AutoPYara: Next-Gen YARA Rule Generator for Malware Family Clustering

Mabon Ninan*, Nhat Nguyen*, Soumyajyoti Dutta, Sidharth Anil, Marcus Botacin
Texas A&M University
{*ninanmm, nmnguy29, soumyajyoti1998, sid.anil, botacin*}*@tamu.edu*
* *Equal Contribution*

arXiv:2611.08608v1 [cs.CR] 5 Nov 2026

***Abstract*—Millions of malware variants threaten users daily, such that clustering these samples is key for fast incident response. AutoYara is the current state-of-the-art for automatic YARA rule generation, but the task remains challenging on realistic datasets, where intra-family diversity and noisy clustering can degrade rule quality. We present AutoPYara, a solution armored with: (1) precise cluster centroid selection based on similarity hashing, and (2) an improved heuristic for estimating subclusters within a malware family using historical VirusTotal data. We implement AutoPYara as an open-source Python package `pip install autopyara` to support reproducibility and practical use. We show that AutoPYara, (1) increases rule coverage by 14 percentage points; (2) reduces variance by 7.1 percentage points, comparing the average variance AutoPYara exhibits to that of AutoYara; and (3) improves threat-hunting accuracy by 10%. AutoPYara produces robust and effective YARA rules than prior automatic approaches in practical settings.**

***Index Terms*—Malware, Clustering, Similarity Hashing**

1. Introduction

Among millions of malware samples threatening users daily, most are variants of the same family. Handling these samples individually does not scale [1]; thus, clustering samples into families is key for fast incident response, as it allows the same remediation to be applied to all family samples [2], [3]. Multiple approaches have been proposed to cluster unknown samples into families, from signatures [4] to Machine Learning (ML) [5]. Despite their individual success, a one-size-fits-all solution for malware family identification remains an open problem. The current State-of-the-Art (SOTA) for this task is AutoYara [6], a hybrid approach that uses ML for determining Indicator of Compromise (IoCs) that uniquely identify the malware, and YARA [7] signatures for the actual matching, being it the industry-standard. It extracts unique N-grams of bytes from goodware and malware to create an extensive database that is further used with a biclustering [8] algorithm to identify unique byte sequences that are only seen in malware. This allows the automatic derivation of rules to match these unique patterns. Although AutoYara was a step ahead in automating rule generation, we identified key blind spots: **(1)** AutoYara was evaluated only on cases with known ground truth. In real-world cases, complete ground truth is not available (attackers do not disclose their templates), such that ground truth is just approximated by AntiVirus (AV) companies via labeling strategies. Different labeling strategies lead to different "groundtruths", hence it is essential to evaluate their impact to understand how sensitive rule generation frameworks (previous and ours) are to these classification uncertainties; **(2)** AutoYara's evaluations only considered samples with known patterns aligned to the pre-trained Bloom filters, ignoring that most in-the-wild samples have unknown patterns caused by concept evolution and byte distribution shift, causing rule generation to fail; **(3)** AutoYara was only evaluated on large cluster sizes, which tend to be more stable. In contrast, in-the-wild samples might be collected at different family sizes, and their impact on rule generation is unknown.

To characterize the impact of these factors on automatic rule generation, we conducted exhaustive experiments on a dataset of $\approx$ 71,000 samples collected over 8 years by an AV company in 3 countries: the US, Brazil, and Japan. We compared the results under different strategies to approximate ground truth: the application of two similarity hashing functions (SSdeep [9] and J-SDhash [10]); and VirusTotal (VT) labels [11]. However, both strategies introduce distinct trade-offs regarding precision and data volume. Despite producing homogeneous clusters, similarity hashing limits sample coverage and introduces methodological variables like the similarity threshold. AV labels are comprehensive and cover most of the dataset, but result in heterogeneous clusters with increased False Positives (FPs).

Our results highlight the need for: **(i)** methodical standards to evaluate rule generation with high variance and uncertain labels, which we address by introducing guidelines and best practices for experiments; and **(ii)** algorithmic solutions to address the inefficient handling of heterogeneous data with high variance. Our analysis of AutoYara revealed that: **(1)** most of the variability in the results arises from the randomness of the original algorithm, which initiates clustering from arbitrary points that may not align well with heterogeneous distributions observed in real scenarios; **(2)** the AutoYara algorithm underestimates the number of subclusters a malware family has due to the use of the Variational Bayesian Gaussian Mixture Model (VBGMM) [12] algorithm as the guesser for the number of clusters.

This manuscript is the version accepted at IEEE ACSAC 2026 and appearing in the proceedings. Another version of this work exists as an extended version with supplementary appendices.

We propose solving the problem **(1)** by adding a crafted centroid selection step to the rule generation process. This ensures that the generation always starts from the best possible centroid to cover a heterogeneous distribution of samples, and in **(2)** we develop a heuristic to replace VBGMM and increase the number of sub-clusters that the malware families are fitted into, allowing heterogeneous samples to be classified in the same family. The heuristic estimates the number of sub-clusters based on historical malware family data obtained from the VirusTotal oracle. To evaluate our contributions, we created AutoPYara, a solution that implements our proposed algorithm. AutoPYara is as efficient as AutoYara, as it shares the same Java backend. However, it is more user-friendly, as it is implemented as a Python front-end and distributed as a PyPI package. Applied to the same dataset, AutoPYara outperforms AutoYara by up to 14% [1] in its best configuration on real-world samples. Our results demonstrate the impact of centroid selection and reopen the possibility of using VT labels for rule generation—not as a labeling strategy, which we show is problematic, but as a heuristic, which we show is powerful.

2. Background

2.1. An Overview of AutoYara Design

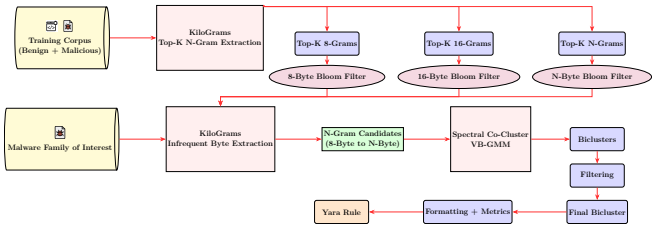


Figure 1. The AutoYara system pipeline.

AutoYara [6] generates rules based on byte patterns that are not frequent in goodware samples (to avoid FPs) and are frequent among the malware samples (to fingerprint them). AutoYara implements this logic via two steps: **(1)** creating a blacklist of goodware byte patterns during training; **(2)** identifying frequent malware-family patterns that are disjoint from previously seen goodware patterns during generation. Figure 1 provides a representation of the framework. AutoYara's training step uses KiloGrams [13] to extract the top-K byte N-grams from the dataset, producing Bloom filters [14] to store the byte blacklists. Next, the generation step extracts new N-grams from the input files and biclusters using Spectral Co-Clustering [8] (details in Appendix A.1) with VBGMM [15]. This reveals distinct byte groups that are filtered using heuristics (e.g., coverage, cluster size, low FP, etc.) and merged into a disjunction of *soft* conjunctions. The disjunctions have the form *at least X conditions match*, where each string corresponds to an extracted N-gram and each condition to a bicluster. AutoYara ships pretrained BFs for the top 1M frequent bytes from EMBER [16].

1. Presented results represent absolute gains. % often means % points.

2.2. A critical review of AutoYara Clustering

AutoYara employs unsupervised clustering [17] to group byte patterns, reducing FPs by requiring simultaneous signature matches. It aims to organize N-gram candidates into the fewest clusters covering the most samples. While the literature offers various Hard [18], [19], [20] and Soft [15], [21], [22] clustering approaches, AutoYara utilizes Gaussian Mixture Models (GMMs) as its soft clustering solution. Table 1 contrasts AutoYara with other literature methods and our proposed approach. While AutoYara addresses the manual k selection of K-Means using GMMs, it remains sensitive to initialization and assumes Gaussian distributions (often underfitting malware biclusters). Other methods like Augmented K-Means [19] offer noise resistance, and DBSCAN [20] handles non-spherical data without a preset k.

TABLE 1. COMPARISON OF CLUSTERING METHODS

Method	Type	Auto-k	Non-Gauss.†	Biclusters
K-Means [18]	Hard	×	×	×
C-Means [21]	Soft	×	×	×
DBSCAN [20]	Density	✓	✓	×
AutoYara	**Soft**	✓	×	✓
AutoPYara (base)	**Soft**	×	✓	✓
AutoPYara (heuristic)	**Soft**	✓	✓	✓

†Handles non-spherical/non-Gaussian distributions.

Hypothesis: We propose combining the benefits of Soft clustering (Fuzzy C-Means [21]) with the density-based automatic k determination of DBSCAN to overcome the initialization and distribution limitations of AutoYara.

3. AutoPYara

3.1. Design

Overview: Based on the review of AutoYara in Section 2, we identified two key aspects that could be improved: **(1)** the centroid selection process; **(2)** the k selection process; and we propose refactoring AutoYara into AutoPYara to integrate these changes. Compared to AutoYara's architecture visualized in Figure 1, the AutoPYara pipeline remains the same, but the internal biclustering algorithm is replaced by our algorithm.

Key Insights We recognize multiple positive aspects in AutoYara, but highlight that some project decisions must be rethought. Improving AutoYara's clustering is not as simple as extending it. Rather, it requires replacing algorithms that work in a paradigm with each other (Section 2). We adopted a deconstruction-plus-reconstruction approach, where we first downgraded AutoYara's algorithms to base algorithms from another paradigm, and then extended this paradigm to surpass the original ones via our observations.

To improve the k selection:

1) We believe that external threat intelligence (e.g., similarity hashing) can be used as additional context for clustering algorithms to improve biclustering performance. The intuition is that by implementing external threat intelligence and domain-specific context, we have a robust framework that is difficult to evade.

2) To add external threat intelligence into the clustering process, we need the algorithm to consider global structures and properties of the handled data.
3) To consider global structures, we must fully control the clustering process to input this data to the algorithms.
4) We cannot control the clustering structure of VBGMM, which auto-determines k. We should default back to K-Means [18] to control centroids and k.
5) Since K-Means produces hard labels [21], it is not comparable to VBGMM, which uses soft labels. We must replace the traditional K-Means with Fuzzy C-Means to use soft labels. In Fuzzy C-Means, a large m will evenly spread out the probability distribution of each sample, leading to all samples having the same probability for all other clusters. A small m will create a fast probability drop-off, which is similar to hard clustering. It provides a mechanism to create overlapping biclusters that are better at handling noise in real-world samples.
6) The final challenge is that we don't have k for clustering (it was previously auto-determined by VBGMM).To control k and improve clustering quality, we devise heuristics grounded in family labels from AvClass [11] and the frameworks' historical behavior on these families. Each sample is queried against VirusTotal using its unique file hash (SHA-256 by default). The resulting raw AV engine scan reports are passed directly to AVClass, which handles internal vendor naming conflicts, normalizes generic terms, and yields a single canonical family label per binary. Empirically, VBGMM underestimates k, so we adjust it upward to enhance clustering performance.

To improve the initial centroid selection:

1) Fuzzy C-Means is controllable and relies on soft labels; we extend it to consider external threat intelligence in the form of provided labels. It becomes AugmentedK-Means Soft (AKMS), our algorithmic contribution, extending Augmented K-Means [19] to operate on soft labels.
2) Given AKMS, we can also benefit from its heuristic predictor to select the best cluster centroids. The intuition is that for real-world applications, external information can be leveraged to produce better clusters. This external information could be provided from SSdeep [23], J-SDhash [10] hashing, or VirusTotal [24].
3) AKMS might ingest external information to guide its heuristic by clustering the provided hashes or labels with DBSCAN [20], yielding *"hint"* clusters whose centroids can initialize the algorithm. As DBSCAN does not make spherical or Gaussian assumptions and automatically clusters points based on their density, this model's real-world samples are accurate due to its arbitrary cluster-shaping and consideration of noisy outliers. No user-specified k is needed; Augmented K-means infers it from labels.

