MLFM: *M*achine *L*earning Meets *F*ormal *M*ethod for Faster Identification of Security Breaches in Network Functions Virtualization (NFV)

Alaa Oqaily[1], Yosr Jarraya[2], Lingyu Wang[1], Makan Pourzandi[2], and Suryadipta Majumdar[1]

[1] CIISE, Concordia University, Montreal, QC, Canada
{a_oqaily, wang, smajumdar}@encs.concordia.ca
[2] Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
{yosr.jarraya, makan.pourzandi}@ericsson.com

Abstract. By virtualizing proprietary physical devices, Network Functions Virtualization (NFV) enables agile and cost-effective deployment of network services on top of an existing cloud infrastructure. However, the added complexity also increases the chance of misconfigurations that could leave the services or infrastructure vulnerable to security threats. To that end, formal method-based security verification is a standard solution for providing rigorous mathematical proofs that the configurations satisfy the desired security properties, or the counterexamples (i.e., misconfigurations). Nonetheless, a major challenge is that the sheer scale of large NFV environments can render formal security verification so costly that the significant delays before misconfigurations can be identified may leave a wide attack window. In this paper, we propose a novel approach, *MLFM*, that combines the efficiency of Machine Learning (ML) and the rigor of Formal Methods (FM) for fast and provable identification of misconfigurations violating security properties in NFV. Our key idea lies in an iterative teacher-learner interaction in which the teacher (FM) can gradually (over several iterations) provide more representative verification results as training data, while the learner (ML) can leverage such data to gradually obtain more accurate ML models. As a result, a small portion of the configuration data will be enough to obtain a relatively accurate ML model, which can then be applied to the remaining data to prioritize the verification of data that are more likely to cause violations. We experimentally evaluate MLFM against existing verification tools to demonstrate its benefits.

1 Introduction

By decoupling network functions from proprietary hardware devices, Network Functions Virtualization (NFV) allows network services to be implemented as software modules running on top of generic hardware or virtual machines. This new paradigm allows service operators to more easily deploy a multi-tenant NFV environment on top of an existing cloud infrastructure, and it also allows NFV tenants to accelerate the provisioning and deployment of their services. Due to such benefits, the popularity of NFV is on the rise, e.g., in the context of 5G and beyond, NFV has become one of the main technology enablers for operators to scale their network capabilities on-demand at a lower cost by virtualizing dedicated physical devices on top of existing clouds [2].

The benefits of NFV may come at the cost of increased complexity. To support the management and orchestration of multiple network slices belonging to different tenants on top of the same cloud infrastructure [11], NFV relies on a mixture of virtualization technologies, e.g., a Virtual Network Function (VNF) such as virtual firewall seen at tenant-level may correspond to several virtual machines (VMs) connected through Software-Defined Networking (SDN) at the cloud infrastructure level [2]. Such increased complexity may also increase the chance of incorrect (e.g., lack of sufficient network isolation between different tenants' network slices [28]) or inconsistent (e.g,. a virtual firewall VNF specified at the tenant level may be bypassed at the underlying cloud infrastructure level [30]) configurations that could leave the services or infrastructure vulnerable to security threats. Therefore, the timely identification of such misconfigurations is important to ensure the security of NFV environments.

To that end, formal method-based security verification solutions (e.g., [26, 27, 31, 39, 44, 54, 59]) can provide rigorous proofs about the compliance or violation (with counterexamples) of the configurations w.r.t. given security properties. However, a key challenge is that the sheer scale of virtual environments can render formal security verification too costly. For instance, a state-of-the-art security verification tool requires around 12 minutes to check whether a guest VM can access any SDN controller with merely 5,000 reachability queries [31]. Such a delay can become much more significant under large NFV environments, resulting in a wide attack window during which the services or infrastructure are left vulnerable. Moreover, the inherent complexity of formal methods [52] can leave little room for further performance improvement, e.g., the aforementioned tool [31] is already heavily optimized (new combined filter-project operator and symbolic packet representation are added to the back-end verifier).

Motivating example. We further illustrate this issue through an example. The left side of Figure 1 shows the simplified view of a large NFV environment where two tenants, Alice and Bob, host their Virtual Network Functions (VNFs). Suppose our goal is to verify network isolation, i.e., whether any of Alice's VNFs can reach any of Bob's (except what is explicitly allowed). Even the verification of such a simple property (all-pair reachability) can become expensive as NFV tenants may own a large number of VNFs. To make things worse, NFV and its underlying cloud infrastructure typically employ distributed and fine-grained network access control mechanisms (e.g., per-VM security groups in OpenStack [43]). Consequently, verifying the reachability of two VNFs/VMs may require inspecting many rules and configuration data scattered among various data sources (e.g., routing and NAT rules in virtual routers along the route, host routes of the subnets, and firewall rules implementing tenant security properties [59]).

The right side of Figure 1 contrasts how the collected audit data will be processed under an existing formal method (FM)-based security verification approach (top) and under our approach (bottom). The barchart-like pattern illustrates the distribution of data records in the audit data where red (or black) bars represent pairs of VNFs that violate (or satisfy) the network isolation property. As the upper pattern shows, a FM-based approach would verify the audit data as is, i.e., all the VNF pairs will be verified in the same order as given in the audit data. In contrast, our approach leverages ML to reorder those data records such that those that (likely) cause violations (the red bars) will be moved forward, i.e., given a higher priority for verification than others (the

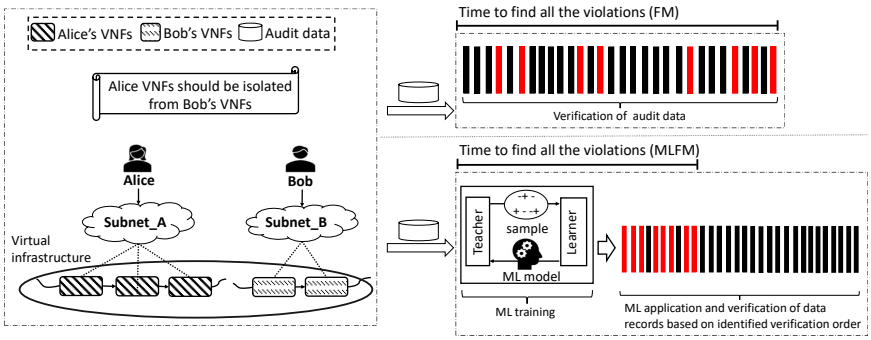


Fig. 1: Motivating example

black bars). Consequently, the verification can identify most of the violations in much less time (even after taking into account the time taken by ML training).

To that end, our main idea is to employ an iterative teacher-learner interaction, as depicted in the middle of Figure 1. In each iteration, the teacher (FM) first selects representative data records from the audit data, and then provides their verification results as training data to the learner (ML). Using such data, the learner (ML) trains an ML model, which is then given back to the teacher (FM) to be tested for identifying more representative data records (e.g., false positives and false negatives) in the next iteration. Over several iterations, such an interaction between the teacher and learner will enable a relatively accurate ML model to be trained using only a small portion of the audit data. The ML model can then be applied to reorder the remaining data for faster identification of violations. More specifically, our main contributions are as follows.

- To be best of our knowledge, *MLFM* is the first approach that combines FM with ML to have the best of both worlds (i.e., the rigor of FM which is essential for proving security compliance, and the efficiency of ML which is critical for a large NFV environment) for prioritizing verification tasks in NFV. Although we focus on NFV, we believe such an approach can potentially find other applications.
- To realize MLFM, we design an iterative teacher-learner interaction methodology with a detailed algorithm. We implement the methodology based on a constraint satisfaction problem solver, namely, Sugar [55], several popular ML algorithms (decision tree, random forest, support vector machine, and XGBoost), and sampling techniques (uncertainty sampling and query-by-committee) borrowed from the active learning literature [50] for identifying representative data records.
- We experimentally evaluate MLFM for two different use cases (one aims at the shortest verification time, and the other at the completeness of the result). The experimental results demonstrate the benefits of MLFM through identifying violations significantly faster than the baseline FM method (e.g., identifying 80% of violations in 28% of time), and further improving the efficiency of a state-of-the-art security verification tool [31] (e.g., identifying 80% of violations in 57% of time).

