Payload Analysis of Adversaries' Tooling: Automated Identification of Fuzzers

Aayush Garg[1], Constantinos Patsakis[2] (*Senior Member, IEEE*), Zanis Ali Khan[3], and Qiang Tang[4]

***Abstract*—API fuzzing, a technique widely used to uncover vulnerabilities in web applications, poses significant security risks when exploited maliciously, leading to service disruptions and data breaches. While firewalls can block unauthorized fuzzing attempts, they limit defenders' ability to gather data on attacker methodologies, reducing actionable cyber threat intelligence. Identifying the responsible fuzzers enables defenders to trace the attacker, uncover their motives, and assess the potential impact, which helps security teams prepare more effectively, mitigate attacks, and develop targeted countermeasures to enhance the security of web APIs. However, analyzing the payloads generated by fuzzers remains largely unexplored and presents significant challenges. For instance, fuzzers often generate similar payloads due to shared initial seeds and similar fuzzing strategies, making accurate fuzzer identification more complex. To analyze this, we experimented with four well-known API fuzzers; APIFuzzer, Kiterunner, RESTler, and Schemathesis, and created a comprehensive dataset of their payloads targeting five different web APIs. Our thorough analysis reveals that the overlapping payloads, i.e., the identical generated payloads across these fuzzers, can be substantially large. For instance, ≈17% of payloads generated with Schemathesis overlapped with ≈12% of the payloads generated with RESTler across different web APIs. As a result, defining distinctive payload features that machine learning models can learn to differentiate and identify their fuzzer accurately becomes more difficult. Alternatively, deep learning techniques, known for their ability to automatically extract features, present a compelling alternative. To evaluate this, we experimented with an architecture combining a bidirectional Transformers-based encoder-decoder and a machine learning classifier to classify fuzzers based on their payloads. Rigorous evaluation using k-fold cross-validation demonstrated high precision and recall, averaging 89%, showcasing this combinatorial architecture's robustness and effectiveness. Our findings demonstrate the potential of combining deep learning and machine learning for fuzzer identification and enhancing web API security.**

I. Introduction

The rapid digital transformation of modern organizations has significantly expanded their attack surfaces, exposing them to a wide range of cyber threats. Many organizations, particularly those for whom cybersecurity is not a core function, invest heavily in awareness, preparedness, and intelligence to mitigate these risks. Cyber threat intelligence plays a vital role by providing insights into potential threats, especially during the early stages of an attack. Of particular interest in this work is the identification of early signs of possible attacks, which can enable timely and effective defensive actions.

According to Lockheed Martin's Cyber Kill Chain [1] and MITRE's ATT&CK framework [2], the first step of an attack is reconnaissance, where adversaries observe their targets to identify entry points. This can involve passive techniques, such as gathering email addresses or DNS entries, or active techniques, such as scanning services (T1595 in MITRE) and identifying vulnerabilities (T1595.002). During this phase, defense mechanisms like firewalls aim to block malicious traffic. However, such measures often miss the opportunity to gather valuable intelligence about the attacker's tools and methodologies, leaving the organization less prepared for future attacks.

Bianco's pyramid of pain [3] suggests that higher-value intelligence, such as tooling identification, provides more actionable insights than simpler indicators like IP addresses or domain names. Identifying the tools used in an attack can reveal the attacker's capabilities, intentions, and modus operandi. More importantly, the targeted victim can also prepare itself in case of a successful attack, as for some threat actors, the modus operandi is well documented (e.g., see MITRE's APT groups[1]). For instance, while many adversaries use common tools like Metasploit or CobaltStrike [4], others, such as APT29, switch to less common options like Sliver and Brute Ratel [5], [6]. Tooling identification enables defenders to anticipate future attack patterns and tailor defensive strategies accordingly, even when traditional indicators of compromise are obscured.

In this vein, current approaches in the literature aim to fingerprint clients using available network information. For instance, JA4 and its variants create fingerprints from the network responses between the host and server, specifically leveraging the unencrypted ClientHello message in the TLS handshake. While this approach can be effective in identifying specific tools and malware, such as CobaltStrike and Qakbot, it is easily bypassed by modifying encryption algorithms or user agent strings. To overcome such limitations, we focus on traffic analysis of API fuzzers to distinguish them based on their payloads. As previously discussed, adversaries often rely on specific tooling during their attacks. By identifying these tools promptly, defenders can infer the attacker's motives, predict other tools that may be used, anticipate targeted endpoints, and assess the potential impact. This intelligence enhances preparedness, strengthens cybersecurity posture, and supports

[1]A. Garg , Z. A. Khan, and Q. Tang are with the Luxembourg Institute of Science and Technology (LIST) Email: aayush.garg@list.lu, zanis-ali.khan@list.lu, qiang.tang@list.lu

[2]C. Patsakis is with the Department of Informatics, University of Piraeus, Greece, and the Information Management Systems Institute of Athena Research Center, Greece. Email: kpatsak@unipi.gr

[1]https://attack.mitre.org/groups/

timely threat mitigation. Notably, our approach enables defenders to identify the attacker's tooling without needing access to the tool itself. For instance, detecting CobaltStrike typically requires observing a beacon launched within the organization's network, implying a perimeter breach. In contrast, our method operates during the reconnaissance phase, prior to an actual attack, without requiring access to the attacker's resources.

To assess the validity of this approach, we experimented with four well-known API fuzzers that an attacker can use to find vulnerabilities in a publicly exposed API. Specifically, we focus on *APIFuzzer*, *Kiterunner*, *RESTler*, and *Schemathesis*. We created a comprehensive dataset of payloads generated by these fuzzers while targeting five different RESTful APIs adhering to OpenAPI specifications. The task is not trivial; for instance, our analysis revealed that the number of overlapping payloads, i.e., the same generated payloads across these fuzzers, can be substantially large. For instance, $\approx$17% of payloads generated with Schemathesis overlapped with $\approx$12% of the payloads generated with RESTler across different web APIs. Thus, defining distinctive features of payloads that machine learning models can learn to accurately differentiate and identify their fuzzer origins is even more challenging. Alternatively, deep learning methods, renowned for their effectiveness in automatically extracting features from text [7], [8], offer a promising alternative. To evaluate this, we developed an architecture combining bidirectional Transformers-based encoder-decoder models with a machine learning classifier to distinguish between fuzzers based on their payloads. Rigorous evaluation using k-fold cross-validation yielded high precision and recall, averaging 89%, demonstrating the robustness and effectiveness of this combinatorial architecture for API fuzzer identification. These findings highlight the potential of combining deep learning and machine learning in advancing traffic analysis strategies for tooling identification and offer a promising direction for future research and development in cyber threat intelligence.

II. Background & Related work

A. Fuzzing

Fuzzing is currently an important vulnerability discovery technique. The primary objective of fuzzing is to create a variety of inputs to identify as many exceptions as possible. Prior to conducting fuzz testing, the input format is defined, and the target program is selected. The workflow consists of four key stages: i) generating test cases, ii) executing the target program, iii) monitoring for exceptions, and iv) managing those exceptions. Compared to other testing techniques, fuzzing is relatively straightforward to implement and boasts excellent extensibility and applicability. Additionally, since fuzzing is carried out in a real execution environment, it tends to achieve a high level of accuracy. Another benefit is its minimal reliance on prior knowledge of target applications, coupled with its ability to scale effortlessly for large-scale systems.

While fuzzing faces some challenges, such as lower efficiency and reduced code coverage, its advantages often overshadow these drawbacks. Consequently, fuzzing has become one of the most powerful and efficient techniques for uncovering vulnerabilities in modern systems. The test effectiveness is directly affected by the quality of the generated test cases. The inputs should align as closely as possible with the requirements of the programs being tested regarding input format. On the other hand, these inputs must be sufficiently malformed to increase the likelihood of causing the program to fail during processing.

