

MINES: Explainable Anomaly Detection through Web API Invariant Inference

Wenjie Zhang*
wjzhang@nus.edu.sg
National University of Singapore
Singapore

Xiwen Teoh
xiwen.teoh@u.nus.edu
National University of Singapore
Singapore

Hongyu Zhang
hongyujohn@gmail.com
Chongqing University
Chongqing, China

Yun Lin†
lin_yun@sjtu.edu.cn
Shanghai Jiao Tong University
Shanghai, China

Xiaofei Xie
xfxie@smu.edu.sg
Singapore Management University
Singapore

Jin Song Dong
dcsdjs@nus.edu.sg
National University of Singapore
Singapore

Kwok Chun Fung Amos
e1373883@u.nus.edu
National University of Singapore
Singapore

Frank Liauw
Frank_LIAUW@tech.gov.sg
GovTech
Singapore

Abstract

Detecting the anomalies of web applications, important infrastructures for running modern companies and governments, is crucial for providing reliable web services. Many modern web applications operate on web APIs (e.g., RESTful, SOAP, and WebSockets), their exposure invites intended attacks or unintended illegal visits, causing abnormal system behaviors. However, such anomalies can share very similar logs (sometimes even identical logs) with normal logs, missing crucial information (which could be in database) for log discrimination. Further, log instances can be also noisy, which can further mislead the state-of-the-art log learning solutions to learn spurious correlation, resulting superficial models and rules for anomaly detection.

In this work, we propose MINES which infers explainable API invariants for anomaly detection from the *schema level* instead of detailed raw log instances, which can (1) significantly discriminate noise in logs to identify precise normalities and (2) detect abnormal behaviors beyond the instrumented logs (e.g., regarding the database state or session state). Our learned invariants can capture API preconditions such as (1) *what is the legitimate database state to initiate the call events?* and (2) *what are the constraints to satisfy between different API calls?*. Then we translate the invariants into executable Python code to verify its consistency with the runtime logs. Technically, MINES (1) converts API signatures into table schema to enhance the original database schema; and (2) infers the

potential database constraints (such as reference constraint and check constraints) on the enhanced database schema to capture the potential relationships between APIs and database tables. MINES uses LLM for extracting potential relationship based on two given table structures; and use normal log instances to reject and accept LLM-generated invariants. Finally, MINES translates the inferred constraints into invariants to generate Python code for verifying the runtime logs. We extensively evaluate MINES on web-tamper attacks on the benchmarks of *Train-Ticket*, *NiceFish*, *Gitea*, *Mastodon*, and *NextCloud* against baselines such as LogRobust, LogFormer, and WebNorm. The results show that MINES achieves high recall (more than 14% over LogRobust, LogFormer, and WebNorm) for the anomalies while introducing almost zero false positives, indicating a new state-of-the-art.

CCS Concepts

• **Security and privacy** → **Web application security**; • **Computing methodologies** → **Anomaly detection**; Natural language processing; • **Software and its engineering** → *Software testing and debugging*; • **Information systems** → *Web services*.

Keywords

Log-based anomaly detection, Specification mining, API analysis, Web application, Database

ACM Reference Format:

Wenjie Zhang, Yun Lin, Kwok Chun Fung Amos, Xiwen Teoh, Xiaofei Xie, Frank Liauw, Hongyu Zhang, and Jin Song Dong. 2026. MINES: Explainable Anomaly Detection through Web API Invariant Inference. In *Proceedings of Proceedings of the 48th International Conference on Software Engineering (ICSE '26)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/XXXXXXX>. XXXXXXXX

1 Introduction

Web applications are crucial infrastructures for running companies and governments in modern society [25, 44, 51]. Many applications operate through web APIs (e.g., RESTful and WebSockets) exposed to the public, which can attract intentional attacks or

*This work was partially conducted when Wenjie Zhang was visiting Shanghai Jiao Tong University

†Yun Lin is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICSE '26, Rio de Janeiro, Brazil

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2018/06
<https://doi.org/XXXXXXX.XXXXXXX>

unintended illegal access, resulting in approximately 35% of abnormal system behaviors, according to a Salt Security report [56]. Therefore, detecting such anomalies is important for operating and maintaining reliable web services [24]. To this end, researchers and industry practitioners have developed a variety of automated log-based anomaly detectors [67]. Traditional log anomaly detection approaches rely on predefined rules [23, 47, 49, 52, 53, 65, 68], which are limited to specific application scenarios and require domain expertise [14]. In recent years, researchers have proposed learning-based approaches to automatically learn normal behaviors from logs, which can be categorized into two types:

- **Model-learning Based Approaches.** Researchers have proposed detecting abnormal logs by training deep learning models in a supervised or unsupervised manner [14, 26, 30, 42, 66, 70]. In supervised learning solutions, anomaly detection is reduced to a binary classification problem [14, 26, 30, 70]. In contrast, unsupervised learning solutions [42, 66] learn normalities from collected normal logs, reporting logs as anomalies if they deviate from these normalities based on predefined metrics. However, both solutions can suffer from limited explainability [21] and the distribution shift problem [34].
- **Rule-learning Based Approaches.** Facilitated by advances in LLMs, recent works [32] learn project-specific rules in the form of first-order logic from collected logs. The learned rules are used to check against runtime logs. While such a solution can improve both the explainability of anomalies and detection accuracy for web-tampering attacks [32], it still suffers from generating false positive rules due to abundant log noise and producing false negatives when crucial information is unavailable in the logs (see Section 2).

State-of-the-art approaches have fundamental limitations in learning normalities solely from raw log instances, which presents challenges in log observability and log noise discrimination. Specifically, the more comprehensive logs we collect, the more informative we can potentially learn a discriminative model. However, the logs can also be more noisy to mislead the model to capture *spurious correlation* rather than causality.

For some APIs, comprehensive logging is necessary to learn true normalities. For example, a log event indicating the *successful* cancellation of a ticket order is considered normal only when (1) this ticket order exists in the database, (2) is in a valid state, and (3) is associated with the logged-in user in session information. Without comprehensive logging and querying of relevant information in the database and API execution context (session environmental information), it is not informative to discriminate such a log event. This brings the challenge of **C1: Observability Beyond Logs**, which means that the true normalities can be defined beyond the instrumented logs (e.g., database states and session environmental information).

However, comprehensive logging can introduce over detailed (or noisy) logs, which contributes to learning false positive rules (see a real-world example in Section 2), and it also incurs significant runtime overhead. This brings the challenge of **C2: True Invariants from Noisy Log Instances**, which means that learning models or rules from noisy and lengthy log instances is challenging.

In this work, we propose MINES (**M**ining **I**nvariants for a**N**omaly **d**etection via **S**chema), which infers explainable API invariants for anomaly detection from the *schema level* instead of from detailed raw log instances. MINES can (1) infer more causal normalities and (2) capture crucial information beyond the log instrumentation. Our rationale is that API specifications partially define the application’s normalities. Thus, specification-derived rules can deductively report anomalies in an explainable manner. Compared to inferring specifications from noisy log instances, we use abstract and precise meta-information in the application, such as API signatures and database schema, to derive the implicit specification, allowing us to

- (1) explore additional information (e.g., database state and session environmental information) and combining them with logs (to address **C1**) and,
- (2) exploit the most crucial information from abstract log schemas rather than concrete raw logs (to address **C2**).

To this end, MINES reduces the anomaly detection problem to an API specification inference problem. Based on the inferred invariants, MINES can detect anomalies by checking the consistency between the logs and the inferred invariants.

Specifically, MINES infers implicit log invariants among APIs, database, and environmental information, e.g., (1) *what is the legitimate database state or environmental information required to call an API?* or (2) *what are the constraints that must be satisfied between different API calls?* To achieve this, we build an augmented ER (entity-relationship) diagram to capture the API-DB constraints, API-API constraints, and API-Env constraints in the subject web application. By normalizing API signatures into entities, we can combine all the new and original entities and use LLMs to infer their constraints, such as reference constraints (e.g., foreign keys), not-null constraints, equality constraints, etc. Specifically, we convert each API (e.g., `cancelOrder(orderId, loginId)`) into a entity type (`cancelOrder`) with a set of attributes (`orderId` and `loginId`). Then, we augment the original ER diagram in the database to an extended ER diagram containing those API entity types, allowing MINES to infer their implicit constraints as invariants using LLMs. As a result, we can infer a specification such as

For each entity `cancelOrder` with an attribute `orderId` (derived from the API, referred to as `cancelOrder.orderId`), there *must* exist an entity `order`, such that `cancelOrder.orderId = order.id` and `order.status = "paid"`.

