Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks

Zhihao Bai*, Ke Wang†, Hang Zhu*, Yinzhi Cao*, Xin Jin†
*Johns Hopkins University, †Peking University

***Abstract*—Regular expression denial of service (ReDoS)—which exploits the super-linear running time of matching regular expressions against carefully crafted inputs—is an emerging class of DoS attacks to web services. One challenging question for a victim web service under ReDoS attacks is how to *quickly* recover its normal operation after ReDoS attacks, especially these zero-day ones exploiting previously unknown vulnerabilities.**

In this paper, we present REGEXNET, the first payload-based, automated, reactive ReDoS recovery system for web services. REGEXNET adopts a learning model, which is updated constantly in a feedback loop during runtime, to classify payloads of upcoming requests including the request contents and database query responses. If detected as a cause leading to ReDoS, REGEXNET migrates those requests to a sandbox and isolates their execution for a fast, first-measure recovery.

We have implemented a REGEXNET prototype and integrated it with HAProxy and Node.js. Evaluation results show that REGEXNET is effective in recovering the performance of web services against zero-day ReDoS attacks, responsive on reacting to attacks in sub-minute, and resilient to different ReDoS attack types including adaptive ones that are designed to evade REGEXNET on purpose.

***Index Terms*—Regular expression Denial of Service (ReDoS); Deep Neural Networks; Adversarial Machine Learning; Online Feedback Loop**

I. INTRODUCTION

Regular expressions, a fundamental tool in computer software, are widely used in web services to manipulate, validate, and scrape user data [1], [2]. Although widely adopted, the matching process of regular expressions, especially against carefully crafted inputs, can take super-linear, i.e., polynomial or even exponential, time with respect to the input length, leading to a so-called regular expression denial of service (ReDoS) [3], [4], [5] and posting a critical threat to web services today [4], [6]. For example, one study has shown that thousands of regular expressions in over 10,000 JavaScript or Python modules are vulnerable to ReDoS [4]. Another study has shown that 339 out of 2,846 popular websites suffer from at least one ReDoS vulnerability, which can be easily exploited to take down the websites [5].

Because of the severe situation of ReDoS attacks, researchers have proposed various defenses [7], [8], [3], [5], [9], [10], [11], which can be roughly categorized into two general types: proactive and reactive. On one hand, proactive defenses mitigate ReDoS attacks by speeding up regular expression matching. For example, several libraries, such as safe-regex [7] and rxxr2 [8], are proposed to check the safety of regular expressions for Node.js applications via an e-NFA structure. Wüstholz et al. [3] also present an algorithm to check the vulnerabilities of regular expressions via an NFA-alike structure. The state of the art from Davis et al. [12] proposes a sound ReDoS defense using selective memoization schemes.

While proactive approaches are effective in defeating ReDoS attacks, there are some tradeoffs in preventing them being deployed in practice. For example, many existing ReDoS defenses [7], [8], [3], [5], [9], [10] are not sound [13], [14], [15], [16], leaving some regular expressions still vulnerable; some defenses, particularly Davis et al. [12], are sound in ensuring linear matching time but take significantly more space, possibly leading to a space-related DoS. In addition, many existing approaches [7], [8], [17], [12] need modifications to either the web application itself or the web framework supporting the application, thus facing obstacles in deployment.

On the other hand, a reactive approach, as opposed to proactive, is to recover web services from ReDoS attacks, especially zero-day ones, during runtime after being taken down. This is also important because a proactive approach may fail or is not in place due to deployment concerns. The most naïve yet still common reactive approach is probably manual inspection, which leads to a gap for a website between being unavailable and going back online. As a concrete example, it took StackOverflow about 34 minutes to recover from an unknown ReDoS vulnerability in its source code, which included 10 minutes to analyze suspicious web requests and identify the cause, 14 minutes to fix problematic code segments, and 10 minutes to roll out the software update [18].

Reactive approaches can be automated. Particularly, Rampart [11]—the state-of-the-art reactive approach—adopts a behavior-based, statistical approach to differentiate normal and malicious requests based on consumed CPU resources. If malicious, Rampart blocks future requests from the same IP address or with exactly the same content to defend against so-called CPU-exhaustion Denial-of-Service (DoS) attacks. However, there is a major drawback of Rampart. Adversaries can bypass Rampart by launching distributed attacks from different IPs using, e.g., a botnet, and more importantly adopt polymorphic attack payloads after content manipulation.

In this paper, we present REGEXNET, the first payload-based, runtime, reactive ReDoS recovery system for web services. Specifically, REGEXNET inspects payloads including those embedded directly in the request itself and those triggered indirectly, e.g., from database queries, as they may also come from previous requests and are stored in the server. One advantage of such a payload-based recovery system is that it disregards the request source, e.g., IP addresses, and is robust

to content manipulations. More importantly, a payload-based recovery system can be combined with existing behavior-based ones to together protect and recover web services.

Our key insight is that malicious payloads triggering ReDoS attacks have to obey certain underlying patterns, which stay invariant during content manipulation across different attacks targeting the same vulnerable regular expression. For example, if a vulnerable regular expression takes super-linear time in matching a series of spaces, the malicious payload has to include such a pattern. REGEXNET relies on recent advancements in deep learning to efficiently learn such an underlying, invariant pattern and detect inputs with the pattern quickly, i.e., in linear time, without human intervention. If a request is detected as a trigger to ReDoS vulnerabilities, REGEXNET will migrate it from corresponding web servers to sandboxes for isolated, controlled execution.

While intuitively simple, deep learning is not a silver bullet. Although it can efficiently learn the underlying pattern and make fairly accurate predictions for unseen payloads, state-of-the-art deep learning makes mistakes, especially under adversarial environments. Here are two scenarios: (i) the learning model itself has some false positives and negatives, and (ii) an adversary can launch an adaptive attack, just as what people did in the vision field [19], [20], against the learning system.

REGEXNET tackles the imperfection of the learning model via an *online feedback loop*, which collects all the client-side inputs related to a web request, such as the request itself and database query responses, and the processing time of the request to update the learning model. The idea—being inspired by adversarial training proposed by Goodfellow et al. [20] and further improved by Madry et al. [21]—keeps improving the model, thus making it stronger over time.

We have implemented a system prototype of REGEXNET, and integrated it with HAProxy [22], a widely-used software load balancer, and Node.js [23], a popular web application framework. We believe that REGEXNET can be deployed as a fast, first measure for ReDoS recovery, in addition to the slow process of fixing and rolling out the source code update with human engineers. We have evaluated REGEXNET in a real-world Node.js web application testbed and measured the system performance under a wide variety of real ReDoS attacks including zero-day and adaptive ones. The evaluation results show that the throughput and latency of a web application under ReDoS attacks can quickly recover to the normal level within sub-minute, which is several orders of magnitude faster than a manual recovery. We also compare REGEXNET with the state-of-the-art reactive ReDoS defense, Rampart, and show that REGEXNET significantly outperforms Rampart in terms of normalized throughput.

In summary, we make the following contributions.

- We propose REGEXNET, the first payload-based, automated ReDoS recovery system for web services that leverages a learning model to classify requests and recover websites after zero-day ReDoS attacks.
- We design an online feedback loop for REGEXNET to collect training data at runtime, continuously train its model online, and automatically update its model to classify and migrate requests, in face of adaptive, unknown attacks.
- We implement a system prototype of REGEXNET and demonstrate its effectiveness, responsiveness, and resiliency with experiments on a testbed with a wide variety of real-world ReDoS attacks on Node.js.

```
/(?:charset|encoding)\s*=\s*['"]?␣*([\w\-]+)/i
```

(a) A vulnerable regular expression.

```
content-type:charset=␣...␣x
```

(␣ is repeated by n times in the input.)

(b) A malicious input to trigger the vulnerability.

```
/(?:charset|encoding)\s*= (\s*['"]␣*|\s*)([\w\-]+)/i
```

(c) A semantically equivalent regular expression without vulnerabilities.

Fig. 1. An illustration of a real-world vulnerable regular expression, its exploits and semantically-equivalent, safe counterpart.

II. REGEXNET OVERVIEW

In this section, we start from a motivating example and explain the key ideas of REGEXNET in recovering web services from ReDoS attacks. Then, we present the threat model of REGEXNET and how REGEXNET is deployed in practice.

