AdVersa: Adversarially-Robust and Practical Ad and Tracker Blocking in the Wild

Chaejin Lim
Department of Electrical and Computer Engineering
Sungkyunkwan University
Suwon, Republic of Korea
chaejin98@skku.edu

Kiho Lee
Electronics and Telecommunications Research Institute (ETRI)
Daejeon, Republic of Korea
kiho@etri.re.kr

Beomjin Jin
Department of Electrical and Computer Engineering
Sungkyunkwan University
Suwon, Republic of Korea
Purdue University
West Lafayette, IN, USA
jin610@purdue.edu

Heewon Baek
Department of Electrical and Computer Engineering
Sungkyunkwan University
Suwon, Republic of Korea
heewb9818@skku.edu

Hyoungshick Kim
Department of Electrical and Computer Engineering
Sungkyunkwan University
Suwon, Republic of Korea
hyoung@skku.edu

Abstract

While machine learning has significantly advanced ad and tracker detection, existing systems face critical challenges in practice. They are vulnerable to adversarial attacks (57–92% evasion rates), fail to generalize to unseen domains due to data contamination, and suffer performance degradation over time, requiring costly retraining. To address these challenges, we present `AdVersa`, a client-side framework for robust and practical ad and tracker blocking. `AdVersa` leverages novel, hard-to-perturb latent features from code and URL embeddings to deliver state-of-the-art performance. On a 2.74M-request dataset, our results show that `AdVersa` achieves a 98.23% F1 score, twice the robustness against adversarial attacks, and strong generalization to unseen domains (91.47% F1 score). For sustainable protection, we demonstrate that a low-cost pseudo-labeling strategy can maintain near-optimal accuracy, reducing maintenance overhead by over 99.8% compared to filter-list curation. Finally, we implement `AdVersa` as a lightweight, standalone client-side application that ensures user privacy by operating without external dependencies.

CCS Concepts

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

Keywords

Ad blocking; Web tracking; Machine learning; Web security and privacy

WWW '26, Dubai, United Arab Emirates

ACM ISBN 979-8-4007-2307-0/2026/04
https://doi.org/10.1145/3774904.3792735

ACM Reference Format:
Chaejin Lim, Kiho Lee, Beomjin Jin, Heewon Baek, and Hyoungshick Kim. 2026. AdVersa: Adversarially-Robust and Practical Ad and Tracker Blocking in the Wild. In *Proceedings of the ACM Web Conference 2026 (WWW '26), April 13–17, 2026, Dubai, United Arab Emirates.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3774904.3792735

Resource Availability:
The source codes and dataset of this paper are publicly available at https://doi.org/10.5281/zenodo.18297209 and https://doi.org/10.5281/zenodo.17295030.

1 Introduction

Despite vigorous research to ensure users' privacy and security on the web, advertisements and web trackers continue to pose risks. Originated to provide enhanced user experiences and customized advertising, not all implementations remain user-friendly [58]. Threat actors exploit advanced techniques (*e.g.*, intrusive ads, malvertising [1], and pixel-based tracking [3]) to hinder web experiences and silently cross the boundaries of user consent.

Conventional mitigation relies on filter lists accessible through browser extensions [12, 21, 23, 25, 47] or built-in browser functionalities [4, 42]. However, crafting and maintaining static blacklists require ongoing crowd-sourced human labor, and evasive methods such as CNAME cloaking [10] and server-side tracking [20] blur the distinction between source websites and third-party content, hindering the effectiveness of filter lists.

Modern research has therefore turned to machine learning-based (ML) approaches, expecting enhanced generalization. These approaches can be broadly categorized by their primary modeling strategy, such as graph-based behavioral methods (*e.g.*, AdGraph [27] and WebGraph [53]) and feature-engineered classifiers (*e.g.*, AdFlush [34]). However, despite their promising detection accuracy, deploying these methods in the wild reveals three critical challenges.

First, the client-side architecture, which is essential for user privacy, inherently exposes models to adaptive adversarial attacks.

For example, the YOPO [52] attack leverages unlimited oracle access to build surrogate models and generate perturbations that evade state-of-the-art detectors with 57–92% success rates.

Second, we find existing evaluation protocols fail to guarantee real-world performance. Prior work often allows domain-level contamination between training and test sets, a flaw exacerbated by the web's highly centralized tracking ecosystem; for instance, a single entity, such as Google, is present on over 78% of top websites [35]. This leakage steers models toward memorizing ubiquitous tracker domains rather than learning generalizable features, resulting in misleadingly high accuracy scores.

Finally, the practical deployment and long-term maintenance of these systems also remain critical challenges. Graph-based approaches, such as AdGraph, require invasive modifications to the browser engine [7, 8]. Standalone blockers, such as AdFlush, mitigate this issue but introduce privacy risks by relying on external services, which can log or expose sensitive user browsing histories. Moreover, all learning-based models suffer from concept drift; our own experiments show that error rates of prior models increase by 23–69% over time, corroborating findings from AdFlush [34]. This highlights a pressing need for robust, private, and maintainable blocking frameworks with rigorous evaluation.

We introduce `AdVersa`, a fully-functioning online client-side framework for robust and practical ad and tracker blocking. By integrating novel, hard-to-perturb features with data-driven strategies, `AdVersa` achieves state-of-the-art results that are resilient to both adversarial attacks (23.64% ASR) and domain shifts (91.47% F1 score on unseen domains). Furthermore, we demonstrate that a low-cost pseudo-labeling strategy can efficiently maintain detection performance over time. Our approach sustains near-optimal accuracy (a mere 0.1–0.6% F1 score gap) while reducing manual labeling costs by over 99.8% compared to traditional filter-list curation. A video demonstration of our system is available at https://youtu.be/3Ld1CnOjEDo.

Our key contributions are:

- We introduce novel latent features from code and URL embeddings and demonstrate that our design is twice as robust against adversarial attacks as state-of-the-art models.
- We design and validate a rigorous, contamination-free evaluation protocol that addresses the limitations of prior work by enforcing strict domain-level separation.
- We collect and release a large-scale, five-month longitudinal dataset, providing our source code and benchmarks.[1]
- We propose and validate an efficient pseudo-labeling strategy for long-term model maintenance, reducing manual labeling costs by over 99.8% while sustaining near-optimal performance.

2 Related Work

2.1 Risks of Ads and Trackers

Modern web tracking employs surveillance techniques that pose privacy and security risks. Large-scale studies [6, 57] reveal widespread deployment of JavaScript-based fingerprinting via canvas [41], WebGL [5], audio context [49], and browser extensions [28], achieving over 83Recent evidence shows email exfiltration on nearly 3,000 websites before form submission [2], while CNAME cloaking [10] enables cross-site profiling and leaks authentication cookies [51]. These networks employ resurrection techniques through ETags, HTML5 storage, and probabilistic matching for persistent surveillance. Real-time bidding exposes personal information to hundreds of companies within milliseconds, creating attack vectors for session hijacking and unauthorized data aggregation.

2.2 Ad and Tracker Detection

Given the pervasive privacy and security risks posed by modern ads and trackers, numerous detection and blocking approaches have been developed. These approaches can be broadly categorized into filter list-based methods and machine learning-based methods, each with distinct strengths and limitations.

Filter List-Based Approaches. Existing blocking solutions rely on browser extensions (AdBlock Plus [47], uBlock Origin [25], Privacy Badger [21], Disconnect [12], Ghostery [23]) and built-in browser features (Firefox [42], Brave [4]) [37]. These depend on manually curated filter lists (EasyList [13], EasyPrivacy [14], Fanboy's List [18]) with limitations [38, 55]: intense manual labor, poor scalability, regional bias [26, 54], and performance degradation (90.16% of EasyList's 60,000+ rules provide no benefit) [22, 55]. Filter lists lag behind evasion tactics including script obfuscation [16] and anti-ad-blocker deployment on 686 websites [43], providing incomplete protection [59]. The fundamental limitations of filter-list approaches have driven research toward ML-based solutions that can adapt to evolving tracking techniques.