This inferred specification indicates that an API call of `cancelOrder` in the log (1) must have a corresponding record in the database table `order` with the corresponding order id and (2) the corresponding order record must be in *“paid”* status. By inferring the invariants at the schema level, we can (1) avoid inducing false positive invariants caused by accidental noisy correlations between two instances and (2) build invariants connecting logged APIs with the internal database state. In addition, we refine and filter the invariants by testing them against a collection of known normal logs and retaining those that do not generate false alarms. Finally, each invariant is translated into executable Python code for runtime log verification.

We extensively evaluate MINES on web-tamper attacks on the benchmarks of *Train-Ticket*, *NiceFish*, *Gitea*, *Mastodon*, and *NextCloud* against baselines such as LogRobust [72], LogFormer [19], and

WebNorm [32]. The results show that MINES achieves high recall (more than 14% over LogRobust, LogFormer, and WebNorm) with almost zero false positive in detecting anomalies. Also, evaluation on three extra popular web applications, *Gitea* [11], *Mastodon* [41], and *NextCloud* [45], shows that MINES can generalize well to popular industrial web applications. Additionally, the performance of generating invariants is stable across a variety of LLMs, indicating a new state-of-the-art anomaly detector for operating web applications.

In summary, the contributions of this paper are as follows:

- **Methodology.** We propose MINES, a schema-based specification-mining technique that unifies API signatures, database schema, and session environmental information, allowing us to exploit the most crucial features in abundant log structures and explore database states not instrumented in logs.
- **Tool.** We implement MINES as a framework which can be applied to any Java-based web application with available database schemas, facilitating real-world deployment.
- **Benchmark.** We build a web-tamper attack dataset, consisting of 31 types of attacks, that can successfully compromise known open-source web applications such as *Train-Ticket* and *NiceFish*. Thus, attacks are dynamic and replicable, allowing users to reproduce them with regenerated abnormal logs.
- **Evaluation.** We extensively evaluate MINES on the benchmark against state-of-the-art anomaly detectors such as LogRobust, LogFormer, and WebNorm, demonstrating its effectiveness in detecting web attacks and establishing a new state-of-the-art.

Given the space limit, more demos, source code, and experimental results are available at [5].

2 Motivating Example

Figure 1 shows abnormal logs caused by a web attack on the *Train-Ticket* system [43], which allows an attacker to successfully refund an order twice. In this real-world example, the normal logs for refunding an order look very similar to the attack-incurred abnormal logs, which causes state-of-the-art machine-learning based solutions such as LogFormer [19] and LogRobust [72] to fail to report the alarm. In addition, the rule-learning based solution [32] is misled to learn superficial rules (i.e., by capturing incorrect factors) from the normal logs, leading to false negatives.

Normal Logs and Their Semantics. The blue dashed rectangle in Figure 1 shows normal logs of refunding a ticket in the *Train-Ticket* system. For clarity, we simplify the logs containing two API calls:

- **API of queryOrder:** The log of this API indicates a query for existing orders. Users usually call this API so that they can choose a specific order to refund in the frontend.
- **API of refundOrder:** This log indicates canceling an order and refunding the money back to a user in the backend.

In normal scenarios, a user must query the orders before selecting one to refund. As a result, the logs of these two APIs can repetitively occur and form a pattern due to the design. However, such a normality of correlation does not indicate causality (i.e., the true normalities of calling **refundOrder**), which misleads existing solutions to learn false predictions or summarize incorrect rules.

Abnormal Logs Caused by Attacks. Unfortunately, based on vulnerabilities in the exposed APIs in *Train-Ticket*, a abnormal user can receive a refund multiple times. Such abnormal logs from the

attack are shown in the red dashed rectangle in Figure 1. Specifically, the attacker can call the exposed **refundOrder** an additional time to receive the extra refund. We tried model-learning based solutions such as LogFormer [19] and LogRobust [72] and rule-learning based solutions such as WebNorm [32] to detect the anomaly, finding their false negatives in practice as follows:

- **Model-learning Based Solutions.** Solutions such as LogRobust [72] and LogFormer [19] suffer from the *unexpected* subtle differences between normal logs (in the blue dashed box) and abnormal logs (in the red dashed box). Generally, these approaches project logs into an embedding space for their predictions [14]. Due to the textual similarity between normal and abnormal logs, the models have high confidence in reporting abnormal logs as normal.
- **Rule-learning Based Solutions.** In contrast, WebNorm summarizes a superficial invariant as first-order logic from the log instances, indicating that (1) there must be a log of **queryOrder** occurring before a log of **refundOrder** and (2) the value of `orderId` in the log of **refundOrder** must be equal to the value of `id` in the log of **queryOrder**. This invariant (a.k.a., detection rule) is superficial because the normality (or specification) of the log of **refundOrder** actually depends on some database states instead of the appearance of the log of **queryOrder**. Even worse, the attack-incurred abnormal logs well match the false invariant, causing a false negative.

As shown in the green rectangle in Figure 1, the true normality depends on the database state of the **order** table. Specifically, given an order to refund, its normality depends on whether there is a corresponding order record in the **order** table with its status as “paid” (see two tables under “*Information beyond Logs*” in Figure 1).

Technical Challenges. To detect the aforementioned log anomalies, we need to address the following technical challenges:

- **C1: Observability Beyond Logs.** The true normalities (or invariants) can be defined beyond the instrumented logs. In this example, we need to connect the logs, database states, and even session information to define a true invariant. However, exhaustively instrumenting database states into logs can incur large overhead. Therefore, we need to address “*how do we achieve the required operational observability beyond log instrumentation?*”. Moreover, the database states can be volatile and dynamic, so we also need to address how to synchronize its frequent changes with the ever-growing logs.
- **C2: True Invariants from Noisy Log Instances.** Learning models or rules from noisy and lengthy log instances is challenging. Inductive deep learning solutions can sometimes capture correlation instead of causality, leading to the well-known spurious correlation problem [67]. We need to address “*how to capture the most crucial facts and features to define the log normalities?*”.

In this work, we propose MINES, which detects runtime anomalies by inferring explainable invariants, as shown in the green box in Figure 1, from a schema level. In contrast to all the state-of-the-art methods that learn from log instances, we learn normalities and invariants from API signatures and database schema in the subject web application. This meta-information is more abstract and precise, allowing us to (1) explore additional information (e.g., database state and session environmental information) even if it is not in the

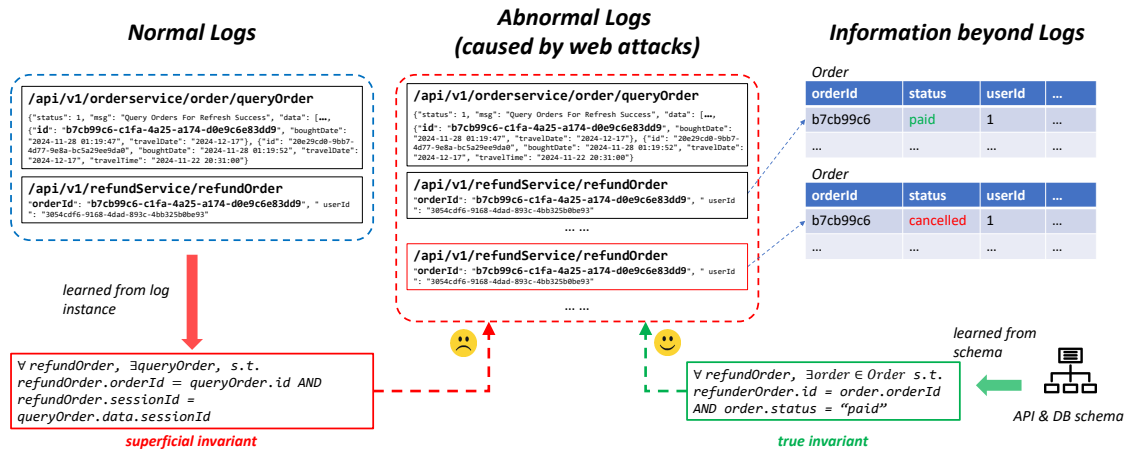


Figure 1: A log anomaly example caused by a real web attack on *Train-Ticket* system [43], which can successfully refund an order twice. The attack logs are similar to the normal logs, which makes log classification/regression models such as LogRobust [72] and LogFormer [19] ineffective. In addition, rule-learning based solution such as WebNorm [32] summarize a superficial rule from normal logs, which fails to detect such anomaly.

logs (to address C1). (2) exploit the most crucial information from the log structures (to address C2).