The remainder of the paper is organized as follows. Section 2 provides the background and threat model. Section 3 details the MLFM methodology. Section 4 describes our implementation. Section 5 presents the experiments. Section 6 reviews the related work. Finally, Section 7 discusses limitations and concludes the paper.

2 Preliminaries

This section provides essential background on NFV, discusses NFV security properties, and defines our threat model.

NFV Background. NFV is a network architecture concept that decouples network functions (e.g., routers, firewalls, and load balancers) from proprietary hardware devices and virtualizes them as Virtual Network Functions (VNFs) running on top of existing cloud infrastructures [2]. Figure 2 presents a simplified view of the ETSI NFV reference architecture [2] (left), and an example NFV deployment corresponding to our motivating example (right). First, the *resource management* level conceptualizes the virtual resources such as subnets and VNFs. Second, the underlying *virtual infrastructure* level implements those virtual resources using virtual networking elements, such as virtual switches (e.g., *OVS_1*), VLANs (for communications within the same server), VxLANs (for communications between servers), and network ports, running on top of physical servers (e.g., *Server_1*). In this paper, NFV configuration data stored in relational databases will be our main inputs.

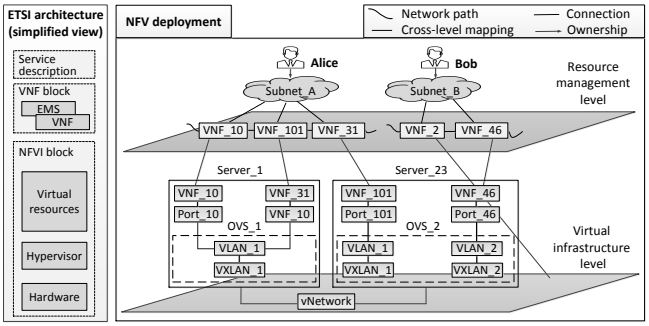


Fig. 2: ETSI NFV reference architecture [2] (left) and an example NFV deployment corresponding to the motivating example (right)

NFV Security Properties. Various security properties can be defined to verify the compliance of NFV environments w.r.t. standards (e.g., ETSI [2] and IETF-RFC7498 [45]) or NFV tenants' requirements. Table 1 (in Appendix) shows some example NFV security properties which we have previously identified [44]. Our approach can support other security properties as long as they can be verified using the chosen formal method tool (e.g., Sugar [55] used in this paper can handle most properties formulated using standard first-order logic). To make our discussions more concrete, we describe two example properties (which will be needed later).

Example 1. First, the property *mapping unicity VLANs-VXLANs* ensures the logic segregation between different tenants' virtual networks through the unique assignment of VxLAN (communications between servers) identifier to each VLAN (communications within one server). Figure 3 (left) depicts a violation of this property (the shaded nodes show *VLAN_1* is mapped to both *VXLAN_10* and *VXLAN_16* on *Server_1*). Note this property can be verified for each VLAN separately. Second, the property *no VNFs coresidence* prevents a tenant's VNFs to be placed on the same physical server with VNFs

of non-trusted tenants (e.g., due to concerns over potential side channel threats). Figure 3 (right) shows a violation of this property where *Alice's VNF_101* and *Bob's VNF_46* on both placed on server *S_23*. In contrast to the previous property, verifying this property could involve more records (all the VNFs of this tenant and the non-trusted tenants).

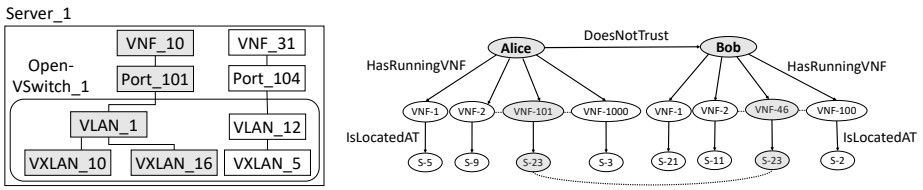


Fig. 3: Two example NFV security properties: *Mapping unicity VLANs-VXLANs* (left) and *No VNFs co-residence* (right) (shaded nodes indicate violations)

Threat Model and Assumptions. Similar to most existing security verification approaches, our *scope* is limited to attacks that (directly or indirectly) cause violations to given security properties, and we assume our solution is deployed by the owner of the NFV environment who has access to the logs, databases, and configuration data needed for the security verification (and the integrity of those input data is protected with trusted computing techniques (e.g., [49])). Under such assumptions, our *in-scope threats* include both external attackers who exploit existing vulnerabilities in the NFV environment to violate the security properties, and insiders such as NFV operators and tenants who cause misconfigurations violating the properties, either through mistakes or by malicious intentions. Conversely, *out-of-scope threats* include attacks that do not cause any violation of the security properties, and attacks launched by adversaries who can erase evidences of their attacks by tampering with the logs, databases, etc.

We assume that the formal specification of security properties as well as the formal verification approach itself are correct and sound. As a security verification solution, our approach can only identify the violation of given security properties, but is not designed to attribute such a violation to the underlying vulnerabilities (responsibility of vulnerability analysis) or specific attacks (responsibility of intrusion detection). Similar to most existing machine learning approaches, we assume that a dataset required for verifying given security properties has been collected. However, we do not require labeled data, which can be difficult to obtain in a real world NFV environment, as the data records will be labeled by the teacher (formal method) in our approach (optionally, a small amount of labeled data records would be helpful for training an initial ML model to speed up the iterative approach). As with most security applications (e.g., spam or intrusion detection), we assume the dataset is unbalanced (i.e., the majority of data records belong to the compliance class w.r.t. the security property), and we make additional efforts in designing our approach to address this issue.

3 Methodology

This section first presents an overview of our approach, followed by details on the iterative teacher (FM)-learner (ML) interaction and the MLFM algorithm.

3.1 Overview

We propose a machine learning-guided formal security verification approach, namely, *MLFM*, for fast and provable identification of data records that violate a given security property in NFV. First, the *ML training* stage employs an iterative teacher (FM)-learner (ML) interaction to train an ML model using only a small portion of the audit data. Second, the *ML application* stage applies the ML model to reorder the remaining audit data, such that those that are more likely to violate the property will be verified first. More specifically, Figure 4 depicts our approach as follows.

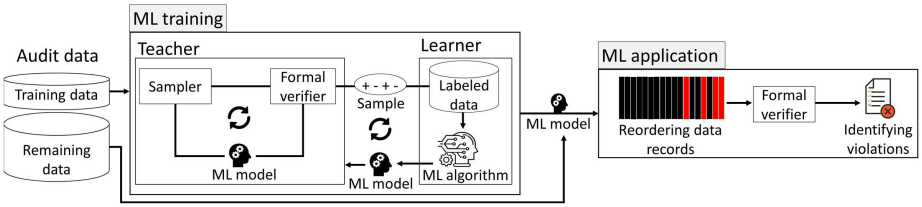


Fig. 4: Overview of the MLFM approach

The ML Training Stage. As Figure 4 (left) shows, in each iteration of the teacher-learner interaction, the teacher first applies a sampling method to select a small data sample of fixed size from the audit data (shown as *Sampler* in the figure) after applying the ML model received from the learner in the previous iteration (an initial ML model is provided for the first iteration). The teacher then verifies the data records inside this data sample, and labels each record based on its verification result (shown as *Formal verifier* in the figure), and sends the labeled data sample to the learner. The learner then combines this newly received data sample with the previously received data samples to train a new ML model to be sent back to the teacher. This iterative interaction ends when reaching a predefined condition, e.g., a fixed iteration count, or lack of significant change in the accuracy of the model between two consecutive iterations.

