ShortcutCatcher: Making Traffic Classification Reliable

ShortcutCatcher: Making Traffic Classification Reliable

YUQI ZHAO, Politecnico di Torino, Italy
MATTEO BOFFA, Politecnico di Torino, Italy
LUCA VASSIO, Politecnico di Torino, Italy
MARCO MELLIA, Politecnico di Torino, Italy

Machine learning has given encrypted traffic classification a new momentum. Yet, once deployed, models often fail due to hidden shortcut features, i.e., spurious correlations learned from training data that do not hold in new environments. Prior work has shown their negative impact through costly manual intervention. Here, we present ShortcutCatcher, an automated, model-agnostic framework that detects and mitigates shortcuts with the help of explainable AI. The key idea is to contrast model behaviour on two datasets: a large training dataset and a separate verification dataset that differs in scenario but shares the same feature schema. ShortcutCatcher integrates feature explanation with cross-scenario evaluation in a closed loop, iteratively removing those critical features that would not be valid in deployment. Across multiple encrypted traffic classification tasks and model architectures, ShortcutCatcher uncovers shortcut dependencies and improves cross-scenario generalisation, up to three times over standard training. In addition, ShortcutCatcher exposes dataset limitations where collection artefacts act as silent shortcuts that have gone so far unnoticed, allowing us to finally expose realistic performance without assuming that the underlying task is intrinsically easy[1].

CCS Concepts: • **Security and privacy → Network security**.

Additional Key Words and Phrases: Traffic Classification, Machine Learning, Generalisation

ACM Reference Format:
Yuqi Zhao, Matteo Boffa, Luca Vassio, and Marco Mellia. 2026. ShortcutCatcher: Making Traffic Classification Reliable. *Proc. ACM Netw.* 4, CoNEXT2, Article 23 (June 2026), 15 pages. https://doi.org/10.1145/3808671

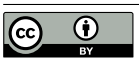
1 Introduction

Traffic classification remains a central building block for network operations [11, 24–26, 33]. With the widespread adoption of encrypted protocols [23, 37], recent work has turned to machine learning (ML) models trained on flow- or packet-level features extracted from encrypted traffic. Although these models often achieve high accuracy under standard train/test splits, their performance degrades sharply when deployed in different environments [41]. This fragility stems from reliance on *shortcuts* – spurious correlations tied to specific data collection settings rather than actionable features [14]. Examples include timestamp patterns, IP addresses, measurement artefacts, or OS-specific behaviours. Such shortcuts remain hidden in conventional pipelines based on single-environment data, yet dominate model decisions and fail when conditions change.

Prior work has exposed ML pitfalls in traffic analysis and cybersecurity, including shortcut reliance [6, 12, 38, 41]. Some approaches [16] iteratively mitigate shortcuts using human-in-the-loop procedures. On the one hand, this requires repeated expert intervention for each model

[1]Code and data available at: https://github.com/SmartData-Polito/ShortcutCatcher

Authors' Contact Information: Yuqi Zhao, yuqi.zhao@polito.it, Politecnico di Torino, Torino, Italy; Matteo Boffa, matteo.boffa@polito.it, Politecnico di Torino, Torino, Italy; Luca Vassio, luca.vassio@polito.it, Politecnico di Torino, Torino, Italy; Marco Mellia, marco.mellia@polito.it, Politecnico di Torino, Torino, Italy.



ACM 2834-5509/2026/6-ART23
https://doi.org/10.1145/3808671

and iteration, limiting scalability across datasets and deployment contexts. On the other hand, some work showed that even humans may fail [41]. Our work automatically translates shortcut identification into corrective action without domain expertise. Addressing this gap is important: automation enables scalability across many models and shortcuts, and provides a principled starting point that reduces manual effort even when expert oversight is retained.

We present ShortcutCatcher, a model-agnostic framework to automatically detect and mitigate shortcut features. The key idea is to compare model behaviour across two datasets: a *training dataset* collected in one environment and a *verification dataset* from a different scenario with the same feature space and task. ShortcutCatcher builds on meta-learning principles: it uses explainable AI (xAI) [7, 34] and cross-scenario evaluation to identify important training features, then tests their impact on verification performance. Features whose removal improves cross-scenario accuracy are flagged as shortcuts. At each iteration, the framework (1) trains the model, (2) identifies important features via xAI, and (3) removes the feature whose elimination most improves verification accuracy. This process repeats until no further gains are observed, progressively eliminating shortcuts and promoting robust features.

ShortcutCatcher offers three key benefits: 1) it automatically exposes and handles shortcut features without domain expertise; 2) it directly optimises for cross-scenario generalisation rather than in-scenario accuracy; 3) it is compatible with classical models (RandomForest (RF)), modern neural architectures (Tabular Foundation Model (TabICL) [30]).

We evaluate ShortcutCatcher on traffic classification tasks in controlled experiments using four open datasets, with both flow- and packet-level predictions. Across all scenarios, it consistently improves performance, with accuracy gains exceeding 3×.

However, these gains should be interpreted with caution. By removing shortcut features, ShortcutCatcher exposes the true difficulty of encrypted traffic classification: once the model is forced to rely only on features that genuinely generalise across scenarios, we observe the real model performance, reflecting the fundamental limitation of the traffic classification task itself. ShortcutCatcher therefore provides the actual performance one can expect in real-world scenarios.

More broadly, our results highlight a systemic issue: widely used open benchmarks – even when presented as challenging for generalisation[19] – contain scenario-specific artifacts that silently act as shortcuts (e.g., TCP timestamps and incorrect TCP Checksum that reflect collection setup limits). ShortcutCatcher systematically identifies these limitations and lets us sanitise datasets.

Overall, this work shows that feature-level shortcut mitigation can be automated, practical, and highly effective, providing a principled path toward more robust deployment of ML solutions in realistic deployment environments.

2 Motivations and Related Work

Traffic Classification and ML: Traffic classification aims to infer labels of a network flow or packet based solely on packet traces.[2] Formally, given a sample s characterised by features $\mathcal{F}(s)$ extracted from a single-packet (*packet-classification*) or N-packets (*flow-classification*), the task is to learn a function $g : s \mapsto y$ that maps s to a label y. The label y can be an application [19], service type [20], or protocol [15]. In this context, ML methods – and particularly ML classifiers – appear as natural candidates to approximate g. Also, as payloads are increasingly encrypted [23, 37] and thus unusable for classification, modern ML-based classifiers rely entirely on protocol headers and behavioural features, such as packet timing and size [11, 24–26, 33].