ML-based Approaches. AdGraph [27] pioneered graph-based representations, building comprehensive models of HTML structure, network requests, and JavaScript behavior to achieve 95.33% accuracy, though its content-based features make it susceptible to adversarial evasions. WebGraph [53] addresses this vulnerability by detecting ads and trackers based on users' web interactions, achieving comparable accuracy while reducing the adversary's success rate from nearly perfect to approximately 8%. Advanced techniques have further refined ML-based detection capabilities. AdCPG [33] presents a graph neural network framework that uses code property graphs to track JavaScript classification. Percival [11] focuses on detecting and blocking image ads by utilizing a CNN classifier trained on representations of the browser's rendering process for practical deployment. PageGraph [54] advances AdGraph and Percival to enhance performance even in underserved regions. AdFlush [34] demonstrated the promise of feature-engineering approaches by distilling deployable, high-value features through a multi-step filtering pipeline and showed effective production-grade blocking of ads and trackers.

Despite these gains, ML-based blockers remain vulnerable to evasion. YOPO [52] shows that universal adversarial perturbations can evade robust blockers under gray-box access, exposing fundamental weaknesses.

3 Threat Model and Goals

Our objective is to build a practical client-side framework, `AdVersa`, for real-time detection of web ads and trackers. To preserve user privacy and minimize latency, `AdVersa` runs entirely on the user's device without server-side classification or external dependencies.

[1] Dataset and code: https://github.com/SKKU-SecLab/AdVersa

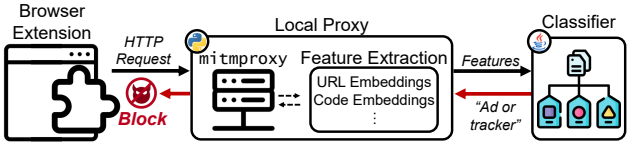


Figure 1: Architecture of AdVersa. The browser extension routes HTTP requests to the local proxy, which performs feature extraction and queries a request-level classifier. The classifier returns the predicted result to the proxy, which blocks the corresponding ad/tracker.

This on-device setting expands the adversarial surface relative to offline classifiers [53]. We assume a realistic attacker who can obtain the deployed model with unlimited query access (*i.e.*, oracle access) and manipulate web artifacts, including URLs (even through CNAME cloaking), DOM/HTML structure, and JavaScript behavior, to craft evasive inputs. Optimization processes of adversarial attacks such as YOPO [52] represent a superset of such evasive acts and effectively show the robustness of target models. Hardware-level exploits and denial-of-service attacks are out of scope.

Guided by this threat model, we set the following goals for AdVersa to structure our design and evaluation:

Goal 1: Performance and efficiency: surpass state-of-the-art detection quality while meeting on-device budgets for latency and memory.

Goal 2: Generalization and temporal stability: sustain high accuracy on previously unseen domains and remain stable under natural web evolution over time.

Goal 3: Adversarial robustness: resist manipulations of web content and behavior designed to evade classification.

4 Architecture of AdVersa

We designed AdVersa as a standalone, client-side application composed of three core components: a browser extension, a local forward proxy, and a request-level classifier. This architecture, illustrated in Figure 1, was deliberately chosen to overcome the limitations of browser extensions, which cannot natively run complex Python/Java-based machine learning models or perform sophisticated request interception, while ensuring user privacy by avoiding any external server communication.

The operational flow begins at the browser extension, which uses the `declarativeNetRequest` and `proxy` APIs to intercept outgoing requests and redirect them to the local proxy. The local proxy, built as a Python add-on to `mitmproxy` [9], then extracts a feature vector from the URL, headers, and associated JavaScript. This vector is passed to the classifier, a pre-trained `H2O MOJO` model running in a Java application. The classifier predicts whether the request is an ad or tracker and returns the label to the proxy. If so, the proxy drops the connection, effectively blocking it. Both proxy and classifier are multi-threaded to handle concurrent requests. Our implementation uses Python 3.13 and JDK 17, and the extension is compatible with Chromium-based browsers (*e.g.*, Google Chrome, Microsoft Edge, Opera).

This standalone design presents a significant advantage over prior work, such as AdFlush, which relies on an external server (`Requestly.io`) for request manipulation, introducing potential privacy risks and latency. By handling all operations on the user's device, AdVersa provides robust ad and tracker blocking without compromising user privacy. The detailed methodology for constructing the classifier itself is presented in Section 6.

Table 1: Per-month domain/request counts for our contamination-free splits: in-domain train (A_i^{Tr}), in-domain test (A_i^{Te}), out-of-domain (B_i), and full monthly crawl (U_i).

Dataset	April		May		June		Total	
	# dom.	# req.	# dom.	# req.	# dom.	# req.	# dom.	# req.
A_i^{Tr}	6,708	486,596	5,565	236,188	6,805	489,439	8,057	1,212,223
A_i^{Te}	5,864	119,919	4,947	59,048	5,818	120,359	7,179	299,326
B_i	1,000	89,593	901	81,286	859	76,933	1,000	247,812
U_i	7,927	696,108	6,525	376,522	7,887	686,731	9,159	1,759,361

Dataset	July			August		
	# dom.	# req.	Purpose	# dom.	# req.	Purpose
A_i	6,903	599,123	Retraining	3,150	266,793	Temporal-shift eval.
B_i	822	78,103	Domain-shift eval.	404	37,709	Domain-shift eval.

5 Dataset Configuration

Data Collection and Labeling. To build a comprehensive longitudinal dataset, we crawled the top 10k websites ranked by Tranco [48] monthly over five months from April to August, yielding over 2.74 million total requests. The crawling process was automated using OpenWPM [16] with five parallel instances running for 16 hours each month. We provide the dates of collection in Appendix A. For ground-truth labeling, we followed established precedent [27, 53] by aggregating five widely-used filter lists: EasyList [13], EasyPrivacy [14], Fanboy [18], Peter Lowe's List [36], and the Adblock Warning Removal List [15].

Data Partitioning. For each monthly dataset U_i (month i), we partition the 10k domains into an in-domain set A_i (9k domains) and an out-of-domain set B_i (1k domains) to assess generalization to unseen domains. The in-domain set A_i is further split at the domain level by an 80:20 ratio to create a training set A_i^{Tr} and an in-domain test set A_i^{Te}. All model training, feature engineering, and hyperparameter tuning via cross-validation are performed exclusively on the training sets from April to June ($A_i^{Tr}, i \in [4, 6]$). We strictly enforce that no domain or request overlaps exist between any training and testing partitions, even across different months (*i.e.*, $A_i^{Tr} \cap A_j^{Te} = \emptyset$ for all i, j), preventing any form of data leakage. The datasets from July and August (U_7, U_8) are held out entirely to evaluate temporal stability. Further details are shown in Table 1[2].

Ethical Considerations. During our data collection process, we minimized server impact with a strict 60-second timeout. We further confined crawling to only target the homepages without subpath traversal, naturally aligning with typical `robots.txt` policies restricting access to critical sub-directories.

6 Classifier Construction

To address the critical challenges of previous work, we design AdVersa through a systematic four-phase approach. *Phase I* introduces novel deep embedding features (URL and code embeddings) that are infeasible to perturb, addressing adversarial robustness.

[2]We report that our dataset collected in August discontinued with a crash. This was due to recurring web driver version compatibility issues that we had once fixed in WebGraph's OpenWPM [50].

Table 2: Feature engineering pipeline from 536 features; parentheses indicate the % retained after each step (point-biserial, RFECV, correlation).

Month	Init. Features	Point-Biserial	RFECV	Correlation
April	536 (100%)	511 (95.34%)	239 (44.59%)	**198** (36.94%)
May	536 (100%)	501 (93.50%)	320 (59.70%)	**214** (39.93%)
June	536 (100%)	509 (94.96%)	265 (49.44%)	**201** (37.60%)

Phase II applies rigorous feature engineering with temporal consistency analysis to identify features that generalize across domains and time. *Phase III* evaluates multiple feature selection strategies to balance performance and efficiency. *Phase IV* selects the optimal model and implements it as a standalone client-side application, eliminating external dependencies. This methodology is grounded in comprehensive longitudinal data collection and strict evaluation protocols that prevent domain contamination.

6.1 Phase I: Robust Feature Representation

Our framework begins by exploring effective features to detect ads and trackers. To increase performance and robustness, we incorporate multiple features with additional data representations that act as black-box features, hindering adversarial input generation. We develop two categories of novel features to effectively capture the real-world characteristics of ads and trackers.

URL Embeddings. We adopt the lightweight `nomic-embed-text-v1.5` [44] model to obtain rich representations of URLs. It uses Matryoshka representation learning [30] to generate variable-size embeddings, providing flexibility. To balance efficiency and richness, we use 128-dimensional vectors, average-pooled for each of the source's fully qualified domain name (FQDN) and the target URL.

