

F-800: Agentic Fuzzing Termination via Function Clustering: Toward Smarter Early Stopping

WENZHANG YANG*, Institute of AI for Industries, Chinese Academy of Sciences, China

LIANG DING*, University of Science and Technology of China, China

YINXING XUE†, Institute of AI for Industries, Chinese Academy of Sciences, China

LIJUN ZHANG, Institute of AI for Industries, Chinese Academy of Sciences, China

Fuzzing is a testing technique that generates a large number of inputs to cause program crashes. As software development accelerates and projects scale, the demand for fuzz testing in software assurance has increased. Performing comprehensive fuzz testing on all functions has become increasingly challenging and resource-intensive. Current methods for determining when to stop fuzz testing activities rely on metrics such as function coverage, potential vulnerability function coverage or crash count. However, these metrics fail to account for the scale of the functions under test. For example, function coverage may lead to excessive testing on non-critical functions, while vulnerability function coverage can result in premature termination if the estimated number of vulnerability functions is too low.

This paper introduces a novel fuzzing termination tool, F-800, an agent based on function clustering. F-800 first establishes a relationship between function clusters and vulnerability distribution, then refines clusters using functional summaries. It subsequently monitors code coverage of each cluster to determine whether each function cluster has been sufficiently tested. The fuzzing campaign terminates once neither the coverage of function clusters nor the number of crash occurrences within specific clusters continues to increase. Our experiments on eight function libraries demonstrate that F-800 significantly improves testing efficiency, reducing fuzzing time by 1.4–7.2 hours (5–30%) across different configurations while maintaining minimal bug loss (averaging 0.25 bugs), outperforming existing approaches such as potential-vulnerability or function-coverage-based methods.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**.

Additional Key Words and Phrases: Fuzzing, AI Agent, Code Similarity

ACM Reference Format:

Wenzhang Yang, Liang Ding, Yinxing Xue, and Lijun Zhang. 2018. F-800: Agentic Fuzzing Termination via Function Clustering: Toward Smarter Early Stopping. *J. ACM* 37, 4, Article 111 (August 2018), 47 pages. <https://doi.org/XXXXXXXX.XXXXXXX>

*Both authors contributed equally to this research.

†corresponding author

Authors' Contact Information: [Wenzhang Yang](mailto:wzhyang@iaii.ac.cn), wzhyang@iaii.ac.cn, Institute of AI for Industries, Chinese Academy of Sciences, Nanjing, Jiangsu, China; [Liang Ding](mailto:dingliang@mail.ustc.edu.cn), dingliang@mail.ustc.edu.cn, University of Science and Technology of China, Hefei, Anhui, China; [Yinxing Xue](mailto:yxxue@ustc.edu.cn), yxxue@ustc.edu.cn, Institute of AI for Industries, Chinese Academy of Sciences, Nanjing, Jiangsu, China; [Lijun Zhang](mailto:zhanglj@ios.ac.cn), zhanglj@ios.ac.cn, Institute of AI for Industries, Chinese Academy of Sciences, Nanjing, Jiangsu, China.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM 1557-735X/2018/8-ART111

<https://doi.org/XXXXXXXX.XXXXXXX>

1 INTRODUCTION

Fuzzing [41], an automated vulnerability detection technique, involves injecting abnormal data into target programs to trigger potential security defects. In recent years, as software complexity [58] has increased, fuzzing techniques have evolved from random mutation to more intelligent approaches. These advances include combining coverage feedback [6, 55, 57], symbolic execution [52], and AI optimization [12]. For instance, large language models are used to generate high-quality test seeds, and probabilistic models like HyperGo [33] enhance vulnerability detection speed. Fuzzing applications are now widespread across various fields, including intelligent connected vehicles [63], databases [61], and operating system kernels [27], significantly improving the depth of vulnerability discovery.

However, existing termination criteria for fuzzing face limitations, especially when dealing with security vulnerabilities, which are a small fraction of all software defects. Conventional fuzzing often wastes resources on non-critical code, and imprecise stopping mechanisms lead to prolonged, inefficient testing [4]. Balancing fuzz testing time and effectiveness is crucial to avoid overlooking important security risks. However, research in this area is limited, highlighting the need to further explore and optimize termination criterion to enhance software security and reliability.

Current fuzz testing activities primarily use crash counts or function coverage to determine termination. Some recent studies [35] suggest using the growth of potential vulnerability function counts as a termination criterion. However, these methods present several notable issues [28]. The reliance on crash counts often leads to numerous duplicate crashes, which can result in either premature or delayed test termination. When tests terminate too early, important vulnerabilities may remain undetected, and excessive duplicates can waste valuable testing time and resources. Similarly, relying on function coverage may lead to testing non-critical areas, wasting resources and diverting attention from more critical sections of the code that may harbor significant vulnerabilities.

The potential vulnerability function coverage criterion effectively resolves the limitations of the two previous criteria. However, it can also cause premature test termination when the predicted number of potential vulnerability functions is too low, risking the omission of deeper, more subtle vulnerabilities that may not be immediately apparent. Overall, these limitations underscore the necessity for more comprehensive and adaptive termination criteria that can effectively balance efficiency with thoroughness in fuzz testing.

We find that a small number of functional clusters typically contain the majority of vulnerabilities, while others remain relatively clean. This demonstrates that similar functions often exhibit related types of vulnerabilities, which can be leveraged to design more effective termination criteria for fuzz testing. Therefore, this paper proposes a novel agentic approach for fuzz testing termination. The fuzzing campaign is stopped when either (1) the number of covered function clusters ceases to increase (though not all functions need to be covered, indicating sufficient test coverage), or (2) crash occurrences within specific clusters no longer grow (demonstrating testing effectiveness). The proposed method involves several key steps. First, agentic cluster analysis is performed on the functions under test to determine cluster membership for each function, providing a clearer understanding of functional relationships. F-800 monitors both the number of clusters that have been exercised and the number of crashes triggered within these clusters to identify the optimal stopping point. Specifically, the termination is triggered when changes in either code coverage across functional clusters or crash occurrences within any cluster fall below a predefined threshold over a specified time window.

The implementation of this study comprises two key components: First, encoding the functions in the target C library with CodeBERT, followed by grouping these functions through a clustering algorithm. The optimal number of clusters is determined using internal clustering metrics and a

Table 1. Crash Dataset Overview

Subject	# Functions	# Crashes	# Clusters	# Buggy Clusters
Libpcap	497	595	96	2
LibTIFF	826	17,701	67	2
Libxml2	2,982	50,907	43	2
nm	2,126	58	77	1
objdump	2,701	4,014	98	3
size	2,101	77	91	1
Total	11,233	73,352	472	14

cluster number selection algorithm. Furthermore, to refine the function clusters, F-800 examines the functional summaries of each cluster and moves functions to more appropriate clusters as needed. The clustering approach is validated on the POJ-104 dataset, and its external metrics are analyzed. Second, the study empirically evaluates the cost-effectiveness trade-offs. We reuse the pre-generated dataset from GreenFuzz [35], which includes eight target programs, to evaluate our approach. The dataset comprises results from eight state-of-the-art grey-box fuzzing tools, each executed in 20 independent runs, yielding a total of 1,280 fuzzing campaigns. Each campaign was conducted for 24 hours.

Contributions. Our work makes the following key contributions:

- We propose a novel relationship between vulnerabilities and function clustering, demonstrating that vulnerabilities tend to concentrate in specific function clusters.
- We design an innovative Multi-Objective Optimal Clustering Number Selection algorithm to automatically determine the optimal number of function clusters, addressing a critical challenge in code analysis.
- We devise an AI agent, F-800, to identify the optimal stopping point for fuzzing. First, it refines the clusters obtained from the traditional vector similarity clustering method to produce more coherent groupings. Second, F-800 monitors both code coverage and the number of crashes during the fuzzing campaign to enable early stopping.
- Through extensive experiments, we show that F-800 significantly outperforms existing methods in terms of computational efficiency while maintaining detection accuracy.
- We open-source all relevant code and datasets to support reproducibility and encourage further research advancement in the community.

To support open science, we release our work at: <https://sites.google.com/view/fcbft>

2 MOTIVATION

In the domain of software security testing, recent years have witnessed a paradigm shift in vulnerability detection from coarse-grained to fine-grained [20] approaches. With the rapid and continuous evolution of modern software applications, traditional fuzz testing methods based solely on code coverage increasingly struggle to balance testing sufficiency and resource efficiency. Academic research has widely observed that vulnerabilities are not uniformly distributed across codebases but exhibit significant correlations with functional units' semantic features, control flow patterns, and dependency networks [36]. This non-uniform distribution characteristic provides a theoretical foundation for constructing intelligent testing termination criteria—by identifying vulnerability-dense regions through function clustering, dynamic regulation of testing processes becomes achievable.

148 However, existing termination criteria predominantly rely on single-dimensional metrics such
149 as time thresholds or coverage saturation, failing to effectively integrate code semantic features
150 with vulnerability risk propagation patterns, which leads to substantial misallocation of testing
151 resources.

152 **Agentic Fuzzing Stopping Criterion based on Function Cluster.** This approach maintains
153 testing coverage while significantly reducing testing time and resource costs. Functions are clus-
154 tered based on characteristics such as functional similarity, potential vulnerability risks, and code
155 dependency relationships. This allows for the identification of high-priority subsets, enabling
156 focused testing on critical areas. In fuzz testing, using crash counts as an effectiveness metric
157 can lead to underestimation or overestimation [22], potentially missing vulnerabilities or causing
158 excessive testing duration. Code coverage, while reducing underestimation risks, may overesti-
159 mate effectiveness by assuming uniform code importance and covering non-critical areas. Lipp et
160 al. [35] proposed an improved method focusing on coverage saturation in potentially vulnerable
161 code regions identified through vulnerability prediction. However, this method may terminate
162 prematurely if too few potential vulnerable functions are predicted.

163 To overcome these limitations, we propose an AI Agentic termination criterion that assesses
164 fuzzing progress at the level of function clusters. The fuzzer F-800 continuously observes, reasons,
165 plans, and executes to optimize both effectiveness and efficiency. The observation phase involves
166 clustering functions, while simultaneously collecting cluster-level coverage and crash information.
167 The agent then reasons over these clusters to identify high-priority areas and assess testing progress.
168 Based on this reasoning, the agent plans the termination of the fuzzing campaign, deciding when
169 further testing is unlikely to yield new insights. Finally, the agent executes the termination by
170 stopping the fuzzing process once the criteria at the cluster level are satisfied. This agentic workflow
171 ensures adaptive, informed decision-making, balancing thorough fuzzing of critical code regions
172 with efficient use of resources.

173 **Relationship between Function Cluster and Vulnerability.** Function clustering [26] is a
174 grouping method based on semantic similarity [31], functional proximity [25], or structural depen-
175 dencies [1] among functions. In recent years, with the application of deep learning techniques in
176 code analysis [56], code-semantic-based function clustering has become a research hotspot. Analyz-
177 ing the relationship between function clustering and vulnerabilities reveals several key points. First,
178 there is a connection between function semantic similarity and vulnerability distribution [19, 51]. In
179 practical software, certain vulnerabilities exhibit distinct reproducible characteristics, where code
180 with similar functionalities tends to harbor similar vulnerabilities. For example, common buffer
181 overflow vulnerabilities in memory management functions and unchecked input vulnerabilities in
182 input handling functions. To validate the relationship between function clustering and vulnerability
183 distribution, we utilize a comprehensive real-world fuzz testing dataset from Fuzztastic [34], which
184 includes basic-block-level coverage and crash logs from 12 subject programs. More specifically, we
185 perform the validation directly on the dataset from GreenFuzz [35], which comprises only eight
186 programs. Among these, two lack crash logs because the fuzzing campaign triggered very few
187 crashes. We begin by performing function clustering for each target program. Taking Libpcap as
188 an example, its 497 functions are organized into 96 clusters. A cluster is designated as buggy if any
189 function within it leads to a crash, as indicated by the corresponding crash logs. Table 1 reports the
190 number of buggy clusters identified after function clustering and crash analysis. Figure 1 presents
191 the proportion of buggy clusters across all target libraries. For each library, the buggy proportion is
192 computed as $\frac{|\text{Buggy Clusters}|}{|\text{All Clusters of 6 Libraries}|}$. Statistical analysis shows that vulnerabilities are confined to
193 only 14 clusters (approximately 2.9%) out of 472, with Figure 1 further confirming their marginal
194
195
196

197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245

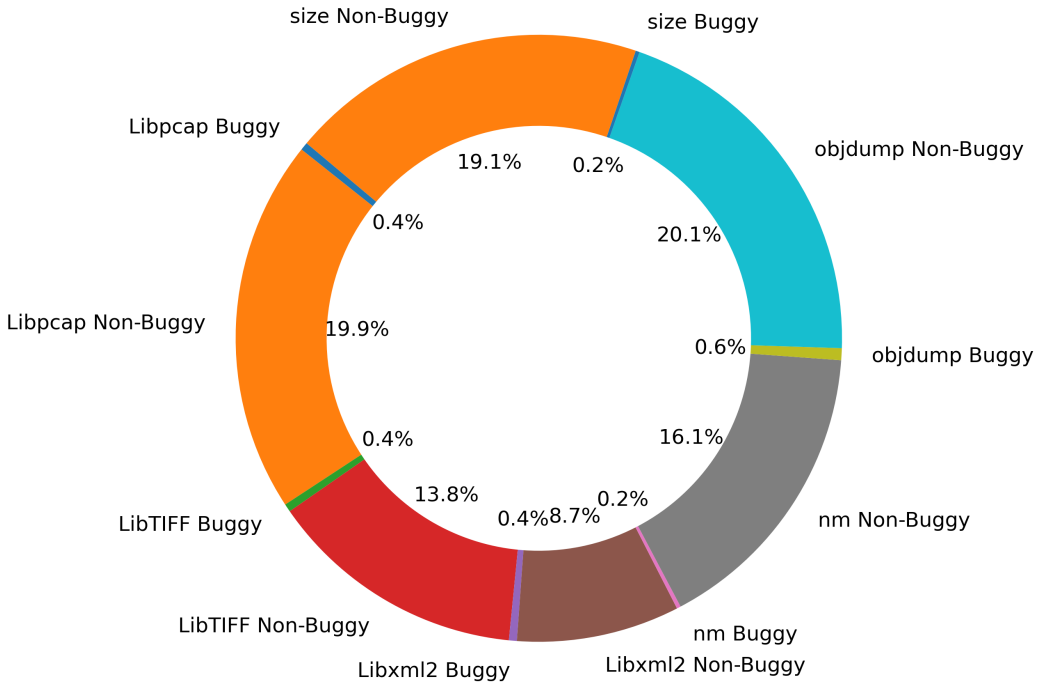


Fig. 1. Proportion of Buggy Function Clusters per Target

presence across all function libraries. Therefore, this study aims to reduce resource wastage in greybox fuzzing by exploiting the statistical distribution of vulnerabilities.

3 APPROACH

This section is divided into three parts. The Function Clustering section presents a CodeBERT-based method for grouping functions and determining the optimal number of clusters. The next section, Cluster Refinement, demonstrates how LLMs reorganize clusters through code summaries. Finally, we present the application of the proposed agentic termination criterion in fuzz testing.

3.1 Function Clustering

Algorithm 1 enables fully automated end-to-end computation by intelligently determining the optimal number of clusters through a multi-objective optimization [49] framework. The process begins by normalizing and vectorizing input functions using CodeBERT [16] embeddings. Next, clustering quality is evaluated for each candidate k using multiple complementary metrics. Potential values of k are identified through a hybrid approach that combines the elbow method with silhouette coefficient peak detection. Non-dominated solutions are then used to construct the Pareto frontier, and the final optimal k is selected via entropy-weighted multi-criteria decision-making, ensuring a balanced trade-off between all quality metrics.

Algorithm 1 Multi-Objective Optimal Clustering Number Selection**Input:** Function set F , Search range of cluster numbers $[k_{\min}, k_{\max}]$ **Output:** Optimal cluster number k^* 1: **1. Code Normalization & Vectorization**2: $\mathbf{X} \leftarrow \emptyset$ 3: **for** each function $f \in F$ **do**4: $f' \leftarrow \text{CodeNormalization}(f)$ 5: $\mathbf{x} \leftarrow \text{CodeBERT}(f')$ 6: $\mathbf{X} \leftarrow \mathbf{X} \cup \{\mathbf{x}\}$ 7: **end for**8: **2. Multi-Criteria Evaluation**9: **for** $k = k_{\min}$ to k_{\max} **do**10: $\{C_1, \dots, C_k\} \leftarrow \text{K-Means}(\mathbf{X}, k)$ 11: Compute SSE, Silhouette Coefficient, CH Index, and DBI for current k 12: **end for**13: **3. Candidate Subset Selection**

14: Compute second-order differences of SSE

15: $\mathcal{K}_{\text{elbow}} \leftarrow \{k \mid \Delta^2(k) < \theta_e\}$ 16: $\mathcal{K}_{\text{sil}} \leftarrow \{k \mid \text{Local maxima of Silhouette}\}$ 17: $\mathcal{K}_{\text{candidate}} \leftarrow \mathcal{K}_{\text{elbow}} \cap \mathcal{K}_{\text{sil}}$ 18: **4. Pareto Frontier Construction**19: $\mathcal{P} \leftarrow \{k \in \mathcal{K}_{\text{candidate}} \mid \nexists k' \in \mathcal{K}_{\text{candidate}}, F(k') \succ F(k)\}$ 20: **5. Entropy-Weighted Decision**

21: Compute entropy weights for each metric

22: Compute score for each candidate cluster number

23: **return** $k^* \leftarrow \arg \max_{k \in \mathcal{P}} \text{Score}(k)$

3.1.1 *Motivation.* Traditional methods for function analysis, which rely on surface-level code structure or text matching, often fail to capture deep semantic relationships. To address this, we leverage CodeBERT, a pre-trained model adept at extracting semantic features from code, to generate function embeddings. These embeddings are clustered using K-Means to group functionally similar code segments. However, assessing clustering quality and determining the optimal number of clusters remain challenging. Instead of ad-hoc heuristics, we employ multiple internal clustering validation indices to evaluate cluster compactness, separation, and stability from complementary perspectives. To robustly select the cluster count, we integrate the elbow method with local peak analysis of internal indices to narrow the search space, then apply Pareto optimization [11] to resolve multi-metric trade-offs. This approach ensures semantically coherent clusters while mitigating biases inherent to single-index evaluation.

3.1.2 *Function Encoding.* In this study, we leverage the CodeBERT model to perform vector encoding on functions, aiming to capture the semantic and structural features of code. Initially, the function code is formatted into an acceptable input format using CodeBERT's tokenizer, with special tokens added. Subsequently, the formatted code is fed into the CodeBERT model to generate high-quality 768-dimensional vector representations. These vectors effectively reflect the contextual information and structural characteristics of the code. Finally, the generated vectors are stored in association with their original functions, providing a solid foundation for the subsequent K-Means clustering analysis.

