# *Caught-in-Translation (CiT)*: Detecting Cross-level Inconsistency Attacks in Network Functions Virtualization (NFV)

Sudershan Lakshmanan, Mengyuan Zhang, Suryadipta Majumdar, Yosr Jarraya, Makan Pourzandi, Lingyu Wang,

**Abstract**—As one of the main technology pillars of 5G networks, Network Functions Virtualization (NFV) enables agile and cost-effective deployment of network services. However, the multi-level, multi-actor design of NFV may also allow for inconsistency between the different abstraction levels to be mistakenly or intentionally introduced, as shown in recent studies. Serious security issues, such as man-in-the-middle, network sniffing, and DoS, may arise at one abstraction level without being noticed by the victims at another level. Most existing solutions are either limited to one abstraction level of NFV or reliant on direct access to lower-level data which could become inaccessible when managed by different providers.

In this paper, by drawing an analogy between cross-level NFV event sequences and natural languages, we propose a Neural Machine Translation-based approach, namely, *Caught-in-Translation (CiT)*, to detect cross-level inconsistency attacks in NFV at runtime. Specifically, we first extract event sequences from different abstraction levels of an NFV stack. We then leverage Long Short-Term Memory (LSTM) to translate the event sequences from one level to another. Finally, we apply both a similarity metric and a Siamese neural network to compare the *translated* event sequences with the *original* ones to detect attacks. We integrate *CiT* into OpenStack/Tacker, a popular open-source NFV implementation, and evaluate its performance using both real and synthetic data. Experimental results show the benefit of leveraging NMT as *CiT* achieves AUC $\geq$ 96.03%, which significantly outperforms traditional SVM-based anomaly detection. We also evaluate *CiT* in terms of its efficiency, scalability, and robustness for detecting inconsistency attacks in NFV platforms.

**Index Terms**—Inconsistency detection, Neural Machine Translation, NFV security

## 1 Introduction

Network functions virtualization (NFV) has emerged as one of the main technology pillars of 5G networks [1], [2]; for instance, in a 2021 survey, over 50% of network service providers have adopted NFV [3], and the NFV market size is projected to grow from $12.9B in 2019 to $36.3B by 2024 [4]. The main benefit of NFV comes from its power in decoupling the network functions, such as firewall or intrusion detection, from dedicated and proprietary hardware appliances. By virtualizing those network functions on top of standard hardware infrastructures, NFV makes it possible for providers to deploy dynamic, agile, scalable, and cost-efficient network services.

When a user requests the deployment of a new network service (e.g., creating a virtual network function (VNF)), a cascade of events occurs across multiple levels of the NFV stack. As each level is operated by autonomous managerial components (i.e., the so-called *multi-actor* design of NFV [5]), potential inconsistencies may arise between the specification of a network service and its actual implementation at lower levels. Worse, an adversary could tamper with a managerial component at an implementation level and the malicious changes could remain unnoticed by the user, thus enabling stealthy attacks. These attacks may cause severe security issues, such as unauthorized modifications of Service Function Chains (SFCs), network eavesdropping, and denial-of-service (DoS) attacks, breaking confidentiality, as enabled by real-world vulnerabilities (e.g., [6]–[9])[1] and as illustrated in recent studies (e.g., [10]–[14]).

Most of the existing works (e.g., [15]–[19] ) focus on verifying the state (e.g., system configurations) of the NFV system to discover the presence of inconsistencies. Such an after-the-fact verification is retroactive in nature and cannot detect attacks as they happen, which fails to prevent irreversible damages, e.g., information leakage. Moreover, some approaches (e.g., [16], [20]–[22]) rely on the access to lower-level configuration data (e.g., network flow rules, flow classifiers), which may become inaccessible when multiple providers are involved [23]. Finally, to the best of our knowledge, none of the existing work considers the implication of all abstraction levels of the NFV stack, and most only focus on part of the NFV stack (e.g., the physical [24] or virtual infrastructures [25], or SFC [15]–[18]).

- *S. Lakshmanan, S. Majumdar and L. Wang are with The Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada.*
  *E-mail: s_akshma, majumdar and wang@encs.concordia.ca*
- *M. Zhang is with The Department of Computing, The Hong Kong Polytechnic University, Hong Kong.*
  *E-mail: mengyuan.zhang@polyu.edu.hk*
- *Y. Jarraya and M. Pourzandi are with The Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada.*
  *E-mail: yosr.jarraya and makan.pourzandi@ericsson.com*

1. CVE-2015-3456, CVE-2015-7835, CVE-2018-10853, CVE-2023-2088, CVE-2022-47951, CVE-2022-47950, CVE-2021-40797

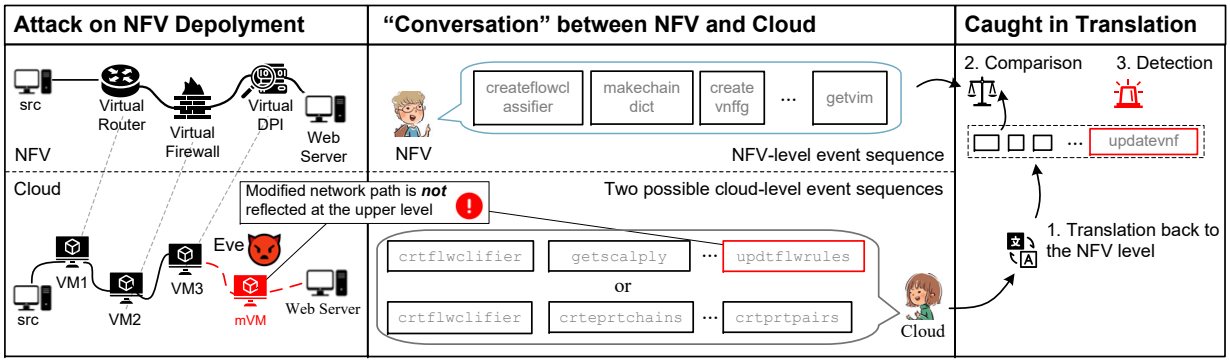



Fig. 1: The motivating example showing an example inconsistency attack (left), the event sequences in NFV and cloud levels highlighted with the natural language analogy (middle), and the steps involved in inconsistency attack detection by *CiT* (right)

To detect inconsistencies at runtime, we could monitor and compare the streams of events as they happen at all levels of an NFV stack (similar ideas exist for cloud platforms [26], [27]). A straightforward approach would be to establish a cross-level mapping of NFV events corresponding to the same originating user request, and report mismatches. However, a main challenge with event-based cross-level inconsistency detection is that the same event sequence at one level of an NFV stack usually corresponds to multiple event sequences at a lower level, depending on the nature of requests and the specification of the network services (e.g., different scaling up/down policies, ports attached to VNFs, or storage volumes). It would be impractical to enumerate all possible combinations to create a rule-based solution (see details in Section 5).

A better approach would be to rely on traditional machine learning-based detection systems. However, those systems usually require carefully chosen features, dependent on the characteristics of each level in an NFV stack. As evidenced by our experimental results, such systems will be outperformed by our solution.

Instead, we draw an analogy between comparing event sequences at different NFV levels and translating sentences between different natural languages. As a result, we can leverage existing Neural Machine Translation (NMT) techniques designed for the latter to detect inconsistency attacks in NFV. Specifically, a user request would automatically trigger a sequence of events at each level of the NFV stack. Those event sequences from different levels are syntactically different, but semantically all correspond to the same user request. This is similar to a situation where several characters who speak different natural languages can converse on the same topic through a translator.

Based on such an analogy, we could "translate" event sequences from one level to another, and subsequently detect any inconsistencies by comparing the translated sequence to the actual one. Translation-based solutions have already shown promising results in other domains (e.g., binary code similarity [28], network traffic anomaly [29], Android malware detection [30]). Nonetheless, to the best of our knowledge, this is the first effort to apply such an idea in the context of NFV.

**Motivating Example.** We present a motivating example (see Figure 1) to further highlight existing challenges in cross-level inconsistency detection for NFV and motivate towards our solution. For simplicity, we refer to the VNF and NFV infrastructure as NFV level and cloud level, respectively. The NFV-related concepts are further explained in Section 2.1.

*The Problem:* The left side of Figure 1 shows the NFV stack of a tenant, which consists of an NFV level with three network functions (*Virtual Router*, *Virtual Firewall*, and *Virtual DPI*) and a cloud level with their corresponding virtual machines. Knowing that an adversary *Eve* could potentially inject a malicious virtual machine (*mVM*) into the tenant's network directly at the cloud level, without causing any noticeable changes to the NFV level[2], this tenant is concerned with the following question: *"Is my service function chain (SFC) properly deployed at the lower levels?"*

As shown in the middle part of Figure 1, a user-level request (e.g., creating a network function forwarding path) triggers a series of events at both the NFV and cloud levels during deployment. This is depicted as a conversation between the two characters (NFV and cloud), spoken in two different languages (i.e., NFV events and cloud events) but on the same topic (i.e., deploying the network service requested by the user). Our question is: *"Is the cloud talking in a manner that is consistent with the NFV?"* Notably, between the two possible cloud-level event sequences, the upper one includes an extra event *updateflowrules* (highlighted in red) that would cause an inconsistency, whereas the lower sequence is consistent.

*Our Ideas:* The right side of Figure 1 demonstrates the main ideas of our approach, Caught in Translation (*CiT*), for detecting such inconsistency attacks. *CiT* first *translates* the cloud-level event sequence back to the NFV level, and it then *compares* the translated event sequence to the original NFV-level event sequence for *detecting* any inconsistency. The two cloud-level sequences demonstrate the key challenge that the same NFV-level event sequence may correspond to multiple cloud-level event sequences, depending on the nature of requests and the specification of the network services. This means there does not exist a trivial mapping between the event sequences that would enable translation with simple rules.

2. For instance, by exploiting existing vulnerabilities CVE-2015-3456, CVE-2015-7835, or CVE-2018-10853 in a specific way [13].

Specifically, we propose a Neural Machine Translation (NMT)-based approach, *Caught in Translation (CiT)*, to detect cross-level inconsistency attacks in NFV. More specifically, we first extract events and event sequences from raw logs and generate their corresponding embeddings. Second, we devise an NMT-based technique (tested with Long Short-Term Memory (LSTM) [31], Gated Recurrent Unit (GRU) [32], and simple Recurrent Neural Network (RNN) [33]) to translate event sequences between different levels of NFV. Finally, we quantify the similarity between the translated and the original event sequences to detect any inconsistencies by applying both a Siamese neural network [34] and a traditional similarity metric [35]. We implement *CiT* and integrate it into OpenStack/Tacker [23], a popular choice for NFV deployment on cloud management platform [36]. We evaluate the accuracy and efficiency of *CiT* using both real and synthetic data. In summary, our main contributions are as follows.

- To the best of our knowledge, *CiT* is the first event-based approach for detecting inconsistency attacks in NFV. In contrast to most after-the-fact approaches, *CiT* can detect malicious events at runtime before such events cause any potentially irrecoverable damages, such as the leakage of sensitive information or DoS.
- *CiT* demonstrates the potential of a translation-based detection approach. First, *CiT* outperforms traditional SVM-based anomaly detection working at one abstraction level in terms of accuracy (AUC $\geq$ 96.03%) under the same system settings. Second, through comparisons with alternative implementations of *CiT*, we observe that translation can significantly improve the detection accuracy. Finally, the translation capability may have other applications in NFV, such as providing translated events at a level where the access to actual events is prohibited (e.g., operated by different providers).
- The practicality of *CiT* is demonstrated through its integration into OpenStack/Tacker. Its accuracy and efficiency are evaluated through extensive experiments using both real and synthetic NFV datasets. The feasibility of training *CiT* under a controlled environment and then applying it to a real production system is evaluated (AUC $\geq$ 83%) and *CiT* is applied to capture anomalous events triggered by real-world bugs and errors. Notably, the trained model demonstrates the ability to detect bugs and errors that are not present in the training dataset, suggesting its potential for identifying issues without prior knowledge.

The rest of the paper is organized as follows. Section 2 provides the preliminaries. Section 3 details the *CiT* methodology. Section 4 presents the experimental results. Section 5 provides more discussions. Section 6 reviews the related work, and Section 7 concludes the paper.

## 2 Preliminaries

This section provides some background on NFV and deep learning, explains the NFV-specific challenges of applying NMT techniques, and defines our threat model.

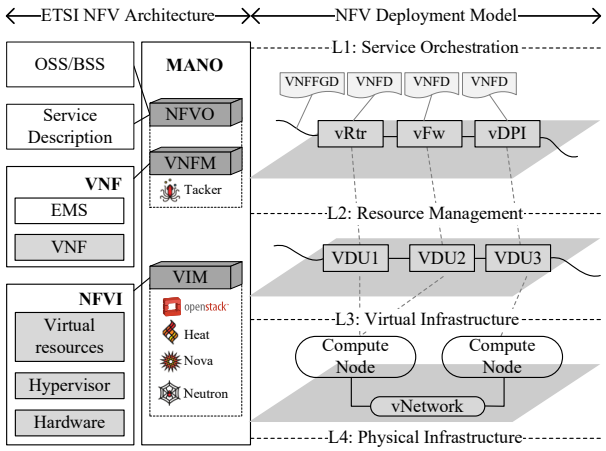



Fig. 2: The ETSI NFV Architecture (left) and the NFV deployment model (right)

### 2.1 NFV Background

**NFV Stack.** Network Functions Virtualization (NFV) is a concept designed to virtualize network functions, such as routers, firewalls, load balancers, etc. Figure 2 illustrates the ETSI NFV reference architecture [37] (left), and a typical NFV stack deployment (right). The ETSI architecture depicts two main abstraction levels, namely, the Virtual Network Function (VNF) level providing a high-level representation of network functions, and the NFV Infrastructure (NFVI) level representing the underlying cloud infrastructure. The managerial components at different levels (e.g., the Network Function Virtualization Orchestrator (NFVO), Virtual Network Function Manager (VNFM), and Virtualized Infrastructure Manager (VIM)) are together known as the NFV Management and Orchestration (MANO), and are responsible for managing and orchestrating the physical and virtual network resources.