3.2. Algorithmic Challenges

Solving the Alignment Problem. Integrating AugmentedK-Means with Spectral Co-Clustering presents a dimensional alignment challenge. The AKMS algorithm requires a robust initialization and an error-ceiling hyperparameter α. The following algorithms detail the core logic and the coordinate-wise estimation subroutine used to achieve a $(1 + O(\alpha))$-approximate clustering. Algorithm 1 outlines the AugmentedKMeans proposed by Ergun et al. [19]. (implementation: src/clustering/augmented_kmeans.py)

In the original AugmentedK-Means, the predictor label vector Π directly aligns with the samples in the matrix input as shown in Algorithm. 1.

Algorithm 1 Learning-augmented k-means clustering

Require: A point set X with labels given by a predictor Π with label error rate λ
Ensure: $(1 + O(\alpha))$-approximate k-means clustering of X
1: **for** $i = 1$ to k **do**
2: Let Y_i be the set of points with label i
3: Run CRDEST for each of the d coordinates of Y_i
4: Let C'_i be the coordinate-wise outputs of CRDEST
5: **end for**
6: **return** $C'_1, \ldots, C'_k$

If there were 8 samples, we should expect the size of Π to be 8. However, Spectral Co-Clustering transforms our original input $S \times F$ matrix into a $(S + F) \times L$ Z matrix, where both samples and features are represented in the same latent space. To maintain the theoretical guarantees of AugmentedK-Means when clustering in this transformed Z space, we must extend Π to cover all rows of Z. Let $\Pi_S = [\pi_1, \pi_2, ..., \pi_S]$ be the initial predictor labels for S samples. For each feature f_j, we define its label as: $\pi_{s+j} = mode(\{\pi_i | sample\ i\ contains\ feature\ j\})$.

Each feature inherits the most frequent label among the samples that contain it. E.g., if feature f_1 are in samples S_1, S_3, S_5, S_6 and the samples are labeled $[0, 0, 1, 2]$ respectively, feature f_1 would be assigned label 0 since it represents the most samples in this label. For size reference, an example of the full dimension transformation is extending $\Pi_S = [0, 3, 1, 3, 2]$ to $\Pi_{S+F} = [0, 3, 1, 3, 2, 0, 2, 2, 1]$ given the sample count of $S = 5$ and feature count of $F = 4$. It preserves information from the AugmentedK-Means predictor while guiding the centroid placement in the transformed spectral Z space.

This extension respects both the original predictor information and the structural relationship revealed by Spectral Co-Clustering. E.g., consider an extreme scenario in the Z latent space where sample points are close, feature points are close, and both the sample and feature *"clouds"* are far. In this scenario, VBGMM might create two biclusters: one with all samples and no features, and one with no samples and all features. This would not be usable for rule generation. AugmentedK-Means recognizes that features are connected to samples and tries to place its centroids between the two *"clouds"* to bridge the gap rather than place its centroids inside each sample/feature cloud. No prior work addresses this, which will be hereafter called **mode extension** since we are extending the size of Π by filling in the feature labels with the label mode.

Solving the Calibration Problem. AugmentedK-Means requires an α hyperparameter as the error ceiling of the predictor. We followed the methodology of Ergun et al. [19] to get α. (implementation: src/clustering/crdest.py)

To do so, we "*iterate over a small range of possible α from .01 to .15 (Algorithm 2) with a step size of 0.01 and select the clustering that results in the lowest objective cost* [19]. We minimized the sum of squared distances as the cost function by using the Coordinate-wise Estimation (CRDEST) to perform this estimation efficiently; a breakdown of its key properties and algorithmic steps is provided in Appendix A.2.

Algorithm 2 Coordinate-wise estimation (CRDEST)

Require: Points $x_1, \ldots, x_{2m} \in \mathbb{R}$, corruption level $\lambda \leq \alpha$

for each trial **do**
  Randomly partition the points into two groups X_1, X_2 of size m
  Let $I = [a, b]$ be the shortest interval containing $m(1 - 5\alpha)$ points of X_1
  $Z \leftarrow X_2 \cap I$
  $z \leftarrow \frac{1}{|Z|} \sum_{x \in Z} x$
  return z
end for

3.3. Heuristics To Estimate k

Our working hypothesis is that VT reports provide an informative signal that positions each sample within our search space better than chance, because vendor family labels carry intrinsic information. We hypothesize that knowledge transfers directly from the AV label domain to the similarity hashing domain. Specifically, diversity knowledge should transfer: if an AV family is **(i)** more or less homogeneous, **(ii)** larger or more compact, or (**iii)** contains more or fewer sub-variants in the AV label domain, we force this same structural diversity into the hash domain to bias k to match the real-world scenario. We can leverage VT to estimate the number of clusters k. However, a key challenge is that VT vendors often disagree and produce noisy labels; hence, VT is not a perfect oracle. A perfect oracle would exist only if all VT engines agreed on a family label for every sample, which we do not observe. Empirically, clusters rarely achieve unanimity. Within a cluster of N samples, we typically observe multiple distinct candidate family labels across vendors rather than a single consensus (and the number of unique candidates generally increases with N). This observed label diversity both informs our estimation of k and gives us a larger range of candidates to estimate an appropriate value. Accordingly, we normalize VT reports using AVClass2 [11] and build a per-family history (an oracle) of k values observed in prior runs. For each new batch, we extract VT-derived family labels and query this oracle to select k. We assume family-level information propagates consistently across samples, so that diversity, heterogeneity, and spread-not label identity-are what AV vendors implicitly preserve, even under vendor disagreement. Concretely, when we query the oracle for a sample, it returns the family's historical k values, how samples from that family have clustered in the past. It is this structural signal, rather than label identity, that we leverage to estimate k. We tested three approaches:
1) Max k: For a given family, choose the largest historically observed k. If that value exceeds the number of available samples, cap it at the largest feasible k. We hypothesize that the original algorithm systematically underestimates the optimal k value, leading to over-generalized rules that fail to capture true diversity within malware families. By using the maximum historically observed k, we ensure that the clustering process accounts for the full complexity and variability that have been documented for each family, preventing the generation of overly broad rules that may produce FPs.
2) Statistical Mode k: For a given family, we choose the most frequently observed k across prior runs, assuming this reflects a stable choice. This approach is based on the hypothesis that the most frequently occurring k value represents the consensus optimal clustering structure for a malware family, filtering out both underestimated and overestimated k values that may result from data anomalies or algorithmic instability. The statistical mode captures the "typical" complexity level of the family by averaging out temporary fluctuations caused by unusual batches, incomplete data, or algorithmic edge cases.
3) Random k: For a given family, sample k uniformly at random from the historical values observed for that family, serving as a variance-injecting baseline.

The VT-based heuristics serve as a practical optimization, selecting k using family histories during operation. In turn, our raw-k baselines (Sec. 5.3) are used solely for evaluation and are not employed to drive k selection.

3.4. Implementation

Our implementation was designed to achieve **efficiency** and **usability**. To make AutoPYara efficient, we kept the original JAVA engine as the base, patching it with our proposed algorithm modifications. To increase usability, we provided a Python API for accessing the JAVA methods, allowing AutoPYara integration in multiple pipelines. This required breaking the original AutoYara architecture from a monolithic solution into a JAVA backend and Python frontend (details in Figure 13 of Appendix A.3). The Java-to-Python communication is performed via JPype [25], and the generated rules are checked for validity via their compilation with Yara-Python libraries [26], [27]. AutoPYara (Sample Rule in Appendix B.1) is released as a library on PyPI.

4. Experimental Setting

4.1. Dataset

We run our experiments on (malware and goodware) Windows (PE) files collected by an AV company from infected customers over 8 years, representing a realistic sample of in-the-wild malware. This dataset has been characterized in a previous study [28] and includes samples collected by the

same company and the same engine in 3 different countries: the USA, Brazil, and Japan. Using the same vendor in the same timeframe makes the samples comparable. Collection in different countries adds diversity and increases external generalization, as malware originating from different countries often displays distinct characteristics [29]. We used the malware for rule generation and the goodware for Bloom Filter training, ensuring no FP arises from biases like data collected in distinct places and times. In total, we leverage 71k malware samples and 22k goodware samples for the experimental results. Unlike studies that focus on specific attack strategies [30], [31], we **do not remove repeated samples** from our experiments, as they may appear across different malware campaigns and thus better reflect real-world conditions. We also do not perform any additional pre-processing on the samples to evaluate them as seen by the AV companies in the wild.

4.2. Experimental Methodology

Our experiments follow a methodical strategy where malware samples are clustered into families, which serve as input for both AutoYara (SOTA) and AutoPYara (our solution) in the comparative analysis. The same set of goodware files is provided to eliminate any goodware bias. We then analyze the generated rules and, to minimize variability from inherent stochasticity, report each setting as the mean over multiple runs. We applied a convergence algorithm to determine the number of runs required to precisely represent each cluster. The framework is set to generate $N = 10$ candidate rules per cluster, but adaptively determines early-stop conditions. After the generation of each rule, it evaluates its performance considering the Accuracy. If the improvement between consecutive evaluations falls below the threshold θ, the algorithm halts rule generation early. The choice of a 5% threshold reflects a balance between sensitivity and stability and is meant to prevent termination due to small fluctuations, ensuring that additional rules are generated if they provide meaningful improvement.

Experimental Setup. We ran our experiments on a high-performance server (Intel Xeon Gold 5318Y, 256GB RAM, running Ubuntu 22.04.3 LTS). We were also able to run AutoPYara on a consumer-grade laptop (Intel i7-1255U, 16GB RAM), showing that it is practical on end-user machines. AutoPYara is hardware-agnostic, requiring minimal resources (1 core, 1GB RAM) and supporting Python 3.9–3.12, Java 11+, and Linux/Windows.

4.3. The Clustering Task.

Unlike a typical ML clustering task, which assigns unlabeled samples to clusters, generating YARA rules for malware families is different. Its goal is to generate rules for already-labeled samples. This happens when an AV company has already identified a malware family and wants to generate YARA rules for it. Figure 2 exemplifies this process, where a set of labeled malware and a reference set of goodware samples (Path 1) are input to AutoYara/AutoPYara, and a set of rules (Path 3) is generated as output.

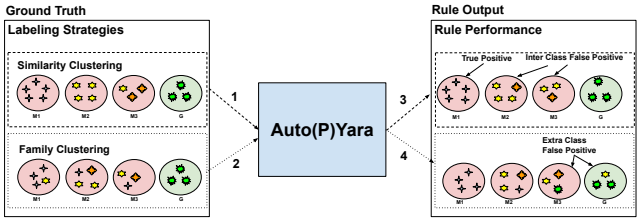


Figure 2. Overview of the rule generation task.

This process is naturally subject to variations in the input clusters. If a different cluster distribution is input (Path 2) to the rule generator, a different output (Path 4) is observed.

While accuracy is naturally a relevant metric for this task, a full understanding of the tool's performance can only be achieved by breaking it down into its components, as they present very different characteristics. On the one hand, the tool aims to maximize the True Positive Rate (TPR) for each cluster, which represents the rule's coverage, i.e., the portion of samples in a cluster that the rule can actually match. On the other hand, its goal is to minimize the False Positive Rate (FPR). It is key to note that not all mistakes (FPs) are the same. FPs can occur in two ways: First, as Extra-class False Positive Rate, where a rule misclassifies goodware as malware; and also as Inter-class False Positive Rate, where a rule incorrectly classifies a sample from a different malware family. The operational consequences of these two types are fundamentally different: intra-cluster FPs obscure malware boundaries, while extra-cluster FPs directly jeopardize goodware and normal operations. This work measures YARA rules using these three metrics. We consider it equally important to ensure that benign software continues to run without interruption and to achieve a high TP rate. An effective YARA rule must achieve a high TP rate while keeping both FP rate extremely low, ideally close to zero.

