URLGEN – Towards Automatic URL Generation Using GANs

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

URLGEN– Towards Automatic URL Generation Using GANs

Rodolfo Valentim†, Idilio Drago‡, Martino Trevisan⋆, Marco Mellia†
†Politecnico di Torino, ‡Università degli Studi di Torino, ⋆Università degli Studi di Trieste
first.last@{polito,unito,dia.units}.it

***Abstract*—URLs play an essential role on the Internet, allowing access to Web resources. Automatically generating URLs is helpful in various tasks, such as application debugging, API testing, and blocklist creation for security applications. Current testing suites deeply embed experts' domain knowledge to generate suitable URLs, resulting in an ad-hoc solution for each given application. These tools thus require heavy manual intervention, with the expensive coding of rules that are hard to maintain.**

We here introduce URLGEN, a system that uses Generative Adversarial Networks (GANs) to tackle the automatic URL generation problem. URLGEN is designed for web API testing and generates URL samples for an application without any system expertise, complementing the existing tools. It leverages Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) architectures, augmented by an embedding layer that simplifies the URL learning and generation process. We show that URLGEN learns to generate new valid URLs from samples of real URLs without requiring any domain knowledge and following a purely data-driven approach. We compare the GAN architecture of URLGEN against other design options and show that the LSTM architecture can better capture the correlation among URL characters, outperforming previously proposed solutions.

Finally, we show that the URLGEN approach can be extended to other scenarios, which we illustrate with two use cases, i.e., cybersquatting domain prediction and URL classification.

***Index Terms*—Web API testing, cybersecurity, generative adversarial networks**

I. Introduction

The day-to-day operation of the Internet involves the use of Uniform Resource Locators (URLs) to address pages, objects, and API-based web services. Content management systems, traffic filtering, malicious content blocking, ad-filtering, and application testing tools require knowledge about the URLs of specific resources in order to function. Web API testing, in particular, requires a high level of attention, as relatively naive mistakes can expose services, leak data, and create serious vulnerabilities.

Web application and API testing require the verification and ad-hoc search of errors by experts. Several tools enable testing of web APIs such as Fiddler [1], Postman [2] and OWASP ZAP [3]. These tools explore the space of possible parameters to check the API sanity and the correctness of the applications' responses. However, these tools require the knowledge of specialists who must create custom test patterns to stress the APIs of the specific application under test.

The research leading to these results has been funded by SmartData@PoliTO center for Big Data technologies.

Machine learning (ML) techniques have also become popular for URL management in various contexts, including cybersecurity [4], traffic management [5], and traffic classification [6]. ML models offer the ability to learn URL patterns and classify (or predict the values of) URLs. More recently, Generative Adversarial Networks (GANs) [7] have emerged as a practical approach to generating new samples given some training data. In a GAN, a *generator* generates candidates, while the *discriminator* evaluates them as real or fake. Adversarial training allows the generator to produce realistic samples, while the discriminator becomes a robust model for identifying samples of a distribution. Researchers from various fields have successfully used GANs, from image to melody and text generation [8, 9, 10].

We here propose URLGEN, an automatic solution that uses GANs to generate URLs for target web services. Our goal is to generate URLs for test services, looking for new and possibly offensive URLs. The process consists of training a model using examples of valid URLs for a target service and then applying the generator model to test the application, as shown in Fig. 1. The training is relatively simple and builds on the model's ability to automatically learn the patterns observed in the URLs and replicate them. Using the trained generator, we stress the target application with novel (generated) URLs, in a search for URLs that may trigger application errors. URLGEN is fully automated, follows a purely data-driven approach, and requires no specific domain knowledge to generate URLs for the target application.

URLGEN is able to create novel URLs that can be used for testing applications, assisting the available knowledge-based tools. We demonstrate the potential of the URL generation technique in different scenarios. Specifically, we show that URLGEN can help generate URL blocklists by producing offending patterns. In detail, the GAN generative ability is instrumental for obtaining novel URL samples in a data augmentation process. Such URLs could be applied, for example, to extend blocklists and proactively prevent abuses. We demonstrate this scenario by generating new domain names that could be abused by attackers in *domain-squatting* operations. Moreover, we show that the ensemble of URLGEN discriminators – each learned for a different URL – can also assist URL classification applications.

It is well-known that GANs have problems when applied to categorical data such as strings. This problem makes the application of GANs to URL generation more complex than, for example, image generation, where nearby pixels contain

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

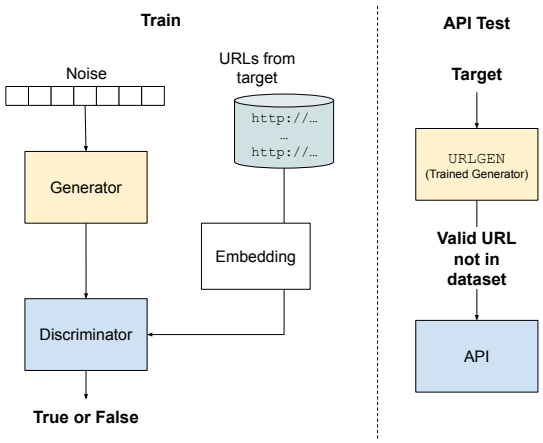


Fig. 1: Architecture of URLGEN, our application agnostic Web API testing tool, showing the training phase and the connection between the trained model and the tested application.

a significant amount of information about the pixels being generated. Therefore, we compare URLGEN architecture based on Long Short-Term Memory (LSTM) and Convolutional Neural Network (CNN) with simple fully connected architectures proposed in the past. Moreover, we add a generic embedding layer that maps characters into a vector space. This step accelerates the convergence of the GAN while improving the generative capability of the system and coping with the strict syntax and semantics of URLs.

All in all, we believe our work is an important step towards fully automated and application-independent Web API testing. However, URLGEN is not a complete replacement for existing expert-based approaches, since it may not cover *all* testing cases devised by an expert. Yet, GANs bring the dynamism missed by the rule-based approaches. To increase URLGEN impact, we offer it to the community as open-source software, as a practical tool to complement alternative approaches.[1]

This work extends [11] offering an improved GAN architecture, which better adheres to the characteristics of the URL generation problem. In detail, we add a character embedding layer that makes the model scalable and robust, and we specifically focus on the URL generation use case instead of the simple traffic classification as in [11]. Finally, we demonstrate our approach's potential for web API testing and introduce the use of GANs for domain-squatting prevention.

The remainder of the paper is organized as follows. Sec. II discusses background. Sec. III presents the proposed approach for testing Web API using GANs. Sec. IV defines our GAN architecture, and discusses model choices and training. Sec. V describes our URL datasets. Sec. VI presents results and the emerging trade-offs. We demonstrate other applications for the approach in Sec. VII. Sec VIII present related works while Sec. IX concludes the paper and present future directions.

II. Preliminaries

We here introduce some background concepts, summarizing the operation of GANs and the idea of vector representation for strings. The reader with a background in this field can skip this section.

A. Generative Adversarial Networks

Our goal is to generate realistic URL samples of a particular class given a set of examples. To this end, we rely on GANs, a model in which two neural networks compete with each other: A generative model G competes with a discriminative model D through a minimax two-player game [12]. The objectives of the models are conflicting. The generator aims at generating new samples following the same distribution of the training data, while the discriminator aims at distinguishing if a sample comes from the training data or is generated by G.

The generator G learns a probabilistic distribution p_g, which should approximate the distribution p_x of the training data x. We define an input noise variable z with probability $p_z(z)$. The generator G is a neural network that maps z to $G(z;\theta_g)$, where θ_g are the parameters of G. The discriminator D is a second neural network that learns to distinguish real samples in the training data from those generated by G. The output of $D(x;\theta_d)$ is the probability that x belongs to the training data, and θ_d are the parameters of D. We train the discriminator D to maximize the probability of identifying fake/real instances while training G to minimize $log(1-D(G(z)))$ – i.e., to bring p_g close to p_x.

B. Vector Representation

Generating URLs consists in producing sequences of discrete tokens, i.e., characters. GANs, in their simplest form, are not well-suited to work with discrete data [13] being them designed to handle continuous, real-valued data. The way the loss propagates within the model is the reason for this difficulty. The gradient update calculated at the discriminator informs the generator how to adjust its weights to generate more realistic samples. These updates rely on the assumption that after each change in the weights, the new output converges to the expected value. This assumption does not hold for discrete tokens and may introduce inconsistent mapping between tokens and gradient updates. Current solutions to overcome this limitation consider the sequence generation procedure as a sequential decision-making process and use gradient policy updates [13]. Another approach is to convert discrete tokens to continuous data using an intermediary embedding process [14, 15]. Here we rely on this second option.

