Detecting and Explaining Anomalies Caused by Web Tamper Attacks via Building Consistency-based Normality

Yifan Liao[#]
Shanghai Jiao Tong University
Shanghai, China
National University of Singapore
Singapore, Singapore
yifan.liao@nusricq.cn

Ming Xu[#]
Shanghai Jiao Tong University
Shanghai, China
National University of Singapore
Singapore, Singapore
mingxu@nus.edu.sg

Yun Lin[*]
Shanghai Jiao Tong University
Shanghai, China
lin_yun@sjtu.edu.cn

Xiwen Teoh
National University of Singapore
Singapore, Singapore
xiwen@nus.edu.sg

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

Ruitao Feng
Singapore Management University
Singapore, Singapore
rtfeng@smu.edu.sg

Frank Liauw
Government Technology Agency of Singapore
Singapore, Singapore
Frank_LIAUW@tech.gov.sg

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

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

ABSTRACT

Web applications are crucial infrastructures in the modern society, which have high demand of reliability and security. However, their frontend can be manipulable by the clients (e.g., the frontend code can be modified to bypass some validation steps), which incurs the runtime anomaly when operating the web service. Existing state-of-the-art anomaly detectors largely learn a deep learning model from the collected logs to predict abnormal logs with a probability. While effective in general, those approaches can suffer from (1) inaccuracy caused by subtle difference between the normal and abnormal/attack logs and (2) additional efforts for root cause analysis.

In this work, we propose WebNorm, an anomaly detection approach to detect and explain the attack-caused anomalies on web applications in a unified way. Our rationale lies in learning the behaviorial normalities of a running web application as invariants. The normalities are designed regarding data normality (e.g., what information must be consistent across different events), flow normality (e.g., what events must happen under certain circumstances), and common-sense normality (e.g., what is the normal range of some parameter). The violation of the invariants indicates both the alarm and its explanation. WebNorm first monitors the normal behaviors of subject application and captures its information flows between entities such as frontend, service, and database. Then, it learns the behaviorial normalities in terms of logical rules so that it can detect and explain behaviorial anomaly by the inconsistency between the learned normalities and the runtime application behaviors. We model the invariants as first-order logics, transferrable to executable Python scripts to generate alarm with explainable root cause. Our extensive experiment showing that, on detecting the tamper attacks on the web applications as *TrainTicket* and *NiceFish*. WebNorm improves the precision and the recall of the baselines such as LogAnomaly, LogRobust, DeepLog, NeuralLog, PLELog, ReplicaWatcher by more than 56.1% and 35.1% respectively, serving as a new state-of-the-art anomaly detection solution.

ACM Reference Format:
Yifan Liao[#], Ming Xu[#], Yun Lin[*], Xiwen Teoh, Xiaofei Xie, Ruitao Feng, Frank Liauw, Hongyu Zhang, and Jin Song Dong. 2024. Detecting and Explaining Anomalies Caused by Web Tamper Attacks via Building Consistency-based Normality. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695024

1 INTRODUCTION

Recent years have seen that the web applications have played an important role in building various modern infrastructure such as government, bank, hospital, and even military applications [13, 14, 36, 41, 46], which has incurred a high demand of security and stability of their service. Any of their functional bugs and security vulnerabilities can cause tremendous loss of the business owners such as governments and banks.

A web application typically consists of a frontend (running in a client browser) and a backend (running in a server). Once it is delivered to the public, curious and knowledgeable users (or attackers) can comprehend, explore, and modify the frontend code,

[#] Both authors contributed equally to the paper.
[*] Corresponding author.

ASE '24, October 27-November 1, 2024, Sacramento, CA, USA

ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695024

potentially tampering the business logics of the web application in an unexpected way. For example, a blog [38] has discovered that Vistara Airlines suffered from payment bypass attacks where an attacker can manipulate input parameters to bypass the payment stage and obtain free goods or services. Those (even failed) explorative code-tampering behaviors are valuable for the service operators to (1) understand the potential vulnerability of the application and (2) be aware of whether and how the application is under exploitation. Since the frontend code can be tampered in a variety of ways, it leaves the runtime log analysis in the backend as an anomaly detection problem in DevOps.

Existing approaches adopt the state-of-the-art log analytic approach to monitoring and detecting the potential anomalies of the web applications. Practical industrial solutions such as Splunk [9], Elastic Stack [11] and IBM QRadar [18] report the anomaly based on their predefined and customized rules (e.g., based on HTTP status, log keywords, and statistical deviation). To capture abnormality in an automatic way, many researchers adopt deep-learning based solutions such as deep (graph) neural network models [10, 16, 31, 55] to learn the abnormal logs by preparing a training dataset in an either supervised or unsupervised ways. While existing solutions could be effective in a way, they are particularly struggling in detecting such tamper-attack-triggered anomalies:

- **C1 (Subtle change of abnormality):** First, unprecedented attacks can cause the generated logs change in a very subtle way, which can be easily ignored by traditional solutions. Given that the frontend code can be tampered in an unexpected way, it leaves great challenges to decide the granularity of the log features (e.g., tokenization [27], normalization [8], and word embedding [48]). Consequently, discriminative information could be abstracted away, or noisy information could be learned, resulting in a overfitting model.
- **C2 (Distribution-shift):** Second, the explorative code tampering behaviors can be evolving. Thus, it incurs challenges to the collect new datasets to update the learned model. The challenge is particularly large for collecting those false-negative logs.
- **C3 (Explainability):** Finally, the state-of-the-art deep learning models usually project a sequence of log into a suspiciousness score, leaving the programmers and application operators large efforts to apply the post-mortem analysis to pinpoint the root cause to make a timely counter-measure.

In this work, we propose, WebNorm, an LLM-based anomaly detection approach for web applications to detect and explain such tamper attack-triggered anomalies in an unified way. Our approach assumes that (1) any web applications need to be tested before their deployment and (2) the *normality* of the application is much more stable than its novel attack. Thus, we collect the runtime logs of a target application in the testing-stage and construct their behavioral normalities in the form of first-order logic. The normalities are designed regarding data normality (e.g., what information shall be consistent across different events), flow normality (e.g., what events shall happen under certain circumstances), and common-sense normality (e.g., what is the normal range of some parameter). For example, WebNorm can infer an invariant that "*the price of an item retrieved from the database should be consistent with the price passed from the front end*". Any violation of the invariant (e.g., the price tampered in the frontend code) can raise an alarm with an explanation (e.g., the price inconsistency in this case).

Technically, WebNorm interleaves between the program analysis, log analysis, and LLM interaction to build the invariants, by capturing the information flows between entities such as frontend, service, and database in the web application. WebNorm maps the bankend code to its generated logs, to address the long-context problem, in comparison to purely inferring the logs. Note that, there can be hundreds (if not thousands) of logs in between the log to retrieve data (e.g., *price*) from the database and the log to receive relevant data from the frontend. Then, WebNorm interacts with LLM to identify the potential entities in the log to build consistency relation. Those invariants are learned as consistency-based rules, translated to executable Python scripts, to check against the collected raw logs to capture any subtle behaviorial changes.

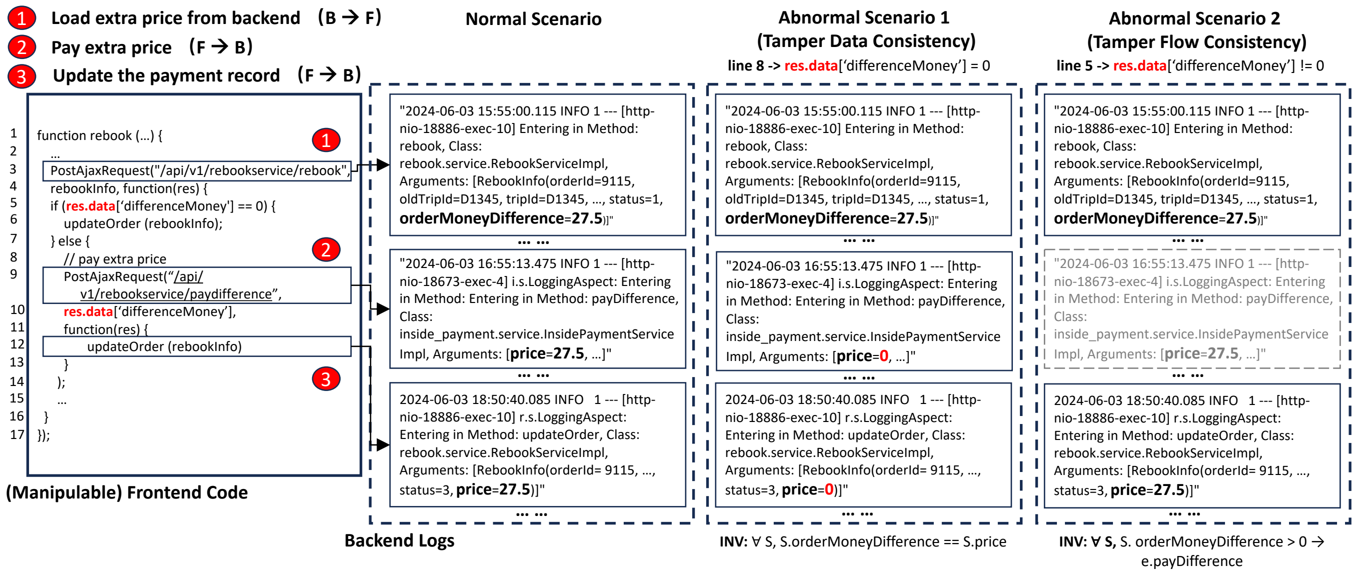
We construct the extensive facilities for our evaluation on WebNorm on the state-of-the-art benchmark as *TrainTicket* [56] and *NiceFish* [39]:

- **Seeding/Testing Scenarios**: A set of pre-defined seeds (i.e., normal scenarios) in addition to Industry Fault in the benchmark for building our reference behavioral model.
- **Abnormal Scenarios**: An attack toolkit towards the *TrainTicket* and *NiceFish* benchmark, called `TT-Attack` dataset, consisting of more than 40 types of tamper attacks.