Distrust in ML Traffic Classification: While ML has achieved remarkable success in many domains [10, 32], networking researchers have coined the term 'credibility crisis' to describe its

[2]We follow the classic 5-tuple definition of flows and consider bidirectional flows, i.e., bi-flows.

shortcomings in networking applications [39]: models that perform well during development often fail once deployed, struggling to generalise beyond their training conditions. Several works have investigated the causes. Arp et al. examined how pipeline-level *pitfalls* can artificially inflate test performance, proposing *Dos and Don'ts* for ML in computer security [1]. Others focus on *dataset biases* – collection artifacts that introduce *spurious correlations* between inputs and labels [16]. For instance, Wickramasinghe et al. showed that several encrypted traffic datasets contain a large fraction of unencrypted flows [38], while Flood et al. demonstrated that many intrusion detection models exploit dataset-specific artifacts – such as *timestamp* ranges or *acknowledgment numbers* – rather than genuine behavioural patterns [12]. This over-reliance on spurious correlations, known as *shortcut learning*, affects even large, state-of-the-art architectures [41] and leads to overly optimistic results that collapse outside the training setting [1, 14].

Regaining Trust - xAI and Challenging Scenarios: Mitigating shortcut learning has become a vital step for network practitioners to regain trust on ML. Two primary research directions have emerged: *explainable AI (xAI)* and the design of *challenging training scenarios*.

• *Explainable AI (xAI)* is widely used to analyse model behaviour and uncover shortcut dependencies [22]. Prior work shows that surrogate models such as decision trees can expose brittle decisions in black-box classifiers [16], while concept-based [27] and hybrid explanations [8] help diagnose distribution shifts. However, xAI remains diagnostic rather than prescriptive: it reveals *what* a model relies on, but not *whether* that reliance is harmful or *how* to correct it, still requiring expert intervention [13, 35, 36].

• *Challenging training scenarios* aim to enforce generalisation by designing evaluation settings that better reflect deployment. This includes avoiding data leakage (e.g., per-packet splits [41]) and adopting *time*- and *space*-based splits [1, 3, 28]. While effective at exposing non-robust models, these approaches provide limited guidance for improving them and often require substantial domain knowledge, as datasets may still contain hidden shortcuts.

Our Contribution: ShortcutCatcher leverages the strengths of both xAI to build an automatic shortcut identification and mitigation system. By introducing an independent split – the *verification set* – which is representative of the model deployment scenario, to stress-test the model generalisation. With it, we create a shortcut identification loop: at each iteration, ShortcutCatcher tries removing one top-ranked feature and observes how performance changes in the verification set. If it improves, that feature is flagged as a shortcut and pruned. This differs from traditional feature selection algorithms that look for the most important features in the training set. ShortcutCatcher instead looks for those features that are meaningless in the verification set. ShortcutCatcher provides a principled approach that significantly reduces the manual effort involved. We validate ShortcutCatcher on public datasets, showing it identifies both known and previously unreported shortcuts.

3 ShortcutCatcher

We propose a general loop to automatically detect and mitigate shortcuts. We summarise our methodology in Figure 1. The framework is model- and task-agnostic and can be applied to other scenarios than traffic classification.

3.1 ShortcutCatcher overview

In ShortcutCatcher, we assume access to two independent, identically distributed datasets: a standard *training set*, large enough for training a machine-learning model; and a much smaller *verification set* used to assess the quality of the learned patterns. In each iteration, xAI returns the top-k most important features according to the current model. To detect potential shortcuts, we remove each of these features, retrain the model, and evaluate it in the verification set, choosing the

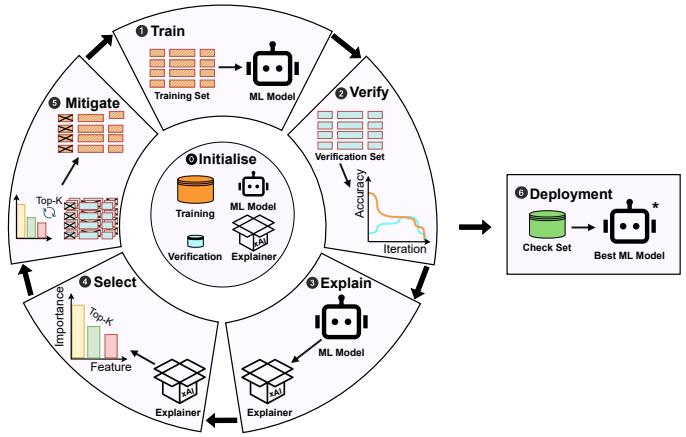


Fig. 1. **ShortcutCatcher overview**. Phases 1-5 mitigate bad features. The best model is deployed.

feature whose removal yields the greatest performance improvement for the desired metric. Despite its apparent importance during training, this feature harms generalisation in the verification set, a characteristic pattern of shortcut behaviour. The loop continues until the performance in the verification set stabilises, i.e., until all shortcuts have been eliminated.

• **Phase 0 – Initialization**: We begin with a training dataset $\mathcal{D}$ with feature set $\mathcal{F} = (f_1, f_2, \ldots, f_d)$ and label y. We assume that $\mathcal{F}$ contains both legitimate features F_{good}, and illegitimate features F_{bad}, which exhibit only spurious correlations with y within $\mathcal{D}$. We introduce the verification dataset $\mathcal{V}$, collected independently from $\mathcal{D}$ (e.g., at a different time, in the deployment scenario), but sharing the same feature schema $\mathcal{F}$. Ideally, only the distribution of the legitimate features F_{good} in $\mathcal{V}$ matches that in $\mathcal{D}$. Consequently, a model seeking a robust mapping from $\mathcal{F} \rightarrow y$ must rely primarily on F_{good} while disregarding F_{bad}.

We complete the initialisation stage by selecting a model g and an explainer X.[3]

• **Phase 1 – Train**: In this step, we perform a standard supervised training of g on the dataset $\mathcal{D}$, following ML best practices, i.e., splitting $\mathcal{D}$ into train, validation and test sets. Note that at this stage, the model will learn from both F_{good} and F_{bad}, likely yielding strong but unrealistic performance. This occurs because both the training and test sets are drawn from the same dataset $\mathcal{D}$, and share the same spurious correlations, F_{bad}.

• **Phase 2 – Verify**: In this phase, we evaluate the trained model in the verification set $\mathcal{V}$. Ideally, if the model has learned to rely on F_{good}, its performance in $\mathcal{V}$ should be comparable to its test performance in Phase 1. However, as proved in previous literature [16, 39, 41], if the model relies on F_{bad}, its performance does not transfer to $\mathcal{V}$.

• **Phase 3 – Explain**: We use X to investigate the decision-making process of the model, i.e., assign an importance score to each feature to reflect how much the predictions rely on that feature.

• **Phase 4 – Select**: We identify the Top-k most important features to verify. Notably, this is the stage at which previous approaches [35] typically required human experts to manually inspect these Top-k features and detect potential shortcuts.

[3]Explainers are typically coupled to the model type (e.g., *Feature Impurity*[2] for Random Forests; *Permutation Methods*[4] for Neural Networks). However, any combination that enables the estimation of the model's most important features is suitable at this stage.