3 Approach

Figure 2 shows an overview of MINES, which parses the API signatures and database schema from the web application and converts them into an augmented entity-relationship diagram (e.g., Figure 6). In such a diagram, entity types consist of not only database tables but also API signatures and environmental information (e.g., sessions). Specifically, for each API signature, we convert it into an entity type, where the name is the API name and the attributes are the API parameters and return values (e.g., Figure 3). For environmental information, we place it into a separate entity type. Then, we reduce the anomaly detection problem into a constraint-inference problem for the ER diagram. To this end, we apply a two-stage inference in our approach:

- **Stage One: Entity-Relationship Inference.** At this stage, we infer relationships among entity types. The inferred relationships allow us to join two entity types (i.e., tables). For example, this allows us to join the `orderId` field in the `API:refundOrder` entity type to the `orderId` field in the `orders` entity type (Figure 6).
- **Stage Two: Invariant Inference.** At this stage, we join the tables by the inferred foreign keys from the first stage. For each joined table (e.g., between `API:refundOrder` entity type and `orders` entity type), we further infer not-null constraints, equality constraints, check constraints, etc., on their attributes. These constraints consist of both intra-entity constraints (e.g., price should be a positive number) and inter-entity constraints (e.g., `orderId` in `API:refundOrder` should match with `orderId` in `orders`). For example, this allows us to infer that the `status` attribute needs to be “paid” (Figure 7). Finally, the inferred constraints are translated into Python code as executable invariants.

Both stages require state-of-the-art LLMs (e.g., ChatGPT, Claude, and DeepSeek) to infer the constraints regarding the semantics.

To mitigate their potential hallucinations, we run the subject web application in a secure environment to obtain normal logs. Any generated invariants that raise false alarms on the normal logs will be removed to reduce false rules.

3.1 Information Requirements

Web applications differ widely in their architectures and implementations. To ensure broad applicability, MINES requires only a minimal and commonly available set of information from the target web application:

- **API Signatures:** Definitions of API endpoints, parameter types, and return types.
- **Database Schema:** Table structures and attribute types.
- **Contextual Information:** API invocation context, including session structures and access methods.
- **API Logs:** Collected records of API requests.
- **Database Binary Logs:** Historical records of database changes.

3.2 Schema Parsing

We parse the subject web application into three types of meta-information for logs, i.e., API signatures, database schema, and environmental information (e.g., sessions). All are processed into the database schema for further analysis.

Converting API Signatures to Schema. We convert each API signature into an *action* entity type, where the API name serves as the table name and the API parameters serve as the table attributes. Intuitively, each API signature is mapped into an entity type, while each API log is mapped into an entity instance. However, unlike traditional relational database table schemas where each attribute is a primitive type (e.g., string, int, or blob), an API can take complex objects (e.g., `OrderInfo` object) as input. Therefore, we parse the object tree structure into a flattened parameter list as shown in Figure 3. Specifically, given a complex input object as tree τ and a threshold th , we expand τ into the list according to the depth of th .

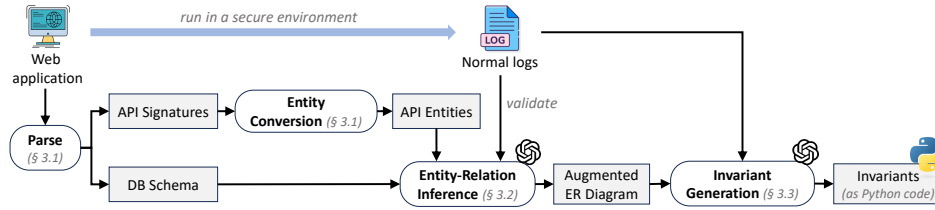


Figure 2: Approach Overview: Given a web application, MINES parses API signatures and database schema into an augmented ER (entity-relation) diagram. By inferring the reference constraints and customized constraints over the generated diagram by LLM, we infer the invariants as Python code for runtime verification. In addition, to avoid hallucination of LLM, we use normal logs to refine the generated constraints.

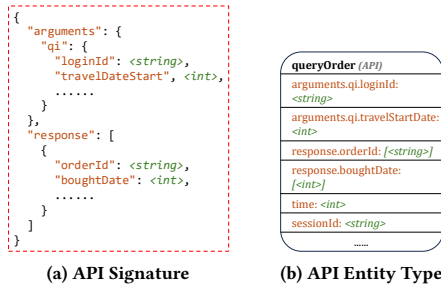


Figure 3: Example of Converting an API Signature to an API Entity Type.

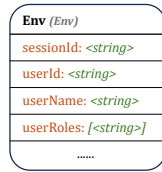


Figure 4: Example of Environmental Information.

Converting Environmental Information to Schema. Web applications utilize sessions for user authentication and authorization. There are many different implementations of session management, such as authentication tokens, cookies, and session IDs. For example, Figure 4 shows an example of environmental information in the *Train-Ticket* system. The environmental information stores the current user ID number, current user role, and current user name. By collecting and analyzing this information, we can ensure better security and a more personalized user experience. To capture structured logs and extract environmental information, we implemented a new instrumentation framework in Java, which is designed to capture environmental information from API handling methods.

3.3 Relationship Inference

This step aims to build the relationships from API entity types to other entity types. Specifically, MINES infers three types of relationships: (1) API-DB Relationships, i.e., foreign keys from API entity types to database tables, (2) API-API Relationships, i.e., temporal

```

# Identity
You are a software engineer who is extremely good at understanding business logic and user requirements for web applications. Given two entities, your task is to find out if there are any relationships between them.
[relationship definition] [input/output format] [in-context learning example]
# Input
- Focal Entity Type: cancel.service.CancelServiceImpl.cancelOrder
{ "userId": <str value>, "orderId": <str value> }
- Target Entity Type: users { "id": <str value>, "name": <str value> }
## Output:
<thought> [chain of thought] </thought>
... json
{
  "relationships": [
    "from_column": "userId",
    "to_column": "id",
  ]
}
...
    
```

Figure 5: Prompt used for relationship inference. The green boxes represent the input information, and the pink box represents the output of the LLM.

dependencies between different API entity types, and (3) API-Env Relationships, i.e., relevance of environmental information for APIs. Formally, we define the three relationships as follows:

- **API-DB Relationships.** Given an API E_{API} with an attribute a_{API} and a database table E_{DB} with an attribute a_{DB} , for every API log instance $e_{API} \in E_{API}$, there exists a database row $e_{DB} \in E_{DB}$ such that $e_{API}.a_{API} = e_{DB}.a_{DB}$.
- **API-API Relationships.** Given a source API E_{API1} and a target API E_{API2} , for every API log instance $e_{API1} \in E_{API1}$, there exists an API instance $e_{API2} \in E_{API2}$ such that $0 < e_{API1}.time - e_{API2}.time < \delta$ and $e_{API1}.session = e_{API2}.session$.
- **API-Env Relationships.** Given an API E_{API} and environmental information E_{Env} , for every API log instance $e_{API} \in E_{API}$, there exists an environmental entity instance $e_{Env} \in E_{Env}$ such that $e_{API}.session = e_{Env}.session$.

MINES infers the relationships (1) using a LLM, and then (2) refines the relationships using heuristic rules. For each pair of entity types, MINES first asks the LLM to find all potential relationships. Figure 5 shows the prompt used for relationship inference. The prompt includes a brief introduction to the definition of the task, the input/output format, and an in-context learning example. Due

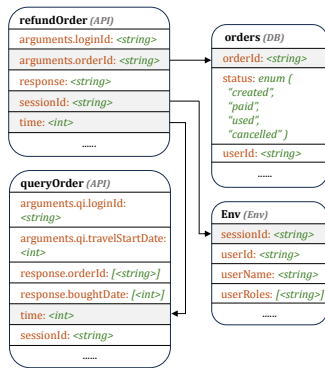


Figure 6: Example of an augmented entity-relationship diagram and inferred relationships. The API refundOrder has three relationships with the database table orders, another API queryOrder, and environmental information Env.

to space limitations, the detailed instructions are available in the anonymous code repository [5].

MINES filters out false relationships using the following heuristic rules:

- **API-DB Relationships:** Discarded if the attribute value overlap between columns is below a threshold.
- **API-API Relationships:** Discarded if their sequence probability, computed by a hidden Markov model (HMM) trained on API invocation logs, is below a threshold.
- **API-Env Relationships:** Discarded if the relevant environmental information is not present in the session logs.

The thresholds used in these heuristics are not specific to any application and can be easily adjusted for new projects with minimal effort, typically requiring only a small validation set or basic log statistics. Also, these heuristic rules serve only as a coarse filtering step; they do not play a central role in the overall analysis, but rather help to efficiently eliminate obvious false relationships before further processing.

Figure 6 shows an example of inferred relationships. The focal API entity type **refundOrder** has three relationships with other entity types, i.e., (1) a foreign key relationship with the **orders** table on the **orderId** attribute, (2) a temporal dependency with the **getOrder** API, and (3) relevancy to the environmental information.

3.4 Invariant Generation

To generate invariants, MINES first joins the entity types based on the inferred relationships. Then, MINES generates candidate invariants on the joined tables. After that, MINES refines the invariants using training logs. Figure 7 shows the process of invariant generation.