We evaluate the performance of WebNorm by comparing it with the state-of-the-art baselines such as LogAnomaly [37], LogRobust [54], DeepLog [10], NeuralLog [28], PLELog [51], ReplicaWatcher [21], regarding their precision, recall, and F1-score. The results show that (1) WebNorm exhibits significant improvement over the baseline by 56.1% increase in precision and 35.1% increase in recall over the state of the art methods at reasonable cost of runtime overhead, (2) WebNorm achieves the explanation accuracy of 92.3%, demonstrating its effectiveness for root cause analysis, and (3) the precision of WebNorm can be robust against the perturbation of the number of seeding scenarios and its recall can be largely preserved when the seeding scenarios can achieve a minimum coverage of 50% of the system.

In summary, we make the following contributions:

- We propose, WebNorm, a solution for defecting and explaining the tamper-attack triggered anomalies of a web application, by addressing the challenge of subtle changes of abnormalities and explainability. WebNorm learns the invariants on raw logs as consistency rules (i.e., data consistency, flow consistency, and common-sense consistency) in the form of first-order logics. Any violation of the invariant indicates both the alarm and the explanation.
- We make a comprehensive evaluation upon the state-of-the-art *TrainTicket* and *NiceFish* benchmark. We identify more than 40 types of tamper attack towards the benchmark based on its vulnerability, which can lay a foundation for the follow-up anomaly detection in the community.
- We deliver WebNorm as a tool, deployable to help the practitioner identify the root cause in practice. A tool demonstration is available at [40].


1 Load extra price from backend (B → F)
2 Pay extra price (F → B)
3 Update the payment record (F → B)
Normal Scenario
Abnormal Scenario 1
(Tamper Data Consistency)
line 8 -> res.data[‘differenceMoney’] = 0
Abnormal Scenario 2
(Tamper Flow Consistency)
line 5 -> res.data[‘differenceMoney’] != 0
(Manipulable) Frontend Code
Backend Logs
INV: ∀ S, S.orderMoneyDifference == S.price
INV: ∀ S, S. orderMoneyDifference > 0 → e.payDifference

```
1  function rebook (…) {
2    …
3    PostAjaxRequest("/api/v1/rebookservice/rebook",
4    rebookInfo, function(res) {
5    if (res.data[‘differenceMoney'] == 0) {
6      updateOrder (rebookInfo);
7    } else {
8      // pay extra price
9      PostAjaxRequest(“/api/
         v1/rebookservice/paydifference”,
10     res.data[‘differenceMoney’],
11     function(res) {
12         updateOrder (rebookInfo)
13     }
14    );
15    …
16  }
17 });
```

Normal Scenario:

```
"2024-06-03 15:55:00.115 INFO 1 --- [http-nio-18886-exec-10] Entering in Method: rebook, Class: rebook.service.RebookServiceImpl, Arguments: [RebookInfo(orderId=9115, oldTripId=D1345, tripId=D1345, …, status=1, orderMoneyDifference=27.5)]"
… …
"2024-06-03 16:55:13.475 INFO 1 --- [http-nio-18673-exec-4] i.s.LoggingAspect: Entering in Method: Entering in Method: payDifference, Class: inside_payment.service.InsidePaymentServiceImpl, Arguments: [price=27.5, …]"
… …
2024-06-03 18:50:40.085 INFO   1 --- [http-nio-18886-exec-10] r.s.LoggingAspect: Entering in Method: updateOrder, Class: rebook.service.RebookServiceImpl, Arguments: [RebookInfo(orderId= 9115, …, status=3, price=27.5)]"
… …
```

Abnormal Scenario 1:

```
"2024-06-03 15:55:00.115 INFO 1 --- [http-nio-18886-exec-10] Entering in Method: rebook, Class: rebook.service.RebookServiceImpl, Arguments: [RebookInfo(orderId=9115, oldTripId=D1345, tripId=D1345, …, status=1, orderMoneyDifference=27.5)]"
… …
"2024-06-03 16:55:13.475 INFO 1 --- [http-nio-18673-exec-4] i.s.LoggingAspect: Entering in Method: Entering in Method: payDifference, Class: inside_payment.service.InsidePaymentServiceImpl, Arguments: [price=0, …]"
… …
2024-06-03 18:50:40.085 INFO   1 --- [http-nio-18886-exec-10] r.s.LoggingAspect: Entering in Method: updateOrder, Class: rebook.service.RebookServiceImpl, Arguments: [RebookInfo(orderId= 9115, …, status=3, price=0)]"
… …
```

Abnormal Scenario 2:

```
"2024-06-03 15:55:00.115 INFO 1 --- [http-nio-18886-exec-10] Entering in Method: rebook, Class: rebook.service.RebookServiceImpl, Arguments: [RebookInfo(orderId=9115, oldTripId=D1345, tripId=D1345, …, status=1, orderMoneyDifference=27.5)]"
… …
"2024-06-03 16:55:13.475 INFO 1 --- [http-nio-18673-exec-4] i.s.LoggingAspect: Entering in Method: Entering in Method: payDifference, Class: inside_payment.service.InsidePaymentServiceImpl, Arguments: [price=27.5, …]"
… …
2024-06-03 18:50:40.085 INFO   1 --- [http-nio-18886-exec-10] r.s.LoggingAspect: Entering in Method: updateOrder, Class: rebook.service.RebookServiceImpl, Arguments: [RebookInfo(orderId= 9115, …, status=3, price=27.5)]"
… …
```

Figure 1: A motivating example showing how the backend anomalies can happen when a curious user (or attacker) explores and modifies the frontend code. The example is taken from the scenario of rebooking a ticket to pay extra price in the *TrainTicket* system. The difference between the logs in the normal scenario and that in the attack scenario is very subtle. In *Abnormal Scenario 1* where the user avoids paying extra price by changing a variable in the frontend, the log difference is a change of integer number (from 27.5 to 0, in red).

- Our extensive experiments show the precision and soundness of the WebNorm. Specifically, WebNorm significantly reaches a high accuracy on the ananomaly detection.

More of the tool videos and other materials are available at [40].

2 MOTIVATING EXAMPLE

Figure 1 shows the log examples, normal and abnormal, on a ticket rebooking scenario in the *TrainTicket* system [56], a popular web application used as benchmark for various DevOps tasks.

Normal Scenario. In such a ticket-rebooking scenario, a user can change its reservation after he or she has booked a ticket. If the new ticket has a different price, he or she need to pay extra price (line 8-13). Otherwise, the system just update the order information without charging the user with extra payment. As showed in the Figure 1, the frontend code can be summarized into three steps (in red circules):

- **Step 1: Load Price (Backend to Frontend).** The system loads the price information and send the calculated extra price to the frontend.
- **Step 2: Payment (optional, Frontend to Backend).** The frontend check whether the extra price is larger than 0. If yes, it invokes the payment service, asking the user to accomplish the additional payment.
- **Step 3: Update Order Information (Frontend to Backend).** The system update the rebooking information (including the extra price information) in the system.

In Figure 1, each interaction between the frontend and the backend derives a log at the backend. We can see that the detailed rebooking information is logged, including the order id (i.e., `orderId=9115`), trip id (i.e., `tripID=D1345`), extra price to pay (i.e., `orderMoneyDifference=27.5`). [1] Next we illustrate how the logs can happen when the curious user explores and modifies the frontend code to avoid the extra payment.

Abnormal Scenario 1 (Tampered Data Consistency). The first tamper can change line 8 in Figure 1 by setting the price of the extra price to be 0 (i.e., `res.data[‘differenceMoney‘]=0`). In this case, even if the payment service is invoked (line 9), the user still pays no extra price. The resulted logs is showed in Figure 1, leaving the logs very similar to that in normal scenario, expect that the `price` parameter in the log is recorded as 0.

Abnormal Scenario 2 (Tampered Flow Consistency). The second tamper can change line 5 in Figure 1 by setting the condition in Javascript code (i.e., `res.data[‘differenceMoney’] != 0`). By this means, the frontend code can no longer exercise the branch to invoke the payment service. As a result, in comparison to the logs in the normal scenario, the resulted logs in the backend miss one log on invoking the payment service.