4.4. Threat Model

We evaluated AutoPYara across realistic environments with two distinct threat models based on operational priorities:

1. Detection (Conservative): We aim to prevent operational disruption and alert fatigue in production pipelines. FPs matter most here, so we require strict FP minimization even if it reduces coverage.

2. Hunting (Aggressive): We aim to discover unseen malware, prioritizing maximum TP rate while accepting higher FP tolerance for analyst triage.

These requirements are often conflicting: strict FP suppression and maximum TP coverage typically trade off against each other. We measure this conflict directly, then show how AutoPYara reconciles the two, addressing conservative detection and aggressive hunting together within a single framework.

5. Evaluation

We evaluated AutoYara and AutoPYara in two conditions. First (Subsec 5.1 to 5.3), we assessed their maximum performance potential by covering the entire dataset. Second

(Subsec 5.5), we evaluate their generalization via train-test splits. The first condition is sub-divided into: **(i)** the effects that the original AutoYara solution was subject to (Subsec 5.1), acting as a guideline for evaluating automatic rule generation procedures, and **(ii)** the individual contribution of each AutoPYara component (Subsec 5.2) to mitigate these effects, working as an ablation study for our tool.

5.1. Characterizing AutoYara & Its Effects

To evaluate whether AutoPYara is superior to the SOTA AutoYara, we have to establish clear Criteria (Cs) for comparison. Although the original paper describes the foundational approaches, it does not analyze how key parameters affect the tool's performance. Identifying these factors allows us to establish fair benchmarks for comparison, ensuring we can quantify improvements and characterize their performance in the real world malware samples.

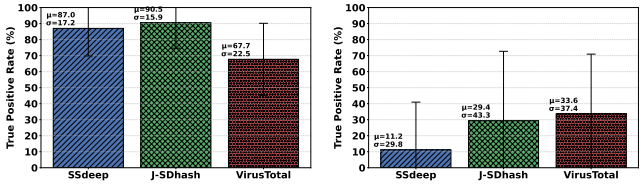
C1. The Impact of Labeling Strategy: A natural first step to evaluate the tool's performance is its TP rate. Since our dataset consists of in-the-wild samples, it lacks family labels for comparison, requiring the derivation of our own labeling strategy for the experiments [32]. But *What is the best labeling strategy?* In the absence of standards, we adopt two different approaches to evaluate their impact on the framework: **(1) similarity-based** clustering via the hashing functions SSdeep and J-SDhash; and **(2) label-based** clustering, built on VirusTotal-reported family labels (normalized by AVClass2 [11]). These clustering methods are used as they are the most popular and widely adopted strategies to group samples in the malware analysis literature [9], [32], [33], [34]. They are fundamentally different, with the first focusing on the binary characteristics of the samples and the second on their high-level behavior [35]. It is plausible to hypothesize that generated labels might vary [10], impacting the rule generation. We applied these labeling methods to all samples and generated the different clusters. We discarded singleton clusters (i.e., clusters containing only a single file) because AutoYara is designed to derive generalized rules from recurring patterns, not from singleton outliers.

TABLE 2. CLUSTER STATISTICS ACROSS METHOD

Metric	SSdeep					J-SDhash					VT
	50	60	70	80	90	50	60	70	80	90	
#Clusters	8807	8874	8874	8761	8588	7753	7921	8169	8135	8158	1121
Max Size	1129	1126	708	708	466	10649	8673	2447	1391	1262	5558
Avg Size	4.87	4.61	4.36	4.18	3.89	6	5.59	5.12	4.89	4.45	38.89

Table 2 shows that different labeling strategies yield datasets with varying sizes and internal structures. These differences alter underlying distribution shapes within the dataset, affecting the evaluation context and influencing the perceived performance. The variation across cluster counts, maximum sizes, and average distributions highlights our central hypothesis: Different labeling methods fundamentally alter how clusters are defined, affecting the task's difficulty and the conclusions one can draw from the results, leading to conflicting and unfair interpretations.

Labeling Strategy variation has operational consequences: it can fragment large clusters into many smaller ones, inflating the apparent dataset size and compounding the computational burden of detection. Every additional cluster corresponds to a new set of rules or signatures that security systems must maintain and apply. Massive clusters may oversimplify detection rules, trading off specificity for scalability and potentially introducing both FPs and missed detections. In practice, this means that the structure of the labels governs the balance between precision, scalability, and operational overhead. Evaluating tools or clustering methods on a single dataset without acknowledging these differences risks drawing conclusions that do not generalize to real-world deployment. We measure this impact via the TP rates across these variations and show how changes in dataset construction (e.g., size and structure) propagate into changes in rule quality and detection outcomes, skewing overall performance.



(a) Without Failed Clusters (b) With Failed Clusters

Figure 3. Comparison of AutoYara performance: (a) baseline excluding failed clusters; (b) observed performance including failed clusters.

Figure 3a shows the impact of different labeling approaches, which create varying clusters that yield markedly different approximations even when applied to the same data. This divergence underscores how the definition of a labeling strategy is not fixed but shaped by the algorithmic choices made during its construction. This variation arises because SSdeep and J-SDhash group samples by *intrinsic code-level similarity*, capturing functional resemblance even across unrelated labels, whereas the label-based approach depends on external categorizations that overlook such relationships. The first method generally surpasses the label-based ones in **TPs**, as it relies on more coherent and internally consistent similarity criteria than on labels that group samples according to heterogeneous standards.

C2. The Impact of Bloom Filters: We next examine the role of Bloom Filters within our framework and hypothesize that, like the labeling strategy, they may also influence its behavior. At first glance, the results shown in Figure 3a suggest that AutoYara has great performance with an average TP rate between 67% and 90.5%. This is comparable to the results presented by the original paper despite using a different dataset. However, there is a methodological oversight in this result, as it only accounts for the clusters that succeeded in producing rules. The true AutoYara performance, considering the failure rates, is shown in Figure 3b, and demonstrates a significant difference in the real-world performance of the tool. The variation arises because the provided Bloom Filters do not contain byte patterns present in malware found

in the wild. The original paper only reported results for the successful clusters, while we considered both successful and unsuccessful cases, as we had more control over the dataset. Note that different strategies cause impacts at different levels, with the hashing-based labels experiencing a significant drop in performance compared to the family-based one. Fig. 3 shows that the family clustering task is more challenging in the real world and that the shift in performance between the cases cannot be neglected.

Recovering BF performance. Outdated Bloom Filters create an initial performance gap, but we can retrain them to improve coverage. We retrain the Bloom Filters and repeat the experiment to compare how coverage and performance change throughout our evaluation.

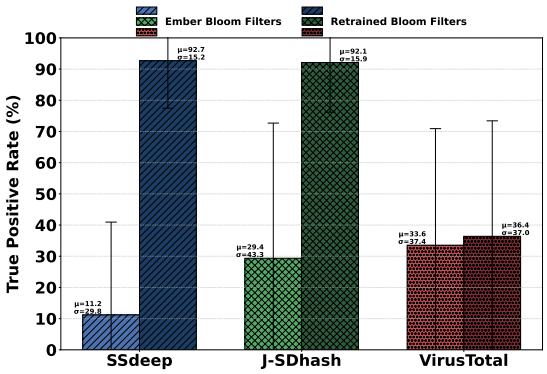


Figure 4. Impact of Bloom Filter retraining on AutoYara, showing recovery of true positive rate compared to the baseline failed cluster configuration.

Figure 4 quantifies the impact of this retraining step. We observe a clear improvement in TP rate performance. Retraining increases the number of generated rules across clusters by incorporating additional N-gram patterns, which reduces the number of failures. This improvement after retraining the Bloom Filters varies across the labeling strategies, given their fundamental differences. Byte-based clustering (SSdeep and J-SDhash) responds strongly to changes at the byte or function level, so retraining the Bloom Filters that use these hashes directly affects performance. The failures typically occur due to insufficient byte-level similarity information captured by the original Bloom Filter configuration. In contrast, VirusTotal labels aggregate samples using broader family concepts, which we find are less sensitive to fine-grained changes in the underlying code. The challenges here are fundamentally different, as VT label-based clustering struggles not from a lack of byte-level information, but from the inherent difficulty of reconciling heterogeneous label assignments. When samples carry conflicting or incompatible family labels, it becomes practically impossible to satisfy all constraints simultaneously, regardless of the underlying binary similarity. The extent of gains from retraining BFs depends on how closely the label criterion tracks true binary similarity rather than external taxonomic grouping.

C3. The Intrinsic High Variance: In the same way that Bloom Filters can impact results, other factors may have a significant effect. The VBGMM clustering step is inherently non-deterministic because it does not use a fixed random seed. Random initialization and probabilistic convergence can yield different cluster assignments even when the input data is identical, sometimes converging to local rather than global optima. This stochastic behavior introduces variability into the rule generation process, leading to rules that differ in coverage, specificity, and quality across runs. To assess the magnitude of this effect, we measured how much variability appears in both the clustering output and the corresponding rule sets. We apply an early-stop algorithm to generate and evaluate rules across multiple runs, and report the cumulative metric. This provides a dynamic view of performance over time, highlighting long-term trends and changes, and reduces the influence of outliers. In Figure 4, we highlight that the variance in the AutoYara framework is ≈30% (very high).

TABLE 3. EXTRA CLASS FALSE POSITIVE RATE (%)

Method	SSdeep	J-SDhash	VT
AY (EBF)	0.99 ± 0.89	1.19 ± 0.91	48.66 ± 33.34
AY (RBF)	1.23 ± 0.83	1.13 ± 0.95	20.36 ± 12.79
AutoPYara	1.10 ± 0.81	1.01 ± 0.91	54.28 ± 22.05
BestK	0.20 ± 0.17	0.20 ± 0.17	12.53 ± 4.95
WorstK	0.22 ± 0.16	0.22 ± 0.16	3.42 ± 2.49

AY: AutoYara, EBF/RBF: Ember/Retrained BF, VT: VirusTotal

C4. Not All Mistakes Are the Same: As per Sec. 4.3, TP gains should not come at the expense of FPs; thus, we also evaluated AutoYara via **Intra** and **Extra** class FPRs. Our experiments produced an Intra-Class FPR below 2% for the original BFs (see Table 4 of Appendix B.2). The rate is zeroed after retraining the BFs. This reinforces the importance of Bloom Filters and the careful selection of labeling strategies. Table 3 presents the Extra-Class FPR across different labeling strategies. SSdeep and J-SDhash exhibit lower Extra-Class FPR compared to the family label-based clustering. The inherent nature of similarity-based clustering is responsible for rules with an Extra-Class FPR. AV label clusters from VT are independent of internal patterns, which results in rules that are forced to be more generic, given files that have conflicting patterns. This causes rules to be more stringent with detection, resulting in high Extra-Class FPR. The VT approach is impractical where goodware is present, risking legitimate software operations. In malware-only workflows, searching within a curated repository or triaging a lab quarantine feed rather than scanning a user's hard drive, this limitation is less severe. Similarity hashing–based clustering provides robust performance, maintaining low Intra- and Extra-class FPR.

C5. The Impact of Threshold Variation: Given similarity hashing as an effective labeling strategy, we now characterize its threshold variation. This clustering method allows us to adjust a similarity threshold, which dictates how selectively the algorithm groups the samples. In practice, it controls the balance between precision and inclusiveness, shaping the quality of the derived labels.