Joining Entities. Given any two entity types E_1 and E_2 with a relationship (API-DB, API-API, or API-Env), we perform a left outer join on their corresponding tables to preserve all API log instances. The join strategy follows the relationship type:

- **API-DB Relationships:** Correspond to foreign key constraints. For E_{API} and E_{DB} with relationship on a_{API} and a_{DB} , the join is $E_{API} \times E_{DB}$, filtered by $a_{API} = a_{DB}$.

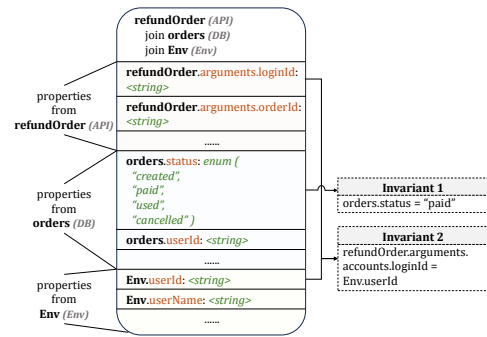


Figure 7: Candidate invariant generation over the joined table. The whole table consists of attributes from three tables, including API refundOrder, database table orders, and environmental information Env. Invariants are inferred from the joined table.

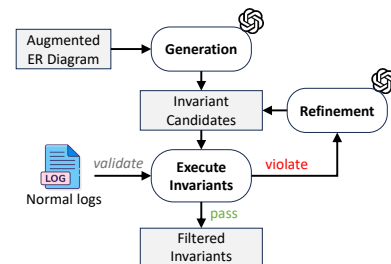


Figure 8: Invariant generation process. MINES first generates invariant candidates from the augmented ER diagram. Then it iteratively refines the candidates by executing the invariants against the training normal logs. If any violations occur, the invariants are refined by feeding back to the LLM again with the error message.

- **API-API Relationships:** Represent temporal dependencies. The join is $E_{API1} \times E_{API2}$, filtered by the temporal condition $0 < e_{API1}.time - e_{API2}.time < \delta$.
- **API-Env Relationships:** Based on environmental context, typically joined via session ID, analogous to a foreign key.

The resulting joined table (e.g., **API:refundOrder-orders-Env** in Figure 7) contains extra information beyond original API logs.

Candidate Invariant Generation. In this step, given the joined table structure, MINES asks LLMs to generate candidate invariants as Python code that checks the constraints on the joined table. Figure 7 shows an example of candidate invariant generation. The input is the joined table structure, which include table names, attributes, and their types. The output is pieces of Python code that checks the constraints on the joined table.

In the prompt, we provide a brief introduction to the task, the input/output format, and an in-context learning example. Due to space limitations, the detailed instructions are available in the anonymous code repository [5].

To help LLMs to generate invariants we want, the prompt guide the LLMs to generate invariants in five categories:

- (1) **Common-sense constraints.** Constraints that are generally applicable to the request (e.g., price should be a positive number).
- (2) **Format constraints.** Constraints on the format or type of the field (e.g., valid email format).
- (3) **Database constraints.** Constraints between the API arguments and the database entity (e.g., orderId should match the order entity).
- (4) **Environment constraints.** Constraints between the API arguments and the environmental information (e.g., userId should match the user ID in the session).
- (5) **Related API constraints.** Constraints between the request arguments and the responses of related APIs (e.g., data flow between different APIs).

Invariant Refinement. MINES iteratively refines the candidate invariants using training logs. Each candidate invariant is evaluated against the training logs. If any violations are detected, i.e., the invariant fails to hold on normal logs, these violations, along with relevant contextual information and error messages, are fed back to the LLM within the same conversation thread. The LLM is then prompted to revise or discard the problematic invariant. This creates a feedback loop in which the LLM iteratively refines its output based on concrete examples of failure. To prevent indefinite retries, this loop is bounded: if the LLM fails to produce a valid invariant within a fixed number of iterations, the system discards the candidate altogether by returning an empty result.

3.5 Runtime Verification

Offline Attack Detection. Based on the generated invariants, we run them against the logs and database records. Our method works in an offline manner, which means that the evaluation of the invariants will be carried out after a period of time following API execution. Offline attack detection minimizes online overhead.

In the offline attack detection setting, once the invariants are learned, the system operates only with two main components: a log collector and an offline checker.

- The **log collector** continuously collects logs during runtime as a producer.
- The **offline checker** executes the learned invariants (compiled as Python code) against the incoming logs to detect violations. Multiple checkers can be deployed in parallel to improve throughput and reduce latency.

Binary Log History Tracking. In the offline setting, we need to address the challenge of database dynamics, i.e., how do we track the historical database records even if the database state changes.

To this end, we replay the binary log, which records all changes to database records and is typically enabled by default in modern web applications, to restore the database state during invariant evaluation. By extracting and applying binary log records, we ensure consistent and accurate evaluation of invariants, mitigating issues from stateful data changes.

4 Benchmark Construction

Executing MINES requires not only API logs, but also database information, environmental information, and binary logs. However, existing datasets for web applications only contain logs and do

Table 1: Test & Attack Scripts Statistics (# denotes the number of)

	<i>Train-Ticket</i>	<i>NiceFish</i>
#Status	21	13
#Normal Operations	25	27
#Test Target APIs	48	26
#Attack Scripts	25	6
#Attack Target APIs	14	13

Table 2: Number of Logs and Attacks

	<i>Train-Ticket</i>		<i>NiceFish</i>	
	#Logs	#Attacks	#Logs	#Attacks
<i>ManualNorm</i>	168,799	0	40,654	0
<i>LLMNorm</i>	8,007	0	3,720	0
<i>ManualAbnormal</i>	6,054	125	4,102	120
<i>InjectAbnormal</i>	8,007	125	3,720	120

not provide necessary information for re-execution. Therefore, we constructed two new datasets for two web applications. These datasets provide dynamic and replicable scripts to generate both normal and attack logs, allowing us to reproduce logs or binary logs as needed.

To align with our baseline WebNorm [32], we collected our datasets from the same two web applications: *Train-Ticket* [43] and *NiceFish* [12], which are widely used in the web development community. Our new datasets contain 31 types of attacks and cover 27 APIs.

4.1 Dataset Construction

To ensure dataset diversity, we generate both normal and abnormal logs using manually written scripts and LLM-based methods. The logs are categorized into four types:

- **ManualNorm:** Manually scripted normal logs, representing typical user operations organized as finite state machines.
- **LLMNorm:** LLM-generated normal logs, using scripts from WebNorm [32] to predict actions from web application screenshots and task descriptions.
- **ManualAbnormal:** Manually scripted abnormal logs, targeting identified web abnormal endpoints by scripting abnormal scenarios for each application functionality.
- **InjectAbnormal:** Abnormal-injected logs, created by injecting abnormal fields into normal logs from *LLMNorm*, following the protocol in WebNorm [32].

Table 1 and Table 2 present the statistics of normal and abnormal scripts and tests for the *Train-Ticket* and *NiceFish* datasets. LLM-generated normal logs more closely reflect real user behavior, while manual scripts provide comprehensive coverage. For abnormal logs, both manual and injection-based methods are used to capture a wide range of attack scenarios. This ensures our dataset thoroughly encompasses all abnormal cases in WebNorm [32]. We constructed all identifiable normal and abnormal scenarios for both web applications. These scripts not only cover every normal and abnormal

scenario in WebNorm [32], but also increase diversity by targeting more APIs and abnormal cases. Detailed examples can be found at [5].

4.2 Division of Training and Evaluation Datasets

Given the four types of generated logs, we construct training and evaluation datasets separately for MINES and the model-based baselines.

Rule-learning approaches, such as WebNorm [32] and our proposed MINES, require only normal logs for training. For this purpose, we use logs generated by *ManualNorm*, which comprehensively cover a wide range of normal operations. For evaluation, we employ logs generated by *LLMNorm*, *ManualAbnormal*, and *InjectAbnormal* to assess generalization and robustness.

In contrast, model-based approaches such as LogRobust [72] and LogFormer [19] rely on both normal and abnormal logs for training. Therefore, in addition to *ManualNorm* logs, their training sets also include normal logs generated by *LLMNorm* and attack logs.

In summary, for training, we use *ManualNorm* for MINES and both *ManualNorm* and *LLMNorm* for model-based baselines. For evaluation, we use *LLMNorm*, *ManualAbnormal*, and *InjectAbnormal* for all approaches.

It is important to note that model-based baselines are trained with a larger volume of data than MINES. This design choice does not compromise fairness; instead, it favors the baselines by giving them access to more expressive training signals. Despite this advantage, MINES still achieves superior performance, demonstrating its effectiveness under a more constrained training setup.

5 Experiments

We evaluate our approach with the following research questions:

- **RQ1:** What is the overall effectiveness and efficiency of our approach compared with the baselines?
- **RQ2:** How do different components (*schema-based deduction*, *contextual relationships*, *binary log history tracking*) and the *invariant refinement* process contribute to the performance and robustness of our approach?
- **RQ3:** How robust is our approach when equipped with different language models?
- **RQ4:** How does the quality and consistency of system naming conventions affect the performance of our approach?
- **RQ5:** Can our approach generalize to popular real-world web applications?

