# Function Clustering-Based Fuzzing Termination: Toward Smarter Early Stopping

Liang Ding
*University of Science and Technology of China*
Anhui, China
dingliang@mail.ustc.edu.cn

Wenzhang Yang*
*Institute of AI for Industries*
Nanjing, China
wzhyang@iaii.ac.cn

Yinxing Xue
*Institute of AI for Industries*
Nanjing, China
yxxue@iaii.ac.cn

***Abstract*—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 testing termination criterion based on function clustering. We compare our criterion with three existing methods. First, by leveraging language model for function encoding and a multi-metric fusion algorithm for determining the number of clusters, we establish a relationship between function clustering and vulnerability distribution. Second, our experiments on eight function libraries demonstrate that the proposed termination criterion 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 criteria like potential vulnerability function coverage-based approaches.**

***Index Terms*—fuzzing, function clustering, stopping criterion**

## I. INTRODUCTION

Fuzzing [1], an automated vulnerability detection technique, involves injecting abnormal data into target programs to trigger potential security defects. In recent years, as software complexity [2] has increased, fuzzing techniques have evolved from random mutation to more intelligent approaches. These advances include combining coverage feedback [3]–[5], symbolic execution [6], and AI optimization [7]. For instance, large language models are used to generate high-quality test seeds, and probabilistic models like HyperGo [8] enhance vulnerability detection speed. Fuzzing applications are now widespread across various fields, including intelligent connected vehicles [9], databases [10], and operating system kernels [11], significantly improving the depth of vulnerability discovery.

However, existing termination criterion 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 [12]. 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 [13] suggest using the growth of potential vulnerability function counts as a termination criterion. However, these methods present several notable issues [14]. 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 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 termination criterion for fuzz testing: the fuzzing campaign can be terminated when either (1) the number of covered function clusters increases (though not necessarily all functions are covered, indicating test coverage adequacy), or (2) crash occurrences grow within specific clusters (demonstrating testing effectiveness). The proposed method involves several key steps. First, cluster analysis is performed on the functions under test to

* Wenzhang Yang is the corresponding author.

TABLE I: 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** |

identify the cluster membership for each function, allowing for a better understanding of how functions relate to one another. By monitoring the number of clusters that have been tested and the number of triggered crashes in these clusters, this criterion provides a effective approach to determine the optimal stopping point for testing. Specifically, the termination criterion is triggered when the change in either code coverage across functional clusters or the crash count within any specific cluster falls below a set threshold over a time window.

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 cluster number selection algorithm. The approach is validated on the POJ-104 dataset, and its external metrics are analyzed. Next, the study empirically evaluates the cost-effectiveness trade-offs. Experiments are conducted on eight target programs. Eight state-of-the-art grey-box fuzzing tools are used, with each tool undergoing 20 repeated test runs (for a total of 1,280 fuzzing campaigns). Each run lasts 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.
- Through extensive experiments, we show that our proposed termination criterion 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

## II. MOTIVATION

In the domain of software security testing, recent years have witnessed a paradigm shift in vulnerability detection from coarse-grained to fine-grained [15] approaches. With the rapid and continuous evolution of modern software applications,

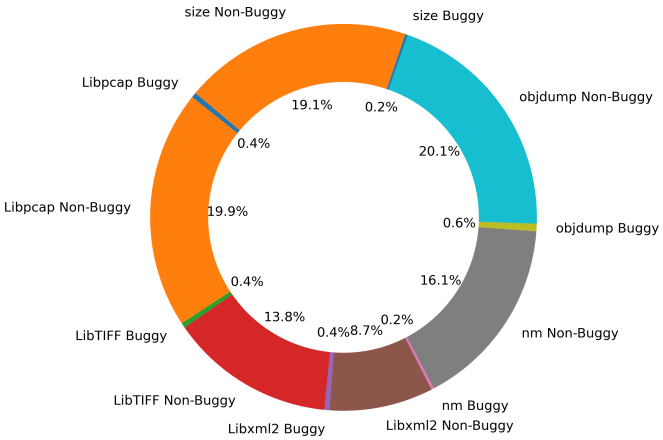


Fig. 1: Proportion of Buggy Function Clusters per Target

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 [16] are not uniformly distributed across codebases but exhibit significant correlations with functional units' semantic features, control flow patterns, and dependency networks. 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. However, existing termination criteria predominantly rely on single-dimensional metrics such as time thresholds or coverage saturation, failing to effectively integrate code semantic features with vulnerability risk propagation patterns, which leads to substantial misallocation of testing resources.