Figure 5.a highlights a 10% difference in the threshold range from 50 to 90 in AutoYara and illustrates its impact on the results. This variation occurs because tighter thresholds produce purer clusters—in other words, the files are more

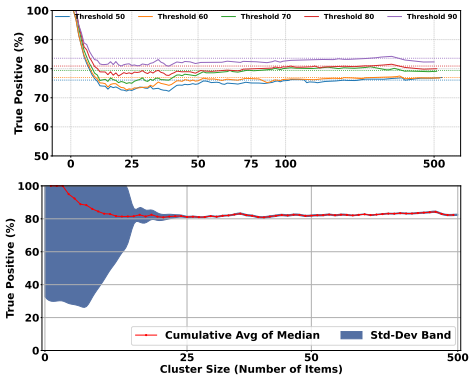


Figure 5. AutoYara across clustering configurations. (a) Difference in performance between clusters derived using different SSDeep similarity thresholds. (b) Variation in performance between small and large clusters.

closely matched at a byte level, resulting in better clusters. Figure 5.a demonstrates that all thresholds exhibit similar variations, suggesting an inherent characteristic of malware clustering, rather than an artifact of our specific approach. This fundamental property indicates that the relationship between clustering parameters and detection performance follows predictable patterns inherent to the nature of similarity analysis. Stricter thresholds generally improve rule quality and effectiveness, albeit at the cost of reduced coverage. This reduces the impact of automatic rule generation, as each unclustered sample must be processed separately by a human analyst. The balance between cluster purity and diversity becomes critical when tuning thresholds for optimal performance and representativeness. When sample sets are well characterized, prioritizing rule purity yields highly precise, specific signatures; by contrast, hunting for unknown threats benefits from greater rule diversity to broaden detection coverage.

C6. The Impact of Cluster Size: We previously demonstrated how data-aggregated analyses can obscure effects by combining results across clusters into a single overall score. Specifically, aggregation may mask variations in performance across different clusters and hide meaningful intra-cluster differences. We show that aggregation can lead to incomplete results, where some cluster configurations perform well in specific areas of the dataset, but underperform in others. Similar issues arise when aggregate performance is evaluated w.r.t. cluster size, potentially overlooking how size-specific configurations can influence results.

The distribution of cluster sizes can impact the framework's performance: a few large clusters may inflate scores, while more granular clusters can reveal weaknesses that would otherwise remain hidden [36], [37]. Larger clusters represent rules that must generalize across more samples. The hypothesis is that rules derived from small, homogeneous clusters should be highly effective within those narrow bounds but might lack generalization. Figure 5.b shows the variation in the TP rate of the rules for a given threshold using AutoYara. Small clusters exhibit higher mean TP rate at the cost of larger variance. This reinforces that finding a few stable patterns among two similar samples is inherently easier than identifying patterns across N more diverse samples. Cluster size directly influences rule effectiveness: performance often starts high for small clusters and declines as cluster size and diversity increase due to broader, less specific patterns. Our experiments show that variance is inherent to the family clustering problem, rather than to AutoYara alone, and can risk misinterpreting the results. Figure 5.b highlights this variance, underscoring the need for more fine-grained analysis. We observed these patterns across all tested thresholds for both SSDeep and JC-SDhash ground truth. Relying on a single summary metric can be misleading; Adopting a fine-grained approach to rule evaluation, we can isolate peak performance and generalization across the dataset, which is key for practical deployment.

5.2. Impact of Centroid Selection: AutoPYara [F1]

In the previous section, we identified key factors and metrics that influence rule generation, and now we will characterize and evaluate the AutoPYara framework. We compare its behavior directly with the original AutoYara framework under the same considerations. To assess AutoPYara's strengths, we follow the same strategy as in the earlier analysis and break down the results by component. We first evaluate centroid selection methods while keeping other parameters constant, then examine the impact of k values, thereby isolating variables to understand how each component shapes the framework's performance. To conduct a fair comparison, we extract the k value from the VBGMM step of the AutoYara framework and apply this same k, with only the centroid selection differing between them. We isolate variables between the two frameworks, avoiding comparisons that fail to control key parameters.

C1. The Impact of Labeling Strategies: We recreate the previous experiment across different labeling strategies. We use AutoPYara with centroid selection, to verify the impact identified on the original AutoYara framework.

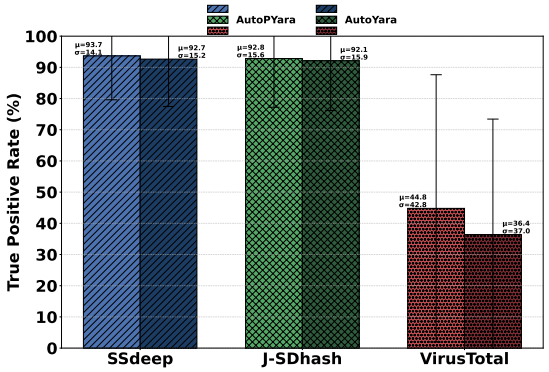


Figure 6. Comparison of labeling strategies between AutoYara and AutoPYara, highlighting performance differences under different configurations.

Figure 6 shows the performance differences across the three labeling strategies. AutoPYara performs, on average, 3.4% better than AutoYara on the same datasets and pre-clusters. Although 3.4% may appear modest at first glance, this gain translates to an average of 2400 additional detections

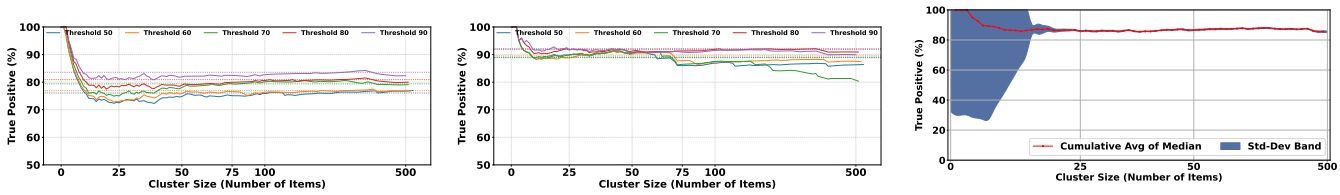


Figure 7. Comparison of SSdeep-based labeling and cluster size variations: (a) threshold variation analysis of **AutoYara**, (b) threshold variation analysis of **AutoPYara**, and (c) cluster size impact on AutoPYara using SSdeep (TH90).

across malware samples in the dataset, which represents a significant absolute improvement that would be highly valuable in the real-world operations of an AV company. The improvement is most pronounced on the VT clusters, which exhibit far greater heterogeneity compared to baselines derived from similarity hashing–based clustering algorithms. In this setting, precise centroid selection in AutoPYara is critical; picking centroids that capture the dominant, stable subpatterns yields clusters with higher intra-cluster coherence and clearer inter-cluster separation. The rule generator can then assemble subpatterns correctly—promoting shared anchors within a cluster and separating variant features across clusters—so conditions are formed in a more disjoint manner rather than overlapping. This reduces rule interference and confusion, enabling rules that generalize to unseen samples from the same family while avoiding spurious matches. Note that, given its nature, the average size of the cluster generated using the AVClass labels is 8 times greater than that of the similarity-based clustering methods. AutoPYara's centroid selection produces larger gains precisely where heterogeneity is highest.

C2. The Impact of Bloom Filters: We analyzed how retraining Bloom Filters affects AutoPYara when using centroid selection. Like AutoYara, we consider all clusters (including those that fail) when we evaluate the performance of centroid selection in AutoPYara. These failed clusters cause a noticeable drop in performance, highlighting the impact of Bloom Filters within the framework.

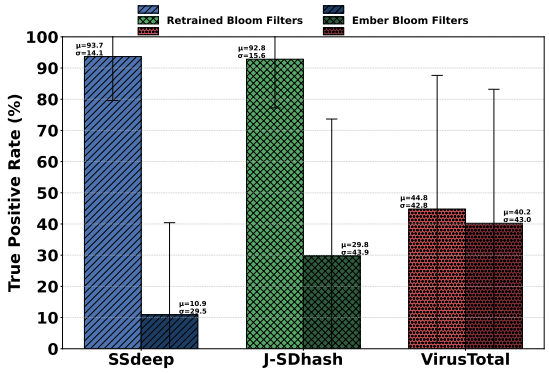


Figure 8. Impact of Bloom Filter retraining on AutoPYara, showing recovery of TPR compared to the baseline failed cluster configuration.

Figure 8 highlights AutoPYara's performance based on the Bloom Filters. Consistent with our previous findings, retraining Bloom Filters on new samples results in a clear performance improvement. Considering SSdeep clusters, AutoPYara with Retrained Bloom Filters achieves an absolute gain of 82.8%, compared to 81.5% for AutoYara. Using J-SDhash as the labeling strategy, the gain is 63.0% compared to 62.7% with AutoYara. For the VT clusters, AutoPYara yields a 4.6% gain compared to 2.8% with AutoYara. Centroid selection mitigates heterogeneity but lacks sufficient statistical power to address all clustering challenges, particularly when compensating for missing data (e.g., absent n-grams) that can significantly impact cluster representation. This limitation positions Bloom Filter retraining as an essential component of the pipeline. The increased sensitivity of retrained filters to centroid selection (AutoPYara) stems from the fact that retraining typically produces more clusters, introducing granularity which magnifies the influence of centroid selection on overall detection.

C3. The Intrinsic High Variance: AutoPYara inherits the lack of determinism of AutoYara. As a result, the framework exhibits performance variations across different runs, consistent with the behavior of the original framework. Figure 6 illustrates this considerable variance through the error bars. If both frameworks display high non-determinism, it indicates that this behavior stems from the underlying problem of family clustering rather than the specific framework. Our results demonstrate notable consistency improvements for SSdeep clusters, where AutoPYara reduces standard deviation from 15.2% to 14.1% and 15.9% to 15.6% for J-SDhash clusters. Conversely, VT clusters exhibit increased variability, with standard deviation rising from 37% to 42.8%. This trend persists and reflects the heterogeneous nature of the VT clusters combined with AutoPYara's centroid selection.

C4. Not All Mistakes Are the Same: We evaluated AutoPYara's FPRs as before. For SSdeep and J-SDhash, AutoPYara achieves an intra-class FPR of 0% (Table 4 in Appendix B.2) and an Extra-class FPR below 2%. These results highlight its potential for real-world deployment, since practical malware detectors must prioritize keeping FPR's low. For VT clusters, it has a large Extra-class FPR of 54.28% compared to 20.36% for AutoYara, as shown in Table 3. Note that this FPR increase does not occur for SSdeep and J-SDhash clusters because AutoPYara's centroid-based approach makes selection more precise for homogeneous clusters, which is not the case for VT labels.

While centroids enable AutoPYara to generalize across heterogeneous VT clusters, the resulting rules also capture broader patterns that were previously unnoticed. These generalized patterns increase coverage but inadvertently also match goodware. Thus, addressing heterogeneity solely at the centroid level is insufficient. The extra detections are

beneficial when only malware families are considered, but the rules are too broad for practical hunting. Given precise clusters, as in SSdeep and J-SDhash, AutoPYara outperforms AutoYara. When clusters are noisy and less precise, centroid selection amplifies that noise.

C5. The Impact of Threshold Variation: Adjusting similarity thresholds directly shapes the diversity of samples in each cluster, which affects the quality of centroids in AutoPYara. A higher threshold creates smaller, more uniform clusters, while a lower threshold results in larger, more diverse groups. If centroids remain effective amid greater heterogeneity, decreasing the threshold should help AutoYara generate broader, more general rules. Using a higher threshold may improve precision by keeping clusters purer, and we test this by varying the threshold and measuring how changes impact rule performance. AutoYara shows a threshold variation of 10 pp, as shown in Figure 7(a). AutoPYara significantly reduces this variation to 2.9 pp, as shown in Figure 7(b), a 7.1 pp reduction in variation.