Fuzzing can be categorized into two main types based on how inputs are generated: *mutation-based*, and *generation-based*. Generation-based fuzzing creates inputs from the ground up, relying on defined grammars [9]–[11] or a valid corpus [12]–[14]. In this approach, inputs are derived directly from a predetermined seed set. In contrast, mutation-based fuzzing takes existing inputs, known as seeds, and modifies them to produce new test cases [15]–[17]. This method involves executing various strategies, such as seed scheduling, byte scheduling, and mutation scheduling, to generate inputs from the seed set.

Recently, REST API fuzzing has become an effective method for detecting security flaws and vulnerabilities in cloud-based services. Traditional fuzzers [18], [19] often struggle with handling complex API dependencies. API fuzzing [20], [21] addresses these challenges by analyzing dependencies between different API requests and adapting to the responses dynamically. This approach enables more effective exploration of the API space, leading to the discovery of critical bugs that are harder to detect with simpler fuzzing methods. This technique is particularly valuable in testing modern, large-scale web services. The following section outlines the specific API fuzzing techniques used in this study.

B. API Fuzzing Techniques

API fuzzing techniques play a vital role in assessing the robustness of applications by testing their responses to malformed or unexpected inputs. In this study, we utilize four widely adopted API fuzzers, each chosen for its effective approach to payload generation and vulnerability detection:

a) APIFuzzer [22]: APIFuzzer is a powerful and versatile tool designed for fuzz testing web APIs. This open-source framework enables developers and security researchers to identify vulnerabilities in RESTful APIs by generating and sending a wide variety of inputs, including valid and malformed data. By employing a systematic approach to testing, APIFuzzer helps uncover security flaws that malicious actors could potentially exploit. The tool supports customizable configuration options, allowing users to specify endpoints, request methods, and input formats, making it adaptable to diverse testing environments. Furthermore, APIFuzzer's integration with existing test suites facilitates seamless incorporation into development workflows, promoting continuous security assessment throughout the software development lifecycle. With its user-friendly interface and comprehensive documen-

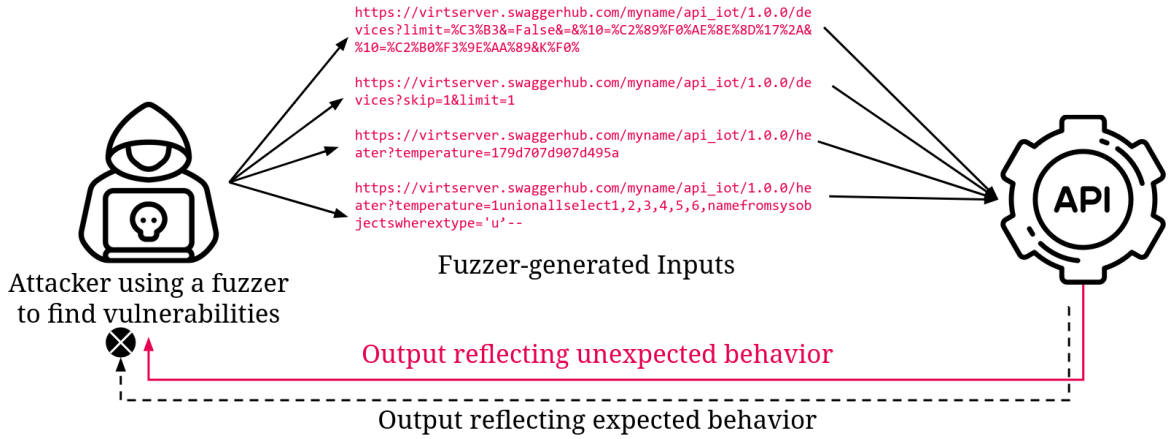

Fig. 1. Overview of the attack scenario: An active adversary sends authenticated requests to a RESTful API and modifies them, including sending malformed ones, to probe for vulnerabilities that could result in unexpected outputs. Potential attack scenarios include DoS, code injection, command execution, and leakage of sensitive information via the API response.

tation, APIFuzzer empowers users to enhance the security and robustness of their API implementations effectively. The tool is publicly available[2].

b) Kiterunner [23]: Kiterunner is a state-of-the-art fuzzer designed to enhance content discovery specifically for modern web applications, particularly APIs. Unlike traditional fuzzers that primarily focus on locating static files and folders, Kiterunner employs a sophisticated approach by leveraging a curated dataset of Swagger specifications to efficiently brute-force routes and endpoints. Modern application frameworks, such as Flask, Rails, Express, and Django, define routes requiring specific HTTP methods, headers, parameters, and values, often overlooked by conventional fuzzing tools. Conversely, Kiterunner utilizes a dataset compiled from multiple sources, including an internet-wide scan of the most common Swagger paths and contributions from GitHub via BigQuery, and APIs.guru [24] database. By condensing this information into its schema, Kiterunner constructs and sends requests that align with the expected configurations of each API endpoint, ensuring the use of the correct HTTP methods and relevant headers and parameters. This targeted approach enables Kiterunner to efficiently perform both traditional content discovery and API endpoint fuzzing, enhancing the accuracy and effectiveness of API testing and exploration. The tool is publicly available[3].

c) RESTler [20]: Restler is a fuzzing tool specifically designed for automated test generation for REST APIs, such as cloud services. It analyzes Swagger specification, which defines the structure of the API, including the types of requests it supports and their corresponding responses. Restler generates test cases by inferring dependencies between API requests. For instance, it tracks whether one request's output might be required as the input for the subsequent request (i.e., a resource ID returned from one request might be needed by the subsequent request). Additionally, Restler dynamically learns from the previous test executions, analyzing API responses to avoid invalid request sequences in future tests. This approach `(dual)` of dependency inference and dynamic feedback allows Restler to exploit a large space of possible request combinations, thus minimizing the invalid tests. In real-world systems like Gitlab, it has been successful in identifying unseen bugs, demonstrating its effectiveness as a tool for enhancing the robustness and security of cloud-based services. The tool is publicly available[4].

d) Schemathesis [21]: Schemathesis is designed for property-based testing of web APIs, described by OpenAPI and GraphSQL [25] schemas. It automatically generates fuzzing tests to identify both crashes and subtle semantic errors, such as violations of the API schema or data exposure. It utilizes the Hypothesis [26] testing library, which focuses on generating valid data based on API schemas and then systematically exploring variations, including edge cases. It transforms API schemas into structured tests that explore different combinations of inputs and also can be customized by the user to focus on specific aspects of the API. Moreover, Schemathesis uses *schema canonicalization* to efficiently generate valid and subtly valid inputs, optimizing error detection and reducing unnecessary test failures. It also incorporates techniques like shrinking, which simplifies failing test cases, and error deduplication, allowing for efficient bug identification. The tool is publicly available[5].

C. Cyber Threat Intelligence

Cyber Threat Intelligence (CTI) plays a pivotal role in cybersecurity, focusing on understanding adversarial tactics and tools to anticipate potential threats. Established frameworks such as Lockheed Martin's Cyber Kill Chain [1] and MITRE ATT&CK [2] categorize the tactics, techniques, and procedures (TTPs) used by adversaries, providing defenders with structured knowledge of each stage in an attack lifecycle [27].

[2]APIFuzzer. https://pypi.org/project/APIFuzzer/0.9.13
[3]Kiterunner. https://github.com/assetnote/kiterunner/releases/tag/v1.0.2
[4]RESTler. https://github.com/microsoft/restler-fuzzer
[5]Schemathesis. https://github.com/schemathesis/schemathesis

By cataloging TTPs and the tools used in different attack phases, CTI helps organizations detect and mitigate threats, highlighting the value of understanding adversarial tools and methodologies [28].