**NFV Acronyms.** Table 1 lists all the main abbreviations we use in this paper.

| Acronym | Full Form |
|---|---|
| GRU | Gated Recurrent Unit |
| LSTM | Long Short-term Memory |
| NFP | Network Forwarding Path |
| NFV | Network Function Virtualization |
| NFVI | Network Function Virtualization Infrastructure |
| NFVO | Network Function Virtualization Orchestrator |
| NMT | Neural Machine Translation |
| NS | Network Service |
| OOV | Out-of-Vocabulary |
| OvS | Open vSwitch |
| RNN | Recurrent Neural Network |
| SDN | Software Defined Networking |
| SDN-C | SDN Controller |
| SFC | Service Function Chain |
| SVM | Support Vector Machine |
| TFIDF | Term Frequency-inverse Document Frequency |
| TOSCA | Topology and Orchestration Specification for Cloud Applications |
| VDU | Virtual Deployment Unit |
| VIM | Virtual Infrastructure Manager |
| VM | Virtual Machine |
| VNF | Virtual Network Function |
| VNFD | VNF Descriptor |
| VNFFG | VNF Forwarding Graph |
| VNFFGD | VNFFG Descriptor |
| VNFM | VNF Manager |

TABLE 1: Acronyms used in this paper

**NFV Attack Surface.** The attack surface of NFV can be broadly divided into three categories, VNFs, cloud infrastructure, and MANO, which are identified as the critical components of an NFV stack [38]. A lack of consistency between these components on how to orchestrate and manage the virtualized network functions can incur security challenges [11] by enabling stealthy attacks in which the adversary can conduct malicious activities without leaving any trace visible to the user. In particular, modern security attacks such as, software flaw attacks (e.g., unauthorized volume access [39]), orchestration attacks that exploit vulnerable virtual components (e.g., CVE-2022-47951 [40] and CVE-2022-47950 [41]), resource monopolization or DoS attacks that consume unauthorized resources (e.g., CVE-2021-40797 [42]), and security misconfiguration attacks (e.g., direct memory access attack [43]) form a major concern to NFV system security by going undetected.

**NFV Tenants and Users.** As defined by OpenStack [44], "Tenants" are an organization/project (e.g., a university), and "Users" are a part of organization/project (e.g., students/staff/researchers).

**NFV Events.** Events are function calls that get triggered upon execution of a user-level request.

**NFV Event Sequences.** The deployment of a network service starts with user-level requests (e.g., create network functions, chain network functions, etc.), which will eventually trigger a sequence of internal events (i.e., function calls) at each level of the NFV stack. *CiT* leverages these event sequences to validate the correctness of a network service implementation across all NFV levels.

### 2.2 Deep Learning Background

**LSTM.** LSTM is an artificial RNN that supports sequence-to-sequence learning (Seq2Seq) [45], a mechanism for training machine learning models to convert text sequences from one language (e.g., English) to another (e.g., French) by capturing the meaning of those sentences using automatically generated features (embedding). The event embeddings are numerical vector representations of events, efficiently capturing the semantic information which facilitates the event sequence translation. Particularly, the LSTM Encoder-Decoder model [46] has been shown to achieve good performance especially for long sequences [45]. Therefore, we adopt this Encoder-Decoder model for translation since the NFV event sequences are relatively long.

**Siamese Network Architecture.** The Siamese network [34] is a deep learning model that has been applied to evaluate similarities between two comparable inputs. Specifically, Siamese network employs two identical deep learning models that share the same weights of parameters; where each model takes encoded inputs and generates its semantic representation as outputs. For calculating the similarity score, Siamese network utilizes Manhattan distance [35], which is preferable to Euclidean distance to evaluate the distance of two high dimensional inputs [47] (which is the case in our context).

### 2.3 Challenges of applying NMT techniques in NFV

NMT-based detection has been used in other domains such as, binary code similarity [29], network traffic anomaly [30], and Android malware detection [31]. However, adopting this technique to NFV presents multiple domain specific challenges as listed below.

- **Cross-level Dependencies in NFV Stack.** A distinct feature of NFV is its multi-level nature, as shown in Figure 2. Therefore, to build a translation-based inconsistency detection model, it is necessary but challenging to capture the relationship between event sequences across the different levels of the NFV stack. Specifically, the multi-level design of the NFV stack poses challenges as higher-level event sequences often correspond to multiple sequences at lower levels. For example, the corresponding event sequence of the VM creation operation may or may not include additional events related to new ports, storage volumes, or auto-scaling policies based on the user specification. We have further explained this challenge in Section 1.
- **Out-of-Vocabulary Challenge.** Raw log records (gathered from NFV service logs or intercepted at runtime using middleware) typically contain many implementation-specific details (e.g., platform-specific APIs) and parameters (e.g., request/resource IDs). Some of these non-essential details, if fed directly into the training modules, could cause the Out-of-Vocabulary (OOV) challenge [48] leading to inaccurate results. For example, the event that converts TOSCA template to HOT template (*represent_odict*) may contain parameters such as, *vnf*, *heatclient*, *inst_req_info*, and *grant_info*. Each parameter may receive unpredictable values based on the state of the system. Therefore, the raw data must be carefully processed in order to avoid the OOV issue while preserving all the essential details such that events belonging to the same sequence can still be identified even if they arrive out of order. We address the OOV challenge in Section 3.2.
- **Robust NFV-specific Detection Model.** Developing a robust detection model that can be readily integrated into existing NFV systems in production is a challenge. The challenge lies in collecting high-quality data, labeling the data properly, training the model, and validating the trained model. More specifically, unlike most other applications where NMT models are trained once using large text corpora (e.g., Word-Net [51] and Wiki) and there exist pre-trained models which can be reused (e.g., GloVe [52] or fastText [53]), there do not exist similar text corpora or pre-trained models for NFV. To the best of our knowledge, we are the first to apply such techniques in the context of NFV, and therefore, one of the key challenges is to build our own domain-specific corpus related to NFV events and train our models from scratch. To ensure the correctness of the training, it is important to maintain a training set containing only normal event occurrences. Any abnormal event sequences could lead to inaccurate outcomes. Furthermore, during the implementation of the model, we should also validate the ability of the model to

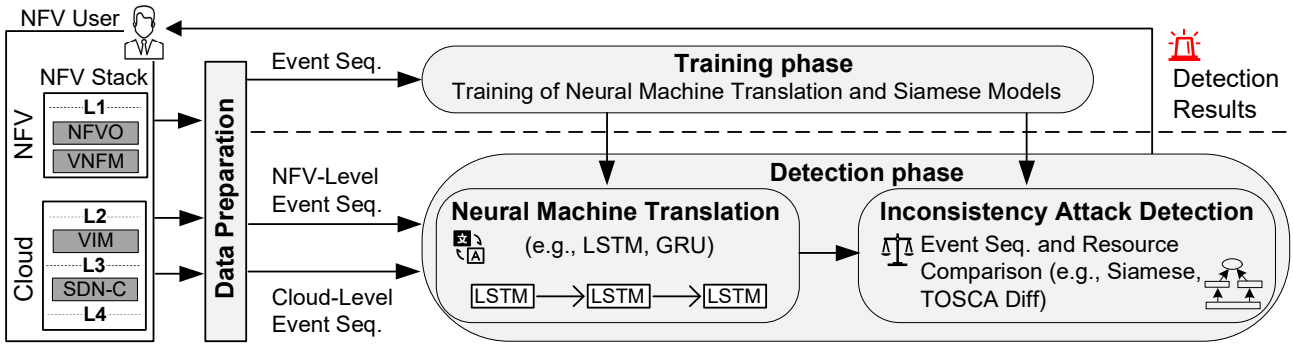



Fig. 3: *CiT* System Overview

accommodate unseen event sequences and avoid issues like overfitting. Section 3.4 explains the methodology of our detection model, and Section 4 illustrates the results obtained from this evaluation.

### 2.4 Threat Model

Our in-scope threats include insiders, such as malicious tenants or tenant administrators (who can only control the NFV-level management, but not the host OS) who may create inconsistencies in an NFV stack either with malicious intentions or by mistakes, as well as external attackers who exploit existing vulnerabilities in the NFV stack to launch inconsistency attacks. Those fall in the third category of the NFV attack surface, i.e., MANO. On the other hand, our solution relies on events captured at different levels of the NFV stack. Similarly to most event-based solutions [26], [27], we assume that the events as captured from the services are intact, and an adversary cannot compromise the integrity of events or their order. Therefore, out-of-scope threats include attacks that do not cause any violation of consistency in the NFV stack, attacks that are not reflected in the events (e.g., CPU pinning, SR-IOV, PCI Passthrough) or their order, and attacks with which adversaries can either escape the capture of events or tamper with the captured results. Finally, our work focuses on event detection, and is not designed to either attribute detected attacks to underlying vulnerabilities or prevent future attacks, although our solution may provide useful inputs to such solutions.

## 3 Methodology

This section first provides an overview of *CiT*, followed by the detailed methodologies of its major components.

### 3.1 Overview

Figure 3 depicts how *CiT* prepares the data and trains the neural network models (training phase), and how it applies the trained models to first translate event sequences from the cloud level to the NFV level, and then compare the event sequences to detect inconsistency attacks (detection phase).

**Training Phase.** This phase (top of Figure 3) consists of three main steps. First, *CiT* extracts events from both NFV and cloud levels, and constructs event sequences per level. Second, the neural machine translation (e.g., LSTM model [49]) and the inconsistency attack detection model (e.g., neural network similarity learning model [34]) are trained based on the prepared datasets. Finally, the trained models are updated through retraining when any substantial change is made to the NFV system (e.g., major service updates that may introduce new event types).

**Detection Phase.** In this phase (bottom of Figure 3) also contains three main steps. *CiT* first extracts the event sequences from both NFV and cloud levels. Then it applies the trained translation model to translate an event sequence from one level (e.g., cloud) to another (e.g., NFV). Finally, the translated and original sequences from the same level are compared in order to detect any inconsistency attacks. Note that translation is bi-directional between two levels, although we will only mention the translation from cloud to NFV for simplicity. Each step of these two phases will be further elaborated in the following subsections.

***CiT* Variations.** To support various use cases (see Section 5) *CiT* couples translation and detection in three different ways as follows.

- $CiT_{ts}$: This variation of *CiT* first translates an event sequence at cloud level into NFV level. Then, it compares the translated event sequences to the original event sequences using neural network similarity learning (e.g., Siamese [34]). Finally, the similarity score is compared to a pre-defined threshold value to determine whether an attack has been detected.
- $CiT_{tm}$: This variation first performs a similar translation step as $CiT_{ts}$, and then compares the event sequences using a traditional similarity metric (e.g., Manhattan distance [35], [47]) to detect any inconsistency.
- $CiT_{s}$: This variation skips the translation step, and directly applies neural network similarity learning (e.g., Siamese [34]) on the event sequences from both levels to detect any inconsistency.

**Summary.** *CiT* captures the relationship between event sequences through different deep learning models, i.e., the translation model and the Siamese model. A range of complex relationships such as, inclusion and hierarchy, can be captured through the translation step. Then, we use Siamese network architecture ($CiT_{ts}$ and $CiT_{s}$) to compare the two sequences ($CiT_{tm}$ only uses traditional similarity metric after the translation step). This step resolves equivalent parts of the event sequences by tokenizing and generating embeddings out of them. It helps to ignore unimportant reordering

of events or optimizations. Finally, the distance between the two sequence embeddings is calculated to detect any inconsistency.

### 3.2 Data Preparation

The data preparation step generates event sequences from raw logs for both translation and inconsistency detection. *CiT* first extracts relevant events from the data intercepted at runtime using middleware or available in service logs. Then, the events are aggregated into a sequence corresponding to the same user-level request based on their parameters, such as the *resource ID* and *timestamps*.

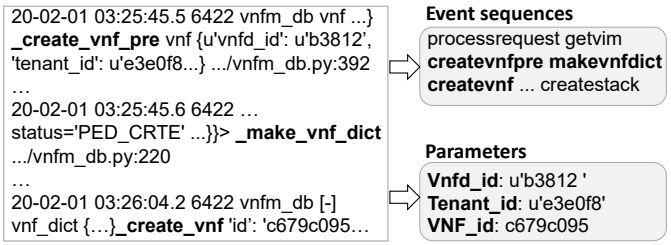



Fig. 4: An excerpt of Tacker log entries and the output of data preparation (event seq. and corresponding parameters)

**Challenge.** A main challenge in preparing NFV data for training is that the raw data typically contains many implementation specific details (e.g., platform-specific APIs) and parameters (e.g., request/resource IDs). Utilizing raw data with such non-essential details could create an infinite vocabulary leading to new events appearing during the detection phase. This is a well-known problem in NMT, namely, the Out-of-Vocabulary (OOV) challenge [48], which ultimately affects the accuracy of any application, e.g., inconsistency detection in our context. Therefore, the raw data must be carefully processed such that the OOV issue can be avoided (by removing non-essential details) while the events belonging to the same sequence can still be identified even if they arrive out of order or interleave (by preserving all the essential parameters).