• **Phase 5 – Mitigate**: We iterate over the Top-k features and progressively try removing each one from the feature set $\mathcal{F}$. For each candidate removal, we train again g on the reduced feature set and evaluate its performance on $\mathcal{V}$. We pick the feature among the top-k that maximises the desired metric on $\mathcal{V}$ and remove it forever from the training set before proceeding to the next iteration. By doing that, we determine which features, when removed, force the model to rely more heavily on F_{good}. While iterating, we record the best model $BestModel$ and $Best\mathcal{F}$.

• **Phase 6 – Deployment**: $BestModel$ is the model that best perform on $\mathcal{V}$. To properly test it, we use a third independent *check set* $\mathcal{C}$ to simulate real-world deployment.

3.2 ShortcutCatcher Algorithm

Algorithm 1: Pseudo-code of ShortcutCatcher.

Input: Origin set $\mathcal{D}$, Verification set $\mathcal{V}$, Feature set $\mathcal{F}$, Explainer $X()$, Number of features to evaluate k
Output: Model g, Feature set $\mathcal{F}$

1: $g \leftarrow Train(\mathcal{D}), CurrMetric \leftarrow Evaluate(g, \mathcal{V}), BestMetric \leftarrow CurrMetric$;
2: **while** $\mathcal{F} \neq \emptyset$ **do**
3: $\quad \mathcal{W} \leftarrow X(g, \mathcal{D}), \mathcal{K} \leftarrow$ Top-$k(\mathcal{W}, k), BestGain \leftarrow -\infty$;
4: $\quad$ **for** $f \in \mathcal{K}$ **do**
5: $\quad\quad \hat{\mathcal{D}} \leftarrow Remove(\mathcal{D}, f)$;
6: $\quad\quad \hat{g} \leftarrow Train(\hat{D}), Acc \leftarrow Evaluate(\hat{g}, \mathcal{V}), Gain \leftarrow Acc - CurrMetric$;
7: $\quad\quad$ **if** $Gain > BestGain$ **then**
8: $\quad\quad\quad BestGain \leftarrow Gain, CandidateFeature \leftarrow f$;
9: $\quad\quad$ **end**
10: $\quad$ **end**
11: $\quad \mathcal{D} \leftarrow Remove(\mathcal{D}, CandidateFeature), \mathcal{F} \leftarrow Remove(\mathcal{F}, CandidateFeature)$;
12: $\quad g \leftarrow Train(\mathcal{D}), CurrMetric \leftarrow Evaluate(g, \mathcal{V})$;
13: $\quad$ **if** $CurrMetric > BestAcc$ **then**
14: $\quad\quad BestMetric \leftarrow CurrMetric, BestModel \leftarrow g, Best\mathcal{F} \leftarrow \mathcal{F}$
15: $\quad$ **end**
16: **end**
17: **return** $BestModel, Best\mathcal{F}$;

We translate the loop into the pseudo-code in Algorithm 1. The algorithm maintains four components: the training dataset $\mathcal{D}$, the verification dataset $\mathcal{V}$, the feature set $\mathcal{F}$, and the Top-k candidate features examined at each iteration. In each step, we train h in $\mathcal{D}$ and evaluate it on $\mathcal{V}$ (lines 1,12). The explainer X computes feature importances $\mathcal{W}$, from which we extract the Top-k features $\mathcal{K}$. In the mitigation phase (lines 4–10), for each $f \in \mathcal{K}$, we remove f to obtain $\hat{\mathcal{D}}$, retrain a temporary model $\hat{h}$, and evaluate it on $\mathcal{V}$. We select the feature whose removal yields the largest gain, and remove it from both $\mathcal{D}$ and $\mathcal{F}$. If performance improves, we update the best model and feature set. The process repeats until no features remain.

3.3 Computational Complexity

In the main loop, the algorithm examines all features for $|\mathcal{F}|$ iterations. Each iteration requires (i) one execution of the explainer X, and (ii) k train g on $\mathcal{D}$ and evaluate on $\mathcal{V}$. Let T_X denote the cost of executing the explainer X and T_g the cost of a single train–evaluate cycle of g. The overall worst-case time complexity of the algorithm is therefore $O\big(|\mathcal{F}|\,(T_X + k\,T_g)\big)$. The hyper-parameter k controls the trade-off between computational cost and selection robustness: larger values of k increase the probability of correctly identifying the best feature to remove at each iteration, at the

expense of a linear increase in the number of full training cycles. For comparison, an exhaustive search of all feature subsets would require $O\left(2^{|\mathcal{F}|}\, T_g\right)$ evaluations, and is often infeasible in practice.

As the computational cost grows with $|\mathcal{F}|$, running ShortcutCatcher to feature exhaustion may become impractical for large feature sets. In practice, dominant shortcuts typically emerge in early iterations – as we consistently observe in Section 5. Therefore, a small budget of iterations $B \ll |\mathcal{F}|$ already suffices to remove the most harmful features.

3.4 Relation to Feature Selection Methods

ShortcutCatcher resembles wrapper-based feature selection [18] and Recursive Feature Elimination (RFE) in its outer structure, but pursues a fundamentally different goal. RFE removes features the model *ignores* – noisy or redundant ones – to maximise in-distribution performance on the training set. ShortcutCatcher removes features the model *exploits* during training but that harm cross-scenario generalisation, i.e., in the verification set. This reversal matters: standard RFE with permutation importance on $\mathcal{D}$ discards features the current model does not use and instead focuses on potential shortcuts. ShortcutCatcher avoids this by eliminating features that are important on $\mathcal{D}$ and harmful on $\mathcal{V}$. At last, domain knowledge driven heuristics exclude known volatile fields, but, as we will show, even domain-informed decisions can miss subtle collection artefacts that ShortcutCatcher spots automatically.

4 Experimental Setup

In this section, we describe the datasets, classification tasks, and evaluation methodology used to validate our framework. Rather than collecting new data, we rely on widely used open datasets that have become de-facto benchmarks in encrypted traffic classification. The usage of open datasets gives us the flexibility to create different scenarios while maintaining full control over the experimental setup and guaranteeing reproducibility.

4.1 Dataset splitting

We present our strategy to split the whole dataset $\mathcal{A}$ into $\mathcal{D}$, $\mathcal{V}$ and C, as sketched in Figure 2.

Main split: First, we follow the *per-flow split* that is fundamental to avoid data leaks [41], i.e., we guarantee all packets of a flow belong to the same split.