Code Embeddings. While AdFlush [34] utilizes `n-grams` of the window size 3 to represent the behaviors of JavaScript source codes, another previous work [19] suggests using at least 4-length windows to properly dissolve the behavior. However, this introduces a trade-off of feature extraction time, crucial in real-time ad and tracker prevention. Therefore, we prefer to utilize `CodeMalt` [40], a static embedding model derived from the distillation of `MPNet` [56]. This model is capable of extracting code representations using a static model of 14.4MB, enhancing efficiency while preserving the effectiveness of rich code representations. This way `AdVersa` obtains code embeddings of 256 dimensions for JavaScript source codes, including those inlined in HTML frames.

Hand-Crafted Features. We incorporate 24 features proven effective in prior work [27, 34, 53]: ad keywords and size hints, typical URL patterns, third-party origins, domain echoes, semicolon-packed parameters, parent-script launches, cookie and storage counts, request fan-out, and AST metrics to capture code complexity.

In this phase, we generate and integrate a comprehensive feature set comprising 536 dimensions (see Appendix B), which encompasses 256 URL embeddings (128 each for FQDNs and target URLs), 256 code embeddings, and 24 handcrafted features.

6.2 Phase II: Feature Engineering and Filtering

Inspired by the validated methodology of AdFlush [34], we designed a systematic, multi-step pipeline to refine our initial feature set into the most effective signals. The outcome of each filtering stage, applied monthly, is summarized in Table 2.

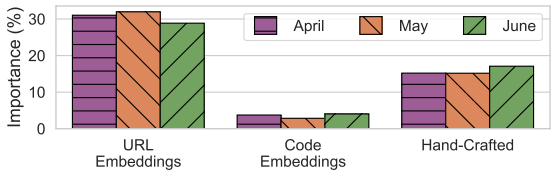


Figure 2: Category-level feature importance by month (%).

Feasibility Analysis. The first stage of our pipeline removes features that are infeasible for a real-time, on-device classifier. We define infeasible features as those that would compromise user privacy by requiring responses from untrusted servers, executing unchecked JavaScript, or aggregating data across multiple requests. This initial screening ensures that all subsequent stages operate exclusively on safe, deployable signals.

Label Correlation Analysis. We then utilize the point-biserial correlation coefficient [29] to eliminate the features that obtain insignificant correlation with the labels (p-value less than 0.05). This way, we can prevent noisy features from affecting performance throughout the feature engineering phase.

Performance Analysis with Cross-Validation. We further remove features with limited performance impact using recursive feature elimination with cross-validation (RFECV) [24] based on Random Forest models. This yields feature subsets that empirically stabilize validation performance for the chosen model.

Feature-wise Correlation Analysis. The final process of feature reduction involves eliminating unnecessary redundancy within features. We retained those having a significant correlation of p-values less than 0.05. Operationally, we prune pairs with strong pairwise Pearson correlation (tested two-sided) by removing one feature from each highly correlated pair, preferring the feature with higher RFECV importance or broader monthly support.

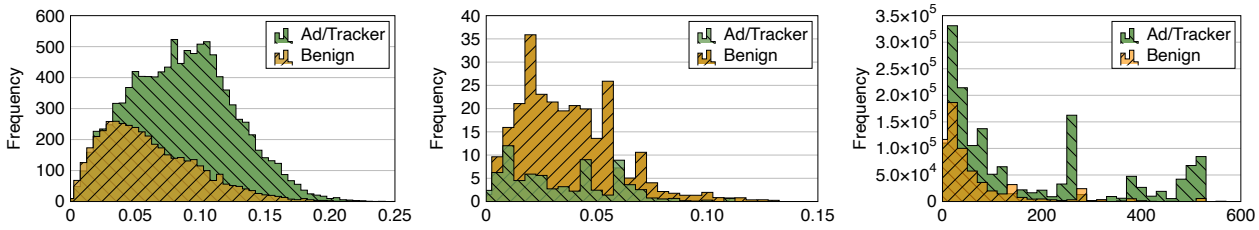
Analyzing Effective Features. We repeat the procedures above for each monthly dataset, yielding effective features per period. Figure 2 shows the total importance score of each embedding type per month, measured by averaging Random Forest and permutation importance. Overall, URL embeddings proved most effective, followed by domain and code embeddings (see Appendix B).

Within URL and code embeddings, indices #8 and #26 were most dominant. Figure 3 shows their separable distributions. Observing captured patterns in URL embedding #8, we find that that high values correlate with longer URLs. Kendall's τ correlation confirms this relationship (0.26, $p < 0.001$), supporting the well-known pattern that ads and trackers use longer URLs.

For code embedding #26, low values correspond to simple code-like responses from servers, as shown in Listing 1: (a) a short snippet for initiating tracker settings and (b) constructing variables containing user data. These privacy-concerning code snippets provide consistent cues for detecting ads and trackers, verifying the effectiveness of deep code embeddings.

6.3 Phase III: Feature Set Selection

In this phase, we consider four candidate feature sets using different heuristic strategies to manage the trade-off between model performance and overhead. By leveraging insights on which features are most effective over time, these strategies aim to create more efficient sets without significant performance degradation.



(a) URL embedding #8 (b) Code embedding #26 (c) Number of get cookie

Figure 3: Separable class-conditional distributions of highly influential features.

```
(function(parameters) {
  var cs=d.currentScript;
  if(cs){
    var uo=cs.getAttribute('data-ueto');
    if(uo&&w[uo]&&typeof w[uo].setUserSignals==='function
        ') { w[uo].setUserSignals({parameter dictionary}}
})(user data);
```

(a) JavaScript sample of setting Microsoft Universal Event Tracking.

```
var pmc_fastly_geo_data = {
    "asn": asn, "city": city, "region": regionID,
    "country": countryID, "continent": continentID}
```

(b) JavaScript sample of defining variables of user data.

Listing 1: JavaScripts with low code embedding #26 values.

- *Importance Consistency (C).* As the importance scores obtained in *Phase II* vary during the period, we define consistent features as those whose importance increased, regardless of their absolute value. They result in a total of 114 consistent features.
- *Intersection (I).* We identify the core features that persist throughout effective features across the entire period. They result in a total of 145 features for the intersection.
- *Frequency Filtering (F).* We further remain features that appear in at least two months. Therefore, by filtering out features that are significant only once, we obtain 205 features.
- *Union (U).* Accumulating all features that were at least once identified as significant any month, we obtain 262 features.

6.4 Phase IV: Model Selection

This final phase selects the optimal classifier by evaluating the candidate feature sets from Phase III. Using `H2O AutoML` [32] with 5-fold cross-validation, we trained and evaluated a model for each of the four feature selection strategies. To establish a robust baseline, we also created a model, hereafter referred to as AdFlush (Re-impl.), by applying the original AdFlush feature engineering process to our entire initial feature set.

The performance of each model on our test set is presented in Table 3. The model using the *frequency filtering* strategy, `AdVersa` (F), emerges as the top performer, achieving the highest F1-score even compared to the model trained on the larger union set. Notably, all four of our proposed strategies significantly outperform the AdFlush (Re-impl.) baseline, validating the effectiveness of our temporal-aware feature selection methodology. While the importance, consistency, and intersection strategies yield competitive performance with fewer features, AutoML paired them with more complex model architectures (*e.g.*, Gradient Boosting Machines), resulting in larger final model sizes.

Table 3: Performance of the models trained on our candidate feature subsets. Notations following our model indicate feature selection strategies: C=Importance Consistency, I=Intersection, F=Frequency Filtering, U=Union.

Model	# Feat.	Size (MB)	Acc. (%)	Pre. (%)	Rec. (%)	F1 (%)
AdFlush (Re-impl.)	213	4.7	97.64	98.23	94.83	96.50
AdVersa (C)	114	15.0	98.57	97.96	97.88	97.92
AdVersa (I)	145	8.8	98.32	97.91	97.16	97.53
AdVersa (F)	205	6.5	**98.79**	98.22	**98.25**	**98.23**
AdVersa (U)	262	5.0	98.73	**98.27**	98.03	98.15

Table 4: Performance of our detection model `AdVersa` compared to state-of-the-art models. Percentage error (PE) is calculated based on AdFlush, with arrows indicating its sign.

