Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Feature Engineering Is the Bottleneck: An Explainability Study of ML-Based Security-Relevant Method Detection

Oshando Johnson

oshando.johnson@iem.fraunhofer.de

Fraunhofer Institute for Mechatronic Systems Design

Rohith Kumar Shanmuganathan

Carl von Ossietzky Universität Oldenburg

Ranjith Krishnamurthy

Paderborn University

Goran Piskachev

Amazon (United States)

Eric Bodden

Paderborn University

Research Article

Keywords: explainable AI, SHAP, AutoML, software security, security-relevant methods, taint analysis

Posted Date: November 12th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10897881/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

Feature Engineering Is the Bottleneck: An Explainability Study of ML-Based Security-Relevant Method Detection

Oshando Johnson [1*], Rohith Kumar Shanmuganathan [1,2], Ranjith Krishnamurthy [1,3], Goran Piskachev [4], Eric Bodden [1,3]

[1*]Fraunhofer IEM, Paderborn, Germany.
[2]Carl Von Ossietzky Universität Oldenburg, Oldenburg, Germany.
[3]Heinz Nixdorf Institute, Paderborn University, Paderborn, Germany.
[4]Amazon Web Services, New York, United States.

*Corresponding author(s). E-mail(s):
oshando.johnson@iem.fraunhofer.de;
Contributing authors: rohith.shanmuganathan@uol.de;
ranjithk@hni.uni-paderborn.de; gpiskach@amazon.de;
eric.bodden@uni-paderborn.de;

Abstract

Static Application Security Testing tools trace untrusted data flows using curated lists of Security-Relevant Methods (SRMs), namely sources, sinks and sanitizers. Machine learning approaches for SRM detection (ML4SRM) report strong cross-validation results yet perform far worse on unseen code, and prior work has not asked why. To assess this disparity we develop WEAVE, a tool combining automated model selection, repeated cross-validation and explainable Artificial intelligence (XAI) using SHapley Additive exPlanations (SHAP) for WEKA/MEKA pipelines. Using WEAVE and SRM-Dataset, our consolidated collection of 10,180 Java SRMs, we conduct an explanation study of three ML4SRM approaches, namely, Dev-Assist, SSCM, and SWAN. Global and local explanations show that the models rely on identifier tokens and parameter signatures, barely on the method body, so they learn dependencies among surface features specific to the training data. On unseen code the models read those same features differently, with 43 of 59 feature group and model combinations shifting significantly, by up to 62 percentage points, and 14 pointing the opposite way for most methods. Nothing inside the models changes. An attribution is a per-instance quantity averaged over the methods explained, not a weight stored

in the model, so one fixed decision function can be read two ways. AutoML, included as a control, raises F1 yet broadens feature use in only four of the seven models, while aggregate metrics hide the instability, with micro-F1 at 0.99 where minority-class F1 is 0.15. Feature engineering, not model selection, is therefore the bottleneck for SRM detection. Progress depends on features that survive renaming, obfuscation stress tests, and per-class metrics alongside aggregate scores.

Keywords: explainable AI, SHAP, AutoML, software security, security-relevant methods, taint analysis

1 Introduction

As software systems continue to grow in complexity, the number of reported software vulnerabilities has risen sharply [1], making vulnerability detection more critical than ever. In response, many organizations have adopted Static Application Security Testing (SAST) tools as part of their secure software development practices. These tools rely on various types of analyses to identify security vulnerabilities in the source code. For example, taint analysis, a type of data flow analysis, track how untrusted data flows through a program. When detecting an SQL injection, SAST tools must know which methods generate user input (sources like `HttpServletRequest.getParameter()`), which methods execute SQL queries (sinks like `Statement.execute()`), and which methods sanitize input (sanitizers like `StringEscapeUtils.escapeSql()`). These critical methods, which handle sensitive operations and can be exploited if left vulnerable, are collectively known as security-relevant methods (SRMs) [2].

To support security analysts with the manual, error-prone, and time-consuming task of identifying SRMs, recent research has explored machine learning approaches for automated SRM detection (ML4SRM), including Dev-Assist [3], SSCM [4], SuSi [5], and SWAN [2]. While these approaches report promising cross-validation results on their training datasets, their performance drops substantially when applied to unseen, real-world source code, revealing a critical generalization problem. We attribute this gap to two principal limitations, the training data and the model selection process. Existing ML4SRM datasets are small (e.g., Dev-Assist is trained on only 1,625 examples [3]), class-imbalanced, and unlikely to be representative of the diversity of real-world code. Model selection, in turn, is performed mostly manually and restricted to a small set of simple classifiers (e.g., SWAN [2] evaluates only 7 of the 80+ classifiers available in the WEKA toolkit), which risks overlooking models with stronger inductive biases or better robustness on out-of-distribution inputs.

Addressing these limitations requires more than simply enlarging datasets or sweeping over more classifiers. We argue that explainable artificial intelligence (XAI) is therefore a necessary component to uncover whether the ML4SRM models are relying on spurious lexical patterns rather than security-relevant semantics. Explainability can help to uncover failure patterns that may be invisible to standard machine learning performance metrics. By surfacing the features driving each prediction, explanations

expose shortcut learning, reveal dataset artifacts, and let analysts validate whether the model's reasoning aligns with genuine SRM characteristics.

To address these limitations, this paper presents an explainability-centric study of three state-of-the-art ML4SRM approaches (Dev-Assist, SSCM, and SWAN). Our central question is *what these models have actually learned*, both as an end in itself and as a prerequisite for any informed claim about generalization. A machine learning model learns dependencies among the provided features. Therefore, which combinations of the tokens, types and structural properties it is given are jointly predictive of security relevance. We report those dependencies through feature-group attributions, which is the level at which they can be compared across approaches whose individual features differ.

Methodologically, we introduce *WEAVE* (WEKA Automated eValuator & Explainer), a tool that integrates automated machine learning (AutoML) [6], repeated cross-validation [7], and eXplainable Artificial Intelligence (XAI) [8, 9] using SHapley Additive exPlanations (SHAP) [10]. *WEAVE* is the apparatus that makes the explainability study possible, because it enables head-to-head comparison across WEKA/MEKA-based ML4SRM pipelines, the toolchain in which the prior literature publishes models. Grounding the study on the published pipelines alone would risk charging ML4SRM as a field with what is really an artefact of its small-scale, manual model selection. *WEAVE* additionally searches a far larger pipeline space and yields a second, AutoML-selected model per detection target. That second model acts as a control throughout, testing whether principled model selection on a better dataset removes the behaviour we report. As an enabling artefact for principled training, evaluation, and explanation, we additionally assemble *SRM-Dataset*, a consolidated collection of 10,180 labelled Java methods that corrects errors and gaps in prior SRM datasets while acknowledging label-noise limitations.

Through SHAP-based global and local analyses we show that, during training, ML4SRM models draw their evidence overwhelmingly from method and class names and from parameter signatures. They barely use the method body, the one group encoding behavioural, rename-invariant signal such as whether a method invokes a known sink. When the same models classify previously unseen methods, the feature groups they rely on shift not only in weight but in direction. Measuring a group's direction as the share of explained methods for which it pushes the prediction towards the SRM class, 48 of the 59 group-model pairs we can compare shift significantly between the training distribution and unseen code, by as much as 62 percentage points. 16 of them cross the halfway mark, so the group points the other way for most methods. Twelve of those sixteen move towards the class and four away from it, which is a less tidy picture than a single direction of failure but a broader one. Reliance is unstable in nearly every group of every model.

AutoML-based model selection does not remedy the instability. It broadens feature use in only four of seven detection targets, and the most evenly balanced model we explain is one of the originally published pipelines. The control tells us that the problem lies in the features rather than in how the pipeline over them was chosen. Aggregate metrics conceal the problem entirely, with micro-F1 reaching 0.99 on targets whose minority-class F1 is 0.15, and macro-F1 ranking two of the models opposite

to the precision–recall area. This instability of feature reliance under distribution shift explains the documented generalization failure of ML4SRM in practice and re-frames feature engineering as the principal bottleneck for SRM detection. We complement this finding with a discussion of how Large Language Models (LLMs) might supplement explainable ML4SRM, and what XAI uniquely contributes when LLMs themselves remain opaque.

In summary, this paper makes the following contributions:

- Explainability findings into how three state-of-the-art ML4SRM approaches make their predictions, including evidence that the direction in which they read a feature group is not stable between their training data and unseen code, and that the metrics conventionally reported for these models conceal that instability.
- *WEAVE*, an AutoML, repeated cross-validation explainability tool for WEKA/MEKA-based ML pipelines [11]. *WEAVE* is the methodological apparatus that enables the explainability findings.
- *SRM-Dataset*, a consolidated dataset of 10,180 labelled Java security-relevant methods [12]. We position this as a prerequisite for the study.

The remainder of this paper is organized as follows. Section 2 motivates explainable SRM detection. Section 3 presents *WEAVE* and the *SRM-Dataset* in Section 4. In Section 5, we report a preliminary model-identification study (RQ1) and then present the global (RQ2) and local (RQ3) explainability results and discuss the findings in Section 6. Related work is described in Section 7 and Section 8 concludes the paper.

2 Motivation

In this section we explain why accurate SRM lists are critical to security tooling and must be produced automatically, why machine learning is a practical means of doing so, and what SRMs are used for beyond configuring SAST tools. Because large language models (LLMs) have recently been applied to vulnerability analysis, we also explain why SRM detection remains valuable alongside them.

2.1 ML-Based Security-Relevant Methods Detection

SAST and taint-analysis tools depend critically on accurate lists of SRMs, because every source, sink, or sanitizer that is missing or mislabelled translates directly into missed vulnerabilities or spurious warnings [13]. Yet the space of security-relevant APIs is vast and constantly growing. Every new library, framework, and release introduces fresh SRMs, therefore, hand-curated lists are inevitably small, biased toward a few popular frameworks, and quickly stale. Producing and maintaining SRM lists at the scale of real software therefore has to be automated.

Machine learning offers a practical route. By learning from labelled examples, ML4SRM approaches [2–5] flag candidate SRMs across large bodies of unseen code at negligible marginal cost per method, surfacing methods that manual lists omit. Their output is a reusable specification. Once a method such as `Statement.execute` is labelled a CWE-89 sink, that label applies to every program that calls it, in every

analysis run. A trained model classifies a method in microseconds on commodity hardware, requires only a call graph rather than executable code or full source, and can be shipped inside an analysis tool and reused across the tooling discussed in Section 2.2.

Large language models, which can now identify and even explain vulnerabilities directly from source code [14], raise the question of whether dedicated SRM detection is still needed. We view the two as complementary rather than competing. SRM detection and per-program LLM analysis operate at different units of work. An SRM list is computed once and reused everywhere, is deterministic and auditable. In contrast, LLM training data is typically undisclosed; analyses must be recomputed for each program execution, increasing costs; and their outputs are non-deterministic which limits reproducibility [15]. SRM detection thus produces a high-leverage, reusable artefact on which cheaper static analyses and LLM-based triage can build.

These advantages, however, are contingent on the quality of the learned models. The central problem we examine is that ML4SRM models trained on small, imbalanced datasets often fail to generalise to unseen code. Because the task is necessary and the approach otherwise attractive, the right response is to diagnose why these models fail rather than to discard them. This is precisely what the explainability study at the core of this paper sets out to do.

2.2 Uses of SRMs beyond SAST configuration

Beyond classical SAST and taint analysis, SRMs serve as a security vocabulary for a broad range of tooling. In fuzzing and dynamic analysis, knowledge of sinks helps prioritise inputs that reach security-critical operations and distinguish reachable from latent vulnerabilities [16]. In code review and triage, SRM annotations let reviewers focus on the commits and diffs that introduce or modify security-critical behaviour [17]. Within IDEs, editors and language servers can highlight calls to SRMs in real time, surface CWE context, and recommend sanitizers, supporting developers who are not security experts. Curated SRM lists are also reusable artefacts for security training and documentation, making security-relevant patterns visible without per-codebase analysis. And for specification mining and policy enforcement, SRMs can be combined with type systems or annotation frameworks to enforce information-flow policies in build pipelines and CI/CD.

This breadth raises the stakes of misclassification. An SRM mislabelled by an ML4SRM model propagates to every tool that consumes the resulting list. Understanding what these models rely on is therefore not an academic curiosity but a precondition for adoption.

3 Approach

WEAVE employs automated machine learning (AutoML) to select candidate machine learning pipelines, which it then subjects to empirical evaluation and interpretation using explainable AI techniques. The overall architecture is depicted in Figure 1. Given a *Dataset* in WEKA's ARFF [18] format and an *Experiment Configuration*, both supplied through the command line interface, *AutoML Model Selection* uses ML$^{(2)}$-Plan [19, 20] to identify a suitable pipeline. *Repeated Cross-Validation*, built

on jaicore-experiments, then configures and orchestrates the cross-validation runs for that pipeline and records the resulting *Performance Metrics* in a database. In parallel, *Explainability* launches the appropriate WEKA [18] or MEKA [21] framework behind a *Prediction Bridge* and drives SHAP [10] against it, so that SHAP queries the bridge for prediction probabilities and returns *Global & Local Explanations*. We name these components as Figure 1 labels them, and each is described in its own section below. Although the workflow reads sequentially, the components are loosely coupled and can be run independently.

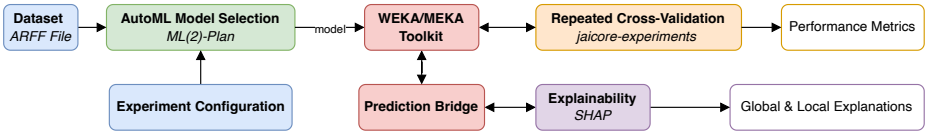


Fig. 1: Overview of *WEAVE*. From a WEKA/MEKA dataset and an experiment configuration, AutoML ($ML^{(2)}$-Plan) selects a pipeline; the selected model is then evaluated by repeated cross-validation and explained with SHAP, with a Prediction Bridge relaying queries between the Python SHAP process and the WEKA/MEKA toolkit.