5.1 Setup

Baselines. We compare MINES with two categories of baselines:

- **Model-learning based approaches:** We chose LogRobust [72] and LogFormer [19] as baselines because they are the latest model-learning based applications and perform best among similar models, as shown in [19, 32].
- **Rule-learning based approaches:** We compare MINES with WebNorm [32], which is the only existing interpretable normality learning method.

Table 3: Overall evaluation of MINES

Model	<i>Train-Ticket</i>		<i>NiceFish</i>	
	Precision	Recall	Precision	Recall
LogRobust [72]	0.120	0.650	0.207	0.540
LogFormer [19]	0.272	0.764	0.301	0.702
WebNorm [32]	1.000	0.704	1.000	0.750
MINES (Ours)	1.000	0.948	1.000	0.917

Table 4: Performance on Two types of Attacks

Attack Type	<i>Train-Ticket</i>	<i>NiceFish</i>
<i>ManualAbnormal</i>	0.896	0.834
<i>InjectAbnormal</i>	1.000	1.000
Overall	0.948	0.917

Benchmarks. We utilize the two benchmarks constructed in the previous section, *Train-Ticket* and *NiceFish*.

Metrics. We use precision and recall as metrics. For normal logs, we split logs into windows of 20 logs each. A window is marked *False Positive* (FP) if any attacks are detected; otherwise, it is *True Negative* (TN). For attack logs, detection of any attacks results in *True Positive* (TP) for all logs; otherwise, they are *False Negative* (FN). Precision and recall are calculated as: Precision = $\frac{TP}{TP+FP}$, and Recall = $\frac{TP}{TP+FN}$.

LLMs Used. We primarily use the GPT-4o model [27], specifically version gpt-4o-2024-08-06, for our evaluation. To assess MINES’s performance across different LLMs, we also employ GPT-4o-mini (gpt-4o-mini-2024-07-18), Claude 3.7 Sonnet [6] (claude-3-7-sonnet-20250219), and DeepSeek-V3 [33] (deepseek-v3-241226).

5.2 Overall Effectiveness and Efficiency (RQ1)

We comprehensively evaluate the effectiveness and efficiency of MINES against baselines on two datasets, as shown in Table 3. MINES consistently outperforms baselines on both *Train-Ticket* and *NiceFish*. Both MINES and WebNorm achieve 100% precision due to refined invariants, ensuring accuracy. In recall, MINES achieves 94.8% on *Train-Ticket* and 91.7% on *NiceFish*, exceeding baselines by over 15%, demonstrating superior attack detection¹.

The attack data includes *ManualAbnormal* and *InjectAbnormal*. We evaluated MINES on these attacks separately, as shown in Table 4. MINES achieves a recall of 1.0 on *InjectAbnormal*, due to the easier detection of simulation attacks, and nearly 0.9 on *ManualAbnormal*, demonstrating its effectiveness on real attacks.

We further measure the training and evaluation performance of MINES, as shown in Table 5. The training overhead and cost are acceptable, and MINES achieves high evaluation throughput, 4×10^5 logs per second on *Train-Ticket* and 2×10^5 logs per second on *NiceFish*, significantly faster than model-learning baselines.

¹The precision and recall values reported for WebNorm differ from those in the original WebNorm paper due to differences in experimental settings. Specifically, our evaluation covers a broader set of scenarios, which, combined with the invariant refinement process, leads to higher precision but lower recall compared to the original results.

Table 5: Performance evaluation of MINES

	<i>Train-Ticket</i>	<i>NiceFish</i>
Number of APIs	48	26
Training Time (s)	644	406
Training LLM Cost (USD)	20.6	13.7
Running Throughput (log/s)	4.0×10^5	2.4×10^5

Table 6: Ablation Study

Model	<i>Train-Ticket</i>	<i>NiceFish</i>
Original (MINES)	0.948	0.917
w/o deducing from schemas	0.896	0.750
w/o API-DB relationships	0.820	0.834
w/o API-API relationships	0.908	0.836
w/o API-Env relationships	0.780	0.750
w/o binlog history tracking	0.884	0.917
WebNorm	0.704	0.750

Table 7: Comparing Input Token Numbers between Raw Log Input and Schema Input

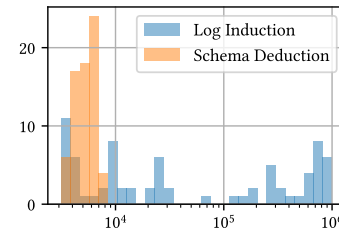
Dataset	Method	Mean	GeoMean	Median
<i>Train-Ticket</i>	Raw Log Input	2.40×10^5	4.34×10^4	2.36×10^4
	Schema Input (Ours)	5.01×10^3	4.86×10^3	5.01×10^3
<i>NiceFish</i>	Raw Log Input	1.50×10^4	6.86×10^3	4.45×10^3
	Schema Input (Ours)	3.76×10^3	4.68×10^3	3.54×10^3

RQ1: MINES achieves both high effectiveness and efficiency. It outperforms baselines in attack detection with 100% precision and over 15% higher recall, while maintaining acceptable training overhead and processing more than 2×10^5 logs per second during evaluation.

5.3 Component Contribution (RQ2)

To understand how different components contribute to MINES’s effectiveness and efficiency, we conducted a comprehensive ablation study and analysis on prompt construction, input representation, and invariant refinement.

Impact of Core Components. We first examine how individual components affect recall while maintaining 100% precision, as shown in Table 6. Removing key elements such as API-DB, API-API, or API-Env relationships causes the recall to drop from 0.948 to 0.820, 0.908, and 0.780, respectively. Among them, API-Env contributes the most, followed by API-DB. This demonstrates that contextual and structured information, linking API calls with their data and environment, is crucial for accurate invariant reasoning. Similarly, omitting binary log history tracking, which aligns each API invocation with its historical database state, significantly reduces recall, confirming the importance of temporal consistency. Notably, WebNorm can be viewed as a composite ablation combining several of these removals, explaining its lower recall in prior comparisons.

**Figure 9: Histogram of the number of tokens in the input logs****Table 8: False Negatives Before and After Refinement**

	<i>Train-Ticket</i>	<i>NiceFish</i>
Dataset Size	168,799	40,654
Number of False Positives (w/o Refinement)	33,317	36,278
False Positive Rate (w/o Refinement)	0.197	0.892
Number of False Negatives (w/ Refinement)	0	0
False Negative Rate (w/ Refinement)	0.000	0.000

Effect of Deducing from Schemas. We further compare *deducing from schemas* with *inducing from raw logs*. Replacing schema-based prompts with raw logs decreases recall from 0.948 to 0.896, showing that schemas provide cleaner and more abstract representations that generalize better than noisy instances. In addition, schema-based deduction dramatically reduces token counts in prompts, by up to two orders of magnitude across mean, geometric mean, and median values, as shown in Table 7 and illustrated in Figure 9. This compression leads to lower inference cost and faster evaluation without loss of accuracy.

Effect of Invariant Refinement. MINES employs a two-stage pipeline for invariant generation: (1) extracting candidate invariants from schemas and (2) refining them with training logs. The refinement step is crucial for filtering false positives, reducing their rate from several percent to zero (Table 8). Without refinement, the system would produce excessive false alarms, especially in large-scale deployments. Refinement also mitigates LLM hallucinations through iterative feedback, where invalid or overfitted invariants are revised or discarded based on validation logs. Only a moderate amount of normal log data is required, making this process lightweight yet essential for practical precision.

RQ2: Each component of MINES, including schema-based deduction, API-DB/API-Env/API-API relationships, binary log tracking, and invariant refinement, plays a vital role in enhancing both recall and efficiency. Schema-based inputs reduce token length by up to two orders of magnitude, while refinement eliminates false positives and ensures robust, deployable precision.

5.4 Comparing LLMs (RQ3)

We evaluated MINES using different LLMs, GPT-4o-mini, Claude 3.7, and DeepSeek-V3, on the *Train-Ticket* and *NiceFish* dataset to assess its robustness across models. As shown in Table 9, Claude 3.7 and

Table 9: Comparison of Different LLMs

	<i>Train-Ticket</i>		<i>NiceFish</i>	
	Precision	Recall	Precision	Recall
GPT-4o	1.000	0.948	1.000	0.833
GPT-4o-mini	1.000	0.800	1.000	0.750
Claude 3.7	1.000	0.888	1.000	0.833
DeepSeek-V3	1.000	0.936	1.000	0.833

Table 10: Impact of Naming Conventions on MINES

	Precision	Recall
Original Names (e.g., QueryByBatch)	1.00	0.94
Snake Case (e.g., query_by_batch)	1.00	0.94
Camel Case (e.g., QueryByBatch)	1.00	0.94
Concat All (e.g., querybybatch)	1.00	0.94
Partial Abbreviation (e.g., query)	1.00	0.93
Extreme Abbreviation (e.g., q)	1.00	0.84

DeepSeek-V3 achieve comparable results to GPT-4o, demonstrating that MINES maintains consistent performance across different architectures.