5.3. Impact of k on AutoPYara

Comparing Figure 5.b to Figure 7.c, we show that AutoPYara consistently outperforms AutoYara by roughly 4% across the samples. Smaller clusters tend to be more homogeneous, making it easier for the framework to derive rules that capture clear and representative patterns. As clusters increase in size and diversity, shared characteristics become diluted, reducing rule specificity and performance. AutoPYara's centroid-based strategy offsets this by anchoring rules around representative points, enabling more robust generalization across samples.

So far, we use k derived from the VBGMM step of AutoYara as our reference point in previous experiments. This approach allowed us to validate our framework's improvements under controlled conditions, but it reflects a lab setting. We now transition to a more realistic setup where AutoPYara independently determines the k value through its own clustering analysis. In this setup, we verify whether there is room to improve performance through k selection and, if so, whether this process can be carried out automatically in a systematic approach. This methodological shift from externally-referenced to internally-determined k values allows us to evaluate system performance under genuine operational conditions, optimize the K-selection algorithm, and demonstrate the framework's robustness without external clustering guidance. This transition is crucial toward developing an independent and optimized AutoPYara. All experiments in this phase isolate the variable of interest by fixing the centroid-based clustering as the baseline while systematically shifting only the k parameter.

[F2] Evaluation of Ideal k Values: Our first test investigates whether there is room for performance improvement through k-selection. To establish an upper bound, we assume access to an oracle, which we simulate via brute-force search. The oracle provides the ideal k for each case, thereby revealing the maximum performance AutoPYara could achieve under perfect conditions.

(1) Exhaustive Search for Optimal k: We refine AutoYara's initial estimate k by exhaustively searching its neighborhood. For each candidate k, we evaluate performance and select the value that produces the best results. This brute-force search is not intended as a practical deployment strategy but rather as a methodological tool to approximate an oracle, allowing us to simulate access to the ideal k values and establish the upper-bound performance AutoYara could achieve. We provide additional information regarding the search space in Appendix B.3.

(2) Worst-Case k: We also evaluate the worst-case k as a lower-bound baseline to verify that the observed gains from AutoYara using VBGMM and AutoPYara reflect genuine improvements. To that end, we repeated the above exhaustive search to identify k with the worst rule quality.

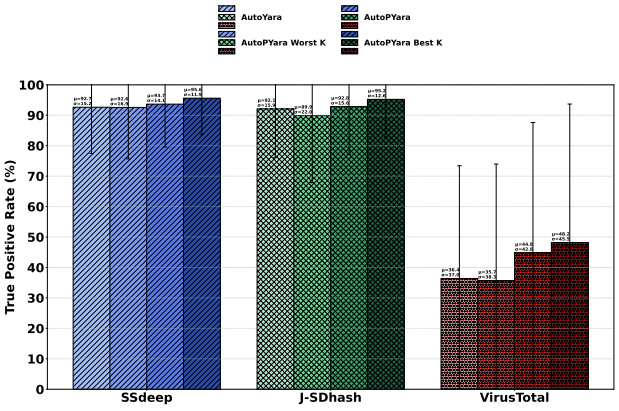


Figure 10. Highlight the impact of k on AutoPYara Using Extreme k Values across different labeling strategies

The value of k clearly affects AutoPYara's performance. Fig. 10 shows that on average, choosing an appropriate k increases the TPR by 3% for SSdeep clusters, 5.3% for J-SDhash clusters, and 12.5% for VirusTotal-derived clusters when comparing the best configuration to its suboptimal values of k. Although these gains may look marginal at first glance, remember that smaller clusters skew the averages. Similarity-based clustering tends to contain a large frequency of clusters with just 2-3 samples. These limit the range of k and thus the achievable gains. VT labeling, in turn, by not relying on intrinsic code-level similarity, produces clusters with significantly more files per family, resulting in larger cluster sizes, with bigger room for improvement. Fig. 10 shows the gap between the best and a suboptimal k configuration is $\approx 12.5\%$ in this setting. These results highlight the importance of selecting k for the framework and the difference between the k used in AutoYara and the optimal k for AutoPYara. While VBGMM performs better than the worst-case baseline as intended, yielding real gains, it remains suboptimal (w.r.t. the best case), typically underestimating k, presenting room for improvement.

[F3] Heuristic Approachs to Estimate k:

(1) Uninformed Heuristic to Estimate k: To establish a principled approach to estimate k without relying on VBGMM, we first explored three naive strategies that do

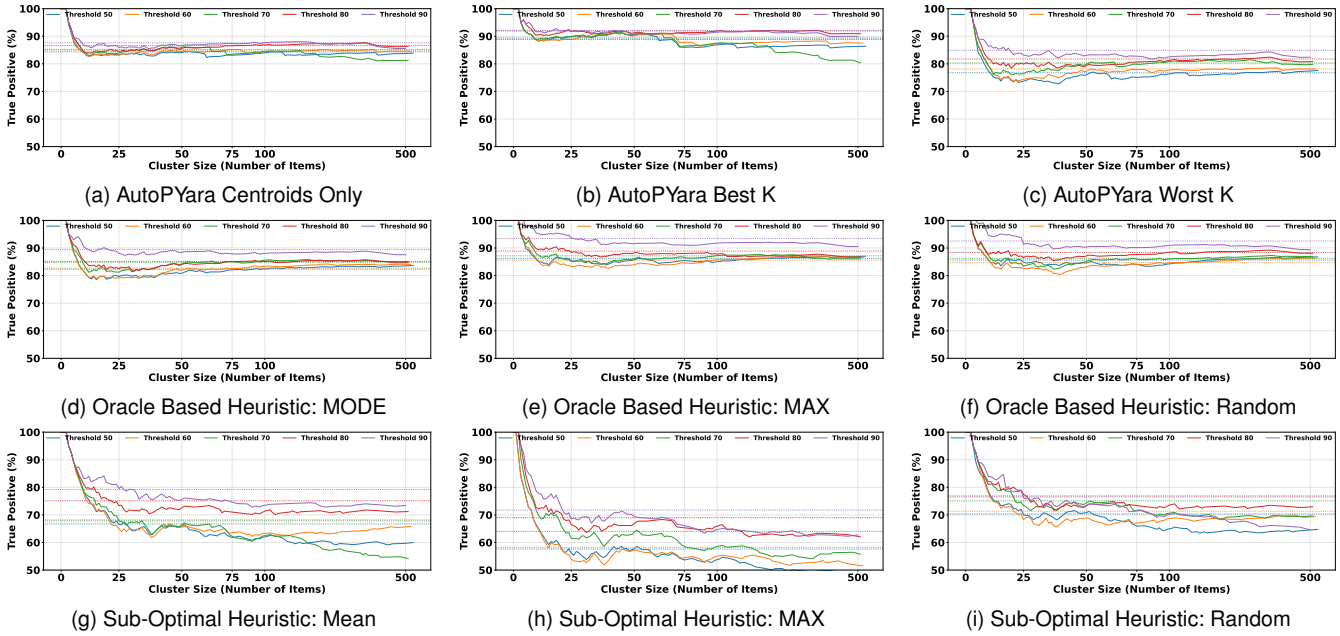


Figure 9. Comparing AutoPYara in Different Configuration demonstrating the overall trend and gains using SSdeep as labeling strategy

not incorporate any semantic understanding of sample types. These heuristics are intended as a substitute for exhaustive search. The three are not exhaustive, but they are representative of different approaches. The three methods are: **(i)** set k to the total number of files in a cluster; **(ii)** set k to the mean length to equally divide the data; and **(iii)** randomly guess an arbitrary value of k. These strategies are naive because they ignore the data's structure and yield unstable, biased clusters. Figures 9g, 9h, 9i reinforce the impact of k selection and highlight how the impact is more pronounced in larger clusters, where there are fewer chances for randomly sampling a proper k value. This confirms that a stronger approach for k definition is warranted; thus, our proposal is an informed heuristic.

(2) Informed Heuristic to Estimate k: Although we demonstrated the potential of a k oracle to improve rule generation performance, true oracles for k do not exist in reality. To approximate an oracle, we require reliable k values—without exhaustive search, better than the original VBGMM estimation, and more effective than a random guess. As per Sec. 3, we adopt a systematic heuristic that leverages historical AutoPYara runs indexed by VT family labels (via AVClass [11]) to estimate k for new samples from the same family. We do not rely on VT for cluster labeling; instead, we repurpose them as a practical proxy to bypass expensive hash-based clustering and to anchor k estimation in observed family trends. While VT's family assignments can be inconsistent and not always semantically meaningful, this usage still improves practical reliability over VBGMM alone and avoids the cost of brute force. VT now serves a different role; it enables family-level grouping to drive our heuristic for estimating k, while hashes are still used for the centroids. Our core hypothesis is that we can leverage historical family information tied to VT's labels to position our search range for optimal k values in regions with a higher probability of success compared to random initialization, thus gaining efficiency and grounding it in practical malware evolution trends. Figures 9d, 9e, and 9f show that our approach performs comparably to the Best configuration within a 2% error margin. It also performs 10% better than AutoYara on the same samples. For larger clusters, our informed choice of k achieves superior performance relative to the heuristic and random settings. Together, these results **rescue** the application of VT to family clustering, which had previously been viewed as impractical.

5.4. The Qualitative AutoPYara Impact

AutoPYara's centroid selection and better k values affect the generated rule's quality, which is a known [38] essential aspects for human analysts. We evaluated rule quality via three metrics: **1)** *Number of Strings*; **2)** *Hex Length*; and **(3)** *Conditions Count*; These metrics compose the *Compactness* metric. Given two rules with similar TP and FP rates, compactness is preferable. Figure 14 provides a high-level overview of the overall distribution of metrics for the generated rules, comparing AutoYara and AutoPYara across four different configurations. AutoPYara creates smaller, more complex rules with greater coverage than the original framework, thus being superior to AutoYara both quantitatively and qualitatively. We provide more details in Appendix B.4. Further, in Appendix B.1, we provide two rules to demonstrate how AutoPYara derives more compact signatures with superior coverage compared to AutoYara.

5.5. AutoPYara as a predictor for Threat Hunting

After characterizing the full potential of AutoYara and AutoPYara using the entire dataset, we split each cluster into

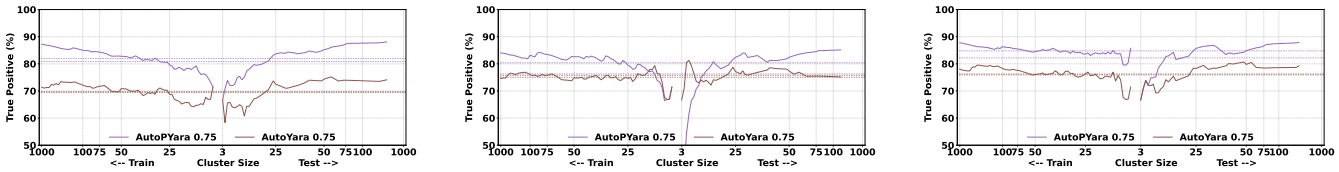


Figure 11. Threat hunting comparison between AutoYara and AutoPYara under different availability assumptions: (a) oracle-based ideal configuration, (b) no-oracle setting, and (c) real-world scenario.

training and test sets to evaluate their generalization on the threat-hunting task. This allows for an assessment by creating detection rules from limited samples and deploying them to identify unknown variants of the same malware families. The training portion consists of limited samples that analysts have access to when crafting rules, while the test portion consists of unseen samples that arrive in operational networks. The evaluation is performed at the cluster level using a split ratio of 75/25. The splits are non-overlapping to prevent data leakage, keeping train and test samples from each cluster strictly isolated.
We evaluated the task in 3 scenarios:

1. When VT labels are available The best scenario for AutoPYara operation is when VT labels are available in the oracle for **all** the targeted families, allowing better k estimation via the informed heuristic. Figure 11.a shows that AutoPYara achieves a 20% gain when leveraging the informed heuristic compared to the baseline. By utilizing the gt labels from the VT oracle, the system more accurately estimates the optimal number of clusters (k), effectively minimizing both over-clustering and family fragmentation.