**Relationship between Function cluster and Vulnerability.** Function clustering [17] is a grouping method based on semantic similarity, functional proximity, or structural dependencies among functions. In recent years, with the application of deep learning techniques in code analysis [18], code-semantic-based function clustering has become a research hotspot. Analyzing the relationship between function clustering and vulnerabilities reveals several key points. First, there is a correlation between function semantic similarity and vulnerability distribution. In practical software, certain vulnerabilities exhibit distinct reproducible characteristics, where code with similar functionalities tends to harbor similar vulnerabilities. For example, common buffer overflow vulnerabilities in memory management functions and unchecked input vulnerabilities in input handling functions. By employing function semantic clustering, we can categorize code with similar functionalities, thereby concentrating testing resources on high-risk categories and enhancing the targeting of vulnerability detection. To validate the relationship between function clustering and vulnerability distribution, we utilize a comprehensive real-world

fuzz testing dataset from Fuzztastic [19], which includes basic-block-level coverage and crash logs from 12 subject programs. We begin by performing function clustering for each target program. Taking Libpcap as an example, its 497 functions are organized into 96 clusters. A cluster is designated as buggy if any function within it leads to a crash, as indicated by the corresponding crash logs. Table I reports the number of buggy clusters identified after function clustering and crash analysis. Figure 1 shows the proportion of buggy clusters across all target libraries. Statistical analysis shows that vulnerabilities are confined to only 14 clusters (approximately 2.9%) out of 472, with Figure 1 further confirming their marginal presence across all function libraries. Second, function clustering enhances the focus on high-risk areas. Through a clustering-based fuzz testing approach, functions can be prioritized for testing, allowing high-risk clusters to be tested first while reducing the intensity of testing in low-risk or less similar clusters. This method effectively minimizes resource wastage and significantly improves testing efficiency.

**Fuzzing Stopping Criterion based on Function Cluster.** This approach maintains testing coverage while significantly reducing testing time and resource costs. Functions are clustered based on characteristics such as functional similarity, potential vulnerability risks, and code dependency relationships. This allows for the identification of high-priority subsets, enabling focused testing on critical areas. In fuzz testing, using crash counts as an effectiveness metric can lead to underestimation or overestimation, potentially missing vulnerabilities or causing excessive testing duration. Code coverage, while reducing underestimation risks, may overestimate effectiveness by assuming uniform code importance and covering non-critical areas. Lip [13] proposed an improved method focusing on coverage saturation in potentially vulnerable code regions identified through static analysis. However, this method may terminate prematurely if too few vulnerable functions are predicted. To address these limitations, this paper proposes a fuzz testing termination criterion based on the number of function clusters, balancing resource efficiency and testing effectiveness.

## III. APPROACH

This section divides our approach into two parts. The Function Clustering part presents a CodeBERT-based method for grouping functions and identifying the optimal number of clusters. The next part demonstrates the application of our proposed termination criterion in fuzz testing.