Research has shown that tool-specific attribution can enhance threat intelligence by linking observed attack patterns to particular threat groups [29]. Studies indicate that attackers often prefer specific tools based on their intended goals, such as data exfiltration, ransomware deployment, or denial-of-service attacks, and may manipulate artifacts not only to hide their traces but also to confuse and deceive [30]. In this sense, timely and accurately determining the tooling of the adversary becomes even more tempting for the defender. In this direction, the current state of the art and practice focus more on artifacts that come in the form of file system changes, domains, IPs, memory dumps, etc. Therefore, the defender blocks domains and IPs that have bad reputation, as collected from various threat intelligence feeds, or collects artifacts from the hosts to assess whether it has or is suffering an attack. On the other hand, network traffic analysis can also detect malicious traffic [31]–[33]. The closest work to ours is perhaps [34] in which the researchers use CNN and RNN to classify payloads and identify potential attacks. Likewise, Liu et al. [35] use a graph convolutional network to detect cross-site scripting attacks. The above suggests that tooling identification from payloads is a field that has not been adequately explored as the literature is mainly focused on attack detection.

III. Problem setting

In what follows, we consider an active adversary, referred to as *Malory*, who targets an internet-exposed RESTful API owned by her victim, denoted as *Valery*. *Malory* can send valid, authenticated requests to *Valery's* API and collect its responses and service status. Using these legitimate requests as a baseline, *Malory* attempts to manipulate the requests, introducing alterations, including malformed inputs, to uncover unexpected behaviors. Such behaviors can be exploited to launch a range of attacks, including, but not limited to, denial of service, code injection, command execution, and leakage of sensitive information through API responses, as illustrated in Figure 1

While the above scenario may appear demanding, it is based on realistic assumptions. The primary reason is that internet-exposed RESTful APIs of targeted victims are relatively easy for adversaries to discover through various methods. For instance, an adversary can reverse engineer publicly available applications, a technique particularly common with mobile apps, to uncover API endpoints and their associated parameters. Additionally, many organizations unintentionally expose API details through public-facing websites, where documentation or endpoint information can sometimes be located. Moreover, specialized cyber threat intelligence services and tools are available that help adversaries identify and analyze APIs linked to specific targets. Similarly, the assumption that the requests are authenticated is also practical. Attackers often register for legitimate access to the services provided by their target, especially in the context of mobile services, where registration processes are typically straightforward. Once registered, attackers can analyze the authenticated calls made by their devices during regular use of the service. These calls can serve as a reference for understanding API behavior and potential vulnerabilities. Additionally, API documentation, often provided through OpenAPI, further simplifies the process by offering detailed specifications of endpoints, methods, and expected inputs. Lastly, it is reasonable to assume that *Malory*, as a skilled adversary, will not perform these tasks manually. Instead, she is likely to rely on specialized tools, such as fuzzers, to automate the generation and submission of requests.

Problem Formalization. Given this setup, the problem can be formalized as follows: Let there be a set of adversaries $M_1, M_2, ..., M_k$, each using a unique fuzzer $F_1, F_2, ..., F_k$ to attack *Valery's* RESTful API. Each adversary submits requests through their corresponding fuzzer, and we assume that each fuzzer operates from a unique IP address. This allows the requests to be grouped by their originating IP address. As a result, each group of requests, denoted as $R_1, R_2, ..., R_k$, contains only the requests generated by a single fuzzer. Each group R_i consists of individual requests $r_1, r_2, ..., r_{n_i}$, i.e., $R_i = \{r_1, r_2, ..., r_{n_i}\}$.

Valery's API includes logging functionality, enabling her to record all unencrypted requests received by the API. This logging creates a dataset of grouped requests, $R_1, R_2, ..., R_k$. The problem is to determine, for any given request $r \in R_i$, the index i corresponding to the originating fuzzer. In essence, the goal is to classify the origin of each request based on its content, identifying which fuzzer generated it.

The above allows us to formulate our research questions as follows. First, to understand the extent to which different fuzzers generate similar payloads, which could impact our model's ability to distinguish between fuzzers, we ask:

RQ1 ***Payload Overlap Analysis:*** **How much overlap exists between the payloads generated by different API fuzzers?**

Second, to evaluate our model's performance in correctly identifying the origin fuzzer of each payload, despite any overlap between fuzzers, we ask:

RQ2 ***Fuzzer Classification Performance:*** **How well can a combined deep learning and machine learning architecture distinguish between payloads to identify their fuzzer origins despite overlapping payloads?**

IV. Approach

We aim to identify the origin, i.e., the fuzzer, responsible for generating the API-attacking payload. To this end, there are two approaches, one focused on the client and one on the payloads. In the former, the goal is to fingerprint the client used to perform the requests. As discussed, this can be easily accomplished using solutions like JA3 and JA4, which indeed create unique fingerprints for the four fuzzers. Despite offering an easy-to-compute method, these fingerprints are very fragile. By making minor changes to the user agent string or using

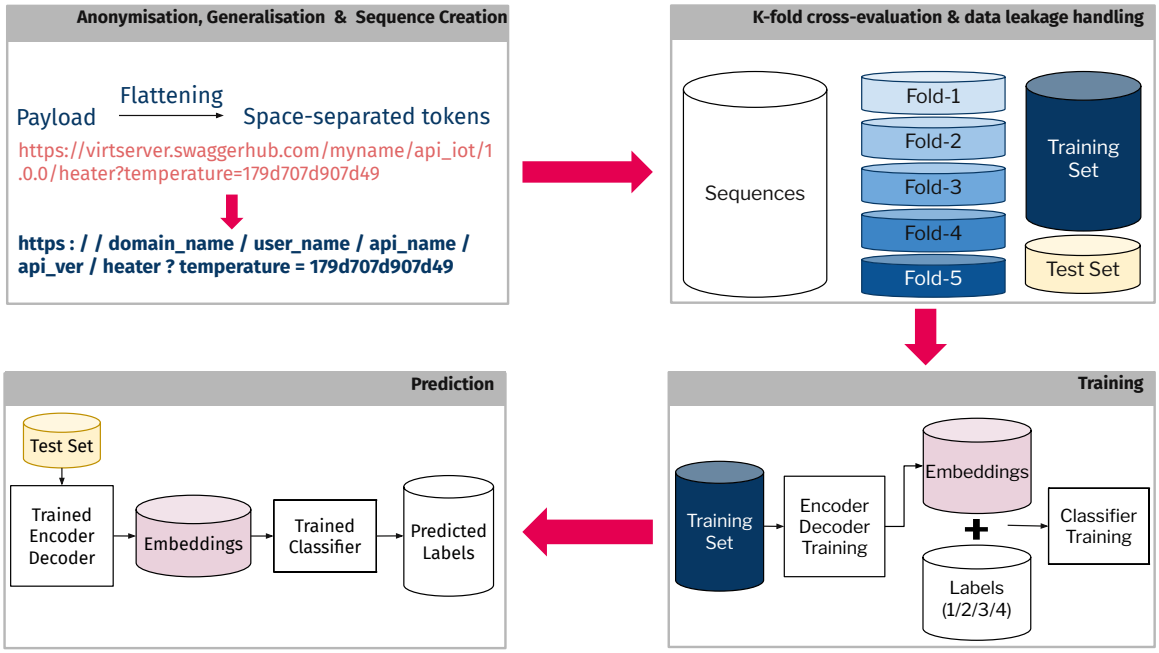

Fig. 2. Overview of our approach: Payloads are anonymized, generalized, and tokenized. The dataset is split into five folds for k-fold cross-validation (k = 5), preventing data leakage by removing sequences from the training set that appear in the test set. During each iteration, a Transformers-based encoder-decoder model is trained using four folds to produce embeddings, which are then used to train a classifier. The classifier is evaluated on the remaining fold, repeating the process for all folds.

another version of the libraries (which can change the encryption algorithms), the extracted fingerprint can be completely different, rendering this approach useless. The above changes can be made deliberately, i.e., the attacker changed them to avoid detection, or inadvertently, i.e., the attacker made a system update. Nevertheless, given that defenders collect such fingerprints and block them see, for instance, the blocklist of Abuse[6], advanced attackers are expected to bypass them.