To overcome this challenge, *CiT* strips out the implementation-specific details and parameters from the sequences while keeping essential information, such as the events' names. The output of our data preparation step is an NFV data corpus ready to be fed into the training modules along with the required parameters, e.g., the implemented VM's memory size, to enable fine-grained resource-level inconsistency detection.

***Example 1.*** *Figure 4 shows an excerpt of Tacker log entries (left) and the corresponding processed event sequences and extracted parameters (right). All the events that correspond to the same user-level request of creating a network function are identified from the raw data. Note that highlighted events correspond to different stages of implementation, e.g., initialization stage (_create_vnf_dict). The parameters are used to generate specifications for resource-level inconsistency detection.*

### 3.3 Neural Machine Translation

*CiT* translates event sequences between different levels of an NFV stack by leveraging neural machine translation (NMT), particularly, long short-term memory (LSTM) [31]. We adopt LSTM over a simple recurrent neural network (RNN) since the latter is not equipped to capture the contextual semantic dependencies to translate long NFV event sequences. In this section, we describe our translation approach with an example.

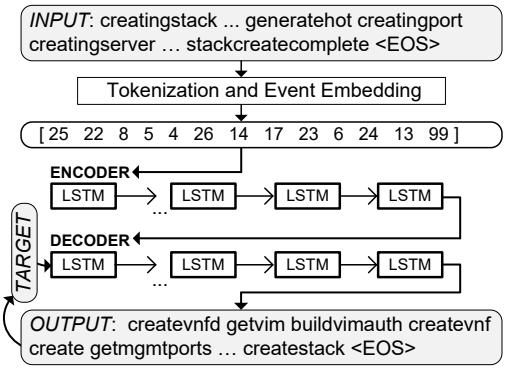



Fig. 5: An example of Seq2Seq translation using the LSTM Encoder-Decoder model for an event sequence

**Event Sequence Translation.** *CiT* performs event sequence translation in two main steps: training LSTM models with hyperparameters, and translating event sequences using the trained models. The hyperparameters (i.e., the parameters that play a major role in shaping and maximizing the performance of translation model [50]) used in LSTM for natural language translation or other domains may not necessarily apply to NFV data. Thus, we evaluate different combinations of the hyperparameters to improve the performance and accuracy of our translation model (see Section 4.5).

To train the LSTM model, *CiT* first constructs pairs of event sequences at two different levels that correspond to the same user-level request. Specifically, normal events at cloud level corresponding to a specific request (e.g., *creating a network function*) is provided as the input to LSTM-based encoder, with the corresponding event sequence at the NFV level as the reference for LSTM-based decoder. We train LSTM models from other levels in a similar manner. Finally, *CiT* applies the trained LSTM models to translate event sequences from one level to another level to facilitate inconsistency detection.

***Example 2.*** *Figure 5 shows an example of event sequences translation using a trained LSTM Encoder-Decoder model. The event sequence (*INPUT*) from cloud level is translated to an event sequence (*OUTPUT*) at NFV level. Both* INPUT *and* OUTPUT *correspond to the same NFV user-level request for creating a network function. More specifically, the event embeddings of* INPUT *are fed into the LSTM Encoder. Each LSTM* cell *in the Encoder accepts an event from* INPUT *and produces a corresponding output to the next cell to retain the contextual meaning of the sentence. The Decoder, which is initialized with an arbitrary start event,* TARGET*, predicts the next event (e.g.,* creatingstack*) using the output from the Encoder. The translated event is then appended to both* OUTPUT *and* TARGET*. This translation process is repeated until the Decoder generates the end-event ($\langle EOS \rangle$).*

### 3.4 Inconsistency Detection Model

After translating event sequences from cloud level to NFV level, a straightforward way to detect inconsistency would be to apply word-to-word similarity comparison. However, as demonstrated in our motivating example (Section 1), one user request may correspond to many possible event sequences which may look completely different but are in fact equivalent. A word-to-word comparison could yield misleading results for such syntactically different but semantically equivalent sequences. Therefore, as mentioned in Section 3.1, we design different variations of *CiT* employing both traditional similarity metrics and Siamese neural network [34] to detect inconsistency attacks in NFV. Our experimental results (Section 4.3) show that translation coupled with Siamese neural network can achieve the best accuracy.

**Data Labeling.** To facilitate the training of Siamese networks for detection, we prepare two types of training datasets: (i) pairs of event sequences at the same level (e.g., NFV level), and (ii) pairs of event sequences at two different levels (e.g., NFV and cloud). For each pair of sequences, we also provide a ground truth similarity score based on whether the two sequences correspond to the same user-level request; if so, a pair is called a *consistent pair*; otherwise, it is an *inconsistent pair*. In this stage, all the data are collected in a controlled environment, and abnormal events, such as execution failures, are excluded from the labeling. High quality training data is essential for building a robust model.

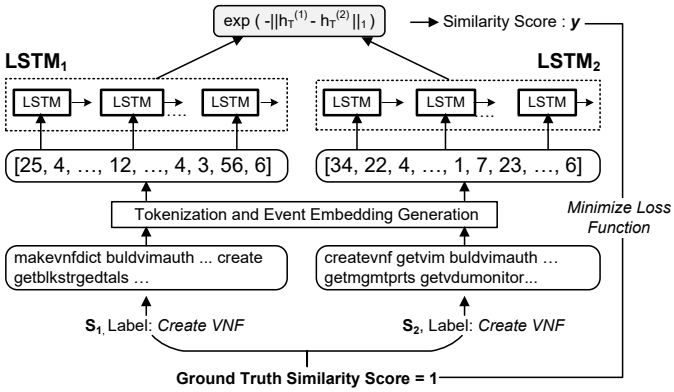



Fig. 6: An example of training the Siamese Manhattan Network with NFV-level event sequences

**Training.** The training process will vary depending on which variation, i.e., $CiT_{ts}$, $CiT_{tm}$, or $CiT_s$ (as mentioned in Section 3.1) of our detection technique is involved. First, since $CiT_{ts}$ compares the translated result to the original event sequence, the Siamese network is trained using the first type of dataset (i.e., pairs of event sequences at the same level). Second, no training is needed for $CiT_{tm}$ as it compares using a similarity metric (e.g., Manhattan distance) instead of Siamese network. Third, since $CiT_s$ directly compares the embedded event sequences between the NFV and cloud levels, the Siamese network is trained using the second type of dataset (i.e., pairs of event sequences from two different levels).

***Example 3.*** *Figure 6 shows the training of the Siamese network with a pair of event sequences* $S_1$ *and* $S_2$*. Since both event sequences correspond to the same user-level request, create network function, this is a consistent pair and its ground truth similarity score is* 1*. First, the events in* $S_1$ *and* $S_2$ *are embedded and fed into two LSTM models,* $LSTM_1$ *and* $LSTM_2$ *respectively. The LSTM networks then learn the sequence embedding for both* $S_1$ *and* $S_2$*. Finally, the similarity score,* y*, is calculated as the negative exponent of the Manhattan distance using the sequence embedding. The training of Siamese network mainly focuses on minimizing the loss between the calculated similarity score and ground truth.*

**Cross-validation.** To ensure the model's ability in accommodating unseen event sequences, the training dataset is partitioned into three distinct subsets, comprising training, validation, and testing data. Importantly, both the validation and the testing datasets are deliberately withheld from the training procedure. The training process strictly follows the standard cross-validation protocol, a widely accepted practice that safeguards against bias and ensures equitable evaluation. This procedural framework enhances the model's generalization capabilities and validates its performance across diverse datasets.

**Robustness.** The robustness of the detection model is established through integrating a high-quality and diversified training dataset. The implementation of cross-validation extends the model's performance generalization capabilities across distinct subsets of the data. Furthermore, we fine-tune all the hyperparameters in the neural network models and implement early-stop to monitor and avoid overfitting.

Effective data quality, diverse dataset representation, robust cross-validation, and careful hyperparameter tuning collectively contribute to building a model that performs well on the training data and generalizes effectively to new, unseen data. These practices enhance the model's ability to handle various challenges and uncertainties in real-world applications. We performed three experiments in Section 4.4 to evaluate the robustness of our trained model.

**Detection.** *CiT* detects inconsistency attacks at two different levels of granularity: event-level (i.e., within the execution of one user-level request) and workflow-level (i.e., during the execution of a sequence of user-level requests), which is detailed in the following.

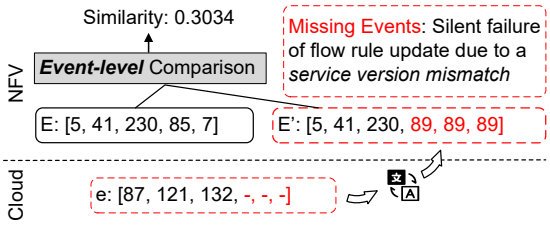



Fig. 7: An example of event-level inconsistency detection

*Event-Level Detection.* As an example, Figure 7 depicts a real-world inconsistency that occurred in our NFV testbed, which is diagnosed to be related to a version mismatch between Neutron (networking service) and Open vSwitches (OvS). Specifically, when the user level request *E* is performed at NFV level, a series of corresponding events denoted as *e* are executed at cloud level. However, due to the version mismatch issue, some events are missing at the cloud level and the user's request for updating

OvS flow rules fails silently (without returning any error message). To detect this inconsistency, *CiT* first translates $e$ to the corresponding NFV level event sequence $E'$. When the translation module hits the unknown part of the event sequence $e$ (missing events in our case), it will repeat the last known event, which can be observed as the repetitive events 89 in the translated event sequence. This inconsistency is detected by the Siamese network through comparing the translated $E'$ to the actual $E$, which results in a low similarity score (Manhattan distance of 0.3034).

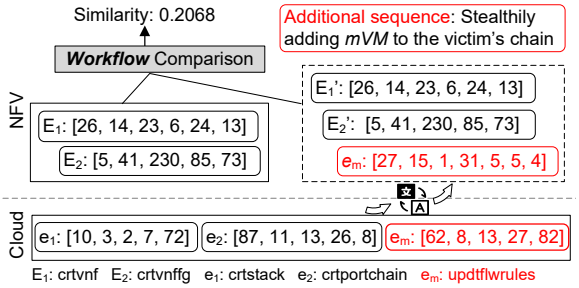



Fig. 8: An example of workflow-level inconsistency detection

*Workflow-Level Detection.* Figure 8 shows an example for workflow-level inconsistency detection, where a malicious resource (e.g., *mVM* in the motivating example) is stealthily added at cloud level without leaving any trace at NFV level. Specifically, the example workflow includes two requests, i.e., $E_1$ and $E_2$. An attacker operating at cloud level generates a request $e_m$ to inject a malicious VM into the victim's chain to sniff sensitive information. *CiT* can detect such inconsistency by first translating the workflow from cloud level back to NFV level and then calculating the similarity score between the translated and the original workflows. Due to the additional event sequence in the translated workflow, the inconsistency is detected through a relatively low (0.2068) similarity score between the original and the translated workflows.

### 3.5 Diff-based TOSCA Verifier

Inconsistencies may still exist after the event sequence similarity comparison as we remove the parameter values from the raw log entries as described in Section 3.2. As a result, we lose few implementation specific details, such as the technical specifications of a VNF, applications installed inside the VNF, etc. To compensate the loss, we utilize the parameters from service logs to generate TOSCA template that reflects the implementation details to verify the correctness of resource allocation. TOSCA template is stored in the YAML format that could be compared between the elements (i.e., requested resources, such as VDUs), e.g., mem_size is an attribute that denotes the memory size of a requested VDU.

In Algorithm 1, Lines 1-2 obtain the logs that associate with the input tenant ID. The logs belong to this tenant gets separated into element, e.g., VDU, and attribute values, e.g., *mem_size = 256MB* in Lines 3-4. Then the element gets identified in the input default TOSCA template and changes the attribute values into the ones obtained in Lines 5-6. Once all the logs are processed, a TOSCA template contains lower level details, which will be returned. Then Algorithm 2 takes the TOSCA user template and TOSCA generated template as inputs and generate the difference from Lines 1-3. In the end, the inconsistency between the requested resources and deployed resources will be return to the end user.

**Algorithm 1:** TOSCA TEMPLATE TRANSLATOR

```
   Input: Services Logs, Tenant_ID, TOSCA Template.Default
   Output: TOSCA Template.Translated
1  for log in Services Logs do
2      Tenant_ID.logs = get(Services Logs, Tenant_ID)
3  for log in Tenant_ID.logs do
4      attribute.value, element = get(log)
5      search element in TOSCA Template.Default
6      add attribute.value to TOSCA Template.element
7  return TOSCA Template.Generated
```

**Algorithm 2:** TOSCA TEMPLATE COMPARISON

```
   Input: TOSCA UserTemplate, TOSCA GeneratedTemplate
   Output: TOSCA diff
1  d = difflib.Differ() for text1, text2 in UserTemplate,
     GeneratedTemplate do
2      diff = d.compare(text1, text2)
3  return newline.join(diff)
```

## 4 EXPERIMENT

In this section, we first provide the implementation details and dataset description, and then present our experimental results (accuracy, efficiency, robustness, and usability) of *CiT* using both real and synthetic data.

