APIECHO: Training-less Anomaly Detection via Intra-API Behavioral Comparison for Web Applications

Yihao Peng*†, Yiming Wu*†, Du Wu*, Shouling Ji§, Hai Wan*(✉) and Xibin Zhao*(✉)
*BNRist, KLISS, School of Software, Tsinghua University, §Zhejiang University
{pyh23,wuym23,wd22}@mails.tsinghua.edu.cn, sji@zju.edu.cn,
{wanhai,zxb}@tsinghua.edu.cn

Abstract*—Anomaly detection is crucial for web application security, yet existing methods like rule-based validation and learning-based models face significant limitations. Rule-based systems struggle with novel attacks, while learning-based approaches require frequent, costly retraining to adapt to dynamic application updates, often leading to high false positives. While recent self-comparison methods address retraining by comparing replicas in microservice scenarios, they are ill-suited for monolithic applications due to functional heterogeneity, offer coarse-grained detection, lack adaptive comparison baselines, and are vulnerable to coordinated poisoning. This paper presents APIECHO, a novel web server intrusion detection method for monolithic applications that operates without large-scale pre-training. APIECHO's core insight is that *legitimate requests to the same API endpoint exhibit highly similar underlying behavioral patterns*. Our system shifts the comparison granularity from replicas to individual requests within the same API, employing dynamic API classification, fine-grained behavioral feature extraction (including sequential and set-based features), per-API adaptive similarity thresholds, and an anti-poisoning sliding window update mechanism. Extensive evaluations on 16 real-world scenarios demonstrate that APIECHO significantly outperforms state-of-the-art methods. It effectively adapts to application updates without retraining, resists coordinated poisoning attacks, surpasses existing methods in average detection score, and achieves attack recall rates exceeding 90% while maintaining benign event detection accuracy above 99%, all with low overhead, processing more than 12000 log events per second with less than 7GB memory consumption.

1. Introduction

Anomaly detection in web applications is crucial for identifying suspicious activities that deviate from normal behavior, thereby bolstering system security against unknown attacks [1]–[3]. Traditional methods predominantly fall into two categories. Rule-based validation checks compliance with predefined security policies [4], [5], offering simplicity but failing to detect novel attacks absent from these rules [6]. Alternatively, learning-based methods model normal behavior from clean data [7]–[13]. While capable of detecting some unknown attacks, these methods struggle with the dynamic nature of modern web applications. Frequent updates alter "normal behavior", necessitating model retraining to avoid excessive false positives, a process that is both resource-intensive and reliant on often elusive clean post-update datasets.

To mitigate the dependency on training baselines, recent research [6] has introduced approaches for microservice architectures leveraging the high behavioral similarity among service replicas. These self-comparison methods detect anomalies by identifying replicas whose behavior deviates significantly from others within the same time window, thereby obviating retraining as all replicas' behaviors co-evolve with updates. However, such replica-centric comparison techniques face several challenges, particularly with monolithic applications or sophisticated attacks:

- **Applicability to Monolithic Architectures**: The core assumption of functional singularity and behavioral homogeneity among replicas is often invalid for monolithic applications. Replicas of monolithic systems typically handle diverse functions, leading to significant behavioral variations even under normal operation, which can trigger false positives for replica-comparison systems.
- **Coarse-grained Detection and Limited Features**: Analyzing aggregated replica behavior over time windows restricts the ability to pinpoint malicious requests and limits feature extraction to primarily statistical, set-based measures. This overlooks subtle, sequence-based anomalies, hindering precise attack tracking and identification.
- **Static Comparison Baselines**: Employing uniform comparison thresholds across all behaviors fails to account for the inherent variability in different system functions (e.g., login vs. file upload operations), potentially leading to both false positives and negatives.
- **Susceptibility to Coordinated Poisoning Attacks**:

†. Both authors contributed equally to the paper.
✉. Corresponding author.

If an attacker compromises most or all replicas with similar malicious behavior simultaneously, the behavioral differences between replicas diminish, potentially evading detection by these comparison schemes.

Our Solution. To address these limitations, we propose APIECHO, a novel web server intrusion detection method designed for monolithic applications. APIECHO retains the advantages of comparison-based detection (i.e., no large-scale pre-training) while enhancing detection accuracy and robustness. It is founded on the key observation that *even within complex monolithic applications, multiple legitimate requests targeting the same API should exhibit highly similar underlying behavioral patterns*. Consequently, APIECHO shifts the self-comparison granularity from inter-replica to intra-API request comparisons. APIECHO incorporates the following key designs:

- **API-Centric Comparison for Monolithic Applicability**: APIECHO classifies incoming requests to their specific API endpoints using a dynamic API classification engine. This engine, inspired by [14] but featuring a novel trie-based matching logic, dynamically identifies URL parameters and adapts to API evolution. By comparing behaviors of requests for the same API, it naturally handles the functional diversity of monolithic applications.
- **Fine-Grained and Sequence-Aware Detection**: APIECHO achieves fine-grained analysis by partitioning system-level audit logs into behavioral units corresponding to individual web requests [15]. This enables pinpointing malicious requests and extracting rich sequential features (e.g., syscall order) alongside set-based features, thereby detecting subtle attacks missed by aggregate analysis.
- **Adaptive Per-API Comparison Baselines**: As different APIs have distinct behavioral norms, APIECHO independently calculates behavioral similarity distributions for each API's request stream within sliding windows and applies adaptive normalization and thresholding, akin to batch normalization.
- **Resilience to Coordinated Poisoning**: APIECHO performs detection at the individual request level. Anomalous requests, identified by deviation from current normal behavior within an API's sliding window, are prevented from being incorporated into subsequent windows, thus preserving integrity of normal behavior baseline against poisoning attempts.

The workflow of APIECHO involves: partitioning system-level audit logs into per-request behavior units; classifying units to specific APIs using dynamic API classification engine; extracting set and sequence features for requests within each API's sliding window; computing inter-request similarities; and identifying anomalies using an adaptive algorithm. Sliding windows for each API are dynamically managed based on detection outcomes and external update notifications to maintain sensitivity and adapt to application evolution.

Experiments and Evaluations. We conducted a series of experiments to comprehensively evaluate the effectiveness of APIECHO. We constructed 16 experimental scenarios using real-world Web applications and vulnerabilities, and compared APIECHO with the state-of-the-art non-pre-training method ReplicaWatcher [6], as well as pre-training-based methods KAIROS [7] and ProvDetector [8]. Results demonstrate APIECHO's significantly superior detection performance, surpassing existing methods in average detection score, achieving attack recall rates exceeding 90% while maintaining benign event detection accuracy above 99%, its effective resistance to coordinated poisoning attacks, and its ability to adapt to application updates without retraining, all with low computational overhead, processing more than 12000 log events per second with less than 7GB memory consumption. Ablation studies further validate the contribution of each design component.

Our Contributions:

- We propose a novel web server intrusion detection method without needs of large-scale pre-training, based on comparing behavior of different requests of the same API, overcoming limitations of existing self-comparison methods for monolithic applications.
- We design and implement the APIECHO system, which integrates dynamic API classification, multi-dimensional feature extraction based on sequences and sets, per-API adaptive thresholds, and anti-poisoning sliding window update mechanisms.
- We conducted extensive evaluations demonstrating that APIECHO adapts to application updates without retraining and effectively resists coordinated poisoning attacks. In 16 scenarios using real-world web applications, APIECHO's detection performance surpasses state-of-the-art pre-training-based methods, achieving attack recall exceeding 90% while maintaining benign event detection accuracy above 99%.

In the spirit of open science, we make our tool available at https://github.com/finall1008/apiecho.

2. Motivation

To illustrate the challenges addressed by APIECHO, consider a scenario with a frequently updated, monolithic FastAPI web application, based on our experimental data (`python-demo`, see Section 5).

Pre-training Anomaly Detection Solutions. Initially, a pre-training solution like ProvDetector [8] was deployed. It performed well on the validation set of attacks, achieving nearly 100% recall and an overall accuracy of 92.2%. However, a routine application upgrade rendered its baseline model obsolete, causing accuracy to plummet to 54.1% due to a surge in false positives. Restoring performance required retraining, a process demanding expert intervention, substantial clean traffic data, and significant computational resources (e.g., GPUs, hours to over a day). For an application with

frequent updates, this operational overhead and slow response were impractical.

Replica-based Self-comparison Methods. Subsequently, a self-comparison method ReplicaWatcher [6] that obviates pre-training was explored, discovering anomalies by comparing behavioral similarity across service replicas within the same time window. However, for a monolithic application with diverse functionalities, different replicas often processed varied request types and executed distinct functional paths simultaneously. This inherent behavioral heterogeneity led to numerous false positives. Adjusting the threshold to mitigate this improved accuracy to 98.5% but drastically reduced attack detection recall to only 50%. Even when a genuine attack was identified, the coarse-grained detection could only flag the anomalous time window, failing to pinpoint the specific malicious request.

More critically, such methods are vulnerable to coordinated poisoning attacks [6]. If attackers launch similar malicious behaviors across most or all replicas, the behavioral differences between them diminish, hindering detection. This vulnerability was demonstrated in a simulated coordinated poisoning attack (repeatedly executing the attack API `/solve`, causing all replicas to process attack requests), where even with further threshold adjustments, attempts to maintain a 55.0% recall resulted in accuracy dropping to 51.4%, rendering the system ineffective.

This predicament underscores need for a novel web server intrusion detection method that: 1) avoids large-scale, time-consuming pre-training and frequent retraining; 2) effectively adapts to complex monolithic web applications, maintaining low false positives and high recall despite behavioral heterogeneity; 3) offers more fine-grained detection to precisely identify malicious activities; and 4) is robust against advanced attacks like coordinated poisoning. APIECHO is designed to address these core challenges.

3. Threat Model

APIECHO primarily targets remote attacks against monolithic web applications launched via public network interfaces. Attackers typically exploit software vulnerabilities through malicious HTTP requests to achieve objectives such as control-flow hijacking, sensitive data theft (e.g., authentication bypass, directory traversal), file system integrity compromise (e.g., file inclusion vulnerabilities), or unauthorized code and command execution (e.g., command injection).