2. When VT labels are not available The worst threat hunting scenario for AutoPYara operation is when VT labels are not available in the oracle for **any** of the targeted families, limiting it to operate only with the uninformed k estimation heuristic. Figure 11.b shows that AutoPYara's performance remains resilient, maintaining a 5% lead over the baseline even when operating under the uninformed heuristic. While the lack of VT labels removes the ability to fine-tune k based on known family distributions, the system manages to outperform generic clustering methods by relying on structural code similarities.

3. When VT labels are partially available In real hunting cases, VT oracle labels will be available to only part of the samples, limiting the benefit of the improved heuristic to them. We simulate this by using the real sample proportions in our dataset with VT labels, applying the informed heuristic to them, and falling back to the uninformed heuristics when needed. Figure 11.c, we highlight that AutoPYara outperforms AutoYara by an absolute gain of $\approx$ 10% in both the training and threat hunting scenarios.

We also evaluated different splits, and AutoPYara consistently outperformed AutoYara. For each split ratio, samples from each cluster are randomly selected for rule generation, and the remaining are used to evaluate detection of the generated rules. This experiment measures how effectively rules derived from varying amounts of training data detect and classify previously unseen test samples.

6. Benchmarking Issues & Contributions

In this paper, we characterize the AutoYara framework in a real-world setting and highlight pitfalls in its evaluation methodology and how these can skew results, inflating the reported performance. Figure 12 illustrates the overall reconstruction from AutoYara through its theoretical maximum. In practice, we show that our oracle-based approach comes within 2% of the best outcomes, whereas AutoYara, if not properly characterized, can perform up to 76% worse than optimal rules in reality. We characterize the performance gains on large clusters in Figure 15 in Appendix. B.5, highlighting the gains in larger (stable) clusters.

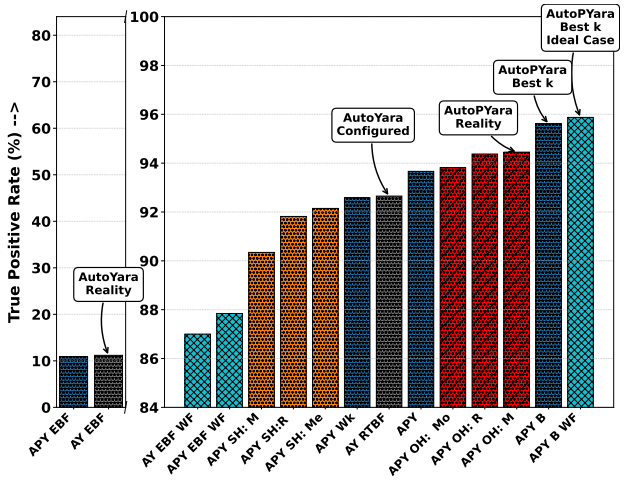


Figure 12. AutoYara and AutoPYara across different configurations

7. Conclusion

This work introduces AutoPYara, an evolved framework for automatically generating YARA rules for malware family clusters. It advances the SOTA via **(1)** precise cluster centroid selection based on similarity hashing and **(2)** better selection of the number of subclusters representing a malware family based on historical VirusTotal data. Our evaluation identified the limits of the automatic rule generation process and demonstrates that AutoPYara's optimal configuration, achieving an 80% improvement, surpasses existing benchmarks to define the current state-of-the-art. Finally, to facilitate the practical application of our findings, we outline guidelines (Appendix C) for achieving this optimal performance.

Ethical Considerations. The dataset utilized consists of malware that have been vetted for academic research. Because the dataset contains no active threats, no personally identifiable information, and no sensitive user telemetry, this research poses no risk of harm to users or third parties.

Artifact Availability Statement. To support open science and full evaluation replication for ACSAC, we provide open-access links to all components:

- **ACSAC Permanent Artifact:** Zenodo [https://zenodo.org/records/23107801]
- **ACSAC Artifact Source:** GitHub [https://github.com/Botacin-s-Lab/AutoPYara/]
- **API Documentation:** GitHub Pages [https://botacin-s-lab.github.io/AutoPYaraPyPI/]
- **PyPI Source Repository:** GitHub [https://github.com/Botacin-s-Lab/AutoPYaraPyPI]
- **Java Backend Engine:** GitHub [https://github.com/Botacin-s-Lab/AutoPYaraBackend]
- **YARA Rules & Benchmark Data:** Zenodo [https://zenodo.org/records/22665898]
- **Python Package (PyPI):** PyPI [https://pypi.org/project/autopyara/] (`pip install autopyara`)

Acknowledgments. We thank the reviewers, shepherd and the artifact evaluators for their helpful insights. We also thank NSF for supporting this work via grant CNS-2327427.

LLM Usage Statement

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality. LLMs also assisted in generating the AutoPYara code, which the authors thoroughly tested, validated, and take full responsibility for.

References

[1] B. Miller, A. Kantchelian, M. C. Tschantz, S. Afroz, R. Bachwani, R. Faizullabhoy, L. Huang, V. Shankar, T. Wu, G. Yiu, A. D. Joseph, and J. D. Tygar, "Reviewer integration and performance measurement for malware detection," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, ser. DIMVA 2016. Berlin, Heidelberg: Springer-Verlag, 2016, p. 122–141. [Online]. Available: https://doi.org/10.1007/978-3-319-40667-1_7

[2] C. Nilă and V. Patriciu, "Taking advantage of unsupervised learning in incident response," in *2020 12th International Conference on Electronics, Computers and Artificial Intelligence (ECAI)*, 2020, pp. 1–6.

[3] C. Nilă, I. Apostol, and V. Patriciu, "Machine learning approach to quick incident response," in *2020 13th International Conference on Communications (COMM)*, 2020, pp. 291–296.

[4] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature generation and detection of malware families," in *Information Security and Privacy*, Y. Mu, W. Susilo, and J. Seberry, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 336–349.

[5] B. A. AlAhmadi and I. Martinovic, "Malclassifier: Malware family classification using network flow sequence behaviour," in *2018 APWG Symposium on Electronic Crime Research (eCrime)*, 2018, pp. 1–13.

[6] E. Raff, R. Zak, G. L. Munoz, W. Fleming, H. S. Anderson, B. Filar, C. Nicholas, and J. Holt, "Automatic Yara Rule Generation Using Biclustering," in *13th ACM Workshop on Artificial Intelligence and Security (AISec'20)*, 2020. [Online]. Available: http://arxiv.org/abs/2009.03779

[7] V. M. Alvarez, "Yara: The pattern matching swiss knife for malware researchers (and everyone else)," 2013. [Online]. Available: https://github.com/virustotal/yara

[8] I. S. Dhillon, "Co-clustering documents and words using bipartite spectral graph partitioning," in *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '01. New York, NY, USA: Association for Computing Machinery, 2001, p. 269–274. [Online]. Available: https://doi.org/10.1145/502512.502550

[9] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital investigation*, vol. 3, pp. 91–97, 2006.

[10] M. Botacin, V. H. Galhardo Moia, F. Ceschin, M. A. Amaral Henriques, and A. Grégio, "Understanding uses and misuses of similarity hashing functions for malware detection and family clustering in actual scenarios," *Forensic Science International: Digital Investigation*, vol. 38, p. 301220, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666281721001281

[11] S. Sebastián and J. Caballero, "Avclass2: Massive malware tag extraction from AV labels," *CoRR*, vol. abs/2006.10615, 2020. [Online]. Available: https://arxiv.org/abs/2006.10615

[12] C. M. Bishop, *Pattern recognition and machine learning*. Springer, 2006.

[13] E. Raff, W. Fleming, R. Zak, H. Anderson, B. Finlayson, C. K. Nicholas, and M. Mclean, "KiloGrams: Very Large N-Grams for Malware Classification," in *Proceedings of KDD 2019 Workshop on Learning and Mining for Cybersecurity (LEMINCS'19)*, 2019. [Online]. Available: https://arxiv.org/abs/1908.00200

[14] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, p. 422–426, Jul. 1970. [Online]. Available: https://doi.org/10.1145/362686.362692

[15] A. Corduneanu and C. Bishop, "Variational bayesian model selection for mixture distributions," in *Proceedings Eighth International Conference on Artificial Intelligence and Statistics*. Morgan Kaufmann, January 2001, pp. 27–34. [Online]. Available: https://www.microsoft.com/en-us/research/publication/variational-bayesian-model-selection-for-mixture-distributions/

[16] H. S. Anderson and P. Roth, "Ember: An open dataset for training static pe malware machine learning models," 2018. [Online]. Available: https://arxiv.org/abs/1804.04637

[17] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data clustering: a review," *ACM Comput. Surv.*, vol. 31, no. 3, p. 264–323, Sep. 1999. [Online]. Available: https://doi.org/10.1145/331499.331504

[18] S. Lloyd, "Least squares quantization in pcm," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[19] J. C. Ergun, Z. Feng, S. Silwal, D. P. Woodruff, and S. Zhou, "Learning-augmented k-means clustering," 2022. [Online]. Available: https://arxiv.org/abs/2110.14094

[20] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD'96. AAAI Press, 1996, p. 226–231.

[21] J. C. Bezdek, R. Ehrlich, and W. Full, "Fcm: The fuzzy c-means clustering algorithm," *Computers & Geosciences*, vol. 10, no. 2, pp. 191–203, 1984. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0098300484900207

[22] D. A. Reynolds *et al.*, "Gaussian mixture models." *Encyclopedia of biometrics*, vol. 741, no. 659-663, p. 3, 2009.

[23] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, pp. 91–97, 2006, the Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287606000764

[24] Google LLC, "Virustotal," https://www.virustotal.com, 2025, accessed: 2025-05-29.

[25] S. Menard, L. Nell *et al.*, "Jpype: A python module to provide full access to java from within python," https://github.com/jpype-project/jpype, 2024, apache-2.0 License. [Online]. Available: https://github.com/jpype-project/jpype

[26] Avast, "yaramod: Parsing of yara rules into ast and building new rulesets in c++," https://github.com/avast/yaramod, 2025, mIT License. [Online]. Available: https://github.com/avast/yaramod

[27] VirusTotal, "yara-python: The python interface for yara," https://github.com/VirusTotal/yara-python, 2024, apache-2.0 License. [Online]. Available: https://github.com/VirusTotal/yara-python

[28] M. Botacin, H. Aghakhani, S. Ortolani, C. Kruegel, G. Vigna, D. Oliveira, P. L. D. Geus, and A. Grégio, "One size does not fit all: A longitudinal analysis of brazilian financial malware," *ACM Trans. Priv. Secur.*, vol. 24, no. 2, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3429741

[29] M. Botacin and H. Gomes, "Cross-regional malware detection via model distilling and federated learning," in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 97–113. [Online]. Available: https://doi.org/10.1145/3678890.3678893

[30] G. Di Pietro, D. C. D'Elia, and L. Querzoni, "Can you run my code? a close look at process injection in windows malware," in *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1600–1616. [Online]. Available: https://doi.org/10.1145/3708821.3736206

[31] A. Kosikowski, D. Cho, M. Ninan, A. Ralescu, and B. Wang, "Evilelf: Evasion attacks on deep-learning malware detection over elf files," in *2023 International Conference on Machine Learning and Applications (ICMLA)*, 2023, pp. 1702–1709.