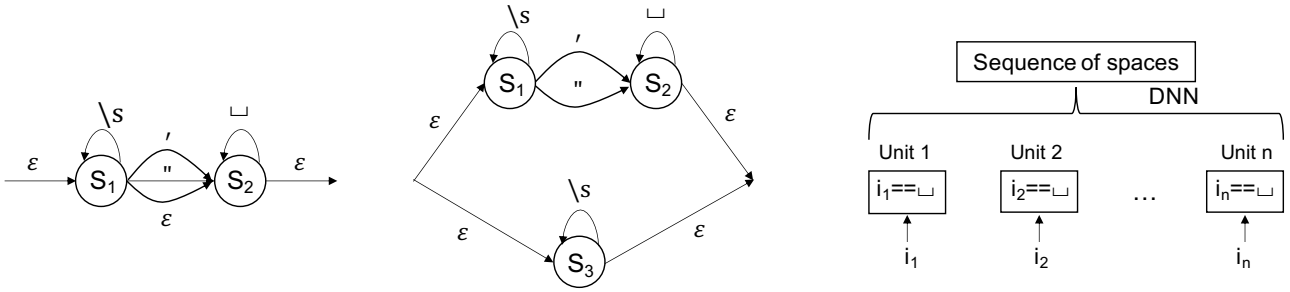
A. A Motivating Example

In this part, we illustrate a real-world vulnerable regular expression in Fig. 1 and explain how it makes websites vulnerable to ReDoS attacks. The vulnerable regular expression [5] is applied upon every request to parse the accepted `charset` in the HTTP header for encoding purposes. An adversary targeting this vulnerable regular expression sends many requests containing malicious payload (e.g., one shown in Fig. 1(b)) in the `charset` HTTP header to the vulnerable website: each request originated from the adversary will occupy the server for a fairly long time and all these add up to a Denial of Service (DoS) consequence.

We now explain why the regular expression in Fig. 1(a) is vulnerable. The vulnerable part in this regular expression is `\s*['']?␣*`, which matches strings with zero or more of `\s` (metacharacter for whitespace, such as space, tab, and carriage return), zero or one of `'` and `''`, and zero or more of ␣ (space character with ASCII code 32_{10}). The malicious payload shown in Fig. 1(b) is a string with n space characters. A space character can be either matched with `\s` or ␣. As such, the matching process needs to split this string into two parts, with one matching `\s*` and the other matching ␣*. Because there are $(n+1)$ different ways to split the string, the regular expression engine needs to exhaust all the $(n+1)$ combinations to figure out the string cannot be matched. Since each combination takes $O(n)$ time to match, the time complexity to match this string is $O(n^2)$, i.e., super-linear.

B. Threat Model and Practical Deployment

Threat Model. REGEXNET's threat model considers a website hosting a web service as a potential victim. The victim may deploy a vulnerable regular expression to match against



(a) A vulnerable state machine (a failed matching of n*␣ is $O(n^2)$). (b) An equivalent, non-vulnerable state machine (the second path matches the vulnerable inputs). (c) An ideal, learned DNN capturing the vulnerable inputs, which mimics the second path in (b).

Fig. 2. An explanation that REGEXNET can capture the correct logics of regular expression matching with O(n) time based on the example in Fig. 1.

incoming HTTP requests. The adversary is the client(s) of the web server, which sends requests with malicious payloads for ReDoS attacks. Borrowing classic terminologies in Cross-site Scripting (XSS) attacks, we further classify ReDoS attacks into two categories, i.e., reflected and stored, based on the adversary's behaviors.

First, a *reflected* ReDoS adversary keeps sending malicious requests to the vulnerable web server. Each request may be polymorphic in contents and origins, i.e., the adversary may adopt different IP addresses and change the malicious payload to avoid being detected. We call this attack reflected because the ReDoS impacts are reflected back to other benign users of the same service.

Second, a *stored* ReDoS adversary just sends one malicious request to the vulnerable web server and then the request or part of the contents are stored at the server side. Then, when a benign user sends a request to the server, the stored contents are fetched and matched against a vulnerable regular expression, leading to a DoS attack. For example, the adversary may post a comment with malicious payloads on a blog and the vulnerable regular expression is a filter or a censorship of illegal contents: Every benign request to the blog will trigger the ReDoS attack. We call this attack stored because the ReDoS payload is stored at the server-side database.

Practical Deployment. REGEXNET is provided as a software appliance that is deployable on general-purpose servers at the application layer of a web server to inspect all the web traffic after decryption. It is an appliance of network function virtualization (NFV), and can be deployed together with other NFV appliances. We consider two types of deployment models.

First, the direct deployment is via web service providers. Specifically, a provider runs an instance of our system co-located with its web servers in the same datacenter to protect itself from ReDoS attacks. Large-scale web service providers with geographically-distributed datacenters need to run a separate instance of REGEXNET in each of its datacenters. These REGEXNET instances communicate with each other to update their learned models.

Second, a public cloud service provider can also deploy REGEXNET via exposing APIs to the users of the public cloud, and provide REGEXNET as a ReDoS recovery service for its users. Specifically, these APIs will provide a callback function for each incoming web request—once REGEXNET identifies a request as ReDoS traffic, the callback function will be invoked to alert cloud users to take corresponding actions. A cloud user can also call the provided APIs to let the cloud provider know that a request is benign or malicious so that the provider can update the model. Note that the cloud service provider needs to maintain a model for each user to avoid cross-contamination, i.e., a model being polluted by other cloud users.

C. Key Ideas

Overview. The key idea of REGEXNET is to quickly analyze all the requests to a web server and predict whether they will lead to a significantly long processing time: If so, REGEXNET will migrate those requests to separate sandboxes and isolate their impact from the rest for a fast recovery. The analysis includes the request itself for reflected ReDoS and all the responses from database queries for stored ReDoS. Next, REGEXNET will directly isolate all the malicious requests launched by a reflected ReDoS adversary; REGEXNET will also isolate all the requests viewing pages containing the malicious payload injected by a stored ReDoS adversary. As a result, all the normal services of the web server under reflected ReDoS will be recovered; the services of pages with no stored ReDoS payload are restored and pages with stored ReDoS payload are isolated by REGEXNET.

Linear-time analysis of requests. Next, we describe how REGEXNET achieves fast recovery via a quick analysis of all requests. REGEXNET relies on deep neural networks (DNNs) to learn the behaviors of matching regular expressions against sequential data among consecutive characters with *a few* samples just like few-shot learning and mimic such matching during the classification stage in a linear time.

Particularly, REGEXNET uses a linear-time DNN to match malicious requests, instead of super-linear time regular expression matching. While this seems counterintuitive at first glance, our approach actually *reflects* how the malicious requests should be matched in *linear* time using a semantically equivalent regular expression *without vulnerabilities*. We use the same example in Fig. 1 to illustrate why the solution works. Fig. 1(c) shows a semantically equivalent regular expression *without vulnerabilities*. The vulnerable part `\s*[' ']?␣*` is divided into two parts, i.e., `\s*[' ']␣*`, and `\s*`. Each

part matches any string in *linear* time, and thus this regular expression eliminates the vulnerability that exists in Fig. 1(a).

We now use Fig. 2 to explain why a DNN can reflect the matching of a semantically equivalent regular expression without vulnerabilities in Fig. 1(c). Note that the figure only shows the portion related to the vulnerable part to simplify the illustration. First, Fig. 2(a) shows the state machine of the vulnerable regular expression in Fig. 1(a) to match a malicious input. There are two states, i.e., S_1 and S_2, which map to \s* and ␣*. Since the space character matches both states and there are $(n+1)$ possible combinations to partition n space characters, the total matching time is $O(n^2)$. Second, Fig. 2(b) shows the state machine of the semantically equivalent regular expression without vulnerabilities to match a malicious input. There are only two possible paths with $O(n)$ matching path: The malicious string does not match the top path, and only matches the bottom path. Lastly, we use Fig. 2(c) to show how a DNN can mimic the behavior of the correct regular expression for linear-time matching. Each unit in the DNN receives a character as input and the DNN is trained to recognize sequences of spaces. Note that Fig. 2(c) is a simplified illustration to show the main idea. The actual model in REGEXNET uses a combination of embedding, 1-d convolutional, spatial pyramid pooling and fully-connected layers that can process variable-length input with linear time.

III. REGEXNET DESIGN

We describe the design of REGEXNET in this section.

A. REGEXNET *Design Goals*

REGEXNET is designed to recovery web services from zero-day ReDoS attacks, with the following goals.

- **Effective.** REGEXNET should effectively recover a web service to resume processing of requests from normal clients after a zero-day ReDoS attack by providing a comparable throughput (in terms of requests processed per second) and latency (in terms of processing latency of each request).
- **Responsive.** REGEXNET should quickly react to a zero-day ReDoS attack, and minimize the downtime of a web service caused by the attack.
- **Resilient.** REGEXNET should be resilient to different ReDoS attack types, no matter whether a ReDoS attack is targeted at a known or unknown vulnerability.
- **Low overhead.** REGEXNET should incur low overhead on the throughput and latency of request processing.
- **Scalable and fault-tolerant.** REGEXNET should be able to scale out based on the amount of web traffic, and tolerate the failures of individual system components.