3.1 Design Rationale

WEAVE is the first integrated apparatus for interrogating existing ML4SRM models on equal footing. Prior ML4SRM approaches each used ad-hoc model selection (often a handful of WEKA classifiers with default hyperparameters [2–4]), single-split cross-validation, and no post-hoc explanation. Comparing them rigorously requires three capabilities in combination: (i) systematic model selection, so that performance differences reflect model behaviour; (ii) repeated cross-validation [7], so that small differences can be separated from sampling variance; and (iii) model-agnostic explainability [10], so that the comparison extends from aggregate metrics to the features each model relies on. *WEAVE* integrates them around the specifics of the SRM problem, which are multi-label classification, heterogeneous WEKA/MEKA pipelines, and large mixed lexical/structural feature spaces.

This integration also dictates the choice of toolchain. The three approaches we study publish their models, feature extractors, and reproduction artefacts in WEKA/MEKA. Re-implementing them in a pure-Python (e.g., scikit-learn) stack would mean evaluating our reimplementations rather than the released artefacts, so any discrepancy with previously reported numbers could not be cleanly attributed to model behaviour. We therefore keep model training and prediction in WEKA/MEKA and expose the selected models to the Python-based SHAP through a lightweight prediction bridge. The three subsections that follow detail these capabilities in turn.

3.2 Automated Model Selection

Given a training dataset in WEKA's Attribute–Relation File Format (ARFF) [18] and a set of model selection options, the *AutoML Model Selection* component employs automated machine learning to automatically construct, configure, and select an appropriate machine learning pipeline. *WEAVE* integrates the AutoML tool ML-Plan[20] and its multi-label extension ML^2-Plan [19], jointly denoted ML$^{(2)}$-Plan hereafter, which optimize the choice of learning algorithm and its hyperparameters for a given classification task. This multi-label capability is essential for accurately selecting models for Dev-Assist, a ML4SRM approach in which multiple labels can be assigned to a single Java method. We adopt ML$^{(2)}$-Plan because they natively operate on the WEKA/MEKA ecosystem, support both single- and multi-label classification, and outperform alternative AutoML tools such as Auto-WEKA, auto-sklearn, and other baselines [19, 20]. This combination of tight toolchain integration, multi-label capability, and principled, resource-aware search makes ML$^{(2)}$-Plan particularly well suited as the model selection backend.

ML$^{(2)}$-Plan adopts a two-phase search strategy based on hierarchical AI planning, instantiated as a best-first search over the space of candidate pipelines [20]. Leveraging the breadth of algorithms implemented in the open-source WEKA and MEKA machine learning toolkits, it hierarchically composes candidate pipelines (including base learners and meta-algorithms), and for each candidate it performs training and cross-validation while employing mechanisms to control overfitting. Within the user-specified timeout, ML$^{(2)}$-Plan identifies the pipeline that achieves the best cross-validation performance under the given resource constraints. *AutoML Model Selection* then returns this pipeline as command line options, comprising the classifier fully qualified class names and hyperparameters.

3.3 Repeated Cross-Validation

The *Repeated Cross-Validation* component assesses the predictive performance of a provided machine learning model using repeated cross-validation [7], averaging the resulting metrics to obtain a robust estimate of model performance. We use the jaicore-experiments package [22], a macro-benchmarking library for parametrizable and reproducible machine learning experiments with native support for Java-based ML ecosystems, to evaluate the models. Using a point-wise evaluation strategy, each fold and repetition is systematically executed from the experiment grid. We compute standard performance metrics (true and false positives/negatives, precision, recall, F-measure) and retain raw per-fold and per-class predictions to enable post hoc analysis, including the explainability analyses described next.

3.4 Model Explanation

The explainability module augments the empirical evaluation of a model by providing insight into how it arrives at its predictions for a given classification task. Existing explainable AI tools for WEKA, such as ExpliClas [23], are limited in several respects. They support only a narrow subset of algorithms (typically tree-based

models), restrict explanations to a small number of features, and primarily only produce natural-language descriptions. As a result, they do not scale to more complex pipelines, nor do they offer the richer, fine-grained explanations available in modern XAI libraries for other ecosystems.

To overcome these limitations, we evaluated model-agnostic, Python-based XAI frameworks that can operate on predictions exported from non-Python libraries such as WEKA. In particular, we considered Shapley Additive Explanations (SHAP) [10] and Local Interpretable Model-Agnostic Explanations (LIME) [24]. SHAP is generally preferred over LIME because it provides both local and global explanations, accounts for feature interactions by considering different coalitions of features, and can capture non-linear relationships. LIME, on the other hand, is known to struggle with interaction effects and can be less stable across runs [25]. Conceptually, SHAP adopts a cooperative game-theoretic view of explanation, treating features as players and the model output as the payoff, and assigns Shapley values to quantify each feature's contribution to a prediction [25]. SHAP produces a vector of real-valued attributions (SHAP values) that quantify the marginal contribution of each feature to a given prediction [10], and these attributions have been shown to align well with human judgments of feature relevance [26]. These attributions can be aggregated across instances to derive global importance scores or inspected at the instance level to understand individual decisions.

Since *WEAVE* must handle a broad range of model types, including non-tree-based and complex meta-classifiers, we cannot rely on model-specific explainers in SHAP such as TreeExplainer (tailored to decision trees and ensembles). Instead, we configure SHAP with KernelExplainer [27], a fully model-agnostic method that estimates Shapley values by repeatedly querying the model under systematically perturbed feature subsets.

4 Security-Relevant Methods Dataset

SRM-Dataset is introduced as an enabling artefact for our study of explainability in ML4SRM. A post-hoc explanation is only as trustworthy as the data on which the explained model was trained. Noisy labels, opaque provenance, or a distribution mismatch between approaches can make a model appear to rely on spurious cues, confounding precisely the explanations we set out to analyse. The role of *SRM-Dataset* is therefore to provide a single, consistently labelled, and fully traceable training and evaluation substrate shared across the ML4SRM approaches we study. This ensures that observed differences in model behaviour, and in the explanations derived from it, can be attributed to the models themselves rather than to dataset mismatch.

The largest publicly available list of Java security-relevant methods to date for ML4SRM [3] comprises 1,625 methods drawn from popular frameworks and applications. While adequate for training an individual classifier, its modest size limits the diversity of behaviours that an explainability study can probe. We therefore take this list as our starting point, yet extend it with SRMs from several complementary sources, applying a uniform processing pipeline to each so that all methods are represented and labelled consistently.

4.1 Sources with existing SRM labels

Three of the four sources already publish a curated list of security-relevant methods, so our work on them is import and reconciliation rather than labelling. Table 1 summarises what each contributes, alongside the manually labelled source of Section 4.2 and the prior list. We take them as follows.

- **TaintBench [16].** We extract SRMs separately for the Java and Android subsets and keep these lists separate rather than merging them, because Android framework methods are not always portable to general Java and vice versa. The Android methods are maintained as a companion dataset and are not used in this study.
- **SecuCheck [17].** We extract SRMs appearing in SecuCheck's CWE-specific rule sets. All 24 fall into SecuCheck's *propagator* category, which is absent from earlier ML4SRM work and which none of the approaches studied here predicts, so they enter *SRM-Dataset* as methods that are not security-relevant in the modelled sense. We retain them because the released dataset is intended to outlive this study, and a propagator label is information a future approach can use.
- **CoDoC [28].** We extract the methods CoDoC had labelled and re-validate their CWE assignments against our coding guidelines to keep the labelling criteria uniform across sources.

For all sources, we applied schema normalisation, merged duplicates, and repaired metadata. This ensures that those roles rest on the curation of the projects that produced them rather than on a judgment of ours. For CoDoC we re-validated the CWE assignments against our coding guidelines.

4.2 Labelling the OWASP Benchmark

The OWASP Benchmark [29] is the one source that arrives without labels of its own, so we produced them. We labelled it for the local explanations experiment, intending to explain predictions on a program whose security-relevant methods we knew. We extracted those methods across the benchmark's 2,740 test cases with JimpleProvider [30], then removed framework entry points and methods already covered by our SRM lists (`doGet`, `doPost` and similar), leaving 162 candidates. Two authors labelled these independently under the same coding guidelines and reconciled disagreements by joint discussion. We ultimately draw the local explanations from an independently annotated codebase instead, and the methods agreed on here serve as training data like those from any other source.

Because these labels are ours rather than inherited, we measure their reliability rather than assume it, keeping its weight in proportion, since these are 162 methods or 1.6% of *SRM-Dataset*. Both labellers judged 110 of the candidates, 95 of which entered the dataset. Over those 95 and the five categories this paper models, pooled agreement is $\kappa = 0.14$ [31] at an observed rate of 0.76, rising to 0.66 for CWE89 where the category is actually exercised. The pooled figure reflects a difference of vocabulary more than ambiguity about security relevance, since labeller 1 assigned *source* to 68 of the 95 methods where labeller 2 assigned it to 2, placing 41 of those outside the five categories altogether, a boundary no model in this study is asked to draw. Counts

we report for the dataset, here and in Table 1, come from the released version rather than from the labelling sheet.

4.3 Consolidating the Catalogue

The resulting dataset, *SRM-Dataset*, contains 10,180 methods annotated with SRM and CWE labels, with the per-source role and CWE breakdown summarised in Table 1. The *SRM-Dataset* is roughly an order of magnitude larger than the largest prior dataset [4], giving explanation methods enough diverse examples to expose model-specific reasoning. the dataset contains both positive examples (security-relevant methods) and negative examples (methods considered to not to be security-relevant). The *SRM-Dataset* thus supports not only training and evaluating ML4SRM models but also more general supervised learning on security-relevant code, while the labelled SRMs can be used for the use cases described in Section 2.

Table 1: Per-source provenance of the 10,180 Java methods in *SRM-Dataset*, broken down by sources (So), sinks (Si), sanitizers (Sa), propagators (Pr), authentication (Au), methods not labelled as security relevant ($\notin_{SRM}$), and methods carrying at least one CWE label (CWE). The upper block lists the sources comprising the prior SRM list [3]; the lower block lists the sources newly added in this work. A method may carry more than one role (e.g., an authentication source), so the role columns need not sum to the per-source method count.

Source	**So**	**Si**	**Sa**	**Pr**	**Au**	$\notin_{SRM}$	**CWE**	**Total**
Prior SRM list [3]								
find-sec-bugs	147	512	24	0	0	1	467	**684**
SSCM	106	170	0	1	0	423	0	**700**
SWAN	42	61	27	0	63	48	136	**221**
TheCodeMaster	23	46	25	1	0	0	23	**94**
Newly added sources								
TaintBench [16]	4,561	1,932	0	0	0	0	0	**6,493**
CoDoC [28]	1,671	131	0	0	0	0	0	**1,802**
SecuCheck [17]	0	0	0	24	0	0	0	**24**
OWASP Benchmark [29]	3	20	0	30	0	118	34	**162**
Total	6,553	2,872	76	56	63	590	660	**10,180**

To make the dataset easily reusable, we provide it as a standalone Maven project that bundles the labelled methods, their metadata, and helper utilities. All entries are stored in JSON. Where the enclosing project could be resolved, the entry also carries the extracted source of the method definition (obtained with JavaParser [32]), and for the remaining methods we record the Maven coordinates needed to retrieve and parse the source on demand. This design lets the feature representations required by ML4SRM models, token sequences, AST-based features, and basic control-flow characteristics, be derived directly from *SRM-Dataset*, which both reduces the need to re-run feature-extraction pipelines and, more importantly for our purposes, ties every

explained feature to a concrete, inspectable code fragment. The project additionally ships helper classes for parsing, exporting, importing, and querying the dataset.

5 Evaluation

The evaluation centres on the explainability of three ML4SRM approaches (DevAssist, SSCM, SWAN) with the *WEAVE* (Section 3) and the *SRM-Dataset* (Section 4), and then explained a second time on previously unseen methods from the EBI *Gene Expression Atlas* (GXA) [33], a Java web application carrying independent SRM labels. We address the following research questions:

- **RQ1**: Which feature groups do the ML4SRM approaches define, and how do the pipelines that *WEAVE* selects for them compare with the originally published models?
- **RQ2**: Which types of source-code features globally drive the predictions of ML4SRM models, and how does the answer differ between published and WEAVE-selected models?
- **RQ3**: Which features do ML4SRM models rely on when classifying previously unseen methods, and where does the local explanation pattern diverge from the global one?

We evaluate each approach twice, once with its originally published pipeline and once with a pipeline that *WEAVE* selects from a far larger search space. Both are retrained on *SRM-Dataset*, evaluated under repeated cross-validation, and explained, so the first evaluation is a replication with explanations added and the second is a control rather than a proposed improvement. Prior ML4SRM work selects models manually from a handful of candidates, so any brittleness we find in the published models could plausibly be blamed on that choice rather than on the approach itself. Comparing the two variants tests that explanation directly, and **RQ2** reports the answer. We report the two model variants side by side throughout **RQ1** to **RQ3**, labelled *published* and *WEAVE* (AutoML-selected), so that every explainability finding can be read against the question of whether better model selection would have prevented it. **RQ1** establishes the models and their cross-validation performance, **RQ2** and **RQ3** explain them globally and locally.

5.1 RQ1: Feature Groups and AutoML Model Selection

In this research question, we first examine the feature groups that the three ML4SRM approaches rely on, and then use *WEAVE* to select, for each approach, the model that grounds our explainability study. Our focus here is not out-of-sample performance; rather, we select models with ML$^{(2)}$-Plan on the *SRM-Dataset* and validate them under repeated cross-validation. This establishes the two model variants, namely the originally published pipeline of each approach and an AutoML-selected pipeline that serves as the study's control.