Challenges. The above examples render very high similarity between logs in normal and abnormal scenarios. Their discriminative features can involve either very detailed parameter value such as `price=0` and `orderMoneyDifference=27.5`, or the existence of *log flows* conditioned on the valuation of specific parameter such as whether the parameter `price` is larger than 0. Those features is domain-specific and logical. Even worse, there could be many logs happen in between those steps, which incurs a long context between critical events. All the above incur great challenges for

[1] We will discuss how we address the observability challenges in generating the logs via program analysis in Section 3.1

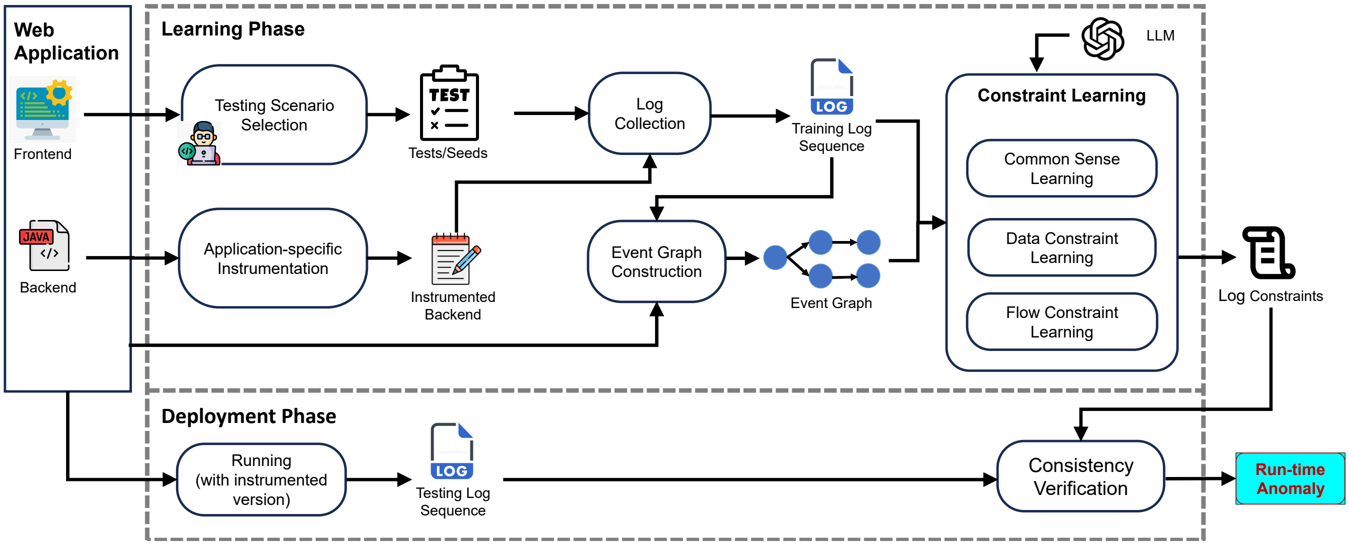


Figure 2: Overview of the design of WebNorm, which extracts the log normalities as first-order logic constraints in the learning phase. The log constraints are used to validate the runtime logs in the deployment phase.

any inductive approach, for example, to learn a deep (language) model in either supervised or unsupervised way.

Solution and its Rationale. In this work, we propose WebNorm to learn *logical* normality to infer the abnormalities, based on the expected *data consistency*, *flow consistency*, and *common-sense consistency* when the web application is operating, which is more deductive approach in comparison to the state-of-the-arts. Our approach consists of a learning phase and a deployment phase. By mapping the static code and their derived logs, WebNorm learns the potential relation between the parameters across different logs. The code analysis allows us to associate the logs (and their events) even if there are a large number of logs happen in-between them. For example, we can infer that the parameter `orderMoneyDifference` (passed from the backend to the frontend) is semantically equivalent to the parameter `price` (passed from the frontend to the backend). In addition, we confirm with LLM on the potential consistency relation between the parameters. As a result, we can build an invariant such as "$\forall s, s.orderMoneyDifference = s.price$" where s is a rebooking session. Similarly, by building the conditional flow consistency relationship between the frontend code and the backend logs in the learning phase, we can build an invariant such as "$\forall s, s.orderMoneyDifference > 0 \rightarrow e.payDifference$", indicating that a log event of paying extra price should happen if the parameter `orderMoneyDifference` is greater than 0.

Note that, those invariants are learned for just once. Then, we can validate the logs in the deployment phase. In addition, the violation of the learned invariants can serve as both the alarm and the explanation to facilitate the follow-up root cause analysis.

3 APPROACH

Figure 2 shows the overview of WebNorm to report runtime anomalies, consisting of a learning phase and a deployment phase. We assume that a web application (especially in the industrial settings) can have a set of representative (GUI) test cases (or seeds) to test its normal functionalities. Thus, given a target web application, we run the (GUI) test cases against its instrumented version[2] to collect the raw logs (**Log Collection**). For necessary observability, we instrument the application to capture the interaction between the web components such as service, database, and frontend. Technically, each called API with their runtime parameter values is instrumented to record an *event*, as showed in Figure 3 (**Application-specific Instrumentation**). As a result, one GUI test case can result in a sequence of event as raw logs. Then, we analyze the source code of both frontend and backend to *link* the relevant events by attaching the control and data flow to convert the event sequence to an event graph (**Event Graph Construction**). Subsequently, we parse the structured log information to derive a list of log constraints in the form of first-order logics (**Constraint Learning**). Each log constraint can be translated to executable Python script to detect and explain the runtime anomalies in the deployment phase (**Consistency Verification**).

Instrumented Logs (Rebook Service as an Example)

Input = RebookInfo (oldTripId=D1345, tripId=D1345, seatType=2, date=2024-06-03)
API = Rebook.service.RebookServiceImpl
Output = Response (orderMoneyDifference=27.5)

Figure 3: An example of instrumented logs for the Rebook API as an event, consisting of API name, and the runtime valuation of the input and the output when it is called.

3.1 Event Graph Construction

An event graph captures a variety of relations between the events of an operating web application. Formally, an event graph can be defined as $G = \langle Evt, R \rangle$ where (1) each $e \in Evt$ is an API call

[2] The implementation of instrumentation (e.g., AOP [3]) is transparent to the developers of the target web application.

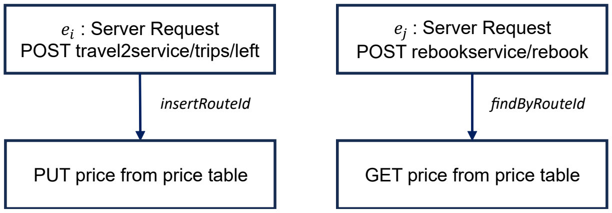

e_i : Server Request
POST travel2service/trips/left
insertRouteId
PUT price from price table
e_j : Server Request
POST rebookservice/rebook
findByRouteId
GET price from price table

Figure 4: The example of two database sharing events ($e_i \preceq e_j$)

with runtime valuation of its input and output, and (2) each $r \in R$ ($R \longleftarrow Evt \times Evt$) is a relation defined between two events. Further, we denote the set of input and output of an event e as $e.in$ and $e.out$ respectively. For example, in Figure 3, the input is a `RebookInfo` object (consisting of `oldTripId`, `tridId`, `seatType`, and `date`) and the output is a `Response` object with the extra payment information (i.e., `orderMoneyDifference`). We define three types of relations as follows.

- **DB Sharing:** Given two events (i.e., API call), denoted as e_i and e_j on an event sequence, we call e_i and e_j have a relation of *DB sharing* if e_i writes a data item to DB and e_j query the DB for the same data item, where $e_i \prec e_j$. Here, we use $\prec$ indicates the e_i happens before e_j. The relation demonstrates that there could be a read/written relation between these events.
- **Data Transition:** Given two events e_i and e_j, we call e_i and e_j have a relation of *data transition* if $\exists p \in e_i.out, q \in e_j.in$ ($e_i \prec e_j$) and $p = q$ and there $\nexists e_k (e_i \prec e_k \prec e_j)$ so that $\exists x \in e_k.out, q \in e_j.in, x = q$.
- **Trigger Condition:** Given two events e_i and e_j on an event sequence, we call e_i and e_j have a relation of *trigger condition* if $\exists p \in e_i.in$ can decide whether e_j can happen or not.

Metaphorically speaking, the relation of *DB sharing* and *data transition* is similar to the concept of data dependency in program analysis. In contrast, the relation of *trigger condition* is similar to that of control dependency. In this work, we conduct program analysis to parse the data and control dependency from the backend code, and map the dependency relations to their corresponding events on the logs. Note that, program analysis has its limitation to achieve precise results. Therefore, we conduct may-analysis to ensure that we can complete but unsound results. Based on the potential relation, we extract more precise invariants by interacting with LLM (see Section 3.2).

Shared Database Extraction. We first identify whether an event is database-relevant by defining a list of DB library calls (e.g., JDBC [19], JPA with Hibernate [50], and Spring Data JPA [20]). Then, given two events e_i and e_j, we parse the calls by tracking whether they share a relation by tracking whether they are processing a table column with the same name. For example, as shown in Figure 4, there are two event calling different APIs, i.e., e_i = `POST /api/v1/rebookservice/rebook` and e_j =`POST /api/v1/travel2service/trips/left`. Both events query from the same table `price`, by `insertRouteIds` and `findByRouteId`) respectively. Note that, the table column name (i.e., `price`) accessed by those services should be identical, ensuring the events query the same database.

Data Transition Extraction. We first parse the backend code into static data flow graph, where the nodes of API call are particularly labelled. Specifically, we denote the data flow graph as

```
function loadArticle(userId, articleId) {
try {
    const userInfo = getUserDetail(userId);
    const articleContent = getPostDetail(articleId);
    // Check user's role
    if (userInfo.role === "Premium") {
        showContent(articleContent);
    } else {
        showpaywall(mainPage);
    }
}
```

Figure 5: A code example at the frontend on Nicefish

$G_d = \langle I, V, R, W \rangle$ where each node $n \in I$ represents a program instruction, each node $v \in V$ represents a local or global variable defined or used in the program, each edge $r \in R$ ($R \longleftarrow N \times V$) represents that a *read* relation between an instruction and a variable, and each edge $w \in W$ ($W \longleftarrow N \times V$) represents that a *write* relation between an instruction and a variable. Then, we map the those API call nodes (or instruction) back to the raw event sequence. Note that, each node in the event sequence is derived by executing an API call. Assume that e_i is mapped to an instruction node i_a, e_j is mapped to an instruction node i_b. If $\exists v \in V$ so that $(i_a, v) \in W$ and $(i_b, v) \in R$, we build a *data transition* relation for e_i and e_j.

Trigger Condition Extraction. We analyze the trigger condition from the frontend code, to identify the if an API call belongs to another API's trigger. For example, as shown in Figure 5, the output of `POST /api/v1/getUserInfo` (i.e., `userInfo.role`) should serve as a trigger condition to the API calls of `POST /api/v1/showContext` and `POST /api/v1/showPaywall`. To this end, we build the link between the frontend code (i.e., ground truth workflow) and the event sequence derived by its execution.