[32] R. J. Joyce, E. Raff, and C. Nicholas, "A framework for cluster and classifier evaluation in the absence of reference labels," in *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, ser. AISec '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 73–84. [Online]. Available: https://doi.org/10.1145/3474369.3486867

[33] S. Lee, W. Jung, S. Kim, J. Lee, and J.-S. Kim, "Dexofuzzy: Android malware similarity clustering method using opcode sequence," *Virus Bull*, vol. 12, 2019.

[34] R. J. Joyce, D. Amlani, C. Nicholas, and E. Raff, "Motif: A malware reference dataset with ground truth family labels," *Comput. Secur.*, vol. 124, no. C, Jan. 2023. [Online]. Available: https://doi.org/10.1016/j.cose.2022.102921

[35] A. Vitale, S. Aonzo, S. Dambra, N. Rani, L. Ippolito, P. Kotzias, J. Caballero, and D. Balzarotti, "The polymorphism maze: Understanding diversities and similarities in malware families," in *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*. Springer, 2025.

[36] P. Li, L. Liu, D. Gao, and M. K. Reiter, "On challenges in evaluating malware clustering," in *Recent Advances in Intrusion Detection*, S. Jha, R. Sommer, and C. Kreibich, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 238–255.

[37] R. Cordeiro de Amorim and C. D. Lopez Ruiz, "Identifying meaningful clusters in malware data," *Expert Systems with Applications*, vol. 177, p. 114971, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417421004127

[38] M. Botacin, "What do malware analysts want from academia? a survey on the state-of-the-practice to guide research developments," in *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 77–96. [Online]. Available: https://doi.org/10.1145/3678890.3678892

Appendix A.
System Architecture and Algorithms

This appendix provides a deep dive into the system's architectural components and mathematical logic.

A.1. Biclustering Algorithms

Section 2.1 discusses AutoYara and its use of Biclustering, in this appendix we expand its definition. Biclustering is the process in which rows and columns are clustered simultaneously in a matrix. A single bicluster consists of a subset of rows that correlate with a subset of columns or vice versa. In the context of YARA rule generation, we have a list of files S and n-gram candidates (signatures) F within a discrete binary matrix $A_{S\times F}$, where $A_{i,j} = 0$ indicates that sample i does not have n-gram j and $A_{i,j} = 1$ indicates that it does. Biclustering this matrix produces clusters that tell us which n-grams are strongly associated with which types of files. This allows us to process the bicluster and construct it into a conditional query for matching: *if a new sample X has at least 3 of the signatures F1, F3, F10, F58, then it belongs in this bicluster and therefore this malware family.*

AutoYara performs Biclustering via the SpectralCoClustering algorithm [8]. By computing the normalized graph Laplacian and eigenvectors of the similarity matrix (in our case, matrix $A_{S\times F}$), it can map both features and samples into a shared latent space using the eigenvectors computed as the points in space (denoted by the matrix Z). This reduces the dimensionality of a dataset, preserves its distance similarity, and treats both samples and features as data points before running a clustering algorithm on it, allowing simultaneous clustering of features and samples. While it has costs in memory and computation time due to the need for the graph Laplacian and eigenvectors, it can be done in parallel. SpectralCoClustering will transform matrix $A_{S\times F}$ into matrix $Z_{S+F,l}$ where $l = log_2(k)$ and k is the number of expected clusters. This transformation uses the subset of eigenvectors corresponding to indices 2 through $l+1$ of the graph Laplacian's eigen decomposition. $log_2(k)$ is chosen as the dimension for Z since each additional k cluster can map to a quadrant of space within. Essentially, each new dimension in l doubles the number of possible cluster assignments. After transformation, any traditional clustering algorithm can be adapted to Z.

A.2. Augmented K-Means

Estimating the hyperparameter α, which is the upper bound on the predictor label error rate is a complex task. To address this limitation, we use CRDEST to estimate α efficiently. This design rests on four key properties. First, while α is a required hyperparameter bound for the label error rate that is difficult to derive analytically, CRDEST enables its empirical estimation. Second, the algorithm's primary contribution lies in robust initialization and centroid estimation. Third, k estimation and point assignments continue to rely on K-Means as the underlying support engine. Finally, because

Algorithm 2 operates in $O(n \log n)$ time due to sorting, its high speed allows for the small-scale brute-force search described in Section 5.

A.3. Framework Representation

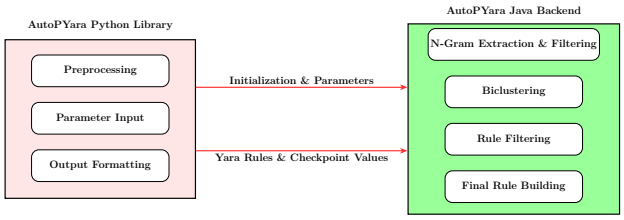


Figure 13. The AutoPYara interface and integration pipeline.

This appendix provides a technical breakdown of the design discussed in Section 2. Figure 13 illustrates the specific Python interface logic, detailing how external modules interact with the core engine.

Appendix B. Empirical Evaluation and Characterization

In Section 5, we demonstrate how different configurations can yield significantly varied results. This necessitates a critical evaluation of these tools at their extremes across different labeling strategies and Bloom filter configurations. In this appendix, we decompose the evaluation results into their core components to visualize how each framework performs against varying constraints.

B.1. AutoPYara V/S AutoYara In Practice

In Section 5.4, we describe how rule complexity plays a vital role in both the interpretability and the practical utility of automatically generated YARA rules. In this appendix, we compare the structural differences between signatures generated by our proposed framework and the baseline AutoYara system. Rules 1 and 2 rules generated for the same malware cluster (Cluster 4).

```
rule Cluster4
{   // Input TP Rate:
    // 609/613
    strings:
    // Benign FP est: -0.0 Malicious FP
    est: -0.0 Entropy: 3.5, 612 files
    $x0 = { 00 00 C7 45 C4 AC 4E 48 00 E8
    95 F8 FF FF 8B 4D }
    // Benign FP est: -0.0 Malicious FP
    est: -0.0 Entropy: 4.0, 612 files
    $x1 = { 51 E8 D4 69 FB FF 83 C4 04 5F
    5B 33 C0 5E 8B E5 }
    // Benign FP est: -0.0 Malicious FP
    est: -0.0 Entropy: 3.75, 612 files
    $x2 = { 15 E4 26 48 00 5F 8B C6 5E 5D
    C2 2C 00 55 8B EC }
    // Benign FP est: -0.0 Malicious FP
    est: -0.0 Entropy: 4.0, 613 files
    $x3 = { 6A 01 8B FA E8 E9 0F FE FF 83
    C4 08 85 C0 74 24 }
    // Benign FP est: -0.0 Malicious FP
    est: -0.0 Entropy: 2.5, 612 files
    $x4 = { 18 40 00 A7 18 40 00 1A 18 40
    00 62 18 40 00 D9 }
    condition:
    5 of ($x0,$x1,$x2,$x3,$x4)}
```

Listing 1. **AutoPYara Rule**

```
rule Cluster4
{   // TP Rate: 515/613
    strings:
    $x0 = { 53 E8 .. E8 D3 FC 01 } //
    Entropy: 6.22, Files: 612
    $x1 = { 83 C8 .. 10 53 56 57 } //
    Entropy: 6.19, Files: 611
    $x2 = { 6A 11 .. 5C EB 10 83 } //
    Entropy: 6.03, Files: 612
        .
        .
        .
    $x14 = { FF 75 .. C4 CC CC }  //
    Entropy: 6.31, Files: 608
    $x15 = { 00 6C .. 00 20 00 }  //
    Entropy: 3.22, Files: 266
    $x16 = { 40 89 .. 83 79 08 }  //
    Entropy: 6.15, Files: 612
    condition:
    16 of ($x0,$x1,$x2,$x3,$x4,$x5,$x6,
    $x7,$x8,$x9,$x10,$x11,$x12,$x13,
    $x14,$x15,$x16)}
```

Listing 2. **AutoYara Rule**

B.2. False Positive Rates In Practice

False positive rates are an important metric crucial to the operational capabilities of an automatic YARA rule generator. In practice, YARA rules need to maintain an FP rate as close to zero as possible to make the generated rules practical to deploy, as discussed in Section 5. Here we characterize the operational precision of AutoPYara by evaluating FPRs across diverse clustering strategies.

TABLE 4. FPR RESULTS (%): EXTRA AND INTRA CLASS

Method	Ember BF	Retrained BF
Part A: Extra-Class FPR (Benign Samples)		
SSdeep		
Auto(P)Yara	0.9 ± 0.8	1.1 ± 0.8
Best/Worst K	–	≈ 0.21
J-SDhash		
Auto(P)Yara	1.1 ± 0.8	1.0 ± 0.9
Best/Worst K	–	≈ 0.22
VirusTotal		
AutoYara	48.7 ± 33.3	20.4 ± 12.8
AutoPYara	44.4 ± 19.4	54.3 ± 22.1
Best/Worst K	9.9 ± 4.2	12.5 ± 5.0
Part B: Intra-Class FPR (Cross-Cluster)		
All Methods		
AutoYara (SSDeep)	0.15 ± 0.33	0.00
All Other Configs	0.00	0.00

B.2.1. Extra-Class Detection. Table 4 presents extra-class FPR (%) across SSDeep, J-SDhash, and VirusTotal for AutoYara and AutoPYara, evaluated against three baselines: SSDeep similarity matches, J-SDhash similarity matches,

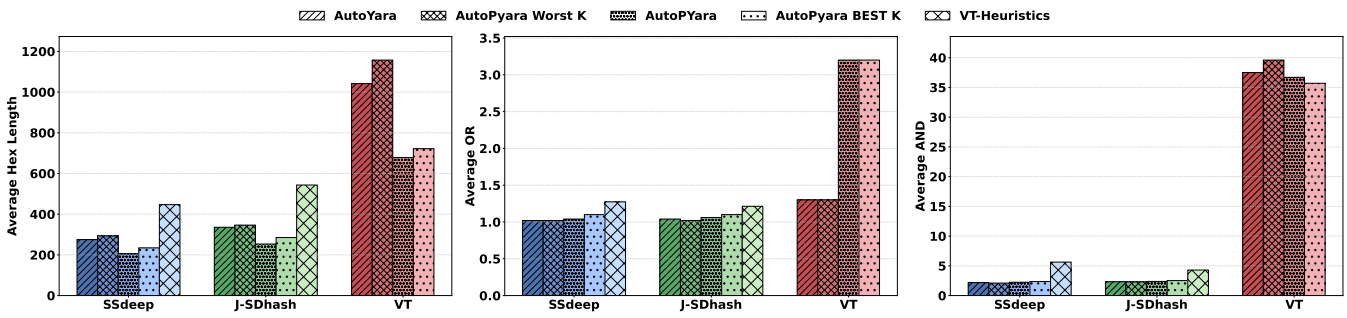


Figure 14. Qualitative Analysis. Comparison of structural rule metrics, (a) hex string length and boolean logic complexity:(b) AND, (c) OR counts

and VirusTotal family-labeling signals. Lower values indicate fewer benign files matched by rules or baseline heuristics, which is desirable for operational deployments. Our experimental results show that byte-based clustering yields a substantially lower (often near-zero) extra class FPR due to its homogeneous nature, compared with heterogeneous, AV family label–based clustering. This is critical for proactive deployment of such systems.

B.2.2. Intra-Class Detection. Table 4 reports intra-class FPR (%), which is uniformly zero in nearly all cases, indicating minimal spurious cross-cluster matches and strong cluster isolation. The only non-zero entry is a negligible 0.15 ± 0.33 for AutoYara under SSDeep with the Ember BF, consistent with measurement noise. These results confirm tight cluster boundaries and a low risk of rule bleed-over in deployment.