The ML Application Stage. As Figure 4 (right) shows, the final ML model from the *ML training* stage is applied to the remaining audit data (i.e., the data not used for training) in order to identify data records that are more likely to violate the given security property, namely, the "to be verified" subset, which will be given a higher priority for verification. On the other hand, the "not to be verified" subset will either be verified afterwards, or not verified at all, depending on the use cases (detailed in Section 3.3).

3.2 Iterative Teacher (FM)-Learner (ML) Interaction

In the following, we provide more details about the key methodology of our approach, i.e., the iterative teacher (FM)-learner (ML) interaction.

Sampling (Teacher). The sampler component of the teacher is designed to select representative data records from the audit data in order for the learner to effectively enhance the ML model over each iteration. Choosing the right data records is important because they could cause either increase or decrease in the accuracy of the next ML model, e.g., data records having the same (redundant) information or those with the same label may cause the model to either not improve, or become biased towards the majority data,

respectively. Our approach borrows sampling strategies (such as uncertainty sampling) from the active learning literature [50]. Although active learning has a different focus (it aims to reduce the effort of human experts in labeling the data, whereas no human expert is involved in our case), its sampling strategies are applicable to our approach, because they are also designed to better represent the characteristics of the property being analyzed such that an ML model can be trained with minimal labeled data.

Example 2. The left side of Figure 5 shows an excerpt of the audit data corresponding to the previous Example 1. Using uncertainty sampling, the sampler (inside the teacher block) selects a sample of size ($m = 2$) as the (shaded) record pairs $(1, 3)$ and $(6, 4)$.

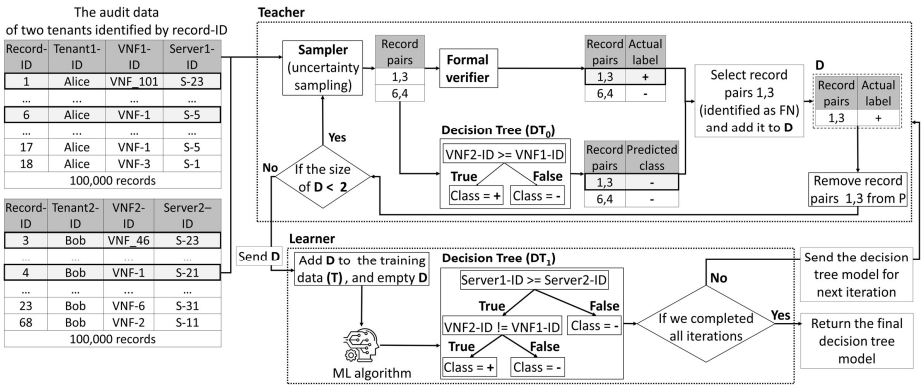


Fig. 5: An example of the iterative teacher (FM)-learner (ML) interaction

Verification (Teacher). The formal verifier component is responsible for labeling the selected sample of data records (which will later be sent to the learner as training data). Labeling here means to annotate the data records with an extra field representing their classes, i.e., whether they are compliant with, or violate, the security property. To obtain such labels, the formal verifier performs formal verification by instantiating the security property (e.g., formulated using first-order logic) with the data records.

Example 3. Following Example 2, Figure 5 shows how the formal verifier labels the selected sample by verifying the *No VNFs co-residence* property (see Section 2). Specifically, the formal verifier finds that the pair $(1, 3)$ violates the property (i.e., Alice's VNF (*VNF_101*) co-resides with Bob's VNF (*VNF_46*) on the same server (*S-23*)), and thus labels it as "+". The other pair $(6, 4)$ is labled as "-", as it does not violate the property.

Records Selection (Teacher). Next, the teacher applies the ML model from the previous iteration (received from the learner) to the labeled sample of data records. Intuitively, this allows the teacher to validate this previous ML model (by comparing its results to the labels provided by the formal verifier) and provide the "mistakes" (false positives and false negatives) as more representative training data to the learner. Specifically, as the ML model from the previous iteration also classifies the data records into two classes, by comparing its results to the ground truth, i.e., the labels assigned by the formal verifier component, the teacher can identify those records that have been correctly classified (i.e., true positives (TPs)) and those incorrectly classified (i.e., false

negatives (FNs) and false positives (FPs)). Then, the teacher adds the TP, FN, and FP records to a new dataset D, which is the training dataset to be sent to the learner. Finally, if the number of records in D is still less than the desired size of the sample (m), the teacher repeats the aforementioned steps as an inner-iteration until it has accumulated totally m records in D. Note that the rationale for selecting (TP, FP, FN) records is twofold. First, as the positive class (i.e., violations) is generally smaller due to data imbalance, adding TP and FN records can augment the positive class to reduce the bias in training [41]. Second, the FN and FP records are incorrectly classified by the previous ML model and thus may contain more useful information for the learner to improve the accuracy of its next model.

Example 4. Following Example 3, Figure 5 shows a decision tree model (DT_0) received from the last iteration is applied to the two pairs of records $(1, 3)$ and $(6, 4)$. The decision tree (DT_0) predicts "+", if the *VNF2-ID* value is no smaller than the *VNF1-ID* value; otherwise, it is predicted as "-". Therefore, both $(1, 3)$ and $(6, 4)$ are predicted as "-". Comparing such results to the labels previously assigned by the formal verifier (see Example 3), we can see the pair $(1, 3)$ is *FN* and should be added to the dataset D (and deleted from the audit data), whereas $(6, 4)$ is *TN* and should not be added. Finally, as the size of *D* is less than the required size (m=2), we will repeat the inner-iteration.

ML Model Building (Learner). Once the teacher's dataset D reaches the required size m, the sample it contains is sent to the learner (D is then emptied in preparation for the next iteration). The learner adds the received sample to its existing training data (i.e., the collection of all previous samples), and utilizes this newly enriched training data to build a new ML model. The ML model is sent back to the teacher if the stopping condition (e.g., the specified number of interactions) has not been reached; otherwise, the interaction ends and the final ML model is given to the next (ML application) stage.

Example 5. Following Example 4, the lower part of Figure 5 shows that, once the teacher's inner-iteration ends, a sample of size two is sent to the learner. The learner adds the received sample to the existing training data (T) while the teacher empties its dataset (D). The new training data (T) is then used to build a new decision tree model (DT_1), which is more accurate than DT_0.

3.3 MLFM Algorithm and Use Cases

Algorithm 1 more formally states our approach. The inputs to the algorithm include the unlabeled audit data, the security property, and the parameters. The initial set of training data allows a system user to influence the algorithm with his/her domain knowledge by manually selecting/labeling data records (otherwise, the data can simply be randomly selected from the audit data and labeled using the formal verifier).

The algorithm has an outer iteration (Lines 2-9) which first builds a new sample through performing the inner iteration (Lines 3-7), and then adds this new sample to the existing training data (Line 8) to train a new ML model (Line 9). The outer iteration is repeated for a fixed number (provided as an input parameter) of times. The final ML model is then applied to reorder the remaining audit data before verifying it (Line 10). The union of all the verification results (Lines 5 and 10) is the final output.

The inner iteration builds a sample D of size m as follows. First, it selects a sample of size m from the audit data by following a given sampling strategy (Line 4). Although

not shown in the algorithm, depending on the sampling strategy being used, this step may involve other parameters such as the current ML model (e.g., with uncertainty sampling [50]) or the training data (e.g., with Query-By-Committee (QBC) sampling [50]). Second, the sample is verified and labeled (with the verification results) using a formal verifier (Line 5). Third, the current ML model is applied to the sample, and the results are compared to the labels (verification results) to identify and add the (TP, FP, FN) records to D (Line 6). Fourth, D is removed from the audit data to avoid being selected again (Line 7). We repeat the above steps until D contains at least m records.