Given two event sequences s_1 = ⟨`getUserDetail`, `getPostDetail`, `showPaywall`⟩ and s_2 = ⟨`getUserDetail`, `getPostDetail`, `showContent`⟩, each derived by executing different branches in the frontend code in Figure 5. Note that, for each event, we also include its runtime parameters and values. Then, we compare s_1, s_2, and the frontend code snippets, to infer if there is a trigger condition relationship between two events. We locate the code snippets where the branch occurs and then use GPTs to determine the trigger relationships between `getUserInfo` and `showContent` or `showPaywall`. In this case, we can build the relation between the event `getUserDetail` (with parameter `role`) and the event `showContent`. By this means, we build the relation for any pairs of the events in the sequence, as the event graph. Note that, those coarse relations are the result of *may*-analysis.

3.2 Constraint Learning

Figure 6 shows how we parse event relations into executable invariants with LLM for runtime validation. The process serves two goals, i.e., (1) relevant parameter discrimination and (2) invariant generation. As for parameter discrimination, given a subset of events (or logs) annotated with a relation (e.g., *DB-sharing*, *data transition*, or *trigger condition*), we use LLM to select the most relevant and

Table 1: The prompt template for generating data consistency invariants, the code (in red text) and the log samples (in blue text) are to be filled.

<Background>:
You are a software engineer that is extremely good at modelling entity relationships in databases. You SHOULD first provide your step-by-step thinking for solving the task. Your thought process should be enclosed using "<thought>" tag.
<Task>:
Here are the definition of two classes [A] and [B] :
Class [A] and its attributes: {class_definition1}
Class [B] and its attributes: {class_definition2}
Instances of both classes [A] and [B] can be found in these logs:
{log samples}

<Guidelines>:
Based on the logs, infer the possible relationships of attributes in [A] and [B] by referencing these common types of relationships: 1. Foreign key: an attribute in an entity that references the primary key attribute in another entity, both attributes must be the same data type. 2. Primary key: attribute(s) that can uniquely identify entities in an entity set. 3. Matching: an attribute(E.g: Price, ID) in an entity that must have the same value as an attribute in another entity, both attributes must be the same data type. (E.g: Price, ID)
You SHOULD construct as many of the most important first-order logic constraints and output it in general format. Examples:
- ∀ x (isDog(x) ⟵ hasFourLegs(x)
- ∀ x (isPerson(x) ⟵ ∃ y (isDog(y) ∧ owns(x, y)))
- ∀ x ∃ y ((isParent(x, y) ∧ isMale(x)) ⟵ isFather(x, y))
- ∃x (isHuman(x) ∧ loves(x, Mary))
- ∀ x (isStudent(x) ∧ studiesHard(x) ⟵ getsGoodGrades(x))
- ∀ x (isAnimal(x) ⟵ (∃ y (isFood(y) ∧ eats(x, y))))

Then, write a function that determines if instances of [A] and [B] are related to each other using their attributes.

<Few-shot Examples>:
```
def is_related(instance_A: dict, instance_B: dict) -> bool:
if instance_A.conditionA != instance_A.conditionB:
raise ValueError('instance_A and instance_A should have the same condition A and B')
return True
```

<Expected Results>: //to-be-generated results

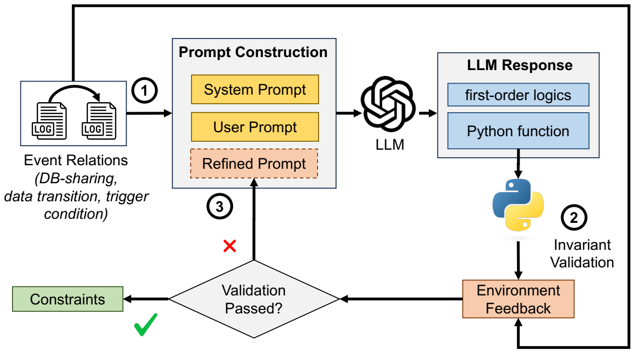


Figure 6: Constraint learning process of WebNorm, consisting of event relation selection, invariant generation, and invariant validation.

useful parameters in the event to construct the invariant of different types (i.e., data consistency, flow consistency, and common-sense consistency). As for the invariant generation, we adopt the practice of chain-of-thought to derive the invariant in the form of both first-order logics and Python script. The generated Python scripts are used to run against the prompt-constructing logs to see whether the scripts can parse them to be the normal logs. If not (because of either the runtime exception or failed instances), we can refine the prompt and ask LLM to regenerate the invariants. Note that, the prompt-refining step can be regarded as few-shot learning for the most *fit* Python script given a subset of events. Given a predefined iteration number and a selected subset of logs, the prompt-refining loops will either end up with a confirmed constraint, or report no constraint for the log subset.

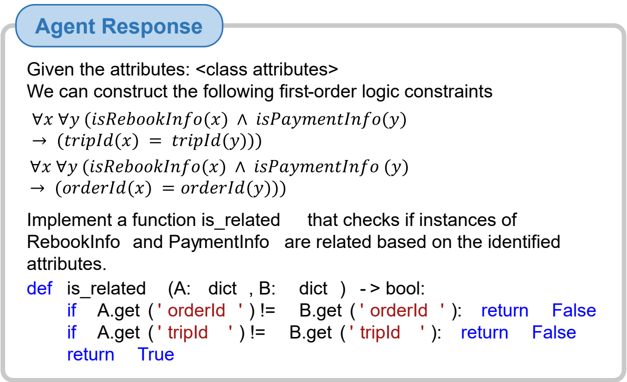
3.2.1 *Constraint Types.*

Given that each event e has its own corresponding API, we denote the API signature of e as $e.API$ (see Figure 3 as an example), which is called as *event type*.

Data Consistency. We say there is a **data consistency** between two event types API_1 and API_2 if $\forall\langle e_i, e_j\rangle$, $e_i \prec e_j$, $e_i.API = API_1$, $e_j = API_2$, $\exists p \in e_i.out, q \in e_j.in$, so that $value(p) = value(q)$. Here $value(.)$ is a valuation function to have the runtime value of a parameter. The data consistency is denoted as $DC(API_1(p), API_2(q))$. For example, in Figure 1, in all the events, we can observe that the value of `price` and that of `orderMoneyDifference` always share the same value in the normal scenarios.

Note that, the data consistency exists for those relations of type *DB-sharing* or *data transition* relations. *DB-sharing* or *data transition* relations are defined on event instances, while the data consistency relation is defined on event types. Further, the data consistency is transitive because of the equality relation in its definition.

Flow Consistency. We say there is **flow consistency** between two event types API_1 and API_2 if $\forall\langle e_i, e_j\rangle$, $e_i \prec e_j$, $e_i.API = API_1$, $e_j =$



Agent Response

Given the attributes: <class attributes>
We can construct the following first-order logic constraints

$\forall x\, \forall y\, (isRebookInfo(x) \land isPaymentInfo(y) \rightarrow (tripId(x) = tripId(y)))$

$\forall x\, \forall y\, (isRebookInfo(x) \land isPaymentInfo\,(y) \rightarrow (orderId(x) = orderId(y)))$

Implement a function is_related that checks if instances of RebookInfo and PaymentInfo are related based on the identified attributes.

```
def is_related (A: dict , B: dict ) -> bool:
    if A.get ('orderId') != B.get ('orderId'): return False
    if A.get ('tripId') != B.get ('tripId'): return False
    return True
```

Figure 7: Data constraint prompt and an example of a agent-environment interaction turn.

API_2, $\exists cond$ as a condition defined on $e_i.in$ so that if $cond(e_i.in) = true$, e_j need to appear in the log sequence within k events after e_i appears. We denote it as $FC(API_1, API_2, k)$

Common-sense Consistency. We say there is **common-sense consistency** on an event type API if $\forall e$, $e.API = API$, $\exists cond$ as a condition defined on $e.in$ so that $cond(e.in)$ is always true. We denote it as $CSC(API)$.

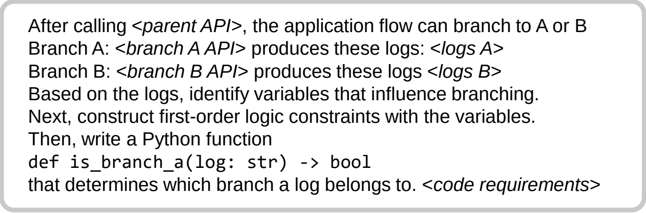
The above definition allows us to filter candidate sets of event instances or event relations, which are potentially generated because of an invariant (of type of data consistency, flow consistency, and common-sense consistency). We first narrow down the scope of candidates by tracking their relation on event instances. Specifically, given a relation $r = \langle e_i, e_j \rangle$ ($e_i \prec e_j$), r can only contribute to learn data consistency if it is *DB-sharing* or *data transition* relation, and learn flow consistency if it is *condition trigger* relation. Then, we filter the candidate sets according to the above definition on event types.

3.2.2 Prompt Construction to Generate Invariants. Then, we synthesize different prompts for different types of invariants. Given a candidate set, we parse its relevant code information and log samples to fill in our pre-defined prompt template.

Table 1 shows our prompt template to generate data consistency invariants. We adopt the practice of chain-of-thought and in-context learning to derive both the first-order logic and a Python script. As for the chain-of-thought design, we provide the *step-wise* thought instruction (see <Background> in Table 1) and ask LLM to generate the first-order logics followed by the Python scripts. As for the in-context learning, we provide examples (of both first-order logics and Python scripts) under a variety of scenarios to help LLM to output the results following strict format. As a result, LLM can output the response as showed in Figure 7.