Similar to many anomaly detection methods based on audit logs [6]–[8], attacks not exhibiting discernible syscall level behavioral differences, such as certain purely application-logic data tampering (if it doesn't alter underlying behavior), are currently outside APIECHO's direct scope. Similarly, side-channel attacks, hardware-level exploits (e.g., Spectre), resource exhaustion denial-of-service attacks (e.g., network floods), remote passive system fingerprinting, network spoofing, and scenarios requiring prior physical access or internal network compromise are not the primary focus of our current method.

We assume the server's operating system provides reliable audit logging for system-level behaviors (including process/thread identifiers) and that APIECHO can access and process these logs. Furthermore, we presume the audit log generation mechanism and APIECHO itself are part of the trusted computing base (TCB), meaning their integrity and availability are protected from attacker tampering—a common and necessary assumption for many log-dependent security analysis and intrusion detection systems [15]–[17].

4. System Design

4.1. System Design Overview

APIECHO's workflow has three parts (Figure 1):

1) **Request Partition and API Classification** (Section 4.2). APIECHO decomposes audit log entries into Web API-classified units.
   - *Unit Partition*. APIECHO first uses the unit partitioning method from [15], analyzing the web application's execution model to embed partition information into audit logs. It then uses this information to decompose audit logs into units, each corresponding to a web request.
   - *Dynamic API Classification*. To identify API of each web request, APIECHO maintains a trie-based API catalog for real-time classification. For each request unit, APIECHO extracts its web request URL and queries the API catalog for the API. Concurrently, this URL updates the API catalog, enabling adaptation to application updates and automatic correction of outdated APIs.
2) **Intra-API Anomaly Detection** (Section 4.3). For each API, APIECHO maintains a detection window of several request units. It compares request unit similarity within this window to output the detection result for the latest unit.
   - *Feature Extraction*. For each request unit in the window, APIECHO extracts sequential and set features from its audit log entries.
   - *Similarity-based Anomaly Detection*. APIECHO calculates similarity scores between request unit features, effectively converting them as vectors. An anomaly detection algorithm then identifies outliers, flagging corresponding request units as anomalous.
3) **Window Management and Baseline Adaptation** (Section 4.4). APIECHO determines the next detection window's content based on the latest unit's detection result. If malicious, the request unit is removed from the window to ensure subsequent

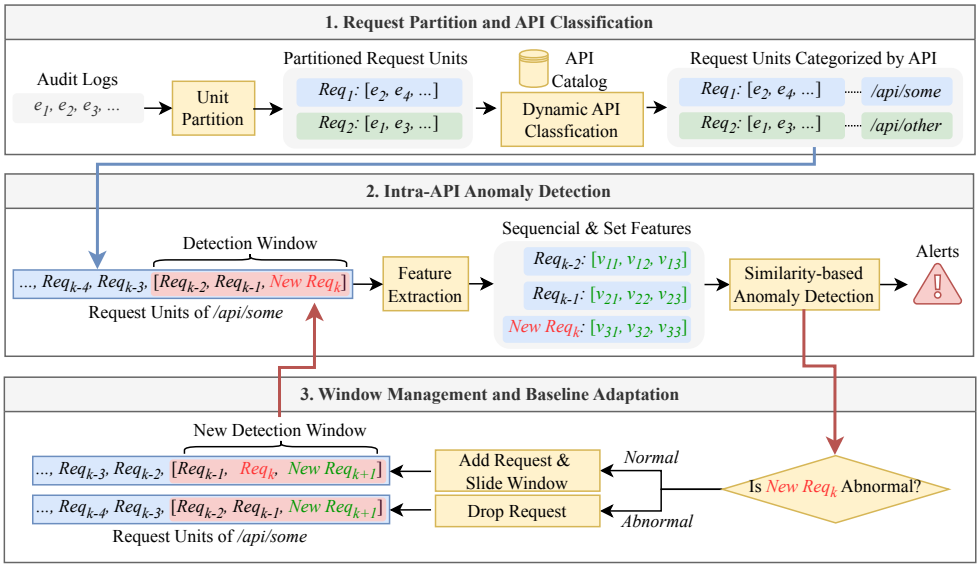

Figure 1. APIECHO Overview

detection accuracy and defend against coordinated poisoning. Otherwise, the unit is retained to adapt to normal behavior drifts.

4.2. Request Partition and API Classification

APIECHO's first stage, Request Partition and API Classification, transforms audit logs into behavioral units and classifies them to an API endpoint respectively. This is crucial for subsequent fine-grained anomaly detection by enabling analysis of comparable behavioral sets. It involves: *Unit Partition* isolates request-specific segments from commingled audit logs and *Dynamic API Classification* maps units to APIs.

4.2.1. Unit Partition. The *Unit Partition* module precisely segments commingled system-level audit logs, associating segments with specific web requests. This is foundational for fine-grained behavioral modeling and comparison of individual requests.

We adopt TeSec's audit log partitioning [15]. This addresses: 1) correlating low-level system behaviors (syscalls) with high-level requests (user API calls), and 2) handling log entry interleaving from web server concurrency (multi-processing/threading, coroutines) to ensure each unit contains only single-request behaviors.

TeSec's core idea is inserting *delimiter logs* into audit logs via lightweight web application instrumentation at critical request processing points. These delimiters contain an unique ID for specific backend requests. Specifically, it generates delimiter logs by writing code to output category, unit ID, and other information to `/dev/null`, which does not affect the application's original behavior but allows these details to be captured in audit logs at the required locations.

This technique's approach varies by web application concurrency model:

1) *Multi-processing / Multi-threading based concurrency model.* The application typically uses a process/thread pool, in which each worker handles requests sequentially, proceeding to the next after finishing one request. Thus, syscalls from the same worker are logged sequentially; PID/TID in logs allows grouping interleaved entries. TeSec instruments the application to insert a delimiter log with a request ID before a worker processes a new request. Audit entries associate with a request by tracing back to the nearest delimiter within their PID/TID log sequence. PID/TID distinction prevents interleaving from affecting intra-worker log sequentiality.
2) *Coroutine-based concurrency model.* For coroutine models, OS audit logs don't capture user-space coroutine switches, so TID alone is insufficient. TeSec addresses this with finer-grained instrumentation, inserting delimiter logs not only at request start but also at each coroutine switch. This allows accurate segmentation of commingled syscalls within a single TID's log stream.

Adopting TeSec's unit partitioning allows APIECHO to extract accurate behavioral units (syscall sequence) per request from raw audit logs, foundational for subsequent steps. TeSec's advantages include lightweight, often plugin-based modifications (no core code changes) and linear time partitioning.

4.2.2. Dynamic API Classification. After *Unit Partition* yields behavioral units for individual web requests, the next step is *Dynamic API Classification*. Its purpose is to accurately map each request unit to its

specific application API. For this, especially with evolving APIs in modern web applications, APIECHO uses a dynamic, trie-based API catalog management and classification mechanism. Inspired by Akto [14] for API auto-discovery, it automatically learns and identifies API structures from URLs (including parameterized segments) and adapts to API changes.

Algorithm 1: Dynamic API Classification

Input :
req_unit: A partitioned request unit containing HTTP method and URL
Output :
api_id: Classified API identifier of *req_unit*
Variables :
catalog: *Trie*, stores API templates
stat_keys: *Set*, unmatched concrete URLs awaiting generalization
param_mons: *Map*, parameter value monitors for API evolution tracking
Functions:
EXTRACTKEY(*req_unit*): Extract normalized HTTP method and URL path
FINDGENKEY(*tgt_key, key_set*): Find a static key that can generalize with target
GENTPL(*k1, k2*): Create parameterized template from two similar keys
UPDATEPMON(*id, key*): Record parameter values for evolution tracking
CHECKSPLIT(*id*): Split API template if parameter diversity becomes low
FINDCOVKEYS(*tpl, key_set*): Find all static keys covered by template

```
1  Function CLASSIFYREQ(req_unit):
2      s_key ← EXTRACTKEY(req_unit);
3      api_id ← catalog.MATCH(s_key);
4      if api_id ≠ NULL then
5          UPDATEPMON(api_id, s_key);
6          CHECKSPLIT(api_id);
7          return api_id;
8      end
9      stat_keys.ADD(s_key);
10     gen_key ← FINDGENKEY(s_key, stat_keys);
11     if gen_key ≠ NULL then
12         new_tpl ← GENTPL(s_key, gen_key);
13         new_id ← catalog.ADDTPL(new_tpl);
14         cov_keys ←
             FINDCOVKEYS(new_tpl, stat_keys);
15         forall k ∈ cov_keys do
16             stat_keys.REMOVE(k);
17             UPDATEPMON(new_id, k);
18         end
19         return new_id;
20     end
21     return catalog.NEWTMPAPI(s_key);
```

Algorithm 1 outlines this: new requests match against the API catalog (a trie), and unmatched URLs are compared with others to discover common patterns and generalize new API templates, updating catalog accordingly. This mechanism also adapts to API structural evolution (e.g., parameterized segments becoming static paths). Figure 2 exemplifies this.

API Catalog Structure and Request Matching. APIECHO uses a trie [18] for its API catalog, efficiently storing and retrieving path-based API definitions. As in Figure 2(a), each node represents an HTTP method or URL path segment. When a new request unit arrives, its HTTP method and URL path with hostname/query parts removed (*search key*) is matched against this trie.

- *Successful match.* The request is classified if its search key matches an API template, respecting parameter type validation (e.g., `<Int>` in Fig. 2(d)).
- *Failed match.* Unmatched search keys (e.g., `POST /api/other/2` in Fig. 2(b) failing against template `POST /api/other/1`) are temporarily stored as static keys, indicating new, ungeneralized URLs.