### A. Function Clustering

Algorithm 1 enables fully automated end-to-end computation by intelligently determining the optimal number of clusters through a multi-objective optimization [20] framework. The process begins by normalizing and vectorizing input functions using CodeBERT [21] 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. Code Normalization & Vectorization**
$\mathbf{X} \leftarrow \emptyset$
**for** each function $f \in F$ **do**
  $f' \leftarrow \text{CodeNormalization}(f)$
  $\mathbf{x} \leftarrow \text{CodeBERT}(f')$
  $\mathbf{X} \leftarrow \mathbf{X} \cup \{\mathbf{x}\}$
**end for**
**2. Multi-Criteria Evaluation**
**for** $k = k_{\min}$ to $k_{\max}$ **do**
  $\{\mathcal{C}_1, ..., \mathcal{C}_k\} \leftarrow \text{K-Means}(\mathbf{X}, k)$
  Compute SSE, Silhouette, CH, and DB indices for current $k$
**end for**
**3. Candidate Subset Selection**
Compute second-order differences of SSE
$\mathcal{K}_{\text{elbow}} \leftarrow \{k \mid \Delta^2(k) < \theta_e\}$
$\mathcal{K}_{\text{sil}} \leftarrow \{k \mid \text{Local maxima of Silhouette}\}$
$\mathcal{K}_{\text{candidate}} \leftarrow \mathcal{K}_{\text{elbow}} \cap \mathcal{K}_{\text{sil}}$
**4. Pareto Frontier Construction**
$\mathcal{P} \leftarrow \{k \in \mathcal{K}_{\text{candidate}} \mid \nexists k' \in \mathcal{K}_{\text{candidate}}, F(k') \succ F(k)\}$
**5. Entropy-Weighted Decision**
Compute entropy weights for each metric
Compute score for each candidate cluster number
**return** $k^* \leftarrow \arg\max_{k \in \mathcal{P}} \text{Score}(k)$

*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 [22] to resolve multi-metric trade-offs. This approach ensures semantically coherent clusters while mitigating biases inherent to single-index evaluation.

*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) 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, Calinski-Harabasz Index, Davies-Bouldin Index and SSE. 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.

**SSE (Sum of Squared Errors).** 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:

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

Here, $k$ denotes the number of clusters, $C_i$ represents the set of data points in the $i$-th cluster, $\mu_i$ 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.** The silhouette coefficient 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:

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

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

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

Here, $n$ represents the total number of data points.

**CH Index (Calinski-Harabasz 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:

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

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

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

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, $\mu$ is the center of all samples, and $\mu_i$ is the center of the $i$-th cluster.

**DBI (Davies-Bouldin Index).** The DBI assesses clustering results based on cluster compactness and separation. A smaller DBI value signifies better clustering. The formula is:

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

Here, $\sigma_i$ measures the compactness of the $i$-th cluster, calculated as:

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

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

$$d(c_i, c_j) = \|c_i - c_j\| \tag{8}$$

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

*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 quantifies the inflection point trend by calculating the first-order difference ($\Delta(k)$) and second-order difference ($\Delta^2(k)$) of SSE, then automatically determines the threshold $\theta_e$ based on the statistical characteristics of the second-order differences (mean + 2 standard deviations). 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 [23]. For instance, optimizing one index may lead to suboptimal results for another, as each index 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 higher variability (lower entropy) 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:

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

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.

**Algorithm 2** Non-dominated Sorting Algorithm

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

```
for all k_i ∈ K_candidate do
    Compute dominance counter n_i and dominated set S_i for k_i
end for
Initialize front rank r ← 1
Initialize front F_r ← ∅
repeat
    for all ungraded k_i do
        if n_i == 0 then
            Assign k_i to front F_r
        end if
    end for
    for all k_j ∈ F_r do
        for all k_m ∈ S_j do
            Decrement n_m by 1: n_m ← n_m − 1
        end for
    end for
    Increment r by 1: r ← r + 1
until all solutions are graded
```

### *B. Termination Criterion for Fuzz Testing*

**Definition.** The growth of covered clusters is defined as:

$$\Delta C(t) = C(t) - C(t-1), \tag{10}$$

where $C(t)$ represents the number of covered clusters at time $t$. As fuzz testing progresses, if $\Delta C(t)$ remains below a threshold $\epsilon_C$ for a sustained period, it indicates that new test cases no longer trigger significant behavioral patterns.

By monitoring the trend of $\Delta C(t)$, the system can dynamically assess whether the newly generated test cases meaningfully expand behavioral coverage.

Similarly, crash count growth is defined as:

$$\Delta F(t) = F(t) - F(t-1), \tag{11}$$

where $F(t)$ represents the number of crashes detected at time $t$. In fuzz testing, crash events are critical indicators for identifying vulnerabilities. If $\Delta F(t)$ remains below a threshold $\epsilon_F$ for an extended duration, it suggests that the fuzzer has reached saturation in detecting new crashes.