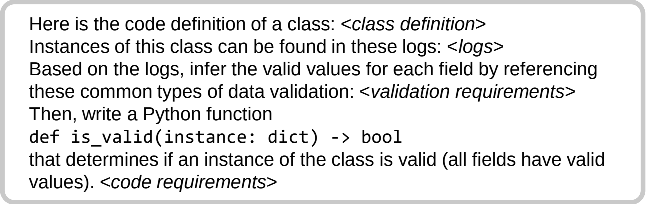
Similarly, we construct prompt template for flow consistency and common-sense consistency invariants as showed in Figure 8 and Figure 9. In both figures, we use <> (e.g., `<class attributes>`) as the placeholder to fill in relevant code and log information. Given the space limit, interested audience can refer to [40] for more detailed prompt templates. As for the common-sense constraints, we ask LLM to focus on the following perspectives:

- **Presence check**: important fields should not be empty (e.g., ID)



After calling *<parent API>*, the application flow can branch to A or B
Branch A: *<branch A API>* produces these logs: *<logs A>*
Branch B: *<branch B API>* produces these logs *<logs B>*
Based on the logs, identify variables that influence branching.
Next, construct first-order logic constraints with the variables.
Then, write a Python function
`def is_branch_a(log: str) -> bool`
that determines which branch a log belongs to. *<code requirements>*

Figure 8: Simplified prompt to generate flow consistency invariants



Here is the code definition of a class: *<class definition>*
Instances of this class can be found in these logs: *<logs>*
Based on the logs, infer the valid values for each field by referencing these common types of data validation: *<validation requirements>*
Then, write a Python function
`def is_valid(instance: dict) -> bool`
that determines if an instance of the class is valid (all fields have valid values). *<code requirements>*

Figure 9: Simplified prompt to generate common-sense constraints

- **Data Type Check**: is the field a valid data type? (e.g., integer for numbers)
- **Code Check**: does the value fall within a valid list of values? (e.g., postal codes, country codes, NAICS industry codes)
- **Range Check**: does the value fall within a logical numerical range? (e.g., temperature, latitude, price).
- **Format Check**: does the value follow a predefined format? (e.g., UUID, email, phone number, country codes).
- **Consistency Check**: are two or more values logically consistent with each other? (e.g., delivery date must be after shipping date).
- **Length Check**: does the value contain a correct number of characters? (e.g., password).

3.2.3 Invariant Refinement. To improve the quality of learned constraints and reduce hallucination, we design a test-driven approach for the agent to self-correct its code iteratively. Given a candidate set of events and relations C, we divide it into C_1 (for generating invariants, similar to the concept of training dataset) and C_2 (for validating and correcting invariants, similar to the concept of testing/validation dataset) where $C = C_1 \cup C_2$. The LLM agent interacts with the environment in a multi-turn setting, where the environment is a Python code interpreter equipped with a unit testing toolkit. The agent submits the generated code to the environment for testing.

We can translate C_2 to a set of test cases T. Then, $\forall t \in T$, we run t against the generated python script *code*, to derive the runtime message $msg(t)$. If any of the test cases have a failure message, we append its failure message as a part of the prompt to ask LLM to regenerate the results. In the meantime, we can have one more element in C_1 and one less in C_2. Given a threshold th, we interactively and iteratively run the procedure up to th times. We do not generate any invariants if the budget of all th times are used up. Otherwise, we record the generated invariants as consistency valuation rules in the deployment time.

4 EVALUATION

In this section, we aim to evaluate our WebNorm's performance, focusing on the following research questions.

- **RQ1**: How is the performance of WebNorm to detect attack-triggered anomalies comparing to the state-of-the-art anomaly detectors?
- **RQ2**: What is the cost of WebNorm to learn invariants and detect anomalies?
- **RQ3**: Whether WebNorm provide accurate explanations to pint point the root cause?
- **RQ4**: How does seed in the learning phase impact the performance of WebNorm?

4.1 Experiment Setup

Baselines. We compare WebNorm with 6 state-of-the-art log-based anomaly detectors (see Table 3) to evaluate the effectiveness of our WebNorm. We show our implementations of our baseline methods as follows and show the basic information in Table 3. Then, we select the existing anomaly detection methods from the similarity-based and deep-learning-based perspective. In the similarity-based approach, we select the recent proposed ReplicaWatch, which features as a training-less method and detects anomalies in containerized microservices. Their principle is to compare the similarity between the microservice's several replicas (i.e., sub microservices), and output an alarm when the inconsistency within the internal replicas reaches a certain threshold, highlighting a training-less manner. To compare with this approaches, we regard the log from an API function as a replica's log, and follow the same threshold 0.5 in our experiment, enabling the observation of performance in training-free rational approaches. We also select five deep-learning based approach, whose principle predicts the next normal events of a log based on the previous events of a log.

These approaches are proposed from between 2017 to 2021. For example, LogRobust [54], proposed in 2019, utilizes an attention-based Bi-LSTM model to identify the varying importance between different log events, providing robust anomaly detection in dynamic log environments. Similarly, DeepLog [10] uses similar methodology with the LSTM. LogAnomaly [37], proposed in 2019, shares similar principle with the DeepLog. LogAnomaly treats a log as an event sequence and considers the counts of different log events as an additional feature. It adopts an LSTM model to learn sequential and quantitative patterns. We follow the same parameter selection as the open-source code [29].

Table 2: The size of the evaluation logs in two platforms. All training logs are normal logs. N refers to normal, A refers to attack in seconds.

TrainTicket			*NiceFish*		
Training (N)	Testing (N)	Testing (A)	Training (N)	Testing (N)	Testing (A)
183,232	72,611	892	74,288	34,555	318

Table 3: The description of baseline models.

	Baseline Approach	Year	Model
Similarly Based [4]	Replica	2024	Training-less
Deep Learning Based [5]	LogRobust	2019	Bi-LSTM
	NeuralLog	2019	Transformer
	PLELog	2021	HDBSCAM+Cluster
	DeepLog	2017	LSTM
	LogAnomaly	2019	LSTM

Benchmark. Our evaluation is based on two real-world web applications, i.e., (1) one is *TrainTicket* v1.0.0 [56], a popular benchmark used in many DevOps tasks and (2) the other is the digital services platform *NiceFish* [39], which are commonly-used open-source platform. To collect the training dataset for all the baselines, we manually define 22 normal scenarios (i.e., seeds) in *TrainTicket* and 11 normal scenarios in *NiceFish*. Then we construct the testing dataset (with both normal and abnormal logs). As for the normal logs, we generate the normal scenarios by employing GPT-4 to simulate user interactions based on different pre-defined tasks [57]. As for the abnormal logs (i.e., attack logs), we refer to the literature [24] and the OWASP [3] to simulate the relevant attack scenarios in two websites. Specifically, we simulate 43 attack scenarios in *TrainTicket* and 14 scenarios in *NiceFish*. Table 2 shows the details of our benchmark. More of the details (e.g., dataset, replication package, and runtime configurations) are available at [40].

In the experiment, we employ the same sliding window (k=20) to segment logs and feed them to train deep-learning-based baseline models. In the similarity-based baseline models and our approach, we input all logs, given that the LLM-agent-based and similarity-based models can capture a log.

Evaluation metrics. We use the precision, recall, and F1-score to measure the effectiveness of anomaly detection based on TP (True Positive), FP (False Positive), and FN (False Negative).

- **Precision:** the percentage of anomalous logs out of all logs detected as anomalies, represented as precision = $\frac{TP}{TP+FP}$.
- **Recall:** the percentage of all anomalous logs that are detected as anomalies, represented as recall = $\frac{TP}{TP+FN}$.
- **F1-Score:** the harmonic mean of precision and recall, represented as $F_1 = 2 \cdot \frac{\text{Precision}\cdot\text{Recall}}{\text{Precision}+\text{Recall}}$.

Further, given an abnormal log (or event sequence) in the ground-truth testing dataset, we can compare it with its normal version. Note that, the abnormal logs are achieve by applying tamper attacks. Then, we can manually compare the log difference with the reported explanation. Thus, we calculate the explanation accuracy rate by $\frac{M}{N}$, where M is the number of reported true anomalies and N is the number of the true anomalies with the true explanation.

4.2 RQ1: The Effectiveness of WebNorm

Table 4 shows the results where we can find that WebNorm can significantly outperform the baseline models. Overall, WebNorm outperforms all the other approaches in terms of precision and recall. We observe that all the baselines generally suffer from the limited abstraction from logs and context of relevant events to report the anomalies. Note that, many attack-triggered anomalies render very subtle difference from the normal logs, which incurs challenges for all the inductive solutions. In contrast, WebNorm is designed based on a deductive principle, which can well capture the appropriate log granularity and is sensitive to the subtle changes of the abnormal logs. Further, some anomalies are rendered in a

[3] https://owasp.org/www-community/attacks/Web_Parameter_Tampering

Table 4: The effectiveness of WebNorm.

Approach	*TrainTicket*			*NiceFish*		
	F1	Precision	Recall	F1	Precision	Recall
WebNorm	**0.918**	**0.929**	**0.907**	**0.922**	**0.928**	**0.917**
LogRobust	0.558	0.477	0.671	0.605	0.500	0.765
ReplicaWatcher	0.169	0.098	0.593	0.012	0.382	0.012
NeuralLog	0.103	0.522	0.057	0.042	0.772	0.022
PLELog	0.099	0.055	0.485	0.091	0.049	0.700
DeepLog	0.098	0.595	0.053	0.025	0.072	0.015
LogAnomaly	0.094	0.593	0.051	0.025	0.072	0.015

Table 5: The effectiveness of each scenario on *TrainTicket*.