B.3. Note on k search

In Section 5.3, we characterize the performance of AutoPYara and demonstrate how it outperforms the original framework. However, as we speculate that AutoPYara may occasionally underestimate the optimal value of k, we utilize a brute-force search across a range of k values to identify the local minima that yield the highest-performing YARA rules. In this appendix, we provide additional information regarding the resulting search space and its impact on rule efficacy. In practice, we test the ±10 neighborhood—i.e., $k - 10$ through $k + 10$— because empirical observations show that the optimal k consistently lies near AutoYara's estimate, making an exhaustive search unnecessary. Although we could brute-force all possibilities to find the optimal k, we observed that the best value consistently lies near AutoYara's estimate.

B.4. Empirical Evaluation of Rule Complexity

To supplement the high-level qualitative findings in Section 5.4, we here provide a granular breakdown of our rule complexity metrics.

1) *Number of Strings* (N): The **number of strings** reflects the breadth of features used and is independent of the logical operators in the condition block.

2) *Hex Length* (H): AutoPYara generates byte n-grams from kilobyte-sized chunks of binaries and encodes them as hexadecimal strings. Given N hex strings $s_1, s_2, ..s_N$, we define Hex Length (H) as average string length.

3) *Conditions Count* (C): Counts logical operators (AND, OR) in the rule's condition section. Higher C indicates complex rules capturing greater malware variability and heterogeneous sub-clusters, where different variants require different string combinations. Complex conditions ($C > 5$) are harder to interpret and maintain, but may be necessary for diverse malware families. Rules with simpler conditions are easier to maintain, while complex families require higher operator counts. Given two rules with similar TP and FP rates, compactness is preferable. We measure compactness using two separate metrics: the number of strings (N) and their average length (H). Fewer and shorter strings simplify rule interpretation and validation. The conditions count (C) is independent and lower, especially when a single logical statement captures many strings. Malware with a simple structure yields concise rules with low C and N, while complex malware families require more strings and potentially more conditions for robust detection, increasing the validation burden. Table 5 provides detailed metrics for all generated rules across the configurations tested across thresholds (Th50–Th90). SSDeep and J-SDhash configurations produce shorter, sparser rules (OR typically near 1; AND roughly 2–5), while VT-driven configurations yield the longest hex strings and highest AND counts, reflecting more complex conjunctions.

B.5. Evaluation of YARA Rules on Large Cluster

In Section 6, we provide a comprehensive characterization of both the AutoYara and AutoPYara frameworks, demonstrating how performance observations can vary significantly if not correctly contextualized. We explore the operational extremes of these tools, including their theoretical performance limits, and offer guidance on achieving optimal results in real-world deployments. However, the initial results encompassed the entire dataset, which includes a high volume of small clusters that may skew the overall performance metrics.

In this appendix, we isolate and demonstrate the impact of larger clusters on tool efficacy. Figure 15 illustrates the

TABLE 5. RULE CHARACTERISTICS: EMBER VS. RETRAINED BLOOM FILTERS

Method	Ember Bloom Filters						Retained Bloom Filters					
	Avg HexLen	Avg OR	Avg AND	Med HexLen	Med OR	Med AND	Avg HexLen	Avg OR	Avg AND	Med HexLen	Med OR	Med AND
AutoYara												
SSdeep-Th50	471.9	1.1	3.8	32	1	3	362.9	1.1	2.6	8	1	1
SSdeep-Th60	446.1	1.1	3.8	24	1	3	315.5	1.0	2.4	8	1	1
SSdeep-Th70	391.1	1.1	3.2	24	1	3	277.4	1.0	2.2	8	1	1
SSdeep-Th80	306.7	1.1	3.3	24	1	3	239.0	1.0	2.0	8	1	1
SSdeep-Th90	296.6	1.1	3.3	24	1	3	180.4	1.0	1.7	8	1	1
J-SDhash-Th50	341.9	1.1	3.1	8	1	0	403.5	1.0	2.7	8	1	1
J-SDhash-Th60	394.8	1.1	2.8	8	1	0	377.0	1.1	2.4	8	1	1
J-SDhash-Th70	410.0	1.1	3.0	8	1	0	337.7	1.1	2.4	8	1	1
J-SDhash-Th80	449.5	1.1	3.2	8	1	0	310.6	1.0	2.2	8	1	1
J-SDhash-Th90	362.3	1.1	2.4	8	1	1	249.4	1.0	1.9	8	1	1
VT	966.1	1.3	26.7	128	1	5	1041.7	1.3	37.5	160	1	5
AutoPYara												
SSdeep-Th50	318.3	1.1	4.2	24	1	3	259.1	1.1	2.8	8	1	1
SSdeep-Th60	296.8	1.1	3.9	24	1	3	243.5	1.1	2.6	8	1	1
SSdeep-Th70	258.5	1.1	3.1	16.0	1	2	201.7	1.0	2.2	8	1	1
SSdeep-Th80	222.0	1.1	3.1	24	1	3	179.9	1.0	1.9	8	1	1
SSdeep-Th90	244.9	1.1	3.0	24	1	3	138.1	1.0	1.7	8	1	1
J-SDhash-Th50	235.3	1.1	3.2	8	1	1	325.1	1.1	2.9	8	1	1
J-SDhash-Th60	208.3	1.1	2.7	8	1	1	287.0	1.1	2.6	8	1	1
J-SDhash-Th70	193.0	1.1	2.7	8	1	1	246.9	1.1	2.3	8	1	1
J-SDhash-Th80	174.8	1.1	2.8	8	1	1	225.6	1.0	2.2	8	1	1
J-SDhash-Th90	138.5	1.1	2.3	8	1	1	181.9	1.0	1.9	8	1	1
VT	671.5	2.7	34.7	80	1	5	678.3	3.2	36.7	96	1	5
AutoPyara BEST K												
SSdeep-Th50	417.2	1.3	4.8	24	1	3	302.3	1.1	3.1	8	1	1
SSdeep-Th60	325.9	1.2	4.1	24	1	3	275.5	1.1	2.6	8	1	1
SSdeep-Th70	316.3	1.2	3.1	16	1	2	237.3	1.1	2.3	8	1	1
SSdeep-Th80	248.7	1.2	3.0	24	1	3	201.2	1.1	2.1	8	1	1
SSdeep-Th90	223.2	1.2	2.9	24	1	3	154.5	1.1	1.7	8	1	1
J-SDhash-Th50	321.4	1.2	3.5	8	1	1	366.8	1.1	3.1	8	1	1
J-SDhash-Th60	278.2	1.2	3.0	8	1	1	314.9	1.1	2.8	8	1	1
J-SDhash-Th70	244.8	1.1	2.8	8	1	1	279.2	1.1	2.5	8	1	1
J-SDhash-Th80	208.8	1.1	2.7	8	1	1	256.2	1.1	2.3	8	1	1
J-SDhash-Th90	180.3	1.1	2.5	8	1	1	208.8	1.1	1.9	8	1	1
VT	727.2	2.7	31.8	88	1	4	721.6	3.2	35.7	96	1	5
AutoPyara Worst K												
SSdeep-Th50	587.8	1.1	4.3	24	1	3	382.3	1.1	2.5	8	1	1
SSdeep-Th60	584.0	1.1	4.1	24	1	3	342.8	1.0	2.3	8	1	1
SSdeep-Th70	459.1	1.1	3.1	16	1	2	300.4	1.0	2.1	8	1	1
SSdeep-Th80	508.6	1.1	3.3	24	1	3	252.4	1.0	1.9	8	1	1
SSdeep-Th90	477.9	1.1	3.3	24	1	3	187.1	1.0	1.7	8	1	1
J-SDhash-Th50	386.5	1.1	3.1	8	1	1	432.1	1.1	2.6	8	1	1
J-SDhash-Th60	357.9	1.1	2.7	8	1	1	388.3	1.0	2.4	8	1	1
J-SDhash-Th70	328.4	1.1	2.7	8	1	1	344.1	1.0	2.4	8	1	1
J-SDhash-Th80	295.8	1.1	3.1	8	1	1	301.2	1.0	2.2	8	1	1
J-SDhash-Th90	261.6	1.1	2.3	8	1	1	265.6	1.0	1.9	8	1	1
VT	1068.2	1.3	33.6	128	1	5	1157.2	1.3	39.6	144	1	5

performance delta between the two frameworks across top-performing configurations. Our findings show that AutoPYara maintains a consistently higher detection rate compared to the baseline AutoYara, even when small clusters are excluded from the analysis. This distinction is critical, as small clusters often introduce noise into evaluation metrics and tend to display inflated performance trends due to a lack of structural variability.

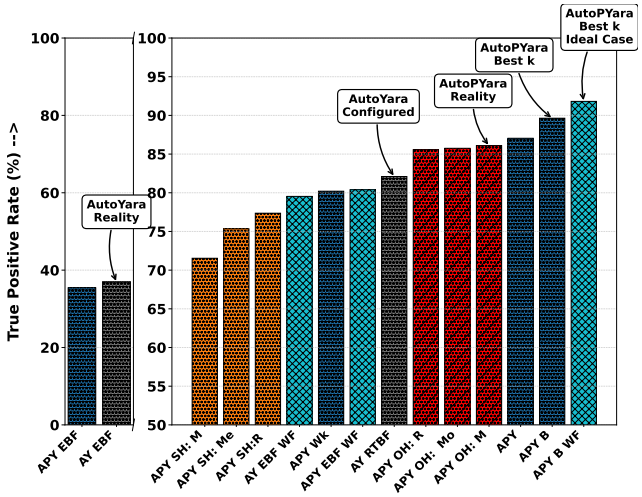


Figure 15. AutoYara v/s AutoPYara using large clusters

Appendix C. Recommendations

Throughout the evaluation in Section 5, we demonstrated how different technical specifications and strategies can result in varied performance, emphasizing that these results should not be taken at face value. To transition our scientific investigation into practical application, we have compiled our findings into the following best-practice guidelines:

1) *Retrain Bloom Filters* [C2]
Retrain with up-to-date malware and goodware samples. Outdated filters miss new byte patterns, reduce rule quality, and increase false positives.

2) *Compare Average not Single Shot Rules* [C3,C6]
Clustering results can change with random seeds. Run small ensembles with different seeds for robustness. Track stability with cumulative metrics rather than single-run averages. Generating a single rule might mislead the user given the non-deterministic nature of the framework.

3) *Accuracy is Important But Ensure Low FP Rate* [C4]
It is important that generated rules are validated against both malicious and critically goodware samples (FP rates matter). Characterized into two categories (Goodware FP rate and Cross Cluster FP rate), Goodware FP is more critical as it impacts usability at endpoint devices.

4) *Favor Byte-Based Similarity Clustering Over AntiVirus Family Clustering* [C1,C4]
Given the heterogeneous nature of the task, favor similarity-based clustering over family-based clustering, as intrinsic correlations in the data are better captured and result in higher quality rules.

5) *Calibrate Threshold for Coverage not Precision* [C5]
Thresholds impact the formation of the clusters. Higher thresholds create tighter clusters and more specific rules; lower thresholds broaden coverage, but the performance gain is not due to the changes in threshold rather just an illusion due to the fewer number (higher the threshold fewer the samples per cluster) of samples.

6) *Calibrate Rule Generation with Centroid Selection to Improve Precision* [F1]
We demonstrate how Centroid selection can be adapted into the AutoYara framework, and show that it results in up to 13 % absolute gain in performance. We also highlight how the gain of centroid selection is greater in larger, more realistic clusters.

7) *Calibrate k Values When Possible* [F2]
We demonstrate that the values of k (in other words, the number of clusters) have a profound impact on the performance of the framework. The absolute difference in performance between the best and worst k can be up to 35% in large clusters, highlighting its importance.

8) *Use AntiVirus Family to Build an Oracle for k* [F3]
Build an oracle to suggest k from historical results. Use family labels as a coarse filter to narrow the k range, and use similarity hashes for centroids in actual clustering. Keep the oracle updated as new data arrives.