The termination mechanism for fuzz testing is governed by saturation and tolerance thresholds. An evaluation metric $C$, defined in terms of indicators such as function coverage, crash counts, coverage of potentially vulnerable functions, and coverage of function clusters, is initialized. Fuzz tests are then executed iteratively with continuous updates of this metric. If the change in $C$ ($\Delta C$) falls below the termination tolerance $\epsilon$ within a specified saturation threshold $\delta$, the process is considered saturated and terminates early; otherwise, testing proceeds until a predefined timeout condition is met. In this

TABLE II: 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 |
| POJ104 | – | 1,885,262 | 168,990 |
| **Total** | – | **2,247,447** | **180,970** |

study, we further refine the termination criterion by incorporating both function-cluster coverage and crash-growth behavior. Specifically, the fuzzing process terminates when either (i) the increase in function-cluster coverage stagnates ($\Delta C(t) < \epsilon_C$), (ii) the growth in crash count within each cluster stagnates ($\Delta F(t) < \epsilon_F$), or (iii) the allocated testing time or resource budget is exhausted.

## IV. EXPERIMENTAL SETUP

### A. 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:

*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?

*2) RQ.2:* Analysis of Fuzzing Termination Criteria. What is the tradeoff between reduced fuzzing time and potential missed security bugs when using function cluster coverage (combined with function crashes) as a stopping criterion, compared to traditional metrics like crash count, regular function coverage, or potential vulnerability function coverage?

### B. Function Clustering

*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 [24] 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.

*2) Function Encoding:* We employ Tree-sitter [25] 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, encompassing both its declaration and full implementation. Then, we employ the CodeBERT-base model to extract features from source code in the POJ104 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 [26] 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.

*3) Clustering Method:* In this clustering analysis, we employed a variety of methods beyond just K-Means. We also utilized DBSCAN [27], which is effective for identifying clusters of arbitrary shapes and handling noise; Agglomerative clustering [28], a hierarchical approach that builds nested clusters; and Gaussian Mixture Models (GMM) [29], which models data as a mixture of Gaussian distributions to capture more complex cluster structures. To demonstrate both the necessity and validity of our proposed cluster selection approach, we adopt four representative clustering algorithms. 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 $\varepsilon$, 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) Evaluation Metrics:* Since the POJ-104 dataset contains ground truth labels, we employ two external evaluation metrics-Adjusted Rand Index (ARI) [30] and Normalized Mutual Information (NMI) [23] 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 be-

TABLE III: Parameter Configuration for Stopping Criteria Evaluation

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

tween 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.

### C. Tradeoff Evaluation

*1) Dataset:* Our study leverages the FuzzTastic [19] dataset, which provides comprehensive crash data and fine-grained coverage metrics from eight state-of-the-art greybox fuzzers: AFL [31], AFLFast [5], AFL++ [32], AFLSmart [33], FairFuzz [34], Honggfuzz [35], MOpt-AFL [36], and MOpt-AFL++ [37]. These fuzzers were evaluated on eight widely used libraries and utilities (Table II), 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 [38] suite. The dataset enables a robust analysis of efficiency-effectiveness trade-offs in fuzzing, particularly how reducing fuzzing time impacts bug detection rates.

*2) Evaluation Metrics:* Following the evaluation criteria established in this reaserch [13], 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 III, for the saturation window $\delta$ and tolerance $\epsilon$ parameters, we adopt the same settings from Green Fuzzing [13] 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 $\Delta t = T_{total} - t_S$ ($T_{total} = 24$h)
- **Missed bug ratio**: $\eta_b = \frac{N_{missed}}{N_{total}} \times 100\%$

*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 Green Fuzzing [13], and track their coverage during fuzzing. 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 ($\epsilon$); in such cases, both the runtime and the number of discovered bugs are recorded.