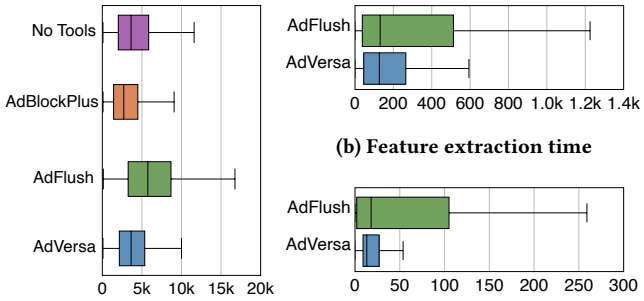
Model	# Feat.	Size (MB)	Acc. (%)	Pre. (%)	Rec. (%)	F1 (%)	PE (%)
AdGraph [27]	28	955.0	93.62	89.92	91.64	90.77	73.84 ⇑
WebGraph [53]	59	1,097.6	90.43	85.53	86.74	86.13	160.76 ⇑
AdFlush [34]	27	2.0	96.33	95.18	94.06	94.62	-
AdFlush [34] (Re-impl.)	213	4.7	97.64	**98.23**	94.83	96.50	35.69 ⇓
AdVersa	205	6.5	**98.79**	98.22	**98.25**	**98.23**	**67.03** ⇓

Considering the superior balance of performance and model compactness, we select `AdVersa` (F) as our final classifier for the `AdVersa` framework. The optimal architecture identified by AutoML is a stacked ensemble, which consistently demonstrated the best performance during cross-validation (see details in Appendix C).

7 Evaluation

7.1 Goal 1: Performance and Efficiency

7.1.1 Detection Performance. We compare `AdVersa`'s detection performance with state-of-the-art models, including AdGraph [27], WebGraph [53], and AdFlush [34], as shown in Table 4. We include AdFlush (Re-impl.), and exclude Percival [11] and PageGraph [54] due to their additional use of image data. To ensure fair comparison, we train each classifier using 5-fold cross-validation on our training set ($A_i^{Tr}, i \in [4,6]$). Performance is reported as the median across five folds in terms of accuracy, precision, recall, F1 score, and percentage error (PE) on our test set ($A_i^{Te}, i \in [4,6]$). PE measures the relative error reduction in accuracy, calculated as $PE(\%) = \frac{Acc_M - Acc_{Cri}}{Acc_{Cri}} \times 100\%$, where Acc_M is the accuracy of model M and Acc_{Cri} is the baseline accuracy. We use AdFlush as the baseline, as it achieves the highest accuracy among prior work.



(a) Full page load time

(b) Feature extraction time

(c) Classifier inference time

Figure 4: Runtime overhead of the vanilla Chrome browser compared to AdBlockPlus, AdFlush, and AdVersa's real-world implementations on top 10k websites.

AdVersa outperforms all baselines with 98.25% recall and 98.23% F1 score, reducing the error rate by 67.03% compared to AdFlush. AdFlush (Re-impl.), AdFlush, AdGraph, and WebGraph achieve F1 scores of 96.50%, 94.62%, 90.77%, and 86.13%, respectively. Despite using more features, AdVersa remains lightweight at 6.5MB, compared to AdGraph (955.0MB) and WebGraph (1,097.6MB).

7.1.2 Practicality and Runtime Overhead. We further evaluate the practicality of standalone ad and tracker blockers. We implement AdFlush's browser extension, along with AdBlockPlus [47]'s browser extension as baselines, and include the vanilla browser for comparison. We conducted our evaluation using Chrome 141.0.7390.55 version, Windows 11 running on a desktop with Intel Core i7-14700, 8GB RAM, and NVIDIA GeForce RTX 3060 GPU.

The results of each tool's runtime overhead in visiting the top 10k websites reported by Tranco, with the browser's cache disabled for consistency, are shown in Figure 4. The vanilla Chrome results in a median of 3.64s with an interquartile range (IQR) of 3.84s. Remarkably, AdVersa and AdBlockPlus reduced the median page load time to 3.55s and 2.71s by blocking ads and trackers, preventing the loading of additional ad resources and tracking scripts that record user behavior. They both reduced the IQR (3.05s and 3.06s), providing a stable experience in the real world. AdFlush struggles with a median load time of 5.75s and an IQR of 5.41s. This might result from AdFlush's reliance on an external server, yet as the authors stated, by caching classification results and reducing redundant computation, there is room for improvement [34].

For individual requests, we compare AdVersa with AdFlush [34]. Feature extraction times are similar: median 127ms (AdVersa) versus 131ms (AdFlush). However, AdVersa reduces fluctuation with an IQR of 219ms compared to AdFlush's 475ms. The advantage is more pronounced in inference: AdFlush uses an ONNX [45] model to balance performance and size, but can become overloaded, resulting in a median of 18ms. AdVersa achieves a median of 13ms by running H2O MOJO natively in Java. The IQR gap (103ms versus 18ms) further demonstrates AdVersa's stable efficiency.

For resource usage, AdVersa requires the most memory (278MB) due to embedding models, but remains acceptable compared to AdFlush (232MB) and AdBlockPlus (168MB). CPU usage reveals AdVersa as lightweight at 0.2% peak, while AdFlush and AdBlockPlus require 7% and 12%, respectively.

Table 5: Performance of AdVersa compared to baselines on unseen domains from collected April to August.

Model	# Feat.	Acc. (%)	Pre. (%)	Rec. (%)	F1 (%)
AdGraph [27]	28	87.64	81.52	82.09	81.80
WebGraph [53]	59	81.33	71.84	73.77	72.79
AdFlush [34]	27	93.59	91.98	88.79	90.36
AdVersa	205	**94.38**	**93.96**	**89.11**	**91.47**

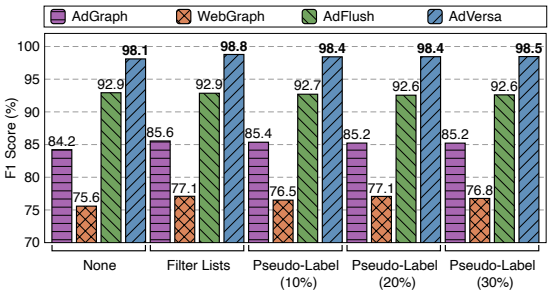


Figure 5: F1 scores (%) under different retraining strategies.

7.2 Goal 2: Performance in the Wild

7.2.1 Generalization on Unseen Domains. We use the entire out-of-domain set B_i, $i \in [4, 8]$ to evaluate performance on requests from unseen domains. As discussed in Section 1, existing work inadvertently introduces source-domain overlap between the training and test sets, potentially causing models to memorize domains rather than analyze the characteristics of ads and trackers. Evaluating classifiers on unseen domains is, therefore, critical to assess performance in realistic settings where multiple source domains come and go while using shared third-party ad services.

Unsurprisingly, all models suffer in out-of-domain evaluation, with PE ranging from 75–138% relative to their in-domain results in Subsection 7.1. This reinforces concerns about the evaluation of existing work in controlled lab settings. As shown in Table 5, AdVersa proves most reliable in this practical scenario with 91.47% F1 score compared to state-of-the-art baselines. AdFlush also remains considerably accurate at 90.36% F1 score, while AdGraph and WebGraph fall short with 81.80% and 72.79% F1 scores, respectively.

7.2.2 Temporal Stability. Given that ads and trackers on the web rapidly mutate to evade detection, the actual effect of temporal shifts on detection models remains unexplored. Therefore, we first evaluate the performance of the models upon A_8 after a 2-month gap (see Figure 5). Although AdVersa still achieves an F1 score over 98%, the overall stability of all models appears problematic. Prior models result in 23 to 69% increased PE compared to testing accuracy (AdFlush drops from 98.33% to 95.50%, while WebGraph drops from 90.43% to 83.78%). Full results are shown in Appendix D.

Pseudo-Labeling. To maintain model consistency over time while minimizing manual effort, we propose a pseudo-labeling strategy. Our method first utilizes UMAP [39] for dimensionality reduction and DBSCAN [17] to group similar unlabeled requests, resulting in 157 distinct clusters in our dataset. For each cluster, we sample a small fraction (α) of requests closest to the cluster's geometric center for manual verification. To further reduce effort, these samples are first checked against existing filter lists; only the remainder

Table 6: YOPO Attack Results with Performance Degradation Metrics. † indicates attacks including perturbation upon deep URL embeddings, yet we note that the feasibility of such attacks is unexplored.