Given flow samples $s(t)$ collected at time t, let $\mathcal{A} = \{s(t)\}$ be the original dataset. We need to split it into $\mathcal{D}$, $\mathcal{V}$ and C. A natural way to achieve this is to consider a *time-split* strategy, i.e., for each label l, let

$$\mathcal{D}^l = \{s(t) \mid t \le t^l_{train}, y(s) = l\}, \quad \mathcal{D} = \cup_l \mathcal{D}^l$$
$$\mathcal{V}^l = \{s(t) \mid t^l_{train} < t \le t^l_{ver}, y(s) = l\}, \quad \mathcal{V} = \cup_l \mathcal{V}^l$$
$$C^l = \{s(t) \mid t > t^l_{ver}, y(s) = l\}, \quad C = \cup_l C^l$$

where $t_{train}(l) < t_{ver}(l)$ are per-class boundaries that consider the data collection may span over different periods.[4] In short, the per-time split simulates the scenario where the model is trained on past data, verified on current data, and checked/deployed in the future.

Trainset $\mathcal{D}$ split: For the ML model training, $\mathcal{D}$ must be further split into a train, validation, and test sets.[5] We consider two possible strategies here:

- A) Stratified-split: samples in $\mathcal{D}$ are randomly assigned to each set, ensuring each class gets the same fraction of samples in each subset. We refer to the stratified split as $\mathcal{D}_s$ in the following.

[4]We enforce that all packets of a flow respect the constraints, and drop flows whose packets extend over the split boundaries.
[5]The test set is not strictly needed. We use it to correctly evaluate the performance of g in $\mathcal{D}$.

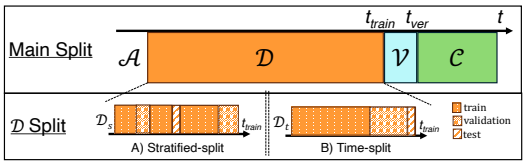


Fig. 2. **Visualisation of splitting strategy**. For each class l, we split flows in $\mathcal{A}$ using t_{train} and t_{ver} to get $\mathcal{D}$, $\mathcal{V}$ and C. We split $\mathcal{D}$ using simple stratified- ($\mathcal{D}_s$) or time-split ($\mathcal{D}_t$) into train, validation and test sets.

- B) Time-split: For each class l, we split $\mathcal{D}^l$ respecting the flow time as done with the main split. We enforce a stratified policy, too. We refer to the time-split as $\mathcal{D}_t$ in the following.

Past works adopted the classic stratified sampling strategy. We apply the time-split strategy as a possible safeguard against shortcuts related to time-related features during training.

4.2 Dataset modification

To simulate the use case where one would like to deploy a model in a different network and at a different time, we modify the IP addresses and the timestamp in the verification and check sets. We define a *forge()* mechanism to emulate this use case. It implements:

IP address forging: we apply a consistent replacement policy: given an IP address, we randomly sample a replacement from the 128.66.0.0/16 subnet and consistently replace all occurrences.

Timestamp forging: TCP Timestamp [17] carry time information that is *relative* to the client and server bootstrap time. The time-split policy cannot guarantee that the TCP timestamp's relative nature creates a potential data leak due to artefacts in the data collection process. To avoid this, we forge TCP timestamps by adding a random offset to flows in $\mathcal{V}$ and C.[6]

After having modified the protocol headers, we recompute the IP and TCP checksums accordingly.

We have two use cases. A *Time Shift* scenario, where the deployment occurs in the same network at a later time ($\mathcal{V}_t = \mathcal{V}$ and $C_t = C$); and a *Time and Space shift* scenario, where the deployment occurs in a different network at a later time ($\mathcal{V}_{t,s} = forge(\mathcal{V})$ and $C_{t,s} = forge(C)$).

4.3 Scenario definition

Having defined two strategies for splitting $\mathcal{D}$ and two use cases for deployment, we have a total of four possible scenarios from which we consider:

(1) Scenario $S_0 \leftarrow \mathcal{D}_s, \mathcal{V}_t$: Apply traditional stratified-split to $\mathcal{D}$, and simple time-split to $\mathcal{V}$.
(2) Scenario $S_1 \leftarrow \mathcal{D}_t, \mathcal{V}_t$: Apply time-split to both $\mathcal{D}$ and $\mathcal{V}$.
(3) Scenario $S_2 \leftarrow \mathcal{D}_t, \mathcal{V}_{t,s}$: Apply time-split to $\mathcal{D}$, and time and space split to $\mathcal{V}$

S_0 is the classic baseline where the model g is (over)optimised on $\mathcal{D}$. S_1 introduces the time-split in $\mathcal{D}$ as a safeguard against temporal features, which are good candidates for $\mathcal{F}_{bad}$. S_2 emulates real deployment cases where spatial and temporal features should be disregarded.

4.4 Task and feature definition

We consider two standard encrypted traffic classification tasks:

(1) *Packet-level classification*: given a single packet, predict its class.
(2) *Flow-level classification*: given the first N packets of a flow, predict its class.

[6]TCP Timestamp is an option in the TCP header that helps measure round-trip time (RTT) accurately; typically, they refer to the Timestamp Counter (TSC), which is reset at bootstrap.

Table 1. **Descriptions of the open datasets we use in the experiments**. The verification set $\mathcal{V}$ is $\sim 10\times$ smaller than the training set $\mathcal{D}$. We report the number of flows for all datasets, and packets for TLS120.

Dataset	Task	$\mathcal{D}$			$\mathcal{V}$	C	Classes
		Train	*Val*	*Test*			
TLS120 [20]	Packet Classification	230,146	10,710	328,665	17,022	328,665	120
VPN [15]	Flow Classification	840	90	240	60	240	6
App-Time [19]	Flow Classification	14,823	1,647	5,130	270	5,130	27
App-Ver [19]	Flow Classification	13,725	1,525	4,750	250	4,750	25

For all tasks, we extract features exclusively from IP, TCP, and UDP headers, ignoring payload data since it is expected to be encrypted and thus semantically uninformative. We intentionally retain all header fields, including those previously reported as potential shortcuts, since ShortcutCatcher is explicitly designed to identify and remove such problematic features. We provide the complete list of extracted fields in Table 3. Notice that we discard those that take constant values. For the IP address, we consider each octet to be a separate feature, i.e., IP(1).IP(2).IP(3).IP(4).

5 Results

In the following, we report the results of ShortcutCatcher. First, we introduce the datasets and models we use; then, we comment on the performance of the scenarios we describe in Section 4.3.

5.1 Experiment Settings

Datasets: We summarise datasets in Table 1. For each dataset, we report the task, the number of samples for splits, and the number of classes.

- *VPN* [15] includes traffic from six service categories (Web, VoIP, Video Streaming, Chat, Email, P2P) generated using different applications, under VPN-encrypted connections.
- *App-Time* [19] contains network traces of 27 Android applications collected in a controlled environment to explicitly design challenging scenarios. Specifically, part of the dataset was collected a month after the other (we use this time-based split setting t_{train} accordingly).
- *App-Ver* [19]: contains two captures obtained from different versions of each app that we use as time-split.
- *TLS120* [20] contains encrypted web traffic generated by visiting 120 TLS 1.3-enabled websites. We use the cleaned traces and labels provided by Zhao et al. [41].