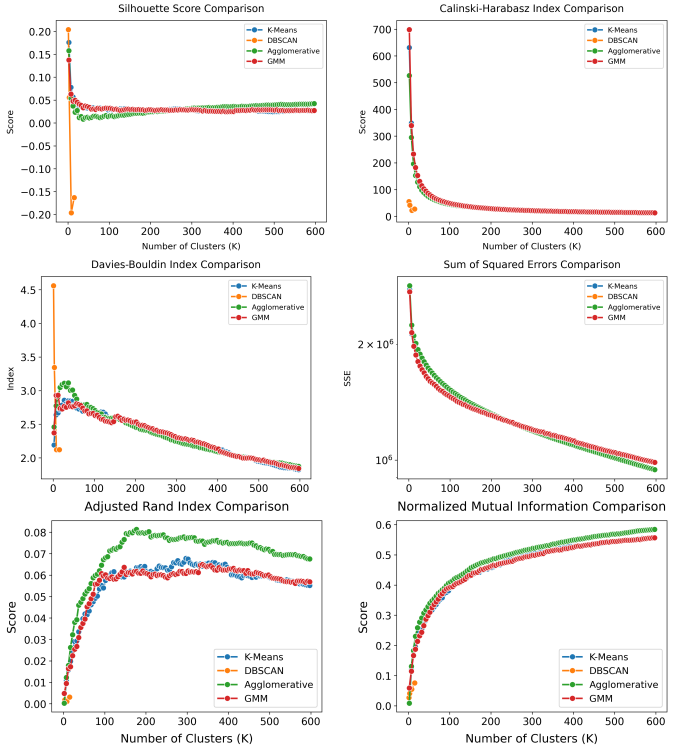


Fig. 2: Clustering Metrics (Top) and External Metrics Comparison (Bottom)

## V. Evaluation

### A. 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.

*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 POJ104. From Figure 2 preliminary implementation of hierarchical clustering demonstrated limited efficacy for functional clustering tasks requiring adaptive determination of optimal cluster numbers. The silhouette coefficient (SH) 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 (SSE) shows an initial steep drop, leveling off after $k = 105$. For clustering algorithms such

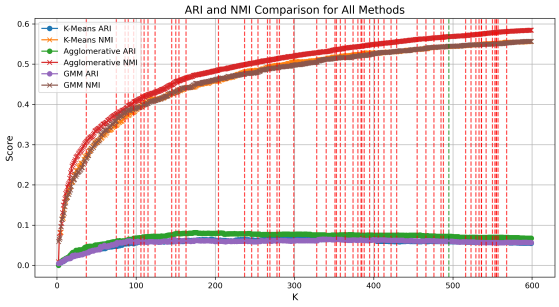


Fig. 3: ARI and NMI vs. Number of Clusters

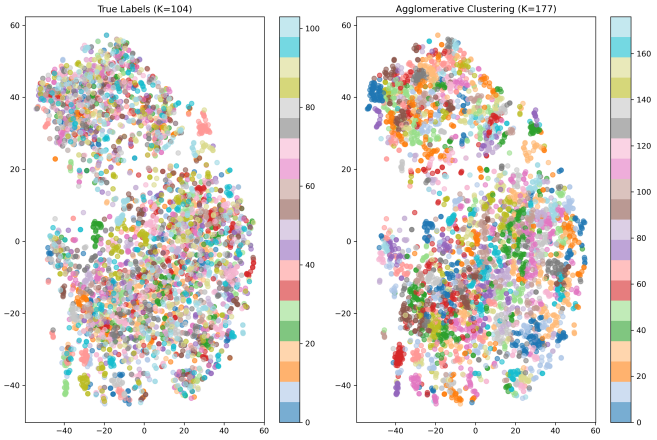


Fig. 4: Visualization of Clustering Results

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 SH 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 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.

*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 POJ104 dataset. The proposed clustering algorithm demonstrates robust performance in automatically selecting the optimal number of clusters across varying $k$ values ($k = [37.0, 75.0, \ldots, 550.0]$) with Figure 3. Evaluation metrics reveal a normalized mutual information (NMI) score of 79% (exceeding the 0.5 threshold) [39], 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.

TABLE IV: 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".

***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.

*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 IV 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: Tradeoff results of our stopping criterion relative to the saturation of potential vulnerability function coverage.