3.1.3 *K-Means Clustering*. Upon completing the function vector encoding, this study adopts the K-Means clustering algorithm for function analysis. As a classical unsupervised learning algorithm, K-Means is widely used in large-scale dataset clustering due to its simplicity and efficiency. To evaluate the quality of the K-Means clustering results, this study employs four internal clustering metrics—Silhouette Score [46], Calinski-Harabasz Index [10], Davies-Bouldin Index [13] and Sum of Squared Errors [40]. These metrics provide a comprehensive assessment of cluster quality by measuring distinct aspects such as compactness, separation, and balance. This multi-metric approach enhances robustness and mitigates bias that may arise from relying on a single evaluation criterion.

Sum of Squared Errors (SSE). SSE is a common metric for assessing clustering compactness. It calculates the sum of squared distances between each data point and its cluster center across all clusters. The formula for SSE is:

$$SSE = \sum_{i=1}^k \sum_{x \in C_i} \|x - \mu_i\|^2 \quad (1)$$

Here, k denotes the number of clusters, C_i represents the set of data points in the i -th cluster, μ_i is the center of the i -th cluster, and x is a data point in the cluster. A smaller SSE value indicates tighter clustering of data points within clusters, reflecting better clustering performance.

Silhouette Coefficient (SC). The SC evaluates clustering results by considering both intra-cluster cohesion and inter-cluster separation. Its value ranges from -1 to 1, with a higher value implying better clustering quality. The calculation process is as follows: For the i -th data point x_i , define:

- $a(x_i)$ as the average distance from x_i to other data points in the same cluster (cohesion)
- $b(x_i)$ as the minimum average distance from x_i to data points in other clusters (separation)

The silhouette coefficient for x_i is:

$$s(x_i) = \frac{b(x_i) - a(x_i)}{\max(a(x_i), b(x_i))} \quad (2)$$

The overall silhouette coefficient is the average of all data points' silhouette coefficients:

$$S = \frac{1}{n} \sum_{i=1}^n s(x_i) \quad (3)$$

Here, n represents the total number of data points.

Calinski-Harabasz Index(CH Index). The CH index assesses clustering quality by comparing inter-cluster dispersion to intra-cluster cohesion. A higher CH value indicates better clustering. The formula is:

$$CH = \frac{TR_{ss} \times (n - k)}{SC_{ss} \times (k - 1)} \quad (4)$$

where TR_{ss} is the between-cluster sum of squares and SC_{ss} is the within-cluster sum of squares. They are calculated as:

$$TR_{ss} = \sum_{i=1}^k n_i \|\mu_i - \mu\|^2, \quad SC_{ss} = \sum_{i=1}^k \sum_{x \in C_i} \|x - \mu_i\|^2 \quad (5)$$

Here, n is the total number of samples, k is the number of clusters, n_i is the number of samples in the i -th cluster, μ is the center of all samples, and μ_i is the center of the i -th cluster.

Davies-Bouldin Index (DBI). The DBI assesses clustering results based on cluster compactness

and separation. A smaller DBI value signifies better clustering. The formula is:

$$DBI = \frac{1}{k} \sum_{i=1}^k \max_{j \neq i} \left(\frac{\sigma_i + \sigma_j}{d(c_i, c_j)} \right) \quad (6)$$

Here, σ_i measures the compactness of the i -th cluster, calculated as:

$$\sigma_i = \sqrt{\frac{1}{n_i} \sum_{x \in C_i} \|x - c_i\|^2} \quad (7)$$

$d(c_i, c_j)$ is the distance between cluster centers c_i and c_j , often computed using the Euclidean distance:

$$d(c_i, c_j) = \|c_i - c_j\| \quad (8)$$

These metrics evaluate clustering from multiple perspectives, offering a comprehensive assessment of clustering quality and reliability.

3.1.4 Determining the Number of Clusters. For the selection of the number of clusters, we designed the Enhanced Elbow Method and Silhouette Coefficient Peak Detection for preliminary screening. Then, the Pareto Frontier Algorithm is utilized to filter a subset of candidate values. Finally, the Entropy-Weight Method is applied to compute the optimal number of clusters.

Enhanced Elbow Method. The traditional elbow method relies on visual inspection of the SSE (Sum of Squared Errors) curve's inflection point to determine the optimal number of clusters, which is subjective and unstable. The enhanced method captures the trend of inflection points by analyzing the first-order and second-order differences of the sum of squared errors (SSE). The first-order difference $\Delta(k)$ measures the change in SSE between consecutive points, reflecting the rate at which the error decreases. The second-order difference $\Delta^2(k)$ measures the change in this rate, highlighting points where the trend accelerates or decelerates—i.e., potential inflection points. Using these second-order differences, the method automatically sets the threshold θ_e as the mean plus two standard deviations, effectively distinguishing significant trend changes from normal fluctuations. Finally, it selects the smallest k value satisfying $\Delta^2(k) < \theta_e$ as the optimal number of clusters. This approach not only reduces human intervention but also more accurately captures the "diminishing returns" point of the SSE curve. It demonstrates strong applicability across different data distributions, significantly improving robustness.

Silhouette Coefficient Peak Detection. The silhouette coefficient evaluates clustering quality by measuring both intra-cluster compactness and inter-cluster separation, with the optimal number of clusters typically corresponding to local maxima of the coefficient curve. While traditional methods simply select the global maximum, this approach can be misled by data noise or random fluctuations. Our enhanced method addresses this limitation by first computing the first derivative ($S'(k)$) of the silhouette coefficient sequence, then identifying candidate peaks through zero-crossing points (where $S'(k) = 0$). These candidates are further refined by selecting only those local maxima whose magnitudes exceed a significant threshold (e.g., 30% of the global maximum). This two-stage process effectively mitigates noise sensitivity while preserving the silhouette coefficient's inherent ability to discern cluster structure, resulting in more stable and robust determination of the optimal cluster number.

Pareto-Optimal Solution Determination. In the context of clustering analysis, the evaluation of clustering quality often involves multiple internal indices, such as the Silhouette Coefficient, Davies-Bouldin Index, Calinski-Harabasz Index, and SSE. These indices, while providing valuable insights into different aspects of clustering performance, often exhibit conflicting characteristics [2]. For instance, optimizing one index may lead to suboptimal results for another, as each index

Algorithm 2 Non-dominated Sorting Algorithm**Input:** Candidate set $K_{candidate}$ **Output:** Non-dominated fronts \mathcal{F}

```

395 1: for all  $k_i \in K_{candidate}$  do
396 2:   Compute dominance counter  $n_i$  and dominated set  $S_i$  for  $k_i$ 
397 3: end for
398 4: Initialize front rank  $r \leftarrow 1$ 
399 5: Initialize front  $\mathcal{F}_r \leftarrow \emptyset$ 
400 6: repeat
401 7:   for all ungraded  $k_i$  do
402 8:     if  $n_i == 0$  then
403 9:       Assign  $k_i$  to front  $\mathcal{F}_r$ 
404 10:    end if
405 11:   end for
406 12:   for all  $k_j \in \mathcal{F}_r$  do
407 13:     for all  $k_m \in S_j$  do
408 14:       Decrement  $n_m$  by 1:  $n_m \leftarrow n_m - 1$ 
409 15:     end for
410 16:   end for
411 17:   Increment  $r$  by 1:  $r \leftarrow r + 1$ 
412 18: until all solutions are graded

```

prioritizes different aspects of clustering quality, such as compactness, separation, or variance ratios. Algorithm 2 outlines a non-dominated sorting process. It starts by computing dominance counters and dominated sets for all candidates. Then, it initializes the first front. In each iteration, it identifies non-dominated solutions (those with a dominance counter of zero) and assigns them to the current front. For each solution in the current front, it updates the dominance counters of the solutions they dominate. This process repeats until all solutions are sorted into fronts, effectively categorizing them based on their dominance relationships.

Entropy-Weight Method. To objectively rank clustering candidates derived from the preceding steps, the Entropy-Weight Method (EWM) is adopted. EWM quantifies the relative importance of evaluation indices by measuring their information entropy, thus reducing subjective bias. Indices with lower entropy, indicating higher discriminative power across clustering candidates, are assigned greater weights, ensuring a balanced and data-driven evaluation. Specifically, the indices considered are SSE, SC, CH Index, and DBI.

Following EWM, the entropy H_j of each index is first computed, and the objective weight w_j is then derived accordingly, providing a data-driven basis for ranking clustering candidates.

The entropy value H_j and the corresponding weight w_j are defined as:

$$w_j = \frac{1 - H_j}{\sum_{k=1}^n (1 - H_k)}, \quad H_j = -\frac{1}{\ln(m)} \sum_{i=1}^m p_{ij} \ln(p_{ij}) \quad (9)$$

Here, m denotes the number of alternatives, $n = 4$ denotes the four evaluation indices considered in this study, x_{ij} the raw score on index j for alternative i , p_{ij} its normalized value obtained by dividing x_{ij} by the total score of index j across all alternatives, H_j the entropy of index j , and w_j its weight.

3.2 Cluster Refiner

While vector-based clustering captures the semantic relationships among functions, achieving a precise fine-grained logical semantic understanding remains a challenge. Therefore, F-800 audits each cluster to identify potential outliers and reassigns them to more appropriate clusters. This agent consists three components: *Function Summarization*, *Outlier Analysis*, *Function Reassignment*.

Function Summarization. In practice, a single cluster often contains numerous functions, making it infeasible to fit all of them into the context window of an LLM. To address this limitation, we leverage LLMs to analyze each function individually and generate a concise summary as its semantic representation. The prompt used by F-800 incorporates both the function name and its source code, as shown below:

Function Analysis and Summarization Prompt

System Prompt: You are a professional software engineering analyst.

User Prompt: Please analyze the following function code and provide a summary in one natural English paragraph covering the following five dimensions:

1. Functional category (e.g., image processing, memory management, file operations, utility functions, etc.)
2. Code complexity (based on lines of code, number of control structures)
3. Algorithm characteristics (algorithm patterns and data processing methods used)
4. Performance characteristics (compute-intensive, I/O-intensive, etc.)
5. Key features (core technical characteristics of the function)

Please generate one natural paragraph summarizing the above aspects.

{FUNCTION NAME}

{CODE}

Outlier Analysis. After the function summarization stage, F-800 proceeds to identify outliers within each cluster. It constructs a structured prompt containing the function name, return type, parameters, and the corresponding function summary, and then queries the LLM to determine which functions do not fit well within the cluster. Here, the symbol `[]*` denotes a list of elements in the prompt template, which may be empty. The prompt template used by F-800 is shown below:

Outlier Analysis Prompt

System Prompt: You are a software engineering researcher responsible for code cluster analysis and understanding.

User Prompt: The following is a set of functions clustered into the same category. Please analyze based on their function names, return types, and parameters:

1. What is the functional basis of this cluster? (e.g., all perform image parsing, pixel iteration, memory operations, etc.)
2. Which functions might be unsuitable for this cluster (if none, return an empty list)?
3. Briefly describe the overall purpose and relationship of these functions.

Please output strictly in the following JSON format:

{FEW SHOT EXAMPLES}

Functions are shown below:

[{FUNCTION NAME}

{FUNCTION TYPE}

{FUNCTION PARAMETERS}

{SUMMARY}]*

Function Reassignment. In this stage, F-800 selects the five closest clusters based on the distance between the cluster centers and the function representation. The excluded functions, together with these five nearest clusters, are then provided to the LLM, which determines the most appropriate target cluster for each excluded function. The prompt template used by F-800 is shown below:

Function Reassignment Prompt

System Prompt: You are a JSON-only assistant. You must output exactly and only valid JSON format.

User Prompt: Reassign an excluded function of one cluster to the most appropriate alternative cluster. A function was excluded from cluster:

{CLUSTER SUMMARY}

You must select the best alternative from the candidate clusters below.

Criteria for Selection:

1. Functional similarity with cluster's purpose
2. Overall behavioral alignment

Function to Reassign:

{FUNCTION NAME}

{FUNCTION RETURN TYPE}

{FUNCTION PARAMETERS}

{FUNCTION CODE}

Candidate Clusters:

[{CLUSTER SUMMARY}]*

Please output strictly in the following JSON format:

{FEW SHOT EXAMPLES}

After the three components, F-800 completes the cluster refinement process. Through this multi-stage pipeline, F-800 enhances the semantic coherence and structural consistency of function clusters. The *Function Summarization* component transforms raw code into compact semantic representations that capture high-level functionality, algorithmic behavior, and performance traits. The *Outlier Analysis* component employs LLM reasoning to detect semantically inconsistent functions that deviate from the cluster's dominant theme. Finally, the *Function Reassignment* component ensures these outliers are relocated to the most appropriate clusters by combining embedding-based proximity with LLM-driven semantic judgment. Overall, this agentic process mitigates the noise and semantic drift commonly observed in pure vector-based clustering, yielding more precise and interpretable cluster structures that better reflect functional and logical relationships among code segments.

3.3 Cluster Cover Determination

While clustering reveals the functional relationships among code segments, determining whether a cluster is actively exercised during fuzzing requires temporal execution analysis. To this end, F-800 continuously evaluates the coverage state of each cluster over time. This component integrates execution coverage data with iterative LLM reasoning to decide whether a cluster is "covered" at each timestamp.

Since traditional coverage (e.g., line, branch, or edge coverage) is often excessively fine-grained and redundant, F-800 aggregates execution statistics at the function level. For each function, we aggregate basic block execution statistics by taking the maximum execution count among all its basic blocks, which is used as the representative function-level coverage metric. We aim to enable more aggressive early stopping by using the maximum execution count. Unlike average or median

counts, which can delay termination due to low-activity or typical blocks, the maximum reflects the most exercised path, allowing fuzzing to stop as soon as at least one path is sufficiently explored. This approach accelerates overall exploration while still focusing on high-activity, potentially critical behaviors, aligning with our goal of efficient and aggressive fuzzing. At each timestamp, F-800 determines whether a cluster is covered based on both its historical and current cluster-level execution coverage. The current coverage data is then recorded as historical coverage for the subsequent timestep. Specifically, it combines the cluster's functional summary with its temporal coverage history and submits them to an LLM prompt that infers whether the cluster should be considered covered. If the previous three timestamps were all marked as covered, F-800 automatically labels the current timestamp as covered to capture sustained activity patterns and reduce redundant LLM queries. The prompt template used by F-800 is shown below:

Cluster Coverage Analysis Prompt

System Prompt: You are a software execution pattern analyst. You determine when code clusters are actively used. Focus on execution patterns, not functional semantics.

User Prompt: Based on the historical test coverage up to the current timestamp and the recent changes in coverage behavior, analyze whether this function cluster is currently 'covered'. This means that the function cluster is actively fully being executed, tested, or explored during fuzzing. Compare current coverage increasing with previous coverage states. Detect significant changes in execution frequency, diversity, or intensity. Determine whether new functional behaviors are being triggered. Assess the proportion of active functions relative to total cluster size. Identify overall trends – e.g., increasing, stable, or declining coverage.

Coverage Decision Criteria:

coverage = 1: Notable or increasing execution activity; multiple functions or key paths are being exercised, indicating ongoing effective fuzzing.

coverage = 0: Minimal, repetitive, or stagnant execution; no meaningful new coverage, suggesting fuzzing may have reached saturation.

Historical Code Coverage of this Cluster:

{Previous Coverage}

Current Code Coverage of this Cluster:

{Current Coverage}

Please output strictly in the following JSON format:

{FEW SHOT EXAMPLES}

3.4 Termination Criterion for Fuzz Testing

Definition. The growth of covered clusters with the saturation time window δ is defined as:

$$\Delta C(t) = C(t) - C(t - \delta), \quad (10)$$

where $C(t)$ represents the number of covered clusters at time t . As fuzz testing progresses, if $\Delta C(t)$ remains below a threshold ϵ_C for a sustained period, it indicates that new test cases no longer trigger significant behavioral patterns. By monitoring the trend of $\Delta C(t)$, F-800 can dynamically assess whether the newly generated test cases meaningfully expand behavioral coverage.

Similarly, crash count growth is defined as:

$$\Delta F(t) = F(t) - F(t - \delta), \quad (11)$$

where $F(t)$ represents the number of crashes detected at time t . All crashes are deduplicated based on call stack traces to ensure that only unique crash events are counted. In fuzz testing, crash events

are critical indicators for identifying vulnerabilities. If $\Delta F(t)$ remains below a threshold ϵ_F for an extended duration, it suggests that the fuzzer has reached saturation in detecting new crashes.

Algorithm 3 Fuzzing Termination Decision

Require: Window size δ , thresholds ϵ_C, ϵ_F , budget T_{\max}

```

1: Initialize window index  $k \leftarrow 0$ 
2: Initialize Sets  $C_{\text{seen}} \leftarrow \emptyset$  {Seen function clusters}
3: Initialize Sets  $\mathcal{F}_{\text{seen}} \leftarrow \emptyset$  {Seen crashes}
4: while  $k \cdot \delta < T_{\max}$  do
5:   Initialize local Sets  $C_{\text{new}} \leftarrow \emptyset$ 
6:   Initialize local Sets  $\mathcal{F}_{\text{new}} \leftarrow \emptyset$ 
7:   Execute fuzzing for window  $[k\delta, (k + 1)\delta)$ 
8:   for all execution  $e$  in window do
9:     Let  $c \leftarrow \text{ClusterID}(e)$ 
10:    Let  $f \leftarrow \text{CrashID}(e)$ 
11:    if  $c \notin C_{\text{seen}} \wedge \text{QueryLLMsIfCovered}(c)$  then
12:      Add  $c$  to  $C_{\text{new}}$  and  $C_{\text{seen}}$ 
13:    end if
14:    if  $f \notin \mathcal{F}_{\text{seen}}$  then
15:      Add  $f$  to  $\mathcal{F}_{\text{new}}$  and  $\mathcal{F}_{\text{seen}}$ 
16:    end if
17:  end for
18:   $\Delta C(k\delta) \leftarrow |C_{\text{new}}|$ 
19:   $\Delta F(k\delta) \leftarrow |\mathcal{F}_{\text{new}}|$ 
20:  if  $\Delta C(k\delta) < \epsilon_C \vee \Delta F(k\delta) < \epsilon_F$  then
21:    break
22:  end if
23:   $k \leftarrow k + 1$ 
24: end while

```

As shown in Algorithm 3, the termination mechanism for fuzz testing is governed by window-based saturation and tolerance thresholds. The fuzzing process is divided into fixed-length saturation windows of size δ . Rather than relying on a single composite evaluation metric, we explicitly track incremental exploration and bug discovery using two signals: newly covered function clusters and newly discovered unique crashes. During fuzzing, the system maintains global sets that record all previously observed function clusters and crashes. For each saturation window, local sets are used to collect function clusters and crashes that are newly discovered within the current window. The incremental progress of exploration and bug discovery is then quantified as the number of newly observed function clusters, $\Delta C(k\delta)$, and the number of newly observed unique crashes, $\Delta F(k\delta)$, respectively. The fuzzing process is considered saturated and terminates early when either exploration stagnates or bug discovery stagnates (line 20). These criteria indicate that additional fuzzing within the current configuration is unlikely to yield substantial new cluster coverage or additional distinct crashes.