GPT-4o-mini, however, performs slightly worse. To better understand this gap, we conducted a manual analysis of 10 representative cases where GPT-4o succeeded but GPT-4o-mini failed. In 9 out of 10 cases, GPT-4o-mini correctly described the invariants in natural language but failed to produce valid executable code, mainly due to formatting issues or semantic inconsistencies. This suggests that the performance gap is mainly attributable to limitations in code generation.

RQ3: MINES achieves consistent performance across LLMs, though smaller models like GPT-4o-mini may struggle with code generation despite adequate reasoning.

5.5 Effect of Naming Conventions (RQ4)

Large language models play a central role in MINES, as they interpret entity and attribute names to induce semantic relationships. Consequently, MINES’s effectiveness inherently depends on the clarity and consistency of naming conventions in the target system. In our default setup, we assume that system identifiers (e.g., API names, table names, and field names) are reasonably descriptive, following common engineering practice.

To systematically assess the impact of naming quality, we conducted controlled experiments by modifying the original entity names using several strategies: (1) replacing words with partial or extreme abbreviations, (2) applying stylistic variations such as snake case and camel case, and (3) combining these transformations. As shown in Table 10, MINES remains robust under most conventional naming styles and moderate abbreviations. However, its performance declines significantly when names are heavily abbreviated (e.g., replaced by single letters or meaningless tokens). In such cases, the model mainly detects superficial format violations rather than deeper inter-attribute inconsistencies.

This result highlights that while MINES can tolerate moderate variation in naming, it fundamentally relies on semantically meaningful identifiers to establish relationships across APIs, database fields, and environmental contexts. Fortunately, such descriptive naming is widely adopted in real-world software systems, so the dependency is realistic and manageable in practice.

RQ4: MINES is generally robust to common naming styles and moderate abbreviations, but its performance degrades when semantic clarity is lost. This demonstrates that the approach relies on meaningful system naming to guide LLM reasoning.

5.6 Generalization to Popular Web Applications (RQ5)

To evaluate MINES’s generalization to popular web applications, we collected three GitHub repositories of web applications. We referred to Gitstar Ranking to select the most starred web applications. We filtered out applications that are not productive or learning projects. We got the first three applications: NextCloud [45], Gitea [11], and Mastodon [41]. The descriptions and statistics are shown in Table 11. Due to space limit, more ablation studies of these applications are available in our repository [5]. For each application, we wrote LLM scripts to generate normal logs. For abnormal logs, we manually injected attacks into normal logs. For each application, we wrote 5 scripts to generate abnormal logs and execute each of them several times to generate enough logs. Due to limitation of space, detailed attack scenarios are not included in this paper, but can be found at [5]. Table 12 show the evaluation results of MINES on these applications. MINES achieves also achieves 100% precision and high recall on all three applications, indicating that MINES generalizes well to popular web applications.

RQ5: MINES generalizes well to popular web applications, achieving high precision and recall on NextCloud, Gitea, and Mastodon.

5.7 Discussion

Despite the strong performance of MINES across benchmarks, several limitations remain.

Migration Complexity. Adapting MINES to new systems may require engineering effort, including handling diverse database backends, session management, and API endpoints. Our framework minimizes technology dependencies by relying only on common components such as APIs and database schemas.

Dependency on Naming Quality. The effectiveness of MINES depends on meaningful and consistent naming in APIs and schemas. While this assumption generally holds, systems with obfuscated or inconsistent naming remain challenging.

Optional but Beneficial Documentation. The current implementation does not utilize application-level documentation. Although not required, documentation such as ER diagrams or type annotations can significantly improve invariant inference. For example, on *Train-Ticket*, MINES failed to detect an attack where the API field `seatClass` was set to 5, while valid values were only 0 and 1. Adding a single line of documentation, “*valid values for seat class are 0 and 1*”, enabled MINES to synthesize the correct invariant and detect the attack. This suggests that even minimal documentation can

Table 11: Extra benchmarks to evaluate generalization of MINES

Project	Line of Code	Language	Web Framework	Database	GitHub Stars	Number of Microservices	Number of Database Tables
<i>Train-Ticket</i>	37.8k	Java	Spring	MySQL	804	41	34
<i>NiceFish</i>	4.7k	Java	Spring	MySQL	732	2	16
<i>Gitea</i>	330.7k	Go	Gin	MySQL	49.6k	1	114
<i>Mastodon</i>	109.8k	Ruby	Ruby on Rails	PostgreSQL	48.6k	1	98
<i>NextCloud</i>	426.0k	PHP	Vanilla PHP	MariaDB	30.2k	1	129

Table 12: Overall evaluation of MINES on extra benchmarks

Benchmark	Metric	LogRobust [72]	LogFormer [19]	WebNorm [32]	MINES (Ours)
Gitea	Precision	0.640	0.481	1.000	1.000
	Recall	0.970	0.474	0.176	0.956
	F1	0.771	0.477	0.299	0.977
Mastodon	Precision	0.233	0.454	1.000	1.000
	Recall	1.000	0.625	0.667	0.833
	F1	0.377	0.526	0.800	0.909
NextCloud	Precision	0.064	0.059	1.000	1.000
	Recall	1.000	0.300	0.750	0.906
	F1	0.120	0.099	0.857	0.950

enhance semantic precision and motivates documentation-aware extensions.

Requirement for Comprehensive Logs. Comprehensive logs are essential for capturing representative behaviors and reducing false positives. While typically available in testing or staging environments, performance may degrade in log-sparse scenarios.

6 Related Work

Log Anomaly Detection. Log anomaly detection can be traced back to execution trace analysis. Some works focus on analyzing the execution of specific APIs or methods to find anomalies [67]. Traditional log anomaly detection approaches rely on predefined rules [23, 47, 49, 52, 53, 65, 68], which are limited to specific application scenarios and require domain expertise [14].

In recent years, researchers have proposed learning-based approaches to automatically learn normal behaviors from logs [1, 2, 4, 8, 9, 28, 29, 35, 37, 46, 48, 54, 55, 57, 60, 62], which can be categorized into two types.

The first category utilizes neural networks to directly predict whether a log sequence is normal or anomalous [3, 10, 14, 16–20, 22, 26, 30, 31, 36, 42, 50, 58, 61, 66, 69, 72, 73]. These methods utilize different neural network architectures to enhance the performance of web anomaly detection, including RNNs [10, 14], CNNs [17, 36], Transformers [19, 26], GNNs [71], pretrained language models [20, 22], and instrumented large language models [50]. Some works also utilize unsupervised or semi-supervised learning to alleviate the need for labeled data [42, 66]. These methods can capture the temporal dependencies in log sequences and improve the performance of log anomaly detection. However, they often lack explainability in their detection results and may struggle to capture subtle changes in abnormal logs.

The second category focuses on learning explainable normalities to detect anomalies [32]. The only work in this category is WebNorm [32]. WebNorm detects web anomalies by learning normality first-order logic rules for web applications. This method offers better

explainability and can capture subtle but crucial changes. However, WebNorm only focuses on analyzing web logs and does not consider the relational integrity between web logs and the underlying database. Our proposed method, MINES, aims to address these limitations by inspecting the relational integrity between web logs and the database and generating normality rules based on the abstract schema and the ER diagram.

RESTful API Security. RESTful APIs employ a stateless architecture and standard HTTP methods, and are now widely used in web applications, making their security a major concern. Numerous studies have addressed RESTful API security, primarily through automated test case generation to detect vulnerabilities [7, 13, 15, 39, 40, 59, 63]. These approaches typically fuzz API sequences and insert attack or detection invocations to uncover issues, leveraging API specifications, data dependencies [40, 59], and neural network predictions [38, 64]. Some works further enhance testing via targeted fuzzing strategies [13, 15]. These methods are mainly designed to reveal vulnerabilities such as SQL injection and cross-site scripting.

In contrast, our work aims to generate normality rules for RESTful APIs, strengthening web application security by specifically detecting attacks that violate the intended normal behaviors of web applications.

7 Conclusion

In this paper, we propose MINES, a novel rule-learning based approach to enhance web application security. By leveraging *deducing from specifications* and utilizing information beyond log instrumentation, MINES is more effective in detecting anomalies in web applications. Experiments on two datasets demonstrate that MINES outperforms existing approaches.

Acknowledgments

This research is supported in part by the National Natural Science Foundation of China (62572300), the Minister of Education, Singapore (MOE-T2EP20124-0017, MOET32020-0004), the National Research Foundation, Singapore and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN), DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008-1B), and Cyber Security Agency of Singapore under its National Cybersecurity R&D Programme and CyberSG R&D Cyber Research Programme Office. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore as well as CyberSG R&D Programme Office, Singapore.