B. REGEXNET *Architecture and Workflows*

Fig. 3 shows the overall architecture of REGEXNET. REGEXNET, a ReDoS recovery system, is a software appliance that recovers web servers under ReDoS attacks from adversarial clients. REGEXNET relies on an online feedback loop to train a customized DNN model, leverages the model to detect malicious ReDoS requests, and then isolates them

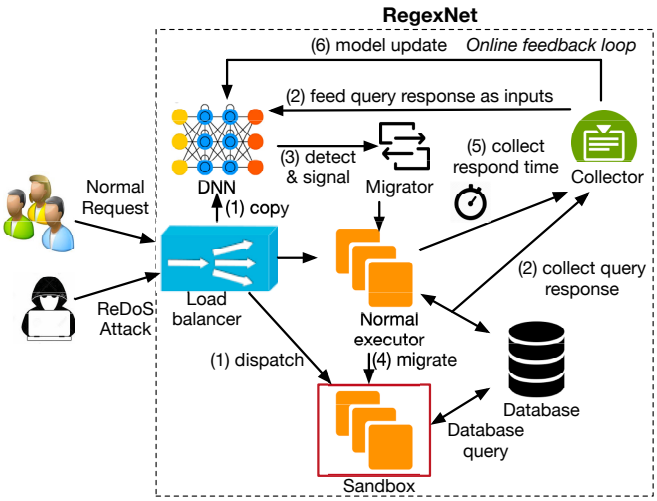


Fig. 3. REGEXNET architecture.

to separate sandboxes—i.e., an elastic bounded (e.g., 10%) fraction of server instances—to mitigate their impact on web services. REGEXNET has a collector, which serves as a *shim layer* running in each web server instance and collects information related to the request execution, such as database query response and runtime execution time, for the feedback loop. Another important module is a load balancer, which spreads web requests over web servers, and copies them to the DNN model for request classification. We provide various load balancing policies for REGEXNET, including round-robin as default, so that all server instances will be equally utilized.

We now describe the workflow of REGEXNET.

1) *Dispatching requests.* The load balancer dispatches web requests based on customized load balancing policies, and copies the requests to the DNN-based detection module.
2) *Intercepting database query response.* The collector intercepts all the queries to the database, records the responses from the database, and then sends the query response back to the DNN-based detection module.
3) *Detecting and signaling.* The DNN-based detection module, taking both the request and the query response, classifies whether the request causes ReDoS, and then signals the migrator to act based on the result. As an optimization, when a database response is malicious, the detector can associate it with the request, and directly classify the following requests to reduce detection latency.
4) *Migrating requests.* The migrator, upon receiving the signal, notifies the corresponding server instance to stop the processing of the request, and sends it to a sandbox.
5) *Collecting processing time.* The collector collects the processing time of each web request, and uses them to update the detection model.
6) *Updating the model.* To close the feedback loop, REGEXNET updates the model with the collected times.

Note that we choose to design and deploy REGEXNET on the network layer due to two reasons. First, it does not impose

any latency overhead on benign requests in normal scenarios, because REGEXNET does not interpose between clients and web servers, and requests are copied to REGEXNET for processing in parallel with normal processing at the web servers. Once a ReDoS attack happens, while the web servers do spend CPU resources on processing malicious requests, they are immediately migrated to sandboxes to isolate their impact after being classified as malicious by the detection module. The CPU resources that can be consumed by the malicious requests at the web servers are capped by the classification time, which is linear. And remember that even normal requests would take linear time to be processed. Because in ReDoS attacks the malicious requests are only a small portion of the total traffic, the system can quickly recover to resume its operation.

Second, it is application-agnostic and thus minimizes deployment efforts. Specifically, the deployment of REGEXNET on the network layer eliminates the need for analyzing and modifying the source code of applications or libraries. Therefore, it simplifies the deployment and can serve a wide range of web applications. As a comparison, an alternative approach is to deploy REGEXNET on the application layer with either the application code or the library of regular expression matching. This application-specific approach requires knowledge about the application code, incurring additionally engineering efforts. More importantly, the source code for certain applications and libraries may not even be available.

C. DNN-based Detection Model

Model design. The detection module uses a DNN model to detect ReDoS attacks. The DNN model takes a web request as input and classifies whether the request is benign or malicious. Specifically, a web request r is represented as an array of characters, i.e., $r = \{c_i | c_i \in C\}$ where C is the set of characters. The DNN model f is applied to r to compute a label in $\{0, 1\}$, i.e., $f(r) \in \{0, 1\}$, where label 0 means benign and label 1 means malicious.

Our DNN architecture has four layers, one embedding layer, one 1D convolutional (conv1d) layer, one spatial pyramid pooling (SPP) layer, and a fully-connected layer. When a web request comes in, the characters are fed into the embedding layer at the character level. The embedding layer generates an embedded vector for each character. Then the conv1d layer takes these vectors as input and performs convolution operations on them. The SPP layer converts the output from the conv1d layer into fixed length with max pooling. Finally, the fully-connected layer outputs a result, i.e., either benign or malicious, for the web request.

We adopt this specific DNN architecture due to the following reasons. First, we choose an embedding layer, because it can turn the input characters into dense vectors, which is essential to the convergence of the DNN. An advantage of the embedding layer is that it is able to encode distances between characters and show their similarity. For example, the distance between the letter "0" and "1" should be smaller than that of "0" and "a", because both "0" and "1" could be matched by the symbol "\d" in regular expression which stands for all the digits. Thus, such property of the embedding layer is important. Second, we choose a conv1d layer, because a conv1d layer is able to extract local information from the sequence of characters, and it is also computationally efficient. The conv1d layer is followed by tanh as a common practice. Third, we choose an SPP layer that draws the feature maps from the output of the convolutional layer only once, and then pools the features in arbitrary regions to generate fixed-length representations for the fully-connected layer. Both the convolutional layer and the SPP layer are able to process data in $O(n)$ time where n is the length of the data. Lastly, we choose a fully-connected layer that allows information to flow between units extracted by the SPP layer, thus being able to capture useful patterns from a global perspective. Note that we choose SPP instead of recurrent neural networks (RNNs), because SPP is more computation-efficient.

Model training and update. REGEXNET provides both an offline-trained model and an online update procedure. The offline training is used to bootstrap the DNN model, and the online training is used to refine the model with real-time measurements to adapt to both known and unknown attacks. Note that offline training is optional. REGEXNET can be deployed without offline training, and only use online training to learn the attack patterns and react to ReDoS attacks. Offline training is useful to reduce the reaction time to known attacks.

The offline training component uses a training dataset to train the initial DNN model. The training dataset could be collected from previous attacks or generated based on the analysis for known vulnerabilities. It contains a set of web requests $R = \{r_i\}$ and their labels $L = \{l_i\}$, which indicate whether each request is malicious or not. We use the cross entropy loss $\sum_i(-\log(\frac{\exp(f(r_i)[l_i])}{\sum_j \exp(f(r_i)[l_j])}))$ to maximize the classification accuracy. We use the common mini-batch gradient descent method to train the model. A challenge in training a model for ReDoS detection is that the datasets are usually imbalanced, i.e., there are more benign requests than malicious requests in the datasets. We sample the benign requests with random undersampling to make the datasets more balanced, in order to achieve high detection accuracy.

The online training component continuously refines the model to detect unknown attacks. There are two steps. First, the component builds the training data with real-time measurements collected from the web servers. The collector, i.e., the shim layer in each server instance, tracks the response time of each web request, and compares the response time with a pre-defined threshold. If the response time is above the threshold, REGEXNET considers the request as malicious because it consumes excessive CPU resources. The shim layer immediately reports the malicious request to the online training component, in order to quickly react to the attack. The collector also tracks the response time of each request in the sandbox. If the execution of a request in the sandbox consumes less time than the threshold, the collector also reports the request to the online training component, because the request is misclassified as malicious.

Second, the online training component adopts a hot-start procedure to update the DNN model to include the newly collected data. Specifically, the component starts from previously converged weights and adds newly collected data into the backward propagation. The advantage of such a hot-start procedure is that the model is almost converged, and thus the convergence speed is much faster.

Threshold selection. REGEXNET adopts a threshold in the online training component for the feedback loop. The threshold is either specified by an administrator or determined based on the statistics of benign requests during the testing phase of the target web application. Specifically, REGEXNET calculates the average processing time of benign requests μ and the standard deviation σ. Then, REGEXNET adopts $\mu+3\sigma$ as the threshold, i.e., the requests of which the processing time is above $\mu+3\sigma$ are fed back to the online training component.

