification result.

Ikram *et al.* [26] proposed a one-class ML classifier to check the similarities of JS snippets by examining semantic and syntactic tokens from the JS code or their PDGs. However, their approach does not fully utilize the structural information of a given JS file. AdCPG addresses this limitation by leveraging a GNN classifier and learning the commonalities in CPGs that engage in ATS behaviors, without losing the semantic and structural information.

Context-based detection. Iqbal *et al.* [29] proposed AdGraph, the first context-based approach to building a graph representation of a given web page and extracting structural and content features from the graph. However, context-based detection inherently suffers from inconsistent classification varying by loading context. Moreover, context-based methods lack explainability; their explanations are important graph features in loading JS scripts, which cannot attribute specific JS code in which actual ATS behaviors take place.

There have been several attempts to supplement AdGraph by augmenting the graph representation with additional web page information [10, 49, 50]. Sjösten *et al.* [50] added perceptual features, which are resource content intercepted from the image rendering pipeline of a browser, into the graph representation of AdGraph. Chen *et al.* [8] generated execution signatures that abstract the ATS behaviors from the graph representation. Siby *et al.* [49] proposed WebGraph, which adds the flow information of the browser storage and network, which captures information sharing patterns between trackers, into AdGraph.

Yang *et al.* [62] constructed the graph representing HTTP network traffic and formulated ATS detection into edge representation learning and classification in the graph.

9 CONCLUSION

We present AdCPG, the first CPG-based classifier specifically designed for ATS classification. AdCPG addresses the technical challenges of adapting JS CPGs for graph classification using a GNN model. For this, we propose novel algorithms for pruning unnecessary nodes and edges in CPGs as well as computing node features tailored for GNN classification. One key strength of AdCPG is its ability to provide code-level explanations by leveraging a GNN explainer. This highlights important JS code that contributes to ATS classification, enabling the computation of intuitive explanations, greatly facilitating the validation process of ATS classification. AdCPG demonstrated the superior performance compared to existing state-of-the-art ATS detection tools. Upon deployment, AdCPG identified 650 distinct JS files that existing ATS filter lists and WebGraph were unable to identify, thus highlighting its complementary role in ATS classification.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their invaluable feedback. This work was supported by Korea Internet & Security Agency (KISA) grant funded by the PIPC (No.1781000003, Development of a Personal Information Protection Framework for Identifying and Blocking Trackers).

REFERENCES

[1] Gunes Acar, Christian Eubank, Steven Englehardt, Marc Juarez, Arvind Narayanan, and Claudia Diaz. 2014. The Web Never Forgets: Persistent Tracking Mechanisms in the Wild. In *Proceedings of the ACM Conference on Computer and Communications Security*. 674—-689.
[2] Adblock. 2023. https://getadblock.com/
[3] Uri Alon and Eran Yahav. 2021. On the Bottleneck of Graph Neural Networks and its Practical Implications. In *International Conference on Learning Representations*.
[4] Mshabab Alrizah, Sencun Zhu, Xinyu Xing, and Gang Wang. 2019. Errors, Misunderstandings, and Attacks: Analyzing the Crowdsourcing Process of Ad-Blocking Systems. In *Proceedings of the ACM Internet Measurement Conference*. 230—-244.
[5] Pouneh Nikkhah Bahrami, Umar Iqbal, and Zubair Shafiq. 2022. FP-Radar: Longitudinal Measurement and Early Detection of Browser Fingerprinting. In *Proceedings of the Privacy Enhancing Technologies Symposium*. 557–577.
[6] A.R.A. Bouguettaya and M.Y. Eltoweissy. 2003. Privacy on the Web: facts, challenges, and solutions. *IEEE Security & Privacy* (2003), 40–49.
[7] Shaked Brody, Uri Alon, and Eran Yahav. 2022. How Attentive are Graph Attention Networks?. In *International Conference on Learning Representations*.
[8] Quan Chen, Peter Snyder, Ben Livshits, and Alexandros Kapravelos. 2021. Detecting Filter List Evasion with Event-Loop-Turn Granularity JavaScript Signatures. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1715–1729.
[9] Savino Dambra, Iskander Sanchez-Rola, Leyla Bilge, and Davide Balzarotti. 2022. When Sally Met Trackers: Web Tracking From the Users' Perspective. In *Proceedings of the USENIX Security Symposium*. 2189–2206.

[10] Zainul Abi Din, Panagiotis Tigas, Samuel T. King, and Benjamin Livshits. 2020. Percival: Making in-Browser Perceptual Ad Blocking Practical with Deep Learning. In *Proceedings of the USENIX Annual Technical Conference*. 387–400.