5.1.1 Feature Groups of the ML4SRM Approaches

To obtain model- and tool-independent insights, we group the machine learning features according to the Java program elements from which they are derived, following the structure of class declarations and method definitions [34]. This abstraction enables cross-model comparison and shifts the focus from individual, low-level features to broader, semantically meaningful categories.

We distinguish three top-level groups: *class declarations (C)*, *method definitions (M)*, and *data-flow (DF)* (i.e., data-flow to and from class and method components). The first two groups follow the terms used in the Java grammar. A class *declaration* names the type, its modifiers and its supertypes, whereas a method *definition* comprises both a declaration (modifiers, return type, name, parameters) and the body that implements it [34]. The distinction matters here because the body is exactly where behavioural, rename-invariant evidence lives. The class and method groups are refined into the standard Java constructs: modifiers (Mo), return type (RT), name (N), parameters (P), and body (B). The data-flow group covers data-flow to and from class and method components. Table 2 summarizes these groups and, for each ML4SRM approach, reports the number of features in each category.

Table 2: Descriptions, examples and counts of features from the Dev-Assist[3], SSCM[4], and SWAN[2] MLSRM approaches divided into groups based on class declarations, method definitions, and data-flow.

			Feature Count		
ID	**Feature Groups**	**Example**	**[4]**	**[2]**	**[3]**
C	***Class Declarations***				
CMo	***Modifier*** Type of class modifier.	Is "public"	3	5	4
CN	***Name*** Bag-of-words approach checks if tokens in the vocabulary appear as prefixes, suffixes, or substrings.	Ends with "servlet"	14	25	25
CB	***Body*** Quantifiable metrics for the class.	Lines of code	0	0	2
M	***Method Definitions***				
MMo	***Modifier*** Method modifiers. *See CMo.*	Is private	1	6	3
RT	***Return Type*** Evaluates the data type of values returned by the method.	Is char[]	7	9	1
MN	***Name*** Method name. *See CN.*	Starts with "do"	22	38	33
P	***Parameter*** Evaluates parameter names (*See CN*), data types, and counts.	Is string	12	10	2
MB	***Body*** Software metrics and method invocations in the method body.	Invokes sink SRM	39	38	29
DF	***Data-Flow*** Data-flow among method components with static analysis.	Flows to sink method	5	22	22
			103	153	121

Across all approaches, 50–59% of the features are derived from method names (21–27%) and method bodies (24–38%), indicating a strong emphasis on lexical and intra-procedural properties at the method level. For SWAN and Dev-Assist, class

name and data-flow features each account for a further 14–21% of features, reflecting the use of contextual and inter-procedural information. In contrast, features based on class modifiers, class body, method modifiers, and return type form small groups (at most 7%, and absent altogether from the SSCM and SWAN class-body group), suggesting that these aspects are under-represented in current ML4SRM feature sets. Having established the feature vocabulary, we now turn to selecting and evaluating the models.

5.1.2 Experimental Setup & Evaluation Metrics

For each ML4SRM approach (Dev-Assist [3], SSCM [4], and SWAN [2]), we re-ran the original feature extraction pipelines to construct new training datasets based on *SRM-Dataset*. In *WEAVE* 's model selection phase, we configured ML$^{(2)}$-Plan with 64 CPU cores, a timeout of 8 hours, a 70:30 train–test split, and the default settings recommended by the tool developers. The empirical evaluation was configured to re-evaluate the published ML4SRM models (as reported in the respective papers) and evaluate the new AutoML-selected models, each under 100 different random seeds using repeated cross-validation. All experiments were executed on the Noctua 2 supercomputer [35], using a single node equipped with two 64-core AMD Milan 7763 2.45 GHz CPUs and 256 GiB of main memory.

To assess predictive quality and practical utility, we use standard metrics derived from repeated cross-validation, namely precision, recall, micro-averaged F1-score, and macro-averaged F1-score [36, 37]. Let TP, FP, and FN denote the number of true positives, false positives, and false negatives, respectively. We define: Precision ($P = \frac{TP}{TP+FP}$), Recall ($R = \frac{TP}{TP+FN}$), micro F1-Score ($F1_{mi} = \frac{2*Precision*Recall}{Precision+Recall}$), and macro F1-Score ($F1_{ma} = \frac{1}{n}\sum_{i=1}^{n} F1_{mi,i}$). These metrics are critical in security contexts, where both false positives and false negatives can lead to significant effort and waste developers time and effort [38] or contribute to residual risk [39].

5.1.3 Selected Models and their Performance

Table 3 summarizes the pipelines selected by *WEAVE* and compares their repeated cross-validation performance against the published ML4SRM models. Overall, the *WEAVE* models are at least competitive and, in several cases, clearly outperform the published pipelines, while also illustrating that different security-relevant tasks favour different classifier families. Notably, most WEAVE-selected pipelines are ensemble methods, which is consistent with their known advantages on imbalanced datasets [40] and indicates that ML$^{(2)}$-Plan systematically exploits ensembles to address the class imbalance in *SRM-Dataset*.

Dev-Assist

The published Dev-Assist model was selected by running ML^2-Plan with a one-hour timeout, yielding a Pruned Sets (PS) + Logistic Model Trees (LMT) pipeline [3]. This timeout is substantially below the four-hour minimum recommended for adequately exploring the model space [41], increasing the risk of prematurely converging on a suboptimal model. The published pipeline relies on LMT, a lazy, locally adapted model

Table 3: Repeated cross-validation evaluation of the published and *WEAVE* ML4SRM models (Bagging (B), Logistic (L), Logistic Model Trees (LMT), LogitBoost (LB), Pruned Sets (PS), RandomForest (RF), RandomSubSpace (RSS), Simple Logistic (SL), and Support Vector Machine (SVM)) showing the Precision (P), Recall (R), and micro F1-Score (F_{mi}) and macro F1 (F_{ma}), averaged over 100 random seeds (standard deviation $\sigma \leq 0.03$ for all cells). $\uparrow/\downarrow$ marks a *WEAVE* macro-F1 that is statistically significantly higher/lower than the published model's ($p < 0.01$, $n = 100$; Wilcoxon signed-rank over matched seeds for the SWAN and SSCM tasks, and the unpaired rank-sum test for Dev-Assist, whose published and WEAVE runs use different seeds).

	WEAVE Models					*Published Models*				
Approach	**Model**	**P**	**R**	**F1**$_{ma}$	**F1**$_{mi}$	**Model**	**P**	**R**	**F1**$_{ma}$	**F1**$_{mi}$
Dev-Assist	PS+J48	0.81	0.69	$0.73^{\uparrow}$	0.88	PS+LMT	0.77	0.70	0.73	0.86
SSCM	B+RF	0.78	0.73	$0.75^{\uparrow}$	0.91		0.74	0.68	0.71	0.90
SWAN-CWE79	RSS+L	0.75	0.55	$0.63^{\downarrow}$	0.99		0.93	0.54	0.68	0.99
SWAN-CWE89	SL	0.92	0.78	$0.84^{\downarrow}$	0.99	SVM	0.90	0.80	0.85	0.99
SWAN-Sanitizer	B+L	0.92	0.91	$0.92^{\uparrow}$	1.00		0.94	0.89	0.91	1.00
SWAN-Sink	LB+RF	0.87	0.87	$0.87^{\uparrow}$	0.89		0.86	0.81	0.83	0.87
SWAN-Source	RF	0.92	0.89	$0.90^{\uparrow}$	0.91		0.90	0.84	0.87	0.88

that performs limited work at training time and tailors predictions at query time based on similarity to training instances.

With a substantially longer timeout in our experiments, ML^2-Plan selects an alternative model that improves all reported metrics. This is consistent with the expectation that a one-hour search is insufficient to thoroughly explore the candidate space and may lead to overfitting or suboptimal choices. By allowing more time, ML^2-Plan evaluates a broader and more diverse set of pipelines and, as a result, is more likely to identify robust, well-performing models [20].

SSCM

For SSCM, the *WEAVE* ensemble (Bagging+RandomForest) model modestly but consistently improves upon the published SVM model. Precision increases from 0.74 to 0.78 and recall from 0.68 to 0.73, raising macro F1-Score from 0.71 to 0.75 and micro F1-Score from 0.90 to 0.91. This suggests that the WEAVE-selected model covers more relevant methods across classes without compromising overall correctness. The ensemble classifier likely reduces variance and mitigates overfitting effects that are common in individual tree-based learners, leading to more reliable in-sample performance on *SRM-Dataset*.

SWAN

SWAN is the only approach that trains a separate binary classifier per SRM category, so it is the only one where the choice of categories arises at all. Dev-Assist learns all five jointly as a multi-label problem and SSCM is a single three-class model, and in both

cases dropping a category would mean discarding the model rather than restricting it. We evaluate five categories, and the reasons differ by kind. Source, sink and sanitizer are the complete vocabulary a taint analysis has to be configured with, so we retain all three however many positives *SRM-Dataset* provides for each. Among the CWE-specific categories we evaluate CWE89 and CWE79, the two for which a published SWAN model exists to compare against, and CWE89 is additionally the only CWE category labelled in GXA, which makes it the only one whose local explanations can be read against ground truth in **RQ3**. Support varies by two orders of magnitude across the five, from Source at 6,007 positives and Sink at 2,611 of 9,229 methods to CWE89 at 118, CWE79 at 87 and Sanitizer at 74. We report those counts rather than filter on them, because the small classes are where the gap between aggregate and per-class performance becomes visible, and the rest of this section treats them accordingly.

Across these categories, both the published and *WEAVE*-based models achieve very strong micro F1-Scores (0.86–1.00), indicating high overall effectiveness. The *WEAVE* models generally match or improve upon the SVM models. For Sanitizers and CWE89, both published and WEAVE-selected models achieve near-perfect micro F1-Score (1.00 and 0.99) and very similar macro F1-Score, suggesting that these tasks are robustly solved by either pipeline. For Sinks and Sources, the *WEAVE* models (LB+RF for Sink, RF for Source) yield clear gains in precision and recall, resulting in higher macro F1-Scores. The tree-based ensembles appear to capture richer patterns than linear SVMs. The main exception is CWE79, where the SVM model attains higher precision and macro F1-Score. The *WEAVE* RandomSubSpace+Logistic model is more permissive and sacrifices some precision, which is plausible given that CWE79 has the fewest positive examples and thus provides limited, highly imbalanced training data.

The strong micro-F1 values mask poor detection of the minority (security-relevant) class. Recomputing precision, recall and F1 on the positive class alone, as opposed to the macro averages of both classes in Table 3, makes this concrete. CWE79 is the starkest case, where both models reach micro-F1 = 0.99 while their positive-class F1 is only 0.17 (*WEAVE*) and 0.15 (published), because the score is carried almost entirely by the negative majority whose recall is 1.00. The two models fail in opposite ways. The published SVM is highly conservative, with positive-class precision 0.88 against recall 0.08, so it is usually right when it fires and almost never fires. On the other hand, *WEAVE*'s model is more permissive (0.50 against 0.10). *WEAVE*'s clearest minority-class gain is on Sink, where positive-class F1 rises from 0.75 to 0.81 on the strength of recall ($0.67 \rightarrow 0.81$).

We keep CWE79 in the study rather than dropping it, because it is the sharpest evidence for the gap between aggregate and per-class performance. Its explanations, however, have to be read for what they are. A model with positive-class F1 of 0.17 is not a working CWE79 detector, so the attributions we report for it in **RQ2** describe how a failing detector distributes its evidence, not what distinguishes a cross-site-scripting sink. We say so again where those attributions appear, and **RQ3** excludes CWE79 altogether because GXA carries no ground truth for it.

The choice of metric does not merely sharpen this picture, it changes the ranking. On macro-F1 the *WEAVE* models are significantly *worse* for CWE79 (-0.05)

and CWE89 (−0.005), yet on the area under the precision–recall curve (the standard summary under heavy class imbalance), *WEAVE* is better on *every* task. This is visible where the imbalance is most severe, at 0.20 against 0.08 for CWE79, 0.75 against 0.49 for CWE89, 0.90 against 0.65 for Sink, 0.88 against 0.69 for Sanitizer and 0.96 against 0.86 for Source. The same holds for SSCM, whose minority class is the *non*-security-relevant `None` category, where the precision–recall area rises from 0.12 to 0.42. Reporting only micro-F1 hides the minority class; reporting only macro-F1 can invert the comparison. This is why class-balanced, per-class metrics should drive model selection on the heavily imbalanced SRM problem and on the metric best suited to it. AutoML-based selection is uniformly the stronger choice.

Statistical Reliability

Across the 100 seeds the metrics are highly stable (standard deviation $\sigma \leq 0.03$ for every cell in Table 3. Macro-F1 $\sigma \leq 0.011$, micro-F1 $\sigma \leq 0.001$), so even small differences between the *WEAVE* and published models are statistically resolvable. A paired Wilcoxon signed-rank [42] test over the matched seeds confirms that the *WEAVE* models attain significantly higher macro-F1 for SSCM (+0.05), Source (+0.03), Sink (+0.03), and Sanitizer (+0.003), and significantly lower macro-F1 for CWE79 (−0.05) and CWE89 (−0.005), all with $p < 0.01$. Dev-Assist's label-macro-F1 is likewise significantly higher (+0.006), here by an unpaired rank-sum test because its WEAVE and published runs do not share seeds. We stress, however, that statistical significance here does not imply practical importance, because the variance is so small, a gain as tiny as 0.003 (Sanitizer) registers as significant. On macro-F1 the only practically meaningful improvements are SSCM, Sink, and Source. That verdict is metric-dependent, however. The two macro-F1 regressions (CWE79 and CWE89) are precisely the tasks on which *WEAVE* gains most in precision–recall area, so the significant negatives should not be read as the *WEAVE* models detecting security-relevant methods less well.