Table 2. Cluster Dataset Overview

Subject	Version	LoC	# Functions
Gif2png	2.5.3	988	27
JasPer	1.900.0	17,385	720
Libpcap	1.9.0	12,076	497
LibTIFF	4.1.0	19,527	826
Libxml2	2.9.10	85,466	2,982
nm	2.29	68,667	2,126
objdump	2.29	89,961	2,701
size	2.29	68,115	2,101
POJ-104	–	1,885,262	168,990
Total	–	2,247,447	180,970

4 EXPERIMENTAL SETUP

4.1 Research Questions

Our fuzzing termination criterion is only effective when the clustering results are accurate. If the clustering performance is poor, the fuzzing campaign may fail to maintain a sufficient scale, thereby inadequately covering most function types. Specifically, an insufficient number of clusters could lead to premature termination before discovering bugs, whereas an excessive number of clusters would render the criterion similar to function coverage-based approaches, losing the advantage of early termination. Based on these considerations, we raise the following two research questions:

4.1.1 RQ.1. Function Clustering Performance. How do different clustering methods compare in functional clustering tasks, and can a multi-metric fusion algorithm effectively identify the optimal number of clusters?

4.1.2 RQ.2. Analysis of Fuzzing Termination Criteria. What is the trade-off between reduced fuzzing time and potential missed security bugs when using function cluster coverage combined with crashes in special clusters as a stopping criterion, compared to traditional metrics like crash count, function coverage, or potential vulnerability function coverage?

4.2 Function Clustering

4.2.1 Subject Programs. To validate the effectiveness of our clustering approach and the proposed algorithm for selecting optimal cluster numbers, we employ the POJ-104 [42] dataset—a well-established benchmark for function-level code analysis. This dataset contains 104 programming problems, each solved by multiple users in C/C++, resulting in numerous syntactically diverse but semantically similar implementations. Such characteristics make POJ-104 ideal for evaluating clustering methods, as solutions to the same problem inherently form functional clusters despite variations in coding style. The dataset’s implicit ground truth (functions solving identical problems) enables rigorous assessment of clustering accuracy. POJ-104’s balance of diversity and consistency has made it a standard choice for studies on code clone detection, AI-assisted programming, and semantic code similarity.

4.2.2 Function Encoding. We employ Tree-sitter [8] to parse the compiled library’s AST and extract the implementation code for all functions, which we identify by filtering nodes where `decl.kind == "Function"`; in this study, a function is strictly defined as a standard C99 function,

Table 3. Parameter Configuration for Stopping Criteria Evaluation

Parameter	Values	Interpretation
Saturation window (δ)	2,4,6,8h	Stability assessment period
Tolerance (ϵ)	Varies by criterion:	
- Crash-based	0,1,2	Maximum new crashes in δ
- Coverage-based	0,1,2	New functions covered in δ
- Cluster-based	0,1,2	New crash clusters in δ
Repetitions	20	Independent experimental runs

encompassing both its declaration and full implementation. Then, we employ the CodeBERT-base model to extract features from source code in the POJ-104 dataset. Each code snippet is truncated/padded to a fixed length of 512 tokens before being fed into the model, and the 768-dimensional vector corresponding to the [CLS] token is extracted as the code representation. The tokenizer normalizes functions with `truncation=True` and `padding=True`, fixing each snippet to 512 tokens—the maximum input length of CodeBERT, since it follows the BERT-base [14] architecture. The experiments are implemented using the PyTorch framework, with the pretrained model loaded via the HuggingFace Transformers library. All feature extraction processes are executed in batches on GPU, and the resulting feature vectors are stored in numpy format for downstream tasks.

4.2.3 Clustering Method. In this clustering analysis, we employed a variety of methods beyond just K-Means. We also utilized DBSCAN [48], which is effective for identifying clusters of arbitrary shapes and handling noise; Agglomerative Clustering [43], a hierarchical approach that builds nested clusters; and Gaussian Mixture Models (GMM) [45], which models data as a mixture of Gaussian distributions to capture more complex cluster structures. Among them, methods such as K-Means, GMM, and Agglomerative Clustering (which require the number of clusters k as an input) are used for internal comparisons to validate the effectiveness of our approach, whereas DBSCAN (which does not require k) is contrasted with these methods to highlight its necessity. The optimal method is selected for subsequent fuzz testing. Pertaining to the algorithm’s parameters, DBSCAN does not require the number of clusters k as an input. Instead, we sample five values of the neighborhood radius ϵ , chosen between the 10th and 90th percentiles of the pairwise distance distribution, to ensure a stable operational range. For the other three algorithms, the number of clusters k is explicitly specified and varied from 2 to 600. The upper bound of 600 is determined by assuming a minimum of three functions per cluster, given the total number of functions in the target library.

4.2.4 Evaluation Metrics. Since the POJ-104 dataset contains ground truth labels, we employ two external evaluation metrics—Adjusted Rand Index (ARI) [53] and Normalized Mutual Information (NMI) [2] to quantitatively assess clustering performance. ARI measures the similarity between the clustering results and ground truth by correcting for chance agreements, while NMI evaluates the mutual dependence between cluster assignments and true labels after normalization. These complementary metrics provide rigorous validation of how accurately each clustering algorithm reconstructs the known classification structure. The combined use of both metrics offers comprehensive insights into different aspects of clustering quality relative to the reference classification.

4.3 Tradeoff Evaluation

4.3.1 *Dataset.* Our study leverages the FuzzTastic [34] dataset, which provides comprehensive crash data and fine-grained coverage metrics from eight state-of-the-art greybox fuzzers: AFL [59], AFLFast [6], AFL++ [17], AFLSmart [44], FairFuzz [32], Honggfuzz [54], MOpt-AFL [37], and MOpt-AFL++ [39]. These fuzzers were evaluated on eight widely used libraries and utilities (Table 2), including Gif2png, JasPer, Libpcap, Libxml2, LibTIFF, nm, objdump and size.

Each fuzzer was executed for 24 hours with 20 repetitions, using initial seeds from either AFL’s repository or the Magma [22] suite. The dataset enables a robust analysis of efficiency-effectiveness trade-offs in fuzzing, particularly how reducing fuzzing time impacts bug detection rates.

4.3.2 *Evaluation Metrics.* Following the evaluation criteria established in this reaserch [35], we conduct a systematic evaluation of dynamic stopping criteria in greybox fuzzing. Our comparative analysis examines both conventional baselines (crash saturation, code coverage and potential vulnerable function coverage metrics) and the proposed approach. In Table 3, for the saturation window δ and tolerance ϵ parameters, we adopt the same settings from Green Fuzzing [35] to ensure a fair and accurate comparison. We employ two primary metrics:

- **Time savings ratio:** $\eta_t = \frac{\Delta t}{T_{total}} \times 100\%$, where $\Delta t = T_{total} - t_S$ ($T_{total} = 24h$)
- **Missed bug ratio:** $\eta_b = \frac{N_{missed}}{N_{total}} \times 100\%$

4.3.3 *Comparison.* To align with the function-level granularity of our approach, we aggregate basic-block coverage data at function boundaries extracted by Tree-sitter.

For vulnerable function coverage, we predict potentially vulnerable functions in the target libraries using six machine learning approaches—Random Classifier, Generalized Linear Model, Multi-Layer Perceptron, Random Decision Forest, Stochastic Gradient Boosting, and Support Vector Machine—adopted from GreenFuzz [35], and track their coverage during fuzzing. All six implementations are obtained from the GreenFuzz GitHub repository, and we re-run the vulnerability prediction experiments. The fuzzing process is terminated if, within a specified saturation window (e.g., 2 hours), no additional vulnerable functions are covered and the coverage increase remains below a defined tolerance (ϵ); in such cases, both the runtime and the number of discovered bugs are recorded.

5 EVALUATION

5.1 RQ1:Function Clustering Performance.

The experimental investigation into clustering methodologies applied to the POJ-104 dataset reveals nuanced patterns in algorithmic performance and cluster number determination. Across K-Means, DBSCAN, Agglomerative, and GMM approaches, the internal validation metrics exhibit characteristic trade-offs between cluster cohesion and separation.

5.1.1 RQ1.a: How do different clustering methods perform for functional clustering tasks?

Based on a comparative analysis of four clustering methods (K-Means, DBSCAN, Agglomerative, and GMM), the internal evaluation metrics reveal distinct trends. Figure 2 presents the performance of the four clustering algorithms evaluated using four internal metrics—SSE, SC, DBI, and CH Index—as well as two external metrics, ARI and NMI in POJ-104. As shown in Figure 2, we implemented a preliminary version of hierarchical clustering. It showed limited effectiveness for functional clustering tasks where the optimal number of clusters must be determined adaptively. The silhouette coefficient (SC) and Calinski-Harabasz index (CH) exhibit a sharp decline at $k = 2$, stabilizing around 0.027 and 14, respectively. The Davies-Bouldin index (DBI) gradually increases to 3.0088 within $k = 2$ to $k = 47$, followed by a slow decline, while the sum of squared errors

785

786

787

788

789

790

791

792

793

794

795

796

797

798

799

800

801

802

803

804

805

806

807

808

809

810

811

812

813

814

815

816

817

818

819

820

821

822

823

824

825

826

827

828

829

830

831

832

833

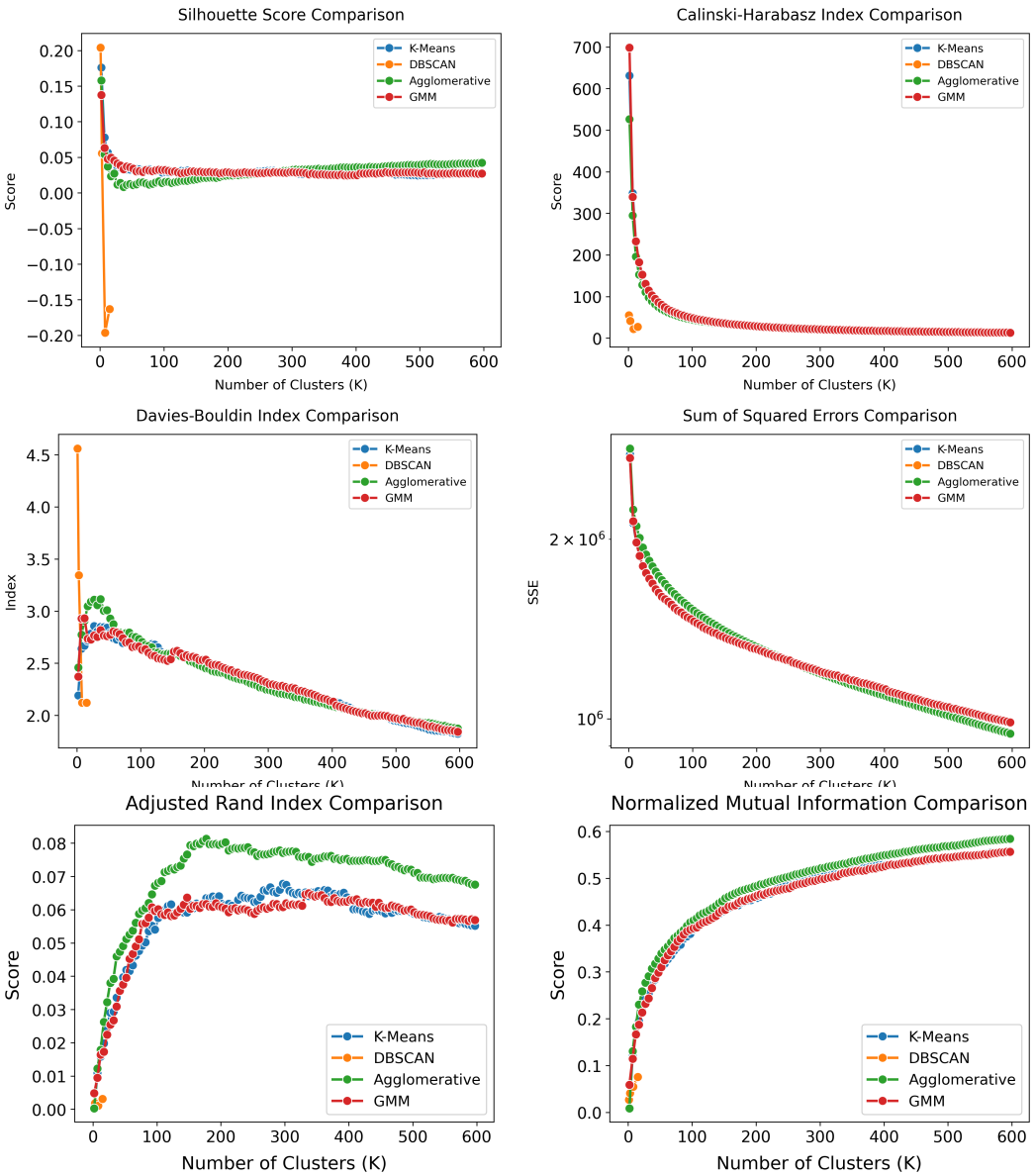


Fig. 2. Clustering Metrics (Top) and External Metrics Comparison (Bottom)

(SSE) shows an initial steep drop, leveling off after $k = 105$. For clustering algorithms such as DBSCAN, which do not require a predefined number of clusters k but instead rely on parameter settings, the resulting clustering performance is generally not superior in terms of evaluation metrics compared with the other three methods, and thus cannot be considered optimal. Among the three primary methods, Agglomerative clustering demonstrates marginally lower SC values until $k = 247$, after which it surpasses the others, though differences remain minimal. For CH, all methods perform similarly. Notably, Agglomerative clustering yields significantly higher DBI

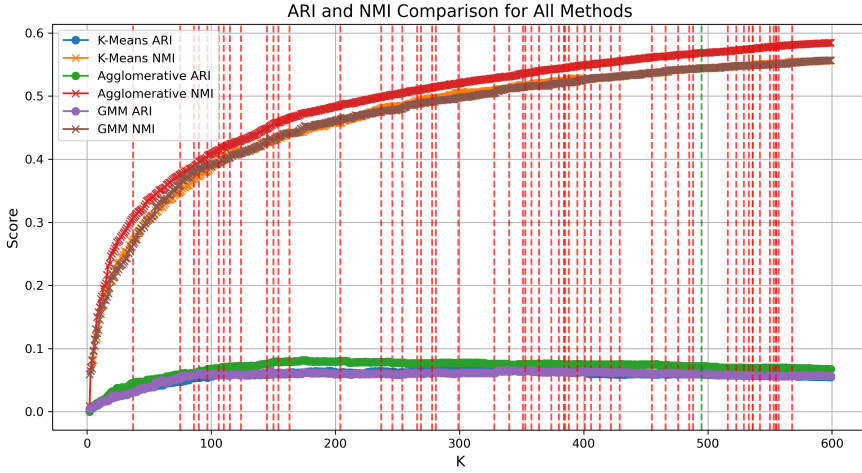


Fig. 3. ARI and NMI vs. Number of Clusters

Table 4. Accuracy (%) of Different Methods

	Complete	w/o Elbow	w/o Silhouette	w/o Pareto
Accuracy (%)	80.00	67.80	75.14	75.00

“w/o” denotes “without”.

values between $k = 12$ and $k = 57$, converging thereafter. In terms of SSE, Agglomerative clustering initially produces slightly higher values before k , but lower values subsequently. These findings suggest context-dependent performance trade-offs among the methods.

Summary(RQ.1-a). Agglomerative clustering demonstrates marginal advantages in specific metrics, such as DBI and SSE, after certain thresholds ($k = 247$ and $k = 57$, respectively). However, all methods exhibit context-dependent trade-offs, with no single method consistently outperforming the others across all evaluation metrics.

5.1.2 RQ1.b: Can the proposed cluster number selection algorithm effectively identify optimal clusters? Figure 3 displays the ARI and NMI scores corresponding to the clusterings selected by the clustering selection algorithm on the POJ-104 dataset. The proposed clustering algorithm demonstrates robust performance in automatically selecting the optimal number of clusters across varying k values ($k = [37.0, 75.0, \dots, 550.0]$) with Figure 3. Evaluation metrics reveal a normalized mutual information (NMI) score of 79% (exceeding the 0.5 threshold) [50], indicating strong cluster separation. However, the adjusted Rand index (ARI) yields consistently low values (< 0.08), which—combined with visual assessment showing no distinct cluster patterns—suggests ARI may be unsuitable for this dataset in Figure 4. For the objective function library, conventional supervised metrics (NMI/ARI) cannot be computed due to missing ground-truth labels.

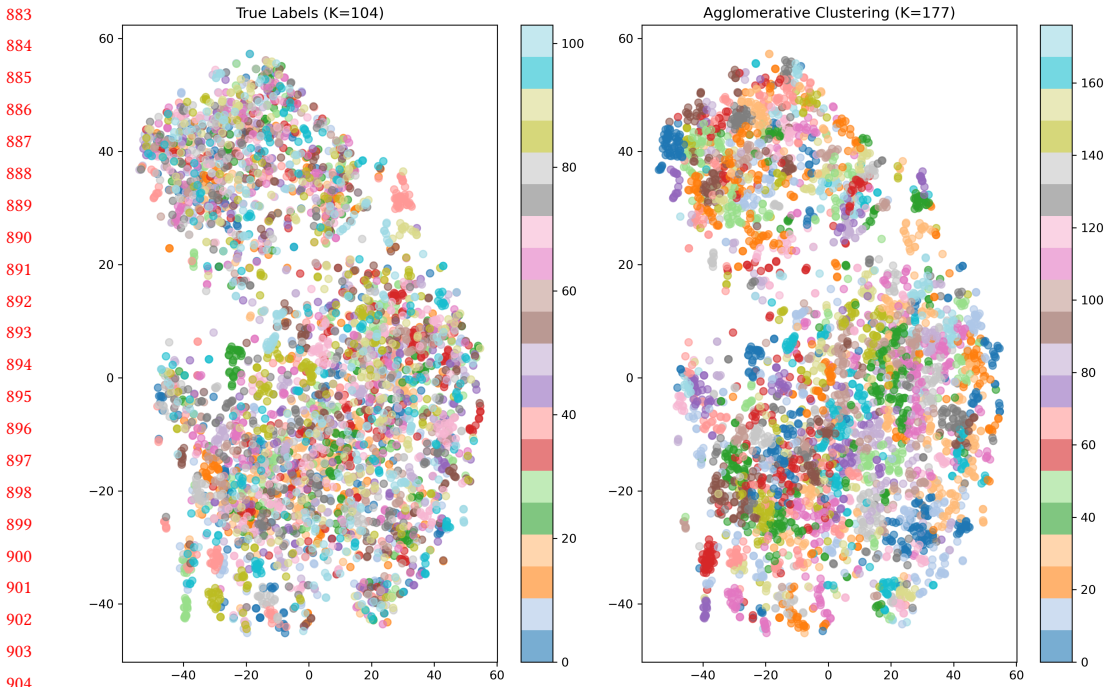


Fig. 4. Visualization of Clustering Results

Summary(RQ.1-b). Despite ARI's low values (< 0.08), the proposed algorithm achieves strong performance, with a normalized mutual information (NMI) score of 79%, surpassing the 0.5 threshold for meaningful cluster separation.