Rule-based Parameter Discovery and API Template Generalization. RESTful APIs use URL path parameters (e.g., `userId` in `/users/userId/profile`), so treating each unique URL as a separate API is infeasible, leading to API explosion and inaccurate structure. Thus, APIECHO discovers parameters and generalizes static keys into API templates. New static keys (e.g. `POST /api/other/2` in Fig. 2(c)) are compared with similar static keys:

- *Predefined parameter rules.* Rules are used to identify and extract types like numbers, UUIDs, hex strings and common file extensions (Table 11).
- *Parameter matching logic.* If two static keys share method, segment count, and differing segments conform to a predefined parameter type (e.g., `1`, `2` as integers in Fig. 2(c)), they form a template with a parameterized node (e.g., `<Int>` in Fig. 2(d)).
- *Arbitrary string parameters.* For multi-segment APIs, at most one segment can be an "arbitrary string" parameter for patternless values (e.g., usernames) if other segments are consistent.

Successfully generalized API templates (e.g., `POST /api/other/<Int>` in Fig. 2(d)) are added to the trie; all covered static keys are removed. Parameterized nodes (e.g., `<Int>`) carry type information for subsequent request matching validation.

Splitting Mechanism for Adapting to API Evolution. Web applications and their API designs evolve. For example, a parameterized path (`/items/category_id`) might become fixed paths (`/items/electronics`, `/items/books`), each with different logic, unsuited for a single parameterized template. APIECHO uses an API splitting mechanism to address this.

- *Parameter value monitoring.* APIECHO tracks observed values for parameterized segments in a recent window. If distinct values appeared consistently stay below a threshold (e.g., few fixed values recently), the segment is inferred to have solidified.
- *Performing the split*: The parameterized template is replaced. New, specific API paths (static or with

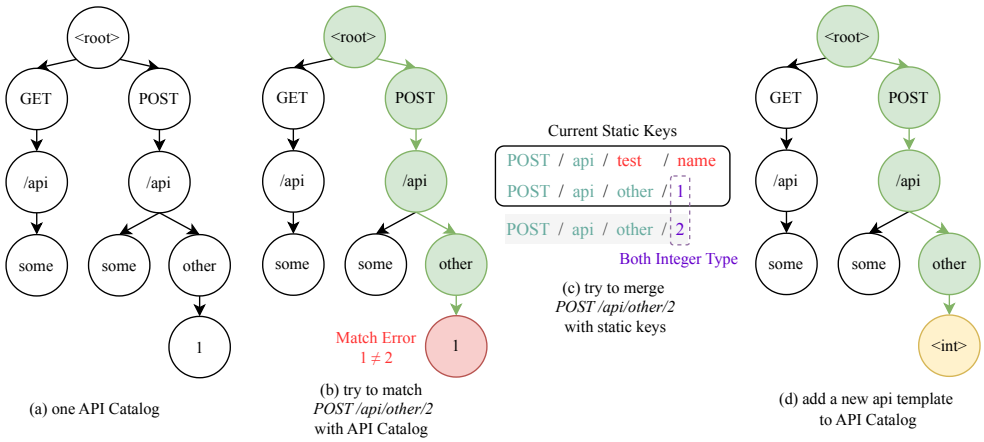

Figure 2. Example of dynamic API classification, demonstrating an new URL matched with another URL, and generalized into an API template.

deeper parameter hierarchy) based on these observed static values are added to the trie. For example, if `/items/itemId` recently only showed `itemId` as `special-A` and `special-B`, it might be split into `/items/special-A` and `/items/special-B`.

Manually configurable static key parameters. In some web applications, there are circumstances where parameters can significantly change behavior of an API (e.g. `/items/abc?action=get` queries item abc, while `/items/abc?action=update` changes it). As an workaround, APIECHO also supports a manual configuration mechanism, allowing operators to specify certain query parameter or URL segments that should be treated as static parts of the API path, thereby preventing incorrect parameterization.

This dynamic API classification, with template generalization and splitting, enables APIECHO to accurately classify request units, keeping robustness amid evolving API structures in monolithic applications.

4.3. Intra-API Anomaly Detection

APIECHO's core task, post-partitioning and API classification, is detailed intra-API anomaly detection on request behavior. APIECHO maintains a fixed-length detection window per API. When full, it detects anomalous units by comparing behavioral similarity among requests in window.

Request units (sequence of audit logs) are vectorized in two steps: first (Section 4.3.1), domain knowledge and web server operational characteristics guide rule-based extraction of core behavioral features (set-based, sequence-based, etc.). Second (Section 4.3.2), to detect behavioral deviations, these features are numerically represented by calculating their similarities, leading to an anomaly score per unit. These normalized scores determine if a unit is anomalous.

4.3.1. Feature Extraction. Similar to ReplicaWatcher [6], APIECHO extracts features covering different aspects of request processing behavior that change during intrusions. Table 12 lists all features extracted.

Syscall Features. Syscalls are fundamental program-kernel interactions [19]. During intrusions, request units likely execute uncommon syscalls (e.g., `fork` and `exec` for command execution). Therefore, the set of syscalls executed by the request unit and their categories (e.g., file or network operations) are used as features. Notably, request processing of some complex applications might involve a wide variety of syscalls, such that even under intrusion, changes of aforementioned two features might not be significant. Therefore, sequence of syscalls executed during request processing is also used as a feature to enhance detection.

Resource Access Features. Files and sockets are primary web application interaction entities. Normal request processing accesses resources within fixed ranges, while intrusions may access external files or communicate with unusual addresses to steal data, gather information, or achieve remote control. Therefore, we also extract features such as accessed file directories, number of distinct IP addresses interacted with and number of distinct port numbers used.

4.3.2. Similarity-Based Anomaly Detection. APIECHO extracts set, sequence, and numerical features, in which set and sequence features require conversion to numerical values/vectors for anomaly detection algorithms. Since our primary concern is how similar a unit is to others, rather than needing an independent vector embedding of it, we, similar to ReplicaWatcher, derive numerical representations by calculating similarities.

Feature Similarity Calculation. For a detection window $U = \{u_1, \ldots, u_{N_w}\}$ with N_w request units, we calculate similarity $S_k(F_{ki}, F_{kj}) \in [0, 1]$ between any two units u_i, u_j for their k-th feature values F_{ki}, F_{kj}.

- *Set features*. For features like syscall sets or file

directory sets, Jaccard similarity [20] is used:

$$S_J(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where $|A \cap B|$ is intersection size, $|A \cup B|$ is union size of the sets.

- *Sequence features*. For features like syscall sequences, order is crucial. Inspired by NLP n-gram models [21], we extract the n-gram frequency distribution from the original sequence. An n-gram is a subsequence of n consecutive elements. For example, for `read, open, write, read, open, close`, 2-grams include `(read, open), (open, write)`. We treat a sequence's n-gram frequency distribution as a high-dimensional vector, where dimensions are n-grams appearing in either sequence and values are their frequencies. Then, cosine similarity is applied between these vectors, serving as original sequence similarity. For vectors $\vec{x}, \vec{y}$:

$$S_C(\vec{x}, \vec{y}) = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}||_2 \cdot ||\vec{y}||_2}$$

where $\vec{x} \cdot \vec{y}$ is dot product, $||\vec{x}||_2$ is L2 norm.

Initial Anomaly Vector Construction. For each unit u_i in window U, an M-dimensional initial anomaly vector $\vec{X}_i = (x_{1i}, \ldots, x_{Mi})$ is constructed, where x_{ki} is the initial anomaly score on the k-th feature.

- *Set and sequence features.* Initial anomaly score x_{ki} is the complement of average similarity of this unit's feature with the corresponding features of all others:

$$x_{ki} = 1 - \frac{1}{N_w - 1} \sum_{j=1, j \neq i}^{N_w} S_k(F_{ki}, F_{kj})$$

where $S_k(F_{ki}, F_{kj})$ is the similarity between units u_i and u_j on the k-th feature. $x_{ki} \in [0, 1]$; closer to 0 means more similar to window average (likely normal), closer to 1 means likely anomalous.

- *Numerical features* (e.g., distinct IP count). Initial anomaly score x_{ki} is deviation from window mean $\mu_k = \frac{1}{N_w} \sum_{j=1}^{N_w} F_{kj}$:

$$x_{ki} = F_{ki} - \mu_k$$

Feature Dimension Normalization. After obtaining initial anomaly scores x_{ki}, inspired by batch normalization, APIECHO normalizes scores on each feature dimension across all units in the window. This addresses varying behavioral variability across APIs or features. For example, the difference in the set of accessed directories between two requests of a file upload API might naturally be greater than that of a simple status query API, where uniform scaling can cause over/under-sensitivity. Normalizing adjusts their distributions to similar ranges, allowing more comparable thresholds.

For the k-th feature dimension, initial anomaly scores $(x_{k1}, \ldots, x_{kN_w})$ are L2 normalized to $(x'_{k1}, \ldots, x'_{kN_w})$:

$$x'_{ki} = \frac{x_{ki}}{\sqrt{\sum_{j=1}^{N_w} (x_{kj})^2}}$$

If denominator is zero, x'_{ki} is zero. Each unit u_i then has a normalized anomaly feature vector $\vec{X'}_i = (x'_{1i}, \ldots, x'_{Mi})$.

Final Anomaly Determination. APIECHO calculates the Euclidean distance of $\vec{X'}_i$ from origin as the final comprehensive anomaly score D_i for unit u_i:

$$D_i = ||\vec{X'}_i||_2 = \sqrt{\sum_{k=1}^{M} (x'_{ki})^2}$$

A larger D_i indicates greater deviation from the collective behavior in the window, hence higher likelihood of being malicious. D_i is compared with a predefined global threshold Θ; if $D_i > \Theta$, unit u_i is anomalous.

4.4. Window Management and Baseline Adaptation

APIECHO's final key component is window management and baseline adaptation, ensuring detection accuracy and sensitivity during web application evolution. Its core involves dynamically maintaining each API's detection window, defending against coordinated poisoning, and responding to external update notifications for lightweight baseline reconstruction. This ensures APIECHO's long-term effectiveness and low operational cost in practical deployments.

4.4.1. Window Management. APIECHO maintains a fixed-size (N_w) sliding detection window per API for recent units. New units enter their API's window.