### 4.1 Implementation and Experimental Details

In both the translation and inconsistency detection modules of *CiT*, we leverage the *word2vec* [51] model from *Genism* [52] to learn event embeddings from the corpus. The embedding vector is then fed into the embedding layer implemented based on *Keras* [53]. The deep learning layer, e.g., LSTM, is implemented based on the *Keras.layers* library to generate the event sequence embeddings. For comparison, we also implement GRU and RNN. The implementation of the evaluation metrics, e.g., *loss* and AUC, is based on *scikit-learn* [54], a well-known ML library. The implementation of data preparation is based on *pandas* [55], a data analysis library. All the modules of *CiT* are developed in Python 3.7.4.

To evaluate *CiT*, we implemented an NFV testbed with OpenStack [56] as the VIM component that manages the virtual infrastructure. Tacker [23], an official OpenStack project for building generic VNFM and NFVO based on the ETSI MANO architectural framework, is used to deploy virtual network services on the VIM. All the experiments (unless explicitly stated otherwise) are performed on a SuperServer6029P-WTR running Ubuntu 18.04 equipped with Intel(R) Xeon(R) Bronze 3104CPU @ 1.70GHz and 128GB of RAM *without* GPUs.

TABLE 2: Dataset statistics (the gray shaded datasets are processed real data)

| Dataset | Training | | | Validation | | | Testing | | | Total | | | Service Relationship |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Con. | Incon. | Total | Con. | Incon. | Total | Con. | Incon. | Total | Con. | Incon. | Total | |
| D1: Tacker-SFC | 16,023 | 15,628 | 31,651 | 1,960 | 1,996 | 3,956 | 1,968 | 198 | 2,166 | 19,951 | 17,822 | 37,773 | Dependency |
| D2: Tacker-Heat | 65,223 | 64,956 | 130,179 | 8,090 | 8,182 | 16,272 | 8,416 | 785 | 9,201 | 81,729 | 73,923 | 155,652 | Dependency |
| D3: Heat-Nova | 84,952 | 77,415 | 162,367 | 10,660 | 9,636 | 20,296 | 10,574 | 972 | 11,546 | 106,186 | 88,023 | 194,209 | Composition & Dependency |
| D4: Heat-Neutron | 65,343 | 62,258 | 127,601 | 8,119 | 7,831 | 15,950 | 8,092 | 785 | 8,877 | 8,1554 | 70,874 | 152,428 | Composition & Dependency |
| D5: Neutron-OvS | 14,022 | 12,344 | 26,366 | 1,674 | 1,622 | 3,296 | 1,742 | 155 | 1,897 | 17,438 | 14,121 | 31,559 | Collaboration & Association |
| D6: Heat-Nova | 64,423 | 37,542 | 101,965 | 8,073 | 4,673 | 12,746 | 8,080 | 466 | 8,546 | 80,576 | 42,681 | 123,257 | Composition & Dependency |
| D7: Heat-Neutron | 76,209 | 39,992 | 116,201 | 9,578 | 4,947 | 14,525 | 9,491 | 503 | 9,994 | 95,278 | 45,442 | 140,720 | Composition & Dependency |
| D8: Neutron-OvS | 187,635 | 67,036 | 254,671 | 23,574 | 8,260 | 31,834 | 23449 | 838 | 24,287 | 234,658 | 76,134 | 310,792 | Collaboration & Association |
| D9: Mixed-levels | 56,758 | 54,625 | 111,383 | 7,078 | 6,845 | 13,923 | 7,061 | 686 | 7,747 | 70,897 | 62,156 | 133,053 | All |
| D10: Mixed-levels | 85,245 | 63,315 | 148,560 | 10,671 | 7,899 | 18,570 | 10,520 | 805 | 11,325 | 106,436 | 72,019 | 178,455 | All |
| **Total** | 715,833 | 495,111 | 1,210,944 | 89,477 | 61,891 | 151,368 | 89,393 | 6,193 | 95,586 | 894,703 | 563,195 | 1,457,898 | |

TABLE 3: Statistics of the original real data (from May 2017 to March 2020) including the # of log entries for each service

| *Size* | # of services | Heat | Nova | Neutron | OvS | *# of event types* |
|---|---|---|---|---|---|---|
| 47.5G | 10 | 951,053 | 1,977,847 | 3,957,313 | 2,950,495,169 | 164 |

### 4.2 Datasets

Table 2 summarizes all the datasets used in our experiements. In total, we obtain 26,356 unique event sequences, and generate 894,703 *consistent* pairs (i.e., two event sequences corresponding to the same user-level request) and 563,195 *inconsistent* pairs.

**Real Data.** We have collected around three years of OpenStack logs from a real cloud hosted at a major telecommunications vendor with hundreds of users. Table 3 shows some statistics of the original data, which we have processed to obtain the datasets D6, D7, D8, and D10 shown in Table 2 following the approach described in Section 3.4. In doing so, we processed 47.5G of raw data and obtained 164 event types from four different services, i.e., Heat (orchestration service), Nova (compute service), Neutron (networking service), and OvS (Open vSwitch). To obtain realistic testing datasets, the inconsistent pairs in testing datasets are generated based on real-world bug patterns (where the inconsistency is caused by OpenStack implementation bugs) and denied event sequences in the real data (where the inconsistency is caused by a policy violation). Among all the processed raw data, 2$\sim$8% of the data corresponds to denied user-level operations in different services. Therefore, we inject *inconsistent* pairs following the similar percentage. Following the standard approach of cross-validation, we split the prepared dataset in a random fashion into three subsets for training, validation, and testing purposes. By doing so, we can evaluate the ability of *CiT* towards handling unseen event sequences.

**Data Generation in the NFV Testbed.** We have implemented an NFV testbed to collect synthetic datasets from the NFV stack, including the Orchestration Level (L1) which is not present in our real data. In Table 2, datasets D1 through D5, and D9 are obtained from the NFV testbed. We developed Python scripts to automatically generate Topology and Orchestration Specification for Cloud Applications (TOSCA templates to deploy NFV entities, such as VNFs and VNF forwarding graphs (VNFFGs). In total we have deployed 31 types of VNFs (e.g., with auto-scaling policies, dedicated subnet, floating IPs, etc.), and 7 variations of VNFFGs to create sufficient diversity in the corresponding event sequences. We have also randomized some important parameters in the template description, such as 1) the number of virtual network ports per VNF, 2) the number of deployment units per VNF, 3) the node flavor specification for each VDU, 4) the number of VNFs for each Network Forwarding Path (NFP), 5) the order of VNFs for each NFP, 6) the flow-classifier criteria for each NFP, and 7) the number of NFPs for each VNFFG.

**Generating Datasets with Ground Truth.** As described in Section 3.4, in the dataset generation, two event sequences corresponding to the same user-level operation are assigned with the *consistent* label, and vice versa. During data generation in the NFV testbed, we preserve the user-level operations, which can be utilized to create *consistent/inconsistent* pairs. For example, event sequences from the same operation but at different levels, such as *create VNF*, are labeled as *consistent* pairs. Unfortunately, the user-level operation is missing from the real data we obtained. To address this issue, we applied the model trained based on the data collected from our testbed to predict the user-level operation for the real data. All the predicted labels are then validated by two domain experts to ensure the correctness of the labeling.

**The Relationships between Different Datasets.** The OpenStack services operate together and enable orchestration, resource provisioning, management, and implementation of the NFV system. Therefore, these services exhibit different kinds of functional relationships between each other such as, association, collaboration, composition, and dependency. Specifically, the *association* relationship refers to a service complementing or enhancing the capabilities of another service (e.g., Neutron-OvS). The *collaboration* relationship is established when two services work together to accomplish a common task (e.g., Neutron-OvS). *Composition* relationship is created when a single entity acts as a combination of functionalities provided by multiple entities (e.g., Heat-Nova & Heat-Neutron). Finally, *dependency* relationship means one service completely relies on another service to perform specific, essential tasks (e.g., Tacker-Heat).

### 4.3 Inconsistency Detection Evaluation

In this section, we evaluate the accuracy of *CiT* and its three variants. First, we compare $CiT_s$ (Siamese network applied to both levels) to a well-known traditional method for sequence classification, Support Vector Machine (SVM) [57] with the TFIDF feature set [58], [59] (which is usually used to extract text features for anomaly detection and failure prediction [60]) based on all the 10 datasets including both real

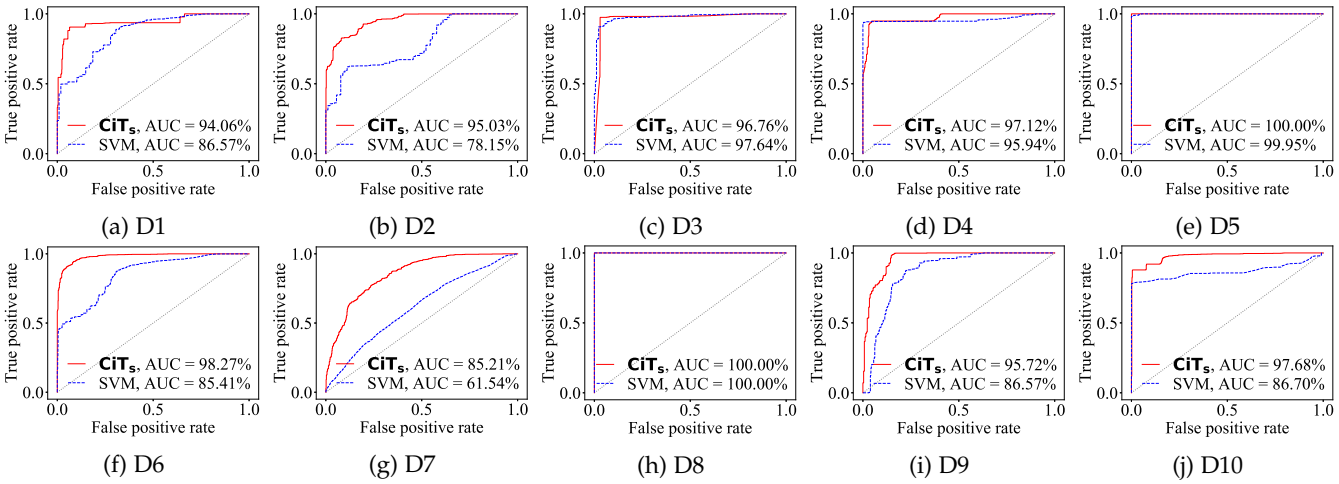



Fig. 9: The ROC evaluation results of $CiT_s$ based on all the 10 datasets

and testbed data. Second, we compare the three variations of *CiT*, i.e., $CiT_s$, $CiT_{tm}$, and $CiT_{ts}$ in terms of accuracy.

#### 4.3.1 $CiT_s$ vs. SVM

We select Support Vector Machine (SVM) as a baseline to compare to $CiT_s$ since SVM is a widely used anomaly detection technique (as reported in several surveys [61], [62]) which has particularly been applied in many sequence-based detection works for anomaly and malware detection [63]–[67]. Specifically, these studies have employed SVM for sequence-based anomaly and malware detection across different domains, including Linux system calls [66], opcode sequences [64], and system call sequences on IoT devices [65], [67] and home routers [63]. We evaluate the SVM model (implementation from scikit-learn) based on five kernels in which kernel *rbf* performs the best followed by *poly*, and *linear*. Thus, we only present the results generated from kernel *rbf* ($\gamma = 1.0$, $c = 1.0$), which achieves the best AUC, in comparison to $CiT_s$ (Siamese network applied to both levels) to study the discriminative power of TFIDF features and deep learning generated features (embedding).

**Model Training.** In this set of experiments, the training, validation and testing datasets follow the statistics presented in Table 2. We use 10 datasets to train $CiT_s$ up to 200 epochs. We implement early stopping [68] to avoid overfitting. We choose the *loss* value of the corresponding validation dataset as the performance measure with the trigger parameter set to *patience=3*. While training the model, *loss* will be monitored as it is calculated after each epoch, and if there is no improvement in the *loss* value for 3 epochs, the training stops. Other hyperparameters, such as the dimensions of the event embedding and the dimension of sequence embedding, are set to 256 and 200, respectively (a more detailed study of hyperparameters is presented in Section 4.5).

The training dataset for the SVM model is the same as that for $CiT_s$. The implementation of TFIDF features is from *scikit-learn*, which converts raw text inputs into a matrix of TFIDF features. We set the dimension of the feature sets to 300 following the literature [59] in Figure 9. We also evaluate the performance of SVM under small feature dimension (*max_features = 5*) and large feature dimension (*max_features = 1000*). Compared to 300 dimensional TFIDF feature sets, small feature dimension performs worse, while large feature dimension does not show stable improvements.

**Inconsistency Detection Results for $CiT_s$.** We then evaluate the accuracy of $CiT_s$ using the corresponding testing datasets of D1 to D10 that include real bugs and denied event sequences. For the testbed datasets, the AUC value of $CiT_s$ for inconsistency detection increases for all the datasets from D1 (AUC = 94.06%) to D5 (AUC = 100%). Note that datasets from the higher level services, such as D1 (Figure 9a) and D2 (Figure 9b), generally consist of longer event sequences, e.g., Tacker service generates orchestration events that could be implemented with multiple lower level services; therefore, $CiT_s$ only achieves AUC = ∼95%. On the other hand, D3 (Figure 9c) to D5 (Figure 9e) consist of lower level service events, which generally have less events and shorter event sequences, and thus $CiT_s$ achieves AUC $\geq$ 97.76%. Notably, the OvS service has only one event, *flow_mods*, to implement the requested flows from the Neutron service. Thus, the inconsistency detection is relatively easy for both $CiT_s$ and SVM (both reaching ∼100%). In general, the value of AUC increases as the complexity of the datasets decreases due to event sequences from only lower-level services. It is worth mentioning that the relationship types between different services (as explained in Section 4.2) do not have any notable impact on the detection accuracy.