Algorithm 1: The MLFM algorithm

```
1  Inputs: Audit data (AD), security property (SP), initial training data (T_0), initial model
   M_0 = TrainClassifier(T_0), per-iteration sample size (m), and iteration count (n)
   /* Outer-iteration                                                              */
2  for i = 0, i < n, i++ do
       /* Inner-iteration                                                          */
3      while | D |< m do
4          S = SelectSample(AD, m)
5          S_i = VerifyAndLabel(S, SP)
6          D = D ∪ TP(S_i, M_i) ∪ FP(S_i, M_i) ∪ FN(S_i, M_i)
7          AD = AD \ D
8      T_{i+1} = T_i ∪ D; D = φ
9      M_{i+1} = TrainClassifier(T_{i+1})
10 return Verify(Reorder(AD, M_n)) ∪ (⋃_i S_i)
```

Complexity Analysis. The worst case complexity of the MLFM algorithm is $O(n \cdot (m \cdot (T_s + T_{v_1}) + T_t) + T_{v_2})$ where T_s, T_{v_1}, T_t, and T_{v_2} are the time for sampling (Line 4), verifying m records (Line 5), training (Line 9), and verifying remaining records (Line 10), respectively. Such times would depend on specific algorithms, e.g., T_s under uncertainty sampling [50] can be estimated as $O(\mid AD \mid)$, since this strategy requires applying the current ML model on the audit data AD. T_{v1} and T_{v2} under a CSP solver is known to be exponential in the number of variables of the instantiated security property [14]. Finally, T_t under a decision tree classifier is $O(n_a \cdot n_t \cdot \log_2(n_t))$ [47] where n_a is the number of attributes and n_t the size of training data (i.e., $O(n \cdot m)$). We will further study the efficiency of the algorithm through experiments in Section 5.

Use Cases. Depending on how the remaining data is verified in Line 10 of the MLFM algorithm, our approach can be applied for two different use cases. First, MLFM running in the *partial verification* case will stop after verifying all the "to be verified" records (which would appear first after the reordering). This can be useful when the system user wants to find violations as quickly as possible (but not necessarily to find all the violations), and our objective in the training is to find an ML model that is the most accurate (since the mis-classifed violations would not be verified, as further explained in Section 5). Second, MLFM in the *priority-based verification* case will verify all the records (with the "to be verified" records verified first). Our objective of the training is to find an ML model that incurs the least overall verification time with acceptable accuracy (since the mis-classified records will still be verified eventually).

4 Implementation

In this section, we describe the architecture and details of our implementation.

System Architecture. Our implementation of MLFM (shown in Figure 11 in Appendix due to space limitation) interacts with an OpenStack/Tacker [8]-based NFV environment to collect audit data. The system also interacts with a user to obtain other inputs, such as the security property to be verified, the formal verifier and the ML model to be applied, and the system parameters (the number of iterations and the sample size, as detailed in Section 3.3). Finally, the system returns an audit report to the user.

Data Collection and Processing. We implement this module using Python and Bash scripts to collect audit data from multiple sources including logs and configuration databases or files. For instance, to verify the *No VNFs co-residence* property, the module collects the identifiers of VNFs from Tacker and Nova databases [7], their corresponding owners (from Nova database), and the identifiers of servers hosting those VNFs (from Nova database). As the audit data are usually scattered among different components of the NFV environment and stored in different formats, the data must first be pre-processed. For instance, to verify the *mapping unicity VLANs-VXLANs* property, the data collected from OpenFlow tables of the OVS databases has unnecessary fields (e.g., *cookie* and *priority*) that must be filtered out. Also, the *port* and *vlan_vid* fields must be correlated to create the relation tuples *IsAssignedVLAN(ovs,port,valn)* for the verification. Finally, such filtered and correlated data must be converted into the corresponding input formats required by the formal verifier as well as for the ML training.

MLFM Manager. We implement this module in Python to manage and coordinate the interactions between other system modules for performing data collection and processing, data sampling, formal verification, ML training, etc., as described in Section 3.

ML Model Learner. We utilize Python 3.6.9 and Scikit-learn 0.24.1 (an open source ML library written in Python) to implement this module. We select decision tree, Support Vector Machine (SVM), and Random Forest (RF) models as they are among the most commonly used supervised classifiers, and are computationally more efficient compared to other classifiers such as K-Nearest-Neighbor (KNN) [40]. We also select XGBoost classifier [15], a scalable tree boosting system with a simpler structure using less resources than most other ML models, which has recently seen wide application for its high accuracy and low false positive rate [16, 42]. As our main aim is to reduce the overall delay before violations can be identified, we do not consider deep learning models as they are well known for higher complexity and longer training time compared to traditional ML models [34].

Sampler. We employ the *modAL* framework [17] to implement sampling strategies in this module. The *modAL* is an active learning framework for Python3, built on top of Scikit-learn [29], which allows to rapidly create active learning workflows with flexibility [17]. We select the uncertainty sampling and query-by-committee (with DT, SVM, and RF for members of the committee) sampling strategies in our implementation, as those are the most computationally efficient ones compared to other strategies [50].

Formal Verifier. We formalize the security properties together with the audit data as a Constraint Satisfaction Problem (CSP), a time-proven technique for expressing complex problems. Using CSP allows the user to specify a wide range of security properties

(due to its expressiveness) in a relatively simple manner (as CSP enables to uniformly present the audit data as well as the security properties, and in a comprehensible and clean formalism, such as first order logic (FOL) [12]). Moreover, there exist many powerful and efficient CSP solver algorithms to avoid the state space traversal [48], which can make our approach more scalable for large NFV environments.

Once formulated as a CSP problem, the security verification is performed using Sugar [55], a well-established SAT-based constraint solver. We choose Sugar as it is an award-winning solver of global constraint categories (at the International CSP Solver Competitions in 2008 and 2009 [9]). Sugar solves a finite linear CSP by translating it into a SAT problem using order encoding method, and then solving the translated SAT problem using the MiniSat solver [18], which is an efficient CDCL SAT solver particularly effective in narrowing the search space [23]. Adapting our MLFM framework to other verification methods (such as theorem proving, model checkers, temporal logic, and Datalog) based on the needs of verification tasks is regarded as a future work.

Example 6. The predicate that corresponds to the negation of the *No VNFs co-residence* property is formulated (by the system user, done only once) as Formula 1 (left), and a predicate instance returned by Sugar to indicate violation is shown as Formula 2 (right) (i.e., both *Alice* and *Bob* have VNFs co-residing on the same server *S_23*).

$$\forall \texttt{t1}, \texttt{t2} \in \texttt{Tenant}, \forall \texttt{vnf1}, \texttt{vnf2} \in \texttt{VNF}, \forall \texttt{s1}, \texttt{s2} \in \texttt{Server} : \texttt{HasRunningVNF}(\texttt{t1}, \texttt{vnf1}) \wedge \texttt{HasRunningVNF}(\texttt{t1}, \texttt{vnf1}) \wedge \texttt{DoesNotTrust}(\texttt{t1}, \texttt{t2}) \wedge \texttt{IsRunningOn}(\texttt{vnf1}, \texttt{s1}) \wedge \texttt{IsRunningOn}(\texttt{vnf2}, \texttt{s2}) \wedge (\texttt{s1} == \texttt{s2}) \quad (1)$$

$$\texttt{HasRunningVNF}(\texttt{Alice}, \texttt{VNF\_101}) \wedge \texttt{HasRunningVNF}(\texttt{Bob}, \texttt{VNF\_46}) \wedge \texttt{DoesNotTrust}(\texttt{Alice}, \texttt{Bob}) \wedge \texttt{IsRunningOn}(\texttt{VNF\_101}, \texttt{S\_23}) \wedge \texttt{IsRunningOn}(\texttt{VNF\_46}, \texttt{S\_23}) \wedge (\texttt{S\_23} == \texttt{S\_23}) \quad (2)$$

5 EXPERIMENTS

This section describes the datasets and experimental settings, and presents our results.