[11] MDN Web Docs. 2023. https://developer.mozilla.org/

[12] EasyList. 2023. https://easylist.to/easylist/easylist.txt

[13] EasyPrivacy. 2023. https://easylist.to/easylist/easyprivacy.txt

[14] Peter Eckersley. 2010. How Unique is Your Web Browser?. In *Proceedings of the Privacy Enhancing Technologies Symposium*. 1—-18.

[15] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-Million-Site Measurement and Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1388—-1401.

[16] Escodegen. 2023. https://github.com/0o120/escodegen-python

[17] Esprima. 2023. https://github.com/Kronuz/esprima-python

[18] Aurore Fass, Dolière Francis Somé, Michael Backes, and Ben Stock. 2021. DoubleX: Statically Detecting Vulnerable Data Flows in Browser Extensions at Scale. In *Proceedings of the ACM Conference on Computer and Communications Security*. 1789—-1804.

[19] PyTorch Geometric. 2023. https://github.com/pyg-team/pytorch_geometric

[20] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. 2017. Neural Message Passing for Quantum Chemistry. In *Proceedings of the International Conference on Machine Learning*. 1263–1272.

[21] David Gugelmann, Markus Happe, Bernhard Ager, and Vincent Lenders. 2015. An Automated Approach for Complementing Ad Blockers' Blacklists.. In *Proceedings of the Privacy Enhancing Technologies Symposium*. 282–298.

[22] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. In *Proceedings of the IEEE International Conference on Computer Vision*. 1026–1034.

[24] Le Hieu, Markopoulou Athina, and Shafiq Zubair. 2021. Cv-inspector: Towards automating detection of adblock circumvention. In *Proceedings of the Network and Distributed System Security Symposium*.

[25] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

[26] Muhammad Ikram, Hassan Jameel Asghar, Mohamed Ali Kaafar, Balachander Krishnamurthy, and Anirban Mahanti. 2017. Towards Seamless Tracking-Free Web: Improved Detection of Trackers via One-class Learning. In *Proceedings of the Privacy Enhancing Technologies Symposium*. 1–21.

[27] Umar Iqbal, Steven Englehardt, and Zubair Shafiq. 2021. Fingerprinting the Fingerprinters: Learning to Detect Browser Fingerprinting Behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1143–1161.

[28] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. 2017. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *Proceedings of the ACM Internet Measurement Conference*. 171—-183.

[29] Umar Iqbal, Peter Snyder, Shitong Zhu, Benjamin Livshits, Zhiyun Qian, and Zubair Shafiq. 2020. AdGraph: A Graph-Based Approach to Ad and Tracker Blocking. In *Proceedings of the IEEE Symposium on Security and Privacy*. 763–776.

[30] Joern. 2023. https://github.com/joernio/joern

[31] Andrew J. Kaizer and Minaxi Gupta. 2016. Towards Automatic Identification of JavaScript-Oriented Machine-Based Tracking. In *Proceedings of the ACM on International Workshop on Security and Privacy Analytics*. 33—-40.

[32] Soheil Khodayari and Giancarlo Pellegrino. 2022. The State of the SameSite: Studying the Usage, Effectiveness, and Adequacy of SameSite Cookies. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1590–1607.

[33] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations*.

[34] Malwarebytes Labs. 2023. https://www.malwarebytes.com/blog/detections

[35] Pierre Laperdrix, Walter Rudametkin, and Benoit Baudry. 2016. Beauty and the Beast: Diverting Modern Web Browsers to Build Unique Browser Fingerprints. In *Proceedings of the IEEE Symposium on Security and Privacy*. 878–894.

[36] Song Li, Mingqing Kang, Jianwei Hou, and Yinzhi Cao. 2022. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *Proceedings of the USENIX Security Symposium*. 143–160.

[37] Su-Chin Lin, Kai-Hsiang Chou, Yen Chen, Hsu-Chun Hsiao, Darion Cassel, Lujo Bauer, and Limin Jia. 2022. Investigating Advertisers' Domain-Changing Behaviors and Their Impacts on Ad-Blocker Filter Lists. In *Proceedings of the ACM Web Conference*. 576—-587.

[38] Fanboy's Annoyance List. 2023. https://easylist.to/easylist/fanboy-annoyance.txt

[39] Peter Lowe's List. 2023. https://pgl.yoyo.org/adservers/

[40] Ilya Loshchilov and Frank Hutter. 2019. Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[41] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized Explainer for Graph Neural Network. In *Advances in Neural Information Processing Systems*.