RQ1 Summary:

The three approaches draw 50–59% of their features from the method name and method body, and almost nothing from class body, modifiers or return type. AutoML matches or improves the published pipelines on most detection targets, with a standard deviation of at most 0.03 across 100 seeds, and it prefers tree ensembles to the linear SVMs of the published models. Those gains are recorded in aggregate metrics that hide the minority class, and both CWE79 models reach micro-F1 0.99 at a positive-class F1 of 0.17 and 0.15. Per-class metrics, not micro-F1, are what distinguish these models.

5.2 RQ2: Global Explainability of ML4SRM Models

In this research question, we investigate which types of source-code features drive the predictions of ML4SRM models at the level of overall model behaviour (global explanations). We use *WEAVE* 's XAI module, based on SHAP, to analyse what the models have learned from the training data in aggregate.

5.2.1 Experimental Setup

We configured SHAP's KernelExplainer with its default settings [43] and initially attempted to explain all instances in the training datasets. Despite multi-threading and parallelization, the runtimes were prohibitive, for the reason the explainer itself documents. KernelExplainer approximates a missing feature by substituting values drawn from a background dataset, so the model is re-evaluated once per background row for every sampled coalition, and cost therefore scales with the size of that background set. Its documentation accordingly advises using the whole training set as background data only for small problems and summarising it otherwise, either down to a single reference value or with k-means [44]. The library ships a utility whose stated purpose is this downsampling [45] and its implementation warns whenever the background set exceeds 100 rows, recommending a sampled or k-means summary in its place [46]. Following that guidance and recent empirical work on background set selection [47], we use stratified subsamples of the training data. We derived the subsample sizes by following prior recommendations to first run pilot experiments with small background sets [47] to approximate computational complexity. After experimenting with sample sizes of 100–1000 instances, we settled on 500 training instances per model for global explanations and 250 previously unseen methods from GXA (Section 5.3.1) for local explanations. Both exceed the 100 rows above which SHAP warns about runtime, a choice we make deliberately for the stability reported below and pay for in the runtimes reported next. The cost of the configuration we abandoned is worth recording, because it is what makes subsampling necessary rather than merely convenient. Explaining every training instance of SWAN-Source against a 500-method background set reached 540 of 9,229 instances in twelve hours and projected a further 192 hours, at that rate. Subsampling is therefore necessary and is only defensible if the quantity we report survives it.

5.2.2 Evaluation Metrics

To quantify the contribution of individual features, we use SHAP values, which provide a game-theoretic, model-agnostic measure of feature importance and direction of impact [26]. For a given prediction, a positive SHAP value indicates that a feature pushes the prediction towards the SRM class, while a negative value pushes it towards the non-SRM class. Aggregating SHAP values over instances yields global importance scores while inspecting them per instance yields local explanations.

SHAP offers several visualizations for these values. Beeswarm plots are often used for information-dense global summaries and waterfall plots for detailed local explanations [43]. Because we compare multiple models and wish to highlight broader patterns, we aggregate SHAP values by feature groups. To keep the contributions comparable across models, we report each group's *share* of a model's total |SHAP|, the fraction of the model's total attribution magnitude carried by that group, so that the shares of all groups sum to one. This says how much of a model's evidence a group supplies, and it is what the heatmap in Figure 2 shows.

Magnitude does not say which way a group pushes, and the two questions need different statistics. We therefore report direction separately, as the fraction of explained

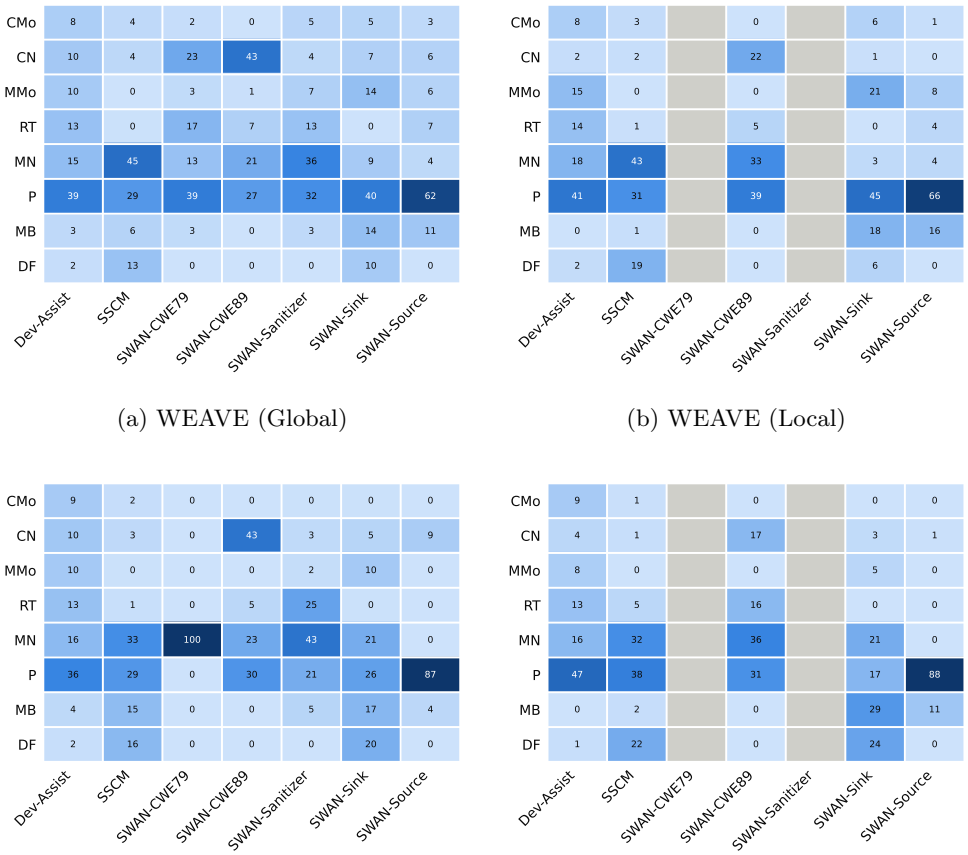
methods for which a group's net contribution is positive, with 50% as its neutral point. Aggregating the signed contributions instead would not serve, because positive and negative per-method values very nearly cancel when summed over hundreds of methods. Their total is a small difference of large numbers whose sign is not statistically resolvable at these sample sizes, whereas a per-method fraction has a standard error we can quote. Section 5.3.2 develops this statistic and uses it for the global→local comparison in Figure 3. Because both quantities are averages over the methods being explained, they characterise a model *together with* the distribution it is applied to, not the model alone, which is what makes that comparison informative rather than tautological.

5.2.3 Global Explanation Results

Two patterns are visible across Figure 2 before turning to individual approaches. The first concerns how much of its feature set a model actually uses, and the observation is that the models are far narrower than those sets suggest. Of the eight feature groups that carry any attribution anywhere in the study, only one to five reach 10% of a given model's attribution, with a median of three across the 24 available panels. Parameters and method name between them lead 21 of those 24 panels (parameters alone leads 14) and parameters falls below 2% in just one panel while method name does so in two. At the other extreme, class modifiers are negligible ($< 2\%$) in 14 panels and data-flow in 15, even though data-flow is one of the larger groups in both the SWAN and Dev-Assist feature sets (Table 2). Feature groups that are well represented in the input are therefore not necessarily used, and the attribution concentrates on the two most superficial, rename-sensitive groups available.

To quantify how broadly a model distributes importance across feature groups, we report the effective number of feature groups using the inverse Simpson index [48] of each group's share of total |SHAP|. Inverting the index expresses concentration as a count rather than a probability, which is the second-order effective number in Hill's unifying notation for diversity measures [49]. A value of k means the model behaves as if relying on k equally weighted groups, and it compresses a column of Figure 2 into a single number.

By that measure the published models hold both extremes. They are the sparser on average, with 3.0 of eight groups falling below 1% against 1.7 for the *WEAVE* models, and two of them collapse almost entirely onto a single group. SWAN-CWE79 draws 100% of its global attribution from method name and SWAN-Source 87% from parameters, which puts them at 1.0 and 1.3 effective groups. For SWAN-CWE79 this is the signature of a detector that does not work, since its positive-class F1 is 0.15–0.17 (Section 5.1), so what the attribution describes is how a failing model spends its evidence, not what characterises method relevant for cross-site-scripting. The single broadest model in the study is nonetheless also a published one, Dev-Assist at 5.0. Across the seven approaches the *WEAVE* model is the broader of the pair in only four cases and the published model in three, and the gains concentrate where the published model was degenerate, with SWAN-CWE79 rising from 1.0 to 3.9 and SWAN-Source from 1.3 to 2.4. Where the published model already distributed its attribution, AutoML narrows it, with SWAN-Sink falling from 5.2 to 4.4, Dev-Assist



(a) WEAVE (Global)

(b) WEAVE (Local)

(c) Published (Global)

(d) Published (Local)

Fig. 2: Share of each feature group in the total |SHAP| of every model, split into four sub-panels so that *global* and *local* explanations for the WEAVE and published models can be compared directly. Darker = a larger share of the model's attribution; grey cells denote a model with no local explanation.

from 5.0 to 4.6 and SSCM from 4.1 to 3.3. Broader feature use is therefore best read as AutoML repairing pathologically narrow models rather than as a general property of the selected pipelines.

The second pattern is that direction is a property of the *model*, not of the feature group. Across the 24 panels for which per-method data is available, parameters point towards the SRM class for most methods in fourteen panels and away from it in nine, and the remaining evenly split groups are method modifiers (7 against 7) and data-flow (6 against 6). Where a group does lean consistently, it leans away from the class. Method name points away for most methods in 19 of the 22 panels in which it registers, class name in 16 of 22, method body in 12 of 17, return type in 11 of 16 and class

Table 4: Effective number of feature groups (inverse Simpson index over each group's share of |SHAP|) for the *WEAVE* and published models, globally and locally. Higher values indicate importance spread over more groups. "–" denotes a cell with no reportable explanation. Local explanations for SWAN-Sanitizer and SWAN-CWE79 are excluded because GXA carries no ground truth (Section 5.3.1). The Dev-Assist rows are restricted to its three GXA-labelled categories.

	WEAVE		**Published**	
ML4SRM Approach	**Global**	**Local**	**Global**	**Local**
Dev-Assist	4.6	4.0	5.0	3.5
SSCM	3.3	3.1	4.1	3.4
SWAN-CWE79	3.9	–	1.0	–
SWAN-CWE89	3.2	3.2	3.1	3.6
SWAN-Sanitizer	3.9	–	3.4	–
SWAN-Sink	4.4	3.5	5.2	4.5
SWAN-Source	2.4	2.1	1.3	1.3

modifiers in 9 of 14. No group can therefore be described as generally indicative of security relevance across the ML4SRM approaches, and those with a stable direction are stably exculpatory. What a group means is learned per detection target. We now discuss each approach in turn.

Dev-Assist

Dev-Assist is the only multi-label approach, and both of its runs export attributions per label, so we restrict both to the three categories GXA annotates (source, CWE89 and sink) and drop CWE79 and sanitizer, for which no ground truth exists. The restriction matters, because pooling all five labels changes the published model's class-modifier group from pointing towards the SRM class for 45% of explained methods to doing so for 65% of them.

Globally the two variants agree closely. The *WEAVE*-selected model is led by parameters (39.0% of |SHAP|), method name (14.7%), return type (13.0%) and class name (10.1%). The published model relies on parameters (35.7%), method name (16.1%), return type (13.2%) and class name (10.4%). No group differs by more than four percentage points in magnitude, and both are led at the feature level by parameter count (37.0% and 33.8%) ahead of return type. They are also the two broadest models in the study (4.6 and 5.0 effective groups, Table 4), with the published model marginally the broader. Their top three groups reach only about two thirds of the attribution, against 95–100% for the most concentrated SWAN models. The modifier groups carry a similar amount in both (8.2% and 9.8% for *WEAVE*, 9.4% and 10.1% for the published model) and both read them as evidence against security relevance more often than for it, for 41 to 45% of methods.

Dev-Assist's breadth therefore comes from spreading weight across generic signature properties rather than from using more security-specific evidence. `ClassNameContainsSQL` contributes only 1.9% and `ClassNameContainsJDBC` does not reach the top twelve. And for this approach, replacing the published PS+LMT pipeline with the WEAVE-selected PS+J48 changed the classifier without materially changing which features the model leans on.

SSCM

SSCM is the only multi-class task, its `tag` attribute distinguishing *Input* (source), *Target* (sink) and *None*; we pool the two security-relevant classes and drop *None*, which is the absence of a detection target rather than one of them. Both variants concentrate on the same two groups. For the *WEAVE*-selected model, method name contributes 44.8% and parameters 28.7% of the attribution, and for the published model 32.5% and 29.4% (Figures 2a and 2c). Together those two groups account for roughly three quarters of the *WEAVE* model's attribution and three fifths of the published model's. Their direction is close to even rather than uniform. Parameters point towards the security-relevant classes for 50% of the explained methods under *WEAVE* and 59% under the published model, method name for 42% and 53%.

Two smaller differences distinguish the pair. Data-flow is a substantial contributor in both (12.6% for *WEAVE*, 16.4% for the published model), which is notable because SSCM devotes only five features to data-flow, the smallest such group of the three approaches (Table 2). A little over one twentieth of its feature budget carries more than a seventh of its attribution. Method body is where the two differ most in weight, at 5.8% for *WEAVE* against 15.2% for the published model, which is the largest method-body share of any published model in the study. As with Dev-Assist, then, replacing the published SVM with an AutoML-selected Bagging+RandomForest ensemble left the model's reliance on lexical and parameter evidence largely intact, and moved it further away from the one group that reads the method's body.