5.1 Datasets and Experimental Settings

We first describe the implementation of our NFV testbed and data generation using the testbed, and then detail the experimental settings.

NFV Testbed Implementation. We choose to build our NFV testbed using OpenStack [7] with Tacker [8] mainly due to their growing popularity in real world [10] (other options such as Open Baton [4], OPNFV [5], and OSM [6] are still at their development stages). More specifically, we rely on the latest version OpenStack Rocky [7] for managing the virtual infrastructure, and we employ Tacker-0.10.0 [8], an official OpenStack project, to deploy virtual network services. Our NFV testbed consists of 20 tenants and 200 VNF forwarding graphs (VNFFGs), with each tenant owning around 10 VNFFGs and each VNFFG consisting of about 10 VNFs.

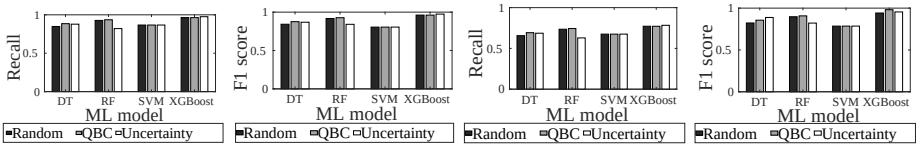
NFV Data Generation. To evaluate the performance of MLFM under large scale NFV environments, we would require a large scale NFV deployment. However, to the best of our knowledge, there do not exist any publicly available large-scale NFV deployment datasets. Therefore, we develop Python scripts to automatically generate various VNF Descriptors (VNFDs) and VNFFG Descriptors (VNFFGDs), which are then uploaded

(also called on-boarding) to our NFV testbed to deploy different network services and generate large scale NFV datasets. We randomize parameters of those descriptors to ensure diversity in the generated data (e.g., the number of network ports per VNF, the flavor of each VNF, the number of VNFs in each Network Function Path (NFP), and the number of NFPs in each VNFFG). Our first dataset, *DS1*, contains 12,500 audit data records for verifying the *mapping unicity VLANs-VXLANs* property (P1 henceforth), and our second dataset, *DS2*, contains 25,000 records for verifying the *no VNFs co-residence* property (P2 henceforth). Each dataset contains around 10% of (uniformly distributed) records that violate the corresponding property.

Experimental Setting. All experiments are performed on a SuperServer 6029P-WTR running the Ubuntu 18.04 operating system equipped with Intel(R) Xeon(R) Bronze 3104 CPU @ 1.70GHz and 128GB of RAM without GPUs. All the experiments are performed using Sugar [55] as the formal verifier (unless mentioned otherwise) and Python 3.6.9 with Scikit-learn 0.24.1 ML packages for the ML method. For all the experiments, we use the default parameters for the ML models. Each experiment is repeated 1,000 times to obtain the average results.

5.2 Experimental Results

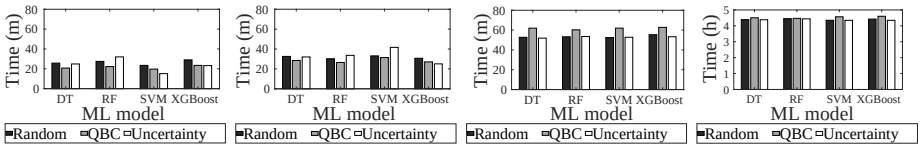
Best Performing Combination of ML Model/Sampling Method. The first set of the experiments aims to find the best performing combination of ML model and sampling method (as components of MLFM), from both the accuracy and time performance point of views. Specifically, Figure 6 shows the recall and F1 score results for different combinations of ML models (DT, RF, SVM and XGBoost, trained on 20% of each dataset) and sampling methods (random sampling, query-by-committee (QBC), and uncertainty sampling) for both security properties (P1 and P2) and datasets (DS1 and DS2). The results in Figures 6 (a) and (b) show that the combination of XGBoost and uncertainty sampling allows MLFM to achieve the highest recall (0.97) and F1 score (0.97) for security property P1. On the other hand, SVM combined with any of these sampling methods has the lowest F1 score (0.80) (i.e., less effective in identifying both classes), and RF with uncertainty sampling has the lowest recall (0.82) (i.e., less effective in identifying the violations). Similarly, Figure 6 (c) shows that XGBoost with uncertainty sampling also has the best recall (0.783) for security property P2. However, as Figure 6 (d) shows, XGBoost has the best F1 score (0.981) when paired with QBC sampling. Nonetheless, as identifying the violations is more important to MLFM, XGBoost with uncertainty sampling is considered the best option for both P1 and P2.



(a) Recall for P1 (b) F1 score for P1 (c) Recall for P2 (d) F1 score for P2

Fig. 6: Recall and F1 score for combinations of ML models and sampling methods, trained on 20% of dataset DS1 for property P1 (a and b) and on DS2 for P2 (c and d)

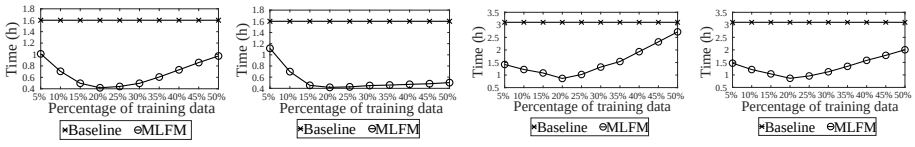
Figure 7 shows how the combinations of ML models and sampling methods affect the running time (in minutes) of MLFM (including both the ML training and application stages). As explained in Section 3.3, the partial verification use case aims to find the majority of violations in the least time. To that end, Figure 7 (a) seems to suggest that SVM paired with uncertainty sampling is the best option as it requires the least time (15.14 minutes). However, upon further investigation, this is not really the case, because the lower time consumption is mainly due to its inaccuracy (it misses more violations and thus, similar to most SAT solvers, Sugar incurs less time when there are less violations to find [55]). Therefore, considering both the accuracy (Figure 6 (a)) and the running time, XGBoost with uncertainty sampling seems to be the best option (with the second least time) for partial verification under P1. Figure 7 (b) shows that XGBoost with uncertainty sampling is the best option for priority-based verification for P1, as it requires the least time (accuracy is less important in this use case as all the records will be verified eventually, as explained in Section 3.3). Similarly, Figures 7 (c) and (d) show XGBoost with uncertainty is also the best combination under P2 for both use cases.



(a) Partial verification time for P1 (b) Priority-based verification time for P1 (c) Partial verification time for P2 (d) Priority-based verification time for P2

Fig. 7: Running time of MLFM for combinations of ML models and sampling methods, with 20% of training data under P1 (a) (b), or P2 (c) (d), for both use cases

Best Performing Parameters m and n. In this set of experiments, we aim to find the optimal parameters of MLFM, i.e., the number of iterations n and the sample size m (see Section 3.3), in terms of the running time for priority-based verification, and also in comparison to the baseline approach (i.e., directly applying the formal verifier to the entire dataset). Specifically, Figure 8 (a) shows how changing the sample size m with a fixed number of iterations ($n = 10$) impacts the time, with the best performing model (i.e., XGBoost with uncertainty sampling) under property P1. The results show that MLFM takes less time ($<$1 hour) than the baseline approach (around 1.6 hours) in all cases. As more training data is used (through larger samples), the time of MLFM initially decreases due to more accurate ML models, and it reaches the lowest value (0.417 hr, or around 25% of the time of baseline) while using about 20% of the dataset for training. The time starts to increase afterwards, since the time needed to verify larger samples in the training stage becomes dominant (compared to the time saved in the application stage). Figure 8 (b) shows how changing the number of iterations n with a fixed sample size ($m = 250$) impacts the time. Similarly, MLFM takes less time than the baseline approach in all cases. The optimal percentage of training data is also around 20% (where $n = 10$). However, afterwards the time of MLFM stays lower than in the previous experiment, which shows that increasing the number of iterations is a safer choice (than increasing sample size) for increasing the training data. Figures 8 (c) and (d) show similar trends for property P2 (the longer time is due to more records involved in verification, as shown in Section 2).