- *Standard sliding and baseline drift adaptation*: New unit u_{new} replaces oldest u_{old}, forming a new window representing current "normal behavior". Continuous sliding enables automatic adaptation to gradual behavioral drifts from code optimizations, configuration changes, load variations.
- *Coordinated poisoning attack defense and baseline protection*: After anomaly detection on the latest unit u_{N_w} (just added to the window):
  1) If u_{N_w} is normal: It remains in the window as part of the baseline. If it brings new API pattern information (e.g., new URL parameter form), the API catalog can also update.
  2) If u_{N_w} is malicious: An alert is issued. Crucially, u_{N_w} is **removed** from the window, and its structural information is **not** used to update the API catalog. This prevents malicious requests from contaminating the "normal" knowledge base, defending against coordinated poisoning where attackers try to "lower" the normal behavior standard or "train" the system to accept malicious patterns via many similar malicious requests.

4.4.2. Incremental Similarity Calculation. Naively recalculating the anomaly feature vector (all pairwise similarities, subsequent normalization) for all N_w units per window slide takes $O(N_w^2 M)$ (M features). For real-time performance in high-throughput environments, APIECHO uses incremental calculation to optimize update complexity to $O(N_w M)$.

When window $W = \{u_1, u_2, \ldots, u_{N_w}\}$ slides to $W' = \{u_2, \ldots, u_{N_w}, u_{new}\}$, initial anomaly scores x_{ki} (defined in Section 4.3.2) will be updated for units retained in the window $u_2, \ldots, u_{N_w}$ and the new unit u_{new}. For a retained unit $u_i \in \{u_2, \ldots, u_{N_w}\}$, its old sum of similarities on feature k was:

$$\hat{S}_{ki} = \sum_{u_j \in W, u_j \neq u_i} S_k(F_{ki}, F_{kj})$$

When the window updates to W', for the same unit u_i, its new sum $\hat{S}'_{ki}$ can be obtained by subtracting the similarity with the removed unit u_1 from the old sum and adding the similarity with the new unit u_{new}:

$$\hat{S}'_{ki} = \hat{S}_{ki} - S_k(F_{ki}, F_{k,u_1}) + S_k(F_{ki}, F_{k,u_{new}})$$

This is $O(1)$ per remaining unit, thus $O(N_w)$ for all $N_w - 1$ units on feature k.

For new unit u_{new}, calculating sum of similarities $\hat{S}'_{k,new}$ on feature k costs $O(N_w)$:

$$\hat{S}'_{k,new} = \sum_{u_j \in \{u_2, \ldots, u_{N_w}\}} S_k(F_{k,new}, F_{kj})$$

After new initial anomaly scores x'_{ki} (from new $\hat{S}'$ values) are updated/calculated, L2 normalization denominator $\sqrt{\sum_{u_j \in W'} (x'_{kj})^2}$ is recalculated. With all x'_{kj} for $u_j \in W'$ known, this is $O(N_w)$. Thus, for M features, update complexity per slide is $O(N_w M)$, enabling efficient continuous request processing.

4.4.3. Baseline Adaptation on Updates. Modern web applications iterate rapidly; legitimate changes can significantly shift underlying behavioral patterns. If an API's behavior changes post-update, APIECHO's existing baseline might misclassify new normal behavior, so a simple baseline reset mechanism is provided.

- When operators confirm persistent alerts are from application updates rather than attacks, they notify APIECHO through an external mechanisms.
- Upon notification, APIECHO clears detection windows for specific APIs (or all for global updates).
- Cleared windows are then refilled with new requests, building a new behavioral baseline.

This lightweight process avoids collecting extensive new logs and resource-intensive retraining (common in traditional methods), enabling rapid adaptation to post-update normal states. Moreover, CI/CD rules can be set to trigger detection windows reset when a new version of application is deployed, thus make this process runs automatically.

5. Evaluation

We designed experiments to answer the following questions:

1) How effective is APIECHO in detecting intrusions compared to existing approaches?
2) To what extent can APIECHO adapt to the diverse behavioral patterns of different APIs?
3) How robust is APIECHO against poisoning attacks?
4) How effectively APIECHO adapting to app updates?
5) What is the impact of APIECHO's key design components and parameters on its overall performance?
6) What are runtime and space overhead of APIECHO?

5.1. Evaluation Setup

Hardware Setup. Most experiments used a Linux server (Intel Xeon Platinum 8255C @2.50GHz, 128GB RAM, Ubuntu 22.04). KAIROS GPU training utilized a separate server (Intel Core i7-12700 @2.1GHz, RTX 3090 24GB GPU, 64GB RAM, Kali Linux).

Evaluation Datasets. We evaluated APIECHO across 16 scenarios (Table 1) to validate its effectiveness on diverse web applications. The scenarios are inherited from AutoLabel [28], which provides open-source datasets and reproducible dataset generation process. These scenarios covered 16 versions of 8 web applications, encompassing 10 real CVE vulnerabilities, and spanning 4 common languages (Python, NodeJS, Java, PHP). We use Sysdig [5] to collect audit logs. In each scenario, automated scripts simulated normal access behaviors (20 min/scenario), and after 10 minutes of normal traffic, attacks exploiting application vulnerabilities were launched. This generated audit logs with mixed normal and attack behaviors for experiments. Some of the 8 web applications required instrumented unit partitioning, for which we instrumented:

1) *FastAPI* [29] (Python asynchronous framework): For `python-demo`, `pgadmin`, and other Python `asyncio`-based ASGI frameworks. We added a FastAPI middleware to inject a request ID into Python's asyncio-native context at the start of the request flow, allowing it to propagate with coroutine derivations; then, we used monkey patching to modify the `asyncio.Handle` type, outputting the request ID from its context before and after it is awakened for execution. This instrumentation is done without altering source code.
2) *Libuv* [30]: For `juice-shop`, `mongo-express`, and other NodeJS web applications. We allocated unique IDs to various handles (including `uv_handle_t`, `uv__work`, `uv__io_t`, etc.) upon creation and output them to logs; delimiter logs were output before and after their callbacks to determine the executing handles and their derivation relationships.
3) *Java*: For applications like `solr` and `OFBiz` using a thread pool model. We used a Java agent

for instrumentation, outputting unique IDs for all `Callable` or `Runnable` objects upon creation, and delimiter logs before and after their execution. This instrumentation is done without altering source code.

Labeling Method. For result evaluation, we use AutoLabel [28] to generate ground-truth labels on log event level. With full control of the attacker, it leverages execution partition and flags injection to track malicious behaviors, thus make sure the label is accurate. Notice that information that AutoLabel used for tracking attacks is transparent to APIECHO and comparative methods.

TABLE 1. SCENARIOS FOR GENERATING DATASETS FOR EXPERIMENTS

#	App Name	Framework or Language	Weakness	Version	Source
1	python-demo	FastAPI (Python)	Injection	2 versions	self-designed
2	pgadmin	Flask (Python)	Injection	6.16 7.6	CVE-2022-4223 CVE-2023-5002
3	juice-shop	NodeJS	Security Misconfiguration, Injection	17.1.1[1]	OWASP Juice Shop
4	mongo-express	NodeJS	Broken Access Control /	0.53.0 1.0.2	CVE-2019-10758 No Attack
5	gitlist	PHP	Injection /	0.6.0 1.0.0	CVE-2018-1000533 No Attack
6	joomla	PHP	Software and Data Integrity Failures Broken Access Control	3.7.0 4.2.7	CVE-2015-8562 CVE-2023-23752
7	Solr	Java	Injection	8.1.1 8.2.0	CVE-2019-0193 CVE-2019-17558
8	OFBiz	Java	Software and Data Integrity Failures SSRF	17.12.01 18.12.15	CVE-2020-9496 CVE-2024-45507

[1] Updated version is made by altering source code.

5.2. Overall Detection Performance

This section evaluates the overall intrusion detection performance of APIECHO against three baselines: ReplicaWatcher, ProvDetector, and KAIROS. The methods differ in native alert units: APIECHO uses request units, KAIROS uses attacking subgraphs, and ReplicaWatcher and ProvDetector uses time windows. For fair comparison across methods, we standardized evaluation at the event level, marking all audit events in an alert unit as malicious. We recorded attack recall and detection accuracy, using their harmonic mean (score) for performance comparison. Experiments were repeated 20 times per scenario for average metrics.

Implementation of Baseline Methods:

- Benign-only audit logs were collected per scenario to train ProvDetector and KAIROS.
- Mixed logs (labeled malicious/benign) were used by all four methods for detection.
- For ReplicaWatcher, multiple replica log files were collected concurrently and segmented into time windows for input. To ensure a fair comparison that favors ReplicaWatcher's replica-based detection assumption, we designed the input such that attack traffic was confined to a single replica per time window, while all other replicas maintained purely benign behavior. This setup maximizes ReplicaWatcher's ability to detect deviations by providing a clear majority of normal replicas for comparison.

Experimental Results. As shown in Table 2, APIECHO detected more than 90% of attacks in all scenarios, with benign event false positives less than 1%. ReplicaWatcher, also using self-comparison, performed significantly worse than APIECHO across all datasets in both attack detection proportion and benign event accuracy. As discussed in Chapter 2, ReplicaWatcher assumes functional singularity and behavioral homogeneity among replicas. This assumption fails for monolithic web applications at the whole-replica comparison level, explaining its suboptimal performance. Conversely, APIECHO uses API unit partitioning for finer-grained self-comparison. Results show the homogeneity assumption holds largely at this API level, yielding superior performance. KAIROS's overall detection was slightly inferior to APIECHO, though with fewer false positives in the `joomla` scenario. However, KAIROS needs benign-only logs for GNN model training, which took several hours to over a day on our hardware. ProvDetector had shorter training times than KAIROS but worse detection performance. In summary, results show APIECHO achieves superior detection performance in web server scenarios without pre-training, outperforming the evaluated training-based methods.

5.3. Case Study

This case study analyzes behavioral differences between APIs of the same web application and whether APIECHO correctly handles them.