References

- [1] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC/FSE*. <https://doi.org/10.1145/1287624.1287630>
- [2] Md Rakibul Alam, Ilias Gerostathopoulos, Christian Prehofer, Alessandro Atanasi, and Tomas Bures. 2019. A framework for tunable anomaly detection. In *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 201–210.
- [3] Crispin Almodovar, Fariza Sabrina, Sarvnaz Karimi, and Salahuddin Azad. 2024. LogFiT: Log anomaly detection using fine-tuned language models. *IEEE Transactions on Network and Service Management* 21, 2 (2024), 1715–1723. <https://doi.org/10.1109/TNSM.2024.3358730>
- [4] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2018. Using finite-state models for log differencing. In *ESEC/FSE*. <https://doi.org/10.1145/3236024.3236069>
- [5] Anonymous Authors. 2025. *Details of MINES*. <https://sites.google.com/view/mines-anomaly-detection/home>
- [6] Anthropic PBC. 2024. *Claude 3.7 Sonnet and Claude Code*. <https://www.anthropic.com/news/claude-3-7-sonnet>
- [7] Vaggelis Atlidakis, Patrice Godefroid, and Marina Polishchuk. 2019. RESTler: Stateful rest api fuzzing. In *ICSE*. <https://doi.org/10.1109/ICSE.2019.00083>
- [8] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *ESEC/FSE*. <https://doi.org/10.1145/2025113.2025151>
- [9] Jakub Breier and Jana Branišová. 2015. Anomaly detection from log files using data mining techniques. In *Information Science and Applications*. 449–457. https://doi.org/10.1007/978-3-662-46578-3_53
- [10] Andy Brown, Aaron Tuor, Brian Hutchinson, and Nicole Nichols. 2018. Recurrent neural network attention mechanisms for interpretable system log anomaly detection. In *MLCS*. <https://doi.org/10.1145/3217871.3217872>
- [11] CommitGo. 2016. *Gitea*. <https://about.gitea.com/>
- [12] Da Mo Qiong Qiu. 2016. *NiceFish*. <https://github.com/mumu-osc/NiceFish>
- [13] Gelei Deng, Zhiyi Zhang, Yuekang Li, Yi Liu, Tianwei Zhang, Yang Liu, Guo Yu, and Dongjin Wang. 2023. NAUTILUS: Automated RESTful API Vulnerability Detection. In *USENIX Security*. <https://www.usenix.org/conference/usenixsecurity23/presentation/deng-gelei>
- [14] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. DeepLog: Anomaly detection and diagnosis from system logs through deep learning. In *CCS*. <https://doi.org/10.1145/3133956.3134015>
- [15] Wenlong Du, Jian Li, Yanhao Wang, Libo Chen, Ruijie Zhao, Junmin Zhu, Zhengguang Han, Yijun Wang, and Zhi Xue. 2024. Vulnerability-oriented testing for restful apis. In *USENIX Security*. <https://www.usenix.org/conference/usenixsecurity24/presentation/du>
- [16] Asbat El Khairi, Marco Caselli, Andreas Peter, and Andrea Continella. 2024. REPLICAWATCHER: Training-less Anomaly Detection in Containerized Microservices. In *NDSS*. <https://doi.org/10.14722/ndss.2024.24286>
- [17] Yuanyuan Fu, Kun Liang, and Jian Xu. 2023. Mlog: Mogrifier lstm-based log anomaly detection approach using semantic representation. *IEEE Transactions on Services Computing* 16, 5 (2023), 3537–3549. <https://doi.org/10.1109/TSC.2023.3289488>
- [18] Maayan Goldstein, Danny Raz, and Itai Segall. 2017. Experience report: Log-based behavioral differencing. In *ISSRE*. <https://doi.org/10.1109/ISSRE.2017.14>
- [19] Hongcheng Guo, Jian Yang, Jiaheng Liu, Jiaqi Bai, Boyang Wang, Zhoujun Li, Tieqiao Zheng, Bo Zhang, Junran Peng, and Qi Tian. 2024. Logformer: A pre-train and tuning pipeline for log anomaly detection. In *AAAI*. <https://doi.org/10.1609/aaai.v38i1.27764>
- [20] Haixuan Guo, Shuhan Yuan, and Xintao Wu. 2021. Logbert: Log anomaly detection via bert. In *IJCNN*. <https://doi.org/10.1109/IJCNN52387.2021.9534113>
- [21] Lynn Vonder Haar, Timothy Elvira, and Omar Ochoa. 2023. An analysis of explainability methods for convolutional neural networks. *Engineering Applications of Artificial Intelligence* 117 (2023), 105606. <https://doi.org/10.1016/j.engappai.2022.105606>
- [22] Xiao Han, Shuhan Yuan, and Mohamed Trabelsi. 2023. LogGPT: Log anomaly detection via GPT. In *BigData*. <https://doi.org/10.1109/BigData59044.2023.10386543>
- [23] Stephen E. Hansen and E. Todd Atkins. 1993. Automated System Monitoring and Notification With Swatch. In *LISA*. <https://dl.acm.org/doi/10.5555/1024753.1024780>
- [24] Shilin He, Pinjia He, Zhuangbin Chen, Tianyi Yang, Yuxin Su, and Michael R Lyu. 2021. A survey on automated log analysis for reliability engineering. *ACM computing surveys (CSUR)* 54, 6 (2021), 1–37. <https://doi.org/10.1145/3460345>
- [25] Andrew Hoffman. 2024. *Web application security*. " O'Reilly Media, Inc".
- [26] Shaohan Huang, Yi Liu, Carol Fung, Rong He, Yining Zhao, Hailong Yang, and Zhongzhi Luan. 2020. HitAnomaly: Hierarchical transformers for anomaly detection in system log. *IEEE transactions on network and service management* 17, 4 (2020), 2064–2076. <https://doi.org/10.1109/TNSM.2020.3034647>
- [27] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o system card. *arXiv* (2024). <https://arxiv.org/abs/2410.21276>
- [28] Qiao Kang, Ankit Agrawal, Alok Choudhary, Alex Sim, Kesheng Wu, Rajkumar Kettimuthu, Peter H Beckman, Zhengchun Liu, and Wei-keng Liao. 2019. Spatiotemporal real-time anomaly detection for supercomputing systems. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 4381–4389.
- [29] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic mining of specifications from invocation traces and method invariants. In *FSE*. <https://doi.org/10.1145/2635868.2635890>
- [30] Van-Hoang Le and Hongyu Zhang. 2021. Log-based anomaly detection without log parsing. In *ASE*. <https://doi.org/10.1109/ASE51524.2021.9678773>
- [31] Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. 2020. Swiss-Log: Robust and unified deep learning based log anomaly detection for diverse faults. In *ISSRE*. <https://doi.org/10.1109/ISSRE5003.2020.00018>
- [32] Yifan Liao, Ming Xu, Yun Lin, Xiwen Teoh, Xiaofei Xie, Ruitao Feng, Frank Liaw, Hongyu Zhang, and Jin Song Dong. 2024. Detecting and Explaining Anomalies Caused by Web Tamper Attacks via Building Consistency-based Normality. In *ASE*. <https://doi.org/10.1145/3691620.3695024>
- [33] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Cheng-gang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv* (2024). <https://arxiv.org/abs/2412.19437>
- [34] Jiashuo Liu, Zheyang Shen, Yue He, Xingxuan Zhang, Renzhe Xu, Han Yu, and Peng Cui. 2021. Towards out-of-distribution generalization: A survey. *arXiv preprint arXiv:2108.13624* (2021). <https://arxiv.org/abs/2108.13624>
- [35] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic generation of software behavioral models. In *ICSE*. <https://doi.org/10.1145/1368088.1368157>
- [36] Siyang Lu, Xiang Wei, Yandong Li, and Liqiang Wang. 2018. Detecting anomaly in big data system logs using convolutional neural network. In *DASC*. <https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00037>
- [37] Scott Lupton, Hironori Washizaki, Nobukazu Yoshioka, and Yoshiaki Fukazawa. 2021. Literature review on log anomaly detection approaches utilizing online parsing methodology. In *2021 28th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 559–563.
- [38] Chenyang Lyu, Jiacheng Xu, Shouling Ji, Xuhong Zhang, Qinying Wang, Binbin Zhao, Gaoning Pan, Wei Cao, Peng Chen, and Raheem Beyah. 2023. MINER: A Hybrid Data-Driven Approach for RESTAPI Fuzzing. In *USENIX Security*. <https://www.usenix.org/conference/usenixsecurity23/presentation/lyu>
- [39] Alberto Martin-Lopez. 2020. Automated analysis of inter-parameter dependencies in web APIs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 140–142.
- [40] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2020. RESTest: Black-box constraint-based testing of RESTful web APIs. In *Service-Oriented Computing: 18th International Conference, IC3SOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings* 18. Springer, 459–475. https://doi.org/10.1007/978-3-030-65310-1_33
- [41] Mastodon gMBH. 2016. *Mastodon*. <https://joinmastodon.org/>
- [42] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, et al. 2019. LogAnomaly: Unsupervised detection of sequential and quantitative anomalies in unstructured logs.. In *IJCAI*. <https://doi.org/10.24963/ijcai.2019/658>
- [43] Microservice.System.Benchmark. 2018. *TrainTicket*. <https://github.com/FudanSELab/train-ticket/>
- [44] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. 2018. An analysis of public REST web service APIs. *IEEE Transactions on Services Computing* 14, 4 (2018), 957–970. <https://doi.org/10.1109/TSC.2018.2847344>
- [45] NextCloud GmbH. 2016. *NextCloud*. <https://nextcloud.com/>
- [46] Anthonia Njoku, Heng Li, and Foutse Khomh. 2025. Kernel-Level Event-Based Performance Anomaly Detection in Software Systems under Varying Load Conditions. In *Companion of the 16th ACM/SPEC International Conference on Performance Engineering*. 26–30.
- [47] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. 2015. Detection of early-stage enterprise infection by mining large-scale log data. In *DSN*. <https://doi.org/10.1109/DSN.2015.14>
- [48] Michael Pradel and Thomas R Gross. 2009. Automatic generation of object usage specifications from large method traces. In *ASE*. <https://doi.org/10.1109/ASE.2009.60>
- [49] James E Prewett. 2003. Analyzing cluster log files using logsurfer. In *Proceedings of the 4th Annual Conference on Linux Clusters*. Citeseer State College, PA, USA, 1–12.
- [50] Jiaxing Qi, Shaohan Huang, Zhongzhi Luan, Shu Yang, Carol Fung, Hailong Yang, Depei Qian, Jing Shang, Zhiwen Xiao, and Zhihui Wu. 2023. LogGPT: Exploring ChatGPT for log-based anomaly detection. In *HPCC*. <https://doi.org/10.1109/HPCC-DSS-SmartCity-DependSys60770.2023.00045>
- [51] Lianyong Qi, Wenmin Lin, Xuyun Zhang, Wanchun Dou, Xiaolong Xu, and Jinjun Chen. 2022. A correlation graph based approach for personalized and compatible web apis recommendation in mobile app development. *IEEE Transactions on Knowledge and Data Engineering* 35, 6 (2022), 5444–5457. <https://doi.org/10.1109/TKDE.2022.3168611>