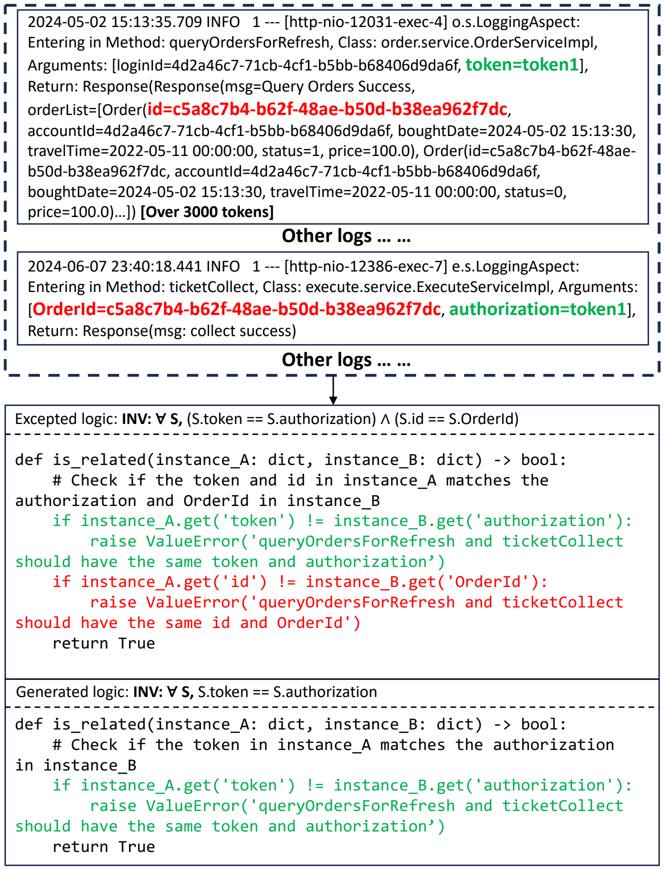
Scenarios	F1	Precision	Recall
Data consistency	0.918	0.909	0.926
Flow consistency	0.930	0.933	0.928
Common sense consistency	0.906	0.945	0.867

Table 6: The total training and testing on *TrainTicket* involved over 180,000 training logs and 70,000 testing logs.

Approach	Training Time (s)	Testing Time (s)
WebNorm	1050.1	21.6
LogRobust	407.1	0.4
ReplicaWatcher	N/A	1.7
NeuralLog	165.0	31.8
PLELog	148.9	9.9
DeepLog	16.2	2.8
LogAnomaly	32.0	7.6

Table 7: The overhead comparison on *TrainTicket* based on over 180,000 logs.

Scenarios	Num. Constraints	Training Time	Cost
Data consistency	31	648.81s	$12.91
Flow consistency	10	108.14s	$2.87
Common Sense consistency	24	293.15s	$6.78



```
2024-05-02 15:13:35.709 INFO   1 --- [http-nio-12031-exec-4] o.s.LoggingAspect:
Entering in Method: queryOrdersForRefresh, Class: order.service.OrderServiceImpl,
Arguments: [loginId=4d2a46c7-71cb-4cf1-b5bb-b68406d9da6f, token=token1],
Return: Response(Response(msg=Query Orders Success,
orderList=[Order(id=c5a8c7b4-b62f-48ae-b50d-b38ea962f7dc,
accountId=4d2a46c7-71cb-4cf1-b5bb-b68406d9da6f, boughtDate=2024-05-02 15:13:30,
travelTime=2022-05-11 00:00:00, status=1, price=100.0), Order(id=c5a8c7b4-b62f-48ae-
b50d-b38ea962f7dc, accountId=4d2a46c7-71cb-4cf1-b5bb-b68406d9da6f,
boughtDate=2024-05-02 15:13:30, travelTime=2022-05-11 00:00:00, status=0,
price=100.0)…]) [Over 3000 tokens]
```

Other logs … …

```
2024-06-07 23:40:18.441 INFO   1 --- [http-nio-12386-exec-7] e.s.LoggingAspect:
Entering in Method: ticketCollect, Class: execute.service.ExecuteServiceImpl, Arguments:
[OrderId=c5a8c7b4-b62f-48ae-b50d-b38ea962f7dc, authorization=token1],
Return: Response(msg: collect success)
```

Other logs … …

Excepted logic: **INV: ∀ S,** (S.token == S.authorization) ∧ (S.id == S.OrderId)

```
def is_related(instance_A: dict, instance_B: dict) -> bool:
    # Check if the token and id in instance_A matches the
authorization and OrderId in instance_B
    if instance_A.get('token') != instance_B.get('authorization'):
        raise ValueError('queryOrdersForRefresh and ticketCollect
should have the same token and authorization’)
    if instance_A.get('id') != instance_B.get('OrderId'):
        raise ValueError('queryOrdersForRefresh and ticketCollect
should have the same id and OrderId')
    return True
```

Generated logic: **INV: ∀ S,** S.token == S.authorization

```
def is_related(instance_A: dict, instance_B: dict) -> bool:
    # Check if the token in instance_A matches the authorization
in instance_B
    if instance_A.get('token') != instance_B.get('authorization'):
        raise ValueError('queryOrdersForRefresh and ticketCollect
should have the same token and authorization’)
    return True
```

Figure 10: The false negative example: The generated code fails to monitor the consistency between queryOrdersForRefresh's 'id' and ticketcollect's 'orderId' due to the LLM's long-context limitation, which hinders capturing all relevant constraints in complex logs.

long context, in this sense, the program analysis can well capture the relevant events even if there are a number of irrelevant events happen in between.

Table 5 breaks down the detailed comparison in different consistency-based invariants. We observe that WebNorm achieves the best performance on detecting flow consistency. In addition, the performance of WebNorm is still acceptable for generating the common-sense consistency invariants.

We further investigate when WebNorm failed to report anomalies. When fed complex logs, the LLM fails to recognize critical invariants, resulting in false negatives. Figure 10 presents a false negative example during data constraints generation, with constraints correctly captured by the LLM in green, and those missed in red. The problem arises when Event A's log exceeds 3000 tokens, and this extensive log is fed into the LLM. The excessive length and complexity of the logs impede the LLM's ability to extract all relevant constraints. Specifically, the constraint requiring Event A's 'Id' to match Event B's 'orderId' is overlooked, resulting in a false negative during detection.

4.3 RQ2: Cost of WebNorm

In this experiment, the total cost of using third-party LLM API services to learn all constraints in *TrainTicket* is $13.59 for parsing 183K logs (see Table 2). Note that, the cost is incurred only during the learning phase. Table 6 and Table 7 further break down the details of runtime overhead, computationally and financially. We can see that, in comparison to the other consistency invariants, the data consistency invariants makes more runtime overhead. In general, it appears more often in the normal scenarios, which makes us to feed more logs to LLM, incurring more failures (and iterations) of the generated test cases.

Furthermore, we observe in Table 6 that the testing time is acceptable across these extremely large logs. Given that we only need to train the LLM-agents only once that can be deployed, the training time is acceptable in real-world settings, although it seems to take a longer time in our learning approach.

4.4 RQ3: The Explainablity of WebNorm

WebNorm is expected to perform more reasonable traces to the developers. We show the performance of WebNorm's explainability in Table 8. We observe that WebNorm can overall locate the specific abnormal events, providing a intuitive feedback to the developers. For example, we can locate whether an event is triggered by the admin, or locate the abnormal events of *paydifference* and *rebook* and its suspicious relations, allowing the developers to directly look into the relevant information about the tamper attacks, enabling to make a quick response to these attacks. We observe that the false explanation largely lies in a spurious correlation in the generated Python code, indicating that passing all test cases does not necessarily reveal the true root cause.In Figure 11, we present a failed explanation case. Specifically, in the code, the condition **changeTime** > **travelTime** results in **status**=**0**, which prevents the order from being updated. However, **status**=**0** could also be triggered by other factors, such as attempting to modify an unpaid order. Thus, solely monitoring **status**=**0** creates a misleading conclusion, as it does not accurately capture the underlying conditions governing the application's behavior. The normal logs in the figure illustrate the expected system behavior: when **changeTime** > **travelTime**, the order cannot be updated, and the system returns to the order page. However, the generated constraint only focuses on monitoring **status**=**0**, which fails to guide developers in identifying the precise invariants.

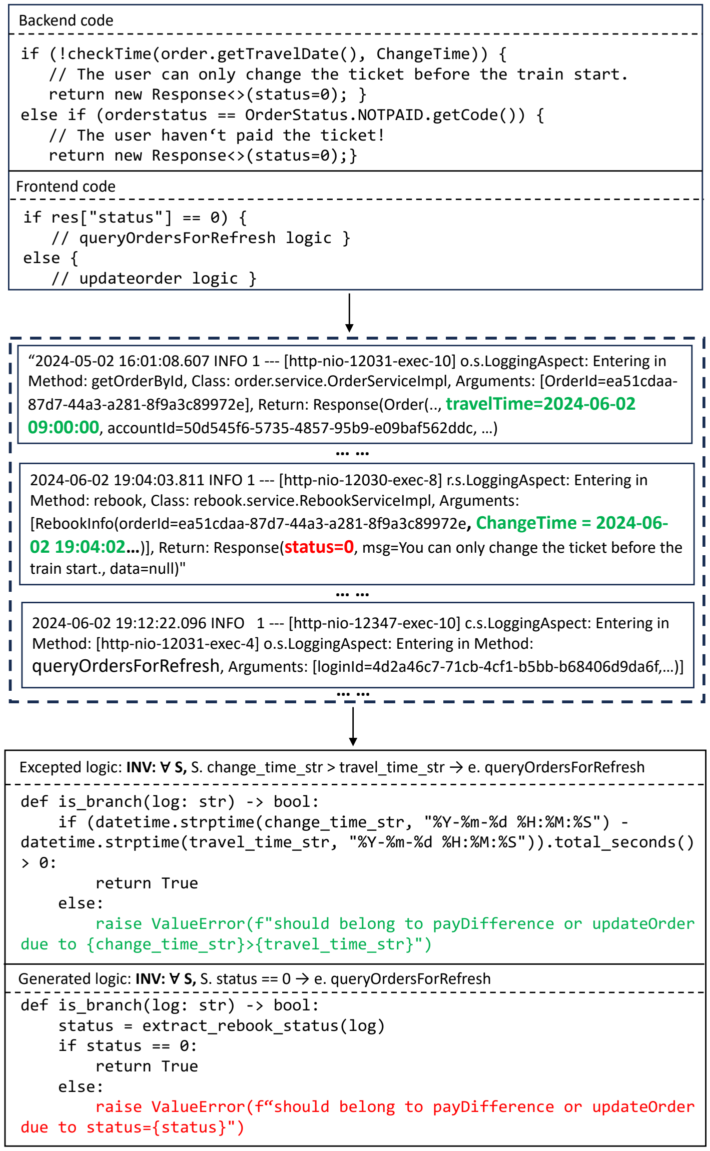
4.5 RQ4: The Seed Coverage of WebNorm

Table 10 generally shows how the change of normal seeds can affect the performance of WebNorm. While the precision keeps intact when the normal seeds are reduced to 10%, 30%, 50%, 70%, and 90% respectively. The recall is largely affected. Generally, those normal seeds serve as the "training dataset" for generating the WebNorm constraints and invariants. We suggest that the practitioners to prepare more representative seeds to apply WebNorm.