Models and Hyper-parameters: For all models, we compute the feature set $\mathcal{F}$ selecting all TCP/UDP/IP header fields – See Appendix A. We set $k = 5$ and accuracy as the metric to be optimised within ShortcutCatcher. For all datasets, we try 3 different ML models: a shallow Random Forest (RF), and two increasingly complex neural networks – 2-Layers Multi-Layer-Perceptron (MLP) and TabICL [31], a newly proposed tabular foundation model. In the sake of space, we report the neural networks' results in the Appendix B. We use AutoGluon v1.4 [9] to train each model and select the best hyper-parameters. We balance samples to handle class imbalance in train and validation sets, while we keep the real distribution for test, verification and check sets. When solving flow-level classification, we use the first five packets to represent flows.

Metrics: For each scenario, we report the initial macro f1-score in the test set $F1^0(\mathcal{D})$ and the check set $F1^0(C)$, i.e., the performance without running ShortcutCatcher. We compare them with the f1-score on the check set $F1^*(C)$ after running it, along with the relative percentage gain $\left(100 * \frac{F1^*(C) - F1^0(C)}{F1^0(C)}\right)$. We use the macro f1-score to take into account class imbalances.

Table 2. **RF results in different scenarios.** $F1^0(\mathcal{D})$ is training performance, $F1^0(C)$ and $F1^*(C)$ are initial and best deployment performance. In red the drop when using g on C. In green the ShortcutCatcher boost.

Dataset	S_0			S_1			S_2		
	$F1^0(\mathcal{D})$	$F1^0(C)$	$F1^*(C)$	$F1^0(\mathcal{D})$	$F1^0(C)$	$F1^*(C)$	$F1^0(\mathcal{D})$	$F1^0(C)$	$F1^*(C)$
VPN	88.0	11.0 (-87.5%)	43.7 (297%)	53.9	22.7 (-57.9%)	40.1 (76.7%)	53.9	11.4 (-78.8%)	36.4 (219%)
App-Time	75.6	55.8 (-27.1%)	59.3 (6.3%)	66.5	54.7 (-18.2%)	58.9 (7.7%)	66.5	17.3 (-74.0%)	26.5 (53.2%)
App-Ver	77.2	60.1 (-22.2%)	61.2 (1.8%)	70.5	58.7 (-16.7%)	58.9 (0.3%)	70.5	16.6 (-76.5%)	28.5 (71.7%)
TLS-120	72.6	69.2 (-4.7%)	69.2 (0%)	61.4	59.4 (-3.3%)	59.4 (0%)	61.4	9.5 (-84.5%)	20.5 (115%)

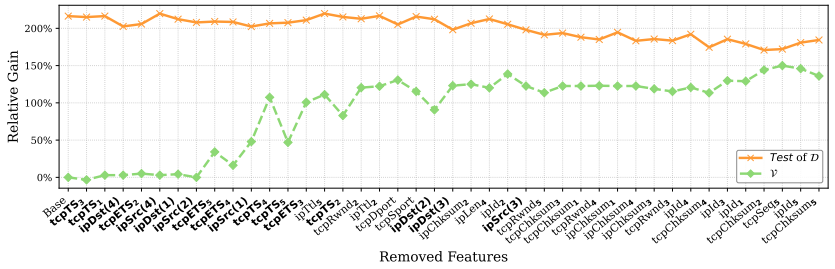


Fig. 3. ShortcutCatcher applied to the RF classifier. Flow classification task on VPN, scenario S_2.

5.2 ShortcutCatcher across scenarios

We show the results of the Random Forest in the three scenarios described in Section 4.3 in Table 2.

Consider the VPN dataset. The RF model faces a flow-classification task over encrypted traffic. Previous work [16] noted that `tcp-timestamp` can act as a shortcut. Our results confirm this by comparing the performance in $\mathcal{D}$ in S_0 and S_1. In S_0, we simply split $\mathcal{D}$ via stratification. Temporal cues boost $F1^0(\mathcal{D})$, but fail in $F1^0(C)$ (-87.5% drop). In S_1, we split $\mathcal{D}$ by time, mitigating temporal shortcuts. $F1^0(\mathcal{D})$ drops, and $F1^0(C)$ increase. Move now to S_2, where we additionally perturb spatial features in C to simulate an independent data collection. We observe a further performance decrease (-78.8%) when moving to the check set. The RF relies on spatial patterns (e.g., IP addresses) that no longer exist.

Now consider the results after applying ShortcutCatcher ($F1^*(C)$). Our loop constrains the RF to ignore the temporal features in S_0 and S_1, and the spatial ones in S_2, boosting the performance in the check set by an almost 3× factor (+292%) in S_0. As expected, the improvement is the strongest where the original RF relies the most on shortcuts (S_0 and S_2).

Figure 3 illustrates ShortcutCatcher's loop between $\mathcal{D}$ and $\mathcal{V}$ for S_2. We highlight the model's relative gain to $F1^0(\mathcal{V})$ at iteration i, i.e., $gain(\mathcal{S}, i) = \frac{F1^i(\mathcal{S}) - F1^0(\mathcal{V})}{F1^0(\mathcal{V})}$, $\mathcal{S} \in \{\mathcal{D}, \mathcal{V}\}$. Initially, the gap is large: the model trained on the original feature set depends on shortcuts and fails to generalise. Then ShortcutCatcher iteratively removes spatial (IP-based) and temporal (`tcp-timestamps`) features. Performance in $\mathcal{V}$ increases, while that in $\mathcal{D}$ decreases. Ultimately, the two curves converge, marking the performance a model would plausibly achieve when deployed.

Now consider the remaining datasets. Although the tasks become more challenging when enforcing a time-based split (S_0 vs S_1), temporal features are not the main drivers here. In fact, the App-Time and App-Ver authors carefully designed a temporal shortcut-free collection. The situation changes in S_2. Across all datasets, we observe a [75-85]% drop from $F1^0(\mathcal{D})$ to $F1^0(C)$. For TLS-120, IP addresses are crucial: the RF memorises addresses in the training data, and fails once these change at deployment. Under spatially consistent conditions (similar addresses at train and then at deployment), IP addresses provide a strong signal. However, under spatial shifts (e.g.,

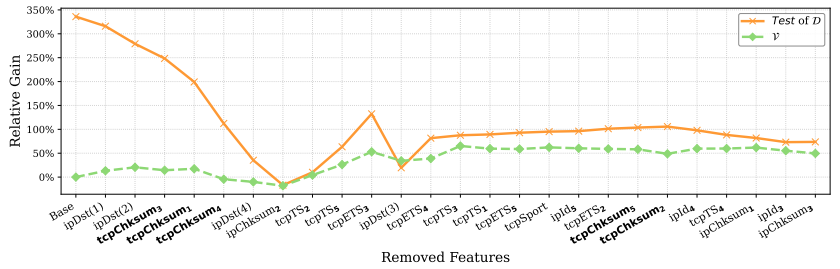


Fig. 4. ShortcutCatcher applied to the RF classifier. Flow classification task on App-Time, scenario S_2.