There are two things worth noting here. First, malicious requests that are executed in less than $\mu + 3\sigma$ time will not trigger the online feedback loop. Such requests will not slow down the web application much because their execution time is relatively small. Additionally, if the attacker chooses to send a large number of such requests, the detection will default back to the traditional network-layer, volume-based DoS detection.

Second, the chosen threshold may also introduce false positives of our DNN-based detection module, e.g., feeding benign requests with a large file upload to the online feedback loop as potentially malicious requests. However, these requests are not dropped. Instead, as shown in Section III-D, they are migrated to the sandbox, and are still executed, albeit slower due to the limited resources in the sandbox. The mislabelling can be fixed by the operator by resetting the threshold, correcting the mislabeled data and re-training the DNN model.

D. Request Migration

The migration module migrates potentially malicious requests to sandboxes. The detailed procedure is as follows. First, the migration module receives the IP address of the scheduled server instance from the load balancer for each web request. The migration module uses the IP address of the scheduled server instance to signal its shim layer. Second, when the shim layer receives the signal, it starts migrating the web request to a sandbox. We use a primary-backup approach to enable the shim layer to stop a web request during processing and be compatible with any web server software. Specifically, we run two instances in each web server: one instance is the primary, and the other one is the backup. The shim layer buffers the web requests in a queue, and sends small batches of requests to the primary for processing. When the shim layer receives a notification from the migration module, it checks whether the request has been sent to the primary or not. If not, the shim layer can simply remove the request from the buffer. Otherwise, the shim layer kills the primary, and makes the backup as the new primary. It resends the previous batch of requests, except for the malicious one, to the new primary for processing, and starts a new backup. At the same time, the malicious request is sent to a sandbox to limit its impact on normal requests. For stateful request processing, we leverage the transaction processing of the database to ensure transactional semantics when killing and restarting requests. Many applications support fault recovery or seamless restart. For example, Zero Downtime Release [24] keeps the partial states in the execution of HTTP requests during restart and replays these requests on other servers. REGEXNET can incorporate such solutions to restart requests smoothly.

E. Scalability and Fault Tolerance

REGEXNET can easily scale out with more servers to handle more web traffic. The shim layer runs on each web server, and scales out with the number of web servers. The number of sandboxes is decided by the server operator, based on how much resource the operator wants to allocate for malicious requests. The detection module and migration module work together to detect and mitigate ReDoS attacks. One instance of the detection module and one instance of the migration module should run together on the server to minimize the communication overhead between the two, but multiple pairs of these instances can run independently on several servers to handle more traffic, as different pairs do not need to coordinate with each other. The training can be done in one instance of the training module, and the trained model can be pushed to all instances of the detection module to update their models.

REGEXNET does not have a single point of failure. REGEXNET handles its individual component failures as follows. (i) *Detection module.* The detection module does not maintain any state. When an instance of the detection module fails, it can be easily replaced by a new instance (e.g., a new server or VM). The new instance gets the latest model from the training module and then begins to classify requests received from the load balancer. (ii) *Migration module.* Similar to the detection module, the migration module does not keep any hard state either. An instance failure of the migration module can be handled by using a new instance. (iii) *Training module.* REGEXNET stores the training data and the trained model in a reliable distributed storage such as HDFS [25]. When the instance of the training module fails, REGEXNET replaces it with a new instance. The new instance restarts the training if the old instance fails in the middle of the training before the model converges. Otherwise, the new instance simply waits for new training data from the shim layer to retrain the model. (iv) *Shim layer, web server and sandbox.* The shim layer, web servers and sandboxes process requests, and they naturally scale out. Their failures can be handled by replacing them with new instances.

IV. IMPLEMENTATION

We have implemented a system prototype of REGEXNET with a total of ∼2,000 lines of code. The code is open source and available at https://github.com/netx-repo/RegexNet.

To demonstrate the practicality of REGEXNET in real-world deployments, the prototype is integrated with HAProxy [22], a widely-used open-source software load balancer, and

TABLE I
A SECURITY ANALYSIS OF REGEXNET AGAINST A VARIETY OF REDOS VULNERABILITIES.

CVE ID	Module	Version	Vulnerable Regular Expression	Recoverable with REGEXNET
CVE-2017-15010	tough-cookie	<2.3.3	`/^(([^=;]+))\s*=\s*([^\n\r\0]*)/`	✓
CVE-2016-4055	moment	<2.11.2	`/(\-)?(?:(\d*)[. ])?(\d+)\:(\d+)(?:\:(\d+)\.?(\d{3})?)?/`	✓
CVE-2015-8858	uglify-js	<2.6.0	`/^\d*\.?\d*(?:e[+-]?\d*(?:\d\.?|\.?\d)\d*)?$/i`	✓
CVE-2015-8854	marked	<0.3.4	`/^\b_((?:__|[\s\S])+?)_\b|^\*((?:\*\*|[\s\S])+?)\*(?!\*)/`	✓
CVE-2015-8315	ms	<0.7.1	`/^((?:\d+)?\.?\d+) *(milliseconds?|msecs?|ms|seconds?|secs?|s`	✓
N/A	charset	<1.0.0	`|minutes?|mins?|m|hours?|hrs?|h|days?|d|years?|yrs?|y)?$/` `/(?:charset|encoding)\s*=\s*['"]? *([\w\-]+)/i`	✓
N/A	content	<3.0.5	`/^([^\/]+\/[^\s;]+)(?:(?:\s*;\s*boundary=(?:"([^"]+)"|([^;"]+)))` `|(?:\s*;\s*[^=]+=(?:(?:"(?:[^"]+)")|(?:[^;"]+))))*$/i`	✓
N/A	fresh	<0.5.0	`/ *, */`	✓
N/A	forwarded	<0.1.0	`/ *, */`	✓
N/A	mobile-detect	<1.3.6	`/Dell.*Streak|Dell.* Aero|Dell.*Venue|DELL.*Venue Pro|Dell Flash|Dell Smoke` `|Dell Mini 3iX|XCD28|XCD35|\\b001DL\\b|\\b101DL\\b|\\bGS01\\b/`	✓
N/A	platform	<1.3.4	`/^ +| +$/g`	✓
N/A	ua-parser-js	<0.7.14	`/ip[honead]+(?:.*os\s([\w]+)*\slike\smac|;\sopera)/`	✓
N/A	useragent	<2.2.1	`/((?:[A-z0-9]+|[A-z\-]+?)?(?:the)?(?:[Ss][Pp][Ii][Dd][Ee][Rr]|[Ss]crape|[A-Za-z0-9-]*(?:[^C]` `[^Uu])[Bb]ot|[Cc][Rr][Aa][Ww][Ll])[A-z0-9]*)(?:(?:[\/]|v)(\d+)(?:\.(\d+)(?:\.(\d+))?)?)?/`	✓

Node.js [23], a popular web framework used by many web services. HAProxy has a customizable module for HTTP request handling that allows developers to add custom functionalities. We customize this module to copy each web request, as well as the corresponding server IP that handles the request, to the detection module of REGEXNET. The detection module is implemented in Python. It buffers web requests received from HAProxy in a queue, and performs classification on each request. It uses PyTorch [26] to run the DNN model for classification, and the model is periodically updated by the training module. When a request is classified as malicious, it signals the migrator module, which stops the request on the corresponding web server, and migrates the request to a sandbox. The collector is implemented in C++. It tracks the response time of each web request and reports them to the training module. The training module is implemented in Python based on PyTorch. It trains the DNN model based on the collected data.

The configuration of the DNN model is as follows. The model takes a web request as input, which is represented as a sequence of characters. The embedding layer maps each character to a 32-dimension vector. The conv1d layer maps 32 channels from the embedding layer to 16 channels, with kernel size 64 and stride 32. The conv1d layer is followed by a tanh activation layer, and then the output is fed into a 3-level SPP layer. The SPP layer pools the feature maps with varied sizes to a fixed size. Finally, the last output of the SPP layer is reduced to two dimensions by a fully-connected layer, and a log softmax layer is applied to generate the label.

V. EVALUATION METHODOLOGIES

In this section, we describe our evaluation methodologies.