[42] Hooman Mohajeri Moghaddam, Gunes Acar, Ben Burgess, Arunesh Mathur, Danny Yuxing Huang, Nick Feamster, Edward W. Felten, Prateek Mittal, and Arvind Narayanan. 2019. Watching You Watch: The Tracking Ecosystem of Over-the-Top TV Streaming Devices. In *Proceedings of the ACM Conference on Computer and Communications Security*. 131—-147.

[43] Marvin Moog, Markus Demmel, Michael Backes, and Aurore Fass. 2021. Statically Detecting JavaScript Obfuscation and Minification Techniques in the Wild. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks*. 569–580.

[44] JavaScript Obfuscator. 2023. https://github.com/javascript-obfuscator/javascript-obfuscator

[45] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *Proceedings of the USENIX Security Symposium*. 729–746.

[46] Adblock Plus. 2023. https://adblockplus.org/

[47] PyTorch. 2023. https://github.com/pytorch/pytorch

[48] Anastasia Shuba and Athina Markopoulou. 2020. NoMoATS: Towards Automatic Detection of Mobile Tracking. In *Proceedings of the Privacy Enhancing Technologies Symposium*. 45—-66.

[49] Sandra Siby, Umar Iqbal, Steven Englehardt, Zubair Shafiq, and Carmela Troncoso. 2022. WebGraph: Capturing Advertising and Tracking Information Flows for Robust Blocking. In *Proceedings of the USENIX Security Symposium*. 2875–2892.

[50] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. 2020. Filter List Generation for Underserved Regions. In *Proceedings of the ACM Web Conference*. 1682—-1692.

[51] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*. 1–24.

[52] ECMAScript 2023 Language Specification. 2023. https://tc39.es/ecma262/2023/

[53] DOM Standard. 2023. https://dom.spec.whatwg.org/

[54] TreeInterpreter. 2023. https://github.com/andosa/treeinterpreter

[55] uBlock Origin. 2023. https://ublockorigin.com/

[56] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

[57] Ben Weinshel, Miranda Wei, Mainack Mondal, Euirim Choi, Shawn Shan, Claire Dolin, Michelle L. Mazurek, and Blase Ur. 2019. Oh, the Places You've Been! User Reactions to Longitudinal Transparency About Third-Party Web Tracking and Inferencing. In *Proceedings of the ACM Conference on Computer and Communications Security*. 149—-166.

[58] Selenium Wire. 2023. https://github.com/wkeeling/selenium-wire

[59] Qianru Wu, Qixu Liu, Yuqing Zhang, Peng Liu, and Guanxing Wen. 2016. A Machine Learning Approach for Detecting Third-Party Trackers on the Web. In *Proceedings of the European Symposium on Research in Computer Security*. 238–258.

[60] Xinyu Xing, Wei Meng, Byoungyoung Lee, Udi Weinsberg, Anmol Sheth, Roberto Perdisci, and Wenke Lee. 2015. Understanding Malvertising Through Ad-Injecting Browser Extensions. In *Proceedings of the ACM Web Conference*. 1286—-1295.

[61] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*. 590–604.

[62] Zhiju Yang, Weiping Pei, Monchu Chen, and Chuan Yue. 2022. WTAGRAPH: Web Tracking and Advertising Detection using Graph Neural Networks. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1540–1557.

[63] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *Advances in Neural Information Processing Systems*.

[64] Donghan Yu, Yiming Yang, Ruohong Zhang, and Yuexin Wu. 2021. Knowledge Embedding Based Graph Convolutional Network. In *Proceedings of the ACM Web Conference*. 1619—-1628.

[65] Zhonghao Yu, Sam Macbeth, Konark Modi, and Josep M. Pujol. 2016. Tracking the Trackers. In *Proceedings of the ACM Web Conference*. 121—-132.

[66] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. 2022. Explainability in Graph Neural Networks: A Taxonomic Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022), 1–19.

[67] Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. 2021. On Explainability of Graph Neural Networks via Subgraph Explorations. In *Proceedings of the International Conference on Machine Learning*. 12241–12252.

[68] Muhan Zhang and Yixin Chen. 2018. Link Prediction Based on Graph Neural Networks. In *Advances in Neural Information Processing Systems*.

[69] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the AAAI Conference on Artificial Intelligence*.

[70] Shitong Zhu, Zhongjie Wang, Xun Chen, Shasha Li, Keyu Man, Umar Iqbal, Zhiyun Qian, Kevin S. Chan, Srikanth V. Krishnamurthy, Zubair Shafiq, Yu Hao, Guoren Li, Zheng Zhang, and Xiaochen Zou. 2021. Eluding ML-based Adblockers With Actionable Adversarial Examples. In *Proceedings of the Annual Computer Security Applications Conference*. 541–553.