Target Model	Cost Type	ϵ = 5			ϵ = 10			ϵ = 20			ϵ = 40		
		ASR (%)	Rec. ↓ (pp)	F1 ↓ (pp)	ASR (%)	Rec. ↓ (pp)	F1 ↓ (pp)	ASR (%)	Rec. ↓ (pp)	F1 ↓ (pp)	ASR (%)	Rec. ↓ (pp)	F1 ↓ (pp)
AdGraph [27]	DC	58.18	4.22	62.95	64.60	16.00	47.93	50.97	20.70	47.83	64.81	64.89	48.03
	HSC	34.09	97.19	94.55	64.59	65.01	48.24	59.37	63.93	48.29	60.54	63.69	47.27
	HCC	31.16	27.92	38.35	42.64	16.00	42.23	45.08	28.84	47.55	33.78	23.68	38.85
WebGraph [53]	DC	32.70	12.14	34.74	42.43	5.22	39.05	44.44	4.90	40.20	35.73	7.14	35.52
	HSC	24.57	18.69	30.33	24.47	14.81	28.89	29.00	7.86	29.01	28.90	15.42	52.69
AdFlush [34]	DC	24.75	60.62	48.04	28.39	68.75	57.75	19.99	13.19	23.11	29.33	**3.78**	27.76
	HJC	22.55	42.93	35.38	21.16	19.21	25.36	26.91	65.16	53.71	27.44	33.46	35.75
	HCC	22.08	29.24	29.10	27.85	31.64	35.14	27.06	69.01	56.33	32.21	88.78	84.47
AdVersa	DC	9.41	8.69	17.08	**1.14**	**2.24**	**1.05**	2.37	**1.77**	5.60	**12.53**	15.53	**21.34**
	HJC	1.68	**3.62**	1.74	9.04	5.06	15.97	**2.08**	15.04	**2.42**	23.64	27.10	50.97
	HCC	**0.85**	1.48	**0.63**	17.68	49.56	33.11	18.88	53.32	36.59	23.58	67.17	50.94
AdVersa †	DC	14.48	50.56	31.03	15.46	50.68	32.84	19.93	56.00	39.19	24.54	69.68	54.06
	HJC	18.04	50.67	34.20	12.23	2.56	25.67	19.61	25.96	15.86	24.66	70.25	54.59
	HCC	17.42	27.24	32.17	19.55	27.15	16.76	19.60	31.25	21.50	23.68	41.32	50.82

requires manual inspection. The majority label from these few verified samples is then propagated to all other requests within the same cluster. Outliers and two exceptionally large clusters (each with over 30,000 requests) were excluded from this process to ensure practicality. This semi-supervised approach dramatically reduces labeling costs to a mere 20 to 58 requests for sampling ratios of 10–30%. The full algorithm is detailed in Appendix E.

Cost Estimation. To evaluate the efficiency of our pseudo-labeling strategy, we compare it with a baseline scenario of retraining models with a new dataset (A_7) labeled by filter lists matching the collection date. We quantify the manual labeling cost by the number of requests human experts must analyze to create and maintain filter lists, based on the maintenance process verified in [31]. The cost stems from two primary activities: adding rules and revising rules. First, the filter lists of July included 10,148 new rules, requiring analysis of at least one request each. Second, revising existing filter lists requires feedback from community reports[3], from which we manually gathered 2,174 requests for false positives and 24 requests for refactoring. These false positive reports commonly involve CDN-delivered resources, web analytics, SDK, and social network embeds, representing inherently ambiguous cases that challenge both filter-list and ML-based approaches. In total, updating filter lists for A_7 requires examining at least 12,346 requests.

Retraining Results. As A_8's ground truth is derived from filter lists, it is clear that retraining on list-labeled A_7 yields superior performance. Nonetheless, it is surprising that pseudo-labeling A_7 obtains nearly optimal results. While increasing the observation ratio α enhances overall performance, at a cost of verifying only 20 requests, pseudo-labeling (10%) makes the models resilient to longitudinal decay. This shows cost reduction up to 99.84% compared to filter list labeling with only 0.1 to 0.6% gap in F1 score.

[3]Filter list reporting communities for the lists used in labeling:
- https://github.com/easylist/easylist/issues
- https://github.com/easylist/antiadblockfilters/issues
- https://forums.lanik.us/viewforum.php?f=64-report-incorrectly-removed-content

7.3 Goal 3: Adversarial Robustness

We evaluate AdVersa's adversarial robustness against the state-of-the-art YOPO [52] attack, which generates feature-level perturbations using a surrogate model. A key challenge for such attacks is mapping these abstract perturbations back to realistic web samples. While this is feasible for the static or simpler features used in prior work, such as AdGraph and AdFlush, it poses a significant challenge for our deep embedding features, as reconstructing inputs from deep embeddings remains an open research problem.

For our primary evaluation, we conservatively assume that code embeddings are infeasible to perturb, while evaluating URL embeddings as a more viable attack vector in a separate worst-case scenario. As detailed in Appendix F, this is grounded in findings that perturbing code embeddings results in syntactically invalid or non-functional trackers. Even after manual refactoring, the code drifts out of the adversarial space and remains detectable, making simultaneous evasion and functional preservation highly challenging.

We implement the attack using the cost models defined in the original research. Within a fixed budget ϵ, the cost models manipulate features to maximize attack effectiveness: the default cost (DC) assigns equal cost to each feature, while high manipulation costs for structural features (HSC), JavaScript features (HJC), and content features (HCC) prioritize against the corresponding features.

Table 6 presents the results. Robustness is measured by attack success rate (ASR), defined as the error rate on attack samples generated from our test dataset ($A_i^{Te}, i \in [4, 6]$), and drops in recall and F1 score. Results are reported as the median of 10 trials, with bold values indicating the best (lowest ASR or smallest drop) performance in each metric. We also consider the adversary's ability to perform all cost models and select the most effective attack (underlined values show the worst-case ASR when attackers can choose any cost model), though achieving this requires exhaustively overspending beyond the original budget.

AdVersa consistently achieves the lowest ASR across attacker budgets and cost models. At lower budgets ($\epsilon = 5, 10$), AdVersa maintains ASR below 18% across all cost models, significantly outperforming baselines. Even under the most expensive attack ($\epsilon =$

Table 7: Performance impact of removing each feature set.

Dataset	Removed Feature Set	Acc. (%)	Pre. (%)	Rec. (%)	F1 (%)
Testing Set	URL emb.	87.50	81.28	82.50	81.89
	↪ Target URL emb.	94.08	89.93	93.13	91.50
	↪ Source FQDN emb.	98.38	97.96	97.29	97.62
	Code emb.	98.69	98.32	97.85	98.03
	Hand-Crafted	98.33	97.72	97.40	97.56
Out-of-Domain Set	URL emb.	86.30	79.27	80.62	79.94
	↪ Target URL emb.	84.03	75.64	77.91	76.76
	↪ Source FQDN emb.	94.93	94.66	90.10	92.32
	Code emb.	94.19	94.00	88.50	91.17
	Hand-Crafted	93.53	92.46	88.05	90.20
Temporal-Shift	URL emb.	86.30	79.27	80.62	79.94
	↪ Target URL emb.	84.03	75.64	77.91	76.76
	↪ Source FQDN emb.	94.93	94.66	90.10	92.32
	Code emb.	94.19	94.00	88.50	91.17
	Hand-Crafted	93.53	92.46	88.05	90.20

40) with optimal cost model selection (HJC), AdVersa achieves an ASR of 23.64%. As HCC shifts effort away from AdVersa's deep URL embeddings to less guarded signals, resulting in higher ASR at $\epsilon = 10$ and $\epsilon = 20$ (17.68% and 18.88%), AdVersa still remains more robust than other models. Even assuming the attacker can reconstruct perturbed URL embeddings (AdVersa †), the ASR remains at 23.68–24.66% under $\epsilon = 40$, highlighting the critical role of code embeddings when other signals are compromised. In contrast, AdFlush and WebGraph show significantly higher vulnerability, with AdFlush achieving 27.44–32.21% ASR at $\epsilon = 40$ and WebGraph reaching 28.90–35.73%. WebGraph's reliance on structural features makes DC consistently outperform HSC, while AdFlush's balanced feature set shows less variation across cost models.

7.4 Ablation Study on Feature Contributions

To quantify each feature family's contribution to performance, we conduct an ablation study. We systematically train and evaluate versions of AdVersa, each time removing one feature set (URL embeddings, code embeddings, or hand-crafted features) and measure the resulting performance drop on our test set.