A. Experimental Setup and Evaluation Metrics

Setup. The experiments are conducted on AWS. The baseline setup runs a web service on an AWS c5n.4xlarge instance, which is equipped with 16 vCPUs (3.0 GHz Intel Xeon Platinum processor) and 43 GB memory. We use ExpressCart [27], a popular open-source e-commerce web application built with Node.js, to run the web service. We insert vulnerable modules into ExpressCart so that it can be attacked. We use HAProxy [22] for the load balancer and Redis [28] for the database. The clients generate a mix of normal and malicious web requests from different IP source addresses. Next, we introduce the setup with REGEXNET. REGEXNET adds the collector and the migrator to the instance that runs the web service and allocates one vCPU as a sandbox to handle malicious requests. REGEXNET runs the detection module in an additional p3.2xlarge instance, equipped with 8 vCPUs (Intel Xeon E5-2686 v4 processor), 61 GB memory and an NVIDIA GPU (Tesla V100). The adaptive attacks are generated by a c5n.18xlarge instance with 72 vCPUs (3.0 GHz Intel Xeon Platinum processor with AVX-512) and 192 GB memory.

Evaluation metrics. In the experiments, we vary the following parameters: the rate of malicious requests, the size of malicious requests and the type of ReDoS attacks. We use the following metrics to evaluate the performance of the system.

- *Throughput.* This is the number of normal requests per minute (RPM) the system can process.
- *Latency.* This is the average time handling normal requests.
- *Recovery time.* This is the time the system takes to recover from a ReDoS attack.

B. Network Traffics Used in the Evaluation

In this part, we first find modules in Node.js that are vulnerable to ReDoS attacks. Specifically, we search the Common Vulnerabilities and Exposures (CVE) database [29] and academic papers [5] to find existing vulnerable regular expressions as shown in Table I. Among them, the vulnerability in `ua-parser-js` is exponential, and those in other modules are polynomial.

Next, we describe the benign and attack network traffics targeting those modules used in the evaluation. The benign traffic is generated by Apache HTTP server benchmarking tool `ab` [30] with 32 concurrent connections. Note that the generated benign requests have relevant headers that trigger the tested modules. Then, we describe how we generate attack traffics for the tested modules. We generate two types of attack

```
GET http://www.foo.com/
User-Agent: ipaosvWipvWipvWipvWipvWipvWipvWipvWipv
   WipvWipvWipvWipvWipvWipvWipvWipvWipvWipvWipvWipv
   WipvWipvWipvWipvWipaIccvipaIc
```

(a) A malicious web request for `ua-parser-js`, which is generated by ReScue.

```
GET http://www.foo.com/
If-None-Match: x (30000 spaces) x
```

(b) A malicious web request for `fresh`, which is manually crafted.

Fig. 4. Examples of malicious web requests.

```
GET http://www.foo.com/
If-None-Match:
   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
   (30000 spaces)
   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

(a) A malicious web request. The consecutive spaces trigger the vulnerability.