5.1.3 RQ1.c: How effective are the individual components of the clustering algorithm according to the ablation study? We conduct an ablation study to assess the contribution of each component in determining the number of clusters by sequentially removing the Enhanced Elbow method, Silhouette Coefficient Peak Detection, and Pareto optimization. The EWM ranks candidate clusterings, so an ablation study is not applicable here. Performance is measured by the ratio of selected k values with $NMI \leq 0.5$. Table 4 presents the results of the ablation study conducted under the K-means setting. The results show that the complete method achieves the highest accuracy (80.00%), underscoring the synergistic effect of all components. Removing the Elbow module results in the largest drop in accuracy (67.80%), indicating its critical importance. Excluding Silhouette or Pareto also reduces performance (75.14% and 75.00%, respectively), but to a lesser extent. Overall, all modules contribute to the final performance, with the Elbow method being the most significant.

Summary(RQ.1-c). The ablation study confirms that each component contributes positively to the overall performance, as evidenced by the drop in accuracy when any module is removed, with the complete method achieving the highest accuracy.

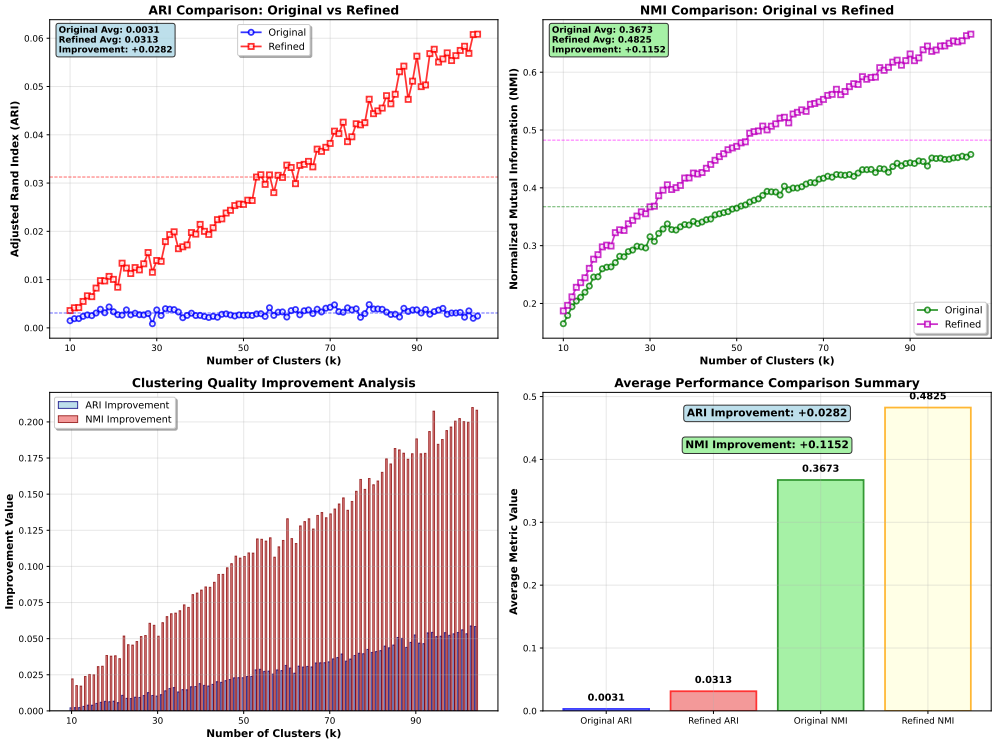


Fig. 5. Comparison of ARI and NMI Scores Before and After Refinement

5.1.4 RQ1.d: How effectively does the Cluster Refiner improve the overall quality of clustering results? To assess the effectiveness of the proposed Cluster Refiner, we compare the clustering quality before and after refinement using ARI and NMI—across different numbers of clusters ($k=10,30,50,70,90$) on the POJ-104 dataset. As shown in Figure 5, both ARI and NMI exhibit consistent improvements after applying the Cluster Refiner. Specifically, the average ARI increased from 0.0031 to 0.0313, yielding an absolute gain of +0.0282. Similarly, the average NMI rose from 0.3673 to 0.4825, corresponding to an improvement of +0.1152. This enhancement demonstrates that the refined clusters align more closely with the underlying ground-truth distribution, indicating a higher level of semantic coherence. The improvement trend is stable across different values of cluster number K , confirming that the refiner effectively enhances clustering quality regardless of the cluster granularity. The ARI improvement suggests that the refined process reduces misassignments between clusters, while the NMI improvement implies that inter-cluster information is more consistent and discriminative.

Overall, these results verify that the Cluster Refiner substantially enhances cluster purity and stability. The improvement in both ARI and NMI metrics indicates that the refiner can better capture latent semantic relationships among functions, thereby producing clusters that are more representative and meaningful for downstream analysis and fuzzing termination decision-making.

Table 5. Robustness of LLM-Based Cluster Refinement Across Repeated Executions

Program	Exclusion Rate (%)	Consistency (%)	Function Stability (%)
gif2png	0.71	5.0	100
jasper	6.79	60.8	99.5
libpcap	9.76	66.8	98.8
libtiff	9.96	62.0	97.7
libxml2	10.38	61.3	99.7
nm	9.26	64.0	99.7
objdump	9.67	56.4	99.6
size	9.93	66.1	99.8

Summary(RQ.1-d). The Cluster Refiner substantially improves clustering quality, with average ARI and NMI increases of +0.0282 and +0.1152, respectively. These consistent gains across different cluster sizes indicate that the refiner effectively enhances semantic coherence and cluster stability, leading to more accurate and meaningful cluster structures for fuzzing termination.

5.1.5 RQ1.e: how robust LLM-based cluster refinement are when repeated multiple

times? Since LLM-based analysis is inherently stochastic, it is critical to evaluate whether our cluster refinement procedure produces stable and reproducible results when executed multiple times with identical inputs and prompts. To assess robustness, we repeated the complete LLM-based cluster refinement process, including both outlier analysis and function reassignment, 20 times for each library using identical prompts, inputs, and hyperparameters. We then quantitatively evaluated the stability of LLM decisions and final clustering structures using three complementary robustness metrics. We adopt the following three metrics to characterize robustness at different granularities:

- **Exclusion Rate.** (E_i) measures the proportion of functions identified as outliers by the LLM at iteration i :

$$E_i = \frac{\sum_{j=1}^M |E_{ij}|}{\sum_{j=1}^M N_{ij}} \times 100\%$$

where M is the number of clusters, $|E_{ij}|$ is the number of excluded functions in cluster j , and N_{ij} is the total number of functions in cluster j .

- **Consistency Score.** (C_i) measures the agreement of LLM outlier decisions across consecutive runs using Jaccard similarity:

$$C_i = \frac{1}{M} \sum_{j=1}^M \frac{|E_{(i-1)j} \cap E_{ij}|}{|E_{(i-1)j} \cup E_{ij}|}$$

where $E_{(i-1)j}$ and E_{ij} denote the excluded-function sets for cluster j in two consecutive iterations.

- **Function Stability Score.** (S_f) quantifies how consistently each function is assigned to the same cluster across repeated runs:

$$S_f = 1 - \frac{\sum_{t=2}^T \mathbb{I}(c_{ft} \neq c_{f(t-1)})}{T - 1}$$

where c_{ft} is the cluster assignment of function f at iteration t , T is the total number of repetitions, and $\mathbb{I}(\cdot)$ is the indicator function.

In Table 5, across all evaluated libraries, the exclusion rate remains within a relatively narrow band for most subjects (6.79%–10.38%), indicating that the initial clustering method is largely consistent with the LLM’s assessments. The only exception is gif2png, which exhibits a substantially lower exclusion rate (0.71%), reflecting the simplicity and homogeneity of its codebase and the resulting scarcity of detected outliers. Consistency scores are generally moderate to high (56.4%–66.8%) for the majority of libraries, demonstrating stable agreement in LLM exclusion decisions between consecutive executions. Although gif2png shows a low consistency score (5%), this behavior is expected given its extremely low exclusion rate with only 20 functions, where small absolute changes in a very small exclusion set can disproportionately reduce consistency. Function stability scores are consistently high across all libraries (97.7%–100%), indicating that the vast majority of functions are assigned to the same clusters across repeated executions. This result demonstrates strong robustness of function-level reassignment and suggests that the refined clustering structures are highly reproducible. Overall, these results confirm that LLM-based cluster refinement is robust under repeated executions, with stable outlier detection behavior and highly consistent function-level assignments.

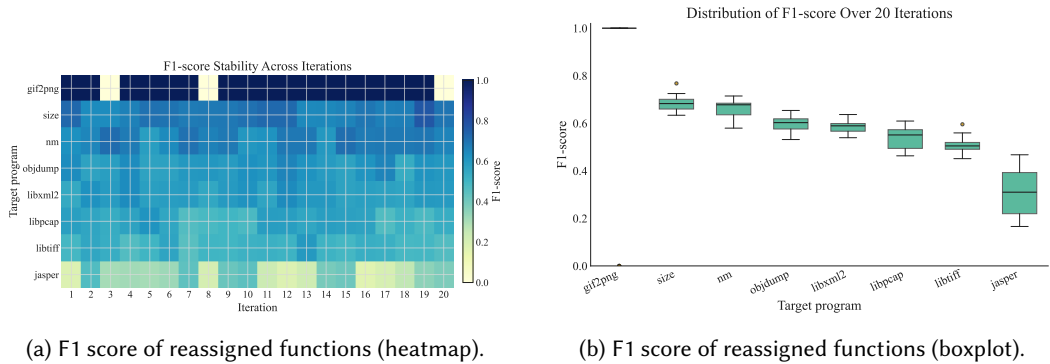


Fig. 6. F1 score results of reassigned functions.

As shown in Figure 6, we further examine the robustness of LLM-based refinement by evaluating the correctness of function reassignment across 20 repeated runs. Specifically, we manually construct a confusion matrix for each run, where **True Positive (TP)** indicates that a function is reassigned to its correct cluster, **True Negative (TN)** indicates that a function correctly remains in its original cluster, **False Positive (FP)** indicates an incorrect reassignment to a wrong cluster, and **False Negative (FN)** indicates that a function should be reassigned but is incorrectly kept in its current wrong cluster.

The heatmap in Figure 6a reports iteration-wise F1-score trajectories, highlighting how reassignment quality varies under stochastic LLM outputs. Overall, most subjects exhibit strong robustness, with consistently high F1-scores across iterations. For instance, size and nm maintain stable performance (typically around 0.65–0.70), implying that both precision and recall remain high across repeated executions. This indicates that the LLM rarely introduces FP (wrong reassignment) and also consistently identifies the majority of functions that truly require reassignment (low FN rate). Similarly, objdump and libxml2 achieve moderately high but stable F1-scores (around 0.55–0.62), suggesting that although these libraries contain more complex semantics and larger

Table 6. Pre-Fuzzing Clustering Runtime Across Evaluated Programs

Program	Analyze Clusters (s)	Reassign Excluded (s)	Total (s)
gif2png	106.40	5.64	112.04
jasper	139.11	241.28	380.38
libpcap	729.32	571.50	1300.82
libtiff	610.40	835.60	1445.99
libxml2	1221.55	2310.10	3531.64
nm	694.42	1030.57	1724.99
objdump	847.92	1370.48	2218.40
size	753.20	1055.61	1808.80

function spaces, the LLM refinement procedure still yields reproducible reassignment decisions with limited accuracy degradation. The boxplot in Figure 6b further confirms this robustness by showing narrow interquartile ranges for most libraries, meaning that accuracy and error rates (FP/FN) do not fluctuate significantly across the 20 runs. In other words, the LLM does not exhibit strong run-to-run instability, and the refinement pipeline is unlikely to produce drastically different final clustering results given identical inputs and prompts.

However, robustness is weaker for certain challenging subjects. `jasper` demonstrates both the lowest median F1-score and the largest variance. A key reason is that `jasper` contains a large number of functions (over 700) but is partitioned into only 14 clusters, resulting in overly coarse-grained clusters with a high intra-cluster functional diversity. Consequently, each cluster includes many heterogeneous functions, making outlier detection and reassignment inherently harder and more frequent. Under such a setting, even small stochastic variations in LLM judgments can trigger different exclusion and reassignment decisions, leading to substantial run-to-run fluctuations.

An interesting outlier is `gif2png`, which achieves near-perfect F1-scores in most iterations, yet shows a few extreme drops to nearly zero. This phenomenon is mainly explained by the small codebase size and extremely low number of reassigned functions: when only a few candidate functions exist, even a single FP or FN can dramatically reduce precision/recall, leading to a sharp decrease in F1-score. Therefore, these occasional failures reflect a small-sample sensitivity rather than a systematic lack of robustness in the LLM refinement mechanism.

Summary(RQ.1-e). The LLM-based cluster refinement demonstrates strong robustness under repeated executions with our experimental setting. Consistency scores are generally moderate to high, and function stability scores are consistently close to 1.0, showing that both outlier detection and function reassignment decisions are highly reproducible and robust.

5.1.6 RQ1.f: what is the overhead introduced by LLM-based function clustering? In Table 6, the total runtime for pre-fuzzing clustering varies significantly across programs, ranging from 112.04 s (`gif2png`) to 3,531.64 s (`libxml2`), reflecting differences in program size and complexity. For smaller programs such as `gif2png` and `JasPer`, the analysis stage dominates the runtime, whereas for larger programs like `libxml2`, `objdump`, and `nm`, the reassignment of excluded functions constitutes the majority of the cost, indicating that this stage scales more steeply with program complexity.

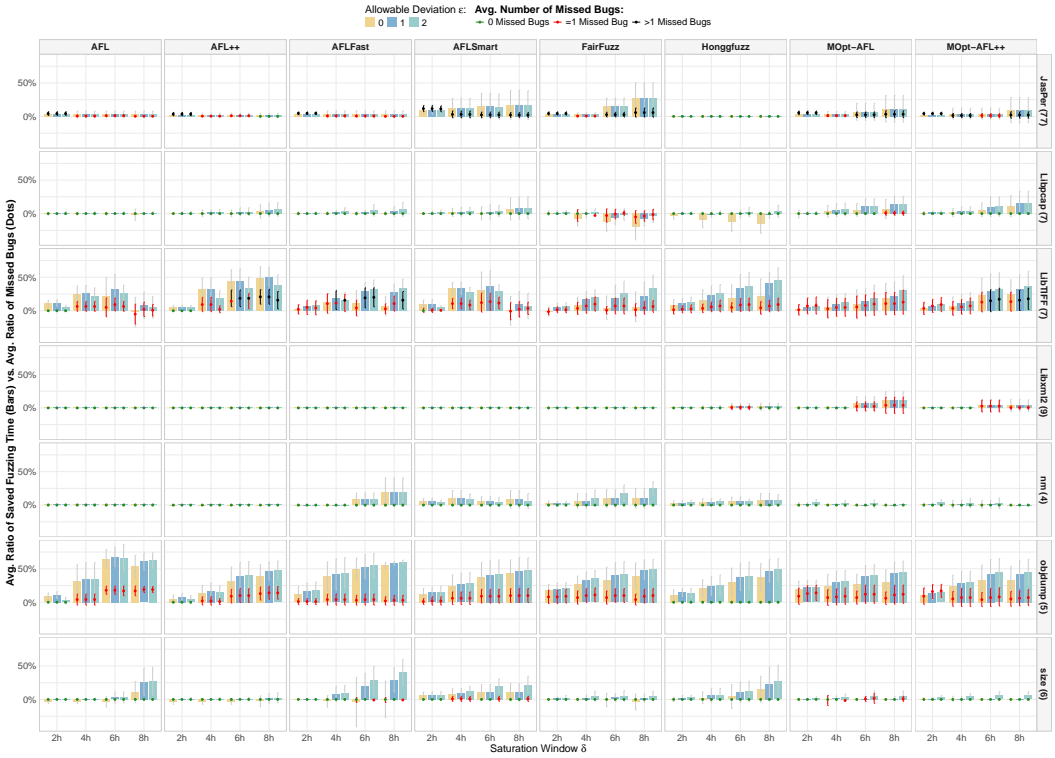


Fig. 7. Tradeoff results of our stopping criterion relative to the saturation of potential vulnerability function coverage.

The observed runtimes demonstrate that the clustering pipeline scales roughly with program size: medium-sized libraries incur tens of minutes of preprocessing, while very large libraries may require up to an hour. Notably, even for the largest programs, the total preprocessing remains practical for typical fuzzing campaigns. Since the termination decision is determined during the pre-fuzzing clustering process—before the actual fuzzing begins—it does not add overhead to the fuzzing runtime itself. Moreover, the preprocessing is executed only once per library and can be reused across multiple fuzzing campaigns, further amortizing its cost. Overall, these results indicate that LLM-based function clustering introduces a moderate and predictable overhead that scales with program size, with the reassignment stage being the primary contributor for large and complex binaries.

Summary (RQ1.f). LLM-based function clustering incurs a moderate overhead prior to fuzzing, with total preprocessing times ranging from a few minutes for small programs to nearly an hour for large libraries. The reassignment stage dominates runtime for complex binaries, and overall, the cost scales predictably with program size, remaining practical for real-world fuzzing pipelines.

Table 7. Average Performance Across All Fuzzers by Subject. T1-T3 (timeout differences in hours) quantify how much earlier our function clustering metric terminates compared to baseline methods.