As a result, we consider that a more robust approach is the detection through behavioral patterns and foster a payload-based approach. This approach is higher in the pyramid of pain compared to the fingerprint approach and, therefore, more difficult for the attacker to bypass. Indeed, as we demonstrate, our AI-based solution can accurately identify the attacker's tooling with little resources.

To minimize engineering and computational overhead, our approach aims to automatically learn relevant features of payloads without relying on manual feature engineering. This is achieved by dividing the problem into two stages: first, generating payload representations using a code embedding technique, and second, identifying the origin, i.e., the fuzzer, based on these embeddings.

A. Overview

Figure 2 illustrates the overall workflow of our approach, which is divided into *4* key steps:

1) *Building a token representation*: This step involves preprocessing the original payload to remove redundant information and standardize the request URL. The result is a homogenized URL code, which is then tokenized into a sequence of tokens. Each payload is transformed into its tokenized representation to be used in subsequent steps.
2) *K-fold cross-validation & Data leakage handling*: The dataset is partitioned into *5* folds for k-fold cross-validation. During each iteration, *4* folds are utilized for training, while the remaining fold is reserved for testing. To prevent data leakage, we remove all token sequences from the training set that appear in the test set, thereby providing a robust evaluation of the model's generalization.
3) *Payload Representation Learning*: A Transformers-based encoder-decoder model is trained to produce embeddings, or vector representations, of the payloads. This architecture was chosen for its ability to capture sequential dependencies within payloads, making it well-suited for tasks involving structured data like API requests. This enables our approach to automatically capture the relevant features of the payloads without the need for explicit feature definition.
4) *Classification*: A classification model is trained to associate the embeddings with their corresponding fuzzers, effectively identifying the origin of each payload.

Our approach aims to learn the properties of the payloads that are useful for associating them with their fuzzer origins. This aligns with recent studies on contextual selection [7], [8] focused on classification. This characteristic makes our

[6]https://sslbl.abuse.ch/ja3-fingerprints/

approach applicable to payloads unseen during training.

The generation and collection of fuzzing payloads are crucial in building a robust dataset for our fuzzer identification task. To accurately capture a diverse range of attack vectors and API responses, the APIs must represent real-world scenarios and varied complexities. For this purpose, we developed *five* distinct APIs, each tailored to different use cases and security configurations, which were attacked using *four* fuzzers, namely APIFuzzer, Kiterunner, RESTler, and Schemathesis. These APIs span from simple CRUD operations to more complex OAuth2 flows and IoT integrations, providing a comprehensive testbed for generating fuzzing payloads that reflect different security contexts and operations. The description of the developed APIs is reported in Table I.

Each API follows the *OpenAPI 3.0* specification, providing a detailed structure for the fuzzing process by defining the API's endpoints, parameters, data types, and expected responses. This standardized format enables the fuzzers to systematically generate relevant payloads and test input variations. The payloads generated from these fuzzers were collected, processed, and utilized to train our classification architecture to identify the fuzzer responsible for each payload. This diverse API setup ensures the fuzzers are tested across different contexts, enhancing the robustness of the dataset and providing realistic attack scenarios for fuzzer identification.

B. Training Sequences Generation

A significant challenge in working with payloads is the presence of private or sensitive information, such as user credentials, API keys, or personally identifiable data, but also deployment specific information, such as domain and API names. To overcome these constraints, we preprocess the payloads by anonymizing and generalizing identifiable entities, including domain names, usernames, API names, API versions, and other private data, replacing them with generic placeholders. This anonymization step ensures that our model focuses on learning the structural and behavioral patterns of the payloads, rather than memorizing sensitive content.

To anonymize and generalize the payloads, we developed a custom script that identifies specific sensitive fields within the payloads and replaces them with reusable placeholders, as shown in Figure 2. Each placeholder follows the format `type`, where `type` indicates the nature of an entity e.g., `domain_name`, `user_name`, `api_name`, and `api_ver`. These placeholders are reused consistently across the payloads wherever the sensitive entity appears, allowing consistency while ensuring privacy. This preprocessing step also enables the model to generalize well to unseen data while protecting sensitive information.

Next, we preprocess each payload by transforming it into a single space-separated sequence of tokens. This tokenized representation captures the complete structure of the payload while maintaining computational efficiency during training [36], [37]. We set the maximum sequence length to 120 tokens, corresponding to the longest payload in our dataset. Training models with this sequence length required less than 3 hours on an NVIDIA A100 GPU.

C. Learning Payload Representations with Transformers-based Encoder-Decoder

The next step in our approach involves generating vector representations, or embeddings, from the tokenized payloads, that are subsequently used to train the classification model. To achieve this, we implement a Transformers-based encoder-decoder architecture, a neural network model widely utilized for representation learning tasks [38], [39]. In this setup, the encoder converts the tokenized payload into an embedding, while the decoder reconstructs the original token representation from the embedding. The training objective is to minimize the binary cross-entropy loss between the input token sequence and the reconstructed output. Once trained, the encoder can generate embeddings for unseen payloads by processing their tokenized representations.

For our Transformers-based encoder-decoder implementation, we employ a bi-directional Recurrent Neural Network (RNN) [40]–[43]. Our approach leverages the *tf-seq2seq* framework [44], a robust and flexible framework for building models. We use Gated Recurrent Units (GRU) [45] as RNN cells, as GRUs have demonstrated superior performance over simpler architectures like vanilla RNNs [46], [47]. To enhance the model's capacity, we integrate an attention mechanism using AttentionLayerBahdanau [48], configured with a two-layer AttentionDecoder and a single-layer BidirectionalRNNEncoder, each comprising 128 units.

To determine the optimal number of training epochs, we conducted a preliminary experiment using a small validation set, separate from the training and test data. This allowed us to monitor the model's convergence and adjust the training duration accordingly. We observed that the model converges effectively after just one training epoch, with sequences of up to 120 tokens in length. Further training beyond the first epoch did not improve the embeddings or overall performance. This suggests that the model quickly captures the essential features of the payloads, and additional training does not enhance the quality of the learned representations. Consequently, we restrict the training process to *one* epoch, which not only ensures efficient convergence but also minimizes the risk of overfitting [49] by preventing excessive iterations that could lead the model to memorize the training data instead of generalizing to unseen payloads.

D. Classifying Payloads to Identify their Fuzzer-Origins

Subsequently, we train a classifier to identify the fuzzer responsible for generating a given payload. The classifier aims to categorize each payload (represented by the embedding generated by the encoder) into one of the four classes, corresponding to the four fuzzers that we study. The primary objective is to maximize classification performance, evaluated using standard metrics, as detailed in Section V-A.

For the classification task, we employ random forests [50], chosen for their computational efficiency and demonstrated

TABLE I
OVERVIEW OF THE APIS USED IN OUR EXPERIMENTS

API Name	Description	Key Operations	Security
Simple Item API	A basic CRUD API demonstrating basic operations on items.	`GET /items`, `POST /items`, `GET /items/{itemId}`, `PUT /items/{itemId}`	None
OAuth2 Password Flow API	An example API using OAuth2 password flow to describe security and multiple operations.	`GET /users`, `POST /users`, `GET /users/{userId}`, `DELETE /users/{userId}`	OAuth2 (password)
OAuth2 Application Flow API	Another OAuth2-based API demonstrating security configurations for application flow.	`GET /example`, `POST /users`	OAuth2 (client credentials)
Swagger Petstore API	A Swagger Petstore server, widely used as a reference for API testing and demonstration purposes.	`GET /pet/{petId}`, `POST /pet`, `GET /store/inventory`, `POST /store/order`	API Key, OAuth2
Home IoT API	An API developed for the EatBacon IoT project[7], representing typical IoT operations (lighting, temperature, etc.).	`GET /devices`, `POST /lighting/dimmers/{devic` `GET /temperature`	None

success in addressing various software engineering problems [51], [52]. The model is configured with standard hyperparameters: 100 trees, Gini impurity for decision node splitting, and the square root of the total number of features (embedding logits) to determine the features considered at each split.