For the real data, $CiT_s$ performs well with D6 (Figure 9f) and D8 (Figure 9h), i.e., AUC increases from 98.27% to 100%, which leads to a similar conclusion drawn from the testbed datasets. However, the inconsistency detection with D7 (Figure 9g) results in only 85.21%. This is mainly because the complexity of this dataset is significantly higher due to the increased diversity of network-related events in real data, e.g., one event sequence from Heat service could be implemented in various ways based on user chosen templates. As shown later, this result will be significantly improved with $CiT_{ts}$ and $CiT_{tm}$ since translation helps to reduce the detection complexity as explained in Section 4.3.2. However, $CiT_s$ still demonstrates good detection ability for both the mixed testbed data and real data, (AUC = 95.72% and 97.68%). AUC is slightly lower for the testbed dataset

because the long sequences from orchestration level (D1 and D2), which are missing in the real dataset, increase its complexity.

**Comparison with SVM.** After the models of $CiT_s$ and SVM are trained, we use the same testing datasets to evaluate both. In general, the comparison between the two models in terms of detecting inconsistencies shows similar results across different datasets. We can observe that for all datasets, the ROC curves corresponding to $CiT_s$ are closer to the left-hand and top border than SVM models, which indicates our model generally has a better accuracy. The majority of our models yield an ROC-AUC value higher than 94%, while SVM stays around or less than 85% for six of the 10 datasets (D1, D2, D6, D7, D9, and D10). Comparing to SVM, the improvement of $CiT_s$ is more significant when tackling the datasets with longer or more diversified event sequences. This observation confirms that the embedding and deep learning algorithms are more capable of handling complicated event sequences.

#### 4.3.2 $CiT_s$ vs. $CiT_{tm}$ vs. $CiT_{ts}$

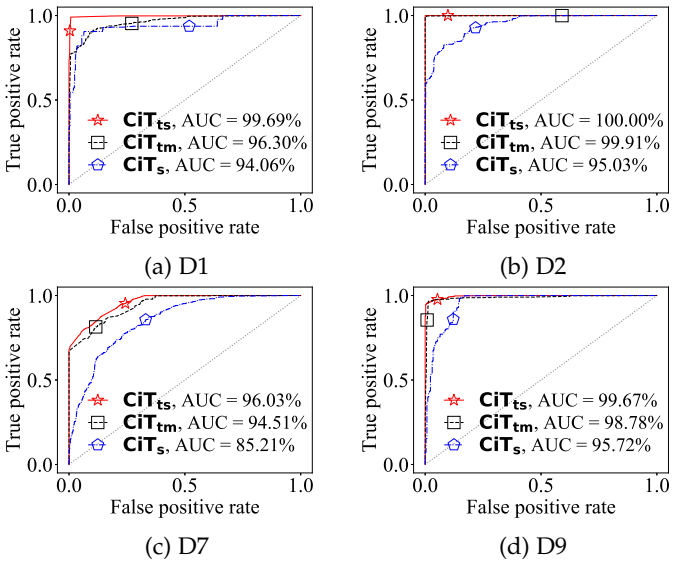



Fig. 10: The ROC evaluation results of the three variations of *CiT* based on four datasets

In this set of experiments, we compare the three variants of *CiT* to evaluate the improvement of accuracy (AUC) due to additional translation involved in $CiT_{tm}$ and $CiT_{ts}$. Recall that $CiT_{ts}$ applies LSTM to translate event sequences before applying Siamese network on the embedded event sequences at the same level for detection, whereas $CiT_{tm}$ translates event sequences but then performs detection based on the Manhattan distance, as described in Section 3.4. For these experiments, we use the datasets (D1, D2, D7, and D9) for which $CiT_s$ achieves relatively lower AUC (<96%).

**Comparison of Inconsistency Detection Results.** We present the ROC curves for the aforementioned four datasets for three variants of *CiT* in Figure 10. We observe that $CiT_{tm}$, detecting inconsistencies based on Manhattan distance, achieves satisfying results (AUC>94.51%). Moreover, $CiT_{ts}$, combining LSTM and Siamese network, achieves AUC = ∼100% in three datasets (D1, D2 and D9) shown in Figure 10a, 10b, and 10d. With the help of translation, the AUC in D7 improved from 85.21% to 94.51% ($CiT_{tm}$) and 96.03% ($CiT_{ts}$), respectively. These results show that the translation module can significantly improve the accuracy of inconsistency detection and overall, $CiT_{ts}$ achieves the best results.

### 4.4 Robustness Evaluation

We conduct three experiments to evaluate *CiT*'s robustness.

#### 4.4.1 Training and Testing with Different Systems

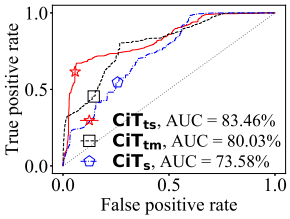
Since a well-known challenge in applying machine learning to security is to obtain attack-free training data (esp. from real production systems), we evaluate the feasibility of training *CiT* using one NFV system (e.g., a testbed deployed in a controlled environment), and then applying the learned models for detection in another NFV system (e.g., a real production system). In particular, we apply the inconsistency detection models trained on our testbed dataset D9 to test the real-world dataset D10. These two datasets are obtained from two very different NFV systems (e.g., different OpenStack release and configurations), with one implemented by ourselves for experimental purposes, and the other hosted at a major telecommunications vendor with hundreds of real-world users. Therefore, this experiment represents a stress test that evaluates the robustness of *CiT*.

**Results.** The experimental results in Figure 11a show that, even under a challenging scenario with two significantly different datasets (e.g., differing in event types), $CiT_s$ still achieves an acceptable AUC (=73.58%). Aligned with Section 4.3, Figure 11a shows that the extra translation step helps to improve the AUC to 80.03% ($CiT_{tm}$) and 83.46% ($CiT_{ts}$), respectively. These results confirm the robustness of *CiT* as well as the feasibility of training *CiT* in a controlled environment and applying it to a real world NFV system.

#### 4.4.2 Real-World Inconsistency Detection

To investigate how *CiT* detects real world inconsistencies, we conduct the following experiment using event sequences including both real-world bugs and denied operations found in our real data. Specifically, the sequences are compared to the normal sequences from the same user-level request for inconsistency detection. As shown in the Table in Figure 11b (third column), most of those bugs and denied operations have a direct implication on security or privacy, with six bugs corresponding to existing CVE vulnerabilities. The table also shows the comparison between *CiT* and NFVGuard [69] in detecting real-world bugs and denied operations, which we explain in Section 4.7.

**Results.** Table in Figure 11b summarizes the results of this case study including a detailed description of the bugs or denied operations and their corresponding similarity scores. As we can observe, majority of the sequences are assigned with a lower similarity score (<0.5). Meanwhile, three of these obtain a comparatively higher similarity score (>0.7). Our investigation shows that this is mainly due to the fact that these sequences are relatively shorter (with less events) and the bugs do not introduce significant differences to

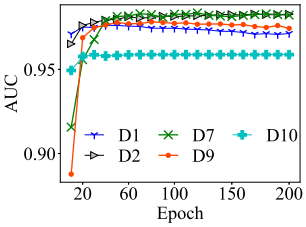



(a) D10 tested on D9 model

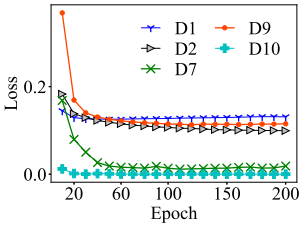
| # | Description | Security Implication | Sim. Score | CiT | NFVGuard |
|---|---|---|---|---|---|
| Bug#1527658 | Block Device Mapping is Invalid | Implementation Error | 0.003811 | Y | Y |
| Sequence#1 | ValueError: ⟨name⟩: nics are required after microversion ⟨Version #⟩ | Configuration Error | 0.01496 | Y | Y |
| Bug#1653164 | CinderVolume ⟨name⟩ Stack ⟨name⟩ [id] timed out | Denial of Service (CVE-2015-5162) | 0.1791 | Y | Y |
| Sequence#2 | NotFound: resource with id <id> not found | Gain Information (CVE-2012-4403) | 0.2003 | Y | Y |
| Sequence#3 | Resource CHECK failed | Implementation Error | 0.2395 | Y | Y |
| Bug#1517355 | Conflict: Port ⟨id⟩ is still in use | Denial Of Service (CVE-2019-9735) | 0.2436 | Y | Y |
| Sequence#4 | OverQuotaClient: Quota exceeded for resources | By Pass (CVE-2017-0887) | 0.33677 | Y | Y |
| Sequence#5 | Error: Volume in use | Resource Saturation (CVE-2013-1664) | 0.4380 | Y | Y |
| Sequence#6 | Resource CREATE failed: You are not authorized to use resource_types ⟨name⟩ | Privilege Escalation (CVE-2016-7404) | 0.7047 | N | Y |
| Bug#1833455 | Forbidden: rule ⟨event⟩ is disallowed by policy | Violation of Policy | 0.7396 | N | Y |
| Bug#1808112 | Forbidden: resources. Offline rule ⟨event⟩ is disallowed by policy | Violation of Policy | 0.7969 | N | Y |

(b) CiT vs. NFVGuard [69] on detecting real-world bugs and denied operations

Fig. 11: Robustness evaluation of *CiT*. The model in Figure 11a is trained in testbed dataset and tested with real data. The bugs and the denied operations in Figure 11b are real event sequences we gathered from the real data.




(a) AUC vs. # of epochs




(b) Loss vs. # of epochs

| Em | Datasets | | | | | |
|---|---|---|---|---|---|---|
| | D9 | | | D10 | | |
| | $CiT_s$ | $CiT_{tm}$ | $CiT_{ts}$ | $CiT_s$ | $CiT_{tm}$ | $CiT_{ts}$ |
| 50 | 93.14 | 95.04 | 97.72 | 95.42 | 96.01 | 98.16 |
| 100 | 93.63 | 96.49 | 98.29 | 95.65 | 96.55 | 98.82 |
| 150 | 94.88 | 97.70 | 98.81 | 95.92 | 97.13 | 99.59 |
| 200 | 95.42 | 98.11 | 99.49 | 96.33 | 97.35 | 99.73 |
| 250 | 95.72 | 98.78 | 99.67 | 97.68 | 98.36 | 99.92 |

(c) AUC (%) vs. event embedding dimensions (U = 256)

| Type | Datasets | | | | | |
|---|---|---|---|---|---|---|
| | D9 | | | D10 | | |
| | $CiT_s$ | $CiT_{tm}$ | $CiT_{ts}$ | $CiT_s$ | $CiT_{tm}$ | $CiT_{ts}$ |
| LSTM | 95.72 | 98.78 | 99.67 | 97.68 | 98.36 | 99.92 |
| GRU | 93.73 | 95.76 | 94.33 | 93.81 | 95.82 | 96.78 |
| RNN | 92.54 | 93.24 | 93.64 | 93.66 | 94.78 | 96.45 |

(d) AUC (%) vs. network hidden unit types (U = 256/Em = 250)

| U | Datasets | | | | | |
|---|---|---|---|---|---|---|
| | D9 | | | D10 | | |
| | $CiT_s$ | $CiT_{tm}$ | $CiT_{ts}$ | $CiT_s$ | $CiT_{tm}$ | $CiT_{ts}$ |
| 32 | 93.24 | 95.37 | 96.77 | 94.55 | 96.49 | 97.76 |
| 64 | 93.87 | 96.16 | 97.46 | 95.61 | 97.24 | 98.07 |
| 128 | 94.81 | 96.86 | 98.12 | 95.85 | 97.64 | 98.55 |
| 192 | 95.12 | 97.38 | 98.73 | 96.36 | 97.99 | 99.38 |
| 256 | 95.72 | 98.78 | 99.67 | 97.68 | 98.36 | 99.92 |

(e) AUC (%) vs. event seq. embedding dimensions (Em = 250)

Fig. 12: The impact of different hyperparameters of *CiT*. Figure 12a and Figure 12b are evaluated based on the corresponding validation datasets, and others are evaluated on the testing datasets.

the events. For example, the event sequence corresponding to *Bug#1808112* is *creatingport createport stackcreatefailed*, whereas the normal sequence is *creatingport createport stackcreatecomplete*, i.e., only one event is different, and both are relatively short sequences. We may overcome such a situation by extending *CiT* with a *weighted* similarity score measure (e.g., [70], [71]) to focus more on the important events, e.g., *stackcreatefailed* (which indicates the requested operation has failed due to an error); which will be considered as a future work.

### 4.4.3 Label Translation for Real Data

Our real data does not include the corresponding user-level requests, which are needed as labels to form *consistent* and *inconsistent* sequence pairs. It is infeasible to label these manually considering that there are 8,192 unique event sequences in total in real data. Thus, we utilize the translation module of *CiT* to translate the corresponding labels (user-level requests) of event sequences from our real-world data by feeding them as testing dataset.