SWAN

For the *WEAVE*-selected SWAN models, global explanations (Figure 2a) are dominated by *parameters*, the largest group in four of the five models (CWE79 39.2%, CWE89 27.5%, Sink 39.9%, Source 62.2%) and the second largest for Sanitizer (32.0%). Parameters are what SWAN reasons with throughout, but how the group is read differs sharply by category. It points towards the target class for most explained methods on Source (62%), Sink (54%) and CWE79 (55%), and for very few on CWE89 (20%) and Sanitizer (7%). Around that, the models diverge. CWE89 pairs parameters with method name (21.5%) and a large class-name group (42.8%), both of which point away from the class for all but a few methods (3% and 4%). Sink adds method body (14.2%) and data-flow (9.8%) as secondary groups. Sanitizer's largest single group is method name (36.1%), pointing towards the class for only 3% of methods. Class modifiers and data-flow contribute little anywhere except Sink. At a higher level, the published models rely more heavily on signature-based features for identifying vulnerability classes (e.g., CWE79 draws all of its attribution from method name). *WEAVE* shifts four of the five binary classifiers away from purely name-based features, and the

combined class-name and method-name share falls for CWE79 (100% → 36%), Sanitizer (46% → 40%), Sink (26% → 16%) and marginally CWE89 (65% → 64%), the exception being Source, where it rises slightly (9% → 10%).

RQ2 Summary:

Every model uses far fewer feature groups than it is given. Of the eight groups that carry any attribution, only one to five reach 10% of a model's attribution, with a median of three. Parameters and method name lead 21 of the 24 panels, while the method body, the one group encoding rename-invariant behaviour, stays at or below 6% for five of the seven AutoML-selected models. Data-flow is negligible in 15 panels despite being one of the largest groups in the SWAN and Dev-Assist feature sets. AutoML broadens feature use in only four of the seven targets, so the concentration is a property of the feature sets rather than of how the pipeline over them was chosen.

5.3 RQ3: Local Explainability of ML4SRM Models

In this research question, we investigate which types of source-code features drive the predictions of ML4SRM models for individual predictions (local explanations). We use *WEAVE* 's XAI module, based on SHAP, to analyse which features they rely on when classifying previously unseen methods.

5.3.1 Experimental Setup

We reuse the SHAP configuration described for the global analysis (Section 5.2.1), however, instead of explaining training instances, we explain previously unseen methods from a separate, independently labelled codebase. This is what lets RQ3 probe prediction-time, rather than training-time, feature reliance.

We use GXA, the Java web application behind the European Bioinformatics Institute's gene-expression service (revision `630aef60b`). Labels come from the dataset of Piskachev et al. [2], whose GXA subset comprises 1,663 method signatures, each annotated independently by two labellers as *Source*, *Sink* and *CWE89*. Agreement varies by label ($\kappa = 0.60$, 0.73 and 0.92 respectively). We use consensus labels only; requiring agreement on all three labels simultaneously retains 1,310 methods (78.8%).

From this strictly agreed set we draw 250 methods, stratified by label combination and module, with positives oversampled (every CWE89 and multi-label method in the agreed set retained, plus 70 *Source*-only, 58 *Sink*-only and 75 negatives) and capped at five methods per declaring class. The sample spans 116 distinct classes and contains 116 source, 80 sink and 28 CWE89 positives; sampling is seeded. Because this deliberately distorts prevalence, SHAP background sets are drawn from the full consensus population rather than from the sample. The dataset labels sources, sinks and CWE89 only. We therefore report local explanations solely for models whose target class is labelled in GXA, and exclude the SWAN-Sanitizer and SWAN-CWE79 local explanations throughout.

Before interpreting any local explanation it is worth asking what the models predict on this sample at all, since an explanation of a detector that never fires describes a different object than an explanation of one that does. Table 5 reports, for each explained model, how many of the sampled methods carry a positive consensus label

and how many the model scores above 0.5. Sources and sinks are detected at a plausible rate. CWE89 is not detected at all. Of the 28 labelled CWE89 methods, SWAN-CWE89 scores one method above the threshold across both variants and Dev-Assist scores two, and none of them is a true CWE89 method. The CWE89 local explanations in this section therefore describe how a model distributes evidence while predicting the negative class throughout, not how it recognises SQL-injection sinks, and we read them accordingly.

Table 5: What the explained models predict on the 250-method GXA sample. *Pos.* counts methods carrying a positive consensus label, *Fired* counts methods the model scores above the decision threshold, and *TP* counts how many of those are genuinely positive. SWAN-Sanitizer and SWAN-CWE79 are omitted, because GXA carries no ground truth for either.

		WEAVE		**Published**	
ML4SRM Approach	**Pos.**	**Fired**	**TP**	**Fired**	**TP**
SWAN-Source	116	129	83	150	97
SWAN-Sink	80	71	32	32	18
SWAN-CWE89	28	0	0	1	0
Dev-Assist-Source	116	142	71	123	57
Dev-Assist-Sink	80	100	41	118	47
Dev-Assist-CWE89	28	2	0	0	0
SSCM-Source	95	137	79	135	79
SSCM-Sink	80	65	41	80	48

5.3.2 Evaluation Metrics

As in RQ2 (Section 5.2.2), we use per-instance SHAP values aggregated by feature group, here computed for each individual unseen method, and reported as signed shares so that local patterns can be compared directly against the global ones (Figures 2 and 3).

As in RQ2 (Section 5.2.2), we aggregate per-instance SHAP values by feature group, here computed for each individual unseen method. We report two quantities. The first is the group's share of the model's total |SHAP|, as in RQ2, which says how much of a model's evidence a group carries (Figure 2). The second is the group's *direction*, the fraction of explained methods for which its net contribution is positive, that is, pushes the prediction towards the SRM class (Figure 3). Fifty percent is the neutral point of that statistic, and because it is a proportion over hundreds of methods we can attach a standard error to it and test the global→local change with a two-proportion z test.

For the reason given in Section 5.2.2, we do not report direction as the sign of a group's aggregate net contribution. On the comparisons below that aggregate amounts

to a median of 4% of a group's own magnitude globally and 0.1% locally. Its sign is not resolvable at these sample sizes, whereas the per-method fraction is estimated from the same data without that defect.

The models are unchanged between the two analyses; only the methods being explained differ, so any difference is a difference in how that model reads a new input distribution.

5.3.3 Local Explanation Results

The central local-explanation finding is that the role of a feature group changes between the training distribution (global) and previously unseen methods (local), and that it changes almost everywhere. Of the 59 group–model pairs that carry a non-negligible share of attribution on either side, 43 shift significantly in direction ($|z| \geq 1.96$), with shifts of up to 62 percentage points. The instability is not confined to marginal groups, since class name accounts for 9 of the significant shifts and parameters for 8, the two groups that carry the most attribution in the first place.

Fourteen of those shifts cross the neutral point, so the group changes which way it points for a *majority* of methods. These crossings do not run predominantly in one direction. Eight run from mostly-away to mostly-towards the SRM class and six the other way, which is a more equivocal picture than a reading based on aggregate signs would suggest. The sharpest single case is the published SWAN-CWE89 model, whose return-type group points towards the SRM class for 70% of training methods but only 8% of GXA methods ($z = -17.4$), while its parameter group moves the opposite way, from 16% to 78% ($z = +14.1$). The *WEAVE*-selected SWAN-CWE89 model shows the same return-type collapse (57% $\rightarrow$ 8%).

The pattern recurs across approaches. Both Dev-Assist models move on the groups that identify methods lexically. The published model's method modifiers rise from 37% to 61% ($z = +11.0$) and its class modifiers from 45% to 65% ($z = +9.5$), while the *WEAVE*-selected model's method name rises from 40% to 56% ($z = +6.9$) as its class name falls from 46% to 30% ($z = -6.8$). For SWAN-Sink the *WEAVE* model's class name all but stops pointing towards the class at all, from 29% to 3% ($z = -11.4$), and for SWAN-Source it moves the other way, 45% to 79% ($z = +7.9$). The two SSCM models move on the same lexical groups but less far, the published model's class name falling from 69% to 57% ($z = -4.4$) and the *WEAVE*-selected model's method name rising from 42% to 50% ($z = +2.9$). Every one of the ten model–variant pairs with both explanations available shows at least one significant shift, and every feature group is implicated in at least three, so the instability is a property of the models rather than of one feature type.

Crucially, the instability is largely invisible in magnitude alone, which is why the two quantities are reported separately. For *WEAVE*'s SWAN-Source model the method-modifier group barely changes in size between global and local explanations (5.8% $\rightarrow$ 8.1% of |SHAP|) while the fraction of methods for which it points towards the SRM class moves from 42% to 51%. Its class-name group shrinks slightly in magnitude while its direction moves 34 points. Parameters stay the dominant group for SWAN-Source in both views (62% $\rightarrow$ 66% of |SHAP|) while the groups around them reorganise. A magnitude-only reading of these models would show a broadly stable

feature profile and miss the change entirely. The common thread is instability of feature reliance under distribution shift, and it is a change of direction at least as much as a change of magnitude.

Figures 3 and 2 visualise the two quantities. The slopegraph (Figure 3) devotes one panel to each approach and traces every feature group's global→local change in direction, with the *WEAVE* model drawn solid and the published one dashed; colour identifies the feature group, and a crossing of the 50% rule marks a group that changes which way it points for most methods. The heatmap (Figure 2) shows how much attribution each group carries, one row per group and one column per model, split into global and local sub-panels, so that a change in how concentrated a model is appears as a shift in colour between its two columns.

Dev-Assist

Restricted to the three categories GXA annotates, both Dev-Assist models are the least stable in the study and they fail in almost the same way. For the *WEAVE*-selected model, four groups shift significantly and three of them cross the neutral point. Method name moves from pointing towards the SRM class for 40% of training methods against 56% of GXA methods, $z = +6.9$), return type (43% → 57%, $z = +6.1$) and method modifiers (44% → 55%, $z = +4.6$), with class name moving the other way without crossing (46% → 30%, $z = -6.8$). Parameters stay the largest group throughout and grow slightly (39.0% → 41.3% of |SHAP|) while pointing away from the class slightly more often (47% → 42%). Class name meanwhile shrinks to near-nothing in magnitude (10.1% → 1.9%). The model does not so much re-weight its evidence as change its reading of it.

The published model shifts on five groups, with both modifier groups crossing the neutral point. Class modifiers move from pointing towards the class for 45% of methods to 65% ($z = +9.5$) and method modifiers from 37% to 61% ($z = +11.0$), while class name (43% → 34%, $z = -4.3$), data-flow (64% → 54%, $z = -4.3$) and method name (38% → 34%, $z = -2.1$) drift away from it. Parameters again hold their place and grow (35.7% → 47.4% of |SHAP|), and return type holds both rank and direction (13.2% → 12.9% of |SHAP|, 40% → 37%). Both variants also narrow in magnitude, from 4.6 to 4.0 and from 5.0 to 3.5 effective groups (Table 4), so unlike *WEAVE*-SSCM the instability here shows up in magnitude as well as direction. That the two pipelines share both the failure and its shape, despite differing in classifier, points at the training data rather than at the model family. Whatever makes parameter counts and method names look like evidence of security relevance in *SRM-Dataset* does not hold on GXA.

Locally (Figure 2d) the published model's attribution concentrates on parameters and method name, which together carry 63.6% of it, with return type a distant third at 12.9%. This narrowing is consistent with the behaviour of LMT-based pipelines, which can focus sharply on a subset of features when making individual predictions.

SSCM

The *WEAVE*-selected SSCM model keeps its dominant groups almost exactly as they were. Method name and parameters carry the same weight in both views (44.8% →

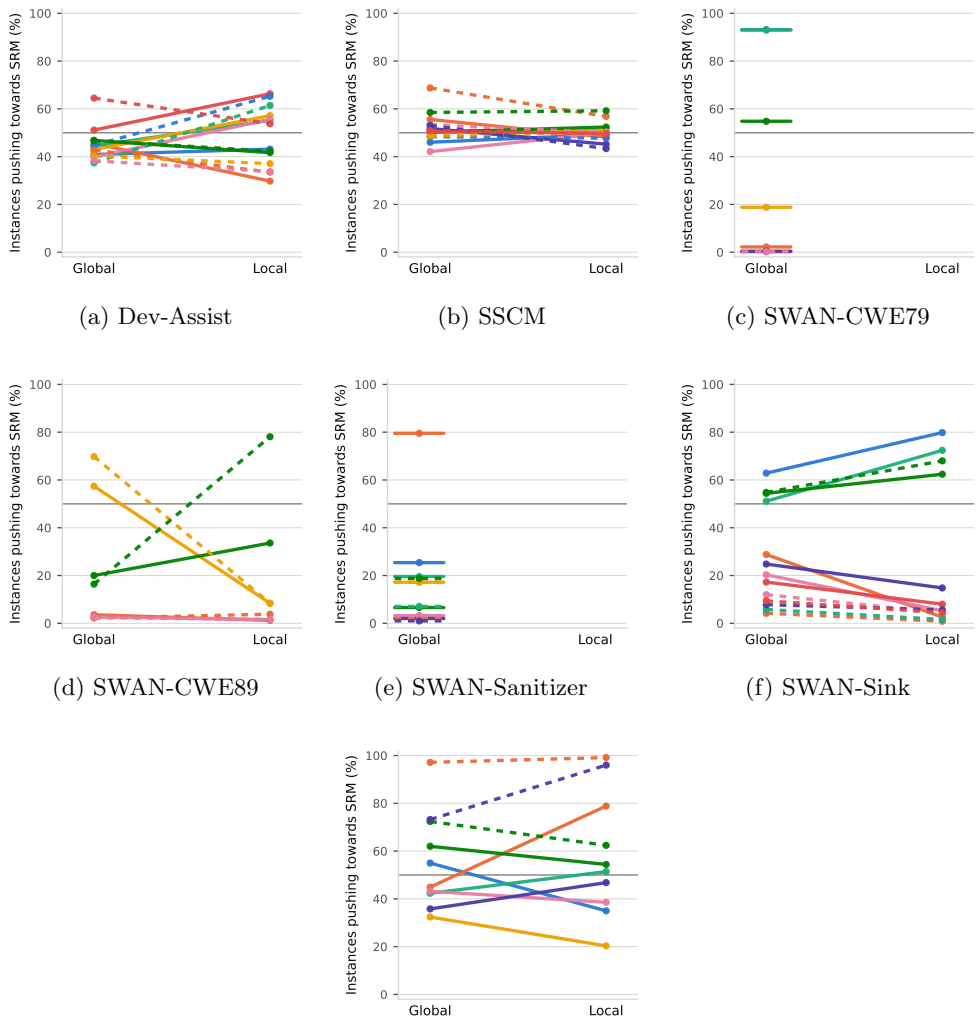


Fig. 3: Global→local change in each feature group's signed share of |SHAP|, per approach, for the WEAVE (solid) and published (dashed) models. A line crossing zero indicates a group that *reverses sign*, switching between pushing predictions towards and away from the SRM class.