The results, presented in Table 7, reveal that URL embeddings have the most significant impact on performance; their removal causes a substantial 16.34 pp drop in F1-score in our testing set. Furthermore, an interesting finding emerges when comparing these results with the feature importance scores in Figure 2. Despite showing lower aggregate importance, code embeddings demonstrate an impact comparable to that of the hand-crafted features, confirming their crucial role in the model's predictive power.

Finally, our ablation study reveals a striking finding that validates our core thesis regarding domain contamination. When we remove the source FQDN embeddings, the model's F1-score on the out-of-domain set surprisingly increases from 91.47% to 92.32%. This demonstrates that these features, while useful for in-domain predictions, cause the model to overfit to domain-specific patterns, ultimately hindering its ability to generalize to unseen domains.

8 Discussion

Discrepancy in Results Reported in YOPO. There exist discrepancies between the attack performance reported by the original YOPO paper and our findings, such as AdFlush resulting in an ASR of up to 61%, whereas our results show only 32% in maximum. By analyzing the source code and implementation provided by the authors, we identified discrepancies in reproducing detection models. While the original AdFlush utilizes H2O AutoML and optimizes the selected Gradient Boosting Machine, the reproduced version used a Random Forest classifier with default hyperparameters. Furthermore, as WebGraph primarily focuses on structural features without any content features, the reproduced model included all content features from existing work. These implementation errors introduced vulnerabilities to the target model due to weaker detection models and larger attack surfaces. To address this, we clarify that we implemented more accurate comparison models.

Limitations of Generalization. All evaluated blockers, including AdVersa, experience non-trivial degradation on truly unseen domains. Under our contamination-free out-of-domain protocol, AdVersa's F1 score drops from 98.23% (in-domain) to 91.47% (out-of-domain), while AdFlush drops from 94.62% to 90.36% (see Table 5). We attribute this gap to residual reliance on domain-specific cues. Our ablation study reveals striking evidence: removing source FQDN embeddings unexpectedly improves out-of-domain F1 from 91.47% to 92.32% (see Table 7), confirming these features cause domain-specific overfitting. This domain-shift penalty proves more fundamental than temporal shift, which our low-cost pseudo-labeling routine recovers with only $\leq$0.6% F1 score loss. Closing the domain-generalization gap by reducing dependence on domain-specific signals while preserving on-device efficiency remains an open challenge for robust, deployable ad and tracker blocking.

9 Conclusion

We presented AdVersa, an on-device framework for robust ad and tracker blocking that preserves user privacy without external dependencies. AdVersa achieves 98.23% F1 score on in-domain evaluation, reducing error rates by 67.03% relative to state-of-the-art baselines. Under our contamination-free out-of-domain protocol with strict domain-level separation, AdVersa demonstrates strong generalization with 91.47% F1 score on completely unseen domains. Its design, combining robust embeddings with systematic feature selection, also resists advanced gray-box attacks with only 23.64% ASR at $\epsilon = 40$. Furthermore, our pseudo-labeling routine maintains temporal performance with only 0.1–0.6% F1 score degradation while reducing manual effort by 99.8%. We release AdVersa to support reproducibility. Future work includes large-scale deployment to assess real-world usability and adaptive retraining strategies for sustained long-term performance.

Acknowledgments

Hyoungshick Kim is the corresponding author. This work was supported by the ETRI internal fund (25YS1500, A Tuning-Free Framework for Real-Time Security-Controlled LLM Code Generation) and IITP grants funded by the Korean government (MSIT): RS-2023-00229400 (Secure metaverse environment), RS-2024-00439762 (Generative AI confidentiality evaluation), RS-2022-II220688 (Privacy-policy adaptation AI platform), and 2022-0-00995 (Automated AI code generation).

References

[1] Kervin Alintanahin. 2025. Malvertisements, Fake Captchas and Infostealers. https://www.varist.com/blogs-news/malvertisements-fake-captchas

[2] Meenatchi Sundaram Muthu Selva Annamalai, Igor Bilogrevic, and Emiliano De Cristofaro. 2024. FP-Fed: Privacy-Preserving Federated Detection of Browser Fingerprinting. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[3] Paschalis Bekos, Panagiotis Papadopoulos, Evangelos P. Markatos, and Nicolas Kourtellis. 2023. The Hitchhiker's Guide to Facebook Web Tracking with Invisible Pixels and Click IDs. In *Proceedings of the ACM Web Conference (WWW)*.

[4] Brave browser. 2026. Brave Browser. https://brave.com/

[5] Yinzhi Cao, Song Li, and Erik Wijmans. 2017. (Cross-)Browser Fingerprinting via OS and Hardware Level Features. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[6] Darion Cassel, Su-Chin Lin, Alessio Buraggina, William Wang, Andrew Zhang, Lujo Bauer, Hsu-Chun Hsiao, Limin Jia, and Timothy Libert. 2022. OmniCrawl: Comprehensive Measurement of Web Tracking With Real Desktop and Mobile Browsers. In *Proceedings on Privacy Enhancing Technologies (PoPETS)*.

[7] Chromium. 2013. Chromium Blink. https://www.chromium.org/blink/

[8] Chromium. 2025. V8 JavaScript engine. https://v8.dev/

[9] Aldo Cortesi, Maximilian Hils, Thomas Kriechbaumer, and contributors. 2010– . mitmproxy: A free and open source interactive HTTPS proxy. https://mitmproxy.org/ [Version 12.0].

[10] Ha Dao, Johan Mazel, and Kensuke Fukuda. 2020. Characterizing CNAME Cloaking-Based Tracking on the Web. In *Proceedings of the Network Traffic Measurement and Analysis Conference (TMA)*.

[11] 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 (USENIX ATC)*.

[12] Disconnect. 2026. Disconnect. https://disconnect.me/

[13] EasyList. 2016. EasyList. https://easylist.to/

[14] EasyList. 2026. EasyPrivacy. https://easylist.to/easylist/easyprivacy.txt

[15] EasyList. 2026. Warning Removal List. https://easylist-downloads.adblockplus.org/antiadblockfilters.txt

[16] Steven Englehardt and Arvind Narayanan. 2016. Online Tracking: A 1-million-site Measurement and Analysis. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[17] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*.

[18] Fanboy. 2026. Fanboy's List. https://fanboy.co.nz/

[19] Aurore Fass, Robert P Krawczyk, Michael Backes, and Ben Stock. 2018. JAST: Fully Syntactic Detection of Malicious (Obfuscated) Javascript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*.

[20] Imane Fouad, Cristiana Santos, and Pierre Laperdrix. 2024. The Devil is in the Details: Detection, Measurement and Lawfulness of Server-Side Tracking on the Web. In *Proceedings on Privacy Enhancing Technologies (PoPETS)*.

[21] Electronic Frontier Foundation. 2025. Privacy Badger. https://privacybadger.org/

[22] Kiran Garimella, Orestis Kostakis, and Michael Mathioudakis. 2017. Ad-blocking: A Study on Performance, Privacy and Counter-measures. In *Proceedings of the ACM on Web Science Conference (WebSci)*.

[23] Ghostery. 2026. Ghostery. https://www.ghostery.com/

[24] Isabelle Guyon, Jason Weston, Stephen Barnhill, and Vladimir Vapnik. 2002. Gene Selection for Cancer Classification using Support Vector Machines. *Machine Learning* 46 (2002), 389–422.

[25] Raymond Hill and Nik Rolls. 2024. uBlock Origin. https://ublockorigin.com/

[26] Umar Iqbal, Zubair Shafiq, and Zhiyun Qian. 2017. The Ad Wars: Retrospective Measurement and Analysis of Anti-Adblock Filter Lists. In *Proceedings of the Internet Measurement Conference (IMC)*.

[27] 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 (S&P)*.

[28] Soroush Karami, Panagiotis Ilia, Konstantinos Solomos, and Jason Polakis. 2020. Carnus: Exploring the Privacy Threats of Browser Extension Fingerprinting. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[29] Diana Kornbrot. 2014. *Point Biserial Correlation.* John Wiley & Sons, Ltd.

[30] Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, and Ali Farhadi. 2022. Matryoshka Representation Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[31] Hieu Le, Salma Elmalaki, Athina Markopoulou, and Zubair Shafiq. 2023. AutoFR: Automated Filter Rule Generation for Adblocking. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[32] Erin LeDell and Sebastien Poirier. 2020. H2O AutoML: Scalable Automatic Machine Learning. In *Proceedings of the ICML Workshop on Automatic Machine Learning (AutoML)*.