### B. RQ2: Analysis of Fuzzing Termination Criterion

**Comparison: Crash Stopping Criterion.** For Gif2png, Jasper, LibTIFF, Libxml2, and objdump, our proposed termination criterion achieved significantly earlier termination compared to the crash-count-based criterion. Table V 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 $\delta$ and the tolerance parameter $\epsilon$, as specified in Table III. Specifically, our method reduced testing time by 5.0 to 9.7 hours, representing 20%–40% of the total fuzzing duration (Table V). 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 V). For optimal cases like AFL on objdump achieved 18-hour reductions (75%), though littile 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

TABLE V: Average Performance Across All Fuzzers by Subject

| Subject | Timeout | | | Bugs | | |
|---|---|---|---|---|---|---|
| | T1 | T2 | T3 | B1 | B2 | B3 |
| 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 |

T1-T3 (timeout differences) quantify how much earlier our function clustering metric terminates compared to baseline methods (crash counts, code coverage, vulnerable function coverage); B1-B3 (bug detection differences) indicate additional bugs missed by our metric.

missed two or more bugs, confirming the criterion's reliability. The trade-off between time savings and bug detection 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 5 and Figure 6 further illustrate the tradeoff results of our stopping criterion in comparison with the saturation of vulnerable function coverage. As shown in the Figure 5, 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 proposed criterion extends fuzzing duration by 0.9 hours, it detects additional bugs, demonstrating its effectiveness.

Figure 6 illustrates that the majority of campaigns (86.5%) achieved complete bug detection, while those that missed bugs demonstrated an average reduction in runtime of 2.1 hours (equivalent to 8%). Instances where two or more bugs were missed accounted for only 5.1%, underscoring the robustness of the proposed criterion. This demonstrates that the method effectively balances efficiency and accuracy, enabling significant time savings while ensuring the majority of vulnerabilities are uncovered.

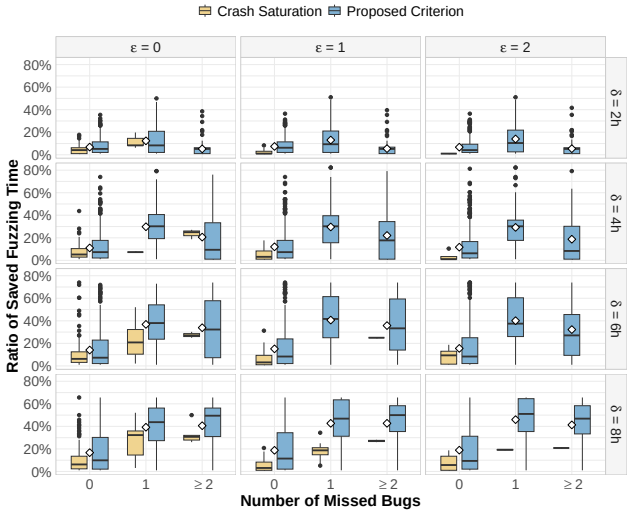


Fig. 6: Time saved by the fuzzing stopping criterion versus undetected bugs compared to vulnerability function coverage saturation.

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

## VI. DISCUSSION

### A. Function Clustering

The Agglomerative Clustering method in our function clustering framework is slower but yields better clustering coefficient results, making it suitable for scenarios where clustering quality is prioritized. Its hierarchical nature leads to higher computational complexity, but the resulting clusters are more meaningful and consistent, which is crucial for downstream tasks like code analysis and software maintenance. DBSCAN, which does not require predefining the number of clusters $k$, performed poorly and still needed parameter tuning for optimal results. The trade-off between speed and quality needs to 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.

### B. Fuzzing Termination Criterion

Our function clustering-based fuzzing termination criterion can terminate fuzzing earlier compared to the current three cri-

teria (crash count, function coverage, and vulnerable function coverage) while missing very few vulnerabilities. In real-world corporate or research scenarios, fuzzing typically runs for 7 days × 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 growth rates, or vulnerable function coverage—as a potential way to improve flexibility and comprehensiveness in testing.

## VII. THREATS TO VALIDITY

### A. External Validity

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

### B. Internal Validity

This threat category pertains to the reasonableness of the methodology and experimental design. Since the actual objective function library lacks authentic clustering metrics, we comprehensively integrated multiple internal clustering indicators to determine an appropriate number of clusters. Given that these indicators are mutually conflicting, we adopted a multi-objective optimization approach, which enables the derivation of a rational cluster count.