43.4% and 28.7% → 31.5% of |SHAP|), parameters also keep their direction (50% → 52% of methods pointing towards the security-relevant classes), and data-flow grows while pointing the same way (12.6% → 18.7%, 51% → 50%). Three groups do shift significantly. Method body and class name each cross the neutral point, from 52% to 45% ($z = -2.4$) and from 56% to 48% ($z = -2.7$), and both shrink to near-nothing in magnitude as they do so (5.8% → 1.4% and 3.9% → 1.6%). Method name moves eight points to sit on the neutral point exactly (42% → 50%, $z = +2.9$) at undiminished weight, so the group the model leans on hardest stops favouring either answer. Its effective number of groups barely moves (3.3 → 3.1, Table 4), which is why a magnitude-only view of this model would have looked stable.

The published SSCM model shows the same shape with the weights redistributed (Figures 2c and 2d). Method name and parameters again dominate, with parameters overtaking method name locally (29.4% → 37.8% against 32.5% → 32.1% of |SHAP|), and data-flow again grows (16.4% → 21.5%); none of the three changes direction significantly. Its two significant shifts are class name, which moves twelve points without leaving its side of neutral (69% → 57%, $z = -4.4$) at negligible weight, and method body, which crosses (53% → 43%, $z = -3.5$) while collapsing from 15.2% to 1.6% of |SHAP|. Method body is the group that records what a method does, such as whether it invokes a known sink, and both SSCM variants shed it on unseen code, in weight and in direction together. The change here is narrower than in Dev-Assist, since the groups carrying most of the attribution keep their reading, but what the two models let go of is the one kind of evidence that renaming cannot touch.

SWAN

Local explanations on the GXA instances (Figure 2b) reveal a shift in emphasis. When classifying unseen methods, the models rely most heavily on parameter features, though not consistently in the same direction. For *WEAVE*'s SWAN models parameters are the largest local group throughout (39.3% for CWE89, 65.8% for Source, 44.8% for Sink of |SHAP|), yet they point towards the target class for only 34% of methods on CWE89 against 54% on Source and 62% on Sink. The published SWAN models are the more concentrated. Their global explanations (Figure 2c) are dominated by a small subset of 1–3 feature groups accounting for at least 80% of the attribution in four of the five models, the exception being SWAN-Sink (67%). Typically, parameter features and, in some cases, method name are almost exclusively used to decide whether a method is security-relevant. The clearest case is the published CWE79 model, which draws 100% of its global attribution from the method-name group alone (Figure 2c) and does so *negatively*. It is a pure name detector that only ever rules methods out, which is consistent with its near-zero positive-class recall (0.08, Section 5.1.3) despite a micro-F1 of 0.99.

RQ3 Summary:

On previously unseen methods the same fixed models distribute their evidence differently. The groups that carry the most attribution change, and for 43 of the 59 group-model pairs the direction of the group changes as well, measured as the share of methods for which it points towards the SRM class; fourteen of those cross the halfway mark, eight towards the class and six away from it. Because attribution is computed per instance and averaged over whichever methods are explained, this is what a fixed decision function meeting a new input distribution looks like, not a change inside the model. The sample also shows the limit of what these explanations describe, since no model scores a single genuine CWE89 method above the decision threshold on GXA, so its CWE89 attributions describe a detector predicting the negative class throughout.

6 Discussion

WEAVE highlights systematic strengths and weaknesses in current ML4SRM approaches that align with known pitfalls in ML for security [39]. We discuss key findings regarding the dataset, model selection and evaluation, feature engineering, transferable lessons, and the role of LLMs.

6.1 Implications for Machine Learning Security-Relevant Methods Detection

6.1.1 Dataset

Despite substantial improvements, *SRM-Dataset* remains subject to several challenges such as sampling bias and label noise. Sampling bias [39] is likely, as the dataset does not accurately reflect the distribution of SRMs in real-world projects, which can inflate apparent performance. Label imbalance allows models to achieve high overall scores while underperforming on minority CWE classes. Label noise is unavoidable, and even with multi-reviewer validation, some SRMs are mislabelled or missing, especially in the long tail of CWEs. The relatively low inter-annotator agreement measured with Cohen's kappa [31], $\kappa = 0.19$ on newly labelled methods suggests that identifying SRMs can be a subjective and difficult task. This ambiguity in the ground truth likely reflects both the inherent complexity of the concept and limitations of our labelling guidelines. Consequently, some of the variance in model performance may be attributable to label noise rather than model behaviour alone, and the reported results should be interpreted with this uncertainty in mind.

6.1.2 Model Selection & Evaluation

AutoML ($ML^{(2)}$-Plan) provides a systematic, resource-aware exploration of the model search space. On *SRM-Dataset*, it consistently identifies ensemble-based pipelines that match or improve state-of-the-art ML4SRM models, with low variance across runs. Performing a comparable search manually is infeasible given the combinatorial search space. At the same time, our explainability results show that good cross-validation

scores on imbalanced, noisy data can still hide brittle behaviour and spurious correlations. Explainability thus complements standard evaluation by revealing how models achieve their performance and whether they are likely to generalize [26].

6.1.3 Feature Engineering

More than half of the current ML4SRM features are simple bag-of-words representations grounded in a fixed security vocabulary, without using robust natural language processing (NLP) techniques. This makes them brittle to deviations from naming conventions [50] and to deliberate name-based obfuscation [51]. Lemmatization, richer tokenization, and part-of-speech tagging [52] would provide more robust lexical features; beyond that, models should rely more heavily on structural and semantic representations (abstract syntax tree (AST), control flow graphs (CFG), data-flow) that are invariant under renaming [53]. Our explanations show this is not merely an opportunity but a current blind spot. The AutoML-selected models draw $\leq 6\%$ of their attribution from method-body features (e.g., whether a method invokes a known sink) or five of the seven approaches, leaning instead on names and parameters. These are precisely the signals most exposed to obfuscation and distribution shift. Complementary remedies are to down-weight misleading groups through feature selection and to train name-masked variants that force the model to rely on structural and behavioural information. A model that detects a sink only when it is named `readPassword()` but fails on `func_a()` behaves more like a search engine than a security analysis tool.

6.2 Lessons for Machine Learning in Software Security

Although our study targets ML4SRM, both the methodology and several findings transfer to machine learning for software security more broadly, to vulnerability detection, malware classification, and bug finding that may train classifiers on imbalanced, weakly labelled code and report aggregate metrics. We distil five transferable lessons.

Lesson 1: Aggregate metrics can hide brittle reasoning

A micro-F1 of 0.99 co-occurred with 8–10% positive-class recall (CWE79) and with models whose feature reliance changes direction on unseen code. Held-out accuracy alone certifies neither minority-class competence nor robust reasoning, so per-class metrics and explanations should be reported alongside it.

Lesson 2: Compare global and local explanations, not just global importance.

The divergence between training-time (global) and prediction-time (local) attributions was our strongest indicator of poor transfer, and reporting only global feature importance, as is common, would have hidden it. Any security classifier intended for deployment should be explained on unseen inputs, not only on its training distribution.

Lesson 3: AutoML optimises the metric, not the reasoning.

We evaluated an AutoML-selected pipeline alongside each published one as a control on precisely this question (Section 5). If brittle feature reliance were an artefact of the

small-scale, manual model selection typical of prior ML4SRM work, a proper search of the pipeline space would have removed it. It did not. AutoML produced models with significantly higher F1 on most detection targets, yet the explanations show it did not remove shortcut reliance and, for some approaches, merely redistributed it. The failure mode is a familiar one under a different name. An AutoML search is an optimiser pointed at a proxy, and cross-validated F1 on an imbalanced, noisy dataset is a poor proxy for security-relevant reasoning, so the search is free to buy score by leaning harder on whatever shortcut the feature set affords. This is Goodhart's law as it appears in machine learning, the same mechanism as reward hacking in reinforcement learning, where an agent optimises the measured objective at the expense of the intended one [54], and it is what makes shortcut learning [55] survive a better search rather than being eliminated by it. The difference from reward hacking proper is that nothing here is adversarial or emergent, because the shortcut is supplied by the feature set, so it is fixable at the feature level rather than requiring a better objective. Higher scores must not be read as better-grounded models. AutoML and explainability are complementary, not substitutes, and the bottleneck the control exposes is the feature set rather than the choice of pipeline over it.

Lesson 4: Treat lexical shortcuts as an adversarial risk.

Models leaned on identifier tokens (e.g., class names containing `SQL`/`JDBC`, method-name prefixes) that vanish under renaming or obfuscation, an attacker-controllable surface. Renaming-invariant features (AST, control-/data-flow), obfuscation stress tests, and adversarial negatives (e.g., `public static` non-SRMs) should be standard practice rather than optional.

Lesson 5: Under noisy labels, explanations are more trustworthy than metrics.

With inter-rater agreement of only $\kappa = 0.19$, absolute accuracy figures are upper bounds, whereas feature-reliance findings hold regardless of whether an individual label is correct. When ground truth is contested, a common in challenge security, XAI is a more stable basis for comparison than headline scores.

Concretely, we recommend that empirical ML-for-software-security studies (i) report per-class and class-balanced metrics, not just aggregate F1; (ii) use repeated cross-validation with significance tests rather than single splits; (iii) publish global *and* local explanations; (iv) stress-test models against renaming and obfuscation; and (v) quantify label noise and interpret results against it.

6.3 Implications for LLM-based Security-Relevant Methods Detection

Given the advancements with Large language models, a natural question is whether LLMs make dedicated ML4SRM detection obsolete. The explainability results suggest that the the properties that distinguish ML4SRM from LLM4SRM become more valuable, not less, once we know that learned detectors can be brittle. These can be summarized under three areas, auditability, determinism, and reusability.

Auditability is the clearest example. Our study could pinpoint why the models fail (reliance on name tokens, the near-unused method-body features, and directions of reliance that shift between training and unseen code) precisely because ML4SRM models expose feature-level attributions. The same diagnosis is far harder for LLMs, which offers no native global/local feature attribution and whose weights and training data are typically undisclosed. Auditability is therefore what made these failures visible and, in turn, fixable at the data and feature level.

Determinism compounds this. Despite the brittleness we exposed, an SRM list is deterministic and inspectable, whereas an LLM adds stochasticity on top of opacity, drifting across model and provider updates even at temperature zero. This is an undesirable property for a security artefact that must be reproducible. Because the SRM list is moreover a reusable specification with earlier and broader coverage (an SRM-aware analysis needs only a call graph and the list, not executable code or full source), the one-time cost of auditing and repairing a detector with XAI amortises across every downstream analysis. On the other hand LLM inferences re-pays its cost on every run and leaves no reusable artefact to audit. The brittleness we found thus argues for keeping a deterministic, auditable, reusable artefact at the core, with LLMs as an auxiliary signal rather than a drop-in replacement.

This complementary role is also bounded by well-known limitations of current LLMs, independent of any accuracy comparison. They are sensitive to prompt design and weaker at fine-grained, multi-class distinctions such as CWE categories, and public code benchmarks may overlap with their pre-training data.

There is a second reading of our results in which LLMs are not a competing detector at all but a replacement for the part of the pipeline that actually failed. What this study indicts is not the classifiers, whose cross-validation performance AutoML could still improve, but the feature sets, which are hand-written bag-of-words vocabularies over identifiers, with the method body barely represented. A language model over source code learns its own representation instead, which removes the specific shortcut we document, since there is no fixed token vocabulary for a renaming to defeat, and it can attend to the body that the current feature groups all but ignore. The trade is the audit surface. Our diagnosis was possible because a feature group is a named, countable thing whose contribution can be compared between two input distributions; a learned representation offers no such handle, so the same brittleness could be present and unmeasurable. That argues for LLM-derived features feeding classifiers that remain inspectable, or for attribution methods that operate on learned representations, rather than for replacing an auditable detector with an opaque one. It also sets the honest limit of this paper, in that we show hand-crafted features are the bottleneck, not that learned features would do better, which is an empirical question this study does not settle.

Our explainability results also suggest concrete ways to apply LLMs more responsibly. First, the diagnostic that proved most informative here (comparison of global and local explanations) transfers to LLM-based detectors. Although exact Shapley values over an LLM are infeasible, sampling-based attribution can be aggregated with the same feature-group taxonomy and divergence metric to check whether an LLM relies

on the same lexical shortcuts. Second, because we found that ML4SRM breaks precisely on name- and parameter-based shortcuts and on minority CWEs, LLM-based detectors should be stress-tested on exactly those cases (renamed or obfuscated identifiers, minority-class methods, and real code) rather than on aggregate accuracy alone. Third, a hybrid design with deterministic guardrails directly addresses the auditability gap, since ML4SRM or static analysis proposes candidate SRMs cheaply and reproducibly, the LLM explains, ranks, or adjudicates borderline cases, and the resulting labels are verified against the deterministic SRM list or a taint analysis, so the pipeline remains inspectable end to end. LLMs could further be fine-tuned or instruction-tuned on datasets such as *SRM-Dataset* to align their judgments with SRM and CWE definitions, while the SRM list remains the system of record.