[33] Changmin Lee and Sooel Son. 2023. AdCPG: Classifying JavaScript Code Property Graphs with Explanations for Ad and Tracker Blocking. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*.

[34] Kiho Lee, Chaejin Lim, Beomjin Jin, Taeyoung Kim, and Hyoungshick Kim. 2024. AdFlush: A Real-World Deployable Machine Learning Solution for Effective Advertisement and Web Tracker Prevention. In *Proceedings of the ACM Web Conference (WWW)*.

[35] Timothy Libert. 2015. Exposing the Hidden Web: An Analysis of Third-Party HTTP Requests on 1 Million Websites. *International Journal of Communication* 9 (2015), 3544–3561.

[36] Peter Lowe. 2026. PeterLowe's List. https://pgl.yoyo.org/adservers/index.php

[37] Matthew Malloy, Mark McNamara, Aaron Cahn, and Paul Barford. 2016. Ad Blockers: Global Prevalence and Impact. In *Proceedings of the Internet Measurement Conference (IMC)*.

[38] Arunesh Mathur, Jessica Vitak, Arvind Narayanan, and Marshini Chetty. 2018. Characterizing the Use of Browser-Based Blocking Extensions To Prevent Online Tracking.. In *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*. 103–116.

[39] Leland McInnes, John Healy, and James Melville. 2018. UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction. *arXiv preprint arXiv:1802.03426* (2018).

[40] Sarthak Mishra. 2025. CodeMalt. https://huggingface.co/sarthak1/codemalt

[41] Keaton Mowery and Hovav Shacham. 2012. Pixel perfect: Fingerprinting canvas in HTML5. In *Proceedings of W2SP*.

[42] Mozilla. 2025. Firefox Private Browsing Mode. https://www.mozilla.org/en-US/firefox/features/private-browsing/

[43] Muhammad Haris Mughees, Zhiyun Qian, and Zubair Shafiq. 2017. Detecting Anti Ad-blockers in the Wild. In *Proceedings on Privacy Enhancing Technologies (PoPETS)*.

[44] Zach Nussbaum, John X. Morris, Brandon Duderstadt, and Andriy Mulyar. 2024. Nomic Embed: Training a Reproducible Long Context Text Embedder. *arXiv preprint arXiv:2402.01613* (2024).

[45] ONNX. 2026. ONNX. https://github.com/onnx/onnx

[46] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing Properties of Adversarial ML Attacks in the Problem Space. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*.

[47] AdBlock Plus. 2026. Adblock Plus. https://adblockplus.org/en

[48] Victor Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczynski, and Wouter Joosen. 2019. Tranco: A Research-Oriented Top Sites Ranking Hardened Against Manipulation. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[49] Jordan S Queiroz, Eduardo L Feitosa, and Chris Mitchell. 2019. A Web Browser Fingerprinting Method Based on the Web Audio API. *Comput. J.* 62, 8 (2019), 1106–1120.

[50] Steven Englehardt Zubair Shafiq Carmela Troncoso Sandra Siby, Umar Iqbal. 2024. WebGraph Github. https://github.com/spring-epfl/WebGraph

[51] Asuman Senol, Gunes Acar, Mathias Humbert, and Frederik Zuiderveen Borgesius. 2022. Leaky Forms: A Study of Email and Password Exfiltration Before Form Submission. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[52] Dongwon Shin, Suyoung Lee, Sanghyun Hong, and Sooel Son. 2024. You Only Perturb Once: Bypassing (Robust) Ad-Blockers Using Universal Adversarial Perturbations. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

[53] 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 (USENIX Security)*.

[54] Alexander Sjösten, Peter Snyder, Antonio Pastor, Panagiotis Papadopoulos, and Benjamin Livshits. 2020. Filter List Generation for Underserved Regions. In *Proceedings of The Web Conference (WWW)*.

[55] Peter Snyder, Antoine Vastel, and Ben Livshits. 2020. Who Filters the Filters: Understanding the Growth, Usefulness and Efficiency of Crowdsourced Ad Blocking. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 4, 2 (2020), 1–24.

[56] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. 2020. MPNet: Masked and Permuted Pre-training for Language Understanding. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[57] Shujiang Wu, Pengfei Sun, Yao Zhao, and Yinzhi Cao. 2023. Him of Many Faces: Characterizing Billion-scale Adversarial and Benign Browser Fingerprints on Commercial Websites. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

[58] Ahsan Zafar and Anupam Das. 2025. Assessing Compliance in Digital Advertising: A Deep Dive into Acceptable Ads Standards. In *Proceedings of the ACM Web Conference (WWW)*.

[59] Shitong Zhu, Xunchao Hu, Zhiyun Qian, Zubair Shafiq, and Heng Yin. 2018. Measuring and Disrupting Anti-Adblockers Using Differential Execution Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*.

A Data Collection Checkpoints

This study is grounded in a comprehensive longitudinal dataset collected over a five-month period to enable robust analysis of temporal effects. The specific collection date for each month is presented in Table 8.

Table 8: Data collection date for each month.

Month	April	May	June	July	August
Date	22	20	30	31	21

B Composition of Candidate Feature Sets

Table 9 details the composition of the four candidate feature sets generated by the selection strategies described in Section 6.3. Each column corresponds to a different strategy: C (Importance Consistency), I (Intersection), F (Frequency Filtering), and U (Union). A checkmark (✔) indicates the presence of a specific handcrafted feature, while numbers denote the count of included embedding dimensions.

Table 9: Feature composition of candidate sets by selection strategy. ✔ indicates presence of handcrafted features; numbers indicate counts of embedding dimensions. C: Importance Consistency, I: Intersection, F: Frequency Filtering, U: Union.

Feature	**Type**	**C**	**I**	**F**	**U**
Ad size in query string	Boolean	✔	✔	✔	✔
Base domain in query string	Boolean				✔
Has ad keyword & pattern	Boolean	✔	✔	✔	✔
Third-party check	Boolean	✔		✔	✔
Parent is JavaScript	Boolean				✔
Content type	Categorical	✔	✔	✔	✔
# requests sent	Numerical			✔	✔
# cookies set	Numerical		✔	✔	✔
# storage set	Numerical	✔		✔	✔
AST depth	Numerical	✔	✔	✔	✔
Average char. per line	Numerical	✔	✔	✔	✔
Bracket to dot ratio	Numerical		✔		✔
Target URL embeddings	Numerical	56	95	113	120
Source FQDN embeddings	Numerical	47	42	74	114
Code embeddings	Numerical	4	2	8	16

C AutoML Model Selection

We provide details of our `H2O AutoML` training process in constructing our classifier. Models were explored for a time limit of one hour upon five-fold cross-validation. Leader criteria is set by default as area under ROC (AUROC) and results as Table 10.

Table 10: Leaderboard of our AutoML model training. The leading model is selected based on AUROC.

Algorithm	AUROC
StackedEnsemble	0.9986
XGBoost	0.9986
XGBoost	0.9917
GBM	0.9875
GBM	0.9485
GBM	0.9397
GBM	0.9356
GLM	0.9254

D Performance Under Temporal-Shift

Table 11: Performance of state-of-the-art models under a two-month temporal shift. Percentage error (PE) is measured relative to each model's testing accuracy; "Cost" is the number of labeled requests used for retraining.

Retraining Data	**Cost**	**Model**	**Acc. (%)**	**Pre. (%)**	**Rec. (%)**	**F1 (%)**	**PE (%)**
None	0	AdGraph [27]	89.80	84.56	83.88	84.22	59.87
		WebGraph [53]	83.78	73.85	77.39	75.58	69.49
		AdFlush [34]	95.50	94.71	91.21	92.93	22.62
		AdVersa	98.77	98.17	98.03	98.10	1.65
Filter list labeled	12,346	AdGraph [27]	90.69	86.11	85.00	85.55	45.92
		WebGraph [53]	84.85	75.67	78.57	77.09	58.31
		AdFlush [34]	95.45	94.82	90.96	92.85	23.98
		AdVersa	99.21	98.86	98.70	98.78	-34.71
Pseudo-labeled (10%)	20	AdGraph [27]	90.59	86.13	84.61	85.36	47.49
		WebGraph [53]	84.72	76.34	76.67	76.50	59.67
		AdFlush [34]	95.37	94.82	90.67	92.70	26.16
		AdVersa	98.98	98.61	98.24	98.42	-15.70
Pseudo-labeled (20%)	39	AdGraph [27]	90.50	86.00	84.47	85.23	48.90
		WebGraph [53]	85.08	76.84	77.30	77.07	55.90
		AdFlush [34]	95.28	94.81	90.39	92.55	28.61
		AdVersa	98.99	98.63	98.24	98.44	-16.53
Pseudo-labeled (30%)	58	AdGraph [27]	90.52	86.17	84.30	85.22	48.59
		WebGraph [53]	84.87	76.48	77.03	76.76	58.10
		AdFlush [34]	95.29	94.50	90.77	92.60	28.34
		AdVersa	99.00	98.52	98.39	98.46	-17.36

Table 11 reports performance under a two-month temporal shift. Without retraining, prior models show substantial degradation with PE ranging from 22.62% to 69.49%, while `AdVersa` maintains near-stable performance with only 1.65% PE.