Subject	Timeout (Hours)			Additional Missed Bugs		
	T1	T2	T3	1	2	3
Gif2png	7.355	0.000	0.000	0.414	0.000	0.000
JasPer	9.701	1.536	1.433	3.692	1.213	1.236
LibTIFF	7.972	6.896	4.332	0.722	0.686	0.467
Libpcap	-1.138	2.792	0.286	-0.017	0.033	-0.009
Libxml2	6.297	2.104	0.226	0.567	0.298	0.027
nm	-0.567	3.610	0.905	-0.006	0.006	0.000
objdump	5.074	8.381	7.221	0.161	0.304	0.284
size	-1.367	4.300	1.210	-0.008	0.013	0.003
Average	4.166	3.702	1.952	0.691	0.319	0.250

Abbreviations: T1: Crash Counts; T2: Code Coverage; T3: Potential Vulnerable Function Coverage;

5.2 RQ2: Analysis of Fuzzing Termination Criterion

Comparison: Crash Stopping Criterion. For Gif2png, Jasper, LibTIFF, Libxml2, and objdump, our proposed termination criterion achieved substantially earlier termination compared to the crash-count-based criterion. Table 7 compares our proposed stopping criterion with three other criteria based on time advancement and the number of missed bugs. The data in the table correspond to average values across different combinations of the saturation window δ and the tolerance parameter ϵ , as specified in Table 3. Specifically, our method reduced testing time by 5.0 to 9.7 hours, representing 20%–40% of the total fuzzing duration (Table 7). In terms of bug detection, our approach missed only 0.69 bugs on average (6% of all bugs) despite earlier termination. For Libpcap, nm, and size, our method slightly delayed termination by 0.5–1.3 hours (2%–5% of total time) but discovered additional bugs. Specifically, our approach reduces testing time by an average of 3.1 hours (13.1%) while maintaining effective bug detection - 77.5% of terminated campaigns miss no bugs, 8.6% miss one bug, and 12.7% miss two or more bugs on average. Remarkably, our criterion additionally discovers at least one extra bug in 1% of campaigns compared to crash saturation.

Summary(RQ.2-a). Our stopping criterion reduces fuzzing time by **4.1 hours** with minimal bug loss (avg. 0.7, 6%) compared with crashes saturation, while occasionally discovering additional bugs (1% cases). The approach demonstrates superior cost-effectiveness, with **77.5%** of campaigns missing no bugs despite significant time savings.

Comparison: Function Coverage Stopping Criterion. Our stopping criterion demonstrates significant efficiency improvements over traditional function-coverage saturation, reducing fuzzing time by 1.5–8.3 hours (6%–34.6%) for most test configurations while missing only 0.31 bugs on average (Table 7). For optimal cases like AFL on objdump achieved 18-hour reductions (75%), though little savings occurred for JasPer due to its high density of vulnerable functions that maintained campaign activity. Our result reveals that 81.8% of campaigns missed zero bugs, with average time savings of 4 hours (16.1%) when bugs were missed. Only 7.6% of campaigns missed two or more bugs, confirming the criterion’s reliability. The trade-off between time savings and bug detection

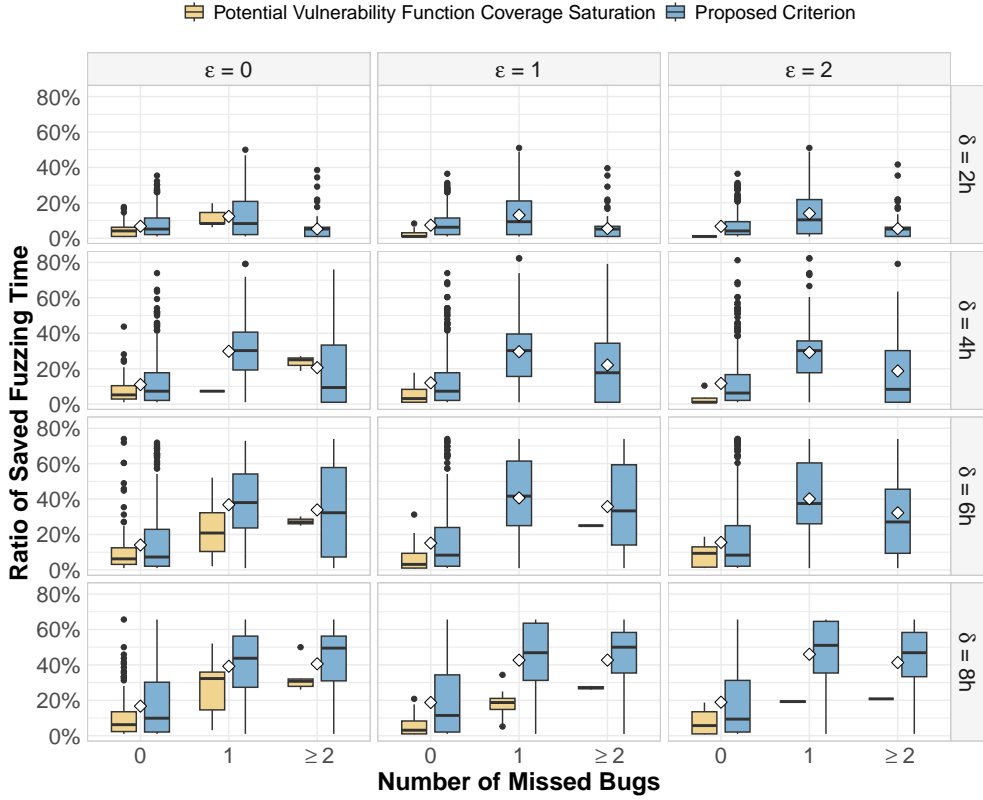


Fig. 8. Time saved by the fuzzing stopping criterion versus undetected bugs compared to potential vulnerability function coverage saturation.

remains favorable, with most configurations benefiting from substantial efficiency gains without compromising vulnerability discovery.

Summary(RQ.2-b). Our stopping criterion achieves significant efficiency improvements, reducing fuzzing time by 1.5–8.3 hours (6%–34.6%) for most configurations while missing only 0.31 bugs on average. Analysis shows 81.8% of campaigns missed zero bugs, with average time savings of 4 hours (16.1%) when bugs were missed, and only 7.6% missed two or more bugs, confirming its reliability and favorable trade-off between efficiency and bug detection.

Comparison: Vulnerability Function Coverage Stopping Criterion. Figure 7 and Figure 8 further illustrate the tradeoff results of our stopping criterion in comparison with the saturation of vulnerable function coverage. As shown in the Figure 7, the proposed fuzzing termination criterion terminates 1.4–7.2 hours earlier than the vulnerability function coverage-based approach for JasPer, LibTIFF, objdump, and size, while missing only 0.25 bugs on average. For nm and Libxml2, our method slightly outperforms the coverage-based strategy. In the case of Libpcap, although the

1275 proposed criterion extends fuzzing duration by 0.9 hours, it detects additional bugs, demonstrating
1276 its effectiveness.

1277 **Figure 8** illustrates that the majority of campaigns saved more testing time, with those that
1278 missed bugs demonstrating an average reduction in runtime of 2.1 hours (equivalent to 8%). Fur-
1279 thermore, compared to GreenFuzz, our criterion preserves complete bug detection in 86.5% of
1280 fuzzing campaigns. Instances where two or more bugs were missed accounted for only 5.1%, under-
1281 scoring the robustness of the proposed criterion. This demonstrates that the method effectively
1282 balances efficiency and accuracy, enabling significant time savings while ensuring the majority of
1283 vulnerabilities are uncovered.

1284
1285 **Summary(RQ.2-c).** The proposed fuzzing termination criterion outperforms the vulnera-
1286 bility function coverage-based approach by reducing runtime by 1.4–7.2 hours for most
1287 targets while missing only 0.25 bugs on average. Additionally, 86.5% of campaigns achieved
1288 complete bug detection, highlighting its robustness and efficiency.

1289
1290 **Sensitivity Analysis of Threshold.** As shown in **Figure 7**, when the allowable deviation is set
1291 to 0, the fuzzing process is highly conservative in its termination criterion, prioritizing higher
1292 effectiveness. This leads to a scenario where fewer bugs are missed (lower average missed bugs ratio),
1293 as the fuzzer continues running until it has explored a sufficient portion of the input space. However,
1294 this conservative approach results in longer fuzzing durations, as it requires more iterations to
1295 reach an effective stopping point. As the columns corresponding to deviation 0 generally show
1296 lower missed bug rates but higher average saved fuzzing time (lower ratios). The fuzzer's operation
1297 remains exhaustive, continuing its search longer even when bug discovery slows down.

1298 As the allowable deviation increases to 1, there is a noticeable shift in the balance between
1299 effectiveness and runtime. The fuzzer becomes more aggressive in stopping earlier, leading to
1300 a tradeoff between saved time and missed bugs. As shown in the figure, the missed bug ratio
1301 typically increases, reflecting the fact that terminating earlier can result in some vulnerabilities
1302 being overlooked. However, this comes at the benefit of a substantial reduction in fuzzing time,
1303 especially in the earlier stages of fuzzing (Saturation Window 2h, 4h). The technique still maintains
1304 a relatively good bug-finding ability, but with quicker evaluations, indicating a more optimized
1305 runtime performance.

1306 With an allowable deviation of 2, the fuzzer becomes even more aggressive in stopping early. The
1307 missed bug rate increases further as fuzzing is terminated sooner, but the saved fuzzing time is the
1308 highest across all thresholds. This shift is most visible in techniques like AFLFast, MOpt-AFL, and
1309 MOpt-AFL++, where the runtime benefits of early termination outweigh the losses in effectiveness.
1310 The most noticeable pattern here is that fuzzers that prioritize speed, like AFLFast and MOpt-AFL++,
1311 are more capable of maintaining low runtime even with an increased threshold, but this comes
1312 with a marginal increase in missed bugs.

1313
1314 **Summary (RQ.2-d).** The analysis of the stopping criterion across different allowable devia-
1315 tion thresholds (0, 1, 2) demonstrates a clear tradeoff between runtime and bug detection. As
1316 the threshold increases, fuzzing time is reduced, with AFLFast and MOpt-AFL++ showing
1317 the most significant time savings without substantial sacrifices in effectiveness. However,
1318 higher thresholds lead to a slight increase in missed bugs, especially for FairFuzz, which
1319 shows higher sensitivity to early termination, while Honggfuzz remains relatively stable
1320 across different thresholds. Overall, the approach achieves a balanced tradeoff, optimizing
1321 runtime while maintaining effective bug detection across various fuzzing techniques.

1322
1323

Table 8. F-800 Performance for Each Fuzzer.

(a) Timeout Performance (Earlier Termination in Hours vs. Baseline)

Subject	A	A++	AF	AS	FF	HF	MA	MA++	Avg
Gif2png	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
JasPer	0.0	0.0	0.0	0.0	0.0	0.0	-0.525	0.0	-0.066
LibTIFF	5.767	4.583	6.417	5.117	3.592	3.283	4.067	2.458	4.41
Libpcap	0.083	1.542	1.142	0.667	3.742	0.758	0.175	0.308	1.052
Libxml2	0.0	-0.083	0.0	-0.083	-0.917	0.0	-0.25	-0.167	-0.188
nm	-0.667	0.0	-0.083	-0.55	2.275	0.833	0.175	-0.817	0.146
objdump	0.083	0.167	-1.317	-1.533	1.283	1.083	2.05	1.55	0.421
size	0.0	0.0	0.0	-0.317	1.8	0.167	0.417	0.733	0.35
Average	0.658	0.776	0.77	0.413	1.472	0.766	0.764	0.508	0.766

(b) Bug Performance (Additional Missed Bugs vs. Baseline)

Subject	A	A++	AF	AS	FF	HF	MA	MA++	Avg
Gif2png	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
JasPer	0.0	0.0	0.0	0.0	0.0	0.0	-0.7	0.0	-0.088
LibTIFF	0.933	0.567	1.0	0.867	0.3	0.1	0.433	0.1	0.538
Libpcap	0.0	0.0	0.0	0.0	0.333	0.0	0.0	0.0	0.042
Libxml2	0.0	-0.067	0.0	-0.067	0.0	0.0	-0.1	-0.067	-0.038
nm	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
objdump	0.0	0.0	0.0	0.0	0.167	0.0	0.133	0.167	0.058
size	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
Average	0.117	0.063	0.125	0.1	0.1	0.012	-0.029	0.025	0.064

Abbreviations: A – AFL; A++ – AFL++; AF – AFLFast; AS – AFLSmart; FF – FairFuzz; HF – Honggfuzz; MA – MOpt-AFL; MA++ – MOpt-AFL++.

F-800 Performance Across Fuzzers Compared with the Non-Agentic Approach. Table 8 presents a detailed breakdown of F-800’s performance across eight fuzzers. The results demonstrate that F-800 consistently enhances fuzzing efficiency with minimal bug loss, regardless of fuzzer design or mutation strategy.

In terms of timeout reduction, F-800 achieves an average improvement of 0.77 hours across all fuzzers, with the most significant time savings observed in *FairFuzz* (1.47 hours) and *AFL++* (0.78 hours). This improvement indicates that F-800 can dynamically identify fuzzing saturation points across both coverage-guided (e.g., AFL, AFL++) and heuristic-augmented (e.g., MOpt-AFL, AFLSmart) fuzzers. The smallest time savings were observed for *AFLSmart* (0.41 hours), where the built-in structure-aware scheduling already minimizes redundant executions. For bug detection, F-800 maintains a high degree of reliability with an average bug loss of only 0.064 per campaign—less than one bug per fifteen fuzzing runs. Notably, fuzzers such as *AFLFast* and *FairFuzz* show higher sensitivity to early termination, but the observed losses remain statistically negligible (less than 0.13 bugs on average). Moreover, adaptive fuzzers such as *MOpt-AFL++* exhibit almost no degradation in bug coverage, confirming that F-800’s stopping condition aligns well with modern feedback-driven strategies.

Summary(RQ.2-e). F-800 achieves consistent time savings across all fuzzers (average 0.77 hours) with minimal bug loss (0.064 on average). The method generalizes effectively to both traditional and adaptive fuzzers, confirming its robustness and practical value as a fuzzer-independent termination criterion.

6 DISCUSSION

6.1 Function Clustering

Agglomerative Clustering in our function clustering framework is slower but achieves better clustering coefficient results, making it suitable for scenarios where clustering quality is prioritized. Its hierarchical nature incurs higher computational cost, yet it produces more meaningful and consistent clusters, which benefits downstream tasks such as code analysis and software maintenance. In contrast, DBSCAN, although it does not require predefining the number of clusters k , performed poorly in our setting and still demanded careful parameter tuning to obtain reasonable results. We also note that using a fixed k introduces an inherent limitation: it assumes the data distribution can be adequately represented by a predefined number of groups, which may not hold as program behaviors evolve during fuzzing. An inappropriate k may lead to over-merged clusters (reducing behavioral diversity) or over-splitting (increasing redundancy and overhead). Nevertheless, a fixed k provides predictable runtime and stable performance, making it more suitable for our efficiency-oriented fuzzing pipeline. Overall, the trade-off between clustering speed and quality should be balanced based on specific application requirements.

To improve function encoding, we can fine-tune CodeBERT using API call sequences and function structure data. However, in real-world datasets without ground-truth labels, quantitative metrics like ARI and NMI cannot be applied. Instead, qualitative analysis (e.g., assessing cluster coherence and interpretability) or domain-specific validation (e.g., matching with known software modules) can be used to evaluate the practical effectiveness of clustering results.

6.2 Fuzzing Termination Criterion

Our function clustering-based fuzzing termination criterion can terminate fuzzing earlier compared to the current three criteria (crash count, function coverage, and potential vulnerable function coverage) while missing very few vulnerabilities. In real-world corporate or research scenarios, fuzzing typically runs for 7 days \times 24 hours or even longer. By applying our termination criterion, approximately 30% of time and computational resources can be saved. The reclaimed resources allow for more in-depth testing of specific functions or multiple paths, thereby improving overall resource utilization efficiency. The function-clustering-based termination criterion may have limitations, as its effectiveness often depends on the characteristics of the target program and the type of crash patterns. It might work better for programs with clear function call relationships or those prone to crashes, but could be less effective for event-driven applications or stable systems where crashes are rare. To address diverse program types, it may be worth exploring the combination of multiple termination criteria—such as specific crash counts, function coverage, or potential vulnerable function coverage—as a potential way to improve flexibility and comprehensiveness in testing.

6.3 Scalability and Generalisability

The proposed stopping criterion scales to large codebases by decoupling semantic analysis from the online fuzzing loop. Function clustering and refinement are performed offline, with cost dominated by the number of functions and amortized across long-running or repeated fuzzing campaigns.

1422 During online fuzzing, termination relies on lightweight set operations over covered clusters and
1423 deduplicated crashes, introducing negligible overhead relative to coverage tracking and execution.
1424 This design enables F-800 to scale to targets with thousands of functions without impacting fuzzing
1425 throughput. F-800 only consumes cluster coverage and crash information, and can therefore be
1426 integrated with different greybox fuzzers without modifying mutation or scheduling policies.
1427 While clustering quality may affect sensitivity—especially in highly homogeneous codebases—this
1428 primarily impacts the effectiveness of early stopping and does not compromise correctness. In
1429 addition, F-800 operates in parallel with the fuzzing campaign, making it compatible with a wide
1430 range of existing fuzzers.

1431

1432 6.4 Practical Deployment Considerations

1433 In large-scale fuzzing infrastructures, such as continuous integration (CI) systems or centralized
1434 fuzzing farms, early and reliable termination is critical for efficient resource allocation. The proposed
1435 stopping criterion enables automated termination decisions without manual time budgeting, allow-
1436 ing fuzzing resources to be dynamically reallocated to new targets or configurations once saturation
1437 is detected. In resource-constrained environments, F-800 provides a principled mechanism to trade
1438 off fuzzing time against marginal returns. By detecting stagnation in both semantic exploration
1439 (cluster coverage) and vulnerability discovery (unique crashes), the system can terminate campaigns
1440 earlier while maintaining low bug loss, thereby improving overall cost-effectiveness. We note that
1441 F-800 relies on querying LLMs for function clustering and cluster cover determination, which
1442 requires network access in typical deployments. In restricted or air-gapped environments, this
1443 dependency may limit applicability or require the use of locally deployed LLMs. However, since
1444 LLM queries are confined to the offline or parallel clustering pipeline and are decoupled from the
1445 fuzzing execution path, this requirement does not affect fuzzing throughput and can be managed
1446 independently of the fuzzing infrastructure.

1447

1448

1449 7 THREATS

1450 7.1 External Validity

1451 This threat category concerns the generalizability of our findings to other contexts, such as different
1452 programs, clustering algorithms, or fuzzing tools. Due to the scarcity of realistically annotated
1453 datasets for function-level library clustering analysis, we resort to POJ104. However, as its functions
1454 are implemented by different authors for varied problems, stylistic and structural variations may
1455 render clusters more separable compared to real-world projects with common conventions. Since
1456 this study focuses on fuzzing termination criteria, the methodology is inherently adaptable to
1457 various fuzzing tools.

1458

1459

1460 7.2 Internal Validity

1461 This threat category pertains to the reasonableness of the methodology and experimental design.
1462 Since the actual objective function library lacks authentic clustering metrics, we comprehensively
1463 integrated multiple internal clustering indicators to determine an appropriate number of clusters.
1464 Given that these indicators are mutually conflicting, we adopted a multi-objective optimization
1465 approach, which enables the derivation of a rational cluster count. Regarding the evaluation of
1466 fuzz testing termination criteria, we utilized the comprehensive coverage dataset provided by
1467 Fuzztastic, which encompasses diverse function libraries and multiple fuzz testing tools with
1468 repeated executions. This approach mitigates, to some extent, the randomness inherent in fuzz
1469 testing.