After training, the random forest classifier is utilized to predict the originating fuzzer for previously unseen payloads. The payloads first pass through the pre-processing pipeline, where sensitive information is anonymized. Then, they are processed by the Transformers-based encoder-decoder architecture to generate embeddings, which are subsequently input into the classifier to determine the predicted fuzzer class.

V. EXPERIMENTAL SETUP

A. Data and Tools

We developed five RESTful APIs, following the OpenAPI specification, to answer our RQs. These APIs were designed to cover a range of use cases, including simple CRUD operations, OAuth2 flows, and IoT interactions, providing a diverse set of scenarios for testing fuzzing techniques. Table I provides an overview of these APIs, including the key operations, descriptions, and the security mechanisms implemented. These APIs provide a diverse set of test cases for enabling a fuzzer's ability to generate diverse payloads.

After creating these APIs, we executed the *4* fuzzers—*APIFuzzer* [22], *Kiterunner* [23], *RESTler* [20], and *Schemathesis* [21]—on each API to generate extensive sets of fuzzing payloads (see section II-B for fuzzer details). The results of each fuzzer's execution are summarised in Table II. These payloads were analyzed to evaluate the overlap between fuzzers and subsequently utilized for training and testing our models.

Our approach's predictions can result in *4* possible outputs corresponding to the *4* fuzzer classes. For multi-class classification, where the goal is to classify payloads by their originating fuzzer, we compute Precision, Recall, and F1-score for each class (fuzzer) individually. We then report the macro-averaged values across all classes, offering a comprehensive assessment of the model's overall performance. Additionally, using the F1-score enables more objective and balanced conclusions about the classifier's effectiveness.

TABLE II
OVERVIEW OF THE PAYLOADS GENERATED BY FUZZERS

Fuzzer	Payloads (#)	Payloads (%)
APIFuzzer	499,255	14.93%
Kiterunner	864,182	25.85%
RESTler	1,128,228	33.74%
Schemathesis	851,970	25.48%
Total	3,343,635	100.00%

B. Experimental Procedure

To address our research questions, we run the *4* fuzzers on the *5* APIs we developed (Section V-A) to generate fuzzing payloads. Each fuzzer is configured with its default setup, providing a comprehensive set of payloads for each API. The experimental process involves evaluating the overlap between these payloads and training a classifier to determine the origin fuzzer for each payload.

To answer *RQ1*, we analyze the generated payloads to assess the extent of overlap across different fuzzers. Specifically, we calculate the percentage of identical payloads shared between each pair of fuzzers. This step is essential to quantify the challenge of overlapping payloads when training a classifier to distinguish between fuzzers. Results are presented as overlap percentages in Table III, which provide insights into the degree of similarity across fuzzers.

To answer *RQ2*, we train models on the generated payloads using a combined deep learning and machine learning architecture. The training and testing process involves k-fold cross-validation (k = 5) to ensure robust evaluation, where we train the classifier on four folds (*80%* of the dataset) and test it on the remaining fold (i.e., the remaining *20%*). Within each iteration, *10%* of the training data is reserved as the validation

TABLE III
PAYLOAD OVERLAP ANALYSIS (RQ1)

Fuzzer	APIFuzzer	Kiterunner	RESTler	Schemathesis
APIFuzzer	-	0.03%	2.89%	1.29%
Kiterunner	0.02%	-	0	0
RESTler	1.28%	0	-	12.86%
Schemathesis	0.76%	0	17.02%	-

set to monitor the model's performance during training, prevent overfitting, and fine-tune hyperparameters. We measure the classifier's performance in accurately identifying the fuzzer (origin) for each payload in the test set(s) using standard prediction performance metrics—Precision, Recall, and F1-score. This evaluation demonstrates the model's effectiveness in distinguishing fuzzers despite the overlapping in payloads.

VI. EXPERIMENTAL RESULTS

A. Payload Overlap Analysis (RQ1)

Table III presents the results of our payload overlap analysis (RQ1), showing the extent of shared payloads between each pair of fuzzers. It reports the total number of payloads generated by each fuzzer, along with the count and percentage of overlapping payloads observed between different fuzzer pairs.

Starting with *APIFuzzer* (which generated 499,255 payloads), we found that 134 payloads (0.03%) overlapped with those generated by *Kiterunner*. This relatively minor overlap indicates that APIFuzzer and Kiterunner largely produce distinct payloads. However, when comparing APIFuzzer with *RESTler*, we observed a higher overlap of 14,462 payloads, amounting to 2.89% of APIFuzzer's total payloads and 1.28% of RESTler's payloads. Similarly, APIFuzzer shared 6,469 payloads (1.29%) with *Schemathesis*, indicating a moderate level of similarity in the payloads generated by these fuzzers.

For *Kiterunner*, which generated 864,182 payloads, there was no overlap with either RESTler or Schemathesis. This lack of shared payloads highlights the uniqueness of Kiterunner's payload generation. Compared to the other fuzzers, Kiterunner's payloads are more distinct.

RESTler, responsible for generating 1,128,228 payloads, exhibited a significant overlap with Schemathesis, sharing 145,039 payloads. This overlap accounts for 12.86% of RESTler's payloads and 17.02% of Schemathesis's payloads, suggesting that these two fuzzers produce notably similar payloads when attacking the same APIs.

Finally, *Schemathesis* generated a total of 851,970 payloads. Notably, there was no overlap between Schemathesis and Kiterunner. This absence of shared payloads further underscores the distinctive patterns in payload generation between these two fuzzers.

These results reveal varying degrees of overlap among the fuzzers, highlighting distinct patterns in their payload generation strategies. RESTler and Schemathesis show the highest similarity, with a significant portion of their payloads overlapping, suggesting that these fuzzers target vulnerability identification in a similar way or use comparable techniques to generate payloads. In contrast, Kiterunner stands out with minimal overlap, indicating a more unique approach to payload generation that sets it apart in terms of diversity. This uniqueness in Kiterunner's payloads suggests that it may cover different testing dimensions or employ distinct strategies compared to the other fuzzers, making it less likely to generate redundant test cases.

To better understand the significant overlap between RESTler and Schemathesis (≈17%), we conducted a payload analysis. Our investigation revealed that both fuzzers employ schema-driven strategies, particularly focusing on edge cases defined by OpenAPI specifications. This results in convergent payloads when targeting APIs with similar structures. For example, RESTler and Schemathesis often generate similar boundary-value inputs for integer parameters, as these are common test cases derived from schema constraints.

Summary of Payload Overlap Analysis (RQ1)

RQ1: How much overlap exists between the payloads generated by different API fuzzers?
Our analysis shows that payload overlap varies among fuzzers. RESTler and Schemathesis exhibit the highest overlap, suggesting similar methodologies, while Kiterunner has minimal overlap, indicating unique payload generation. These findings reflect distinct approaches among fuzzers in targeting API vulnerabilities.

B. Fuzzer Classification Performance (RQ2)

Table IV summarizes the classification performance achieved by our combined deep learning and machine learning architecture in identifying the originating fuzzer of each payload. The metrics—precision, recall, and F1-score—offer insight into the model's accuracy across each fuzzer class, highlighting the strengths and weaknesses in distinguishing between fuzzers based on their payload characteristics.

Starting with *APIFuzzer*, the model achieved a precision of 0.81, a recall of 0.94, and an F1-score of 0.87. These results indicate that while the model is relatively accurate in identifying APIFuzzer's payloads, with a high recall indicating most APIFuzzer payloads are correctly classified, the slightly lower precision suggests a few misclassifications of other fuzzers as APIFuzzer. This balance between precision and recall results in an F1-score of 0.87, demonstrating overall reliable classification for APIFuzzer.