CDN or location changes, as in S_2), IP addresses in $\mathcal{D}$ act as scenario-specific shortcuts. Removing them forces the model to rely on more robust features.

More interestingly, in the App datasets, we uncover an unknown collection bias that effectively behaves as a shortcut. Looking at ShortcutCatcher evolution in Fig. 4, we observe that TCP Checksums are among the most important features to remove. This unveils a previously unknown data collection issue: authors collected data on clients where TCP checksum offloading was enabled: all packets within a flow share the same checksum. To the model, these checksums act as proxies for flow IDs. Coupled with the sequential per-class collection (one class after another), the range of flow IDs and timestamps directly maps to the class, echoing the issue discussed by Zhao et al. [41]. Our *forge()* spatial modification recomputes the checksums, so that they immediately appear as shortcuts. The final value $\mathcal{F}\infty^*(\mathcal{C}) = 26.5$ for App–Time, compared to the higher yet unrealistic $\mathcal{F}\infty'(\mathcal{D}) = 66.5$, illustrates the true difficulty of the task. Importantly, the feature retained after ShortcutCatcher pruning is semantically meaningful for the task: the IP length of the fourth packet of each flow, which reflects application-level design characteristics. We also report the computational time of ShortcutCatcher under this scenario in the Appendix D.

6 Limitations

While ShortcutCatcher provides an automated and effective approach to identifying and mitigating shortcut features, some limitations remain.

Dependence on the verification set. Our framework relies on the availability of a labelled verification set $\mathcal{V}$. Its effectiveness depends on how well $\mathcal{V}$ reflects the target deployment scenario. In practice, $\mathcal{V}$ should capture expected distribution shifts (e.g., temporal or spatial variations) and include a sufficient number of samples per class to ensure stable evaluation. In Appendix C we show that that even small verification sets can be informative, although we expect larger ones to improve robustness.

A key risk is the potential overfitting to a specific $\mathcal{V}$, yielding performance gains on that dataset without guaranteeing improved generalisation. Furthermore, if $\mathcal{V}$ still contains some valid features which may be shortcuts in general, these would not be removed.

A possible extension is a nested validation setup, where features are selected on one scenario and verified on a separate, disjoint dataset. Our current design partially follows this idea through the use of the check set $\mathcal{C}$, which is supposed to be in-distribution w.r.t. $\mathcal{V}$. $\mathcal{C}$ remains untouched during feature selection and is used only for final evaluation. Because feature removal is guided by performance on $\mathcal{V}$, the framework may still overfit. Potentially, one can extend ShortcutCatcher to consider multiple verification sets, each referring to different possible deployments, and then create a feature removal loop that generalises the performance on all of them.

In the use case of this paper, the verification set is synthetically generated through the *forge()* mechanism (Section 4.2) that simulates the perturbation of IP addresses and timestamps and emulates deployment shifts. While this enables controlled experiments, it focuses on well-known artefacts and may introduce some circularity, as it assumes prior knowledge of potential shortcut features. Although ShortcutCatcher can uncover previously unknown shortcuts (e.g., TCP checksum offloading), its effectiveness ultimately depends on the diversity and realism of available verification datasets. More broadly, the lack of datasets with natural spatio-temporal variation remains a limitation of the field.

Greedy feature removal. ShortcutCatcher adopts a greedy, one-feature-at-a-time removal strategy. While effective in practice, this approach cannot capture interactions between features, where combinations jointly act as shortcuts even if individual features appear benign. Consequently, higher-order dependencies may go undetected. Extending the framework to account for such interactions – e.g., through group-wise removal or Shapley-based importance methods [5, 21] – is an important direction for future work.

Mitigation Strategy. The current framework mitigates the shortcuts by removing features entirely once identified as harmful. We experimented with other options: injecting noise and permuting features – i.e., replacing their values with those from other samples [29, 40]. Empirically, the feature removal strategy performed the best. Refined strategies based on partial weighting or regularisation could provide a better trade-off and warrant further investigation.

7 Conclusion

This work introduced ShortcutCatcher, a model-agnostic framework that automates the detection and removal of shortcut features in encrypted traffic classification. By coupling feature-level explanations with cross-scenario verification, ShortcutCatcher systematically isolates spurious correlations and enforces reliance on features that generalise. Experiments across diverse datasets, models, and deployment settings show consistent and often substantial gains on deployed models, while exposing the intrinsic difficulty of encrypted traffic classification once shortcuts are removed. Beyond improving robustness, ShortcutCatcher reveals structural flaws in widely used benchmarks, highlighting the need for more reliable datasets. Overall, our results demonstrate that shortcut mitigation can be practical, automated, and effective, providing a principled path toward trustworthy ML-based traffic analysis.

8 Acknowledgements

This work was supported by the AI4CTI FISA under Project #FISA-2023-00168, funded by the Italian Ministry of University and Research (MUR). Yuqi Zhao has been supported by the China Scholarship Council (Grant No. 202306470001). Computational resources were provided by HPC@POLITO (https://hpc.polito.it).

Ethics

This work does not raise ethical issues.

Generative AI

Generative AI tools were used exclusively to assist in language refinement and do not affect the technical content of this work.

References

[1] Daniel Arp, Erwin Quiring, Feargus Pendlebury, Alexander Warnecke, Fabio Pierazzi, Christian Wressnegger, Lorenzo Cavallaro, and Konrad Rieck. 2022. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*. 3971–3988.

[2] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[3] Theo Chow, Mario D'Onghia, Lorenz Linhardt, Zeliang Kan, Daniel Arp, Lorenzo Cavallaro, and Fabio Pierazzi. 2025. Breaking Out from the TESSERACT: Reassessing ML-based Malware Detection under Spatio-Temporal Drift. *arXiv preprint arXiv:2506.23814* (2025).

[4] Ian Covert, Scott Lundberg, and Su-In Lee. 2021. Explaining by removing: A unified framework for model explanation. *Journal of Machine Learning Research* 22, 209 (2021), 1–90.

[5] Ian Covert, Scott M Lundberg, and Su-In Lee. 2020. Understanding global feature contributions with additive importance measures. *Advances in neural information processing systems* 33 (2020), 17212–17223.

[6] Alberto Dainotti, Antonio Pescape, and Kimberly C Claffy. 2012. Issues and future directions in traffic classification. *IEEE network* 26, 1 (2012), 35–40.

[7] Rudresh Dwivedi, Devam Dave, Het Naik, Smiti Singhal, Rana Omer, Pankesh Patel, Bin Qian, Zhenyu Wen, Tejal Shah, Graham Morgan, et al. 2023. Explainable AI (XAI): Core ideas, techniques, and solutions. *ACM computing surveys* 55, 9 (2023), 1–33.