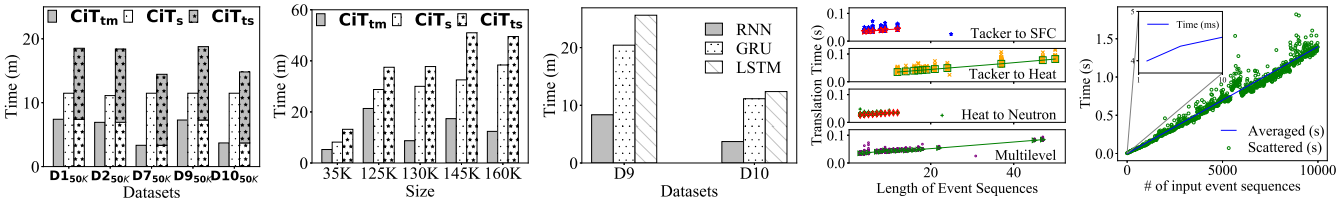
**Correctness of the Label Translation.** The translated labels are randomly selected (∼10%) and cross-validated by two domain experts to ensure the correctness of the labeling.

**Results.** The label translation results are presented in Table 4. We only show the most commonly occurring user-level requests for each service and their corresponding

TABLE 4: Sequence label translation for the real data, (%) indicates the percentage of correctly translated labels

| Service (%) | Event Sequence Label (%) | Description |
|---|---|---|
| Heat (∼94) | CreateServer (100) | Creating a VM |
| | CreateStack (84) | Multiple operations |
| | CreateCinderVolume (94) | Creating a Cinder Volume |
| | DeleteServer (100) | Deleting a VM |
| | DeleteStack (89) | Multiple operations |
| Neutron (∼75) | CreatePort (100) | Creating a virtual port |
| | CreateFloatingIP (100) | Creating a floating IP |
| | UpdateFlowRuleStatus (70) | Updating OvS flow rules |
| Nova (∼88) | CreateServer (100) | Creating a VM |
| | OSServerExternalEvents (85) | Attaching a floating IP |
| | OSSecurityGroupRules (100) | Assigning a security group |

accuracy of label translation. Our translation module can correctly label up to ∼94% of the events for the Heat service followed by the Nova (∼88%) and Neutron (∼75%) services. Also, most of the events are labeled with 100% accuracy except a few. Particularly, the *CreateStack* event sequences of Heat service achieve a success rate of ∼84%. Upon investigation, we find that the real event sequences that fall under *CreateStack* category include multiple *create*, *update* and *delete* event types since a "stack" executes multiple requests at once. These event sequences are not correctly labeled by the model trained based on the testbed data, since D9 contains only *create* event type for the label *CreateStack*. Furthermore, the Neutron service achieves comparatively the lowest success rate of ∼75%. The reason is that the event




(a) *CiT* Training Time (b) Time vs. Seq. Pairs (c) Hidden Layers (d) *CiT* translation time (e) *CiT* detection time

Fig. 13: *CiT* efficiency study. The results in Figure 13a to Figure 13c are obtained based on LSTM with Em = 256 and U = 250. All the results shown in the figures are the training time per epoch. Figure 13d and Figure 13e show the testing time for *CiT*.

sequences labeled with *UpdateFlowRuleStatus* in D9 contain the Tacker service events, which are not included in dataset D10. The overall results again confirm the robustness of *CiT* in applying its trained models to a different system. The results also show *CiT* can serve as a tool for recovering missing information (e.g., user-level requests in our case) in real-world datasets.

### 4.5 Hyperparameter Selection

In this section, we evaluate the impact of hyperparameters in training *CiT*. Specifically, we first study the AUC and loss metrics in terms of number of epochs using five datasets (D1, D2, D7, D9, and D10), and then investigate the impact of (both event and sequence) embedding dimensions and network hidden unit types based on the mixed datasets (D9 and D10) for all three *CiT* variations.

**Number of Epochs.** Figures 12a and 12b show the results of AUC and loss metrics for translation training on the aforementioned five datasets, respectively. We train the translation model for 200 epochs and evaluate the model after each epoch. The curves of both AUC and loss metrics become flat after 20 epochs for 4/5 datasets (only D7 requires around 50 epochs to become stable, as the complexity of the dataset is higher). We can also observe that the early stop implemented in each model stops around 20 epochs. In summary, we conclude that 20 epochs could be enough to obtain a good model in *CiT*.

**Event and Sequence Embedding Dimension.** We study the impact of event embedding dimension (*Em*) and sequence embedding (*U*, the number of LSTM network units) in Figures 12c and 12e, respectively. D10 achieves better results with all three variations than D9, which aligns with the results we presented in Section 4.3. We conclude that embedding dimensions, for both event and sequence embedding, contribute positively to the accuracy of the models.

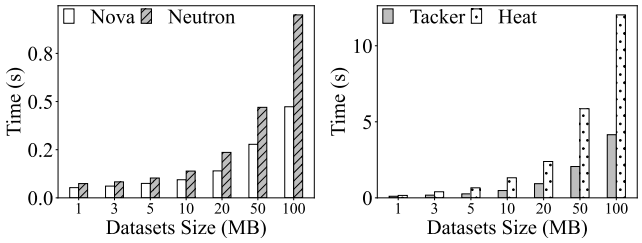
**Other Deep Learning Algorithms.** As the last part of the hyperparameter study, we conduct experiments on three types of neural networks including LSTM, Gated Recurrent Unit (GRU), and simple Recurrent Neural Network (RNN). As shown in Figure 12d, LSTM outperforms both GRU and RNN, while GRU and RNN perform similarly. Therefore, in our implementation, all the *CiT* approaches are trained based on LSTM to achieve the best detection results.

### 4.6 Efficiency and Scalability

In this section, we study the efficiency and scalability of *CiT* by first analyzing the training time for its three variations with five datasets (D1, D2, D7, D9, and D10), and the impact of hyperparameters with D9 and D10. Then we investigate the testing time for both translation and detection. We also evaluate the cost incurred by *CiT* in terms of log processing, bandwidth, and resource consumption.

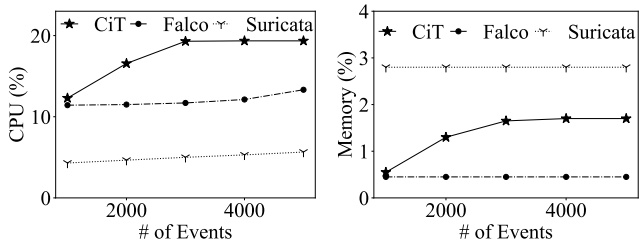
**Training Time.** Figure 13a shows the training time of *CiT* on the five datasets. For fair comparison, we take the same number (50,000) of input pairs from each dataset. In general, $CiT_{ts}$ (with both LSTM and Siamese) requires the longest training time. The color separation of the bars shows the training time for both translation and detection for this variation. However, in practice the translation and detection models can be trained in parallel to reduce the overall training time. We can also observe that the corpus size and the length of sequences both affect the training time. Since the total number of input pairs is the same for different datasets, the corpus size becomes the main factor that impacts the overall training time. For instance, the corpus size in D7 is smaller and with shorter sequences, the training time is shorter than other datasets. Figure 13b studies how the size of training dataset affects the training time for the three variations. In general, training time increases when the size of the dataset increases. The longest time needed for training the largest dataset is 49 minutes per epoch in $CiT_{ts}$. Parallel training for both models at the same time will reduce the overall training time to 38 minutes per epoch. The training time for network hidden layer types is shown in Figure 13c. We observe that LSTM requires longer training time than RNN and GRU due to its complexity (which also helps achieve better accuracy).

**Testing Time.** We investigate the testing time of *CiT* for both translation and inconsistency detection. Figure 13d shows the translation time required for each dataset. Generally, longer sequences naturally require longer translation time; however, our results show that even the longest sequence only takes 0.12s to finish translation. In Figure 13e, we evaluate the execution time of similarity score calculation versus the number of input event sequences by testing the trained $CiT_s$ model with pairwise input sequences. In the zoomed-in chart, we can observe the detection time is around *4ms* for a single event sequence pair. Furthermore, the detection model only takes 1.5s to calculate similarity scores for 10,000 pairs of inputs. In contrast, according to our real-world data, a user-level operation would take OpenStack around *77* seconds on average to execute in a real cloud. Therefore, we conclude that *CiT* is an efficient solution for detecting inconsistency attacks.




(a) Nova and Neutron Logs vs. Time (b) Tacker and Heat Logs vs. Time

Fig. 14: The log processing time in different NFV levels




(a) CPU Cost (b) Memory Cost

Fig. 15: The CPU and memory cost: $CiT$ vs. Falco & Suricata

**Log Processing Cost.** Figure 14 studies the log processing time at two different NFV levels: the implementation-level in Figure 14a and the orchestration-level in Figure 14b. In general, the orchestration services, Tacker and Heat, contain diverse event sequences with rich semantic context leading to longer processing times. Nevertheless, it only requires around *10s* to process 100MB of Tacker log data. On the other hand, the implementation-level with Nova and Neutron takes less than one second to extract the event sequences for *CiT*. For instance, it is reported that an Azure tenant with over 100,000 users generates about 2MB of audit log data per minute [72], which can be processed in less than 0.4 seconds based on our experiments. These results demonstrate that *CiT* has negligible log processing costs even for large-scale cloud deployments.

**Bandwidth Cost.** The bandwidth cost depends on the system's components and their corresponding technical architecture. Specifically, the NFV-level audit logs are generated by Tacker, which is the intended integration point for *CiT* as it compares the actual NFV implementation to the user specification. Therefore, collecting NFV-level audit logs does not incur any additional cost. Additionally, collecting logs from the cloud-level (e.g., Heat, Nova, Neutron) will only result in additional network bandwidth cost if they are deployed on a remote OpenStack controller that is located on a different node from where $CiT$ is located. However, such services are usually co-located with OpenStack controller in a real world deployment [73].

**Resource Consumption.** Figure 15 presents the results of CPU and memory utilization for $CiT$ compared to Falco and Suricata, two state-of-the-art real-time intrusion detection tools, across the event detection ranging from 1000 to 5000 incidents. We chose Falco [74], a cloud-native host intrusion detection system (HIDS) for Linux systems, which employs custom rules on kernel events to provide real-time alerts. Falco is proved to outperform Auditd, a native feature to the Linux kernel for monitoring system calls, in terms of resource consumption [75]. In addition, we also evaluated Suricata [76], a high-performance network intrusion detection tool (NIDS), which can be used as a VNF in OpenStack deployments [77]. Both Falco and Suricata were evaluated with their default configurations (including the pre-configured set of rules) against $CiT_s$. To perform these experiments, we follow a similar system specification that Falco used (a system with 8 CPU cores and 4GB RAM, operating with Ubuntu 22.04) in their evaluation against Auditd.

As shown in Figure 15a, Suricata outperforms both Falco and $CiT_s$ in the CPU consumption by utilizing only about 5% constantly. Conversely, the CPU consumption of $CiT_s$ increases from about 12% to 19% for 1000 to 3000 events, subsequently maintaining this level while achieving a decent performance. Figure 15b shows that Falco shows the lowest memory consumption (0.9%) while $CiT$ outperforms Suricata with about 3% as its highest usage.

Given that $CiT$ is an anomaly-based detection solution, it is expected to consume more resources than rule-based solutions such as, Falco and Suricata. This is a trade-off that allows $CiT$ to potentially detect unknown (0-day) attacks (that are reflected in the event sequences) which will not be detected by any rule or signature-based solution.

### 4.7 Limitations

To evaluate the potential limitations of *CiT*, we perform a comparison with NFVGuard [69], a formal verification tool to verify consistency of an NFV system. Table in Figure 11b shows the comparison of *CiT* and NFVGuard on detecting real-world bugs and denied NFV operations. As shown in the last two columns of the table, NFVGuard successfully detected all inconsistencies, thus showing better accuracy than $CiT$. This is expected and can be explained by the fact that NFVGuard relies on collecting the configuration (system state) data after the NFV operations (or the attack) are completed. In contrast, $CiT$ performs the inconsistency detection at runtime, which allows it to uncover the inconsistencies faster, with some limitations in the accuracy.

In terms of efficiency, the execution time for *CiT* only takes around 0.12s for the longest sequences (as shwon in Figure 13d and Figure 13e), while NFVGuard requires 1$\sim$5s to verify the consistency properties. Given that $CiT$ is an online tool, such efficiency is more important due to the impact on the timeliness of online detection. Regarding resource consumption, NFVGuard utilizes 18% to 21% of the CPU and occupies 0.8% to 1.2% of the memory to verify 1000 to 5000 VNFFGs. On the other hand, *CiT* only requires around 10% of the CPU and maximum 2% as the memory usage. It means that *CiT* shares a similar memory cost and requires less CPU compared to NFVGuard. Nonetheless, the overhead in resource usage incurred by $CiT$ persists constantly as it runs continuously, whereas NFVGuard runs on demand, resulting in temporary resource consumption.