(a) (b) (c) (d)

Fig. 8: Running time of MLFM vs. the baseline (FM only) under property P1 (a) and (b) or P2 (c) and (d), using different percentages of training data either by changing the sample size m (a) and (c) or by changing the number of iterations n (b) and (d)

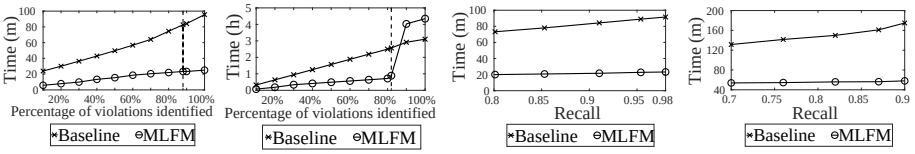
Comparing MLFM to Other Approaches. In this set of experiments, we compare the performance of MLFM to both the baseline approach (i.e., directly applying the formal verifier, Sugar [55]) and a state-of-the-art security verification tool, NOD [31] [3]. All experiments use the best performing model and parameters (i.e., XGBoost with uncertainty sampling, 20% training data, $m = 250$, and $n = 10$).

First, Figures 9 (a) and (b) show the time (in minutes) needed by the baseline approach (upper curve) and by MLFM (lower curve) for identifying different percentages of violations under properties P1 (a) and P2 (b), respectively. The figures depict both the priority-based verification use case (the entire curve) and the partial verification use case (part of the curve before the dashed line). Specifically, Figure 9 (a) shows that MLFM outperforms the baseline throughout the percentages, e.g., for partial verification, MLFM can identify 88% of the violations in around 23.3 minutes, which takes the baseline 82.7 minutes. Similarly, Figure 9 (b) shows that MLFM outperforms the baseline in case of partial verification for property P2, where it identifies 82% of the violations in about 53.3 minutes, while the baseline takes almost 2.4 hours. However, in case of priority-based verification (after the 82%) for property P2, MLFM takes more time than the baseline. The reason lies in the difference between the two properties. As explained in Section 2, unlike P1 (which can be verified for each VLAN independently), P2 may involve all the VNFs of a tenant, which means the remaining 18% of violations can only be identified using the baseline approach. Fortunately, there exists an alternative solution, i.e., we run MLFM and the baseline in parallel, and terminate MLFM as soon as the baseline finishes (as we already have all the results). As Figure 9 (b) shows, this would allow MLFM to identify around 86% of violations faster than the baseline, while bounding the overall running time by what is taken by the baseline.

Next, Figures 9 (c) and (d) show the tradeoff between the running time (in minutes) and the recall values of partial verification (i.e., the percentage of violations identified by the end of partial verification) for P1 (c) and P2 (d). Both figures show similar results, i.e., while the baseline naturally requires more time for identifying more violations, MLFM can achieve a high recall value of 0.98 (P1) and 0.9 (P2) (by increasing the percentage of training data from 10% to 20%) with negligible change in running time (the difference will be greater for verifying the remaining records, as shown in Figure 8).

Finally, Figures 10 (a) and (b) show the time (in minutes) needed by NOD [31] (lower curve) and MLFM integrated with NOD (upper curve) for identifying different percentages of violations under the *virtual network reachability* property [31] (as this

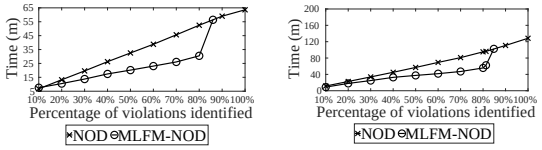
[3] Among existing security verification tools, we do not compare to NFVGuard [44] as it actually forms the basis of our verification component, and we do not compare to TenantGuard [59] as it is based on custom algorithms instead of formal method.



(a) (b) (c) (d)

Fig. 9: The time (in minutes) for identifying different percentages of violations by MLFM and the baseline for P1 (a) or P2 (b). The tradeoff between running time and recall values of MLFM and the baseline for partial verification of P1 (c) or P2 (d)

property is similar to P2, we run MLFM in parallel with NOD, as discussed above). We use the benchmarks provided in [31] to create two datasets with 25,000 and 50,000 reachability pairs, respectively, and around 10% of violations injected randomly. The results show that MLFM can help NOD to identify around 80% (a) and 81.3% (b) of violations, respectively, in less (57% and 65%, respectively) time.



(a) (b)

Fig. 10: The time (in minutes) for identifying different percentages of violations by NOD [31] and by MLFM integrated with NOD, using 25,000 (a) and 50,000 (b) records

6 RELATED WORK

Most existing solutions related to security verification for NFV (e.g., [20, 21, 39, 54, 56, 58, 60, 61]) focus on the verification of service function chaining (SFC). Those works employ either custom algorithms (such as [20, 58, 60]), graph-based methods (such as [21, 56, 61]), or formal methods (such as [39, 54]). Unlike those existing works (which focus on the SFC only), our previous work, NFVGuard [44], aims to verify the entire NFV stack (including both SFC and underlying infrastructure, and their consistency) using formal method. However, the increased scope also leads to increased complexity and longer verification time, which has motivated us to propose MLFM.

Besides NFV, there also exist security verification solutions for other virtual infrastructures, such as cloud and SDN (e.g., [27, 31–33, 37, 38, 59]), including formal method-based ones [31–33, 37]. Unlike MLFM, most such solutions do not specifically address the delay in verification (so they may benefit from MLFM in that aspect), with the exception of NOD [31] which is optimized for large applications (our experiments in Section 5 show it can further benefit from MLFM). In contrast to formal method, custom algorithms (e.g., [27] and [59]) may enjoy improved efficiency for specific properties but they generally lack the level of expressiveness of formal method-based approaches (including MLFM). Also designed to reduce verification time, the proactive approach

(e.g., [35, 36]) performs the verification in advance based on predicted events, which is parallel to, and can be integrated with, our approach.

There exist works that combine machine learning and formal method in other contexts, such as automated program verification (for synthesizing invariants used to verify the correctness of a program, e.g., [19, 22, 46, 57]). In particular, Ezudheen et al. [19] develop learning-based algorithms for synthesizing invariants for programs that generate Horn-style proof constraints. Garg et al. [22] propose the ICE-learning framework for not only taking (counter-)examples but also handling implications. Ren et al. [46] propose a method based on selective samples to improve the efficiency of invariant synthesizing. Finally, Vizel et al. [57] study the relationship between SAT-based Model Checking (SAT-MC) and Machine Learning-based Invariant Synthesis (MLIS). Although the goals are very different (efficient verification vs. invariant synthesizing), our teacher-learner approach is similar to those existing works, with a key difference being that we additionally employ the sampling strategies from the active learning literature [50] to more effectively identify representative samples.

7 CONCLUSION

We have presented MLFM, a novel approach to security verification in NFV that could combine the rigor of formal methods with the efficiency of machine learning for faster identification of security violations. Specifically, we designed an iterative approach for the teacher (FM) to gradually provide more representative data samples, such that the learner (ML) could train an ML model using a small portion of the data; the ML model was then applied to the remaining data to prioritize the verification of likely violations. We implemented MLFM based on OpenStack/Tacker, and our experimental results showed significant performance improvements over baseline approaches.