[8] Abduarraheem Elfandi, Hannah Sagalyn, Ramakrishan Durairajan, and Walter Willinger. 2024. Bootstrapping trust in ML4Nets solutions with hybrid explainability. In *Proceedings of the 3rd Workshop on Practical Adoption Challenges of ML for Systems*. 1–5.

[9] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505* (2020).

[10] OpenAI et al. 2024. GPT-4 Technical Report. arXiv:2303.08774 [cs.CL] https://arxiv.org/abs/2303.08774

[11] Michael Finsterbusch, Chris Richter, Eduardo Rocha, Jean-Alexander Muller, and Klaus Hanssgen. 2013. A survey of payload-based traffic classification approaches. *IEEE Communications Surveys & Tutorials* 16, 2 (2013), 1135–1156.

[12] Robert Flood, Gints Engelen, David Aspinall, and Lieven Desmet. 2024. Bad design smells in benchmark nids datasets. In *2024 IEEE 9th European Symposium on Security and Privacy (EuroS&P)*. IEEE, 658–675.

[13] Felix Friedrich, Wolfgang Stammer, Patrick Schramowski, and Kristian Kersting. 2023. A typology for exploring the mitigation of shortcut behaviour. *Nature Machine Intelligence* 5, 3 (2023), 319–330.

[14] Robert Geirhos, Jörn-Henrik Jacobsen, Claudio Michaelis, Richard Zemel, Wieland Brendel, Matthias Bethge, and Felix A Wichmann. 2020. Shortcut learning in deep neural networks. *Nature Machine Intelligence* 2, 11 (2020), 665–673.

[15] Gerard Drapper Gil, Arash Habibi Lashkari, Mohammad Mamun, and Ali A Ghorbani. 2016. Characterization of encrypted and VPN traffic using time-related features. In *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP 2016)*. SciTePress Setúbal, Portugal, 407–414.

[16] Arthur S Jacobs, Roman Beltiukov, Walter Willinger, Ronaldo A Ferreira, Arpit Gupta, and Lisandro Z Granville. 2022. Ai/ml for network security: The emperor has no clothes. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. 1537–1551.

[17] Van Jacobson, Robert Braden, and Dave Borman. 1992. RFC1323: TCP extensions for high performance.

[18] Ron Kohavi and George H John. 1997. Wrappers for feature subset selection. *Artificial intelligence* 97, 1-2 (1997), 273–324.

[19] Xinjie Lin, Gang Xiong, Gaopeng Gou, Wenqi Dong, Jing Yu, Zhen Li, and Wei Xia. 2025. Respond to Change with Constancy: Instruction-tuning with LLM for Non-IID Network Traffic Classification. *IEEE Transactions on Information Forensics and Security* 20 (2025), 5758–5773.

[20] Xinjie Lin, Gang Xiong, Gaopeng Gou, Zhen Li, Junzheng Shi, and Jing Yu. 2022. Et-bert: A contextualized datagram representation with pre-training transformers for encrypted traffic classification. In *Proceedings of the ACM Web Conference 2022*. 633–642.

[21] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. *Advances in neural information processing systems* 30 (2017).

[22] Alfredo Nascita, Giuseppe Aceto, Domenico Ciuonzo, Antonio Montieri, Valerio Persico, and Antonio Pescapé. 2024. A survey on explainable artificial intelligence for internet traffic classification and prediction, and intrusion detection. *IEEE Communications Surveys & Tutorials* 27, 5 (2024), 3165–3198.

[23] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. 2014. The cost of the" s" in https. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. 133–140.

[24] Thuy TT Nguyen and Grenville Armitage. 2008. A survey of techniques for internet traffic classification using machine learning. *IEEE communications surveys & tutorials* 10, 4 (2008), 56–76.

[25] Fannia Pacheco, Ernesto Exposito, Mathieu Gineste, Cedric Baudoin, and Jose Aguilar. 2018. Towards the deployment of machine learning solutions in network traffic classification: A systematic survey. *IEEE Communications Surveys & Tutorials* 21, 2 (2018), 1988–2014.

[26] Eva Papadogiannaki and Sotiris Ioannidis. 2021. A survey on encrypted network traffic analysis applications, techniques, and countermeasures. *ACM Computing Surveys (CSUR)* 54, 6 (2021), 1–35.

[27] Sagar Patel, Dongsu Han, Nina Narodytska, and Sangeetha Abdu Jyothi. 2025. Agua: A concept-based explainer for learning-enabled systems. In *Proceedings of the ACM SIGCOMM 2025 Conference*. 329–346.

[28] Feargus Pendlebury, Fabio Pierazzi, Roberto Jordaney, Johannes Kinder, and Lorenzo Cavallaro. 2019. TESSERACT: Eliminating Experimental Bias in Malware Classification across Space and Time. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 729–746. https://www.usenix.org/conference/usenixsecurity19/presentation/pendlebury

[29] Gregory Plumb, Marco Tulio Ribeiro, and Ameet Talwalkar. 2021. Finding and fixing spurious patterns with explanations. *arXiv preprint arXiv:2106.02112* (2021).

[30] Jingang Qu, David Holzmüller, Gaël Varoquaux, and Marine Le Morvan. 2025. TabICL: A Tabular Foundation Model for In-Context Learning on Large Data. In *International Conference on Machine Learning*.

[31] Jingang Qu, David Holzmüller, Gaël Varoquaux, and Marine Le Morvan. 2025. TabICL: A Tabular Foundation Model for In-Context Learning on Large Data. In *International Conference on Machine Learning*.

[32] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*. PmLR, 8748–8763.

[33] Shahbaz Rezaei and Xin Liu. 2019. Deep learning for encrypted traffic classification: An overview. *IEEE communications magazine* 57, 5 (2019), 76–81.

[34] Gaith Rjoub, Jamal Bentahar, Omar Abdel Wahab, Rabeb Mizouni, Alyssa Song, Robin Cohen, Hadi Otrok, and Azzam Mourad. 2023. A survey on explainable artificial intelligence for cybersecurity. *IEEE Transactions on Network and Service Management* 20, 4 (2023), 5115–5140.

[35] David Steinmann, Felix Divo, Maurice Kraus, Antonia Wüst, Lukas Struppek, Felix Friedrich, and Kristian Kersting. 2024. Navigating shortcuts, spurious correlations, and confounders: From origins via detection to mitigation. *arXiv preprint arXiv:2412.05152* (2024).

[36] Stefano Teso and Kristian Kersting. 2019. Explanatory interactive machine learning. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*. 239–245.

[37] Martino Trevisan, Danilo Giordano, Idilio Drago, Marco Mellia, and Maurizio Munafo. 2020. Five Years at the Edge: Watching Internet From the ISP Network. *IEEE/ACM Transactions on Networking* 28, 02 (2020), 561–574.

[38] Nimesha Wickramasinghe, Arash Shaghaghi, Gene Tsudik, and Sanjay Jha. 2025. Sok: Decoding the enigma of encrypted network traffic classifiers. In *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1825–1843.