In detail, we use an intermediate *vector representation* block to map each string into a sequence of vectors in continuous space as opposed to a sequence of discrete tokens. In our case, the vector representation, also called *embedding*, represents every character of the URL as a vector in an e-dimensional space where e is a hyper-parameter defining the number of dimensions of the resulting embedding. Intuitively, the more compact the space, the more information one could lose during the mapping.

The vector representation has two main advantages: (i) it properly allows gradient propagation; (ii) it allows the model to generate more realistic samples. The latter is possible by changing some characters by others in the same category

[1] URLGEN is available at https://github.com/SmartData-Polito/urlgen

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

(i.e., nearby in the embedded space), while not breaking the URL syntax. To learn the vector representation, we use the word2vec skip-gram model, proposed by Mikolov et al. [16, 17]. Introduced for natural language processing, given a sequence of words, the skip-gram model tries to predict the context words for a given target word. This approach allows for learning effective vector representations. In our case, we train a model to predict the previous and following characters given the i-th character in a string (the context). The resulting embedding maps characters that often appear in the same context in the same portion of the embedded space.

C. Web API Testing Approaches

There is no standard for Web API testing currently, and multiple ad-hoc approaches are used according to the specific application requirements. We here summarize a general set of steps usually employed in real scenarios.

The first step is to parse the available documentation. Common documentation formats are OpenAPI/Swagger, GraphQL, and WSDL. These documents contain service descriptions and are necessary to collect not only the API endpoints but also the API parameters and patterns. This task can be automatic or manual and heavily depends on the documentation quality.

Having the endpoints, an expert can start scanning the API for common vulnerabilities. This search usually starts with *fuzzing tests*, in which a testing software (often automated or semi-automated) provides random, invalid, and unexpected data as input to the API. Several tools enable such tests, such as Fiddler [1], Postman [2], and OWASP ZAP [3]. They explore the API parameter space to check the API sanity and the correctness of the application responses.

In the end, the quality of the tests depends on the API tester's know-how and on the quality of the API documentation. Conversely, `URLGEN` requires only the collection of samples of URLs to start generating possible malformed API. For this reason, automatic testing tools are important assets to automate the procedure and add stability to the process, and complement the domain experts' driven solutions.

III. `URLGEN`– Agnostic Web URL Generation

During the development of a web application that exposes APIs to the public, it is essential to have tools to test the reliability of the system. Classic testing tools require knowing the syntax of URLs, e.g., knowing possible parameters and values they may take, and then generating URLs that mix parameters and values. Volunteers and companies have to build these lists – see for example SecLists [18], Assetnote Wordlists [19], Payloads All The Things Tweet [20] and FuzzDB [21]. Such an approach is expensive and limited. Having automatic means to generate test API calls – regardless of the application-under-test – would reduce the testing costs while generalizing and increasing the test coverage.

GAN is a natural candidate solution for a task where one wants to produce new samples that follow given patterns. Compared to the domain-specific approaches, a GAN could automatically extract the characteristics of URLs/APIs of the application-under-test. To describe the approach used in `URLGEN`, we will use the `accuweather.com` API URLs described in Table I as an example.

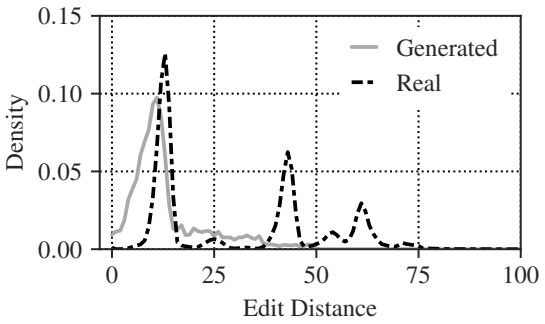


Fig. 2: Edit Distance histogram for real and generated data (without embedding).

This API consists of a fixed path that identifies the endpoint, `search.json`, the `q=` parameter that indicates the geographical coordinates (latitude and longitude) for which the client application requests information, a second parameter (`apikey=`) that is the authentication token composed of 32 characters (different for each user, here anonymized), and a third (`language=`) parameter that indicates the API language. Parameters may appear in a different order. The API returns weather information about the specific location, which can be cities or points of interest.

If we compare different valid API URLs, we observe that they may differ by one element (e.g., the same key, the same language, but different coordinates) or multiple elements (e.g., all three fields differ). We quantify such differences by measuring the Edit Distance [22][2] between pairs of URLs in a dataset containing volunteers' real requests to the `accuweather.com` API, which we will describe in detail in Section V. We show the distribution of the Edit Distance between pairs of URLs for this example in Fig. 2 with dashed lines. The multiple modes in the distribution reflect the variability of the real API requests.

Our objective is to use GANs to generate URLs, without even knowing the strict syntax of the URL and the name of the parameters. Our best model, which we will describe in detail in the next section, i) generates URLs that respect the URL syntax, ii) automatically identifies parameter names, and iii) mimic parameter values, reflecting the distance between URLs in the original training data as shown in Fig. 2. In addition, the `URLGEN` GAN-based approach still introduces errors in other parts of the URL, which is the desired behavior for API testing. For instance, `URLGEN` generates URLs with an almost constant and valid API key, learned from the majority of the URLs present in the training data. However, it sometimes injects errors in the key, and even in the parameters' names.

To visualize the quality of URLs generated by `URLGEN`, Fig. 3 shows in a map the generated coordinates along with samples of the original data used to train `URLGEN` in this

[2]The edit distance between two strings s_1, s_2 measures the minimum number of edits (additions, removals, replacements) that would be required to transform s_1 into s_2.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

TABLE I: URL samples of the Accuweather API calls.

Base URL	Coordinate	API Key	Language
api.accuweather.com/locations/v1/cities/geoposition/search.json?	q=45.372202,8.1753055	apikey=ANONYMIZED	language=it
api.accuweather.com/locations/v1/cities/geoposition/search.json?	q=45.3022434,8.4490561	apikey=ANONYMIZED	language=it
api.accuweather.com/locations/v1/cities/geoposition/search.json?	q=46.0189849,8.2591772	apikey=ANONYMIZED	language=en

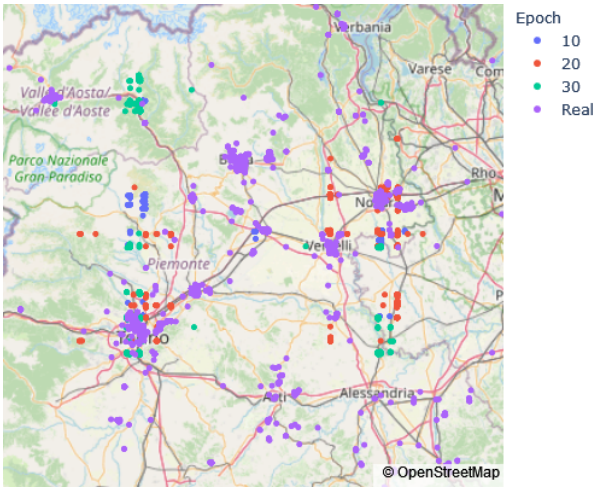


Fig. 3: Map showing coordinates in valid URLs generated by the trained model.

example. Considering a total of 1 000 generated URLs, about 300 URLs are unique. Of these unique URLs, 60% results in valid samples – valid URLs are those that lead to a valid response from the service.[3] None of the valid URLs is present in the training data. Interestingly, most generated URLs are formed by coordinates falling in the same area of points in the training data. While the training proceeds, the generated URLs vary as shown by samples taken at different training epochs (see the dot colors).

This example illustrates the benefits of the GAN-based approach in the context of Web API testing. Without exploiting any domain knowledge on the application-under-test, the GAN automatically learns to generate both unseen but valid samples (e.g., as in Fig. 3), as well as invalid samples that resemble the real ones (e.g., URLs with wrong API keys), thus useful for testing invalid API requests.

Next, we dig into the design of URLGEN, exploring different models and the challenges one faces to build such a GAN-based approach.