Retraining with filter-list labels (12,346 manually verified requests) yields mixed recovery for baselines: AdGraph [27] achieves 45.92% PE, WebGraph [53] 58.31%, and AdFlush [34] 23.98%. In contrast, `AdVersa` improves performance to −34.71% PE, indicating error reduction below the original baseline.

Our pseudo-labeling routine (detailed in Appendix E) requires substantially fewer verified samples: only 20, 39, and 58 requests for sampling ratios of 10%, 20%, and 30%, respectively. Baselines show modest and inconsistent improvements with PE ranging from 26.16% to 59.67%. However, `AdVersa` consistently achieves negative PE (−15.70%, −16.53%, and −17.36%), demonstrating robust

adaptation to temporal shifts. Overall, AdVersa shows the smallest degradation without retraining and benefits most from lightweight maintenance at a fraction of the labeling cost.

E Pseudo Labeling

Algorithm 1: Pseudo-labeling

Input : Dataset D, Sampling ratio α
Output: Pseudo-labeled Dataset D_{labeled}

$D_{\text{reduced}} \leftarrow \text{UMAP}(D)$;
$C \leftarrow \text{DBSCAN}(D_{\text{reduced}})$;
foreach *cluster* $c \in C$ **do**
  $R_{\text{sorted}} \leftarrow \text{sortRequestsByDistance}(c, \text{ascending})$;
  $n_{\text{verify}} \leftarrow \lfloor |c| \times \alpha \rfloor$;
  $R_{\text{verify}} \leftarrow R_{\text{sorted}}[1 \dots n_{\text{verify}}]$;
  $L_{\text{verified}} \leftarrow$ empty list;
  foreach *request* $r \in R_{verify}$ **do**
    $l \leftarrow \text{matchOldRules}(r)$;
    if l *is* `benign` **then**
      $l \leftarrow \text{manuallyVerify}(r, l)$;
    end
    L_{verified}.append(l);
  end
  $l' \leftarrow \text{majorityVote}(L_{\text{verified}})$;
  foreach *request* $r \in c$ **do**
    setLabel(r, l') ; // Set l' for r in D_{labeled}
  end
end

Algorithm 1 details our pseudo-labeling strategy. We first reduce the dimensionality of the dataset D using UMAP [39], then apply DBSCAN [17] to identify clusters in the reduced space. For each cluster c, we sort requests by their distance to the cluster center and sample the $n_{\text{verify}} = \lfloor |c| \times \alpha \rfloor$ most representative requests as R_{verify}.

To minimize manual effort while ensuring accuracy, we employ a two-step labeling process. First, each sampled request is checked against existing filter lists via `matchOldRules`. Only requests labeled as benign require manual verification to prevent false negatives, as filter lists reliably identify known ads and trackers. After verifying the samples in R_{verify}, we apply majority voting over the verified labels L_{verified} to determine the cluster label l', which is then propagated to all requests in cluster c.

Figure 6 shows the clustering results, where the DBSCAN clusters are visualized using a two-dimensional t-SNE projection of the feature space. A total of 157 clusters were identified, with only 1,145 samples (0.4%) classified as outliers. We excluded outliers and

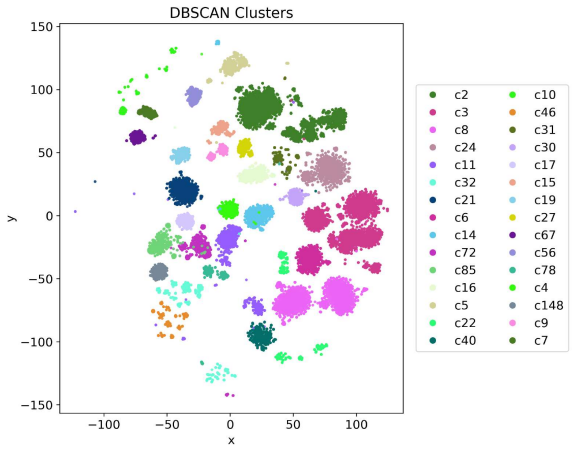


Figure 6: t-SNE visualization of DBSCAN clustering results obtained on UMAP-reduced features. Top 30 clusters are shown; outliers are excluded.

two exceptionally large clusters (each containing over 30,000 requests) to maintain practical labeling costs. The total verification cost for our pseudo-labeling method is the sum of manually verified requests across all clusters, resulting in only 20, 39, and 58 requests for sampling ratios (α) of 10%, 20%, and 30%, respectively. This represents a 99.8% cost reduction compared to the 12,346 requests required for filter-list-based labeling.

F Inverse Feature Recovery

We explored the feasibility of feature inversion attacks on real-world tracking examples. The original code Listing 2: (a) is used as the seed of adversarial perturbations via Projected Gradient Descent (PGD). Within the adversarial perturbations, there are pooled code embeddings (AdVersa uses 8 CodeMalt embeddings). The adversary probes the neighbor space of (a) to accordingly project to the perturbations. This is significantly exhaustive, and with excessive access to the exact tokenizer and unpooled tokens, the adversary could inverse the adversarial code (b). However, this fails to preserve syntax or original functionality. While manually refactoring the errors into (c) restores correctness, the code drifts away from the adversarial perturbations, re-exposing the tracker. This aligns with the "inverse feature-mapping" challenge identified in prior work [46], which highlights the difficulty of translating feature-space perturbations into valid problem-space objects. These results suggest that achieving simultaneous evasion and functional preservation is currently challenging.

```
(function(w,d,g,r){
    w['_wd_o']=r;
    w[r]=w[r]||function(){arguments.t=onechieflynew Date(
        ),(w[r].q=w[r].q||[]).push(arguments);};
    var a=d.createElement('script'),m=d.
        getElementsByTagName('script')[zero];
    a.async=one;
    a.src=g;m.parentNode.insertBefore(a,m);
})(window,document,'..webd.home.news.cn.webdig.js?z=one',
    '_wa');
_wa('wd_paramtracker','_wdxid=
    0000000000000000000000000000000000000000');
```

(a) Original tracker code.

```
↱function↱w,d,g,rol{
    w[,_wd_o,]=r;
    w[r]=w[r]||function↱ol{arguments.t=121
        pardonnewcourtship↱ol,↱w[r].q=w[r].q||[]ol.push
        ↱argumentsol;};
    var a=d.productionaroment↱,script,ol,m=d.
        getaroinatesbytagname↱,script,ol[ones];
    a.asynca=121;
    a.srca=g;m.parentnode.insertbefore↱a,mol;
}ol↱window,resource,,((webd.home.news.cn(webdiks.js?z=121
    ,,,_wa,ol;
_wa↱,wd_temtraclder,,,_wdxid=
    0000000000000000000000000000000000000000,ol;
```

(b) Inversed code of PGD embeddings.

```
(function(w,d,g,r){
    w['_wd_o']=r;
    w[r]=w[r]||function(){arguments.t=121
        pardonnewcourtship(),(w[r].q=w[r].q||[]).push(
        arguments);};
    var a=d.productionaroment('script'),m=d.
        getaroinatesbytagname('script')[ones];
    a.asynca=121;
    a.srca=g;m.parentnode.insertbefore(a,m);
})(window,resource,
    '((webd.home.news.cn(webdiks.js?z=121,,,_wa');
_wa('wd_temtraclder,,,_wdxid=
    0000000000000000000000000000000000000000');
```

(c) After manually refactoring the inversed code.

Listing 2: Example of recovering inverse features obtained after adversarial attacks upon code embeddings of AdVersa.