- [52] John P Rouillard. 2004. Real-time Log File Analysis Using the Simple Event Correlator (SEC). In *LISA*. <https://dl.acm.org/doi/10.5555/1052676.1052694>
- [53] Sudip Roy, Arnd Christian König, Igor Dvorkin, and Manish Kumar. 2015. PerfAugur: Robust diagnostics for performance anomalies in cloud services. In *ICDE*. <https://doi.org/10.1109/ICDE.2015.7113365>
- [54] Vilc Queupe Rufino, Mateus Schulz Nogueira, Alberto Avritzer, Daniel Sadoc Menasche, Barbara Russo, Andrea Janes, Vincenzo Ferme, Andre Van Hoorn, Henning Schulz, and Cabral Lima. 2020. Improving predictability of user-affecting metrics to support anomaly detection in cloud services. *IEEE Access* 8 (2020), 198152–198167.
- [55] Sigurd Schneider, Ivan Beschastnikh, Slava Chernyak, Michael D Ernst, and Yuriy Brun. 2010. Synoptic: Summarizing system logs with refinement. In *Workshop on Managing Systems via Log Analysis and Machine Learning Techniques (SLAML 10)*.
- [56] Salt Security. 2025. *State of API Security Report 2025*. Technical Report. Salt Security. <https://content.salt.security/state-api-report.html> Accessed: March 11, 2025.
- [57] Andrea Stocco and Paolo Tonella. 2020. Towards anomaly detectors that learn continuously. In *2020 IEEE international symposium on software reliability engineering workshops (ISSREW)*. IEEE, 201–208.
- [58] Shimin Tao, Yilun Liu, Weibin Meng, Zuomin Ren, Hao Yang, Xun Chen, Liang Zhang, Yuming Xie, Chang Su, Xiaosong Oiao, et al. 2023. Biglog: Unsupervised large-scale pre-training for a unified log representation. In *IWQoS*. <https://doi.org/10.1109/IWQoS57198.2023.10188759>
- [59] Emanuele Viglianisi, Michael Dallago, and Mariano Ceccato. 2020. Reststestgen: automated black-box testing of restful apis. In *ICST*. <https://doi.org/10.1109/ICST46399.2020.00024>
- [60] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring finite-state models with temporal constraints. In *ASE*. <https://doi.org/10.1109/ASE.2008.35>
- [61] Zhiwei Wang, Zhengzhang Chen, Jingchao Ni, Hui Liu, Haifeng Chen, and Jiliang Tang. 2021. Multi-scale one-class recurrent neural networks for discrete event sequence anomaly detection. In *KDD*. <https://doi.org/10.1145/3447548.3467125>
- [62] Xingfang Wu, Heng Li, and Foutse Khomh. 2023. On the effectiveness of log representation for log-based anomaly detection. *Empirical Software Engineering* 28, 6 (2023), 137.
- [63] Ming Xu, Chuanwang Wang, Jitao Yu, Junjie Zhang, Kai Zhang, and Weili Han. 2021. Chunk-level password guessing: Towards modeling refined password composition representations. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 5–20.
- [64] Ming Xu, Jitao Yu, Xinyi Zhang, Chuanwang Wang, Shenghao Zhang, Haoqi Wu, and Weili Han. 2023. Improving real-world password guessing attacks via bi-directional transformers. In *32nd USENIX Security Symposium (USENIX Security 23)*. 1001–1018.
- [65] Kenji Yamanishi and Yuko Maruyama. 2005. Dynamic syslog mining for network failure monitoring. In *KDD*. <https://doi.org/10.1145/1081870.1081927>
- [66] Lin Yang, Junjie Chen, Zan Wang, Weijing Wang, Jiajun Jiang, Xuyuan Dong, and Wenbin Zhang. 2021. PLELog: Semi-supervised log-based anomaly detection via probabilistic label estimation. In *ICSE*. <https://doi.org/10.1109/ICSE43902.2021.00130>
- [67] Wenqian Ye, Guangtao Zheng, Xu Cao, Yunsheng Ma, and Aidong Zhang. 2024. Spurious correlations in machine learning: A survey. *arXiv preprint arXiv:2402.12715* (2024). <https://arxiv.org/abs/2402.12715>
- [68] Ting-Fang Yen, Alina Oprea, Kaan Onarlioglu, Todd Leetham, William Robertson, Ari Juels, and Engin Kirda. 2013. Beehive: Large-scale log analysis for detecting suspicious activity in enterprise networks. In *ACSAC*. <https://doi.org/10.1145/2523649.2523670>
- [69] Yali Yuan, Sripriya Srikant Adhatarao, Mingkai Lin, Yachao Yuan, Zheli Liu, and Xiaoming Fu. 2020. Ada: Adaptive deep log anomaly detector. In *Ieee Infocom 2020-ieee Conference on Computer Communications*. <https://doi.org/10.1109/INFOCOM41043.2020.9155487>
- [70] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. 2021. WATSON: Abstracting Behaviors from Audit Logs via Aggregation of Contextual Semantics. In *NDSS*. <https://www.ndss-symposium.org/ndss-paper/watson-abstracting-behaviors-from-audit-logs-via-aggregation-of-contextual-semantics/>
- [71] Chenxi Zhang, Xin Peng, Chaofeng Sha, Ke Zhang, Zhenqing Fu, Xiya Wu, Qingwei Lin, and Dongmei Zhang. 2022. DeepTraLog: Trace-log combined microservice anomaly detection through graph-based deep learning. In *ICSE*. <https://doi.org/10.1145/3510003.3510180>
- [72] Xu Zhang, Yong Xu, Qingwei Lin, Bo Qiao, Hongyu Zhang, Yingnong Dang, Chunyu Xie, Xinsheng Yang, Qian Cheng, Ze Li, et al. 2019. Robust log-based anomaly detection on unstable log data. In *ESEC/FSE*. <https://doi.org/10.1145/3338906.3338931>
- [73] Nengwen Zhao, Junjie Chen, Zhaoyang Yu, Honglin Wang, Jiesong Li, Bin Qiu, Hongyu Xu, Wenchi Zhang, Kaixin Sui, and Dan Pei. 2021. Identifying bad software changes via multimodal anomaly detection for online service systems. In *ESEC/FSE*. <https://doi.org/10.1145/3468264.3468543>