IV. GAN for URL Generation

We now discuss the methodology used to compose the architecture of URLGEN. Given a dataset of URLs belonging to a given class (e.g., URLs belonging to *site.com*), we train a GAN where the discriminator network shall distinguish real samples from samples generated by the adversary network. URLGEN must be trained for each application-under-test (i.e., class of URLs). We build URLGEN using recurrent and convolutional neural networks, which are known to be very good architectures for this task. Later in Section VI-D we compare with possible alternatives.

We design our system with an intermediate embedding layer to transform each character from a traditional one-hot encoding into a vector representation. When used, the GAN with the embedding layer operates on sequences of embedded vectors, both in the generator and in the discriminator models. We start by discussing how to represent URLs using such an embedding layer.

A. URL Representation

URLs are encoded using 50 of the American Standard Code for Information Interchange (ASCII) Character Set [23]. Additional characters not present in the original set may be encoded by a triplet consisting of the character "%" followed by the two hexadecimal digits. The HTTP protocol does not limit URL size, and servers are required to handle any URL independently of the length. However, RFC 7230 [24] suggests that URLs should respect a limit length of 2 000 characters, and in fact, most browsers do not accept entries above this recommendation. Given these constraints, we remove all unused characters resulting in a URL representation set composed of 50^{2000} possible sequences.

In text classification, each sentence can be broken into words and, later, each word is converted to tokens that feed the model, e.g., via the one-hot-encoding mechanism. In URLs, this task is not efficient because the structure of a URL is too diverse when compared to sentences in natural language. Website names and links might not respect grammar rules. The domain names, paths, attributes, and parameters might be composed of letters, numbers, and special characters with a strict syntax, but loose semantics.

Because of this, we generate and discriminate URLs at the character level. This decision reduces the vocabulary size required for the *tokenization* and avoids retraining when new words are introduced to the vocabulary. This approach is in common with other works that tackle URL representation with machine learning [4, 25]. On the downside, by analyzing each character independently, we may lose the contextual information in the sequences of characters that compose the URL. To alleviate the problem, we use an embedding layer that encodes part of the contextual information. In addition, we use a generator architecture based on the Long Short-Term Memory (LSTM) network, described in Section IV-C, which is a neural network architecture aware of the context of each token.

[3] We use regular expressions and manual inspection to classify URLs.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

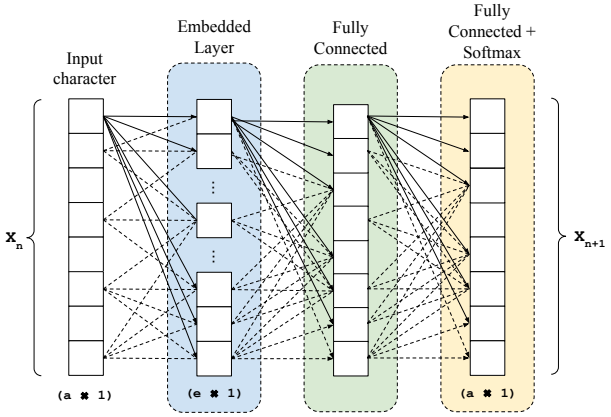


Fig. 4: Network used to train the embedding. The embedding vectors are extracted from the Hidden Units. We use a soft-max layer to predict the next character and evaluate the model. a is the alphabet size and e the embedding size.

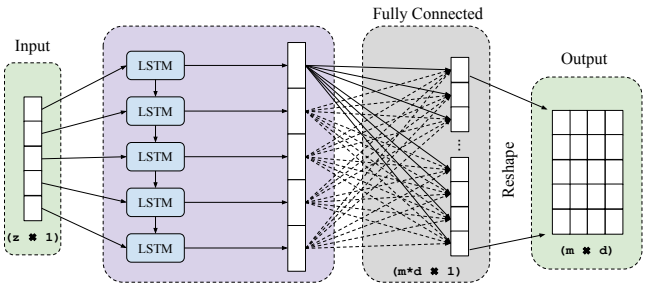


Fig. 5: Generator model - It receives a noise vector and generates a URL in the embedded space. z is the noise size; m is the maximum number of URL characters; and d is the output dimension, which depends on the use or not of embedding.

B. Learning the Embedding

We train an embedding layer to represent the 50 ASCII characters admitted in URLs as e-dimensional vectors.[4] We build on the skip-gram model to learn the mapping from each character to a vector. We depict the training process in Fig. 4. The first layer maps characters to vectors using a standard one-hot encoding map. Each character in the input is embedded in a e-dimensional space from the initial $a = 50$ dimension space. The fully connected layer afterward uses a soft-max to obtain output probabilities that are chosen to predict the next character in the URL. In a nutshell, for every character x_n, we train the network to predict the next character x_{n+1} by projecting the initial space of a dimensions into an embedding of $e < a$ dimensions.

We train the embedding layer using URLs that cover multiple services and APIs. Indeed, the embedding built with generic URLs will reflect the typical URL characteristics, thus learning semantics that is generic and applicable to multiple use cases. To map embedded URLs back to the one-hot encoding ones, we multiply the embedded URL by the Moore-Penrose pseudo-inverse of the embedding matrix [26]. Then, using a dictionary, we convert the obtained one-hot encoding to ASCII characters.

C. GAN Architecture

The choice for the discriminator D and generator G architectures is not straightforward in our case. Intuitively, we need a generator that is able to reconstruct possible URLs on a character-by-character basis; and a discriminator that can compare strings in which the sequence of parameters may be altered on purpose by the generator.

In general, the generator G receives a random input z and maps it to a $G(z)$ that shall follow the real data distribution. The noise is a k-length vector and each element is sampled from a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 1$. The generated output is $\tilde{x} \rightarrow \tilde{X}_e \in R^{m \times d}$, where m is the maximum URL length and d is the dimension of the encoding. The dimension can be e when the embedding is used or a for the alphabet size in the one-hot encoding alternative. `URLGEN` supports both cases. The discriminator D receives an URL in the encoded space, and $D(x_e; \theta_d)$ outputs a scalar in the interval $[0, 1]$.

In the following, we describe the neural networks we employ for the generator G and discriminator D.

1) Generator: Our previous work [11] used a fully connected neural network (FCCN) where all the neurons in one layer are connected to the neurons in the next layer through a dropout regularization. In this architecture, each character is generated independently of the others in the sequence. We expect this independence to limit the generator's capability of producing realistic samples as each character will be drawn based solely on its position in the URL and independently on the previous and next characters in the string. In addition, FCNNs have a very high number of hyper-parameters, as each neuron holds independent weights, making the training more complex.

`URLGEN` instead relies on Recurrent Neural Networks (RNNs). RNNs are well-suited to operate on sequential data, which is the case of URLs if interpreted as sequences of characters. Among the many variants of RNNs, we select the Long Short-Term Memory (LSTM) [27] units. An LSTM unit is an element able to remember information for arbitrary long intervals and use this memory to generate the output. An LSTM cell has three gates: input, forget, and output. These gates govern whether or not to allow new input in, forget old information, and affect output at the current time step, respectively. Famous applications of LSTMs are in the field of Natural Language Processing (NLP), such as text generation [28, 29] and classification [30, 31] and chat bots [32].

We depict the generator architecture in Fig. 5. As a consequence of its design, the LSTM provides an output at every time step, which depends on the input and on previous states. Therefore, after the LSTM unit, we use a fully connected layer that shapes the output to an encoded URL $\tilde{x} \rightarrow \tilde{X}_e \in R^{m \times d}$.

2) Discriminator: In [11] we used again a fully connected neural network (FCNN) for the discriminator (D) because it was symmetric with respect to the generator. However even

[4]Here we consider lowercase characters to reduce the complexity. The extension to the case-sensitive case is straightforward.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

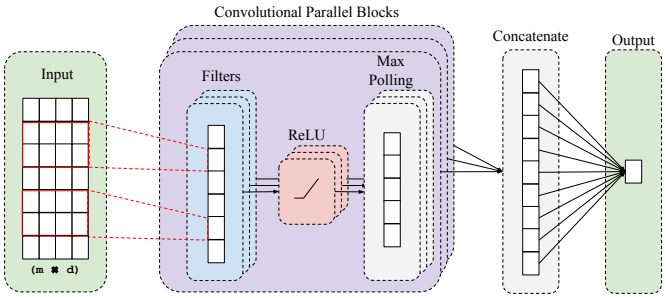


Fig. 6: Discriminator model - It receives embedded URLs and outputs the probability the URL comes from real samples. m is the maximum number of URL characters; d is the output dimension, which depends on the use or not of embedding.

though FCNNs are sufficient to discriminate simple URLs, they are not resilient to shifts in the input, having independent weights for each neuron. Conversely, we desire URL classification that is robust to input shifts, as a small variation in the URL prefix can shift all the remaining characters. Also, URL parameters can have different orders at different samples. For instance, *www.site.com* and *www1.site.com*, or *site.com?a=b&c=d* and *site.com?c=d&a=b* would result totally different for an FCNN discriminator.