Table 3 details behavioral features for two `python-demo` APIs: the `/solve` API, which parses and executes expressions, and the `/status` API, which checks the application's running status. Their normal behaviors are in the first two columns. The `/solve` API, vulnerable to arbitrary command execution, exhibits behavior under attack (third column). The table shows the attacked `/solve` API's behavior, differing from its normal state, resembles the normal `/status` API behavior. Figure 3(a) (PCA of aggregated anomaly scores) shows attacked `/solve` data points near normal `/status` points, indicating attack masking by inter-API behavioral diversity when not differentiating APIs. This shows that for monolithic web applications, methods without API differentiation like ReplicaWatcher suffer from interference where one API's normal behavior (e.g., `/status`)

TABLE 2. OVERALL DETECTION PERFORMANCE OF APIECHO AND BASELINE METHODS

#	App	APIECHO			ReplicaWatcher			KAIROS			ProvDetector		
		Recall%	Accu.%	Score	Recall%	Accu.%	Score	Recall%	Accu.%	Score	Recall%	Accu.%	Score
1	python-demo	100.0%	99.8%	**0.9990**	50.0%	98.5%	0.7952	70.0%	99.9%	0.9028	100.0%	92.2%	0.9594
2	pgadmin	100.0%	99.9%	**0.9996**	60.0%	47.3%	0.5799	100.0%	98.5%	0.9924	70.0%	57.1%	0.6747
3	juice-shop	95.0%	99.2%	**0.9831**	95.0%	21.5%	0.3523	45.0%	93.4%	0.7458	5.0%	99.9%	0.1739
4	mongo-express	90.0%	99.9%	**0.9725**	30.0%	65.1%	0.5401	90.0%	86.1%	0.9019	70.0%	56.0%	0.6669
5	gitlist	95.0%	99.7%	0.9855	15.0%	90.5%	0.4050	100.0%	98.7%	**0.9935**	90.0%	18.3%	0.3067
6	joomla	100.0%	99.1%	0.9954	70.0%	51.1%	0.6304	100.0%	99.9%	**0.9994**	100.0%	94.8%	0.9734
7	Solr	100.0%	99.0%	**0.9950**	60.0%	47.5%	0.5816	100.0%	94.6%	0.9723	10.0%	99.9%	0.3076
8	OFBiz	100.0%	99.5%	**0.9975**	52.6%	55.5%	0.6149	94.7%	88.3%	0.9257	26.3%	99.3%	0.5868
	Average	97.5%	99.5%	**0.9910**	54.1%	59.6%	0.5624	87.5%	94.9%	0.9292	58.9%	77.2%	0.5812

TABLE 3. BEHAVIOR FEATURES OF SOME APIS IN PYTHON-DEMO

Feature	/solve	/status	/solve (attacked)
Syscalls	recvfrom, sendto, write	execve, clone, read, write, ...(15)	execve, vfork, read, write, ...(14)
Syscall Categories	File, net	File, net, process	File, net, process
Accessed Directories	/tmp/logs	/dev, /etc, /usr/lib/locale, ...	/tmp/logs, /etc, /usr/lib/locale, ...
Used Fd Types	ipv4, file	ipv4, file, pipe, pidfd	ipv4, file
Executed Processes	python3.12	python3.12, dash, grep	python3.12, touch

masks another's anomaly (e.g., attacked `/solve`). APIECHO addresses this via API classification (from API unit partitioning). When similarity is calculated within the segregated behaviors of two APIs respectively, the PCA visualization (Figure 3(b)) shows a clear separation. This highlights that isolating API behaviors makes the `/solve` anomaly distinct, proving API-specific analysis effectiveness.

While API classification mitigates inter-API interference, intra-API behavioral variability poses another challenge. Normal behavior variation *within* an API can differ significantly across APIs, often due to functionality. This varying intra-API variability makes a single global anomaly detection threshold ineffective, as a threshold suitable for one API might be too lenient or too strict for another. For instance (Figure 3(c)), `OFBiz`'s `rainbowstone` API has a normal behavior point (upper right dot) distinct from its other normal behaviors. A threshold accommodating this `rainbowstone` point would miss an actual attack on the `xmlrpc` API (lower right red cross). APIECHO addresses this by independently normalizing each API's anomaly score vectors. This standardizes each API's anomaly score distribution, making them uniform despite inherent intra-API variability. Consequently, a single detection threshold hyperparameter applies effectively across all APIs, enabling adaptive detection. Figure 3(d) shows normalized `rainbowstone` normal behavior points cluster tightly, enabling successful `xmlrpc` attack detection while correctly classifying `rainbowstone`'s diverse normal behaviors.

5.4. Robustness against Poisoning Attacks

This section evaluates APIECHO's robustness against coordinated or progressive poisoning attacks.

First, for coordinated poisoning attacks, we prepared log datasets mirroring RQ1's setup but launched continuous attacks at a frequency higher than normal access after an initial period. These poisoned logs were evaluated using RQ1's procedure. We compared APIECHO's performance with ReplicaWatcher, analyzing score changes relative to non-poisoned RQ1 results. To isolate the contribution of APIECHO's anomalous unit discarding mechanism, we included APIECHO-NoThrow omitting this mechanism for comparison.

Experimental Results. Table 4 shows results, with ΔScore indicating score change from RQ1, whose positive values indicate raising scores. APIECHO outperformed ReplicaWatcher and APIECHO-NoThrow in all four scenarios. Notably, APIECHO's score improved across all scenarios versus RQ1 due to the increased volume of attack allowing a tighter decision threshold, enabling tuning to enhance accuracy while maintaining recall. Although this could benefit all methods, ReplicaWatcher and APIECHO-NoThrow scores significantly dropped in some scenarios. ReplicaWatcher's performance decline under coordinated poisoning stems from its core detection principle: identifying outliers via self-comparison within time window batches. With more than 50% attack behaviors in our simulations, it often misclassified attacks as benign. To mitigate this, APIECHO uses a sliding window with an anomalous unit discarding mechanism, anchoring benign baseline by dynamically accumulating verified units. Results support that APIECHO-NoThrow (without this mechanism) showed performance decline in several scenarios,

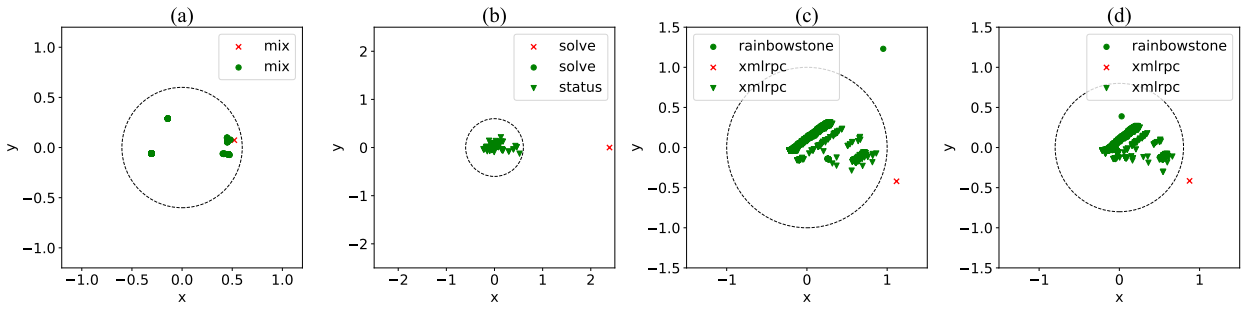

Figure 3. Visualization of unit similarities in case study. (Red crosses represents malicious units and green dots represents benign units)

demonstrating the mechanism's efficacy.

To further assess APIECHO's robustness, we simulated progressive poisoning attacks to determine if a sophisticated, gradual approach could evade detection. These attacks began with high-frequency operations mimicking benign behavior (e.g., innocuous code execution in a code injection context), then gradually introduced more overtly malicious operations with increasing prevalence. The rationale was that if initial, subtle anomalies evaded detection and weren't discarded by APIECHO's anomalous unit discarding mechanism, they could contaminate the benign behavior reference model within the detection window, potentially enhancing subsequent, more harmful attacks.

We conducted progressive poisoning on the `OFBiz` scenario, where attacks involve injecting Groovy code into XML to execute shell commands. Our staged progressive attack (Table 5) included: innocuous printing, reading normally accessed files, accessing a payload-relevant directory, and finally, the original malicious payload. Stage transitions aligned with APIECHO's sliding window (100), allowing a stage's units to populate the detection window if not immediately discarded before the next stage commenced. Despite this strategy, APIECHO maintained 99.9% benign event accuracy, indicating its anomaly detection is sensitive enough to detect these progressive attacks at some stage and mitigate them via its anomalous unit discarding process.

5.5. Adaptation to Application Updates

In this section, we evaluate APIECHO's ability to maintain detection performance after application updates. For each application scenario, we create a new application version by introducing new vulnerabilities or new normal behaviors. We collect audit logs from the new version of the application and then append them to the old version logs obtained in RQ1, thereby simulating a scenario where the application is dynamically updated while the real-time intrusion detection is running. We compare the detection performance of detection methods on the concatenated audit logs. Notably, for KAIROS and ProvDetector, we only use the normal behavior logs from the old version for training.

Experimental Results. Table 6 shows that, compared to RQ1, baselines had significantly declined detection performance on multiple datasets. KAIROS and ProvDetector, trained on old version, struggled to fit new version normal behavior, causing their performance degradation. ReplicaWatcher adapts gradually via its updating time windows, but if a time window simultaneously contains normal behaviors from both versions, one might be misjudged as anomalous, impacting performance. However, on some datasets, baselines' scores did not decrease significantly or even improved. Possible reasons include: 1) small inter-version behavioral differences allowing old models to fit new versions; and 2) new version attacks being more obvious to baseline models. Unlike baselines, APIECHO showed no significant performance decline in any scenario. APIECHO clears its detection window upon receiving an update notification. Results show this mechanism helps APIECHO refit new version behavior, and thus improve detection performance.