```
GET http://www.foo.com/
If-None-Match:
   1D9xx52xxx9x5x9x25x0xxexOrxO622D350x2U5xxx5g8Gx2x
   exnfxx759xr0x9x25ex02j22x25360X920e99x22xbrO2150518
   (30000 spaces)
   x2x2x22x2r5Sx0Zx0x0vOS1Oxxxxxx20xr6x2x2xF9xSx1SD
   0Fe29xI29I9x7IIxx1xxi1e9D0xxx15x0x55xx3xf5rcho2x2cxx
```

(b) A malicious web request generated by adversarial learning. The consecutive spaces trigger the vulnerability, but the characters before and after the spaces bypass the DNN model.

Fig. 5. An example of an adaptive attack targeted at the `fresh` module. The vulnerable regular expression is shown in Table I. REGEXNET can quickly recover the web service from adaptive attacks as shown in Fig. 7.

traffic: normal, untargeted with random contents, and adaptive with carefully-crafted contents targeting our learning model.

1) Normal Attacks with Random Exploit Contents: We first describe how we generate a large number of normal exploits, i.e., those with fairly random contents targeting each vulnerable regular expression. First, we rely on an automated exploit generation tool, ReScue [13], to generate a series of exploits. Fig. 4(a) shows an example exploit for `ua-parser-js` . Second, if ReScue fails to generate exploits for some regular expressions, we manually analyze the regular expression, find a pattern for exploits, and generate a series of exploits. `fresh` is such an example. The pattern for its exploits is "(random string) + (a random number of spaces) + (random string)". Fig. 4(b) shows an example exploit for `fresh`.

Based on our experience, ReScue is only able to generate attacks for `ua-parser-js` and `marked`, because the minimum sizes of their exploits to consume significant CPU time are hundreds of bytes. Except this, ReScue fails to generate exploits for `useragent` of which the minimum size of exploits is 1 KB. For other vulnerable modules, which require tens of thousands of bytes, ReScue does not finish in five days, and we use manually crafted exploits instead.

2) Adaptive Attacks with Carefully-crafted Exploits: We now describe how we generate adaptive attacks with carefully-crafted exploits. Because REGEXNET adopts a machine learning model, i.e., a DNN, previous research [20], [31], [19] in adversarial machine learning teaches us that one can generate adversarial examples to evade the learning model and bypass the recovery system. The method is similar to gradient descent, while we update the input data instead of the model parameters. To handle discrete values like characters in web requests, we use the method proposed in [31]. The basic idea is to compute the gradients of the embedded vectors of the input data and find the closest embedded vector in the opposite direction of the gradients for each vector.

In our experiments, we consider the DNN model as a whitebox—which is a strong threat model, because the adopted DNN is usually a blackbox to the adversary in practice—and then generate adversarial requests for a vulnerable regular expression. Note that the original attack may change any characters in a sample, which may alter the malice of the request. Therefore, we add additional constraints to ensure that the key exploit triggering the vulnerable regular expression remains the same during adversarial sample generation, and the web request is still legitimate in terms of the format.

Fig. 5 shows an example of an adaptive attack targeted at the vulnerability of the `fresh` module. The vulnerable regular expression is shown in Table I, which can be triggered by consecutive spaces as part of a web request shown in Fig. 5(a). We assume that an adversary can obtain the trained DNN model and generate an adversarial input following prior work [31]—the generated input, as shown in Fig. 5(b), still preserve its malice, i.e., consecutive spaces, but has crafted characters before and after the spaces to evade the model.

VI. EVALUATION

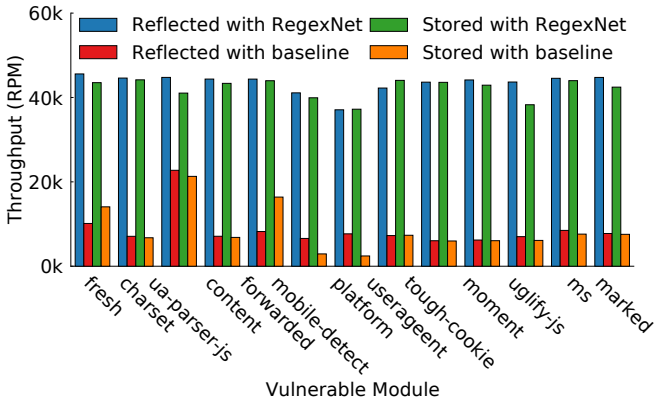
In this section, we evaluate REGEXNET with a wide variety of real-world ReDoS attacks. Our evaluation aims to answer the following research questions (RQs).

- *RQ1:* How resilient is REGEXNET against various ReDoS attacks, e.g., different vulnerable regular expressions, reflected vs. stored, and normal vs. adaptive?
- *RQ2:* How fast is REGEXNET in recovering web service under ReDoS attacks? How effective is REGEXNET's online feedback loop in dealing with zero-day ReDoS attacks?
- *RQ3:* How does REGEXNET compare with state-of-the-art reactive defense, namely Rampart [11]?
- *RQ4:* How effective is REGEXNET under different malicious loads and message sizes?
- *RQ5:* What is the accuracy of REGEXNET's DNN model especially with an imbalanced or polluted training set?

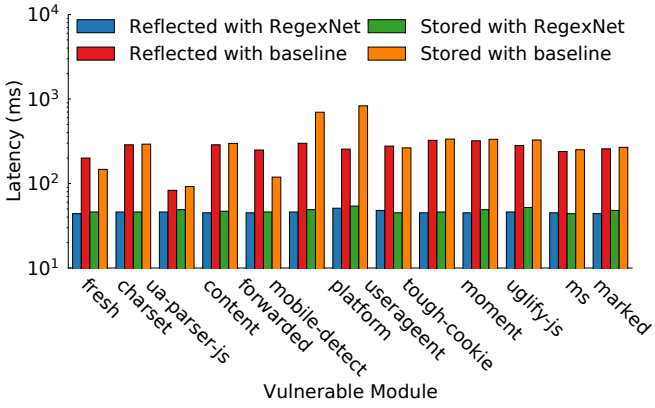
A. RQ1: Resilience against Various ReDoS attacks

In this research question, we evaluate the resiliency of REGEXNET against various ReDoS attacks from three aspects: (i) vulnerabilities in different vulnerable regular expressions, (ii) different vulnerability types, i.e., reflected vs. stored, and (iii) adaptive attacks that generate adversarial samples to bypass a trained DNN model.

Resiliency under attacks on vulnerabilities in different regular expressions. Fig. 6 shows the system performance under ReDoS attacks on different vulnerabilities. Particularly, Fig. 6(a) shows the throughput and Fig. 6(b) shows the latency. In this experiment, we set the malicious load to be 60 RPM, and the maximum message size to be tens of KB



(a) Throughput vs. vulnerable modules.



(b) Latency vs. vulnerable modules.

Fig. 6. System performance under ReDoS attacks on different vulnerable modules. REGEXNET is resilient to different ReDoS attack types.

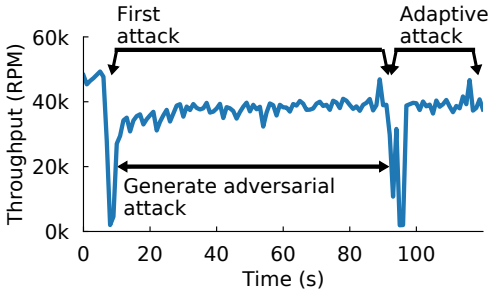


Fig. 7. System performance under an adaptive attack. REGEXNET updates its model online for adversarial examples, and is resilient to adaptive attacks.

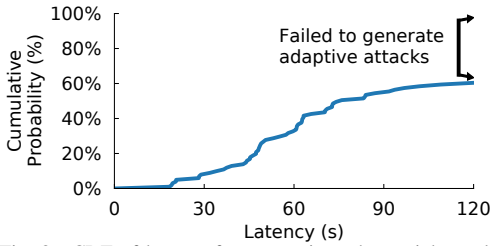


Fig. 8. CDF of latency for generating adversarial attacks.

for polynomial vulnerable modules and hundreds of bytes for exponential vulnerable modules. The adversary exploits different vulnerabilities in Table I, and we measure the system performance under these attacks. As we can see from the figure, the adversary can effectively reduce the throughput and increase the latency of the web service by exploiting these vulnerabilities. REGEXNET is able to successfully handle all these attacks on different vulnerabilities.

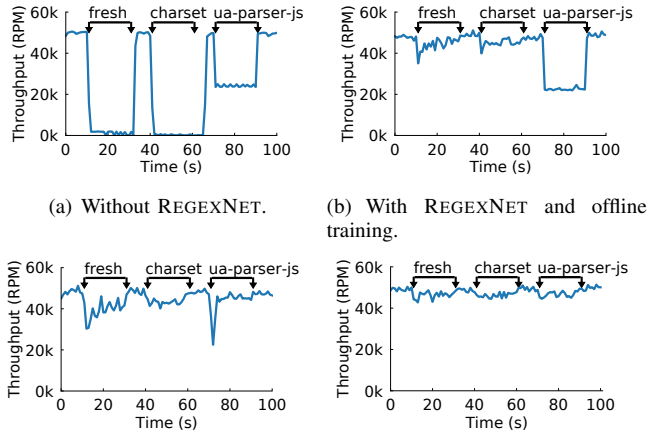
Resiliency under attacks on different types. Fig. 6 also shows the performance of REGEXNET under different attack types, i.e., reflected vs. stored. Note that although some modules are not designed to filter stored contents, e.g., `charset`, as their purpose is to inspect HTTP headers, we still apply them to the stored contents for demonstration and experimental purposes. Although the attack strategy differs in terms of reflected and stored attacks, the resiliency of REGEXNET against these two attacks is similar—particularly, REGEXNET can learn the patterns of all ReDoS attacks and recover the performance of the web service, making the web service still usable to its normal clients.

Resiliency under adaptive attacks in whitebox setting. We evaluate the resiliency of REGEXNET under adaptive attacks. The methodology is as follows. The adversary first launches a zero-day attack against a vulnerable module, namely `fresh`, generates an adversarial payload under a whitebox setting, and then launches the second attack immediately once the adversarial payload is available. The malicious load is 60 RPM. The maximum message size is 32 KB (including 30 KB spaces and 2 KB random strings), and the adversary uses a reflected attack.

Fig. 7 shows the system performance under such an adaptive attack—REGEXNET can quickly recover the web service. Specifically, at the time 7 seconds, the adversary launches the attack, making the throughput drop to almost 0. REGEXNET quickly collects the malicious requests to train its model, enabling it to recognize the following malicious requests and migrates them to sandboxes. The throughput is quickly recovered within a few seconds. Next, at time 10 seconds, the adversary starts to generate an adaptive attack using a whitebox, gradient-based approach, and successfully creates one at time 92 seconds. This adversarial example brings the throughput down to almost 0 again, but REGEXNET quickly recovers the web service and restores the system to the original throughput within 5 seconds at the time of around 97 seconds.

We continue this game between the adversary and REGEXNET for 100 independent rounds, and show the cumulative (CDF) of the latency for generating adaptive attacks in Fig. 8. An adversary can only successfully generate an adaptive attack in 60% cases with an average latency of 56 seconds. The adversary fails to generate any adaptive attacks for the rest 40% cases after a 120-second timeout—this is why Fig. 8 is capped at 60%.

It is worth noting that the whitebox assumption of an adversary, i.e., the adversary knows every detail of our DNN model, is unrealistic. In practice, the adversary cannot calculate the gradients so easily as the adversary has to probe REGEXNET and build her own surrogate model, which takes an even longer time, to create one adversarial example. The experiment shows an ideal, *upper-bound* case for an adaptive attacker.



(a) Without REGEXNET.

(b) With REGEXNET and offline training.

(c) With REGEXNET and online training.

(d) With REGEXNET and both offline and online training.

Fig. 9. System performance under three ReDoS attacks over time. With a combination of offline and online training, REGEXNET reacts to attacks in sub-minute.

Summary: REGEXNET is resilient against all kinds of—i.e., reflected vs. stored and normal vs. adaptive—ReDoS attacks against different vulnerable regular expressions in terms of throughput and latency.

B. RQ2: Recovery Speed under Known and Unknown Attacks

In this research question, we evaluate how fast REGEXNET can recover web services under ReDoS attacks, especially those zero-day ones. Note that REGEXNET uses a combination of offline and online training—the DNN model is trained with known attacks offline, and continuously updated with an online feedback loop for zero-day attacks. We show how this combination enables REGEXNET to quickly react to ReDoS attacks on both known and unknown vulnerabilities.