To address these shortcomings, URLGEN includes Convolutional Neural Networks (CNNs) for D. CNNs have a better ability to detect features with varying positions in the data, and thus have the desired robustness to shifts. CNNs have been originally proposed for image classification [33] and for various tasks of computer vision [34, 35]. Also, they are used in text [36, 37] and URL classification [4, 38, 39].

Fig. 6 shows the 1-dimensional CNN we use for D over the sequence of characters of the input URL. The size of the kernel defines the number of characters the convolutional unit analyses at each iteration. Following the convolution, there is a LeakyReLU activation layer and a unidimensional max pooling to reduce the feature dimension and identify the most important features. We consider different sizes for the kernel so that the convolution can process n-grams of different sizes. Thus, we perform several convolutional operations and concatenate the max-polling outputs. Finally, a fully connected layer reduces the output to the single output scalar.

D. Training the GAN

Training GANs is complicated by possible model collapse and convergence failure. Given that we combine different types of neural network architectures with different learning speeds, the training process might easily fall into convergence failure – e.g., one of the two models prevails. We discuss how to control this issue when presenting our results in Section VI-C.

Moreover, recall that we target the creation of a GAN for each class of URLs, i.e., a specific service. This makes the task of the discriminator simpler, as it can learn specific patterns present on the real URLs. To produce robust discriminators, we include some unbiased noise – i.e. a negative dataset – during training, to make the discriminator less specific. This negative dataset contains some random URL samples, extracted from other services, which are different from the target URLs. The goal of this modification is to induce the discriminator to recall not only the samples from the target URL class but also to differentiate these samples from other samples.

The training process is divided into the following steps. We first update the discriminator model weights using the binary cross-entropy loss calculated over (i) real class samples; (ii) generate samples; (iii) negative samples. Then, we update the generator model weights with the loss calculated as $(1 - D(G(z)))$. Minimizing this loss means that the generator creates samples that – for the discriminator – are indistinguishable from the real class samples.

We use a batch size of 64, with half a batch of real samples and half a batch of generated samples. We use the Adam optimizer with $LR = 0.0005$, $\beta_1 = 0.9$, $\beta_2 = 0.98$, $\epsilon = 10^{-9}$.

We implement URLGEN using Python Keras with the TensorFlow backend. We run all experiments using a high-end server with 1 Nvidia Tesla-V100 GPU (16 GB GPU memory). The running time for each epoch is no longer than a few seconds. We usually run the training for 10 epochs. As such, the training for a single service is thus lightweight and can be completed with off-the-shelf hardware.

TABLE II: URL classes.

ID	Hostname	Type of Service
0	api.accuweather.com	Geo API
1	mmsns.qpic.cn	Image Repository
2	pcdn.any.sky.it	Video Streaming
3	tlu.dl.delivery	Software Updates
4	windowsupdate.com	Software Updates
5	b.scorecardresearch.com	Web Tracking
6	b2everyrai-lh.akamaihd.net	Video Streaming
7	gs2.ww.prod.dl.playstation.net	Gaming
8	la7livehls-lh.akamaihd.net	Video Streaming
9	su.ff.avast.com	Software Updates
10	t.nyaatracker.com	Torrent Tracker
11	Negative dataset	Mixed

V. Dataset

We use real URLs of different services to evaluate URLGEN. We leverage URLs collected by a set of more than 500 volunteers that we recruited among participants on social networks, specialized forums, and in our Universities. Volunteers installed and kept active a MiTM proxy on their PCs for at least one month. The MiTM proxy logs all URLs they visit and uploads this data to a centralized repository on our campus. The volunteers have explicitly approved our data collection and they could opt out at any time and impose filtering rules at their will. Using the MiTM proxy, we gain visibility on all URLs accessed by the volunteers, regardless of whether they are served via HTTP or HTTPS.

The dataset contains millions of URLs from thousands of websites. Among them, we select 11 classes of URLs belonging to specific services that we use to study the effectiveness of our approach. Our goal is to consider diverse use cases, from simple API-style URLs to complex services with a multitude of parameters in the URLs.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

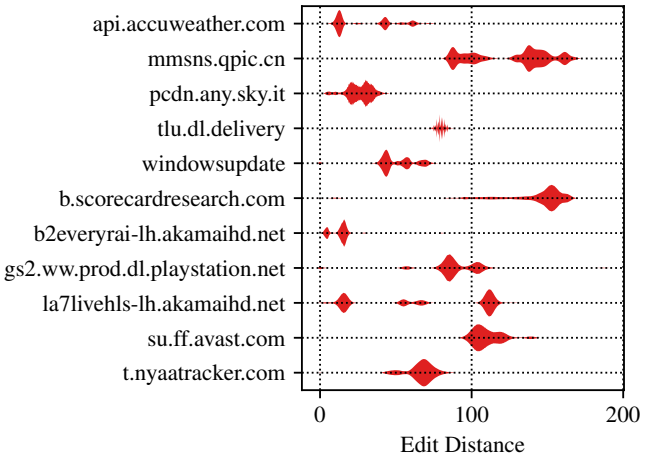


Fig. 7: Edit Distance density between all pairs of URLs for each class. Each peak represents a dataset mode. The maximum length is 200 characters.

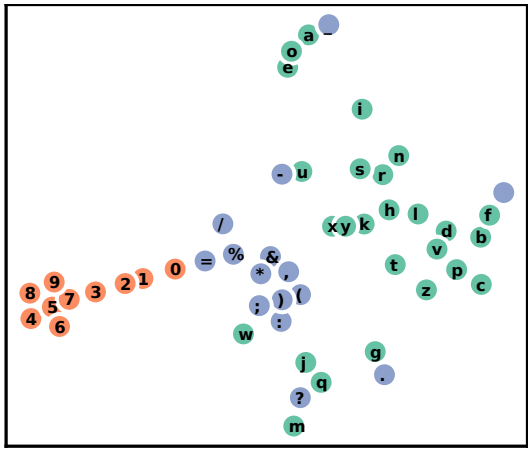


Fig. 8: tSNE projection in two dimensions of the assigned vector to each character allowed in a URL. We use different colors to highlight the difference between letters, digits, and special characters.

In Table II we summarize the 11 URL classes. For each class, we randomly pick 10 000 samples. They belong to various services that present URLs with a particular structure that we aim to recreate. Besides the `accuweather.com` we used as an example, there are three popular video streaming services. These APIs encode in the URL the name of the TV channel, the timestamp of the requested chunk, and the video quality for multiple-resolution videos. Software update services instead require the URL to specify the requested archive using a fixed-size digest. Tracking, gaming, and image repositories complete our target services, each having very complex URLs. We complete our dataset with 10 000 random URLs not belonging to any class that we use to add unbiased noise during the training process – i.e., the negative dataset.

To illustrate the variability of the URLs in our dataset Fig. 7 presents the distribution of the Edit Distance for each class (cfr. Fig. 2). We compute the Edit Distance between all pairs of samples for each class and plot the Probability Distribution Function using violin plots.

Prominent peaks represent the relevant modes for the URLs in each class. Indeed, URLs within a class are not necessarily homogeneous, and we can find some sub-groups as seen for *api.accuweather.com*. The peaks in Fig. 7 reflect the subgroups of URLs/APIs of each group. Most services result in multi-modal distributions, reflecting pairs of URLs that belong to different subgroups. The next section verifies whether `URLGEN` can reproduce this behavior for several classes.

VI. Evaluation

We now evaluate the performance of `URLGEN`. First, we illustrate how the embedding layer is formed and how it impacts performance. The evaluation of the generative abilities of a GAN is a complex task and often requires human interaction. Here we evaluate the generator performance by quantifying how the generated URLs are similar to the training examples. Finally, we evaluate the impact of model parameters and alternative models.

A. Character Embedding

We first show how the embedding layer organizes the characters in the embedded space. Recall that we create the embedding by training the neural network to predict the next character in a string. We use all URLs in our dataset as input strings so to generate a generic embedded space. The final embedding maps each character to an e-dimensional latent space: Characters that often appear close get mapped into the same portion of the latent space.