Whereas previous update experiment allowed APIECHO time to learn new version behaviors before attacks, we further tested its window-clearing mechanism by immediate post-update attacks. Updated version logs from the `python-demo` scenario is modified, placing attack behaviors at the very beginning. Theoretically, APIECHO initiates detection only after its detection window populates with subsequent benign units, ensuring accuracy. Results showed APIECHO successfully detected all attacks, and benign event accuracy remained undiminished. This demonstrates APIECHO's robustness to immediate post-update attacks, though alerts might be slightly delayed while its detection window accumulates sufficient requests.

5.6. Ablation Study and Parameter Sensitivity

This section evaluates the impact of specific APIECHO mechanisms on detection performance via ablation studies and parameter tuning.

Normalization Mechanism. APIECHO uses normalization for diverse intra-API behavioral variations, enabling a single detection threshold across APIs.

TABLE 4. PERFORMANCE OF APIECHO AND BASELINE METHODS AGAINST POISONING ATTACKS

#	App	APIECHO				APIECHO-NoThrow				ReplicaWatcher			
		Recall%	Accu.%	Score	ΔScore	Recall%	Accu.%	Score	ΔScore	Recall%	Accu.%	Score	ΔScore
1	python-demo	100.0%	99.9%	**0.9995**	0.0005	100.0%	99.9%	**0.9995**	0.0005	55.0%	51.4%	0.5962	-0.1990
3	juice-shop	100.0%	84.2%	**0.9142**	-0.0689	100.0%	75.5%	0.8604	-0.1227	55.0%	60.8%	0.6549	0.3026
7	Solr	100.0%	99.9%	**0.9995**	0.0045	100.0%	95.4%	0.9765	-0.0185	95.0%	55.2%	0.7047	0.1231
8	OFBiz	100.0%	99.9%	**0.9995**	0.0020	100.0%	99.7%	0.9985	0.0010	75.0%	83.6%	0.8464	0.2315
	Average	100.0%	96.0%	**0.9782**	-0.0155	100.0%	92.6%	0.9587	-0.0349	70.0%	62.8%	0.7006	0.1146

TABLE 5. PAYLOADS OF PROGRESSIVE POISONING EXPERIMENTS

Epoch	Groovy Code Payload
1-200	`println 'Hello world!';`
201-300	`new File('/usr/src/ofbiz/...').text;`
301-400	`new File('/tmp').eachFileRecurse { file -> println file }`
401+	`touch /tmp/success'.execute();`

Section 5.3 qualitatively demonstrated its effectiveness. To quantify its impact, we evaluated APIECHO-NoNormalize (normalization disabled) in selected scenarios. Results in Table 7 show disabling normalization has a more pronounced negative impact on applications with complex functionalities and diverse API behaviors like `OFBiz` and `Solr`, and less impact on simpler ones. These findings align with Section 5.3 and confirm normalization's benefit, especially with high API behavioral diversity.

Sequential Features. APIECHO incorporates syscall sequential features for stealthier attack detection. We evaluated performance with these disabled (APIECHO-NoSeq), and the results is shown in Table 7. Unlike disabling normalization, removing sequential features caused more substantial performance degradation for simpler applications like `python-demo` and `juice-shop`. This might be because their normal request processing involves shorter, consistent syscall sequences, making attack-induced novel n-grams or patterns highly distinguishable. Disabling these features removes a critical detection source. Overall, results affirm sequential features' positive contribution to detection.

API Classification. APIECHO uses API classification (Section 5.3) for inter-API normal behavior differences. Thus, API classification errors could negatively impact detection. We found API classification errors via parameter type matching are very rare. Errors mostly stem from the arbitrary string parameter rule which, lacking type requirements, might merge APIs differing only by a URL segment (treated as a parameter). To evaluate this, we manually introduced API classification errors in `OFBiz` scenario. We exported the generated API catalog and simulated errors by merging groups of APIs fitting the description. After importing the erroneous catalog, we observed changes in correctly identified benign event proportion, as shown in Table 8. Accuracy dropped from 99.5% to 99.3% with two merged APIs, and merging up to 24 APIs caused no further decrease in this setup. This suggests API classification errors in most evaluated scenarios do not drastically degrade accuracy. Furthermore, reliable parameter type matching and the arbitrary string parameter rule's restriction, which allows at most one of such parameter, make severe API classification errors unlikely.

Sliding Window Size. APIECHO uses a sliding window for self-comparison. In each round, the window includes a new unit, and similarity among all units, including the new one, is calculated for anomaly detection. Theoretically, a larger window reduces a new (potentially malicious) unit's proportional representation, diminishing its distorting influence on the normal behavior baseline, which should benefit detection. However, computational overhead per new unit increases linearly with window size, as analyzed in Section 4.4.2. We experimented in the `Solr` scenario, varying window size while keeping other parameters constant, observing benign event identification changes (Table 9). A window size of 10 showed a significant accuracy drop versus default (100). At size 50, accuracy was almost identical to default. A size of 500 was needed for slight improvement. Considering the overhead of larger windows (detailed in Section 5.7), the default size of 100 is appropriate.

5.7. Performance Overhead

This section evaluates APIECHO's performance overhead. Theoretically, sliding window size is the configurable parameter with the most significant impact. As detailed in Section 4.4.2, APIECHO's computational overhead for detecting each new unit scales linearly with window size, and so does their maximum memory footprint. We evaluated the relationship between sliding window size and APIECHO's memory consumption and log detection throughput. Experiments used the `OFBiz` scenario (many APIs), where independent sliding windows per API make overall memory consumption changes more pronounced.

Experimental Results. Table 10 details average memory consumption and log throughput (events/sec) across varying window sizes. As window size increases, memory consumption rises and throughput falls. However, the memory increase largely plateaus beyond a window size of 100. While throughput decreases, its rate of decline also diminishes with larger window sizes.

TABLE 6. PERFORMANCE OF APIECHO AND BASELINE METHODS IN UPDATING SCENARIOS

#	APIECHO				ReplicaWatcher				KAIROS				ProvDetector			
	Recall%	Accu.%	Score	Δ	Recall%	Accu.%	Score	Δ	Recall%	Accu.%	Score	Δ	Recall%	Accu.%	Score	Δ
1	100.00%	99.80%	**1.00**	0.00	80.00%	57.00%	0.70	-0.10	100.00%	99.10%	**1.00**	0.09	85.00%	54.10%	0.68	-0.28
2	90.00%	100.00%	**0.97**	-0.03	40.00%	68.00%	0.62	0.04	85.00%	95.30%	0.94	-0.06	75.00%	28.80%	0.43	-0.24
3	87.50%	95.70%	**0.95**	0.01	90.00%	59.30%	0.73	0.38	75.00%	92.70%	0.89	0.15	17.50%	99.90%	0.46	0.29
4[1]	90.00%	98.90%	**0.97**	-0.01	30.00%	13.00%	0.20	-0.34	90.00%	55.70%	0.70	-0.20	40.00%	76.20%	0.62	-0.05
5[1]	100.00%	99.80%	**1.00**	0.01	40.00%	85.30%	0.68	0.28	100.00%	87.50%	0.93	-0.06	0.00%	95.70%	0.00	-0.31
6	100.00%	99.10%	**1.00**	0.00	60.00%	46.30%	0.57	-0.06	50.00%	99.60%	0.80	-0.20	90.00%	70.90%	0.81	-0.16
7	100.00%	99.40%	**1.00**	0.00	85.00%	54.00%	0.68	0.10	100.00%	99.00%	**1.00**	0.02	50.00%	45.60%	0.54	0.23
8	100.00%	99.40%	**1.00**	0.00	11.10%	96.30%	0.33	-0.28	100.00%	92.70%	0.96	0.04	10.80%	98.50%	0.33	-0.26
Avg.	95.9%	99.0%	**0.98**	0.00	54.5%	59.9%	0.56	0.00	87.5%	90.2%	0.90	-0.03	45.4%	71.2%	0.48	-0.10

[1] No attack in new version of this application.

TABLE 7. ABLATION STUDY RESULTS FOR APIECHO COMPONENTS

#	App	APIECHO			APIECHO-NoNormalize				APIECHO-NoSeq			
		Recall%	Accu.%	Score	Recall%	Accu.%	Score	ΔScore	Recall%	Accu.%	Score	ΔScore
1	python-demo	100.0%	99.8%	**0.9990**	100.0%	99.8%	**0.9990**	0.0000	20.0%	99.9%	0.4999	-0.4991
3	juice-shop	95.0%	99.2%	**0.9831**	85.0%	73.7%	0.8180	-0.1651	25.0%	99.9%	0.5713	-0.4118
7	Solr	100.0%	99.0%	**0.9950**	100.0%	93.2%	0.9648	-0.0302	100.0%	98.4%	0.9919	-0.0030
8	Ofbiz	100.0%	99.5%	**0.9975**	100.0%	83.1%	0.9077	-0.0898	100.0%	99.1%	0.9955	-0.0020
	Average	98.8%	99.4%	**0.9936**	96.3%	87.5%	0.9224	-0.0713	61.3%	99.3%	0.7646	-0.2290

TABLE 8. ACCURACY IN OFBIZ SCENARIO WITH API CLASSIFICATION ERRORS

# Merged APIs	0	2	8	12	24
Accuracy%	99.5%	99.3%	99.3%	99.3%	99.3%

TABLE 9. ACCURACY IN SOLR SCENARIO WITH DIFFERENT WINDOW SIZES

Window Size	10	50	100	200	500
Accuracy%	95.0%	98.3%	98.3%	98.3%	98.4%

Considering these overheads and Section 5.6's findings (accuracy stabilizes for window sizes 50-200), a window size of 100 effectively balances detection accuracy and resource utilization.

TABLE 10. MEMORY CONSUMPTION AND LOG THROUGHPUT OF APIECHO WITH DIFFERENT WINDOW SIZES

Window Size	10	50	100	200	500
Memory (GB)	6.45	6.70	6.72	6.72	6.72
Log Throughput	13026	12680	12041	11990	11842

6. Limitation and Discussion

1) **Cold Start Problem.** Upon initial deployment or after a reset, APIECHO requires a warm-up period to build its API Catalog and populate detection windows with sufficient normal request units. Until an adequate baseline is established, its anomaly detection is limited, particularly for low-traffic or infrequently invoked APIs, prolonging this "cold start" phase. This problem can be mitigated by involving canary release: the new version is released first in a relatively trusted environment, in which APIECHO does its warm-up and gets ready for a full release. Also, CI/CD could be leveraged to automatically fill the detection window with requests collected in testing. Both of these tools are widely used in production.