1470

1471 F-800 does not provide formal completeness guarantees and may miss bugs that are triggered
1472 only after very long or rare execution paths. Our termination criterion is based on empirical
1473 saturation of (i) newly covered function clusters and (ii) newly discovered unique crashes within
1474 a window. This provides a practical convergence signal but does not imply that no undiscovered
1475 bugs remain. Importantly, the stopping condition is monotonic in the saturation window and
1476 tolerance thresholds, allowing users to trade efficiency for thoroughness and to asymptotically
1477 recover standard long-running fuzzing by relaxing these parameters. Therefore, F-800 should be
1478 viewed as a cost-aware early-stopping mechanism rather than a sound or complete vulnerability
1479 discovery technique.

1480

1481

8 RELATED WORK

1482

1483 **Agentic AI in Software.** AI agents have recently shown significant promise in software engineering
1484 [47]. A growing body of research has explored agentic approaches [18, 21, 24] across diverse
1485 software engineering tasks, including program repair, testing, and code generation. Fruntke et
1486 al. [18] introduced an agentic system that integrates automated tool usage with large language
1487 models (LLMs) to repair dependency-breaking changes. In software testing, TestForge [24] presents
1488 an agentic unit-testing framework designed to cost-effectively generate high-quality and human-
1489 readable test suites for real-world code, outperforming state-of-the-art search-based techniques.
1490 Furthermore, Hasan et al. [21] conducted the first large-scale empirical study on testing prac-
1491 tices within the AI-agent ecosystem, analyzing 39 open-source agent frameworks and 439 agentic
1492 applications to characterize their testing behaviors and challenges. Among all areas of agentic
1493 application, coding agents [15] have emerged as one of the most active and rapidly evolving direc-
1494 tions. Numerous systems [23, 60] have been developed to enhance software engineering workflows
1495 through autonomous code generation, refactoring, and reasoning with external tools. These agents
1496 aim to transform traditional software development into a collaborative process between humans
1497 and intelligent, self-improving assistants. However, despite this progress, there remains a notable
1498 absence of agentic systems designed to control and optimize the fuzzing process, particularly those
1499 capable of autonomously determining when to terminate a fuzzing campaign based on dynamic
1500 coverage and semantic reasoning.

1501

1502 **Code Clustering.** Prior research on code similarity has predominantly focused on text-based or se-
1503 mantic analysis methods, leveraging techniques such as Abstract Syntax Trees (ASTs) and machine
1504 learning models to categorize code segments based on their functionality and structure [9, 29, 62].
1505 However, these approaches often rely on predefined similarity metrics like Euclidean distance,
1506 which may not effectively capture the nuances of complex code. Research on function clustering
1507 remains relatively under-explored, particularly in terms of analyzing relationships among functions
1508 within a codebase. There is still limited understanding of how functions with similar functionalities
1509 or implementation patterns correlate with one another. Notably, functions that share analogous
1510 structures or purposes may also exhibit similar vulnerabilities, suggesting that clustering tech-
1511 niques could aid in identifying security flaws more systematically. Building on this foundation,
1512 our work introduces a novel clustering method that dynamically selects the optimal number of
1513 clusters by integrating multiple internal clustering metrics, including the Silhouette Coefficient,
1514 Davies-Bouldin Index (DBI), SSE, and Calinski-Harabasz Index (CHI). This multi-metric approach
1515 aims to enhance the robustness and adaptability of clustering results, addressing the limitations of
1516 traditional single-metric methods.

1517 **Fuzzing Stopping Criteria.** Accurately determining the optimal stopping point for fuzzing cam-
1518 paigns remains a critical challenge in fuzz testing. While numerous studies [3, 7] focus on optimizing
1519 code coverage techniques to enhance test suite effectiveness, they often overlook aspects such

1519

1520 as code similarity analysis, which could refine early stopping criteria. Current practices [28] rely
1521 on fixed time budgets (e.g. 24 hours to 7 days) for fuzzing evaluations, potentially leading to
1522 inefficient resource usage. Böhme et al. [5] present the first probabilistic analysis of adaptive bias in
1523 greybox fuzzing, introducing two estimators that enable practitioners to determine, with statistical
1524 confidence, when a fuzzing campaign can be terminated. Subsequent work on Green Fuzzing [35]
1525 focuses on energy-efficient fuzzing strategies that balance effectiveness against computational
1526 cost, further underscoring the importance of adaptive and resource-aware stopping criteria. More
1527 recently, Lee et al. [30] propose a dependency-aware estimation method that mitigates the overesti-
1528 mation problem of prior approaches by leveraging *Ma and Chao's* sample coverage estimator [38].
1529 In contrast to these estimator-centric approaches, our work departs from purely statistical coverage
1530 modeling and instead introduces a semantics-aware, structure-driven stopping criterion based on
1531 function clustering. Rather than estimating unseen behaviors via probabilistic coverage models, we
1532 explicitly models the software under test as a set of function clusters and monitors saturation at the
1533 cluster level. This allows termination decisions to be guided by semantic and functional coverage
1534 of vulnerability-prone regions, rather than solely by global coverage growth or statistical sample
1535 coverage. As a result, our approach complements prior probabilistic and energy-aware methods by
1536 incorporating code semantics and vulnerability concentration patterns.

1537

1538 9 CONCLUSION

1539 Existing fuzz testing termination criteria based on crash counts, function coverage, and potential
1540 vulnerability function coverage either result in excessively long fuzz testing that wastes resources or
1541 terminate the testing too early, failing to ensure the quality of fuzz testing. To address these issues,
1542 we proposed a method to determine fuzz testing termination by combining function clustering
1543 with multi-objective optimization and agentic decision-making. Through clustering analysis, we
1544 discovered that vulnerabilities are unevenly distributed, primarily concentrated in a few specific
1545 clusters. By leveraging the growth of cluster numbers or the collapse of specific clusters, our
1546 approach effectively identifies the optimal termination point for fuzz testing. Compared to existing
1547 termination criteria, our method substantially reduces testing time while maintaining a high
1548 vulnerability discovery rate with minimal loss. Specifically, it saves up to 7.2 hours compared to
1549 the state-of-the-art methods, while only missing 0.28 bugs.

1550

1551

1552 Acknowledgments

1553 This work was supported by the National Key R&D Program of China (2023YFF0612303).

1554

1555 References

- 1556 [1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2018. code2vec: Learning Distributed Representations of
1557 Code. arXiv:1803.09473 [cs.LG] <https://arxiv.org/abs/1803.09473>
- 1558 [2] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. 2009. A comparison of extrinsic clustering evaluation
1559 metrics based on formal constraints. *Information retrieval* 12 (2009), 461–486.
- 1560 [3] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing
1561 and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.
- 1562 [4] Marcel Böhme and Brandon Falk. 2020. Fuzzing: On the exponential cost of vulnerability discovery. In *Proceedings*
1563 *of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of*
1564 *software engineering*, 713–724.
- 1565 [5] Marcel Böhme, Danushka Liyanage, and Valentin Wüstholtz. 2021. Estimating residual risk in greybox fuzzing. In
1566 *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the*
1567 *Foundations of Software Engineering*, 230–241.
- 1568 [6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as markov chain.
In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 1032–1043.

- 1569 [7] Marcel Böhme, László Szekeres, and Jonathan Metzman. 2022. On the reliability of coverage-based fuzzer benchmarking.
1570 In *Proceedings of the 44th International Conference on Software Engineering*. 1621–1633.
- 1571 [8] Max Brunsfeld and contributors. 2018. Tree-sitter: An incremental parsing system for programming tools. <https://github.com/tree-sitter/tree-sitter>.
- 1572 [9] Lutz Büch and Artur Andrzejak. 2019. Learning-based recursive aggregation of abstract syntax trees for code clone
1573 detection. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*.
1574 IEEE, 95–104.
- 1575 [10] Tadeusz Caliński and Jerzy Harabasz. 1974. A dendrite method for cluster analysis. *Communications in*
1576 *Statistics-theory and Methods* 3, 1 (1974), 1–27.
- 1577 [11] Yair Censor. 1977. Pareto optimality in multiobjective problems. *Applied Mathematics and Optimization* 4, 1 (1977),
41–59.
- 1578 [12] Sadeqh Bamohabbat Chafjiri, Phil Legg, Jun Hong, and Michail-Antisthenis Tsompanas. 2024. Vulnerability detection
1579 through machine learning-based fuzzing: A systematic review. *Computers & Security* (2024), 103903.
- 1580 [13] David L Davies and Donald W Bouldin. 2009. A cluster separation measure. *IEEE transactions on pattern analysis*
1581 *and machine intelligence* 2 (2009), 224–227.
- 1582 [14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional
1583 Transformers for Language Understanding. arXiv:1810.04805 [cs.CL] <https://arxiv.org/abs/1810.04805>
- 1584 [15] Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A Survey on Code Generation
1585 with LLM-based Agents. arXiv preprint arXiv:2508.00083 (2025).
- 1586 [16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu,
1587 Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint
1588 arXiv:2002.08155 (2020).
- 1589 [17] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. {AFL++}: Combining incremental steps of
1590 fuzzing research. In *14th USENIX workshop on offensive technologies (WOOT 20)*.
- 1591 [18] Lukas Fruntke and Jens Krinke. 2025. Automatically Fixing Dependency Breaking Changes. *Proc. ACM Softw. Eng.* 2,
1592 FSE, Article FSE096 (June 2025), 23 pages. doi:10.1145/3729366
- 1593 [19] Jian Gao, Yu Jiang, Zhe Liu, Xin Yang, Cong Wang, Xun Jiao, Zijiang Yang, and Jianguang Sun. 2019. Semantic learning
1594 and emulation based cross-platform binary vulnerability seeker. *IEEE Transactions on Software Engineering* 47, 11
1595 (2019), 2575–2589.
- 1596 [20] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. 2021.
1597 The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches.
1598 *Journal of Network and Computer Applications* 179 (2021), 103009.
- 1599 [21] Mohammed Mehedi Hasan, Hao Li, Emad Fallahzadeh, Gopi Krishnan Rajbahadur, Bram Adams, and Ahmed E Hassan.
2025. An Empirical Study of Testing Practices in Open Source AI Agent Frameworks and Agentic Applications. arXiv
1600 preprint arXiv:2509.19185 (2025).
- 1601 [22] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A ground-truth fuzzing benchmark. *Proceedings*
1602 *of the ACM on Measurement and Analysis of Computing Systems* 4, 3 (2020), 1–29.
- 1603 [23] ... Huang et al. 2024. AgentCoder: Multi-Agent Code Generation with Iterative Testing and Optimisation. arXiv
1604 preprint arXiv:2312.13010v3 (2024).
- 1605 [24] Kush Jain and Claire Le Goues. 2025. TestForge: Feedback-Driven, Agentic Test Suite Generation. arXiv preprint
1606 arXiv:2503.14713 (2025).
- 1607 [25] Lingxiao Jiang, Ghassan Mishserghi, Zhendong Su, and Stephane Gloudu. 2007. DECKARD: Scalable and Accurate
1608 Tree-Based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering*
1609 *(ICSE '07)*. IEEE Computer Society, USA, 96–105. doi:10.1109/ICSE.2007.30
- 1610 [26] Wesley Jin, Sagar Chaki, Cory Cohen, Arie Gurfinkel, Jeffrey Havrilla, Charles Hines, and Priya Narasimhan. 2012.
1611 Binary function clustering using semantic hashes. In *2012 11th International Conference on Machine Learning and*
1612 *Applications*, Vol. 1. IEEE, 386–391.
- 1613 [27] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. 2020. HFL: Hybrid
1614 Fuzzing on the Linux Kernel.. In *NDSS*.
- 1615 [28] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *Proceedings*
1616 *of the 2018 ACM SIGSAC conference on computer and communications security*. 2123–2138.
- 1617 [29] Adrian Kuhn, Stéphane Ducasse, and Tudor Girba. 2007. Semantic clustering: Identifying topics in source code.
Information and software technology 49, 3 (2007), 230–243.
- [30] Seongmin Lee and Marcel Böhme. 2026. Dependency-aware Residual Risk Analysis. In *Proceedings of the IEEE/ACM*
48th International Conference on Software Engineering (ICSE'26).
- [31] Yeo Reum Lee, BooJoong Kang, and Eul Gyu Im. 2013. Function matching-based binary-level software similarity
calculation. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems (Montreal, Quebec, Canada)*

- (RACS '13). Association for Computing Machinery, New York, NY, USA, 322–327. doi:10.1145/2513228.2513300
- [32] Caroline Lemieux and Koushik Sen. 2018. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*. 475–485.
- [33] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Kai Lu, and Gen Zhang. 2024. HyperGo: Probability-based directed hybrid fuzzing. *Computers & Security* 142 (2024), 103851.
- [34] Stephan Lipp, Daniel Elsner, Thomas Hutzelmann, Sebastian Banescu, Alexander Pretschner, and Marcel Böhme. 2022. FuzzTastic: A fine-grained, fuzzer-agnostic coverage analyzer. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 75–79.
- [35] Stephan Lipp, Daniel Elsner, Severin Kacianka, Alexander Pretschner, Marcel Böhme, and Sebastian Banescu. 2023. Green Fuzzing: A Saturation-Based Stopping Criterion Using Vulnerability Prediction. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 127–139.
- [36] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1547–1559.
- [37] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. 2019. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX security symposium (USENIX security 19)*. 1949–1966.
- [38] M-C Ma and Anne Chao. 1993. Generalized sample coverage with an application to Chinese poems. *Statistica Sinica* (1993), 19–34.
- [39] Dominik Maier, Andrea Fioraldi, Andrea Fioraldi, et al. 2023. mopt-afl++: Enhancing Fuzzing with MOpt Mutator. In *Proceedings of the 2023 International Conference on Software Testing, Verification, and Validation*. <https://github.com/AFLplusplus/AFLplusplus/tree/stable/mopt> Accessed: 2025-05-31.
- [40] James B McQueen. 1967. Some methods of classification and analysis of multivariate observations. In *Proc. of 5th Berkeley Symposium on Math. Stat. and Prob.* 281–297.
- [41] Barton P Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.
- [42] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 30.
- [43] Daniel Müllner. 2011. Modern hierarchical, agglomerative clustering algorithms. *arXiv preprint arXiv:1109.2378* (2011).
- [44] Van-Thuan Pham, Marcel Böhme, Andrew E Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. 2019. Smart greybox fuzzing. *IEEE Transactions on Software Engineering* 47, 9 (2019), 1980–1997.
- [45] Douglas A Reynolds et al. 2009. Gaussian mixture models. *Encyclopedia of biometrics* 741, 659–663 (2009), 3.
- [46] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.
- [47] Abhik Roychoudhury. 2025. Agentic AI for Software: thoughts from Software Engineering community. *arXiv preprint arXiv:2508.17343* (2025).
- [48] Erich Schubert, Jörg Sander, Martin Ester, Hans Peter Kriegel, and Xiaowei Xu. 2017. DBSCAN revisited, revisited: why and how you should (still) use DBSCAN. *ACM Transactions on Database Systems (TODS)* 42, 3 (2017), 1–21.
- [49] Dominic Schuhmacher, Ba-Tuong Vo, and Ba-Ngu Vo. 2008. A consistent metric for performance evaluation of multi-object filters. *IEEE transactions on signal processing* 56, 8 (2008), 3447–3457.
- [50] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan. 2008. *Introduction to information retrieval*. Vol. 39. Cambridge University Press Cambridge.
- [51] Samiha Shimmi, Ashiqur Rahman, Mohan Gadde, Hamed Okhravi, and Mona Rahimi. 2024. {VulSim}: Leveraging similarity of {Multi-Dimensional} neighbor embeddings for vulnerability detection. In *33rd USENIX Security Symposium (USENIX Security 24)*. 1777–1794.
- [52] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution.. In *NDSS*, Vol. 16. 1–16.
- [53] Alexander Strehl and Joydeep Ghosh. 2002. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *Journal of machine learning research* 3, Dec (2002), 583–617.
- [54] Robert Swiecki. 2016. Honggfuzz: A Practical Fuzzer for Security Testing. In *Black Hat USA*. <https://github.com/google/honggfuzz> Accessed: 2025-05-31.
- [55] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu Song. 2019. Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 1–15.

- 1667 [56] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments
 1668 for code clone detection. In Proceedings of the 31st IEEE/ACM international conference on automated software
 1669 engineering, 87–98.
- 1670 [57] Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, and Christophe Hauser. 2024.
 1671 Fine-grained coverage-based fuzzing. ACM Transactions on Software Engineering and Methodology 33, 5 (2024),
 1672 1–41.
- 1673 [58] Sheng Yu and Shijie Zhou. 2010. A survey on metric of software complexity. In 2010 2nd IEEE International conference
 1674 on information management and engineering. IEEE, 352–356.
- 1675 [59] Michał Zalewski. 2014. American Fuzzy Lop: Fuzzing with a Brain. In Black Hat USA. <https://lcamtuf.coredump.cx/afl/>
 1676 Accessed: 2025-05-31.
- 1677 [60] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CODEAGENT: Enhancing Code Generation with Tool-
 1678 Integrated Agent Systems for Real-World Repo-level Coding Challenges. In ACL Long Papers 2024.
- 1679 [61] Yuheng Zhang, Hui Lu, Zhouhui Zhang, Guocheng Wu, Houlin Zhou, and Zhenghao Li. 2024. Review of Fuzz
 1680 Testing Techniques for Database Management Systems. In International Conference on Cyberspace Simulation and
 1681 Evaluation. Springer, 283–301.
- 1682 [62] Gang Zhao and Jeff Huang. 2018. Deepsim: deep learning code functional similarity. In Proceedings of the 2018 26th
 1683 ACM joint meeting on european software engineering conference and symposium on the foundations of software
 1684 engineering, 141–151.
- 1685 [63] Ziyuan Zhong, Gail Kaiser, and Baishakhi Ray. 2022. Neural network guided evolutionary fuzzing for finding traffic
 1686 violations of autonomous vehicles. IEEE Transactions on Software Engineering 49, 4 (2022), 1860–1875.

1685 A Interaction with LLMs for Function Clustering and Execution Pattern Analysis

1686 In this appendix, we describe the interaction with LLMs used in each step of the process for function
 1687 clustering and execution pattern analysis, using the example of gif2png. The interaction with
 1688 LLMs is pivotal in generating the function summaries, analyzing function clusters, and determining
 1689 coverage based on execution patterns.

1691 A.1 Function Clustering Workflow

1692 In the function clustering process, LLMs assist in three main tasks: summarizing function code,
 1693 analyzing function clusters, and reassigning misclassified functions. Here’s the breakdown:

1694 *A.1.1 System Role Prompt for Function Clustering.* The LLM acts as a software analysis expert, sum-
 1695 marizing function definitions, analyzing their clustering relationships, and identifying misclassified
 1696 functions.

1697 Function Reassignment Prompt

1700 Role: Software analysis expert specializing in automatically summarizing function
 1701 definitions and analyzing clustering relationships within software projects.