In this experiment, the adversary launches a dynamic ReDoS attack that targets at different ReDoS vulnerabilities over time. Specifically, at time 10 seconds, it begins to send malicious requests which target the vulnerability in the `fresh` module for 20 seconds. Then it pauses for 10 seconds, and at time 40 seconds, it changes to send malicious requests on the vulnerability in the `charset` module for 20 seconds. After another pause of 10 seconds, it changes to send malicious requests on the `ua-parser-js` module for 20 seconds. During the attack, the malicious load is set to 60 RPM, and the maximum message size is set to 30 KB for `fresh` and `charset` and 128 B for `ua-parser-js`. Fig. 9(a) shows the system throughput without REGEXNET under this dynamic attack. We can see that the throughput drops significantly when the adversary generates malicious requests targeted at the three vulnerabilities during time 10–30 seconds, 40–60 seconds, and 70–90 seconds. The extra duration with low throughput for 60–65 seconds is due to the queued malicious attacks.

Only with offline training, the system is able to handle known attacks, but cannot handle unknown ones. Fig. 9(b) shows the time series of the system throughput with REGEXNET, which is only trained with the vulnerability of the `fresh` module offline, but do not do any adaptation online.

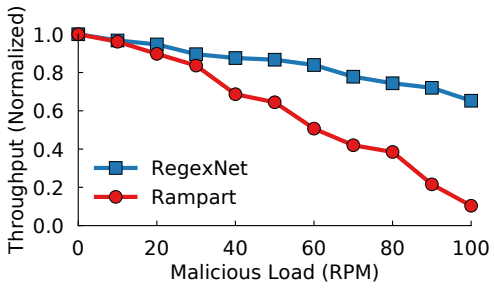


Fig. 10. System performance under different malicious loads, with REGEXNET and Rampart respectively.

At time 10-30 seconds, the attack on the `fresh` module is launched, and REGEXNET is able to recover the system throughput to 44K RPM, as REGEXNET has already learned the attack pattern by offline training. For the attack on the `charset` module , although REGEXNET has not been trained on this module, because the attack pattern of the `charset` module is similar to that of the `fresh` module, REGEXNET is still able to recover the system. However, at time 70-90 seconds, because the attack pattern of the `ua-parser-js` module is different and the system does not train the model online, the system cannot handle this unknown attack and the performance drops to 24K RPM.

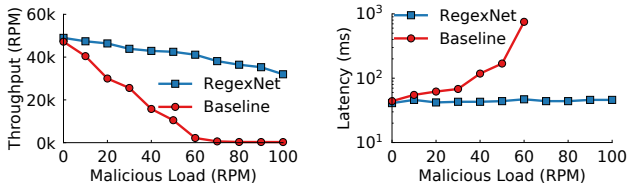
With online training, the system is able to adapt to unknown attacks. Fig. 9(c) shows the time series of the system throughput with REGEXNET, using only online training to handle attacks. With the online feedback loop, REGEXNET can learn the patterns of all three attacks, even if the model has not been trained for them offline. The system throughput can achieve about 40K RPM if the dynamic, zero-day attack happens. For all the three vulnerabilities, REGEXNET can use its online feedback loop to recover the system throughput in sub-minute.

Finally, we show the benefits of both offline and online training. Fig. 9(d) shows the time series of the system throughput with REGEXNET, which is trained with the vulnerability of the `fresh` module offline, and uses the online feedback loop to update the model at runtime. Because the attack pattern of the `fresh` module at time 10-30 seconds has already been learned by REGEXNET offline, the system is able to directly recover, without the delay to collect data and update the model as in Fig. 9(c). Since the attack pattern of the `charset` module is similar to that of the `fresh` module, the system is also able to quickly recover. At time 70-90 seconds, the attack pattern of the `ua-parser-js` module is different and has not been learned. REGEXNET quickly learns the attack and shows nearly no performance loss.

Summary: REGEXNET can quickly recover web services under zero-day ReDoS attacks within a minute.

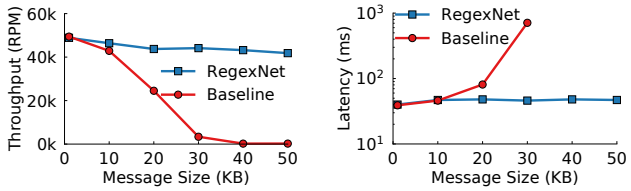
C. RQ3: Comparison with State of the Art

In this research question, we compare REGEXNET with the state-of-the-art reactive defense, Rampart [11], against reflected ReDoS attacks. While Rampart is able to block future attacks by recording some information from malicious



(a) Throughput vs. malicious load. (b) Latency vs. malicious load.

Fig. 11. System performance under different malicious loads, with and without REGEXNET.



(a) Throughput vs. message size. (b) Latency vs. message size.

Fig. 12. System performance under different message sizes, with and without REGEXNET.

requests like IP addresses, the attacker could easily circumvent Rampart by using different IPs for each malicious request. Consequently, Rampart can only detect ongoing attacks and stop them, and thus performs worse than REGEXNET.

Fig. 10 shows this result. The numbers of CPU cores for Node.js and PHP are both set to 1. For a fair comparison, we use normalized throughput, because they have different implementations (Node.js vs. PHP). In addition, the maximum message size is set to 30 KB for REGEXNET and 40 KB for Rampart, because such malicious requests consume approximately the same CPU time for each implementation. We can see that the throughput of Rampart decreases significantly when the malicious load increases, and drops to one-tenth with a malicious load of 100 RPM. On the contrary, the throughput of REGEXNET drops little even with high malicious load.

Summary: REGEXNET significantly outperforms Ramparts as the malicious load increases.

D. RQ4: Different Malicious Payloads and Message Sizes

In this research question, we evaluate the throughputs and latencies of REGEXNET under different malicious payloads and message sizes Specifically, we inject a ReDoS attack on the `fresh` module, and measure the throughput and latency without and with REGEXNET.

Effectiveness under different malicious loads. Fig. 11 shows the throughput and latency of the web service under different malicious loads. The maximum message size is set to 30 KB. When there is no malicious load, the throughput of REGEXNET is nearly the same as that of the baseline (i.e., without REGEXNET). It means that the shim layer introduces negligible overhead. The throughput of the baseline drops to one-third when the malicious load is just 40 RPM, and drops to almost 0 when the malicious load is 60 RPM. Accordingly, the latency of the baseline grows quickly with the increase of the malicious load. This result confirms the serious damage that can be caused by ReDoS attacks in previous work [4], [5], as an adversary only needs a small number of malicious requests to take down a web service. Note that we do not show latencies for malicious loads greater than 60, because in this case the server is overloaded and the latency will increase to arbitrarily large. In comparison, REGEXNET can maintain throughput and latency even under a heavy malicious load.

Effectiveness under different message sizes. Fig. 12 shows the system performance under different maximum message sizes. The load of malicious requests is set to 60 RPM. The adversary uses a uniform distribution to choose a size close to but no greater than the maximum message size for each malicious request. We avoid setting the size to the maximum size for all malicious requests as this would make the attack easily identifiable. As it takes super-linear time for the regular expression engine to match a malicious request, the bigger the message size is, the longer CPU time the request is going to consume. This intuition is confirmed by the behavior of the baseline. The throughput of the baseline halves when the message size is 20KB, and drops to almost 0 when the message size is 30 KB. The latency also increases significantly. Similar to Fig. 11(b), we do not show latency for message sizes greater than 30 KB due to an arbitrarily large value. REGEXNET can maintain high throughput and low latency even when the message size is 50 KB. We want to emphasize that the vulnerability in the `fresh` module is quadratic, so it requires a relatively large message size to take down the service. It is possible that a web service can limit the request size to handle such attacks. However, limiting the request size is not always an option, as some web services do allow large requests, e.g., receiving product reviews on e-commerce sites, and more importantly, there are vulnerabilities that are exponential.

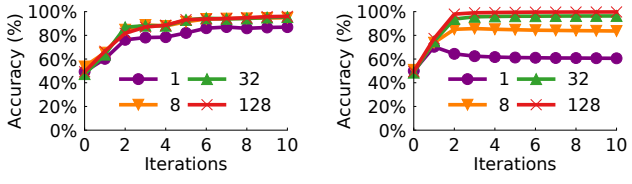
Summary: The throughput of REGEXNET decreases very slowly and the latency stays almost the same as the malicious load and message sizes increase.

E. RQ5: DNN's Accuracy

In this research question, we evaluate the accuracy of the DNN model adopted by REGEXNET. We first create datasets to test the accuracy of the classifier in typical scenarios, and then test the classifier with imbalanced and polluted datasets.

Dataset generation. For each vulnerable module, the dataset contains a training dataset and a test dataset, both of which consist of samples labeled as benign or malicious. We insert the corresponding field into the header of a basic HTTP request. For malicious samples, the content of the field is attacking content, which is crafted manually or generated by ReScue [13]. For benign samples, the content of the field is a random string. The composition of the training dataset depends on the experiment. The test dataset consists of 100 benign samples and 100 malicious samples. Each iteration is to train the model with a batch of size 64. Each batch consists of 32 malicious samples and 32 benign samples, which are selected randomly from the training dataset and allow duplication.

Accuracy over imbalanced datasets. A typical scenario of REGEXNET is to train the classifier with real-time feedback,



(a) fresh (b) ua-parser-js

Fig. 13. Classification accuracy for imbalanced datasets.

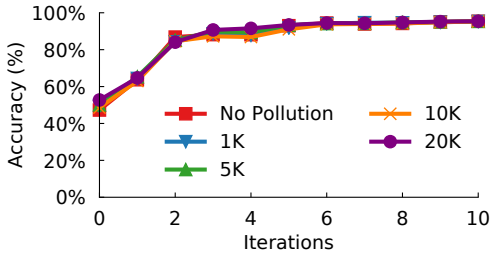


Fig. 14. Classification accuracy for polluted datasets.

where there are a large number of benign samples and a small number of malicious samples. To simulate such a scenario, we test the accuracy of the classifier over an imbalanced training dataset, which consists of 1024 benign samples and a few malicious samples. We test the classifier for two vulnerable modules, `fresh` and `ua-parser-js`. Fig. 13 shows that the accuracy on the test dataset increases when the training proceeds, and each curve means different numbers of malicious samples in the training dataset. Since randomness plays an important role in such a small number of iterations, the reported accuracy is averaged over 100 runs. For `fresh` in Fig. 13(a), the accuracy increases to more than 90% within 5 iterations for 8 or more malicious samples. Even for only 1 malicious sample, it can quickly converge to an accuracy of more than 80%. For `ua-parser-js` in Fig. 13(b), the accuracy increases to more than 90% in about 3 iterations with 32 or more malicious samples. In addition, the accuracy is more than 80% for only 8 malicious samples. However, the curve for only 1 malicious sample shows overfitting and low accuracy, because the pattern of the attacking messages of `ua-parser-js` is more complicated than that of `fresh`. In conclusion, this figure shows that the classifier is able to achieve high accuracy with a small number of malicious samples, and the convergence takes only a couple of iterations.