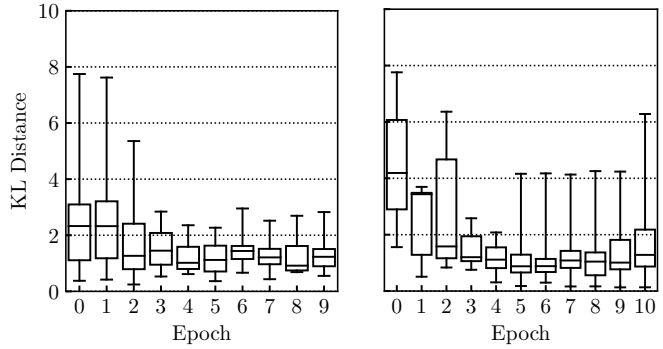
To illustrate the embedded space, we use a t-Distributed Stochastic Neighbor Embedding (tSNE) [40] projection of the conversion matrix between the original character space and the embedding space. The result is in Fig. 8. We use different colors to represent digits, letters, and special characters. The results clearly show clusters of characters, with digits, letters, and special characters typically placed in a different portion of the projected space. Given the peculiarities of URLs, we notice that some characters are placed nearby each other, e.g., `/` and `:` or `%` are placed close to digits due to, e.g., frequent hexadecimal encoding for special characters in URLs. Intuitively, we expect the embedding to favor the generation of URLs by choosing characters that appear to be close with higher probability than characters that appear far away.

B. Generation

As illustrated in our toy example in Section III, we compute the Edit distance between all pairs of URLs (i) in the training dataset; and (ii) in the generated URLs. We compute the probability density function of the obtained edit distances. These curves allow us to compare the characteristics of the URLs present in the training dataset against the generated ones.

For evaluating the generation procedure we rely on the visual similarity of the two curves, for real and generated samples. For this, we compute the Kullback-Leibler (KL) divergence [41] between the generated and real curves as a

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311



(a) With embedding (b) Without embedding

Fig. 9: Kullback–Leibler divergence between Edit Distance density curves for all datasets and each architecture.

similarity metric.[5] The KL divergence interval is $[0, \inf]$, and the closer to zero, the more similar the two distributions are. By calculating this metric over different training epochs, we measure how the generator mimics the real distribution during training.

Fig. 9a and 9b show the KL divergence evolution during the training process (on the x-axis). For each class and epoch, we generate 1 000 URLs, compute the Edit distance between all pairs and measure the KL divergence from the distribution of the edit distance among training samples (cfr. Fig. 7). We thus get 11 measures, one for each class, and we plot them using boxplots. The central stroke represents the median class (the sixth in our case), and the box spans from the third to the ninth class ranked by KL divergence. The two whiskers represent the class with minimum and maximum KL divergence, respectively.

Both Fig. 9a and 9b show that the GANs produce URLs that become more similar to the real ones as the training advances in all cases. The KL divergence metric, on average, improves with training. Further proceeding with training leads to marginal improvements in performance. We observe better and more stable convergence of the distributions when the embedding layer is present. These results confirm that offering the GAN a more structured latent space helps convergence. In a nutshell, results confirm that the LSTM architecture captures the string patterns, producing URLs that mimic structures in the real samples.

To provide a deeper analysis of the generation process and show how generated URLs converge towards the real ones, Fig. 10 compares the distribution of the Edit Distance over epochs. We consider three classes. The solid line represents the generated data while the black dashed line represents the distribution of the original URLs that the GAN shall learn to recreate.

We notice that, as we advance the training, we begin to see sharper peaks in the distributions of generated URLs that get more similar to the training URL curves. In the cases where the training dataset contains only one mode, the distributions of real and generated URLs mostly overlap – see *scorecardresearch.com* for instance. When the training dataset contains multiple modes with multiple peaks in the distribution, the generated URLs tend to follow either the average among the peaks – see *accuweather.com* – or the strongest mode – see *qpic.cn*.

In the case of *api.accuweather.com* (Fig. 2), the GAN without embedding focuses on the generation of some URL parameters (e.g., the coordinates of the location of the weather query) while keeping fixed other parts of the URL (e.g., the API key).

Although this behavior can be desirable in some cases, it is a limitation of the GAN when coupled with this embedding layer. For completeness, recall Fig. 10 that shows the example of generation with embedding for *accuweather.com*. Compare Fig. 10 b). with Fig. 2 and notice that in the first the generator tends to replicate only the most predominant mode. While in the latter the embedding helps the GAN model often mix the multiple sub-classes in the training data.

[5]We take inspiration from [42], which shows that, under some assumptions, GANs minimize the Jensen-Shannon (JS) divergence, or with a slight modification the reverse-KL divergence between the real and generated distributions of the target variable.

TABLE III: Model Variables.

Variable	**Suggested values**	Used Values
Noise Dimension (Z)	10 >= Z <= 20	10
Maximum URL length (m)	[100, 200]	200
Embedding size (e)	[30,50]	30
Alphabet size (a)	50	50

TABLE IV: Convolutional Hyperparameters.

Hyperparameter	**Used values**
Parallel Blocks	6
Kernel Values (one for each branch)	(2, 3, 4, 5, 6, 7)
Pooling size (a)	2
Leaning rate	0.0005

C. Complexity and Parameter Impact

One of the drawbacks of GANs and NN in general is the large set of hyper-parameters that one could explore to optimize performance. This drawback clearly turns into much complexity and requires CPU time for training. Here we summarize our suggested ranges for the hyper-parameter choice in Tables III, IV and V. For some parameters, we rely on default values, while for others we run a sensitivity analysis to give coarse suggestions.

For the creation of the embedding, which is mostly instrumental for simplifying the GAN training and convergence, we test different values of e, and pick the smallest one to reduce the embedding space dimensions.

TABLE V: LSTM hyperparameters.

Hyperparameter	**Used values**
LSTM Units	1000
Dropout	0.6
Leaning rate	0.0005

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

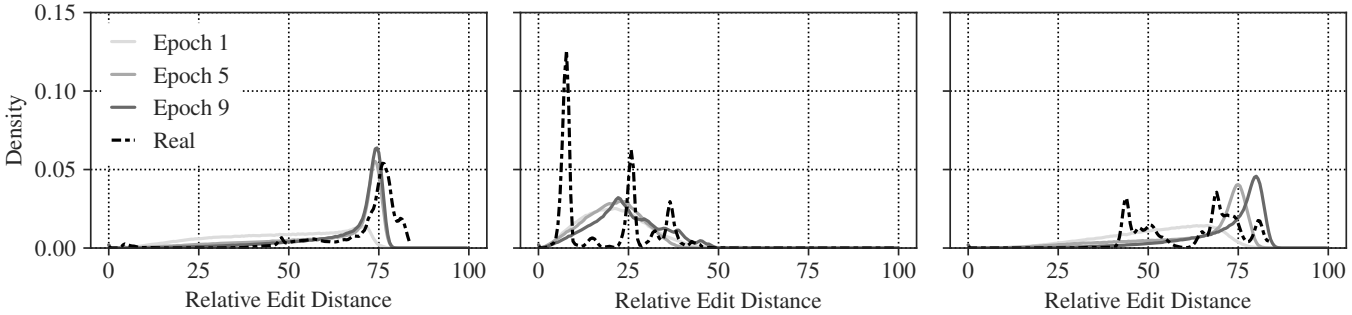


Fig. 10: Edit Distance density for three classes at different epochs during training – URLGEN using a GAN with embedding.

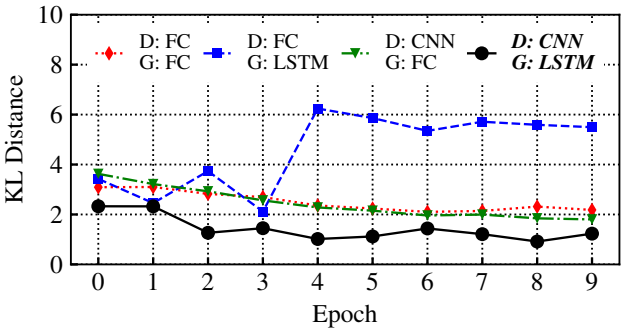


Fig. 11: Median Kullback–Leibler divergence between Edit Distance density curves for different architectures. The black solid line represents the URLGEN architecture.