Finally, beyond producing labels, LLMs can explain and justify *why* a method is (or is not) security-relevant, in natural language and with reference to concrete code fragments, similar to recent work on explaining vulnerabilities or taint flows [14]. Such explanations can help security analysts triage and prioritise findings, tool developers refine SRM definitions and heuristics, and application developers who are unaware of the security implications of particular APIs.

6.4 Threats to Validity

We summarize threats to internal, construct, external, and conclusion validity of using machine learning (ML) for computer security [39] and how we mitigated them.

6.4.1 Internal Validity

Our results depend on the correctness of *SRM-Dataset*, *WEAVE*; bugs in extraction, configuration, or aggregation could bias outcomes. We mitigate this by reusing established tools (WEKA/MEKA, Soot, JavaParser, ML$^{(2)}$-Plan, SHAP), cross-checking settings against prior work, and manually inspecting random samples. For SHAP, we use stratified subsamples due to computational constraints. Different samples might slightly change importance rankings; we therefore focus on stable, high-level patterns (feature groups). We also report SHAP values aggregated by feature group rather than per individual feature, which mitigates, though does not eliminate, the dependence of KernelExplainer estimates on the chosen background sample and on the additivity assumption underlying Shapley attributions. A related limitation concerns how we report direction. Summing signed per-method contributions within a group yields a quantity that nearly cancels, a median of 4% of the group's own magnitude globally and 0.1% on GXA, so its sign cannot be estimated reliably at these sample sizes; earlier drafts of this analysis reported such signs. We therefore report direction as the fraction of methods for which a group's contribution is positive, which is well estimated here but coarser, in that it says how often a group points towards the SRM class, not by how much.

6.4.2 Construct Validity

Security-relevant methods are defined via *SRM-Dataset*, which may still contain mislabelled or missing SRMs, especially for rare CWEs, despite multi-reviewer checks.

The low inter-rater agreement ($\kappa = 0.19$, Section 4) is the sharpest form of this threat, because it bounds how confidently any reported metric or attribution can be deemed "correct" against the labels, so RQ1 performance numbers should be read as upper bounds under ambiguous ground truth rather than as estimates of true accuracy. The explainability findings (RQ2, RQ3) are comparatively robust to this noise, because they describe which feature groups a model relies on regardless of whether its predicted label is ultimately correct. This directly affects training, model selection, cross-validation, and explainability. Precision, recall, and F1 are standard but only approximate the true security impact of misclassification. SHAP values indicate correlations under a given model and background set, not causation, so we interpret them as behavioural signals.

6.4.3 External Validity

SRM-Dataset, though larger and cleaner than previous SRM datasets, may not fully reflect the diversity of real-world Java projects, frameworks, and vulnerability patterns; OWASP Benchmark is a synthetic suite. Results may not generalize to other languages, domains, or industrial-scale code bases. We also restrict ourselves to WEKA/MEKA-compatible algorithms, a specific AutoML framework ($ML^{(2)}$-Plan), and a particular set of LLM families and versions available at evaluation time. The AutoML budget (64 cores, 8-hour timeout; Section 5.1.2) bounds the pipeline space searched and hence the RQ1 selection, though it is unlikely to change the qualitative RQ2/RQ3 findings, which characterise which feature groups any well-performing model relies on rather than which classifier is best. The local analysis rests on a single subject program, and that bounds the direction-shift counts specifically. GXA is a bioinformatics web application whose identifiers (gene, assay, experiment, ontology) share little vocabulary with the security-flavoured tokens the models were trained on, so the shift in identifier-derived features is plausibly larger here than it would be on, say, an e-commerce or CMS codebase, and the number of groups whose direction shifts should be read as a property of this pairing rather than as a rate that transfers. The mechanism behind it does transfer, because attributions are computed per instance and averaged over whichever methods are explained (Section 5.2.2), so any change of input distribution can move them, and a codebase whose vocabulary matches *SRM-Dataset* more closely would be expected to shift less, not differently. New datasets, tools, or model generations may behave differently, so our findings should be seen as indicative, not exhaustive.

6.4.4 Conclusion Validity

We compare models using repeated cross-validation over 100 seeds and report paired Wilcoxon signed-rank tests for the *WEAVE*-versus-published comparison (Section 5.1.3). Because the across-seed variance is very small ($\sigma \leq 0.03$), even differences of a few thousandths reach statistical significance; we therefore distinguish statistical significance from practical importance and emphasise only the larger, consistent effects (SSCM, Sink, Source) rather than treating every significant gap as meaningful. To reduce over-interpretation, we emphasize consistent, larger effects (e.g., ensembles vs. SVMs) and report both micro- and macro-F1. For SHAP, we

aggregate over instances and feature groups to smooth variance, but individual plots remain sample- and model-dependent; we focus on recurring trends (e.g., reliance on names/bodies vs. modifiers/data-flow) rather than fine-grained differences.

7 Related Work

We situate this work along four axes, namely, ML for SRM detection, the static and taint analysis frameworks that consume SRM lists, LLM-based security analysis, and XAI in software security.

7.1 ML for security-relevant method detection

The approaches we explain in this paper build on a line of ML-based SRM detectors. SuSi [5] detects Android sources and sinks using 26 feature types over two classification stages. SWAN [2] extends SuSi to general Java code and adds sanitizer and authentication categories with CWE-aware classifiers, replacing SuSi's second stage. SSCM [4] generalises SRM detection to Java libraries and motivates CWE classes, modifying SuSi's features to that end. Dev-Assist [3] adopts a multi-label formulation with ML^2-Plan-based AutoML. Beyond these, SinkFinder [56] learns sink *pairs* for C/C++ via sub-word embedding from a seed pair, and Merlin [57] infers SRMs for string-based vulnerabilities through probabilistic inference over propagation graphs rather than ML. The same gap has been documented by an approach that abandons hand-crafted features altogether. CoDoC [28] learns its representation from a method's source code and documentation, reaches precision, recall and F1 of 91% in 10-fold cross-validation on Android API methods, and then reports false positives above 90% for sources on unseen methods. Its authors read this as evidence that models of abstract concepts such as privacy fail in practice despite good laboratory results, attributing the failure to the concept and to the ground truth; we locate it instead in what the models attend to, and can say so per feature group because the models we study expose attributions.

Each of these evaluates with single-round cross-validation on its own dataset and provides no systematic post-hoc explanation of which features drive predictions, so their claims about generalization, which the authors themselves flag as fragile [2, 4], rest on aggregate metrics alone. We re-examine the three Java approaches under a uniform evaluation and explainability harness to explain that generalization gap rather than to propose a new classifier.

7.2 Static Analysis Frameworks consuming SRM Lists

SRM lists are the primary input to a range of analysis tools. Taint analyses such as those evaluated in TaintBench [16] and rule-based checkers like SecuCheck [17] consume SRMs to configure flow- and context-sensitive detection, and the accuracy of such a configuration is bounded by the accuracy of the SRM list it consumes [13], the very motivation for automating SRM detection. Because a mislabelled SRM propagates simultaneously to every consumer (Section 2), explaining the model that produces the

labels is as important as the labels themselves. Stöckle et al. [58] take a complementary natural language processing (NLP) route, identifying security-relevant *settings* via sentiment analysis and topic modelling rather than methods.

7.3 LLM-based security analysis

A growing body of work applies LLMs to vulnerability detection, code review, and explanation. Johnson et al. [14] use LLMs to explain detected vulnerabilities and taint flows in natural language, providing developer-facing rationale rather than another detector. Krishnamurthy et al. [59] extend this direction from detection to prevention, using lightweight code metrics to flag security-critical methods before a vulnerability is introduced and an LLM to generate prevention-oriented explanations for that code. Our work is orthogonal to both, in that we do not propose an LLM-based SRM detector, and use LLM observations only to discuss practicality and complementarity (Section 6). Crucially, LLM-based detectors face the same opacity problem we identify for ML4SRM, arguably a worse one, given undisclosed training data and behaviour that drifts across provider updates, so the XAI methodology applied here is, in principle, transferable to LLM-derived classifiers.

7.4 Explainable AI in software security

Recent work has applied eXplainable AI to a range of software security tasks, including vulnerability, malware, and intrusion detection. Sotgiu et al. [60] use explainability-based debugging to analyse and improve vulnerability discovery models, showing how explanations can expose hidden feature dependencies and decision boundaries. In intrusion detection, Neupane et al. [61] survey explainable intrusion detection systems (X-IDS) and emphasize the need for stakeholder-specific explanations in operational settings. For malware detection, Manthena et al. [62] review interpretable ML techniques for analysing binaries and classifying malware, while Saqib et al. [63] propose a taxonomy of explainability in malware hunting and discuss the risk that explanations may aid adversaries.

By contrast, explainability for SRM detection has been largely unexplored. Existing ML4SRM approaches rely on feature selection over large hand-crafted feature sets to control overfitting [2] but rarely give systematic insight into how individual features or feature groups drive predictions. Given the size and heterogeneity of SRM feature spaces and the downstream security implications of misclassification, understanding feature contribution is essential. This paper closes that gap by applying SHAP-based global and local explanations to three state-of-the-art ML4SRM models, on a consolidated dataset, under a uniform evaluation protocol.

8 Conclusion & Future Work

This paper examined why machine learning models for security-relevant method detection succeed in cross-validation yet fail to generalise. To study the three published ML4SRM approaches (Dev-Assist, SSCM, SWAN) on equal footing, we built *WEAVE*, an integrated AutoML, repeated cross-validation, and SHAP explainability harness

for WEKA/MEKA pipelines. The explainability experiments use an assembled *SRM-Dataset*, a consolidated collection of 10,180 labelled Java methods used as a uniform training and evaluation substrate. A preliminary model-identification study (RQ1) confirmed that AutoML explores a richer space of pipelines and tends to select ensembles that match or modestly exceed the published models on imbalanced data; this established the models we then explained.

The central finding (RQ2 and RQ3) is a divergence between global and local explanations. Globally, ML4SRM models capture meaningful correlations from method names and bodies; locally, when classifying previously unseen methods, the feature groups they rely on shift in both weight and direction. Measuring direction as the share of methods for which a group pushes towards the security-relevant class, 43 of 59 group-model pairs shift significantly and 14 change which way they point for most methods, with modifier groups becoming markedly more inclusive on unseen code for both Dev-Assist models. This instability of feature reliance explains the documented generalization failure of ML4SRM in practice and locates the bottleneck in feature engineering rather than in the choice of classifier. The same opacity affects LLM-based detectors, so the explainability methodology, not any single classifier, is the durable contribution.

8.1 Future Work

As future work, we plan to:

- **Break the shortcut.** The most actionable finding is that modifier-based shortcuts dominate at prediction time. Future ML4SRM work should stress-test models against renamed and obfuscated identifiers, add adversarial negatives that share surface modifier patterns with SRMs, and weight structural and data-flow features more heavily during feature selection.
- **Learn the representation instead of writing it.** Our results locate the bottleneck in the feature sets rather than in the classifiers over them, which makes representation learning the natural next step. Graph neural networks over ASTs and control- and data-flow graphs, or pre-trained code models, learn features that no renaming can defeat and that reach the method body the current groups barely use [53]. The one prior attempt in this direction is a caution rather than an encouragement. CoDoC learned its representation from method source code and documentation, reached 91% precision, recall and F1 in cross-validation, and still produced false positives above 90% for sources on unseen methods [28]. Learned features remove the lexical shortcut we document without, on that evidence, closing the gap it opens. The condition we would attach is the one this paper argues for throughout, namely that such a detector has to be evaluated on unseen code and under the obfuscation stress tests above, and paired with an attribution method that works on learned representations, or it will trade a measurable weakness for an unmeasurable one.
- **Lift the XAI methodology to LLM-based detectors.** The global-vs-local SHAP comparison can, in principle, be applied to LLM-derived SRM classifiers via

sampling-based attribution, enabling the same auditing questions of LLM-based security tooling.
- **Reduce *SRM-Dataset* label noise.** The low inter-rater agreement bounds the strength of any downstream finding; a larger expert-annotated subset under a more constrained labelling protocol (multi-CWE allowed, threat-model explicit) would tighten this bound.

Declarations

Funding. This work was supported by the German Federal Ministry of Research, Technology and Space under grant 16KIS1257 (AI-DevAssist), and by the state of North Rhine-Westphalia through its Ministry of Economic Affairs, Industry, Climate Action and Energy (MWIKE) and its Ministry of Culture and Science (MKW) under the project Cyberresilience.nrw.

Competing interests. The authors declare no competing interests.

Acknowledgements. We thank Linus Bennin for his contribution to the curation of *SRM-Dataset*.

Data availability. *SRM-Dataset* is available at [12]. The repository holds the catalogue of 10,180 labelled Java methods described in Section 4 as JSON, with each record carrying the method signature and parameters, its SRM and CWE labels, the associated documentation, and the project or JAR it was discovered in, which is the provenance Table 1 summarises. The parser that produces the dataset is included.

Code availability. *WEAVE* is available at [11], together with setup scripts that reproduce its environment. Its `evaluation/ml4srm` directory holds the material needed to repeat the experiments for each approach, namely the training sets and the GXA explanation sets in WEKA's ARFF format, the experiment configurations, the extracted method signatures, and the runner scripts for each SRM category. It also holds the SHAP outputs underlying **RQ2** and **RQ3**, including the per-instance attribution files from which the feature-group shares, the direction statistic and every percentage we report in those sections are computed, so that our aggregation can be checked rather than taken on trust. The scripts that turn those outputs into the results (heatmap, the slopegraph and the concentration table) reported here are released alongside them.