For *Kiterunner*, the model exhibited near-perfect performance, achieving a precision of 0.99, recall of 1.0, and F1-score of 0.99. This high level of accuracy suggests that Kiterunner payloads have unique features that are easily recognizable by the model, resulting in minimal misclassifications. The model's performance with Kiterunner underscores Kiterunner's distinctive payload characteristics, as seen previously in the low overlap with other fuzzers.

The model's performance on *RESTler* payloads achieved a precision of 0.83, recall of 0.97, and F1-score of 0.90. With a relatively high recall, most RESTler payloads were

TABLE IV
FUZZER CLASSIFICATION PERFORMANCE (RQ2)

Fuzzer	Precision	Recall	F1-score
APIFuzzer	0.81	0.94	0.87
Kiterunner	0.99	1.00	0.99
RESTler	0.83	0.97	0.90
Schemathesis	0.96	0.64	0.77
Macro Avg	**0.90**	**0.89**	**0.88**

accurately identified, but the slightly lower precision indicates that a few payloads from other fuzzers were misclassified as RESTler. This moderate level of precision, combined with a high recall, results in a solid F1-score of 0.90, reflecting the model's effectiveness in classifying RESTler payloads, though with some overlapping characteristics with other fuzzers.

Schemathesis presented a different profile, with a high precision of 0.96 but a lower recall of 0.64, resulting in an F1-score of 0.77. This high precision indicates that the model's predictions of a payload as Schemathesis are highly accurate; however, the lower recall indicates that many Schemathesis payloads were missed, potentially due to their similarity with RESTler payloads, as observed in the overlap analysis in RQ1. The F1-score of 0.77 reflects this balance, indicating room for improvement in accurately capturing the breadth of Schemathesis payload characteristics.

The *Macro Average* scores across all fuzzers show an overall precision of 0.90, recall of 0.89, and F1-score of 0.88. These averages highlight that, while the models perform well in distinguishing between fuzzers, certain fuzzers (for instance, Schemathesis) present a greater challenge due to shared characteristics with other fuzzers.

We conducted an error analysis to identify common misclassifications. For instance, 24.91% of Schemathesis payloads were misclassified as RESTler, reflecting the high overlap identified in *RQ1*. This aligns with Schemathesis's relatively low recall of 0.64. Our analysis showed that Schemathesis payloads often share significant similarities with RESTler payloads, especially in parameters influenced by schema constraints, such as boundary values for integers. These similarities in parameter structures and values make it difficult for the classifier to differentiate between the two fuzzers effectively. Conversely, RESTler payloads were rarely misclassified as Schemathesis (0.56% misclassified), supporting its high recall of 0.97. Similarly, Kiterunner and APIFuzzer showed excellent performance, with Kiterunner achieving near-perfect recall and precision due to its distinct payload generation strategy.

Summary of Fuzzer Classification Performance (RQ2)

RQ2: How well can a combined deep learning and machine learning architecture distinguish between payloads to identify their fuzzer origins, despite overlapping payloads? Our classification model performs well across fuzzers, achieving high precision and recall overall. Kiterunner and APIFuzzer are identified with high accuracy, while RESTler and Schemathesis present more challenges due to overlapping payload characteristics.

VII. DISCUSSION

Our study introduces a new approach to identifying the attacker's tooling, focusing on fuzzers. To this end, we focus on the origins of fuzzed payloads, aiming to enhance cybersecurity defenses by tracing payloads back to the fuzzers; and potentially, the attack groups behind them. Our findings, in addition to validating the feasibility of fuzzer identification based on payloads, also reveal critical insights into the characteristics of different fuzzers, offering implications for threat attribution and strategic defense planning. Given the randomized way that fuzzers work, payload identification is far more challenging than, e.g., trying to identify tools that deliver static payloads.

The payload overlap analysis reveals distinct patterns among the *four* fuzzers, highlighting a substantial overlap between RESTler and Schemathesis, while APIFuzzer and Kiterunner exhibit more distinct payload patterns. For instance, RESTler and Schemathesis share significant similarities, due to common approaches in payload generation and targeted vulnerability types, which suggests that these fuzzers target similar aspects of API security. Conversely, Kiterunner stands out with minimal overlap with other fuzzers, implying a unique methodology targeting different vulnerabilities. While a high overlap indicates commonly exploited API weaknesses or similar algorithms to generate the payloads, unique payloads suggest novel or specialized attack vectors that warrant particular attention.

The classification performance achieved by our combined deep learning and machine learning architecture underscores the potential for accurately identifying fuzzers based on payload characteristics. The high precision and recall scores for APIFuzzer and Kiterunner highlight the model's capacity to differentiate payloads, especially for fuzzers with distinctive patterns. However, the moderate recall for Schemathesis indicates future works to fully distinguish between fuzzers with overlapping payload characteristics, such as RESTler and Schemathesis. While the current study focuses on payload content for classification, additional model features could potentially improve accuracy by capturing behavioral patterns. For example, analyzing the timing and frequency of payload generation may reveal insights into the operational characteristics of different fuzzers. Prior work in network traffic analysis [53] has demonstrated the utility of such temporal features in improving classification tasks. Exploring these features in the future can potentially help differentiate between fuzzers with overlapping payload content.

An important implication of fuzzer identification lies in its potential to strengthen strategic threat attribution and defense planning. In cybersecurity, different attack groups often favor specific tools and techniques aligned with their objectives, such as data theft, ransomware, or API exploitation. This comes as a result of the specialization of some groups in specific attack stages. For instance, among others, there are initial access brokers, ransomware operators, carders, and malware authors. Beyond their attack specialization, each threat actor uses its tooling [54]. In this regard, our work focuses on initial access brokers that attempt to penetrate an organization by exploiting an open API. By accurately identifying the fuzzer used in an attack, defenders can trace the intrusion back to particular groups with known goals and tactics, creating a framework for threat intelligence that aligns defense strategies with anticipated attacker behavior. For instance, if a detected payload is traced back to a fuzzer typically associated with an attack group focused on data theft, the blue teams may prioritize data protection and monitor possible exfiltration attempts. Conversely, if the identified fuzzer aligns with groups known for ransomware or extortion, an organization could preemptively secure backups, increase access controls, and prepare incident response protocols for rapid action. This level of attack attribution and preemptive defense planning enables security teams to go beyond a generic response, adopting targeted defenses that address the specific goals and methodologies of the attacker. In a dynamic threat landscape, this approach can enhance the organization's resilience against multi-stage, complex attacks by ensuring that defense mechanisms are tailored to address the particular risks associated with different attack groups. Furthermore, by tracing back to specific fuzzer tools, organizations can track trends in attack techniques, observing which fuzzers or payload types are becoming more prevalent and potentially forecasting emerging threats based on the usage patterns of different fuzzers across various attack groups.

Limitations. While our results demonstrate our architecture's effectiveness in fuzzer identification, certain limitations should be acknowledged. *Firstly*, our dataset consists of *4* fuzzers and *5* API cases, which, while diverse, may limit the generalizability of our findings. Future research could expand this dataset to include additional fuzzers and APIs, which can provide a broader foundation for model training and enable further refinement of the classification model. *Secondly*, our current approach focuses primarily on the content of the payloads. Incorporating supplementary features such as request frequency, timing, or metadata, could further enhance the model's ability to distinguish fuzzers, particularly those with overlapping payload patterns. This could increase robustness in real-world scenarios where additional context beyond payload content may improve fuzzer identification. *Thirdly*, our methodology relies on payload preprocessing that may influence model performance. For example, we used the maximum payload length (120 tokens) observed in the dataset as the sequence length to ensure the model captures the full range of payload features. While this approach avoids constraining sequence length artificially, it may not generalize to datasets with significantly longer or more variable payloads. *Fourthly*, the evaluation metrics we use—precision, recall, and F1-score—are well-suited for classification tasks but may not capture all nuances of performance. Nonetheless, these metrics are standard in classification tasks and provide a reliable measure of our model's effectiveness in identifying fuzzers. Additionally, our overlap analysis considers only payload content and does not incorporate behavioral features like timing or frequency, which could further inform classification accuracy. *Finally*, while our Transformers-based encoder-decoder model has proven effective for this classification task, alternative architectures could be explored to capture even finer-grained differences in payloads. Moreover, deploying a classification model in real-world settings poses challenges, particularly in achieving near-real-time predictions. Our initial tests indicate that our architecture takes approximately *100ms* to process a single payload and produce a prediction (i.e., identify the fuzzer) on standard hardware. While this is suitable for low-traffic environments, it may present limitations for high-traffic APIs where rapid processing of numerous payloads is required.