5 RELATED WORKS

In this section, we illustrate the relevant works about the detection of tamper attacks on web applications. Existing detection methods generally rely on sequence-based manners [10, 17, 37, 51, 54], which mainly assume the linear and sequential execution of events. However, these approaches are not well-suited for detecting tamper attacks in web applications. This is because web applications typically exhibit varying data and multiple operational flows, which lack sequential events that can be specifically monitored. Here, we mainly summarize two categories of tamper attacks and their corresponding detection solutions.

Log-based Anomaly Detection. Many literature focus on dealing with inferring models from systems' execution logs [1, 4, 33, 42, 45, 47]. Wang et al. [49] use the sets of temporal invariants to distingwish the difference between the logs. Goldstein et al. [15] compare the path of 2 logs then visulized the differences. 2kdiff [2] was the first to incorporate log modeling into log analysis. Nonetheless, the models developed by these works are coarse-grained, which results in a detection precision that is inferior to that of WebNorm. Besides, Some literatures on program analysis [5, 12, 26] incorporate invariants into their models; in the context of microservices. But in microservices, the larger amount and complexity of logs and invariants make it hard for these methods to track and analyze effectively. On the DevOps part, DeepTraLog [53] utilizes a Graph Gated Neural Networks (GGNNs)-based deep SVDD [43] model for identifying anomalies in both traces and corresponding logs. SCWarn [55] utilizes multimodal learning from diverse, heterogeneous data sources to detect problematic software changes. Khairi [21] proposed a training-less approach to resist run-time replica faults.



Backend code

```
if (!checkTime(order.getTravelDate(), ChangeTime)) {
   // The user can only change the ticket before the train start.
   return new Response<>(status=0); }
else if (orderstatus == OrderStatus.NOTPAID.getCode()) {
   // The user haven‘t paid the ticket!
   return new Response<>(status=0);}
```

Frontend code

```
if res["status"] == 0) {
   // queryOrdersForRefresh logic }
else {
   // updateorder logic }
```

```
“2024-05-02 16:01:08.607 INFO 1 --- [http-nio-12031-exec-10] o.s.LoggingAspect: Entering in Method: getOrderById, Class: order.service.OrderServiceImpl, Arguments: [OrderId=ea51cdaa-87d7-44a3-a281-8f9a3c89972e], Return: Response(Order(.., travelTime=2024-06-02 09:00:00, accountId=50d545f6-5735-4857-95b9-e09baf562ddc, …)
… …
2024-06-02 19:04:03.811 INFO 1 --- [http-nio-12030-exec-8] r.s.LoggingAspect: Entering in Method: rebook, Class: rebook.service.RebookServiceImpl, Arguments: [RebookInfo(orderId=ea51cdaa-87d7-44a3-a281-8f9a3c89972e, ChangeTime = 2024-06-02 19:04:02…)], Return: Response(status=0, msg=You can only change the ticket before the train start., data=null)"
… …
2024-06-02 19:12:22.096 INFO   1 --- [http-nio-12347-exec-10] c.s.LoggingAspect: Entering in Method: [http-nio-12031-exec-4] o.s.LoggingAspect: Entering in Method: queryOrdersForRefresh, Arguments: [loginId=4d2a46c7-71cb-4cf1-b5bb-b68406d9da6f,…)]
… …
```

Excepted logic: **INV: ∀ S,** S. change_time_str > travel_time_str → e. queryOrdersForRefresh

```
def is_branch(log: str) -> bool:
    if (datetime.strptime(change_time_str, "%Y-%m-%d %H:%M:%S") -
datetime.strptime(travel_time_str, "%Y-%m-%d %H:%M:%S")).total_seconds()
> 0:
        return True
    else:
        raise ValueError(f"should belong to payDifference or updateOrder
due to {change_time_str}>{travel_time_str}")
```

Generated logic: **INV: ∀ S,** S. status == 0 → e. queryOrdersForRefresh

```
def is_branch(log: str) -> bool:
    status = extract_rebook_status(log)
    if status == 0:
        return True
    else:
        raise ValueError(f“should belong to payDifference or updateOrder
due to status={status}")
```

Figure 11: The false explanation example: travelTime is the key invariant for determining the next flow, not status. Status=0 provides a spurious explanation, whereas the correct constraint is travelTime must be less than changeTime..

Table 8: The explainable examples for three constraints.

Constraints	Event1	Event2	Relation	Explanation Description
Flow Constraint	Rebook.service. paydifference	Rebook.service. rebook	Event2 riggers Event1	Paydifference should be triggered if differenceMoney returned by rebook is positive.
	User.service. getAllUsers	Auth.service. getToken	Event2 triggers Event1	GetAllUsers should be triggered if the role returned by getToken equals admin.
Data Constraint	Order.service. queryOrdersForRefresh	Inside_payment.service. pay	Event2 transfer price to Event1	QueryOrdersForRefresh and pay should have the same price
	contacts.service. findContactsByAccountId	preserve.service. preserve	Event2 transfer ContactId to Event1	FindContactsByAccountId and preserve should have the same contactId
Common Sense Constraint	Consign.service. updateConsignRecord	N.A	N.A	UpdateConsignRecord's weight must be a non-negative number

Table 9: *TrainTicket*'s explanation precision.

	Explanation Precision
Data consistency	0.933
Flow consistency	0.875
Common Sense consistency	0.961

Table 10: *TrainTicket*'s seeds coverage impact on the result.

Seed Coverage	10%	30%	50%	70%	90%
Recall	0.367	0.559	0.752	0.838	0.902

Log Instrumentation. Current detection models are generally based on log instrumentation. Yuan et al.[52] proposed the software logging practices in large open-source software projects. Then, based on an observation of seven open-source systems, Li et al. [30] present a deep learning framework that automatically suggests logging locations in source code. Liu et al. [32] propose an approach to recommend logging variables for developers during software development by learning from existing logging statements. Among them, LANCE [34] is the most recent and valuable research, it utilizes a Text-To-Text-Transfer-Transformer (T5) model [35] trained on several Java projects to assist developers in instrumenting logs. However, it has not been trained to run against our custom instrumented logs, so it cannot fully replace our existing instrumentation rules.

Tamper Attack Detection. Data tamper attacks refer to that an malicious web user modifies the client-side data, for example, the database in the server-end neglect the price validation. To detect such attacks, studies [44] always focus on detecting data tamper attacks on E-Commerce Applications. Parameter tamper attacks refer to that an malicious user manipulates the returned parameters, responses from a client-side, for example, the server-end only considers the positive or negative value of several returned parameter, and ignores specific returned value like the pieces of the clothing purchased. To detect such attacks, Bisht [6] proposed NoTamper detection framework based on client-side javascript code analysis techniques specialized to form validation code. Later, Bisht [7] also optimized the NoTamper with the white-box manner. Khodayari [22] also claimed the current Content Security Policy (CSP) and Cross-Origin Opener Policy (COOP) defense strategies are not sufficient to detect request hijacking attacks. Moreover, JSFlowTamper [25] and BFTDETECTOR [23] employ DOM monitoring for flow tamper detection. However, these methods fail when there are no significant DOM changes before and after the tamper attacks. In contrast, our approach leverages instrumented logs to provide comprehensive monitoring of web applications.

Many web applications suffer from the data and parameter tamper attacks simultaneously, which means that, only detecting one kind of tamper attacks can still lead to serious finial losses. However, few works propose a detection method capable of detecting both data and parameter tamper attacks. We, in this paper, for the first time propose a comprehensive detection framework to systemically detect multiple tamper attacks in the run-time. By addressing both data and parameter tamper attacks, organizations can significantly reduce the risk of security breaches and financial losses in their web applications architectures.

6 CONCLUSION AND FUTURE WORK

We introduce the WebNorm framework, designed to enhance existing microservice anomaly detection algorithms and generate explanatory models for various anomalies. We demonstrate that WebNorm is able to detect a range of network attacks and industrial faults more effectively than other methods on the train-ticket platform. Additionally, WebNorm allows for customizing rules for log instrumentation within microservices. Our extensive experiments indicate that WebNorm can improve existing microservice detection frameworks, with rule-based logs effectively monitoring internal microservice activities.

In future work, we plan to train a large language model tailored explicitly for microservices to reduce false positives. Furthermore, we aim to fully automate the generation of normal seeds, thereby enabling the complete automation of the WebNorm system.