Conversely, the training of the generator requires careful parameter tuning to guarantee and speed up the convergence. We use the training process proposed by Arjovsky et al. [43]. The authors introduce a new hyper-parameter that controls how fast the discriminator learns in comparison with the generator. After some tuning, and following the best practice, we observe the best results with a proportion of 4:1:1 for training the discriminator with real samples, the discriminator with negative samples, and the generator, respectively.

D. Architecture Comparison

In URLGEN, the generator model uses an LSTM network, while the discriminator adopts a CNN. Here we provide a comparison with possible alternative architectures.

In particular, we compare the URLGEN architecture with the simple one we used in our previous work [11]. There, we proposed the usage of fully connected neural networks with 3 layers for both generator and discriminator, where all the neurons in one layer are connected to the neurons in the next layer through a dropout regularization and the hyperbolic tangent activation function. The layers have different sizes for each block. We consider this as a baseline model to compare against. With this architecture, each character is generated independently of the others in the sequence. We expect this independence to limit the generator's capability of producing URLs, and the discriminator's to be limited in distinguishing generated samples as each character will be compared based only on its position. In URLGEN, LSTM and CNN architectures instead are able to leverage the information of the previous and surrounding characters in the string.

For the sake of completeness, we compare URLGEN also with two architectures in which we change only the generator or only the discriminator. In total, we consider four possible architectures for the generator and the discriminator: (i) G:LSTM D:CNN (i.e., URLGEN), (ii) G:FCN D:FCN, (iii) G:FCN D:CNN, and (iv) G:LSTM D:FCN.[6] Not reported here, we optimized the hyperparameter of each architecture as detailed in Sec. VI-C for the URLGEN architecture. We report the best results for each architecture.

In Fig.11, we show the KL distance between generated and real samples over the training epochs. Different lines refer to different architectures, while the y axis reports the median of the KL distance among the classes of our dataset. The architecture chosen for URLGEN (solid black line) converges faster than the others and settles to lower values of KL distance, meaning generated URLs have similar distributions as real ones. Architectures adopting a Fully Connected generator (red and green dashed line) arrive at convergence, even if slower than URLGEN and with worse KL distance values. Conversely, the architecture with the LSTM generator and Fully Connected discriminator (blue line) does not converge, and the generation capacity to represent the training URL class distribution almost completely disappears after a few epochs. Our intuition is that the LSTM architecture cannot compete in the GAN's two-player zero-sum game in this case.

E. Final considerations

As we have seen, the choice of a CNN network as a discriminative model and the LSTM network as a generative model has many advantages compared with fully connected networks. These results stem from the intuition that the URL syntax and formatting rules can be captured by an architecture that considers the context of character appearance and not only

[6]We also tested other combinations. As expected we observe those GANs using LSTM for discriminator tend to quickly collapse (typically during the first training epoch), while using a CNN as the generator brings useless complexity.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

their position. The usage of the embedding layer lets the GAN work in a continuous space where frequently co-occurring characters are mapped into the same portion of the space. The embedding layer is also instrumental in stabilizing the discriminator. The embedding speeds up the GAN convergence and opens the door to other applications that we illustrate in the next section.

VII. Use Cases

While URLGEN is designed for API testing, we argue the approach it relies upon can be applied in other scenarios. Here we illustrate two other use cases. We first show how URLGEN can be used to generate new samples of *cybersquatting* domains. We then show how the discriminator model can support a URL classifier.

A. Domain Squatting

Phishing is a cyberattack in which the attacker tries to convince victims to reveal personal information through fraudulent messages. With *cybersquatting*, the attacker registers Internet domains similar to legitimate services to fool the victims in a phishing attempt. The typical defenses consist of blocklists composed of squatting domains whose timely collection and update are key to blocking the attack. The management of these lists is time-consuming and often based on human intervention, posing scalability and economic issues. Moreover, they are inefficient against zero-day attacks.

Researchers already investigated methodologies to tackle the problem. For example, Tian et al. [44] search and detect squatting phishing domains in the wild using deterministic tools that enumerate possible domains using their expert knowledge. They generate more than $600\,000$ squatting domain candidates, about $1\,000$ of which have been confirmed to be used for phishing that evaded blocklists.

Here we propose the use of URLGEN to generate domain squatting candidates, focusing on typo-squatting.[7] Our rationale is that GANs can learn data distributions and generate new samples from a target domain, reproducing some squatting techniques. We are not the first to suggest the use of generative models to augment data used in cybersquatting identification systems [46]. However, the previous work builds on image generation, where the domain is converted to an image, and a GAN generates new images that eventually are used to train homograph phishing identifier systems. We instead directly aim at generating phishing URLs as strings.

We focus on three classes of URLs, training a GAN for each one. We do use the embedding layer in this case, which we modify to inject variations in the data generation. These variations are responsible for the typo-squatting candidates. We thus "move" those characters typically abused for cybersquatting so that they are close in the embedded space. For instance, a "0" is close to a "o" in our modified embedded space.[8] As other examples, we move the pairs $(5, s)$ and $(1, l)$ so that they are closer compared to other pairs in the embedded space. We let URLGEN generate $1,000$ URLs, from which we extract only the domain part. Next, we evaluate whether these generated domains could be realistic cybersquatting candidates using manual inspection supported by some automatic regular expression checks and the edit distance values between the original domain and the generated candidates.

TABLE VI: Samples of generated data for 3 brands.

Examples	Unique samples
ap1.accuveather.c0m api.accuweathet.com	21
micro50ft.com m1cros0ft.com	19
dovnload.w1ndow5update.c0m downl0ad.w1ndow5update.c0m	489

Table VI reports some examples of cybersquatting domains URLGEN generates. URLGEN creates a limited number of unique domains. Among these candidates, we have found 2 registered domains for `api.accuweather.com`, one of which is likely to be phishing (determined by manual inspection of the website). Similarly, we found 4 existing domains for `microsoft.com` of which 3 appear to be phishing. None of these domains are present in public blocklists at the time of writing.

In a nutshell, the generation abilities of URLGEN, coupled with the flexibility offered by the embedding layer, allows us to obtain likely squatting domains with almost no domain knowledge and at low costs when compared to the exhaustive enumeration of all derived strings [44]. This experiment illustrates the possibility of using a GAN to produce typo-squatting candidates for target brands and including them into blocklists even before abuses, thus changing the reactive nature of the phishing blocklists to a proactive search, anticipating new attacks.

B. URL Classification

Since URLGEN uses negative samples during the training of the discriminator, the obtained discriminator turns out to be a powerful URL identifier. Combining discriminators trained for multiple services results in a good URL classifier as a side-product of the GANs. Notice that, by construction, a discriminator should not be able to distinguish samples generated by its adversarial generator from the real samples for the given URL class. Yet, the negative samples lead to a discriminator that can refute URLs belonging to other classes.

Using the 11 classes in our dataset, we set up the following multi-class classification problem. For each class, we use $9\,000$ URLs to train a GAN for the given class, and separate the remaining $1\,000$ for a (mixed) test set. To assign a class label to a URL in the complete test set, we compare the scores obtained by the 11 discriminators and assign the class with the largest score to the sample. If no score exceeds the threshold of 0.5, we classify the URL as belonging to the "Other" class. To check possible false positives, we include $1\,000$ URLs not belonging to any class from our *negative* dataset to the test set.

[7] Typo-squatting is the intentional registration of misspellings of popular website addresses. This attack was first described in [45] and it is still a very effective type of scamming technique.

[8] Such vector representation can be learned from data to include errors humans make with captcha or in textual documents. Exploring such approaches will be the center of our future work.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

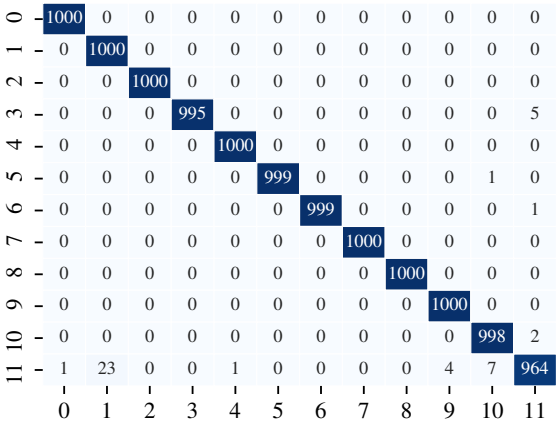


Fig. 12: Classification results for each class and model.