## 5 Discussion

**Motivation for NMT.** As we observed in our study, one event sequence from NFV level usually corresponds to

multiple event sequences from another level. Even a simple operation, *CreateVNF*, could generate at many as 11 different event sequences depending on different system states at NFV level (e.g., available VMs, subnets, ports). Our experiments show that, on average, at least three different implementation-level event sequences could match each NFV-level event sequence. To put this into perspective, considering only 15 operations of 14 services, a rule-based approach would require at least $15 \times 11 \times 3 \times \binom{14}{2} = 45,045$ rules. With more operations/services in real NFV systems, it would be practically infeasible to manually create and maintain an exhaustive ruleset. This limitation motivated us to adopt an NMT-based approach to automatically learn the cross-level mapping between event sequences.

**Training and Retraining.** A well-known challenge faced by machine learning-based security solutions is to gather attack-free data for training. As shown in Section 4.4, the *CiT* models trained using data collected from our experimental testbed provided satisfactory results when tested on data from a completely different, real-world system. Such results confirm the feasibility of training *CiT* under a controlled environment and then applying it to a real-world NFV system (the training system could be made more similar to the real system than in our case, which will further improve the accuracy). We also recommend retraining to ensure high accuracy when there is a major system upgrade that introduces new event types, e.g., Tacker version 9.0.0 deprecated some legacy events and introduced new event types in OpenStack 2023.1 Antelope release [78]. Furthermore, since major upgrade or change typically happens periodically (every 6 months in case of Openstack [79]), retraining will be practically feasible and may not introduce service downtime as it is done offline. Nonetheless, our experiments (trained with both real and testbed datasets in Figure 11a) show that, even when the systems are several generations apart, the accuracy is still acceptable.

**False Positives.** Additionally, like any NMT-based application, $CiT$'s model needs to be retrained or refined periodically or upon major changes made to the cloud infrastructure. Otherwise, $CiT$ may misclassify certain benign events (such as load balancer events, telemetry log collector events, accounting events, monitoring events, and non-malicious flow redirects) as malicious, if such events are not included in the training data. We can also adjust the detection threshold to a lower value to accommodate such unforeseen events to tackle this issue. But we recommend retraining the model instead of adjusting the threshold as the latter may introduce a risk of potentially allowing malicious events to go undetected.

**Cross-level Detection.** In an NFV stack (see Figure 2), the Service Orchestration level (also known as the NFV-level in our examples) is the user specification of the system and all the underlying levels are the actual deployment of this specification. Therefore, we focus on the pair-wise consistency between the NFV level (user specification) and every underlying level (actual deployment) to address NFV tenants' concerns. However, assuming "transitivity" of consistency (i.e., if underlying levels are consistent with the NFV level then those levels must also be consistent among themselves), detecting pair-wise inconsistency between every underlying level and the NFV level would enable us to identify inconsistency between multiple levels of the NFV stack. We consider this as a future work.

**Evasion Attacks.** Since *CiT* detects cross-level inconsistencies, the only way for attackers to evade detection is by generating a lower-level event sequence that exactly matches the corresponding higher-level event sequence. This is only possible if the attacker has total control of the lower-level events' logging mechanism (which is out-of-scope according to our threat model given in Section 2.4). Note if an attacker modifies a higher-level event sequence to match the lower-level "attack" sequence, then the attack would no longer be stealthy (visible to end-users).

**Adapting to other NFV Platforms.** Most of the modules of *CiT* are platform agnostic. Thus, it can potentially be adapted to other NFV platforms (e.g., OSM [80] and OPNFV [81]) for inconsistency detection, since it has been designed based on the generic NFV architecture and deployment model.

## 6 Related Work

This section reviews existing approaches to NFV security, anomaly detection, and translation-based security solutions, and compares them to our work.

**NFV Security.** Most existing solutions in NFV for inconsistency detection (e.g., [15]–[18], [20], [21], [82]) verify the cloud-level configuration information (e.g., flow rules and flow classifier) while focusing on one particular level (mostly SFC). ChainGuard [17] and SFC-Checker [18] both verify the correct forwarding behavior of SFCs. vNFO [20] and SLAVerifier [21] verify a wide-range of SFC functionalities (e.g., performance and accounting). Wang et al. [16] propose a framework to detect the dependencies and conflicts between network functions. Unlike those approaches which rely on configurations, *CiT* is an event-based approach which means it can potentially catch an attack before it incurs any damage.

**Anomaly Detection on Sequential Data.** There exist many works (e.g., [29], [30], [60], [83]–[85]) that conduct anomaly detection on sequential data (e.g., event logs, credit card transactions, and network traffic). Particularly, DeepLog [60] and Brown et al. [85] leverage RNNs to detect anomalies in system logs. Jurgovsky et al. [84] use the LSTM-based classification to detect anomalies in credit card transactions. Radford et al. [29] utilize LSTM to learn and predict communications between two IPs for anomaly detection. Similarly, Xiao et al. [30] utilize the semantic information of system call sequences for Android malware detection. Besides focusing on a different context (NFV), *CiT* applies the additional translation step before detection, which outperforms traditional anomaly detection (see Section 4.3).

**Translation-based Security Approaches.** Most of the existing solutions that leverage NMT (e.g., [28], [30], [86]–[88]) focus on binary code analysis, e.g., to support multiple hardware architectures like x86 and ARM. In particular, SAFE [88] leverages GRU RNN [89] and learns function embedding automatically where each assembly instruction is considered as a word and each sequence of instructions as a sentence. To find similar functions from binary code,

INNEREYE [86] leverages LSTM and considers assembly instruction with its operands as a single word and the basic block as a sentence. Xu et al. [87] and Asm2vec [28] apply neural networks to translate binaries for code similarity detection. Although in a different context, those works show the potential of applying neural machine translation to security, and have inspired us for our work.

In summary, though inspired by those existing works, *CiT* differs from them due to its special focus on NFV, event-based approach, and use of neural machine translation for inconsistency detection.

## 7 Conclusion

In this paper, we proposed an event-based, Neural Machine Translation (NMT)-powered detection approach, namely, *CiT*, for cross-level inconsistency attacks in NFV. Specifically, we leveraged the Long Short-term Memory (LSTM) model to translate the event sequences between different levels of an NFV stack. We applied both similarity metric and Siamese neural network to compare the translated event sequences with the actual sequences to detect inconsistency attacks. As a proof of concept, we have integrated *CiT* into OpenStack/Tacker and conducted extensive experiments using both real and synthetic data to demonstrate the efficiency, accuracy, and robustness of our solution. The main limitations of our work are as follows. First, since *CiT* relies on the cloud provider for collecting event sequences, a malicious provider could potentially mislead the detection mechanism by tampering with the input data. How to ensure the integrity of such data (e.g., through trusted computing techniques) is a future research direction. Second, we have not considered malicious attacks on *CiT* using adversarial machine learning techniques which could potentially aid an attacker to evade detection or disrupt retraining by introducing anomalous training data. Addressing this issue in the particular context of NFV is a potential future direction. Finally, *CiT* currently relies on the access to events at different levels, and anonymizing such events to avoid privacy concerns while still allowing the translation and detection is an interesting challenge.

## Acknowledgments

We thank the reviewers for their valuable comments. This work was supported by the Natural Sciences and Engineering Research Council of Canada and Ericsson Canada under the Industrial Research Chair in SDN/NFV Security and the Canada Foundation for Innovation under JELF Project 38599.

## References

[1] Intel, "Realising the benefits of network functions virtualisation in telecoms networks," 2014, available at: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/benefits-network-functions-virtualization-telecoms-paper.pdf, last accessed on: July 24, 2020.

[2] F. Z. Yousaf, M. Bredel, S. Schaller, and F. Schneider, "NFV and SDN—key technology enablers for 5G networks," *IEEE Journal on Selected Areas in Communications*, vol. 35, no. 11, pp. 2468–2478, 2017.

[3] Ovum, "NFV underpins service providers' ability to deliver reliable connectivity and new services post-pandemic," 2021, available at: https://omdia.tech.informa.com/OM013977/NFV-underpins-service-providers-ability-to-deliver-reliable-connectivity-and-new-services-post-pandemic, last accessed on: May 6, 2021.

[4] G. M. Insights, "Network function virtualization (NFV) market size," 2020, available at: https://www.gminsights.com/industry-analysis/network-function-virtualization-nfv-market, last accessed on: July 23, 2020.

[5] M. Bursell, A. Dutta, H. Lu, M. Odini, K. Roemer, K. Sood, M. Wong, and P. Wörndle, "Network functions virtualisation (NFV), NFV security, security and trust guidance, v1.1. 1," in *Technical Report, GS NFV-SEC 003*. European Telecommunications Standards Institute, 2014.

[6] OpenStack, "Stack update failed: port still in use," 2015, available at: https://bugs.launchpad.net/heat/+bug/1517355, last accessed on: July 23, 2020.

[7] ——, "User is not allowed to create port with fixed IP on shared network via RBAC," 2019, available at: https://bugs.launchpad.net/neutron/+bug/1833455, last accessed on: July 23, 2020.

[8] ——, "Cinder fails to create volume with gateway time-out error under high load," 2016, available at: https://bugs.launchpad.net/mos/+bug/1653164, last accessed on: July 23, 2020.

[9] ——, "Rule:shared is not respected in port/subnet create," 2018, available at: https://bugs.launchpad.net/neutron/+bug/1808112, last accessed on: July 23, 2020.

[10] ETSI, "Network functions virtualisation (NFV) release 3; Management and orchestration; Architecture enhancement for security management specification," 2018.

[11] E. U. A. for Cybersecurity (ENISA), "NFV SECURITY IN 5G," 2022, available at: https://www.enisa.europa.eu/publications/nfv-security-in-5g-challenges-and-best-practices/, last accessed on: May 12, 2023.

[12] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, and A. Meddahi, "NFV security survey: From use case driven threat analysis to State-of-the-art countermeasures," *IEEE Communications Surveys & Tutorials*, vol. 20, no. 4, pp. 3330–3368, 2018.

[13] L. T. Sudershan, M. Zhang, A. Oqaily, G. S. Chawla, L. Wang, M. Pourzandi, and M. Debbabi, "Modeling NFV deployment to identify the cross-level inconsistency vulnerabilities," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2019, pp. 167–174.

[14] M.-K. Shin, Y. Choi, H. H. Kwak, S. Pack, M. Kang, and J.-Y. Choi, "Verification for NFV-enabled network services," in *2015 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2015, pp. 810–815.

[15] X. Zhang, Q. Li, J. Wu, and J. Yang, "Generic and agile service function chain verification on cloud," in *2017 IEEE/ACM 25th International Symposium on Quality of Service*, 2017, pp. 1–10.

[16] Y. Wang, Z. Li, G. Xie, and K. Salamatian, "Enabling automatic composition and verification of service function chain," in *2017 IEEE/ACM 25th International Symposium on Quality of Service*, 2017, pp. 1–5.

[17] M. Flittner, J. M. Scheuermann, and R. Bauer, "ChainGuard: Controller-independent verification of service function chaining in cloud computing," in *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2017, pp. 1–7.

[18] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J.-M. Kang, "Sfc-checker: Checking the correct forwarding behavior of service function chaining," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks*, 2016, pp. 134–140.

[19] M. Zoure, T. Ahmed, and L. Réveillère, "Verines: runtime verification of outsourced network services orchestration," in *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, 2021, pp. 1138–1146.

[20] S. K. Fayazbakhsh, M. K. Reiter, and V. Sekar, "Verifiable network function outsourcing: requirements, challenges, and roadmap," in *Proceedings of the workshop on Hot topics in middleboxes and network function virtualization*, 2013, pp. 25–30.

[21] Y. Zhang, W. Wu, S. Banerjee, J.-M. Kang, and M. A. Sanchez, "SLA-verifier: Stateful and quantitative verification for service chaining," in *INFOCOM*, 2017.

[22] G. Gardikis, K. Tzoulas, K. Tripolitis, A. Bartzas, S. Costicoglou, A. Lioy, B. Gaston, C. Fernandez, C. Davila, A. Litke *et al.*, "SHIELD: A novel NFV-based cybersecurity framework," in *2017 IEEE Conference on Network Softwarization*, 2017, pp. 1–6.