[39] Walter Willinger, Ronaldo A Ferreira, Arpit Gupta, Roman Beltiukov, Satyandra Guthula, Lisandro Z Granville, and Arthur S Jacobs. 2025. When Something Looks Too Good To Be True, It Usually Is! AI Is Causing A Credibility Crisis In Networking. *ACM SIGCOMM Computer Communication Review* 55, 1 (2025), 10–15.

[40] Shirley Wu, Mert Yuksekgonul, Linjun Zhang, and James Zou. 2023. Discover and cure: Concept-aware mitigation of spurious correlation. In *International Conference on Machine Learning*. PMLR, 37765–37786.

[41] Yuqi Zhao, Giovanni Dettori, Matteo Boffa, Luca Vassio, and Marco Mellia. 2025. The Sweet Danger of Sugar: Debunking Representation Learning for Encrypted Traffic Classification. In *Proceedings of the ACM SIGCOMM 2025 Conference* (São Francisco Convent, Coimbra, Portugal) *(SIGCOMM '25)*. Association for Computing Machinery, New York, NY, USA, 296–310. doi:10.1145/3718958.3750498

Appendix

A Extracted Feature Details

Table 3 shows the complete set of features. We consider all IP/TCP/UDP header fields, removing only those that assume constant values. For IP addresses, we represent them as 4 1-byte features, each referring to one octet, i.e., IP(0).IP(1).IP(2).IP(3). For flow classification tasks, we consider the first 5 packets of each bi-flow, independently of their direction.

Table 3. Packet fields selected for the model training. Features are extracted from raw traces using the Python Scapy package (https://scapy.net).

Protocol	Considered packet fields
IPv4	*Source and Destination addresses, Type of service, Internet Header Length, ID, Checksum, Flags, Length, Protocol, Version, TTL, Fragmentation*
UDP	*Source and Destination ports, Checksum, Length*
TCP	*Source and Destination ports, TSval, TSecr, Reserved Window, Urgent pointer, Data offset, Flags, SACK Checksum, Sequence and Acknowledgement numbers, Options*

B ShortcutCatcher and Neural Networks

On Table 4 shows results for the MLP and TabICL classifiers on S_2. Results are consistent and show how ShortcutCatcher can remove bad features, finally improving performance to realistic values in deployment. TabICL complexity prevents it from converging on TLS-120, questioning the usage of foundational models [41].

Table 4. ShortcutCatcher with MLP and TabICL, scenario S_2

Dataset	MLP			TabICL		
	$F1^0(\mathcal{D})$	$F1^0(C)$	$F1^*(C)$	$F1^0(\mathcal{D})$	$F1^0(C)$	$F1^*(C)$
VPN	53.2	21.2	21.1 (-0.5%)	58.9	4.9	26.5 (440.8%)
App-Time	64.7	8.3	22.1 (166.3%)	63.6	6.3	30.3 (381.0%)
App-Ver	67.9	12.5	25.0 (100.0%)	67.3	16.0	27.2 (70%)
TLS-120	57.0	2.6	15.8 (507.7%)	-	-	-

C Ablation study

Table 5 shows the impact of reducing the number of samples in the verification set $\mathcal{V}$ in the App-Ver setup. Providing 4-10 samples per class (100-250 samples in total with 25 classes) suffices to let ShortcutCatcher identify and remove temporal and spatial shortcuts. Repeating the experiment with five different random seeds to sample the points to put in $\mathcal{V}$ shows large variability when few samples are selected. For instance, we observe the best results with 100 samples (i.e., 4 samples for each class), but with a large variability. Selecting the right sample via active learning is a viable solution to balance diversity in $\mathcal{V}$ and limit the need for manual labelling.

Table 5. Ablation study on verification set size. Flow classification task on App-Ver, scenario S_2.

$\lvert\mathcal{V}\rvert$	**25**	**50**	**100**	**200**	**250**
$F1^*(C)$	27.1 ± 6.6	27.9 ± 5.7	31.2 ± 6.6	28.6 ± 2.3	28.5 ± 0.0

Table 6. Ablation study on the number of features k to evaluate at each iteration of ShortcutCatcher. Flow classification task on App-Time and TLS-120, scenario S_2, $F1^*(C)$ reported.

k	**1**	**3**	**5**	**7**
App-Time	17.3 (0%)	30.9 (78.4%)	26.5 (52.6%)	25.6 (47.6%)
TLS-120	10.4 (10.0%)	20.4 (115%)	20.5 (116%)	20.5 (116%)

Table 6 shows the impact of k, the number of candidate features ShortcutCatcher considers for removal (Phase 5 - Mitigate). Choosing $k = 1$ is not effective since ShortcutCatcher is forced to remove the most important feature, which may be an actual good feature. Making $k > 1$ allows the algorithm to choose the best-to-remove feature among the most important ones. Ideally, one would choose among all features (i.e., $k = |\mathcal{F}|$ at each iteration), at the cost of growing the training time. In practice, $k \in \{3, 5, 7\}$ offers a good tradeoff, speeding up ShortcutCatcher execution.

D Computational Time Example

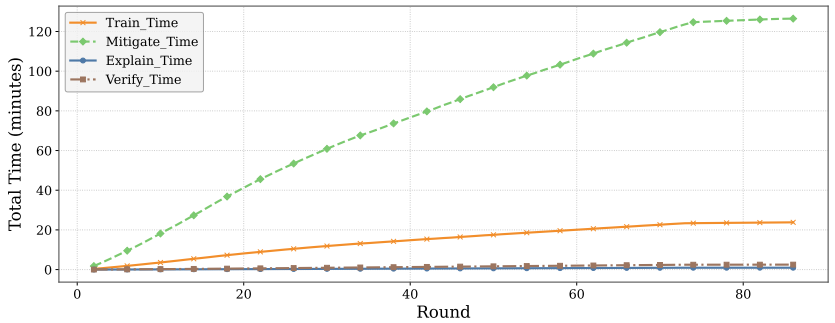


Fig. 5. Cumulative computational time cost for ShortcutCatcher. Same scenario with Fig 4.

Fig. 5 reports the cumulative runtime of each phase in the same scenario as the one in Fig. 4. We run ShortcutCatcher until feature exhaustion (86 rounds), for a total of ≈ 2.5 hours. The Mitigate phase (green curve) dominates the overall cost: since it retrains the model $k = 5$ times per round, its cumulative time is consistently ≈ 5× that of Train phase (orange curve). Both curves are concave, as the per-round cost decreases with the shrinking feature set. Explain (blue curve) and Verify (brown curve) are negligible in comparison – the latter being a single inference pass over the trained model. Overall, the full pipeline completes within a few hours on a single machine, making ShortcutCatcher practical as a one-shot auditing step before deployment.

Received December 2025; accepted April 2026