Regarding the evaluation of fuzz testing termination criteria, we utilized the comprehensive coverage dataset provided by Fuzztastic, which encompasses diverse function libraries and multiple fuzz testing tools with repeated executions. This approach mitigates, to some extent, the randomness inherent in fuzz testing.

## VIII. RELATED WORK

**Code clustering.** Prior research on code similarity has predominantly focused on text-based or semantic analysis methods, leveraging techniques such as Abstract Syntax Trees (ASTs) and machine learning models to categorize code segments based on their functionality and structure [40]–[42]. However, these approaches often rely on predefined similarity metrics like Euclidean distance, which may not effectively capture the nuances of complex code. Research on function clustering remains relatively under-explored, particularly in terms of analyzing relationships among functions within a codebase. There is still limited understanding of how functions with similar functionalities or implementation patterns correlate with one another. Notably, functions that share analogous structures or purposes may also exhibit similar vulnerabilities, suggesting that clustering techniques could aid in identifying security flaws more systematically. Building on this foundation, our work introduces a novel clustering method that dynamically selects the optimal number of clusters by integrating multiple internal clustering metrics, including the Silhouette Coefficient, Davies-Bouldin Index (DBI), SSE, and Calinski-Harabasz Index (CHI). This multi-metric approach aims to enhance the robustness and adaptability of clustering results, addressing the limitations of traditional single-metric methods.

**Fuzzing Stopping Criteria.** Accurately determining the optimal stopping point for fuzzing campaigns remains a critical challenge in fuzz testing. While numerous studies [43], [44] focus on optimizing code coverage techniques to enhance test suite effectiveness, they often overlook aspects such as code similarity analysis, which could refine early stopping criteria. Current practices [14] rely on fixed time budgets (e.g. 24 hours to 7 days) for fuzzing evaluations , potentially leading to inefficient resource usage. Recent advancements in green fuzzing [13] emphasize energy-efficient fuzzing strategies, seeking to balance computational cost and effectiveness, further highlighting the need for adaptive and resource-aware stopping criteria in fuzzing campaigns.

## IX. CONCLUSION

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

## ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the CCF-Huawei Huyanglin Fund (CCF-HuaweiSE202313).

## REFERENCES


[1] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[2] S. Yu and S. Zhou, "A survey on metric of software complexity," in *2010 2nd IEEE International conference on information management and engineering*, pp. 352–356, IEEE, 2010.

[3] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "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)*, pp. 1–15, 2019.

[4] W.-C. Wu, B. Nongpoh, M. Nour, M. Marcozzi, S. Bardin, and C. Hauser, "Fine-grained coverage-based fuzzing," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 5, pp. 1–41, 2024.

[5] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1032–1043, 2016.

[6] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution.," in *NDSS*, vol. 16, pp. 1–16, 2016.

[7] S. B. Chafjiri, P. Legg, J. Hong, and M.-A. Tsompanas, "Vulnerability detection through machine learning-based fuzzing: A systematic review," *Computers & Security*, p. 103903, 2024.

[8] P. Lin, P. Wang, X. Zhou, W. Xie, K. Lu, and G. Zhang, "Hypergo: Probability-based directed hybrid fuzzing," *Computers & Security*, vol. 142, p. 103851, 2024.

[9] Z. Zhong, G. Kaiser, and B. Ray, "Neural network guided evolutionary fuzzing for finding traffic violations of autonomous vehicles," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1860–1875, 2022.

[10] Y. Zhang, H. Lu, Z. Zhang, G. Wu, H. Zhou, and Z. Li, "Review of fuzz testing techniques for database management systems," in *International Conference on Cyberspace Simulation and Evaluation*, pp. 283–301, Springer, 2024.

[11] K. Kim, D. R. Jeong, C. H. Kim, Y. Jang, I. Shin, and B. Lee, "Hfl: Hybrid fuzzing on the linux kernel.," in *NDSS*, 2020.

[12] M. Böhme and B. Falk, "Fuzzing: On the exponential cost of vulnerability discovery," in *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pp. 713–724, 2020.

[13] S. Lipp, D. Elsner, S. Kacianka, A. Pretschner, M. Böhme, and S. Banescu, "Green fuzzing: A saturation-based stopping criterion using vulnerability prediction," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 127–139, 2023.