2) **Reliance on External Notifications for Adapting to API Changes.** When legitimate, significant shifts in API behavior occur (e.g., due to application upgrades), APIECHO currently relies on external notifications to trigger baseline resets. This is due to the challenge of autonomously distinguishing legitimate changes from sophisticated, slow poisoning attacks. However, once notified, APIECHO's adaptation is automated and more lightweight than traditional retraining. Integration with CI/CD systems could automate these notifications.

3) **Potential Diversity of Internal Behavior within the Same API.** APIECHO assumes high behavioral similarity for legitimate requests to the same API endpoint. However, some complex APIs may execute vastly different internal logic based on input parameters (e.g., a generic data processing API with "read," "write," or "transform" operations). This inherent diversity, even with per-API normalization, can widen the "normal" behavioral profile, potentially reducing sensitivity to subtle anomalies for such versatile APIs. Our design includes a manual override for such cases, which allows operators manually define a set of parameters that should be treated as part of static API URL. But, how to automatically analyze and determine the appropriate parameters set that have impact on the API logic remains a difficult problem.

7. Related Work

Anomaly Detection based on Audit Logs. Research includes rule-based and machine learning-based methods. Rule-based systems (e.g., HOLMES [31], MORSE [32], CONAN [33] and P-Gaussian [34]) offer low false positives and interpretability but struggle with novel attacks. Machine learning methods (e.g., GNN-based [7], [8], [10], [11], [35]–[38] and RNN-based [39]) detect unknown attacks but require resource-intensive retraining after application updates, hampered by difficulty obtaining clean post-update data. ReplicaWatcher [6], a training-free self-comparison approach for microservices, identifies anomalies by comparing inter-replica behavioral similarity, avoiding retraining. However, its applicability to monolithic applications is limited by functional heterogeneity, coarse detection granularity, lack of adaptive baselines, and vulnerability to coordinated poisoning. APIECHO addresses these issues with a novel API-level self-comparison strategy, inheriting the training-free benefit while overcoming limitations through fine-grained analysis, enhancing adaptability for monolithic applications.

Unit Partitioning. Precise analysis of individual web request behavior requires accurately segmenting mixed system-level audit logs, challenged by "dependency explosion" in long-running, concurrent web servers [15], [40]–[46]. Early work like BEEP [40] introduced partitioning for provenance. Subsequent methods like MPI [42], OmegaLog [44] and WinLog [47] made improvements but had limitations with modern asynchronous models. APIECHO leverages the unit partitioning technique from TeSec [15], which comprehensively handles various asynchronous execution models (multi-process, multi-thread, thread pools, event loops) in web applications. TeSec automatically partitions low-level syscalls to specific web requests, enabling the fine-grained request behavior extraction crucial for APIECHO.

API Classification. API identification methods vary. Some infer APIs from browser behavior (e.g., Akita Software Chrome Extension [48]), others use existing specifications or patterns for probing (e.g., ZAP [49], Kiterunner [50]). A third category employs passive network traffic monitoring (e.g., Cloudflare's API gateway [51], Akto [14]). APIECHO's API classification engine is inspired by passive discovery approaches (similar to Akto) and introduces a novel trie-based implementation for API matching and classification, with mechanisms to adapt to evolving APIs.

8. Conclusion

We propose APIECHO, a novel, training-free web server intrusion detection method. APIECHO enhances detection accuracy and robustness for monolithic applications by comparing individual requests targeting the same API, rather than entire replicas. Its design incorporates dynamic API classification, fine-grained request behavior extraction via unit partitioning, per-API adaptive normalization, and an anti-poisoning sliding window update mechanism. These features allow APIECHO to operate without extensive training data, improve applicability to monolithic systems, offer finer detection granularity, utilize adaptive comparison benchmarks, and effectively resist coordinated poisoning attacks. Experiments using 16 real-world web applications and vulnerabilities demonstrated that APIECHO adapts to application updates without retraining, exhibits strong resilience against coordinated poisoning, and outperforms the state-of-the-art in both training-free and pre-training-based methods. APIECHO achieved attack recall rates exceeding 90% and benign event detection accuracy over 99%, with low overhead, processing more than 12000 log events per second while consuming less than 7GB memory. Our implementation has been made open source.

Acknowledgments

This work was partially supported by the NSFC (No. 6212780016), the Guangdong S&T Programme (No. 2024B0101030002), the Ministry of Industry and Information Technology of China, the National Key Research and Development Program of China (No. 2023YFB3307500), and the Science and Technology Innovation Project of Hunan Province (No. 2023RC4014).

References

[1] O. Foundation, "OWASP Top Ten | OWASP Foundation." [Online]. Available: https://owasp.org/www-project-top-ten/

[2] M. Siwach and D. S. Mann, "Anomaly Detection for Web Log Data Analysis: A Review," *JOURNAL OF ALGEBRAIC STATISTICS*, vol. 13, no. 1, pp. 129–148, May 2022, number: 1. [Online]. Available: https://publishoa.com/index.php/journal/article/view/68

[3] M. Jensen, N. Gruschka, and R. Herkenhöner, "A survey of attacks on web services," *Computer Science - R&D*, vol. 24, pp. 185–197, 11 2009.

[4] "Falco: container native runtime security." [Online]. Available: https://falco.org/

[5] "Sysdig Secure." [Online]. Available: https://sysdig.com/wp-content/uploads/2019/03/sysdig-secure-compliance-feature-brief.pdf

[6] A. El Khairi, M. Caselli, A. Peter, and A. Continella, "ReplicaWatcher: Training-less Anomaly Detection in Containerized Microservices," in *Proceedings 2024 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2024. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2024-286-paper.pdf

[7] Z. Cheng, Q. Lv, J. Liang, Y. Wang, D. Sun, T. Pasquier, and X. Han, "KAIROS: Practical Intrusion Detection and Investigation using Whole-system Provenance," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2023. [Online]. Available: http://arxiv.org/abs/2308.05034

[8] Q. Wang, W. U. Hassan, D. Li, K. Jee, X. Yu, K. Zou, J. Rhee, Z. Chen, W. Cheng, C. A. Gunter, and H. Chen, "You Are What You Do: Hunting Stealthy Malware via Data Provenance Analysis," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24167.pdf

[9] J. Zengy, X. Wang, J. Liu, Y. Chen, Z. Liang, T.-S. Chua, and Z. L. Chua, "SHADEWATCHER: Recommendation-guided Cyber Threat Analysis using System Audit Records," in *2022 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2022, pp. 489–506. [Online]. Available: https://ieeexplore.ieee.org/document/9833669/

[10] S. Wang, Z. Wang, T. Zhou, H. Sun, X. Yin, D. Han, H. Zhang, X. Shi, and J. Yang, "THREATRACE: Detecting and Tracing Host-Based Threats in Node Level Through Provenance Graph Learning," *IEEE Transactions on Information Forensics and Security*, vol. 17, pp. 3972–3987, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9899459/

[11] X. Han, T. Pasquier, A. Bates, J. Mickens, and M. Seltzer, "Unicorn: Runtime Provenance-Based Detector for Advanced Persistent Threats," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24046.pdf

[12] W. U. Hassan, M. Lemay, N. Aguse, A. Bates, and T. Moyer, "Towards Scalable Cluster Auditing through Grammatical Inference over Provenance Graphs," in *Proceedings 2018 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2018. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_07B-1_Hassan_paper.pdf

[13] Y. Xie, D. Feng, Z. Tan, and J. Zhou, "Unifying intrusion detection and forensic analysis via provenance awareness," *Future Generation Computer Systems*, vol. 61, pp. 26–36, Aug. 2016. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167739X16300188

[14] "akto-api-security/akto." [Online]. Available: https://github.com/akto-api-security/akto

[15] R. Wang, Y. Peng, Y. Sun, X. Zhang, H. Wan, and X. Zhao, "TeSec: Accurate Server-side Attack Investigation for Web Applications," in *2023 IEEE Symposium on Security and Privacy (SP)*, May 2023, pp. 2799–2816, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/document/10179402/?arnumber=10179402

[16] R. Paccagnella, P. Datta, W. U. Hassan, A. Bates, C. Fletcher, A. Miller, and D. Tian, "Custos: Practical tamper-evident auditing of operating systems using trusted execution," in *Network and distributed system security symposium*, 2020.

[17] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy {Whole-System} provenance for the linux kernel," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 319–334.

[18] Paul E. Black, "trie," Feb. 2009. [Online]. Available: https://xlinux.nist.gov/dads/HTML/trie.html

[19] Geeksforgeeks, "Introduction of System Call," 2019, section: Misc. [Online]. Available: https://www.geeksforgeeks.org/introduction-of-system-call/

[20] M. Levandowsky and D. Winter, "Distance between sets," *Nature*, vol. 234, no. 5323, pp. 34–35, 1971.

[21] C. E. Shannon, "The redundancy of english," in *Cybernetics; Transactions of the 7th Conference, New York: Josiah Macy, Jr. Foundation*, 1951, pp. 248–272.

[22] "pgadmin." [Online]. Available: https://www.pgadmin.org/

[23] "Owasp juice shop." [Online]. Available: https://owasp.org/www-project-juice-shop/

[24] "mongo-express." [Online]. Available: https://github.com/mongo-express/mongo-express

[25] "Gitlist." [Online]. Available: https://gitlist.org/

[26] "Apache solr." [Online]. Available: https://solr.apache.org/

[27] "Apache ofbiz." [Online]. Available: https://ofbiz.apache.org/

[28] Y. Peng, T. Zhang, J. Lai, Y. Zhang, Y. Wu, H. Wan, and X. Zhao, "{AutoLabel}: Automated {Fine-Grained} log labeling for cyber attack dataset generation," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 547–566.