Limitations and Future Work. First, we have limited our scope to NFV in this work, and a future direction is to apply MLFM to other large-scale virtual infrastructures (e.g., clouds and SDNs). Second, while MLFM only focuses on security verification, a natural next step is integrating MLFM with security enforcement mechanisms to turn faster violation identification into more responsive attack prevention. Third, MLFM is static in the sense that its verification is on-demand based on data snapshots, and one future direction is to support continuous security verification (monitoring) based on data streams. Finally, we will also investigate other ML-specific issues such as the possibility of using deep learning for offline pre-training (due to its complexity), and defence against potential adversarial attacks on the training process of MLFM.

Acknowledgements We thank the anonymous 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. Cloud Security Alliance, https://cloudsecurityalliance.org/research/ccm/. Last accessed 11 September 2021
2. ETSI: Network Functions Virtualisation Architectural Framework, https://www.etsi.org/. Last accessed 11 September 2021
3. Network Functions Virtualisation (NFV); NFV Security; Problem Statement https://www.etsi.org/. Last accessed 11 September 2021
4. Open Baton, https://openbaton.github.io/. Last accessed 11 September 2021
5. Open Platform for NFV, https://www.opnfv.org/. Last accessed 11 September 2021
6. Open Source MANO, https://osm.etsi.org/. Last accessed 11 September 2021
7. OpenStack, https://www.openstack.org/. Last accessed September 11, 2021
8. OpenStack Tacker, https://releases.openstack.org/teams/tacker.html. Last accessed September 11, 2021
9. Sugar: a SAT-based Constraint Solver, https://cspsat.gitlab.io/sugar/. Last accessed 8 November 2021
10. Verizon launches industry-leading large OpenStack NFV deployment, https://www.openstack.org/news/. Last accessed 11 September 2021
11. Barakabitze, A.A., Ahmad, A., Mijumbi, R., Hines, A.: 5G network slicing using SDN and NFV: A survey of taxonomy, architectures and future challenges. Computer Networks **167**, 106984 (2020)
12. Ben-Ari, M.: Mathematical logic for computer science. Springer Science & Business Media (2012)
13. Bursell, M., Dutta, A., Lu, H., Odini, M., Roemer, K., Sood, K., Wong, M., Wörndle, P.: Network Functions Virtualisation (NFV), NFV security, security and trust guidance, v. 1.1. 1. In: Technical Report, GS NFV-SEC 003. European Telecommunications Standards Institute (2014)
14. Buss, S., Nordström, J.: Proof complexity and sat solving. Handbook of Satisfiability **336**, 233–350 (2021)
15. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining. pp. 785–794 (2016)
16. Chen, Z., Jiang, F., Cheng, Y., Gu, X., Liu, W., Peng, J.: XGBoost classifier for DDoS attack detection and analysis in SDN-based cloud. In: IEEE international conference on big data and smart computing (BigComp). pp. 251–256 (2018)
17. Danka, T., Horvath, P.: modAL: A modular active learning framework for Python. arXiv preprint arXiv:1805.00979 (2018)
18. Eén, N., Sörensson, N.: An extensible sat-solver. In: International conference on theory and applications of satisfiability testing. pp. 502–518. Springer (2003)
19. Ezudheen, P., Neider, D., D'Souza, D., Garg, P., Madhusudan, P.: Horn-ice learning for synthesizing invariants and contracts. In: Proceedings of the ACM on Programming Languages **2**(OOPSLA), 1–25 (2018)
20. Fayazbakhsh, S.K., Reiter, M.K., Sekar, V.: Verifiable network function outsourcing: requirements, challenges, and roadmap. In: Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization. pp. 25–30 (2013)
21. Flittner, M., Scheuermann, J.M., Bauer, R.: Chainguard: Controller-independent verification of service function chaining in cloud computing. In: IEEE Conference on Network Function Virtualization and Software Defined Networks. pp. 1–7 (2017)
22. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Ice: A robust framework for learning invariants. In: International Conference on Computer Aided Verification. pp. 69–87. Springer (2014)

23. Gong, W., Zhou, X.: A survey of sat solver. In: Proceedings of AIP Conference. vol. 1836, p. 020059. AIP Publishing LLC (2017)
24. IEC ISO Std: ISO 27017. Information technology-Security techniques (DRAFT) (2012)
25. IETF, SFC: Internet Engineering Task, SFC Active WG Working Group Documents (2020), https://www.redhat.com/en/blog/2018-year-open-source-networking-csps
26. Jayaraman, K., Bjørner, N., Outhred, G., Kaufman, C.: Automated analysis and debugging of network connectivity policies. Microsoft Research pp. 1–11 (2014)
27. Kazemian, P., Chang, M., Zeng, H., Varghese, G., McKeown, N., Whyte, S.: Real time network policy checking using header space analysis. In: 10th {USENIX} Symposium on Networked Systems Design and Implementation (NSDI'13). pp. 99–111 (2013)
28. Kotulski, Z., Nowak, T.W., Sepczuk, M., Tunia, M., Artych, R., Bocianiak, K., Osko, T., Wary, J.P.: Towards constructive approach to end-to-end slice isolation in 5G networks. EURASIP Journal on Information Security **2018**(1), 1–23 (2018)
29. Kramer, O.: Scikit-learn. In: Machine learning for evolution strategies, pp. 45–53. Springer (2016)
30. Lakshmanan Thirunavukkarasu, S., Zhang, M., Oqaily, A., Singh Chawla, G., Wang, L., Pourzandi, M., Debbabi, M.: Modeling NFV deployment to identify the cross-level inconsistency vulnerabilities. IEEE CloudCom (2019)
31. Lopes, N.P., Bjørner, N., Godefroid, P., Jayaraman, K., Varghese, G.: Checking beliefs in dynamic networks. In: 12th {USENIX} Symposium on Networked Systems Design and Implementation (NSDI'15). pp. 499–512 (2015)
32. Madi, T., Jarraya, Y., Alimohammadifar, A., Majumdar, S., Wang, Y., Pourzandi, M., Wang, L., Debbabi, M.: ISOTOP: auditing virtual networks isolation across cloud layers in OpenStack. ACM Transactions on Privacy and Security (TOPS) **22**(1), 1–35 (2018)
33. Madi, T., Majumdar, S., Wang, Y., Jarraya, Y., Pourzandi, M., Wang, L.: 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. pp. 195–206 (2016)
34. Maji, P., Mullins, R.: On the reduction of computational complexity of deep convolutional neural networks. Entropy **20**(4), 305 (2018)
35. Majumdar, S., Jarraya, Y., Madi, T., Alimohammadifar, A., Pourzandi, M., Wang, L., Debbabi, M.: Proactive verification of security compliance for clouds through pre-computation: Application to OpenStack. In: European Symposium on Research in Computer Security. pp. 47–66. Springer (2016)
36. Majumdar, S., Jarraya, Y., Oqaily, M., Alimohammadifar, A., Pourzandi, M., Wang, L., Debbabi, M.: LeaPS: Learning-based proactive security auditing for clouds. In: European Symposium on Research in Computer Security. pp. 265–285. Springer (2017)
37. Majumdar, S., Madi, T., Wang, Y., Jarraya, Y., Pourzandi, M., Wang, L., Debbabi, M.: Security compliance auditing of identity and access management in the cloud: Application to OpenStack. In: IEEE 7th International Conference on Cloud Computing Technology and Science. pp. 58–65 (2015)
38. Majumdar, S., Madi, T., Wang, Y., Jarraya, Y., Pourzandi, M., Wang, L., Debbabi, M.: User-level runtime security auditing for the cloud. IEEE Transactions on Information Forensics and Security **13**(5), 1185–1199 (2017)
39. Marchetto, G., Sisto, R., Yusupov, J., Ksentini, A.: Virtual network embedding with formal reachability assurance. In: 14th International Conference on Network and Service Management. pp. 368–372 (2018)
40. Mohamed, A.E.: Comparative study of four supervised machine learning techniques for classification. Information Journal of applied science and technology **7**(2) (2017)
41. Monard, M.C., Batista, G.E.: Learmng with skewed class distrihutions. Advances in Logic, Artificial Intelligence, and Robotics: LAPTEC **85**(2002), 173 (2002)