ACKNOWLEDGMENT

This research / project is supported by the National Research Foundation, Singapore, and the Cyber Security Agency under its National Cybersecurity R&D Programme (NCRP25-P04-TAICeN). 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 and Cyber Security Agency of 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 *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1287624.1287630

[2] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2018. Using finite-state models for log differencing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 49–59.

[3] Spring Aop. 2024. Spring AOP. https://docs.spring.io/spring-framework/reference/core/aop.html

[4] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 267–277.

[5] Ivan Beschastnikh, Perry Liu, Albert Xing, Patty Wang, Yuriy Brun, and Michael D Ernst. 2020. Visualizing distributed system executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 2 (2020), 1–38.

[6] Prithvi Bisht, Timothy L. Hinrichs, Nazari Skrupsky, Radoslaw Bobrowicz, and V. N. Venkatakrishnan. 2010. NoTamper: automatic blackbox detection of parameter tampering opportunities in web applications. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov (Eds.). ACM, 607–618. https://doi.org/10.1145/1866307.1866375

[7] Prithvi Bisht, Timothy L. Hinrichs, Nazari Skrupsky, and V. N. Venkatakrishnan. 2011. WAPTEC: whitebox analysis of web applications for parameter tampering exploit construction. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, Yan Chen, George Danezis, and Vitaly Shmatikov (Eds.). ACM, 575–586. https://doi.org/10.1145/2046707.2046774

[8] Zhuangbin Chen, Jinyang Liu, Wenwei Gu, Yuxin Su, and Michael R. Lyu. 2021. Experience Report: Deep Learning-based System Log Analysis for Anomaly Detection. *CoRR* abs/2107.05908 (2021). arXiv:2107.05908 https://arxiv.org/abs/2107.05908

[9] CISCO. 2024. Splunk. https://www.splunk.com/

[10] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. 2017. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1285–1298.

[11] Elastic. 2024. Elasticsearch. https://www.elastic.co/elasticsearch

[12] Seyedeh Sepideh Emam and James Miller. 2018. Inferring extended probabilistic finite-state automaton models from software executions. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 27, 1 (2018), 1–39.

[13] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. *Acm sigplan notices* 47, 4 (2012), 37–48.

[14] Adam Gluck. 2020. Introducing domain-oriented microservice architecture. *Uber Engineering Blog* (2020).

[15] Maayan Goldstein, Danny Raz, and Itai Segall. 2017. Experience report: Log-based behavioral differencing. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 282–293.

[16] Jun Huang, Yang Yang, Hang Yu, Jianguo Li, and Xiao Zheng. 2023. Twin Graph-based Anomaly Detection via Attentive Multi-Modal Learning for Microservice System. arXiv:2310.04701 [cs.LG]

[17] 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

[18] IBM. 2024. IBM Security QRadar SIEM. https://www.ibm.com/products/qradar-siem

[19] JDBC. 2024. Spring JDBC. https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/

[20] Spring Jpa. 2024. spring-data-jpa. https://spring.io/projects/spring-data-jpa

[21] Asbat El Khairi, Marco Caselli, Andreas Peter, and Andrea Continella. 2024. REPLICAWATCHER: Training-less Anomaly Detection in Containerized Microservices. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 23-26, 2024*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/replicawatcher-training-less-anomaly-detection-in-containerized-microservices/

[22] Soheil Khodayari, Thomas Barber, and Giancarlo Pellegrino. 2024. The Great Request Robbery: An Empirical Study of Client-side Request Hijacking Vulnerabilities on the Web. In *Proceedings of IEEE Symposium on Security and Privacy (SP 2024), 21-23 May 2024, San Francisco, California, USA*.

[23] I Luk Kim, Weihang Wang, Yonghwi Kwon, and Xiangyu Zhang. 2023. BFTDETECTOR: Automatic Detection of Business Flow Tampering for Digital Content Service. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*.

[24] I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. 2020. Finding client-side business flow tampering vulnerabilities. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 222–233. https://doi.org/10.1145/3377811.3380355

[25] I Luk Kim, Yunhui Zheng, Hogun Park, Weihang Wang, Wei You, Yousra Aafer, and Xiangyu Zhang. 2020. Finding client-side business flow tampering vulnerabilities. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 222–233. https://doi.org/10.1145/3377811.3380355

[26] Ivo Krka, Yuriy Brun, and Nenad Medvidovic. 2014. Automatic Mining of Specifications from Invocation Traces and Method Invariants *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 178–189. https://doi.org/10.1145/2635868.2635890

[27] Van-Hoang Le and Hongyu Zhang. 2021. Log-based Anomaly Detection Without Log Parsing. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 492–504. https://doi.org/10.1109/ASE51524.2021.9678773

[28] Van-Hoang Le and Hongyu Zhang. 2021. Log-based anomaly detection without log parsing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 492–504.

[29] Van-Hoang Le and Hongyu Zhang. 2022. Log-based anomaly detection with deep learning: how far are we? *(ICSE '22)*.

[30] Zhenhao Li, Tse-Hsun Chen, and Weiyi Shang. 2020. Where shall we log? studying and suggesting logging locations in code blocks. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 361–372.

[31] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, and Dan Pei. 2020. Unsupervised Detection of Microservice Trace Anomalies through Service-Level Deep Bayesian Networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. 48–58. https://doi.org/10.1109/ISSRE5003.2020.00014

[32] Zhongxin Liu, Xin Xia, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2019. Which variables should i log? *IEEE Transactions on Software Engineering* 47, 9 (2019), 2012–2031.

[33] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic Generation of Software Behavioral Models. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 501–510. https://doi.org/10.1145/1368088.1368157

[34] Antonio Mastropaolo, Luca Pascarella, and Gabriele Bavota. 2022. Using Deep Learning to Generate Complete Log Statements. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2279–2290. https://doi.org/10.1145/3510003.3511561

[35] Antonio Mastropaolo, Simone Scalabrino, Nathan Cooper, David Nader Palacio, Denys Poshyvanyk, Rocco Oliveto, and Gabriele Bavota. 2021. Studying the usage of text-to-text transfer transformer to support code-related tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 336–347.

[36] Tony Mauro. 2015. Adopting Microservices at Netflix: Lessons for Architectural Design. *Nginx Blog* (2015).

[37] Weibin Meng, Ying Liu, Yichen Zhu, Shenglin Zhang, Dan Pei, Yuqing Liu, Yihao Chen, Ruizhi Zhang, Shimin Tao, Pei Sun, and Rong Zhou. 2019. Loganomaly: Unsupervised Detection of Sequential and Quantitative Anomalies in Unstructured Logs. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (Macao, China) *(IJCAI'19)*. AAAI Press, 4739–4745.

[38] Mr.MG. 2023. Payment Bypass Using Parameter Manipulation (POCs). https://medium.com/@mrmaulik191/payment-bypass-using-parameter-manipulation-ee1bc9977f0c

[39] N.A. 2022. *NiceFish*. https://gitee.com/mumu-osc/NiceFish-React

[40] N.A. 2024. *WebNorm*. https://sites.google.com/view/webnorm/home

[41] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.".

[42] Michael Pradel and Thomas R. Gross. 2009. Automatic Generation of Object Usage Specifications from Large Method Traces. In *2009 IEEE/ACM International Conference on Automated Software Engineering*. 371–382. https://doi.org/10.1109/ASE.2009.60

[43] Lukas Ruff, Robert Vandermeulen, Nico Goernitz, Lucas Deecke, Shoaib Ahmed Siddiqui, Alexander Binder, Emmanuel Müller, and Marius Kloft. 2018. Deep one-class classification. In *International conference on machine learning*. PMLR, 4393–4402.

[44] Fangqi Sun, Liang Xu, and Zhendong Su. 2014. Detecting Logic Vulnerabilities in E-commerce Applications. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*.

The Internet Society. https://www.ndss-symposium.org/ndss2014/detecting-logic-vulnerabilities-e-commerce-applications

[45] Deyu Tian. 2021. Detecting user-perceived failure in mobile applications via mining user traces. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 123–125.

[46] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*. IEEE, 583–590.

[47] Neil Walkinshaw and Kirill Bogdanov. 2008. Inferring Finite-State Models with Temporal Constraints. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. 248–257. https://doi.org/10.1109/ASE.2008.35

[48] Jin Wang, Changqing Zhao, Shiming He, Yu Gu, Osama Alfarraj, and Ahed Abugabah. 2022. LogUAD: Log Unsupervised Anomaly Detection Based on Word2Vec. *Comput. Syst. Sci. Eng.* 41, 3 (2022), 1207–1222. https://doi.org/10.32604/CSSE.2022.022365

[49] Qianqian Wang, Yuriy Brun, and Alessandro Orso. 2017. Behavioral Execution Comparison: Are Tests Representative of Field Behavior?. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 321–332. https://doi.org/10.1109/ICST.2017.36

[50] JPA with Hibernate. 2024. Spring JPA with Hibernate. https://www.baeldung.com/learn-jpa-hibernate

[51] 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 230–231. https://doi.org/10.1109/ICSE-Companion52605.2021.00106

[52] Ding Yuan, Soyeon Park, and Yuanyuan Zhou. 2012. Characterizing logging practices in open-source software. In *2012 34th International Conference on Software Engineering (ICSE)*. 102–112. https://doi.org/10.1109/ICSE.2012.6227202

[53] 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 *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 623–634. https://doi.org/10.1145/3510003.3510180

[54] 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 *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 807–817.

[55] 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 *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 527–539.

[56] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. 2018. Benchmarking microservice systems for software engineering research. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 323–324.

[57] Daniel Zimmermann and Anne Koziolek. 2023. GUI-Based Software Testing: An Automated Approach Using GPT-4 and Selenium WebDriver. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023 - Workshops, Luxembourg, September 11-15, 2023*. IEEE, 171–174. https://doi.org/10.1109/ASEW60602.2023.00028