[29] "Fastapi." [Online]. Available: https://fastapi.tiangolo.com/

[30] "Libuv." [Online]. Available: https://github.com/libuv/libuv/

[31] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-Time APT Detection through Correlation of Suspicious Information Flows," in *2019 IEEE Symposium on Security and Privacy (SP)*, May 2019, pp. 1137–1152, iSSN: 2375-1207. [Online]. Available: https://ieeexplore.ieee.org/document/8835390/?arnumber=8835390

[32] M. N. Hossain, S. Sheikhi, and R. Sekar, "Combating Dependence Explosion in Forensic Analysis Using Alternative Tag Propagation Semantics," in *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2020, pp. 1139–1155. [Online]. Available: https://ieeexplore.ieee.org/document/9152772/

[33] C. Xiong, T. Zhu, W. Dong, L. Ruan, R. Yang, Y. Cheng, Y. Chen, S. Cheng, and X. Chen, "Conan: A Practical Real-Time APT Detection System With High Accuracy and Efficiency," *IEEE Transactions on Dependable and Secure Computing*, vol. 19, no. 1, pp. 551–565, Jan. 2022, conference Name: IEEE Transactions on Dependable and Secure Computing. [Online]. Available: https://ieeexplore.ieee.org/document/8979384/?arnumber=8979384

[34] Y. Xie, Y. Wu, D. Feng, and D. Long, "P-Gaussian: Provenance-Based Gaussian Distribution for Detecting Intrusion Behavior Variants Using High Efficient and Real Time Memory Databases," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8935406/

[35] A. Goyal, G. Wang, and A. Bates, "R-CAID: Embedding Root Cause Analysis within Provenance-based Intrusion Detection," in *2024 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2024, pp. 3515–3532. [Online]. Available: https://ieeexplore.ieee.org/document/10646671/

[36] X. Han, X. Yu, T. Pasquier, D. Li, J. Rhee, J. Mickens, M. Seltzer, and H. Chen, "SIGL: Securing Software Installations Through Deep Graph Learning," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[37] F. Liu, Y. Wen, D. Zhang, X. Jiang, X. Xing, and D. Meng, "Log2vec: A heterogeneous graph embedding based approach for detecting cyber threats within enterprise," in *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*, 2019, pp. 1777–1794.

[38] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. Stoller, and V. Venkatakrishnan, "{SLEUTH}: Real-time attack scenario reconstruction from {COTS} audit data," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 487–504.

[39] M. Du, F. Li, G. Zheng, and V. Srikumar, "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. Dallas Texas USA: ACM, Oct. 2017, pp. 1285–1298. [Online]. Available: https://dl.acm.org/doi/10.1145/3133956.3134015

[40] K. H. Lee, X. Zhang, and D. Xu, "High Accuracy Attack Provenance via Binary-based Execution Partition," in *Proceedings 2013 Network and Distributed System Security Symposium*, 2013. [Online]. Available: https://www.ndss-symposium.org/ndss2013/ndss-2013-programme/high-accuracy-attack-provenance-binary-based-execution-partition/

[41] L. Yu, S. Ma, Z. Zhang, G. Tao, X. Zhang, D. Xu, V. E. Urias, H. W. Lin, G. F. Ciocarlie, V. Yegneswaran, and A. Gehani, "Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation," *NDSS*, 2021.

[42] S. Ma, J. Zhai, X. Zhang, F. Wang, K. H. Lee, and D. Xu, "MPI: Multiple Perspective Attack Investigation with Semantics Aware Execution Partitioning," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ma

[43] M. Alhanahnah, S. Ma, A. Gehani, G. F. Ciocarlie, V. Yegneswaran, S. Jha, and X. Zhang, "autoMPI: Automated Multiple Perspective Attack Investigation With Semantics Aware Execution Partitioning," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2761–2775, Apr. 2023. [Online]. Available: https://ieeexplore.ieee.org/document/9996963/

[44] W. U. Hassan, M. A. Noureddine, P. Datta, and A. Bates, "OmegaLog: High-Fidelity Attack Investigation via Transparent Multi-layer Log Analysis," in *Proceedings 2020 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2020. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2020/02/24270.pdf

[45] J. Zeng, C. Zhang, and Z. Liang, "Palantír: Optimizing attack provenance with hardware-enhanced system observability," in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, H. Yin, A. Stavrou, C. Cremers, and E. Shi, Eds. ACM, 2022, pp. 3135–3149.

[46] Y. Ji, S. Lee, M. Fazzini, J. Allen, E. Downing, T. Kim, A. Orso, and W. Lee, "Enabling refinable cross-host attack investigation with efficient data flow tagging and tracking," in *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, W. Enck and A. P. Felt, Eds. USENIX Association, 2018, pp. 1705–1722.

[47] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Proceedings of the 31st Annual Computer Security Applications Conference*, 2015, pp. 401–410.

[48] "Chrome extension: Openapi spec generator." [Online]. Available: https://docs.akita.software/docs/google-chrome-extension

[49] "Exploring apis with zap." [Online]. Available: https://www.zaproxy.org/blog/2017-04-03-exploring-apis-with-zap/#spidering

[50] "Kiterunner: Contextual content discovery tool." [Online]. Available: https://github.com/assetnote/kiterunner

[51] "Automatically discovering api endpoints and generating schemas using machine learning." [Online]. Available: https://blog.cloudflare.com/ml-api-discovery-and-schema-learning/

[52] Joshua Tauberer, "email-validator: A robust email address syntax and deliverability validation library." 2024. [Online]. Available: https://github.com/JoshData/python-email-validator

Appendix A.
API Parameter Types and Validation Rules

Table 11 listed parameter types used by API Catalog, and their validation rules used in template generalization and matching in the trie.

TABLE 11. PARAMETER TYPES AND VALIDATION RULES IN API TEMPLATES

Type	Validation Rule
Integer	Value is a digit string
Float	Value is not a digit string and convertible to floating number
Null	Value is "null" or "none", ignoring cases
Boolean	Value is "true" or "false", ignoring cases
Hex String	Value is a hexadecimal string whose length more than 7
Email	Validated by `email-validator` [52]
Url	Validated by python's `urllib` standard library
UUID	Validated by python's `uuid` standard library
IP	Validated by python's `ipaddress` standard library
Static File	Value can be splitted as 2 parts by ".", and the second part is in a extension names list

Appendix B.
Description of Unit Features

Table 12 gives a more detailed description of the features extracted from request units for anomaly detection.

TABLE 12. FEATURES EXTRACTED FROM UNITS

Category	Name	Description
Syscall	syscalls	Set of used syscalls
	syscall_categories	Categories of used syscalls: Net, File, Memory
	syscall_sequenced	Sequence of used syscalls
File Descriptors	filenames	Names of accessed files
	directories	Parent directories of accessed files
	fd_types	Types of used file descriptors: File, IPv4, IPv6
	n_ips	Number of distinct accessed IPs
	n_ports	Number of distinct used ports
Process	proc_paths	Program paths executed by application

Appendix C.
Meta-Review

The following meta-review was prepared by the program committee for the 2026 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

C.1. Summary

This paper presents APIEcho, a novel intrusion detection system tailored for monolithic web applications that overcomes the limitations of rule-based systems and learning-based models. APIEcho performs anomaly detection on individual API endpoints based on the insight that legitimate requests exhibit highly similar behavior patterns.

C.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field
- Establishes a New Research Direction

C.3. Reasons for Acceptance

1) The paper establishes a new direction for web application intrusion detection by building profiles for individual endpoints, overcoming the limitations of prior approaches.
2) The evaluation demonstrates that the approach works well in real-world scenarios.

C.4. Noteworthy Concerns

1) The paper could have done more to highlight the limitations of the approach when handling complex APIs.
2) The paper should clarify how the comparative evaluation setup favors ReplicaWatcher.
3) The paper should provide more details on how logs are collected for analysis.
4) The evaluation methodology of comparing APIEcho to prior work that could in principle have been applied at an API endpoint level might underreport the efficacy of baseline techniques.
5) The technique is not truly training-less as claimed since concept drift must be manually handled by a human operator, leading to a retraining phase.

Appendix D.
Response to the Meta-Review

We thank the shepherd and reviewers for their constructive feedback. We provide additional context for the noteworthy concerns.

1) **On handling complex APIs:** Our behavioral similarity assumption is most applicable to single-responsibility APIs. For complex, polymorphic endpoints (e.g., where a parameter like "?action=..." dictates behavior), our design allows operators to define such parameters as part of the static API key. This partitions a single endpoint into distinct logical sub-APIs for independent analysis. While fully automatic discovery of such parameters is an open problem, this provides a robust solution for real-world deployments.
2) **On the ReplicaWatcher evaluation setup:** The comparison was designed to highlight the challenges of inter-replica comparison in monolithic settings. To create a strong baseline, the experiment was designed in ReplicaWatcher's favor: attack traffic was confined to a single replica per window, ensuring a clear majority of benign replicas for comparison. The performance gap, even under these ideal conditions, underscores the necessity of our intra-API approach.
3) **On log collection details:** Our methodology builds upon the reproducible framework from AutoLabel [28]. The lightweight instrumentation (e.g., middleware for FastAPI) inserts delimiters by writing to "/dev/null", a standard practice to capture contextual IDs in audit logs without altering application I/O.
4) **On the efficacy of baseline techniques:** A core contribution is our shift in granularity to per-request, intra-API analysis. Prior methods were not designed for this granularity. Applying them at this level would require significant re-engineering (e.g., new feature extraction), thus constituting a new research direction rather than a direct comparison of existing systems.
5) **On the "training-less" claim:** Our use of "training-less" signifies the absence of large-scale, offline pre-training, the primary burden of traditional ML systems. Adapting to concept drift is a lightweight, online baseline reset, not a "retraining phase." This process can be fully automated in CI/CD pipelines, where a new deployment triggers a reset and "warm-up" using test or canary traffic, which is fundamentally different from conventional model retraining.