VIII. Conclusions

In this study, we introduced a method for classifying API fuzzers by analyzing the payloads they generate, aiming to enhance defensive strategies for API security. By evaluating four prominent fuzzers—APIFuzzer, Kiterunner, RESTler, and Schemathesis—across a diverse set of APIs, we demonstrated that identifying fuzzers based on payload characteristics is feasible, even with considerable overlap in payload content between fuzzers like RESTler and Schemathesis. Our results show that a combined deep learning and machine learning architecture achieves high classification accuracy, effectively distinguishing fuzzers with notable precision and recall, particularly for fuzzers with unique payload patterns, such as Kiterunner.

The payload overlap analysis highlighted that while some fuzzers produce unique payloads, others share significant overlap, which can complicate fuzzer classification. Our approach successfully overcomes these complexities by focusing on each payload's distinct behavioral and structural elements. This enables a more nuanced understanding of each fuzzer's methodology, providing valuable insights into how different tools target vulnerabilities in APIs.

This work also has implications for broader threat intelligence and attack attribution efforts. Identifying the fuzzer responsible for a set of API payloads, and in general, the adversary's tooling, can help trace back to specific attack groups, many of which use consistent tooling and techniques. With this knowledge, defenders can tailor their response to the tactics of particular groups—whether data theft, ransomware, or extortion-focused—thereby strengthening their security posture. Future work may expand upon these findings by incorporating additional features, such as temporal and frequency-based analysis, to further improve fuzzer classification accu-

racy and resilience against emerging threats but also extend to other tools that adversaries may use in other attack stages.

IX. Data Availability

The dataset generated and analyzed during our study, including all payloads produced by APIFuzzer, Kiterunner, RESTler, and Schemathesis, is available in our GitHub repository[8]. The source code and trained models of our combined deep learning and machine learning architecture, along with supplementary analysis scripts, are also available in this repository.

References

[1] Lockheed Martin, "The Cyber Kill Chain," https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html, 2024.

[2] B. E. Strom, A. Applebaum, D. P. Miller, K. C. Nickels, A. G. Pennington, and C. B. Thomas, "Mitre att&ck: Design and philosophy," in *Technical report*. The MITRE Corporation, 2018.

[3] D. Bianco, "The pyramid of pain," https://detect-respond.blogspot.com/2013/03/the-pyramid-of-pain.html, 2013.

[4] K. Sheridan, "Cobalt Strike & Metasploit tools were attacker favorites in 2020," https://www.darkreading.com/cyber-risk/cobalt-strike-metasploit-tools-were-attacker-favorites-in-2020, 2021.

[5] National Cyber Security Cenre, "Further ttps associated with svr cyber actors," https://www.ncsc.gov.uk/files/Advisory%20Further%20TTPs%20associated%20with%20SVR%20cyber%20actors.pdf, 2021.

[6] M. Harbison and P. Renals, "When pentest tools go brutal: Red-teaming tool being abused by malicious actors," https://unit42.paloaltonetworks.com/brute-ratel-c4-tool/, 2022.

[7] C. Liang, Q. Wei, J. Du, Y. Wang, and Z. Jiang, "Survey of source code vulnerability analysis based on deep learning," *Comput. Secur.*, vol. 148, p. 104098, 2025.

[8] A. Garg, R. Degiovanni, M. Jimenez, M. Cordy, M. Papadakis, and Y. L. Traon, "Learning from what we know: How to perform vulnerability prediction using noisy historical data," *Empir. Softw. Eng.*, vol. 27, no. 7, p. 169, 2022. [Online]. Available: https://doi.org/10.1007/s10664-022-10197-4

[9] H. J. Abdelnur, R. State, and O. Festor, "Kif: a stateful sip fuzzer," in *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*, 2007, pp. 47–56.

[10] K. Dewey, J. Roesch, and B. Hardekopf, "Language fuzzing using constraint logic programming," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 725–730.

[11] ——, "Fuzzing the rust typechecker using clp (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 482–493.

[12] P. Godefroid, R. Singh, and H. Peleg, "Machine learning for input fuzzing," Apr. 20 2021, uS Patent 10,983,853.

[13] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 643–653.

[14] X. Liu, X. Li, R. Prajapati, and D. Wu, "Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 01, 2019, pp. 1044–1051.

[15] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, 2017, pp. 2329–2344.

[16] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[17] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "Collafl: Path sensitive fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 679–696.

[18] Peach, "Peach fuzzer," https://about.gitlab.com/solutions/security-compliance/, 2024, last accessed 2024-10-17.

[8]https://github.com/garghub/fuzzing

[19] SPIKE, "Spike fuzzer," http://resources.infosecinstitute.com/fuzzerautomation-with-spike/, 2024, last accessed 2024-10-17.

[20] V. Atlidakis, P. Godefroid, and M. Polishchuk, "Restler: Stateful rest api fuzzing," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 748–758.

[21] Z. Hatfield-Dodds and D. Dygalo, "Deriving semantics-aware fuzzers from web API schemas," in *44th IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 2022, pp. 345–346. [Online]. Available: https://doi.org/10.1145/3510454.3528637

[22] P. Kiss, "Apifuzzer," https://pypi.org, April 2022, accessed: 2024-10-18.

[23] S. Yeoh, "Kiterunner," AssetNote, April 2021, accessed: 2021-11-04.

[24] I. Goncharov, "Apis.guru - a directory of openapi (swagger) specifications," accessed: 2024-10-30. [Online]. Available: https://apis.guru/

[25] O. Hartig and J. Pérez, "Semantics and complexity of graphql," in *Proceedings of the 2018 World Wide Web Conference*, 2018, pp. 1155–1164.

[26] D. R. MacIver, Z. Hatfield-Dodds *et al.*, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[27] N. Naik, P. Jenkins, P. Grace, and J. Song, "Comparing attack models for it systems: Lockheed martin's cyber kill chain, mitre att&ck framework and diamond model," in *2022 IEEE International Symposium on Systems Engineering (ISSE)*, 2022, pp. 1–7.

[28] L. M. Fadzil, S. Manickam, and M. A. Al-Shareeda, "A review of an emerging cyber kill chain threat model," in *2023 Second International Conference on Advanced Computer Applications (ACA)*, 2023, pp. 157–161.

[29] N. Rani, B. Saha, and S. K. Shukla, "A comprehensive survey of advanced persistent threat attribution: Taxonomy, methods, challenges and open research problems," *CoRR*, vol. abs/2409.11415, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.11415

[30] F. Skopik and T. Pahi, "Under false flag: using technical artifacts for cyber attack attribution," *Cybersecur.*, vol. 3, no. 1, p. 8, 2020. [Online]. Available: https://doi.org/10.1186/s42400-020-00048-4

[31] G. Zhao, K. Xu, L. Xu, and B. Wu, "Detecting apt malware infections based on malicious dns and traffic analysis," *IEEE access*, vol. 3, pp. 1132–1142, 2015.

[32] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng, "Malware traffic classification using convolutional neural network for representation learning," in *2017 International conference on information networking (ICOIN)*. IEEE, 2017, pp. 712–717.