[23] Tacker, "OpenStack Tacker," available at: https://wiki.openstack.org/wiki/Tacker, last accessed on: July 24, 2020.
[24] T. Madi, Y. Jarraya, A. Alimohammadifar, S. Majumdar, Y. Wang, M. Pourzandi, L. Wang, and M. Debbabi, "ISOTOP: auditing virtual networks isolation across cloud layers in OpenStack," *ACM Transactions on Privacy and Security*, vol. 22, no. 1, pp. 1–35, 2018.
[25] T. Madi, S. Majumdar, Y. Wang, Y. Jarraya, M. Pourzandi, and L. Wang, "Auditing security compliance of the virtualized infrastructure in the cloud: Application to OpenStack," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016, pp. 195–206.
[26] OpenStack, "OpenStack Congress," 2015, available at: https://wiki.openstack.org/wiki/Congress, last accessed on: July 24, 2020.
[27] S. Bleikertz, C. Vogel, T. Groß, and S. Mödersheim, "Proactive security analysis of changes in virtualized infrastructures," in *Proceedings of the 31st annual computer security applications conference*, 2015, pp. 51–60.
[28] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 472–489.
[29] B. J. Radford, L. M. Apolonio, A. J. Trias, and J. A. Simpson, "Network traffic anomaly detection using recurrent neural networks," *arXiv preprint arXiv:1803.10769*, 2018.
[30] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and lstm," *Multimedia Tools and Applications*, vol. 78, no. 4, pp. 3979–3999, 2019.
[31] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[32] R. Dey and F. M. Salemt, "Gate-variants of gated recurrent unit (gru) neural networks," in *2017 IEEE 60th international midwest symposium on circuits and systems (MWSCAS)*. IEEE, 2017, pp. 1597–1600.
[33] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.
[34] J. Mueller and A. Thyagarajan, "Siamese recurrent architectures for learning sentence similarity," in *thirtieth AAAI conference on artificial intelligence*, 2016.
[35] C. Sammut and G. I. Webb, Eds., *Encyclopedia of Machine Learning and Data Mining*. Springer, 2017.
[36] OpenStack, " Verizon launches industry-leading large OpenStack NFV deployment," 2016, available at: https://www.openstack.or /news/view/215/verizon-launchesindustry-leading-large-openstack-nfv-deployment, last accessed on: July 24, 2020.
[37] ETSI, "Network Functions Virtualisation (NFV); Architectural Framework," available at: https://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.02.01_60/gs_nfv002v010201p.pdf, last accessed on: May 29, 2023.
[38] M. Pattaranantakul, R. He, Q. Song, Z. Zhang, and A. Meddahi, "NFV security survey: From use case driven threat analysis to state-of-the-art countermeasures," *IEEE Communication Surveys Tutorials*, vol. 20, no. 4, pp. 3330–3368, 2018.
[39] National Institute of Standards and Technology, "CVE-2023-2088 Detail," available at: https://nvd.nist.gov/vuln/detail/CVE-2023-2088, last accessed on: May 29, 2023.
[40] ——, "CVE-2022-47951 Detail," available at: https://nvd.nist.gov/vuln/detail/CVE-2022-47951, last accessed on: May 12, 2023.
[41] ——, "CVE-2022-47950 Detail," available at: https://nvd.nist.gov/vuln/detail/CVE-2022-47950, last accessed on: May 12, 2023.
[42] ——, "CVE-2021-40797 Detail," available at: https://nvd.nist.gov/vuln/detail/CVE-2021-40797, last accessed on: May 12, 2023.
[43] A. T. Markettos, C. Rothwell, B. F. Gutstein, A. Pearce, P. G. Neumann, S. W. Moore, and R. N. Watson, "Thunderclap: Exploring vulnerabilities in operating system IOMMU protection via DMA from untrustworthy peripherals," *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.
[44] OpenStack, "Openstack Operations Guide," available at: https://docs.openstack.org/operations-guide/common/glossary.html, last accessed on: May 29, 2023.
[45] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, 2014, pp. 3104–3112.
[46] P. Malhotra, A. Ramakrishnan, G. Anand, L. Vig, P. Agarwal, and G. Shroff, "LSTM-based encoder-decoder for multi-sensor anomaly detection," *arXiv preprint arXiv:1607.00148*, 2016.
[47] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *International conference on database theory*, 2001, pp. 420–434.
[48] M. Creutz, T. Hirsimäki, M. Kurimo, A. Puurula, J. Pylkkönen, V. Siivola, M. Varjokallio, E. Arisoy, M. Saraçlar, and A. Stolcke, "Morph-based speech recognition and modeling of out-of-vocabulary words across languages," *ACM Transactions on Speech and Language Processing (TSLP)*, vol. 5, no. 1, pp. 1–29, 2007.
[49] S. M. Lakew, M. Cettolo, and M. Federico, "A comparison of transformer and recurrent neural networks on multilingual neural machine translation," in *Proceedings of the 27th International Conference on Computational Linguistics*. Santa Fe, New Mexico, USA: Association for Computational Linguistics, Aug. 2018, pp. 641–652, https://www.aclweb.org/anthology/C18-1054.
[50] N. Reimers and I. Gurevych, "Optimal hyperparameters for deep LSTM-networks for sequence labeling tasks," *arXiv preprint arXiv:1707.06799*, 2017.
[51] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
[52] R. Rehurek and P. Sojka, "Gensim—statistical semantics in Python," *Retrieved from genism. org*, 2011.
[53] Keras, "Keras: The Python Deep Learning library," https://keras.io/.
[54] scikitlearn, "scikit-learn: Machine Learning in Python," available at: https://scikit-learn.org/stable/, last accessed on: July 24, 2020.
[55] pandas, "pandas - Python Data Analysis Library," available at: https://pandas.pydata.org/, last accessed on: July 24, 2020.
[56] OpenStack, "OpenStack," available at: https://www.openstack.org/, last accessed on: July 24, 2020.
[57] Z. Xing, J. Pei, and E. Keogh, "A brief survey on sequence classification," *ACM Sigkdd Explorations Newsletter*, vol. 12, no. 1, pp. 40–48, 2010.
[58] J. Ramos *et al.*, "Using TF-IDF to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242. New Jersey, USA, 2003, pp. 133–142.
[59] W.-t. Yih, K. Toutanova, J. C. Platt, and C. Meek, "Learning discriminative projections for text similarity measures," in *Proceedings of the fifteenth conference on computational natural language learning*, 2011, pp. 247–256.
[60] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 1285–1298.
[61] B. Lindemann, B. Maschler, N. Sahlab, and M. Weyrich, "A survey on anomaly detection for technical systems using lstm networks," *Computers in Industry*, vol. 131, p. 103498, 2021.
[62] M. Mohammadi, T. A. Rashid, S. H. T. Karim, A. H. M. Aldalwie, Q. T. Tho, M. Bidaki, A. M. Rahmani, and M. Hosseinzadeh, "A comprehensive survey and taxonomy of the svm-based intrusion detection systems," *Journal of Network and Computer Applications*, vol. 178, p. 102983, 2021.
[63] N. An, A. Duff, G. Naik, M. Faloutsos, S. Weber, and S. Mancoridis, "Behavioral anomaly detection of malware on home routers," in *2017 12th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2017, pp. 47–54.
[64] A. I. Elkhawas and N. Abdelbaki, "Malware detection using opcode trigram sequence with svm," in *2018 26th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*. IEEE, 2018, pp. 1–6.
[65] D. K. Hoang, D. L. Vu *et al.*, "Iot malware classification based on system calls," in *2020 RIVF International Conference on Computing and Communication Technologies (RIVF)*. IEEE, 2020, pp. 1–6.
[66] X. Liao, C. Wang, and W. Chen, "Anomaly detection of system call sequence based on dynamic features and relaxed-svm," *Security and Communication Networks*, vol. 2022, 2022.
[67] M. Shobana and S. Poonkuzhali, "A novel approach to detect iot malware by system calls using deep learning techniques," in *2020 International Conference on Innovative Trends in Information Technology (ICITIIT)*. IEEE, 2020, pp. 1–5.
[68] R. Caruana, S. Lawrence, and C. L. Giles, "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping," in *Advances in neural information processing systems*, 2001, pp. 402–408.
[69] A. Oqaily, S. L. T, Y. Jarraya, S. Majumdar, M. Zhang, M. Pourzandi, L. Wang, and M. Debbabi, "Nfvguard: Verifying the security of multilevel network functions virtualization (NFV)

stack," in *12th IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2020, Bangkok, Thailand, December 14-17, 2020*. IEEE, 2020, pp. 33–40.
[70] K. Chen, Z. Zhang, J. Long, and H. Zhang, "Turning from TF-IDF to TF-IGM for term weighting in text classification," *Expert Systems with Applications*, vol. 66, pp. 245–260, 2016.
[71] B. Li and L. Han, "Distance weighted cosine similarity measure for text classification," in *International Conference on Intelligent Data Engineering and Automated Learning*. Springer, 2013, pp. 611–618.
[72] Microsoft, "Azure AD activity logs in Azure Monitor ," 2023, available at: https://learn.microsoft.com/en-us/azure/active-directory/reports-monitoring/concept-activity-logs-azure-monitor, last accessed on: May 14, 2023.
[73] OpenStack, "OpenStack Overview," 2023, available at: https://docs.openstack.org/nova/queens/install/overview.html#controller, last accessed on: June 14, 2023.
[74] Falco, "Detect security threats in real time," 2023, available at: https://falco.org/, last accessed on: Aug 18, 2023.
[75] ——, "Detect security threats in real timeFalco vs. AuditD from the HIDS perspective," 2021, available at: https://sysdig.com/blog/falco-vs-auditd-hids/, last accessed on: Aug 18, 2023.
[76] Suricata, "Suricata - Observe. Protect. Adapt." 2023, available at: https://suricata.io/, last accessed on: Aug 18, 2023.
[77] Mirantis, "IDPS Suricata deployment as a VNF on OpenStack with OpenContrail," 2023, available at: https://suricata.io/https://docs.mirantis.com/mcp/q4-18/mcp-security-best-practices/use-cases/idps-vnf.html, last accessed on: Aug 18, 2023.
[78] OpenStack, "2023.1 Series Release Notes," 2023, available at: https://docs.openstack.org/releasenotes/tacker/2023.1.html, last accessed on: Aug 05, 2023.
[79] ——, "OpenStack Releases," 2023, available at: https://releases.openstack.org/, last accessed on: Aug 05, 2023.
[80] OSM, "Open source MANO," available at: https://osm.etsi.org/, last accessed on: July 24, 2020.
[81] OPNFV, "OPNFV," available at: https://www.opnfv.org/, last accessed on: July 24, 2020.
[82] G. Marchetto, R. Sisto, J. Yusupov, and A. Ksentini, "Virtual network embedding with formal reachability assurance," in *CNSM*, 2018.
[83] Y. Shen, E. Mariconti, P. A. Vervier, and G. Stringhini, "Tiresias: Predicting security events through deep learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 592–605.
[84] J. Jurgovsky, M. Granitzer, K. Ziegler, S. Calabretto, P.-E. Portier, L. He-Guelton, and O. Caelen, "Sequence classification for credit-card fraud detection," *Expert Systems with Applications*, vol. 100, pp. 234–245, 2018.
[85] A. Brown, A. Tuor, B. Hutchinson, and N. Nichols, "Recurrent neural network attention mechanisms for interpretable system log anomaly detection," in *Proceedings of the First Workshop on Machine Learning for Computing Systems*, 2018, pp. 1–8.
[86] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Network and Distributed Systems Security (NDSS) Symposium*, 2019.
[87] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *CCS*, 2017, pp. 363–376.
[88] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2019, pp. 309–329.
[89] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

**Sudershan Lakshmanan** Sudershan Lakshmanan is currently working as a Security Engineer at University of Basel, Switzerland. He received his M.A.Sc in Information Systems Security from Concordia University, Montréal, Quebec, Canada. He obtained his M.Sc. in Software Engineering from Coimbatore Institute of Technology, Coimbatore, India. He then worked as a Software/Network Security Engineer at Zoho Corporation Pvt. Ltd., India for 3 years. His research interests include Cloud Computing Security and Privacy, Network Security and Software Security.

**Mengyuan Zhang** received the PhD degree in information and systems engineering from Concordia University, Montreal, Canada. She is a research assistant professor with the Department of Computing in the Hong Kong Polytechnic University, Hong Kong. Previously, she worked as an experienced researcher at Ericsson Research, Montreal, QC. Canada. Her research interests include security metrics, attack surface, cloud computing security, and applied machine learning in security. She has published several research papers and book chapters on the aforementioned topics in peer-reviewed international journals and conferences such as the IEEE Transactions on Information Forensics and Security, IEEE Transactions on Dependable and Secure Computing, CCS, and ESORICS.

**Suryadipta Majumdar** Suryadipta Majumdar is currently an Assistant Professor in Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, Canada. Previously, Suryadipta was an Assistant Professor in the Information Security and Digital Forensics department at University at Albany – SUNY, USA. He received his Ph.D. on cloud security auditing from Concordia University. His research mainly focuses on cloud security, Software Defined Network (SDN) security and Internet of Things (IoT) security.

**Yosr Jarraya** Yosr Jarraya is a Master Researcher at Ericsson since 2016 focusing on security and privacy in cloud, SDN and NFV. She received a Ph.D. in electrical and computer engineering from Concordia University Montreal, Canada, in 2010. She has several patents granted or pending. She co-authored two books and over 40 research papers in peer-reviewed international journals and conferences such as TOPS, TIFS, TDSC, JCS, NDSS, and ESORICS.

**Makan Pourzandi** Makan Pourzandi received the M.Sc. degree in parallel computing from École Normale Supérieure de Lyon, France, and the Ph.D. degree in computer science from the University of Lyon I, France. He is currently a Researcher with Ericsson, Canada. He has over 15 years of experience in security for telecom systems, cloud computing, distributed systems security, and software security. He is the inventor of over 28 patents granted or pending. He has published over 50 research papers in peer-reviewed scientific journals and conferences.

**Lingyu Wang** Lingyu Wang is a professor in the Concordia Institute for Information Systems Engineering (CIISE) at Concordia University, Montreal, Quebec, Canada. He holds the NSERC/Ericsson Industrial Research Chair (IRC) in SDN/NFV Security. He received his Ph.D. degree in Information Technology in 2006 from George Mason University. His research interests include SDN/NFV security, cloud computing security, network security metrics, software security, and privacy. He has co-authored seven books, two patents, and over 100 refereed conference and journal articles including many published at top journals/conferences, such as TOPS, TIFS, TDSC, TMC, JCS, S&P, CCS, NDSS, ESORICS, PETS, ICDT, etc. He is serving as an associate editor for IEEE Transactions on Dependable and Secure Computing (TDSC) and Annals of Telecommunications (ANTE) and an assistant editor for Computers & Security, and he has served as the program (co)-chair of seven international conferences and the technical program committee member of over 150 international conferences.