Author contributions. O.J. wrote the main manuscript text and prepared the figures, with R.K.S. R.K. and G.P. drafting individual sections. O.J., R.K.S. and R.K. designed the methodology, developed the tooling, curated the dataset and ran the experiments. O.J., R.K.S. and G.P. analysed the results, and R.K.S. validated them. R.K.S., G.P. and E.B. revised and edited the text. All authors reviewed the manuscript.

References

[1] National institute of standards and technology(NIST): Published CVE Records (2026). https://www.cve.org/about/Metrics

[2] Piskachev, G., Do, L.N.Q., Bodden, E.: Codebase-adaptive detection of security-relevant methods. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2019, pp. 181–191. Association for Computing Machinery, New York, NY, USA (2019)

[3] Johnson, O., Piskachev, G., Krishnamurthy, R., Bodden, E.: Detecting security-relevant methods using multi-label machine learning. In: Proceedings of the 1st ACM/IEEE Workshop on Integrated Development Environments. IDE '24, pp. 101–106. Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3643796.3648464 . https://doi.org/10.1145/3643796.3648464

[4] Sas, D., Bessi, M., Fontana, F.A.: Automatic detection of sources and sinks in arbitrary java libraries. In: 2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 103–112 (2018)

[5] Arzt, S., Rasthofer, S., Bodden, E.: Susi: A tool for the fully automated classification and categorization of android sources and sinks. In: Network and Distributed System Security Symposium 2013. NDSS'13 (2013)

[6] Hutter, F., Kotthoff, L., Vanschoren, J.: Automated Machine Learning - Methods, Systems, Challenges, (2019). https://doi.org/10.1007/978-3-030-05318-5

[7] Kim, J.-H.: Estimating classification error rate: Repeated cross-validation, repeated hold-out and bootstrap. Computational Statistics & Data Analysis **53**(11), 3735–3745 (2009) https://doi.org/10.1016/j.csda.2009.04.009

[8] Gunning, D., Aha, D.: Darpa's explainable artificial intelligence (xai) program. AI magazine **40**(2), 44–58 (2019)

[9] Angelov, P.P., Soares, E.A., Jiang, R., Arnold, N.I., Atkinson, P.M.: Explainable artificial intelligence: an analytical review. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery **11**(5), 1424 (2021)

[10] Lundberg, S.M., Lee, S.-I.: A unified approach to interpreting model predictions. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. NIPS'17, pp. 4768–4777. Curran Associates Inc., Red Hook, NY, USA (2017)

[11] Johnson, O., Shanmuganathan, R.K., Krishnamurthy, R.: WEKA Automated eValuator & Explainer (WEAVE) (2026). https://github.com/fraunhofer-iem/weave

[12] Johnson, O., Krishnamurthy, R., Bennin, L., Shanmuganathan, R.K.: Security Relevant Methods Dataset (2026). https://github.com/secure-software-engineering/srm-dataset

[13] Piskachev, G., Becker, M., Bodden, E.: Can the configuration of static analyses make resolving security vulnerabilities more effective?-a user study. Empirical Software Engineering **28**(5), 118 (2023)

[14] Johnson, O., Fomina, A., Krishnamurthy, R., Chaudhari, V., Shanmuganathan, R.K., Bodden, E.: Explaining software vulnerabilities with large language models. In: 2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pp. 194–198 (2025). IEEE

[15] Atıl, B., Aykent, S., Chittams, A., Fu, L., Passonneau, R.J., Radcliffe, E., Rajagopal, G.R., Sloan, A., Tudrej, T., Ture, F., Wu, Z., Xu, L., Baldwin, B.: Non-determinism of "deterministic" LLM system settings in hosted environments. In: Akter, M., Chowdhury, T., Eger, S., Leiter, C., Opitz, J., Çano, E. (eds.) Proceedings of the 5th Workshop on Evaluation and Comparison of NLP Systems, pp. 135–148. Association for Computational Linguistics, Mumbai, India (2025). https://doi.org/10.18653/v1/2025.eval4nlp-1.12 . https://aclanthology.org/2025.eval4nlp-1.12/

[16] Luo, L., Pauck, F., Piskachev, G., Benz, M., Pashchenko, I., Mory, M., Bodden, E., Hermann, B., Massacci, F.: Taintbench: Automatic real-world malware benchmarking of android taint analyses. Empirical Softw. Engg. **27**(1) (2022) https://doi.org/10.1007/s10664-021-10013-5

[17] Piskachev, G., Krishnamurthy, R., Bodden, E.: Secucheck: Engineering configurable taint analysis for software developers. In: 2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 24–29 (2021). https://doi.org/10.1109/SCAM52516.2021.00012

[18] Witten, I.H., Frank, E., Hall, M.A., Pal, C.J.: Data Mining, Fourth Edition: Practical Machine Learning Tools and Techniques, 4th edn. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2016)

[19] Wever, M., Mohr, F., Hüllermeier, E.: Automated Multi-Label Classification based on ML-Plan (2018)

[20] Mohr, F., Wever, M., Hüllermeier, E.: Ml-plan: Automated machine learning via hierarchical planning. Machine Learning **107**(8-10), 1495–1515 (2018)

[21] Read, J., Reutemann, P., Pfahringer, B., Holmes, G.: MEKA: A multi-label/multi-target extension to Weka. Journal of Machine Learning Research **17**(21), 1–5 (2016)

[22] starlibs: jaicore-experiments. https://starlibs.github.io/AILibs/projects/

jaicore-experiments/

[23] Alonso, J.M., Bugarín, A.: Expliclas: automatic generation of explanations in natural language for weka classifiers. In: 2019 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), pp. 1–6 (2019). IEEE

[24] Ribeiro, M.T., Singh, S., Guestrin, C.: " why should i trust you?" explaining the predictions of any classifier. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 1135–1144 (2016)

[25] Salih, A.M., Raisi-Estabragh, Z., Galazzo, I.B., Radeva, P., Petersen, S.E., Lekadir, K., Menegaz, G.: A perspective on explainable artificial intelligence methods: Shap and lime. Advanced Intelligent Systems **7**(1), 2400304 (2025) https://doi.org/10.1002/aisy.202400304

[26] Linardatos, P., Papastefanopoulos, V., Kotsiantis, S.: Explainable ai: A review of machine learning interpretability methods. Entropy **23**(1) (2021) https://doi.org/10.3390/e23010018

[27] Lundberg, S.: API Reference (2018). https://shap.readthedocs.io/en/latest/api.html

[28] Samhi, J., Kober, M., Kabore, A.K., Arzt, S., Bissyandé, T.F., Klein, J.: Negative results of fusing code and documentation for learning to accurately identify sensitive source and sink methods: An application to the android framework for data leak detection. In: 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 783–794 (2023). IEEE

[29] OWASP: OWASP Benchmark. https://owasp.org/www-project-benchmark/. Online; accessed January 2020 (2020)

[30] Krishnamurthy, R., Piskachev, G., Bodden, E.: To what extent can we analyze kotlin programs using existing java taint analysis tools? In: 2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM), pp. 230–235 (2022). IEEE

[31] Cohen, J.: A coefficient of agreement for nominal scales. Educational and Psychological Measurement **20**(1), 37–46 (1960) https://doi.org/10.1177/001316446002000104 https://doi.org/10.1177/001316446002000104

[32] Bruggen, D.: JavaParser. https://github.com/javaparser/javaparser Accessed 2024-07-09

[33] Institute, E.B.: Gene Expression Atlas. https://github.com/gxa/gxa

[34] Campione, M., Walrath, K.: The Java Tutorial: Object-oriented Programming for

the Internet. Addison-Wesley Longman Publishing Co., Inc., ??? (1996)

[35] Bauer, C., Kenter, T., Lass, M., Mazur, L., Meyer, M., Nitsche, H., Riebler, H., Schade, R., Schwarz, M., Winnwa, N., al.: Noctua 2 supercomputer. Journal of large-scale research facilities **9** (2024) https://doi.org/10.17815/jlsrf-8-187

[36] Pendleton, M., Garcia-Lebron, R., Cho, J.-H., Xu, S.: A survey on systems security metrics. ACM Comput. Surv. **49**(4) (2016) https://doi.org/10.1145/3005714

[37] Naidu, G., Zuva, T., Sibanda, E.M.: A review of evaluation metrics in machine learning algorithms. In: Silhavy, R., Silhavy, P. (eds.) Artificial Intelligence Application in Networks and Systems, pp. 15–25. Springer, Cham (2023)

[38] Nachtigall, M., Schlichtig, M., Bodden, E.: A large-scale study of usability criteria addressed by static analysis tools. In: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 532–543. ACM, South Korea (2022). https://doi.org/10.1145/3533767.3534374

[39] Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., Rieck, K.: Dos and don'ts of machine learning in computer security. In: Proc. of the USENIX Security Symposium 2022 (2022). 46.23.01; LK 01

[40] Feng, W., Huang, W., Ren, J.: Class imbalance ensemble learning based on the margin theory. Applied Sciences **8**(5) (2018) https://doi.org/10.3390/app8050815

[41] Wever, M., Tornede, A., Mohr, F., Hüllermeier, E.: Automl for multi-label classification: Overview and empirical evaluation. IEEE transactions on pattern analysis and machine intelligence **43**(9), 3037–3054 (2021)

[42] Woolson, R.F.: Wilcoxon signed-rank test. Wiley encyclopedia of clinical trials, 1–3 (2007)

[43] Lundberg, S.: Welcome to the SHAP documentation (2018). https://shap.readthedocs.io/en/latest/index.html

[44] SHAP developers: shap.KernelExplainer. SHAP API documentation for the kernel explainer, including its guidance on summarising large background datasets. https://shap.readthedocs.io/en/latest/generated/shap.KernelExplainer.html Accessed 2026-08-27

[45] SHAP developers: shap.utils.sample. SHAP API documentation for the sampling utility provided for downsampling a background dataset. https://shap.readthedocs.io/en/latest/generated/shap.utils.sample.html Accessed 2026-08-27

[46] SHAP developers: KernelExplainer implementation,

`shap/explainers/_kernel.py`. Kernel explainer implementation; warns when the background dataset exceeds 100 rows and recommends summarising it. https://github.com/shap/shap/blob/master/shap/explainers/_kernel.py Accessed 2026-08-27

[47] Yuan, H., Liu, M., Kang, L., Miao, C., Wu, Y.: An empirical study of the effect of background data size on the stability of SHapley Additive exPlanations (SHAP) for deep learning models (2023). https://arxiv.org/abs/2204.11351

[48] Simpson, E.H.: Measurement of diversity. nature **163**(4148), 688–688 (1949)

[49] Hill, M.O.: Diversity and evenness: a unifying notation and its consequences. Ecology **54**(2), 427–432 (1973)

[50] Butler, S., Wermelinger, M., Yu, Y.: Investigating naming convention adherence in java references. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 41–50 (2015). https://doi.org/10.1109/ICSM.2015.7332450

[51] Xu, W., Zhang, F., Zhu, S.: Jstill: mostly static detection of obfuscated malicious javascript code. In: Proceedings of the Third ACM Conference on Data and Application Security and Privacy. CODASPY '13, pp. 117–128. Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2435349.2435364 . https://doi.org/10.1145/2435349.2435364

[52] Manning, C.D., Surdeanu, M., Bauer, J., Finkel, J., Bethard, S.J., McClosky, D.: The Stanford CoreNLP natural language processing toolkit. In: Association for Computational Linguistics (ACL) System Demonstrations, pp. 55–60 (2014). http://www.aclweb.org/anthology/P/P14/P14-5010

[53] Siow, J.K., Liu, S., Xie, X., Meng, G., Liu, Y.: Learning program semantics with code representations: An empirical study. In: 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 554–565 (2022). https://doi.org/10.1109/SANER53432.2022.00073

[54] Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., Mané, D.: Concrete problems in ai safety. arXiv preprint arXiv:1606.06565 (2016)

[55] Geirhos, R., Jacobsen, J.-H., Michaelis, C., Zemel, R., Brendel, W., Bethge, M., Wichmann, F.A.: Shortcut learning in deep neural networks. Nature Machine Intelligence **2**(11), 665–673 (2020)

[56] Bian, P., Liang, B., Huang, J., Shi, W., Wang, X., Zhang, J.: SinkFinder: Harvesting Hundreds of Unknown Interesting Function Pairs with Just One Seed, pp. 1101–1113. Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3368089.3409678

[57] Livshits, B., Nori, A.V., Rajamani, S.K., Banerjee, A.: Merlin: Specification inference for explicit information flow problems. SIGPLAN Not. **44**(6), 75–86 (2009)

[58] Stöckle, P., Wasserer, T., Grobauer, B., Pretschner, A.: Automated identification of security-relevant configuration settings using nlp. (2022)

[59] Krishnamurthy, R., Johnson, O., Piskachev, G., Bodden, E.: From detection to prevention: Explaining security-critical code to avoid vulnerabilities. In: Proceedings of the 3rd ACM/IEEE International Workshop on Integrated Development Environments, pp. 46–49 (2026)

[60] Sotgiu, A., Pintor, M., Biggio, B.: Explainability-based debugging of machine learning for vulnerability discovery. In: Proceedings of the 17th International Conference on Availability, Reliability and Security, pp. 1–8 (2022)

[61] Neupane, S., Ables, J., Anderson, W., Mittal, S., Rahimi, S., Banicescu, I., Seale, M.: Explainable intrusion detection systems (x-ids): A survey of current methods, challenges, and opportunities. IEEE Access **10**, 112392–112415 (2022)

[62] Manthena, V., Jang, J., Gondara, L., Bobba, R.B.: Explainable Machine Learning for Malware Analysis: A Survey. arXiv preprint arXiv:2402.06618 (2024)

[63] Saqib, M., Mahdavifar, S., Fung, B.C., Charland, P.: A comprehensive analysis of explainable ai for malware hunting. ACM Computing Surveys **56**(12), 1–40 (2024)