42. Neutatz, F., Mahdavi, M., Abedjan, Z.: Ed2: A case for active learning in error detection. In: Proceedings of the 28th ACM International Conference on Information and Knowledge Management. pp. 2249–2252 (2019)
43. OpenStack Training Labs: OpenStack Training Labs, available at: https://wiki.openstack.org/wiki/Documentation/training-labs
44. Oqaily, A., Sudershan, L., Jarraya, Y., Majumdar, S., Zhang, M., Pourzandi, M., Wang, L., Debbabi, M.: NFVGuard: Verifying the Security of Multilevel Network Functions Virtualization (NFV) Stack. In: 2020 IEEE International Conference on Cloud Computing Technology and Science. pp. 33–40. IEEE (2020)
45. Quinn, P., Nadeau, T.: Rfc 7948, problem statement for service function chaining. Internet Engineering Task Force (IETF), ed (2015)
46. Ren, S., Zhang, X.: Synthesizing conjunctive and disjunctive linear invariants by K-means++ and SVM. Int. Arab J. Inf. Technol. **17**(6), 847–856 (2020)
47. Sani, H.M., Lei, C., Neagu, D.: Computational complexity analysis of decision tree algorithms. In: International Conference on Innovative Techniques and Applications of Artificial Intelligence. pp. 191–197. Springer (2018)
48. Sassi, I., Anter, S., Bekkhoucha, A.: A graph-based big data optimization approach using hidden markov model and constraint satisfaction problem. Journal of Big Data **8**(1), 1–29 (2021)
49. Schear, N., Cable II, P.T., Moyer, T.M., Richard, B., Rudd, R.: Bootstrapping and Maintaining Trust in the Cloud. In: Proceedings of the 32Nd Annual Conference on Computer Security Applications. pp. 65–77 (2016)
50. Settles, B.: Active learning literature survey (2009)
51. Shin, M.K., Choi, Y., Kwak, H.H., Pack, S., Kang, M., Choi, J.Y.: Verification for NFV-enabled network services. In: ICTC (2015)
52. Souri, A., Navimipour, N.J., Rahmani, A.M.: Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review. Computer Standards & Interfaces **58**, 1–22 (2018)
53. SP, NIST: 800-53. Recommended security controls for federal information systems pp. 800–53 (2003)
54. Spinoso, S., Virgilio, M., John, W., Manzalini, A., Marchetto, G., Sisto, R.: Formal verification of virtual network function graphs in an sp-devops context. In: European Conference on Service-Oriented and Cloud Computing. pp. 253–262. Springer (2015)
55. Tamura, N., Banbara, M.: Sugar: A CSP to SAT translator based on order encoding. Proceedings of the Second International CSP Solver Competition (2008)
56. Tschaen, B., Zhang, Y., Benson, T., Banerjee, S., Lee, J., Kang, J.M.: Sfc-checker: Checking the correct forwarding behavior of service function chaining. In: IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN). pp. 134–140 (2016)
57. Vizel, Y., Gurfinkel, A., Shoham, S., Malik, S.: IC3-flipping the E in ICE. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 521–538. Springer (2017)
58. Wang, Y., Li, Z., Xie, G., Salamatian, K.: Enabling automatic composition and verification of service function chain. In: IEEE/ACM 25th International Symposium on Quality of Service (IWQoS). pp. 1–5 (2017)
59. Wang, Y., Madi, T., Majumdar, S., Jarraya, Y., Alimohammadifar, A., Pourzandi, M., Wang, L., Debbabi, M.: TenantGuard: Scalable runtime verification of cloud-wide VM-level network isolation. In: The Network and Distributed System Security Symposium (2017)
60. Zhang, X., Li, Q., Wu, J., Yang, J.: Generic and agile service function chain verification on cloud. In: IEEE/ACM 25th International Symposium on Quality of Service. pp. 1–10 (2017)

61. Zhang, Y., Wu, W., Banerjee, S., Kang, J.M., Sanchez, M.A.: Sla-verifier: Stateful and quantitative verification for service chaining. In: IEEE INFOCOM 2017-IEEE Conference on Computer Communications. pp. 1–9 (2017)

Appendix

Table 1: Examples of NFV security properties [44]

Security Properties	Sub-Properties	Description	Standards
Physical resource isolation [32]	No VNFs co-residence	VNFs of a tenant should not be placed on the same server as VNFs of a non-trusted tenant	ISO [24], NIST800 [53], CCM [1], ETSI [13]
Virtual resource isolation [32]	No common ownership	Tenant-specific resources should belong to a unique tenant, unless permitted by a user-defined policy	CCM [1], ETSI [13], IETF-RFC7665, RFC-7498 [25]
Topology isolation [32]	Mapping unicity VLANs-VXLANs	VLANs and VXLANs should be mapped one-to-one on a given server	ISO [24], NIST800 [53], CCM [1], ETSI [13], IETF-RFC7665, RFC-7498 [25]
	Correct association Ports-Virtual Networks	VNFs should be attached to the virtual networks they are connected to through the right ports	
	Overlay tunnels isolation	In each VTEP end, VNFs are associated with their physical location (at L2) and to the VXLAN assigned to the networks they are attached to at L1	
	Mappings unicity Virtual Networks Segments	Virtual networks and segments should be mapped one-to-one	
	Mappings unicity Ports-VLANs	Ports should be mapped to unique VLANs	
	Mappings unicity Ports-Segments	vPorts should be mapped to unique segments	
Policy and state correctness [51]	-	A policy can be dynamically changing. The changed policy should be reconfigured in VNF node as soon as possible	ETSI [13, 3], IETF-RFC7665, RFC8459[25]
Functionality of VNF and VNFFGs [20, 60]	-	Check if VNFs and the composition (i.e., service chaining) of these functions work as intended	ETSI [3], IETF-RFC-7665, RFC8459 [25]
SFC ordering and sequencing as defined by the specification [21]	-	SFCs should maintain the order of VNFs with the correct traffic forwarding behavior as defined by the specifications	ETSI [13, 3], IETF-RFC7665, RFC8459 [25]
Topology consistency [32]	VNFFG configuration consistency between L1/L2	Consistency between the size of VNFFGs, the sequences of VNFs and the classifiers at L1 and their parallel implementation at L2	ISO [24], NIST800 [53], CCM [1], IETF-RFC-8459 [25], ETSI [13, 3]
	Virtual links consistency	VNFs should be connected to the VLANs and VXLANs in L2 that corresponds to the virtual networks they are connected to in L1	
	VNF location consistency	Consistency between VNFs locations at L2 and L1	
	CPs-Ports consistency	Consistency between CPs defined at L1 and their created counterparts; Ports in (L2)	

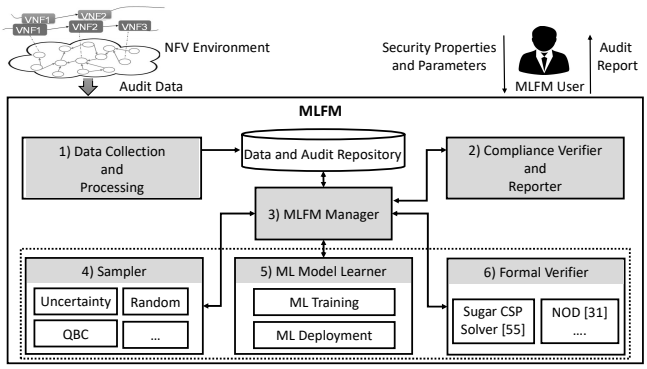


Fig. 11: The MLFM system architecture