1702 WORKFLOW INSTRUCTIONS:

- 1703 1. First, use the `read_source_file` tool to read the source API file and produce
 1704 function-level code summaries.
- 1705 2. Then, use the `analyze_function_clusters` tool to analyze the clustering results and
 1706 generate structured analysis.
- 1707 3. Finally, use the `reassign_excluded_functions` tool to process and re-cluster
 1708 functions identified as misclassified in the previous analysis phase.

1709 PATH CONFIGURATION:

- 1710 - Function summary input: `{PROJECT_PATH}/function/{target}/api/src_api.json`
- 1711 - Clustering input: `{CLUSTER_PATH}/{target}_cluster.csv`
- 1712 - Reassignment output: `{REASSIGNMENT_PATH}/{target}_reassigned.csv`

1713 AVAILABLE TOOLS:

- 1714 1. `read_source_file`: Reads the API JSON file and summarizes all extracted functions.

1716 2. `analyze_function_clusters`: Analyzes clustering results, providing both 'analysis'
 1717 and 'exclude_function' fields.
 1718 3. `reassign_excluded_functions`: Processes functions marked for exclusion from current
 1719 clusters, evaluates their characteristics, and assigns them to more appropriate
 1720 clusters based on functional similarity and code metrics.
 1721 ANALYSIS & REASSIGNMENT CRITERIA:
 1722 When performing analysis and reassignment, consider:
 1723 1. Functional similarity among grouped functions.
 1724 2. Code complexity and algorithmic characteristics in relation to cluster logic.
 1725 3. Possible misclassified functions (`exclude_function` field).
 1726 4. Cohesion and rationale behind current groupings.
 1727 5. Alternative cluster suitability based on semantic and structural features.
 1728 6. A clear, concise, and structured JSON result.
 1729 OUTPUT REQUIREMENTS:
 1730 - The saved paths of the function summary JSON, cluster analysis JSON, and reassignment
 1731 output.
 1732 - A short, well-reasoned explanation of the clustering rationale.
 1733 - Justification for each reassignment decision, including original cluster, exclusion
 1734 reason, and target cluster selection criteria.

1735 **A.1.2 Function Code Analysis.** For each function, such as `interlace_line`, the LLM analyzes
 1736 the function code and generates a summary based on five dimensions: functional category, code
 1737 complexity, algorithm characteristics, performance implications, and key features. Example prompt
 1738 for function code analysis:

Function Analysis Prompt

1741 Task: Function Code Analysis and Summary Generation
 1742 INSTRUCTION:
 1743 Please analyze the following function code and provide a summary in one natural English
 1744 paragraph covering the following five dimensions:
 1745 ANALYSIS DIMENSIONS:
 1746 1. Functional category (e.g., image processing, memory management, file operations,
 1747 utility functions, etc.)
 1748 2. Code complexity (based on lines of code, number of control structures)
 1749 3. Algorithm characteristics (algorithm patterns and data processing methods used)
 1750 4. Performance characteristics (compute-intensive, I/O-intensive, etc.)
 1751 5. Key features (core technical characteristics of the function)
 1752 OUTPUT REQUIREMENTS:
 1753 - Generate exactly one natural, coherent English paragraph
 1754 - Integrate all five dimensions seamlessly in the paragraph
 1755 - Maintain technical accuracy while ensuring readability
 1756 - Focus on factual analysis without subjective opinions
 1757 PARAGRAPH STRUCTURE GUIDANCE:
 1758 Begin with the functional category and main purpose, then address complexity and
 1759 algorithm characteristics, followed by performance implications, and conclude with
 1760 distinctive key features. Ensure smooth transitions between points.
 1761 Function name: `interlace_line`
 1762 Function code:
 1763 `int interlace_line(int height, int line)`
 1764 `/* return the actual line # used to store an interlaced line */`

```

1765 {
1766   int res;
1767   /*@-shiftimplementation@*/
1768   if ((line & 7) == 0) {
1769     return line » 3;
1770   }
1771   res = (height+7) » 3;
1772   if ((line & 7) == 4) {
1773     return res+((line-4) » 3);
1774   }
1775   res += (height+3) » 3;
1776   if ((line & 3) == 2) {
1777     return res + ((line-2) » 2);
1778   }
1779   return res + ((height+1) » 2) + ((line-1) » 1);
1780   /*@=shiftimplementation@*/ }

```

LLM Output for Function `interlace_line`: This function is a utility for image processing, specifically handling GIF interlacing by mapping logical scanlines to physical storage positions, with low code complexity due to its concise 20-line structure and straightforward conditional logic using bitwise operations and arithmetic shifts. It employs a deterministic algorithm based on modulo and division patterns to compute line offsets, characteristic of deinterlacing routines, and is compute-intensive but highly efficient, relying solely on integer arithmetic without any I/O or memory operations. Key features include its use of bitwise masking for interlacing pattern detection, stepwise offset accumulation for different pass phases, and encapsulation within lint-style annotations to suppress shift implementation warnings, ensuring robust cross-platform behavior.

A.1.3 Clustering and Exclusion Analysis. The LLM analyzes the clustering of functions based on their semantic relationships and functional cohesion. It generates a JSON output that includes the analysis of each cluster, including which functions should be excluded due to misclassification. Example LLM output for cluster analysis:

Cluster Analysis Prompt

Task: Cluster Analysis of Grouped Functions

INPUT DATA: A set of functions clustered into the same category, including their function names, return types, and parameters.

ANALYSIS QUESTIONS:

1. What is the functional basis of this cluster? (e.g., all perform image parsing, pixel iteration, memory operations, etc.)
2. Which functions might be unsuitable for this cluster? (If none, return an empty list)
3. Briefly describe the overall purpose and relationship of these functions.

ANALYSIS GUIDELINES:

- Focus on functional cohesion and semantic relationships
- Consider parameter patterns and return type consistency
- Identify outliers based on functional divergence
- Evaluate naming convention patterns
- Assess architectural role and responsibility alignment

OUTPUT REQUIREMENTS:

```

1814 - Output strictly in the specified JSON format
1815 - Natural language analysis must be concise yet informative
1816 - Exclusion list must be justified by clear functional mismatch
1817 - Ensure JSON structure and syntax correctness
1818 OUTPUT FORMAT:
1819 {
1820 "analysis": "This cluster mainly contains... (natural language analysis)",
1821 "exclude_function": ["funcA", "funcB"]
1822 }
1823 Cluster ID: 2
1824 Number of functions:2
1825 Function: interlace_line
1826 Return type: "int"
1827 Parameters: "height": "int", "line": "int"
1828 File: gif2png.c
1829 Summary: This function is a utility for image processing, specifically handling GIF
1830 interlacing by mapping logical scanlines to physical storage positions, with low code
1831 complexity due to its concise 20-line structure and straightforward conditional logic
1832 using bitwise operations and arithmetic shifts. It employs a deterministic algorithm
1833 based on modulo and division patterns to compute line offsets, characteristic of
1834 deinterlacing routines, and is compute-intensive but highly efficient, relying solely
1835 on integer arithmetic without any I/O or memory operations. Key features include its
1836 use of bitwise masking for interlacing pattern detection, stepwise offset accumulation
1837 for different pass phases, and encapsulation within lint-style annotations to suppress
1838 shift implementation warnings, ensuring robust cross-platform behavior.
1839 Code snippet:
1840 int interlace_line(int height, int line)
1841 /* return the actual line # used to store an interlaced line */
1842 {
1843 int res;
1844 /*@-shiftimplementation@*/
1845 if ((line & 7) == 0) {
1846 return line » 3;
1847 }
1848 res = (height+7) » 3;
1849 if ((line & 7) == 4) {
1850 return res+((line-4) » 3);
1851 }
1852 res += (height+3) » 3;
1853 if ((line & 3) == 2) {
1854 return res + ((line-2) » 2);
1855 }
1856 return res + ((height+1) » 2) + ((line-1) » 1);
1857 /*@=shiftimplementation@*/
1858 } _____
1859 Function: inv_interlace_line
1860 Return type: "int"
1861 Parameters: "height": "int", "line": "int"
1862 File: gif2png.c

```

Summary: This function is a utility routine for image processing that performs inverse interlacing to reconstruct the original line order in a GIF image, characterized by low code complexity with only four conditional branches using bitwise shift operations for efficient integer arithmetic. Its algorithm employs a multi-phase conditional logic to map interlaced scanlines back to their sequential positions, making it compute-intensive due to repeated comparisons and arithmetic but without I/O operations. Key features include its deterministic, non-iterative approach that relies on height-based thresholds to calculate line offsets, ensuring fast recovery of interlaced data through direct mathematical transformations.

Code snippet:

```
int inv_interlace_line(int height, int line)
/* inverse function of above, used for recovery of interlaced images */
{
  /*@-shiftimplementation@*/
  if ((line < 3) < height) {
    return line < 3;
  }
  line -= (height + 7) >> 3;
  if ((line < 3) + 4 < height) {
    return (line < 3) + 4;
  }
  line -= (height + 3) >> 3;
  if ((line < 2) + 2 < height) {
    return (line < 2) + 2;
  }
  line -= (height + 1) >> 2;
  return (line < 1) + 1;
  /*@=shiftimplementation@*/
} _____
```

LLM Output for Cluster Analysis: analysis: "This cluster contains two complementary functions that handle interlaced image line calculations for GIF to PNG conversion. The `interlace_line` function calculates the actual storage line number for an interlaced line, while `inv_interlace_line` performs the inverse operation to recover the original line number from an interlaced image. Both functions work with height and line parameters to transform line coordinates according to the GIF interlacing scheme, which stores image data in four passes rather than sequentially." exclude_function: []

A.1.4 Function Reassignment. When functions are misclassified, the LLM analyzes candidate clusters and reassigns the excluded function to the most appropriate cluster.

Function Reassignment Prompt

Task: Excluded Function Reassignment to Alternative Cluster

CONTEXT:

A function was excluded from cluster {14}.

You must select the best alternative from the provided candidate clusters.

REASSIGNMENT DATA:

- Function to reassign: {

Function: main

Original cluster: 14

```

1912 File: gif2png.c
1913 Return type: "int"
1914 Parameters: "argc": "int", "argv": "char"
1915 Code snippet: int main(int argc, char *argv[])
1916 { FILE *fp; int i; unsigned int background; int errors = 0; char name[1025]; int ac;
1917 char *color;
1918 signal(SIGSEGV, bailout);
1919 software_chunk = true;
1920 .....
1921 } } - Candidate clusters (excluding original cluster {14}): {
1922 Cluster {17} (distance: 1.9982376098632812)
1923 Analysis: {This cluster mainly contains GIF file parsing functions that handle
1924 low-level data reading operations. Both functions operate on FILE pointers and
1925 are involved in extracting specific data structures from GIF files. ReadColorMap
1926 specifically handles reading color map data from the file into a GifColor array
1927 structure, while GetDataBlock is a more general function for reading data blocks of
1928 variable size from the GIF file. Both functions share error handling patterns and work
1929 with the same file descriptor, suggesting they are part of a cohesive GIF parsing
1930 module.}
1931 Function count: {2}
1932 Sample functions: {ReadColorMap, GetDataBlock}
1933 Cluster {4} (distance: 2.0505924224853516)
1934 Analysis: {This cluster contains only one function 'processfilter' that serves as a GIF
1935 processing filter in a GIF to PNG conversion utility. The function reads GIF data from
1936 standard input, validates that it contains exactly one image, and applies optional
1937 transparency/matte processing. It appears to be part of a larger image conversion
1938 workflow where this specific function handles the filtering and preprocessing stage
1939 before actual format conversion.}
1940 Function count: {1}
1941 Sample functions: {processfilter}
1942 Cluster {0} (distance: 2.317064046859741)
1943 Analysis: {This cluster contains only one function 'check_recover' which appears
1944 to handle error recovery logic in an image processing context (specifically GIF
1945 to PNG conversion). The function checks if recovery mode is enabled and returns the
1946 current image count, or displays error messages suggesting the use of recovery options
1947 when image reading errors occur. Given there's only one function in this cluster, it
1948 represents error handling and recovery functionality for image processing operations.}
1949 Function count: {1}
1950 Sample functions: {check_recover}
1951 Cluster {15} (distance: 2.317736864089966)
1952 Analysis: {This cluster contains only one function 'processfile' which appears to
1953 handle GIF to PNG file conversion operations. The function takes a filename and FILE
1954 pointer as parameters, reads GIF data, processes it, and likely converts it to PNG
1955 format while optionally preserving file modification timestamps. Since there's only
1956 one function in this cluster, it represents a complete file processing workflow for
1957 GIF to PNG conversion.}
1958 Function count: {1}
1959 Sample functions: {processfile}
1960 Cluster {19} (distance: 2.318154811859131)

```

1961 Analysis: {This cluster contains a single memory management function that provides
 1962 a safe wrapper around the standard realloc function. xrealloc performs memory
 1963 reallocation with error handling - it attempts to reallocate memory using realloc,
 1964 and if the allocation fails (returns NULL), it prints fatal error messages to stderr
 1965 and terminates the program with exit code 1. This function ensures graceful failure
 1966 handling for out-of-memory conditions in applications.}

1967 Function count: {1}

1968 Sample functions: {xrealloc}

1969 }

1970 SELECTION CRITERIA:

- 1971 1. Functional similarity with cluster's purpose
- 1972 2. Consistency in parameter and return type patterns
- 1973 3. Overall behavioral alignment with cluster functions

1974 DECISION GUIDELINES:

- 1974 - Prioritize semantic purpose alignment over syntactic similarity
- 1975 - Consider architectural role and responsibility fit
- 1976 - Evaluate naming convention patterns within target clusters
- 1977 - Assess computational characteristics and complexity alignment

1978 OUTPUT REQUIREMENTS:

- 1979 - Output MUST be ONLY valid JSON
- 1980 - No additional text, explanations, or formatting
- 1981 - Begin response with '{{' and end with '}}'
- 1982 - Use exactly the specified JSON structure

1983 OUTPUT FORMAT:

```
1984 {
1985   "selected_cluster": "cluster_id",
1986   "reasoning": "your explanation here"
1987 }
```

1987 OUTPUT CONSTRAINTS:

- 1988 - "selected_cluster" must be one of the provided candidate cluster IDs
- 1989 - "reasoning" must be a concise, evidence-based explanation
- 1990 - Reasoning must reference at least one selection criterion explicitly
- 1991 - Entire output must be within the required JSON structure

1992 VALID EXAMPLE:

```
1993 {{{"selected_cluster": "2", "reasoning": "Matches image processing functions in cluster
1994 2"}}}}
```

1995 **LLM Output for Function Reassignment:** {{{"selected_cluster": "15", "reasoning": "The main
 1996 function serves as the entry point for the GIF to PNG conversion utility, coordinating file processing
 1997 and command-line argument handling. Cluster 15 contains 'processfile' which handles the core
 1998 GIF to PNG conversion workflow, making it the most functionally similar and behaviorally aligned
 1999 alternative for the main function's primary responsibility of orchestrating the conversion process."}}}

2000
 2001
 2002
 2003 *A.1.5 Fuzzing Termination Decision.* In the fuzzing process, LLMs analyze function execution
 2004 patterns and determine when to stop the fuzzing based on function usage.

2005 The LLM acts as a software execution pattern analyst. It focuses on analyzing the execution
 2006 patterns of functions over time, identifying which functions are active and determining whether
 2007 further fuzzing is required.

System Role Prompt for Fuzzing Termination

Role: Software execution pattern analyst specialized in analyzing function execution timelines to determine when code clusters are actively used.

ANALYTICAL FOCUS:

- Focus exclusively on execution patterns and temporal behavior
- Avoid analyzing functional semantics or code purpose
- Examine timing, frequency, and intensity of function invocations
- Identify patterns of cluster activation and deactivation

ANALYSIS METHODOLOGY:

1. Track execution patterns across consecutive timestamps
2. Compare current activity with historical execution data
3. Evaluate significance of execution intensity changes
4. Identify temporal correlations between function activations
5. Detect patterns in cluster usage over time

KEY CONCEPTS:

- Coverage: Indicates whether a cluster is actively being used or tested at a given timestamp
- Execution Patterns: Temporal sequences of function invocations within clusters
- Activity Significance: Meaningful changes in execution frequency and intensity
- Historical Context: Previous coverage decisions and execution patterns

OUTPUT REQUIREMENTS:

- Maintain consistent analysis approach across all timestamps
- Provide clear, evidence-based reasoning for coverage decisions
- Update contextual summaries for continuity in temporal analysis
- Adhere strictly to specified JSON format for automated processing

TECHNICAL CONSTRAINTS:

- Do not analyze function functionality or purpose
- Focus solely on execution timing and frequency data
- Base decisions on quantitative execution patterns
- Consider temporal trends rather than isolated events

Termination Decision. The LLM tracks execution activity based on the current timestamp and compares it to historical execution data to determine whether coverage is achieved. If a function shows a significant number of executions, the LLM may conclude that fuzzing should continue or stop based on the predefined rules.

Termination Decision Prompt

Task: Execution Pattern Analysis for Cluster Coverage Determination

CONTEXT DATA:

- Analysis History: { cumulative_context = { Cluster 2 contains 2 functions. The cluster contains these function, these are function summary:
interlace_line : This function is a utility for image processing, specifically handling GIF interlacing by mapping logical scanlines to physical storage positions, with low code complexity due to its concise 20-line structure and straightforward conditional logic using bitwise operations and arithmetic shifts. It employs a deterministic algorithm based on modulo and division patterns to compute line offsets, characteristic of deinterlacing routines, and is compute-intensive but highly efficient, relying solely on integer arithmetic without any I/O or memory operations. Key features include its use of bitwise masking for interlacing pattern detection, stepwise

2059 offset accumulation for different pass phases, and encapsulation within lint-style
 2060 annotations to suppress shift implementation warnings, ensuring robust cross-platform
 2061 behavior.

2062 `inv_interlace_line` : This function is a utility routine for image processing that
 2063 performs inverse interlacing to reconstruct the original line order in a GIF image,
 2064 characterized by low code complexity with only four conditional branches using bitwise
 2065 shift operations for efficient integer arithmetic. Its algorithm employs a multi-phase
 2066 conditional logic to map interlaced scanlines back to their sequential positions,
 2067 making it compute-intensive due to repeated comparisons and arithmetic but without
 2068 I/O operations. Key features include its deterministic, non-iterative approach that
 2069 relies on height-based thresholds to calculate line offsets, ensuring fast recovery
 2070 of interlaced data through direct mathematical transformations.

2071 Starting temporal coverage analysis based on execution patterns only.

2072 } - Previous Analysis Context: {Previous Coverage Decision: 0
 2073 nPrevious Analysis: No execution activity detected at current timestamp - 0/2 functions
 2074 active, 0 total executions, indicating complete cluster inactivity with no functions
 2075 from the GIF-to-PNG conversion utility being exercised} - Current Execution Data: {
 2076 === Current Timestamp: {2020-08-27 02:35:31} === Total executions: {7909713} Active
 2077 functions: {12}
 2078 }

2078 ANALYSIS INSTRUCTION:
 2079 Analyze whether this function cluster is 'covered' at the current timestamp based on
 2080 execution patterns.