The ensemble of discriminators shall classify these samples as "Others".

We show the performance of this classifier built with the ensemble of discriminators reporting the complete confusion matrix. Each row represents a given class (true label), and each column represents the assigned classes. Cells on the diagonal show the number of samples that are correctly classified, i.e., the true positives; cells outside the main diagonal account for wrong classifications.

As shown in Fig. 12, the performance is almost perfect, missing only a few samples. In a nutshell, the combination of multiple URLGEN discriminators can effectively be used as classifiers for URLs. This is in line with previous findings [4].

VIII. Related Work

We are aware of two works that apply GANs for cybersecurity, focusing on the generation of domains. Degani et al. [47] use GANs to perform subdomain enumeration. They use GANs to learn the distribution of publicly available datasets, employing the model to enumerate subdomains for target domains. Authors claim to improve the quality of the enumerator when compared to deterministic approaches. They propose to add such a data-driven method to existing workflows, thus assisting deterministic methods.

Authors of [48] propose PhishGAN to augment and identify homoglyph phishing attacks. PhishGAN relies on the imprecision of image reconstruction for building a conditional GAN network. They convert the domain to an image and reconstruct the image using a conditional GAN, which introduces some noise to the output image. Later, they use their reconstructed image to train a homoglyph identifier system and augment their datasets.

Within the field of computer networks, Lin et al. [49] use GANs to generate time series of network measurements, such as bandwidth measurements or web sessions. Yin et al. [50] use GANs to enhance existing models for botnet detection, while Lin et al. [51] generate malicious traffic records aiming to evade the detection at IDSs. In a similar direction, Charlier et al. [52] propose SynGANs to generate malicious flow mutations and improve IDS attack detection effectiveness, while Usama et al. [53] use an equivalent approach to check how GANs can be used to evade ML-based IDSs.

Ring et al. [54] evaluate different GAN-based approaches to generate flow records, a challenging task given the presence of categorical features such as IP addresses and port numbers. Similarly, Cheng et al. [55] use Convolutional Neural Network GANs to generate network traffic at the packet level, including ICMP, DNS, and HTTP flows.

In our preliminary work [11], we show how simple, fully connected GANs are effective in generating realistic URLs given a limited set of examples. Here, we dig deeper into this direction, evaluating the impact of different GAN architectures and introducing the use of an embedding layer. We contribute URLGEN for automated Web API testing. Moreover, we show how GANs can produce new realistic URL samples with different applications in cybersecurity.

We leverage the vector representation technique *word2vec*, proposed by Mikolov et al. [17]. The intuition is that the embedding produced by analyzing the sequence of characters results in a more compact representation of URLs than a classic one-hot encoding. These results are beneficial to speed up GAN convergence and increase its accuracy in URL generation. Word2vec has already been used for natural language processing [56, 57, 58], in the context of programming code [59] and mobile app classification [60]. In computer networks, vector representation has been proposed for grouping domain names [61, 62], cluster darknet traffic [63] and, again, to improve IDS performance [64, 65]. Similar to our work, Le et al. [4] use a vector representation of characters to classify URLs with convolutional architectures. However, the final goal is different, as we aim to generate new URL samples given a limited set of examples.

IX. Conclusion

We introduced URLGEN, a system to generate URLs for web API testing. URLGEN relies on GANs to automatically learn URL patterns as highly structured sequences of characters. We use an embedding layer to model the characters in an informative latent space. Then we employ a CNN discriminator and an LSTM generator in a typical adversarial training setup. The generator shows excellent capabilities in generating strings that mimic the strict syntax of URLs, opening new doors to the automatic testing of applications with little domain knowledge. Moreover, we showed other use cases in which URLGEN delivers promising results. For example, our experiments with an *ad-hoc* embedding show that it is possible to use URLGEN to automatically generate plausible cybersquatting domains at a minimal cost.

Our future research will explore the use of more complex network architectures or the stacking of multiple and diverse layers to face complex web API structures. Indeed, in this work, we showed that GANs can generate URLs with a well-defined structure, but we left out more complex cases, e.g., web APIs receiving posted parameters of complex types. We will also explore the other use cases mentioned in the paper, in particular the application of generative models to other types of cybersquatting attacks.

REFERENCES

[1] Progress Software Corporation, "Telerik fiddler."

[2] Postman, Inc., "Postman api."

[3] ZAP Dev Team, "Owasp zed attack proxy (zap)."

[4] H. Le, Q. Pham, D. Sahoo, and S. C. Hoi, "Urlnet: Learning a url representation with deep learning for malicious url detection," *arXiv preprint arXiv:1802.03162*, 2018.

[5] A. Morichetta and M. Mellia, "Lenta: Longitudinal exploration for network traffic analysis from passive data," *IEEE Transactions on Network and Service Management*, vol. 16, no. 3, pp. 814–827, 2019.

[6] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé, "Mobile encrypted traffic classification using deep learning: Experimental evaluation, lessons learned, and challenges," *IEEE Transactions on Network and Service Management*, vol. 16, no. 2, pp. 445–458, 2019.

[7] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," *Advances in neural information processing systems*, vol. 27, 2014.

[8] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Processing Magazine*, vol. 35, no. 1, pp. 53–65, 2018.

[9] J. Gui, Z. Sun, Y. Wen, D. Tao, and J. Ye, "A review on generative adversarial networks: Algorithms, theory, and applications," *arXiv preprint arXiv:2001.06937*, 2020.

[10] I. K. Dutta, B. Ghosh, and A. Carlson, "Generative Adversarial Networks in Security : A Survey," *2020 IEEE 11th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference*, 2020.

[11] M. Trevisan and I. Drago, "Robust url classification with generative adversarial networks," *ACM SIGMETRICS Performance Evaluation Review*, vol. 46, no. 3, pp. 143–146, 2019.

[12] R. B. Myerson, *Game theory*. Harvard university press, 2013.

[13] L. Yu, W. Zhang, J. Wang, and Y. Yu, "Seqgan: Sequence generative adversarial nets with policy gradient," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 31, 2017.

[14] W. Nie, N. Narodytska, and A. Patel, "Relgan: Relational generative adversarial networks for text generation," in *International conference on learning representations*, 2018.

[15] A. Budhkar, K. Vishnubhotla, S. Hossain, and F. Rudzicz, "Generative adversarial networks for text using word2vec intermediaries," *arXiv preprint arXiv:1904.02293*, 2019.

[16] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[17] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," *arXiv preprint arXiv:1310.4546*, 2013.

[18] D. Miessler, J. Haddix, and g0tmi1k, "Seclists." https://github.com/danielmiessler/SecLists, 2021.

[19] assetnote.io, "Assetnote wordlists." https://wordlists.assetnote.io/, 2021.

[20] "Payloads all the things tweet." https://github.com/swisskyrepo/PayloadsAllTheThings, 2021.

[21] "Fuzzdb." https://github.com/fuzzdb-project/fuzzdb, 2021.

[22] W. W. Cohen, P. Ravikumar, S. E. Fienberg, *et al.*, "A comparison of string distance metrics for name-matching tasks.," in *IIWeb*, vol. 3, pp. 73–78, 2003.

[23] T. Berners-Lee, L. Masinter, and M. McCahill, "Rfc1738: Uniform resource locators (url)," 1994.

[24] R. Fielding and J. Reschke, "Hypertext transfer protocol (http/1.1): Message syntax and routing," tech. rep., RFC 7230, June 2014 (TXT), 2014.

[25] X. Yan, Y. Xu, B. Cui, S. Zhang, T. Guo, and C. Li, "Learning url embedding for malicious website detection," *IEEE Transactions on Industrial Informatics*, vol. 16, no. 10, pp. 6673–6681, 2020.

[26] A. Ben-Israel and T. N. Greville, *Generalized inverses: theory and applications*, vol. 15. Springer Science & Business Media, 2003.

[27] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[28] T.-H. Wen, M. Gasic, N. Mrksic, P.-H. Su, D. Vandyke, and S. Young, "Semantically conditioned lstm-based natural language generation for spoken dialogue systems," *arXiv preprint arXiv:1508.01745*, 2015.

[29] D. Pawade, A. Sakhapara, M. Jain, N. Jain, and K. Gada, "Story scrambler-automatic text generation using word level rnn-lstm," *International Journal of Information Technology and Computer Science (IJITCS)*, vol. 10, no. 6, pp. 44–53, 2018.