[33] Z. Fu, M. Liu, Y. Qin, J. Zhang, Y. Zou, Q. Yin, Q. Li, and H. Duan, "Encrypted malware traffic detection via graph-based network analysis," in *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, 2022, pp. 495–509.

[34] H. Liu, B. Lang, M. Liu, and H. Yan, "Cnn and rnn based payload classification methods for attack detection," *Knowledge-Based Systems*, vol. 163, pp. 332–341, 2019.

[35] Z. Liu, Y. Fang, C. Huang, and J. Han, "Graphxss: an efficient xss payload detection approach based on graph convolutional network," *Computers & Security*, vol. 114, p. 102597, 2022.

[36] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 25–36.

[37] A. Garg, M. Ojdanic, R. Degiovanni, T. T. Chekam, M. Papadakis, and Y. L. Traon, "Cerebro: Static subsuming mutant selection," *IEEE Trans. Software Eng.*, vol. 49, no. 1, pp. 24–43, 2023. [Online]. Available: https://doi.org/10.1109/TSE.2022.3140510

[38] S. Pouyanfar, S. Sadiq, Y. Yan, H. Tian, Y. Tao, M. E. P. Reyes, M. Shyu, S. Chen, and S. S. Iyengar, "A survey on deep learning: Algorithms, techniques, and applications," *ACM Comput. Surv.*, vol. 51, no. 5, pp. 92:1–92:36, 2019. [Online]. Available: https://doi.org/10.1145/3234150

[39] A. Garg, R. Degiovanni, M. Jimenez, M. Cordy, M. Papadakis, and Y. L. Traon, "Learning to predict vulnerabilities from vulnerability-fixes: A machine translation approach," *CoRR*, vol. abs/2012.11701, 2020. [Online]. Available: https://arxiv.org/abs/2012.11701

[40] J. Song, S. Kim, and S. Yoon, "Alignart: Non-autoregressive neural machine translation by jointly learning to estimate alignment and translate," in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual*

Event / Punta Cana, Dominican Republic, 7-11 November, 2021, M. Moens, X. Huang, L. Specia, and S. W. Yih, Eds. Association for Computational Linguistics, 2021, pp. 1–14. [Online]. Available: https://doi.org/10.18653/v1/2021.emnlp-main.1

[41] D. Britz, A. Goldie, M. Luong, and Q. V. Le, "Massive exploration of neural machine translation architectures," *CoRR*, vol. abs/1703.03906, 2017. [Online]. Available: http://arxiv.org/abs/1703.03906

[42] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds., 2014, pp. 3104–3112.

[43] A. Garg, R. Degiovanni, M. Papadakis, and Y. L. Traon, "Vulnerability mimicking mutants," *CoRR*, vol. abs/2303.04247, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2303.04247

[44] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[45] 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," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. [Online]. Available: https://aclanthology.org/D14-1179

[46] A. Shewalkar, D. Nyavanandi, and S. A. Ludwig, "Performance evaluation of deep neural networks applied to speech recognition: Rnn, LSTM and GRU," *J. Artif. Intell. Soft Comput. Res.*, vol. 9, no. 4, pp. 235–245, 2019. [Online]. Available: https://doi.org/10.2478/jaiscr-2019-0006

[47] A. Garg, R. Degiovanni, M. Papadakis, and Y. L. Traon, "On the coupling between vulnerabilities and llm-generated mutants: A study on vul4j dataset," in *IEEE Conference on Software Testing, Verification and Validation, ICST 2024, Toronto, ON, Canada, May 27-31, 2024*. IEEE, 2024, pp. 305–316. [Online]. Available: https://doi.org/10.1109/ICST60714.2024.00035

[48] D. Bahdanau, J. Chorowski, D. Serdyuk, P. Brakel, and Y. Bengio, "End-to-end attention-based large vocabulary speech recognition," in *2016 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2016, Shanghai, China, March 20-25, 2016*. IEEE, 2016, pp. 4945–4949.

[49] R. Roelofs, V. Shankar, B. Recht, S. Fridovich-Keil, M. Hardt, J. Miller, and L. Schmidt, "A meta-analysis of overfitting in machine learning," in *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, Eds., 2019, pp. 9175–9185.

[50] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[51] P. A. A. Resende and A. C. Drummond, "A survey of random forest based methods for intrusion detection systems," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 48:1–48:36, 2018.

[52] A. Garg, R. Degiovanni, F. Molina, M. Cordy, N. Aguirre, M. Papadakis, and Y. L. Traon, "Enabling efficient assertion inference," in *34th IEEE International Symposium on Software Reliability Engineering, ISSRE 2023, Florence, Italy, October 9-12, 2023*. IEEE, 2023, pp. 623–634. [Online]. Available: https://doi.org/10.1109/ISSRE59848.2023.00039

[53] J. Koumar, K. Hynek, and T. Cejka, "Network traffic classification based on single flow time series analysis," in *19th International Conference on Network and Service Management, CNSM 2023, Niagara Falls, ON, Canada, October 30 - Nov. 2, 2023*. IEEE, 2023, pp. 1–7.

[54] C. Patsakis, D. Arroyo, and F. Casino, "The malware as a service ecosystem," in *Malware: Handbook of Prevention and Detection*. Springer, 2024, pp. 371–394.

Aayush Garg received his Master's degree in Computer Science with a focus on Security from Boston University, USA, and completed his PhD in Computer Science, specializing in Quality Assurance and Deep Learning, at the University of Luxembourg. His early career as a Software Engineer in the Fintech sector provided a solid foundation in Software Development and QA. Building on this expertise, he transitioned to a Researcher role at the Interdisciplinary Centre for Security, Reliability, and Trust (SnT), University of Luxembourg. Currently, Aayush is a Research and Technology Scientist at the Luxembourg Institute of Science and Technology (LIST), Luxembourg. He actively contributes to National and European research initiatives, and his work appears frequently in top international conferences and journals. His primary research interests include Software Security, Quality Assurance, and Deep Learning.

Constantinos Patsakis received his B.Sc. in Mathematics from the University of Athens, Greece, an M.Sc. in Information Security from Royal Holloway, University of London, and a PhD in Cryptography and Malware from the University of Piraeus, Greece. In the past, he has worked as a researcher at the UNESCO Chair in Data Privacy, at Rovira i Virgili, at Trinity College, Dublin, and the Luxembourg Institute of Science and Technology. He is currently a Professor at the University of Piraeus and an Adjunct Researcher at the Athena Research and Innovation Center. He has authored numerous publications in prestigious peer-reviewed international conferences and journals and participated in several national and European Research and Development projects. His main areas of research include cryptography, security, privacy, blockchains, and cybercrime.

Zanis Ali Khan is an R&T Scientist at the Luxembourg Institute of Science and Technology (LIST), where he specializes in vulnerability detection and patching. He earned his PhD in Computer Science from the University of Luxembourg, focusing on log analysis and anomaly detection. Following his PhD, Zanis undertook a postdoctoral research position to further investigate methods for identifying and addressing system anomalies. He also holds a Master's degree in Computer Engineering from Sapienza University of Rome, Italy. His research interests include log parsing, vulnerability detection, vulnerability patching, and anomaly detection.

Qiang Tang received his B.Sc. in Applied Mathematics from the Yantai University, China, an M.Sc. in Information Security and Cryptography from Peking University, China, and a PhD in Information Security and Cryptography from Royal Holloway, University of London, UK. In the past, he has worked as a researcher at the École Normale Supérieure (Paris), France, at the University of Twente, the Netherlands, at the University of Luxembourg, Luxembourg. He is currently a Group Leader at the Luxembourg Institute of Science and Technology (LIST), Luxembourg. He has authored many research papers in prestigious peer-reviewed international conferences and journals and participated in several national and European Research and Development projects. His main areas of research include applied cryptography, privacy enhancing technologies (PETs), and the security and privacy issues in AI and Telecommunication.