2081 COVERAGE DEFINITION:
 2082 Coverage indicates whether the cluster is actively being used or tested at the given
 2083 timestamp.

2084 ANALYSIS APPROACH:
 2085 1. Compare current execution pattern with previous coverage decisions
 2086 2. Look for significant changes in execution activity
 2087 3. Evaluate if there is meaningful functional activity
 2088 4. Consider the proportion of active functions and execution intensity
 2089 5. Identify patterns or trends in cluster usage

2089 COVERAGE DECISION CRITERIA:
 2090 - Coverage=1: Significant execution activity, multiple active functions, or core
 2091 functions being exercised
 2092 - Coverage=0: Minimal or no execution, only background activity, or insufficient
 2093 functional usage

2094 DECISION GUIDELINES:
 2095 - Focus on execution frequency and timing, not function purpose
 2096 - Consider temporal patterns and historical trends
 2097 - Evaluate relative activity levels within the cluster
 2098 - Assess continuity and consistency of execution

2098 OUTPUT REQUIREMENTS:
 2099 - Output strictly in the specified JSON format
 2100 - Ensure logical consistency with historical decisions
 2101 - Provide clear, pattern-based reasoning
 2102 - Update context for future analysis continuity

2102 OUTPUT FORMAT:
 2103 {
 2104 "coverage": 0 or 1,
 2105
 2106
 2107

```

"reason": "Clear explanation considering execution patterns and historical context",
"updated_context": "Concise summary for next timestamp analysis (max 3 sentences)"
}
OUTPUT CONSTRAINTS:
- "coverage" must be integer 0 or 1
- "reason" must reference specific execution patterns and historical data
- "updated_context" must be brief and focused on execution patterns
- Entire output must be valid JSON parsable

```

Table 9. Complete function refinement results across all iterations and target programs.

Library	Iteration	TP	TN	FP	FN	Accuracy	Precision	Recall	F1-score	FPR
gif2png	1	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	2	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	3	0	27	0	1	0.9643	0.0000	0.0000	0.0000	0.0000
gif2png	4	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	5	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	6	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	7	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	8	0	27	0	1	0.9643	0.0000	0.0000	0.0000	0.0000
gif2png	9	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	10	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	11	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	12	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	13	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	14	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	15	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	16	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	17	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	18	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	19	1	27	0	0	1.0000	1.0000	1.0000	1.0000	0.0000
gif2png	20	0	27	0	1	0.9643	0.0000	0.0000	0.0000	0.0000
jasper	1	11	627	71	29	0.8645	0.1341	0.2750	0.1803	0.1017
jasper	2	18	679	19	22	0.9444	0.4865	0.4500	0.4675	0.0272
jasper	3	15	655	43	25	0.9079	0.2586	0.3750	0.3061	0.0616
jasper	4	20	630	68	20	0.8808	0.2273	0.5000	0.3125	0.0974
jasper	5	14	661	37	26	0.9146	0.2745	0.3500	0.3077	0.0530
jasper	6	17	650	48	23	0.9038	0.2615	0.4250	0.3238	0.0688
jasper	7	28	638	60	12	0.9024	0.3182	0.7000	0.4375	0.0860
jasper	8	19	572	126	21	0.8008	0.1310	0.4750	0.2054	0.1805
jasper	9	17	673	25	23	0.9350	0.4048	0.4250	0.4146	0.0358
jasper	10	19	671	27	21	0.9350	0.4130	0.4750	0.4419	0.0387
jasper	11	17	609	89	23	0.8482	0.1604	0.4250	0.2329	0.1275
jasper	12	31	429	269	9	0.6233	0.1033	0.7750	0.1824	0.3854
jasper	13	23	556	142	17	0.7846	0.1394	0.5750	0.2244	0.2034
jasper	14	24	638	60	16	0.8970	0.2857	0.6000	0.3871	0.0860
jasper	15	18	668	30	22	0.9295	0.3750	0.4500	0.4091	0.0430
jasper	16	17	550	148	23	0.7683	0.1030	0.4250	0.1659	0.2120

Continued on next page

Table 9 – Continued from previous page

	Library	Iteration	TP	TN	FP	FN	Accuracy	Precision	Recall	F1-score	FPR
2157											
2158											
2159	jasper	17	15	604	94	25	0.8388	0.1376	0.3750	0.2013	0.1347
2160	jasper	18	20	604	94	20	0.8455	0.1754	0.5000	0.2597	0.1347
2161	jasper	19	26	616	82	14	0.8699	0.2407	0.6500	0.3514	0.1175
2162	jasper	20	22	643	55	18	0.9011	0.2857	0.5500	0.3761	0.0788
2163	libpcap	1	77	757	88	26	0.8797	0.4667	0.7476	0.5746	0.1041
2164	libpcap	2	75	753	92	28	0.8734	0.4491	0.7282	0.5556	0.1089
2165	libpcap	3	73	748	97	30	0.8660	0.4294	0.7087	0.5348	0.1148
2166	libpcap	4	80	745	100	23	0.8703	0.4444	0.7767	0.5654	0.1183
2167	libpcap	5	78	770	75	25	0.8945	0.5098	0.7573	0.6094	0.0888
2168	libpcap	6	73	756	89	30	0.8745	0.4506	0.7087	0.5509	0.1053
2169	libpcap	7	69	719	126	34	0.8312	0.3538	0.6699	0.4631	0.1491
2170	libpcap	8	76	709	136	27	0.8281	0.3585	0.7379	0.4825	0.1609
2171	libpcap	9	80	707	138	23	0.8302	0.3670	0.7767	0.4984	0.1633
2172	libpcap	10	77	702	143	26	0.8217	0.3500	0.7476	0.4768	0.1692
2173	libpcap	11	79	756	89	24	0.8808	0.4702	0.7670	0.5830	0.1053
2174	libpcap	12	77	759	86	26	0.8819	0.4724	0.7476	0.5789	0.1018
2175	libpcap	13	81	735	110	22	0.8608	0.4241	0.7864	0.5510	0.1302
2176	libpcap	14	83	745	100	20	0.8734	0.4536	0.8058	0.5804	0.1183
2177	libpcap	15	81	746	99	22	0.8724	0.4500	0.7864	0.5724	0.1172
2178	libpcap	16	80	744	101	23	0.8692	0.4420	0.7767	0.5634	0.1195
2179	libpcap	17	74	704	141	29	0.8207	0.3442	0.7184	0.4654	0.1669
2180	libpcap	18	79	720	125	24	0.8428	0.3873	0.7670	0.5147	0.1479
2181	libpcap	19	70	726	119	33	0.8397	0.3704	0.6796	0.4795	0.1408
2182	libpcap	20	82	721	124	21	0.8470	0.3981	0.7961	0.5307	0.1467
2183	libtiff	1	96	1015	164	35	0.8481	0.3692	0.7328	0.4910	0.1391
2184	libtiff	2	89	1076	103	42	0.8893	0.4635	0.6794	0.5511	0.0874
2185	libtiff	3	96	1043	136	35	0.8695	0.4138	0.7328	0.5289	0.1154
2186	libtiff	4	99	982	197	32	0.8252	0.3345	0.7557	0.4637	0.1671
2187	libtiff	5	96	1022	157	35	0.8534	0.3794	0.7328	0.5000	0.1332
2188	libtiff	6	94	1068	111	37	0.8870	0.4585	0.7176	0.5595	0.0941
2189	libtiff	7	97	977	202	34	0.8198	0.3244	0.7405	0.4512	0.1713
2190	libtiff	8	93	1032	147	38	0.8588	0.3875	0.7099	0.5013	0.1247
2191	libtiff	9	102	1001	178	29	0.8420	0.3643	0.7786	0.4964	0.1510
2192	libtiff	10	99	1024	155	32	0.8573	0.3898	0.7557	0.5143	0.1315
2193	libtiff	11	97	1032	147	34	0.8618	0.3975	0.7405	0.5173	0.1247
2194	libtiff	12	100	1019	160	31	0.8542	0.3846	0.7634	0.5115	0.1357
2195	libtiff	13	98	1079	100	33	0.8985	0.4949	0.7481	0.5957	0.0848
2196	libtiff	14	92	1040	139	39	0.8641	0.3983	0.7023	0.5083	0.1179
2197	libtiff	15	97	1011	168	34	0.8458	0.3660	0.7405	0.4899	0.1425
2198	libtiff	16	92	1052	127	39	0.8733	0.4201	0.7023	0.5257	0.1077
2199	libtiff	17	110	994	185	21	0.8427	0.3729	0.8397	0.5164	0.1569
2200	libtiff	18	90	1030	149	41	0.8550	0.3766	0.6870	0.4865	0.1264
2201	libtiff	19	91	1031	148	40	0.8565	0.3808	0.6947	0.4919	0.1255
2202	libtiff	20	94	998	181	37	0.8336	0.3418	0.7176	0.4631	0.1535
2203	libxml2	1	182	3128	192	119	0.9141	0.4866	0.6047	0.5393	0.0578
2204	libxml2	2	189	3198	122	112	0.9354	0.6077	0.6279	0.6176	0.0367
2205	libxml2	3	200	3142	178	101	0.9229	0.5291	0.6645	0.5891	0.0536

Continued on next page

Table 9 – Continued from previous page

	Library	Iteration	TP	TN	FP	FN	Accuracy	Precision	Recall	F1-score	FPR
2206											
2207											
2208	libxml2	4	197	3196	124	104	0.9370	0.6137	0.6545	0.6334	0.0373
2209	libxml2	5	193	3143	177	108	0.9213	0.5216	0.6412	0.5753	0.0533
2210	libxml2	6	203	3129	191	98	0.9202	0.5152	0.6744	0.5842	0.0575
2211	libxml2	7	196	3152	168	105	0.9246	0.5385	0.6512	0.5895	0.0506
2212	libxml2	8	181	3142	178	120	0.9177	0.5042	0.6013	0.5485	0.0536
2213	libxml2	9	205	3131	189	96	0.9213	0.5203	0.6811	0.5899	0.0569
2214	libxml2	10	198	3176	144	103	0.9318	0.5789	0.6578	0.6159	0.0434
2215	libxml2	11	183	3149	171	118	0.9202	0.5169	0.6080	0.5588	0.0515
2216	libxml2	12	206	3180	140	95	0.9351	0.5954	0.6844	0.6368	0.0422
2217	libxml2	13	176	3209	111	125	0.9348	0.6132	0.5847	0.5986	0.0334
2218	libxml2	14	194	3168	152	107	0.9285	0.5607	0.6445	0.5997	0.0458
2219	libxml2	15	189	3139	181	112	0.9191	0.5108	0.6279	0.5633	0.0545
2219	libxml2	16	186	3161	159	115	0.9243	0.5391	0.6179	0.5759	0.0479
2220	libxml2	17	180	3167	153	121	0.9243	0.5405	0.5980	0.5678	0.0461
2221	libxml2	18	188	3139	181	113	0.9188	0.5095	0.6246	0.5612	0.0545
2222	libxml2	19	199	3153	167	102	0.9257	0.5437	0.6611	0.5967	0.0503
2223	libxml2	20	195	3155	165	106	0.9252	0.5417	0.6478	0.5900	0.0497
2224	nm	1	90	1436	57	52	0.9333	0.6122	0.6338	0.6228	0.0382
2225	nm	2	102	1423	70	40	0.9327	0.5930	0.7183	0.6497	0.0469
2226	nm	3	104	1448	45	38	0.9492	0.6980	0.7324	0.7148	0.0301
2227	nm	4	99	1442	51	43	0.9425	0.6600	0.6972	0.6781	0.0342
2228	nm	5	91	1412	81	51	0.9193	0.5291	0.6408	0.5796	0.0543
2229	nm	6	96	1431	62	46	0.9339	0.6076	0.6761	0.6400	0.0415
2230	nm	7	103	1445	48	39	0.9468	0.6821	0.7254	0.7031	0.0322
2231	nm	8	100	1433	60	42	0.9376	0.6250	0.7042	0.6623	0.0402
2232	nm	9	101	1440	53	41	0.9425	0.6558	0.7113	0.6824	0.0355
2233	nm	10	88	1431	62	54	0.9291	0.5867	0.6197	0.6027	0.0415
2234	nm	11	97	1417	76	45	0.9260	0.5607	0.6831	0.6159	0.0509
2234	nm	12	103	1437	56	39	0.9419	0.6478	0.7254	0.6844	0.0375
2235	nm	13	102	1439	54	40	0.9425	0.6538	0.7183	0.6846	0.0362
2236	nm	14	100	1401	92	42	0.9180	0.5208	0.7042	0.5988	0.0616
2237	nm	15	99	1454	39	43	0.9498	0.7174	0.6972	0.7071	0.0261
2238	nm	16	103	1434	59	39	0.9401	0.6358	0.7254	0.6776	0.0395
2239	nm	17	94	1453	40	48	0.9462	0.7015	0.6620	0.6812	0.0268
2240	nm	18	99	1444	49	43	0.9437	0.6689	0.6972	0.6828	0.0328
2241	nm	19	98	1449	44	44	0.9462	0.6901	0.6901	0.6901	0.0295
2242	nm	20	93	1443	50	49	0.9394	0.6503	0.6549	0.6526	0.0335
2243	objdump	1	129	1912	61	87	0.9324	0.6789	0.5972	0.6355	0.0309
2244	objdump	2	119	1885	88	97	0.9155	0.5749	0.5509	0.5626	0.0446
2245	objdump	3	130	1870	103	86	0.9137	0.5579	0.6019	0.5791	0.0522
2246	objdump	4	138	1893	80	78	0.9278	0.6330	0.6389	0.6359	0.0405
2247	objdump	5	124	1874	99	92	0.9127	0.5561	0.5741	0.5649	0.0502
2248	objdump	6	125	1873	100	91	0.9127	0.5556	0.5787	0.5669	0.0507
2249	objdump	7	132	1882	91	84	0.9201	0.5919	0.6111	0.6014	0.0461
2250	objdump	8	129	1891	82	87	0.9228	0.6114	0.5972	0.6042	0.0416
2250	objdump	9	132	1872	101	84	0.9155	0.5665	0.6111	0.5880	0.0512
2251	objdump	10	135	1891	82	81	0.9255	0.6221	0.6250	0.6236	0.0416
2252											

Continued on next page

2253

2254

Table 9 – Continued from previous page

	Library	Iteration	TP	TN	FP	FN	Accuracy	Precision	Recall	F1-score	FPR
2255											
2256											
2257	objdump	11	123	1906	67	93	0.9269	0.6474	0.5694	0.6059	0.0340
2258	objdump	12	138	1905	68	78	0.9333	0.6699	0.6389	0.6540	0.0345
2259	objdump	13	141	1873	100	75	0.9201	0.5851	0.6528	0.6171	0.0507
2260	objdump	14	135	1865	108	81	0.9137	0.5556	0.6250	0.5882	0.0547
2261	objdump	15	125	1870	103	91	0.9114	0.5482	0.5787	0.5631	0.0522
2262	objdump	16	135	1883	90	81	0.9219	0.6000	0.6250	0.6122	0.0456
2263	objdump	17	130	1921	52	86	0.9370	0.7143	0.6019	0.6533	0.0264
2264	objdump	18	122	1852	121	94	0.9018	0.5021	0.5648	0.5316	0.0613
2265	objdump	19	121	1914	59	95	0.9296	0.6722	0.5602	0.6111	0.0299
2266	objdump	20	129	1888	85	87	0.9214	0.6028	0.5972	0.6000	0.0431
2267	size	1	120	1397	59	32	0.9434	0.6704	0.7895	0.7251	0.0405
2268	size	2	103	1386	70	49	0.9260	0.5954	0.6776	0.6338	0.0481
2269	size	3	95	1404	52	57	0.9322	0.6463	0.6250	0.6355	0.0357
2270	size	4	109	1387	69	43	0.9303	0.6124	0.7171	0.6606	0.0474
2271	size	5	111	1402	54	41	0.9409	0.6727	0.7303	0.7003	0.0371
2272	size	6	111	1401	55	41	0.9403	0.6687	0.7303	0.6981	0.0378
2273	size	7	105	1403	53	47	0.9378	0.6646	0.6908	0.6774	0.0364
2274	size	8	103	1408	48	49	0.9397	0.6821	0.6776	0.6799	0.0330
2275	size	9	107	1403	53	45	0.9391	0.6687	0.7039	0.6859	0.0364
2276	size	10	111	1403	53	41	0.9415	0.6768	0.7303	0.7025	0.0364
2277	size	11	113	1407	49	39	0.9453	0.6975	0.7434	0.7197	0.0337
2278	size	12	115	1404	52	37	0.9447	0.6886	0.7566	0.7210	0.0357
2279	size	13	101	1395	61	51	0.9303	0.6235	0.6645	0.6433	0.0419
2280	size	14	105	1388	68	47	0.9285	0.6069	0.6908	0.6462	0.0467
2281	size	15	102	1400	56	50	0.9341	0.6456	0.6711	0.6581	0.0385
2282	size	16	118	1384	72	34	0.9341	0.6211	0.7763	0.6901	0.0495
2283	size	17	108	1392	64	44	0.9328	0.6279	0.7105	0.6667	0.0440
2284	size	18	101	1407	49	51	0.9378	0.6733	0.6645	0.6689	0.0337
2285	size	19	119	1417	39	33	0.9552	0.7532	0.7829	0.7677	0.0268
2286	size	20	102	1415	41	50	0.9434	0.7133	0.6711	0.6915	0.0282
2287	gif2png	AVG	0.85	27	0	0.15	0.9946	0.8500	0.8500	0.8500	0.0000
2288	jasper	AVG	19.55	618.65	79.35	20.45	0.8648	0.2458	0.4888	0.3094	0.1137
2289	libpcap	AVG	77.20	736.10	108.90	25.80	0.8579	0.4196	0.7495	0.5365	0.1289
2290	libtiff	AVG	96.10	1026.30	152.70	34.90	0.8568	0.3909	0.7336	0.5082	0.1295
2291	libxml2	AVG	192	3157.85	162.15	109	0.9251	0.5444	0.6379	0.5866	0.0488
2292	nm	AVG	98.10	1435.60	57.40	43.90	0.9380	0.6348	0.6908	0.6605	0.0384
2293	objdump	AVG	129.60	1886	87	86.40	0.9208	0.6023	0.6000	0.5999	0.0441
2294	size	AVG	107.95	1400.15	55.85	44.05	0.9379	0.6604	0.7102	0.6836	0.0384
2295	OVERALL	AVG	90.17	1285.96	87.92	45.58	0.9120	0.5435	0.6826	0.5918	0.0677
2296											
2297											
2298											
2299											
2300											
2301											
2302											
2303											