[30] C. Zhou, C. Sun, Z. Liu, and F. Lau, "A c-lstm neural network for text classification," *arXiv preprint arXiv:1511.08630*, 2015.

[31] G. Liu and J. Guo, "Bidirectional lstm with attention mechanism and convolutional layer for text classification," *Neurocomputing*, vol. 337, pp. 325–338, 2019.

[32] A. Xu, Z. Liu, Y. Guo, V. Sinha, and R. Akkiraju, "A new chatbot for customer service on social media," in *Proceedings of the 2017 CHI conference on human factors in computing systems*, pp. 3506–3510, 2017.

[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[34] Y. Li, Z. Hao, and H. Lei, "Survey of convolutional neural network," *Journal of Computer Applications*, vol. 36, no. 9, pp. 2508–2515, 2016.

[35] W. Wang, Y. Yang, X. Wang, W. Wang, and J. Li, "Development of convolutional neural network and its application in image classification: a survey," *Optical Engneering*, vol. 58, no. 4, p. 040901, 2019.

[36] R. Johnson and T. Zhang, "Effective use of word order for text categorization with convolutional neural networks," *arXiv preprint arXiv:1412.1058*, 2014.

[37] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," *arXiv preprint arXiv:1509.01626*, 2015.

[38] Y. Kim, "Convolutional neural networks for sentence classification," 2014.

[39] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," 2016.

[40] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[41] S. Kullback and R. A. Leibler, "On information and sufficiency," *The annals of mathematical statistics*, vol. 22, no. 1, pp. 79–86, 1951.

[42] L. Theis, A. v. d. Oord, and M. Bethge, "A note on the evaluation of generative models," 2015.

[43] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein gan," 2017.

[44] K. Tian, S. T. Jan, H. Hu, D. Yao, and G. Wang, "Needle in a haystack: Tracking down elite phishing domains in the wild," *Proceedings of the ACM SIGCOMM Internet Measurement Conference, IMC*, pp. 429–442, 2018.

[45] T. Moore and B. Edelman, "Measuring the perpetrators and funders of typosquatting," in *International Conference on Financial Cryptography and Data Security*, pp. 175–191, Springer, 2010.

[46] L. J. Sern, Y. G. Peng David, and C. J. Hao, "Phishgan: Data augmentation and identification of homoglyph attacks," in *2020 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*, pp. 1–6, 2020.

[47] L. Degani, F. Bergadano, S. A. Mirheidari, F. Martinelli, and B. Crispo, "Generative adversarial networks for subdomain enumeration," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pp. 1636–1645, 2022.

[48] L. J. Sern, Y. G. P. David, and C. J. Hao, "PhishGAN: Data augmentation and identification of homoglyph attacks," in *2020 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*, IEEE, nov 2020.

[49] Z. Lin, A. Jain, C. Wang, G. Fanti, and V. Sekar, "Using gans for sharing networked time series data: Challenges, initial promise, and open questions," in *Proceedings of the ACM Internet Measurement Conference*, pp. 464–483, 2020.

[50] C. Yin, Y. Zhu, S. Liu, J. Fei, and H. Zhang, "An enhancing framework for botnet detection using generative adversarial networks," in *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*, pp. 228–234, IEEE, 2018.

[51] Z. Lin, Y. Shi, and Z. Xue, "Idsgan: Generative adversarial networks for attack generation against intrusion detection," *arXiv preprint arXiv:1809.02077*, 2018.

[52] Charlier *et al.*, "Syngan: Towards generating synthetic network attacks using gans," *arXiv preprint arXiv:1908.09899*, 2019.

[53] M. Usama, M. Asim, S. Latif, J. Qadir, *et al.*, "Generative adversarial networks for launching and thwarting adversarial attacks on network intrusion detection systems," in *2019 15th international wireless communications & mobile computing conference (IWCMC)*, pp. 78–83, IEEE, 2019.

[54] M. Ring, D. Schlör, D. Landes, and A. Hotho, "Flow-based network traffic generation using generative adversarial networks," *Computers & Security*, vol. 82, pp. 156–172, 2019.

[55] A. Cheng, "Pac-gan: Packet generation of network traffic using generative adversarial networks," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pp. 0728–0734, IEEE, 2019.

[56] Cho *et al.*, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[57] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[58] A. M. Dai, C. Olah, and Q. V. Le, "Document embedding with paragraph vectors," *arXiv preprint arXiv:1507.07998*, 2015.

[59] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: value-flow-based precise code embedding," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[60] Q. Ma, S. Muthukrishnan, and W. Simpson, "App2vec: Vector modeling of mobile apps and applications," in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 599–606, IEEE, 2016.

[61] W. Lopez, J. Merlino, and P. Rodriguez-Bocca, "Vector representation of internet domain names using a word embedding technique," in *2017 XLIII Latin American Computer Conference (CLEI)*, pp. 1–8, IEEE, 2017.

[62] Gonzalez *et al.*, "Net2vec: Deep learning for the network," in *Proceedings of the Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, pp. 13–18, 2017.

[63] L. Gioacchini, M. Mellia, I. Drago, L. Vassio, Z. Ben Houidi, and D. Rossi, "Darkvec - automatic analysis of darknet traffic with word embeddings," in *Proceedings of the ACM CoNEXT*, ACM, 2021.

[64] J. Cui, J. Long, E. Min, and Y. Mao, "Wedl-nids: improving network intrusion detection using word embedding-based deep learning method," in *International Conference on Modeling Decisions for Artificial Intelligence*, pp. 283–295, Springer, 2018.

[65] H. Gwon, C. Lee, R. Keum, and H. Choi, "Network intrusion detection based on lstm and feature embedding," *arXiv preprint arXiv:1911.11552*, 2019.

This article has been accepted for publication in IEEE Transactions on Network and Service Management. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TNSM.2022.3225311

Rodolfo Vieira Valentim has a Bachelor's Degree in Computer Engineering at Universidade Federal do Espírito Santo (UFES). Also, at UFES, he obtained his master's degree in Informatics. In 2015, he was awarded a fully-funded scholarship to spend one year at the Hanze Institute of Technology in the Netherlands as an exchange student. His research interests are Software-Defined Networks, Data Center Networks, Cloud Computing, Network Security, Artificial Intelligence, and Anomaly Detection. Currently, he is a Ph.D. student under the supervision of Marco Mellia and Idilio Drago, conducting his research at ET / SmartData@PoliTo Center. His research aims an AI-assisted approach for network security based on multiple darknets and honeypots.

Idilio Drago is an Assistant Professor at the University of Turin, Italy, in the Computer Science Department. His research interests include network security, machine learning and Internet measurements. He is particularly interested on how big data and machine learning can help to extract knowledge from traffic data and help securing the network and automating network management tasks. Drago has a Ph.D. in computer science from the University of Twente, the Netherlands, and a Master's degree from the Federal University of Espirito Santo, Brazil. He was awarded an Applied Networking Research Prize in 2013 by the IETF/IRTF for his work on cloud storage traffic analysis.

Martino Trevisan is an Assistant Professor at the Department of Engineering and Architecture of the University of Trieste. He received his M.Sc. (2015) and Phd (2019) in Computer Engineering from Politecnico di Torino. During his career, he visited Télécom ParisTech (Paris), Cisco Systems labs (San Josè, US), AT&T (Bedminster, US) and the Universidade Federal de Minas Gerais (Brazil). He published more than 40 papers in prestigious journals and conferences in the field of networking and big data. His research interests are mainly focused on big data methodologies for Web and Internet analysis. He also studies the operation of Online Social Networks and their implications on user behaviour.

Marco Mellia (F'21) is the coordinator of the SmartData@PoliTO center, a interdisciplinary lab involving more than 50 researchers with focus on Big Data analytics and Data Science, with applications to network management, cybersecurity, smart cities, intelligent transport systems, and predictive maintenance. He has co-authored over 250 papers published in international journals and presented in leading conferences. He hold 11 patents. He won the IRTF ANR Prize at IETF-88, and best paper awards at IEEE P2P'12, ACM CoNEXT'13, IEEE ICDCS'15, ACM CCR'16, ITC'18. He is part of the editorial board of IEEE Transactions on Network and Service Management, Elsevier Computer Networks, ACM Computer Communication Review, and Editor in Chief of the Proceedings of the ACM on Networking. He now holds a position as Full Professor at Politecnico di Torino, Italy.