[14] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pp. 2123–2138, 2018.

[15] H. Hanif, M. H. N. M. Nasir, M. F. Ab Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches," *Journal of Network and Computer Applications*, vol. 179, p. 103009, 2021.

[16] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "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*, pp. 1547–1559, 2020.

[17] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan, "Binary function clustering using semantic hashes," in *2012 11th International Conference on Machine Learning and Applications*, vol. 1, pp. 386–391, IEEE, 2012.

[18] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pp. 87–98, 2016.

[19] S. Lipp, D. Elsner, T. Hutzelmann, S. Banescu, A. Pretschner, and M. Böhme, "Fuzztastic: A fine-grained, fuzzer-agnostic coverage analyzer," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, pp. 75–79, 2022.

[20] D. Schuhmacher, B.-T. Vo, and B.-N. Vo, "A consistent metric for performance evaluation of multi-object filters," *IEEE transactions on signal processing*, vol. 56, no. 8, pp. 3447–3457, 2008.

[21] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[22] Y. Censor, "Pareto optimality in multiobjective problems," *Applied Mathematics and Optimization*, vol. 4, no. 1, pp. 41–59, 1977.

[23] E. Amigó, J. Gonzalo, J. Artiles, and F. Verdejo, "A comparison of extrinsic clustering evaluation metrics based on formal constraints," *Information retrieval*, vol. 12, pp. 461–486, 2009.

[24] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, 2016.

[25] M. Brunsfeld and contributors, "Tree-sitter: An incremental parsing system for programming tools." https://github.com/tree-sitter/tree-sitter, 2018.

[26] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[27] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, "Dbscan revisited, revisited: why and how you should (still) use dbscan," *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

[28] D. Müllner, "Modern hierarchical, agglomerative clustering algorithms," *arXiv preprint arXiv:1109.2378*, 2011.

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

[30] A. Strehl and J. Ghosh, "Cluster ensembles—a knowledge reuse framework for combining multiple partitions," *Journal of machine learning research*, vol. 3, no. Dec, pp. 583–617, 2002.

[31] M. Zalewski, "American fuzzy lop: Fuzzing with a brain," in *Black Hat USA*, 2014. Accessed: 2025-05-31.

[32] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "{AFL++}: Combining incremental steps of fuzzing research," in *14th USENIX workshop on offensive technologies (WOOT 20)*, 2020.

[33] V.-T. Pham, M. Böhme, A. E. Santosa, A. R. Căciulescu, and A. Roychoudhury, "Smart greybox fuzzing," *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1980–1997, 2019.

[34] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE international conference on automated software engineering*, pp. 475–485, 2018.

[35] R. Swiecki, "Honggfuzz: A practical fuzzer for security testing," in *Black Hat USA*, 2016. Accessed: 2025-05-31.

[36] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "{MOPT}: Optimized mutation scheduling for fuzzers," in *28th USENIX security symposium (USENIX security 19)*, pp. 1949–1966, 2019.

[37] D. Maier, A. Fioraldi, A. Fioraldi, *et al.*, "mopt-afl++: Enhancing fuzzing with mopt mutator," in *Proceedings of the 2023 International Conference on Software Testing, Verification, and Validation*, 2023. Accessed: 2025-05-31.

[38] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A ground-truth fuzzing benchmark," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 4, no. 3, pp. 1–29, 2020.

[39] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*, vol. 39. Cambridge University Press Cambridge, 2008.

[40] A. Kuhn, S. Ducasse, and T. Gîrba, "Semantic clustering: Identifying topics in source code," *Information and software technology*, vol. 49, no. 3, pp. 230–243, 2007.

[41] L. Büch and A. Andrzejak, "Learning-based recursive aggregation of abstract syntax trees for code clone detection," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 95–104, IEEE, 2019.

[42] G. Zhao and J. Huang, "Deepsim: deep learning code functional similarity," in *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pp. 141–151, 2018.

[43] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Transactions on Software Engineering*, vol. 32, no. 8, pp. 608–624, 2006.

[44] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, pp. 1621–1633, 2022.