Accuracy over polluted datasets. A challenging scenario is that the attacker tries to pollute the training dataset. This can be achieved by sending pollution requests which have the same pattern as malicious requests but does *not* exceed the running time due to the small size. For example, for `fresh` module, a malicious request requires more than 20K spaces to consume a significant amount of CPU time, while a pollution request might contain only thousands of spaces. Consequently, the running time of these pollution requests does not exceed the threshold, and will be labeled as benign to pollute the training dataset. To show the performance of REGEXNET under this type of attack, we add such pollution samples to the training dataset and test the accuracy of the classifier. We use `fresh` module in this experiment. Similar to the imbalanced dataset, we generate malicious requests and benign requests to form the dataset. Besides, we generate pollution requests as stated above, and mark them as benign samples. The training dataset consists of 896 purely benign samples, 128 pollution samples (labeled as benign) and 32 malicious samples. The validation dataset consists of 100 purely benign samples and 100 malicious samples. Fig. 14 shows the result of the classifier over the polluted dataset. Each curve means a different size for pollution requests. For example, 1K means the size of the pollution requests is about 1K. We can see that all curves for polluted datasets are very close to the curve for No Pollution, which means that pollution does not affect the training. In other words, REGEXNET is able to classify benign and malicious requests under such pollutions.

Summary: The DNN used in REGEXNET is with >95% accuracy even with imbalanced or polluted training dataset and converges quickly under a few iterations.

VII. DISCUSSION

In this section, we discuss a few common questions that people may have for REGEXNET.

HTTPS traffic. REGEXNET is able to handle HTTPS traffic just like HTTP ones. The reason is that REGEXNET is deployed at the load balancer of a website, a module for distributing web requests at the application layer higher than the transportation layer. That is, all the encrypted traffic, e.g., those transmitted in HTTPS, has already been decrypted and available for analysis. Particularly, our implementation of REGEXNET adopts HAProxy, an open-source load balancer that supports the distribution of TLS/SSL connections.

Character encodings in malicious requests. REGEXNET is able to detect malicious ReDoS attacks even if malicious contents are encoded, e.g., replacing spaces with "%20". The reason is that after a few requests, the online feedback loop of REGEXNET, just like in the case of an adaptive attack, will capture the encoded attack pattern and update the DNN-based detection module for the detection.

Cross-request states. REGEXNET does not change or affect any cross-request states, especially those in the same session. Even if only one request belonging to a session is migrated and isolated, the request can still be correctly processed in the sandbox. The reason is that even without REGEXNET, the current load balancer may distribute requests of the same session to different server instances. Cross-request states, e.g., authentication credentials and a shopping cart of an e-commerce website, are maintained via client-side cookies and server-side databases, which are both shared across different server instances.

Continuous adaptive attack. Adaptive attacks, as shown in §VI-A and Fig. 8, take a relatively long time even under an ideal, whitebox setting as compared to fast recovery supported by REGEXNET. That is, if an adversary continuously launches an adaptive attack, REGEXNET can still quickly recover the

web service and make it at the target throughput most (i.e., >90%) of the time. More importantly, as prior adversarial training work shows, more adversarial examples will just make attacks harder, i.e., the longer time and with a higher chance of failure. As we have shown in §VI-A, the adversary fails to generate malicious requests for 40% of the cases during a continuous attack. To summarize, we believe that REGEXNET is resilient to continuous adaptive attacks.

VIII. RELATED WORK

Regular expression. Regular expression, a popular search pattern, is widely used in many scenarios, such as data manipulation and validation [1], [2], [32] and processing texts [33], [34]. There are some prior works that try to accelerate the matching algorithms of regular expressions. For example, Thompson et al. [16] propose a new regular expression search algorithm for better performance. Sidhu and Prasanna [35] rely on hardware, i.e., FPGAs, to accelerate regular expression matching. As a comparison, the purpose of REGEXNET is different, i.e., recovering an affected website after being attacked instead of finding and fixing the vulnerability beforehand. Such a task is important to bring an affected website back online in an incident while none of the prior works accelerating regular expression can.

DDoS attacks. Distributed denial-of-service (DDoS) attacks disrupt the normal operation of a target, e.g., with a flood of network traffic from many different sources. There have been extensive efforts contributed to DDoS attack detection and defense [36], [37], [38], [39]. First, several algorithms have been proposed to detect DDoS attacks. For example, Barford et al. [36] present an algorithm based on signal analysis. Moore et al. [38] use backscatter analysis to quantitively understand the nature of DDoS attacks. Jin et al. [40] propose to deploy hop-count filtering to detect DDoS attacks efficiently. Lakhina et al. [41] use traffic feature distributions to mine network anomalies. Second, researchers have proposed SDN/NFV-based methods, being orthogonal to algorithm-based methods, for defense. For example, FRESCO [42] provides modular composable security services in Software-Defined Networks (SDN). AvantGuard [43] can be used for developing more scalable and resilient SDN security services by introducing two data plane extensions, which are connection migration and actuating triggers. Lastly, Bohatei [44] introduces more flexibility and elasticity by setting up tag-based forwarding rules proactively. In contrast, REGEXNET focuses on protecting web services against low-bandwidth ReDoS attacks—none of the aforementioned works is able to do so.

ReDoS attack and defense. ReDoS is proposed as a new class of low-bandwidth DoS attacks that exploit algorithmic deficiencies [6]. ReDoS is also a common type of attack from a software engineering perspective. Lauinger et al. [45] show that the use of client-side JavaScript libraries may induce vulnerabilities to ReDoS attacks. Davis et al. [4] also explore super-linear regular expression engines that make ReDoS attacks possible in Python core besides JavaScript.

ReDoS defense is an important problem in academia and industry. Substack develops safe-regex [7], which detects ReDoS attacks by limiting the star height to 1. rxxr2 [8], developed by Rathnayake et al., presents a static analysis that forms powers and products of transition relations, and thereby reduces the ReDoS problem to reachability. Weideman et al. [46] apply results from ambiguity of non-deterministic finite automata to the problem of determining the asymptotic worst-case matching time. Wüstholz et al. [3] present Rexploiter that is able to automatically identify vulnerable regular expressions and determines whether a malicious input string can be matched against a vulnerable regular expression. Besides ReDoS attack detection, Merwe et al. [47] investigate techniques which can be used to transform vulnerable regular expressions into harmless equivalent expressions. Slowfuzz [9] and Singularity [10] proposed automated tools to explore algorithmic complexity vulnerability in a blackbox. Node.cure [17] modifies Node.js framework to enforce timeouts on different API usages and defend against ReDoS attacks.

As a comparison with existing work, REGEXNET is the first application-agnostic ReDoS recovery system by leveraging a DNN model to classify normal and malicious requests for web services. Prior works focus on defending against ReDoS attacks beforehand, but zero-day attacks may still penetrate and affect a vulnerable web service—REGEXNET can recover these web services that are under attack.

Attacks against learning-based systems. It is well known in adversarial machine learning [48], [49] that existing learning-based systems can be evaded [50], [51], [52], [53], [19]. REGEXNET is the same; however, the online feedback loop introduced in REGEXNET can correct these mistakes, e.g., evasive samples, so that the influence of such adaptive attacks introduced by evasive inputs will be limited to the same as a zero-day ReDoS attack—as shown in our evaluation, REGEXNET can also recover the affected web services from adaptive attacks under one minute.

IX. CONCLUSION

In conclusion, we present REGEXNET, a payload-based, recovery system to recover web services from zero-day ReDoS attacks. REGEXNET leverages the observation that requests triggering the super-linear running time of a vulnerable regular expression usually contain a particular string pattern, which can be learned using a DNN model. We design an online feedback loop for REGEXNET so that the DNN model is continuously trained and updated online based on data collected from web servers at runtime. We have implemented a prototype of REGEXNET, integrated it with HAProxy and Node.js, and demonstrated its effectiveness, responsiveness and resiliency with experiments on a testbed with real-world ReDoS attacks.

Acknowledgments. We thank our shepherd Giancarlo Pellegrino and the anonymous reviewers for their valuable feedback. Xin Jin (xinjinpku@pku.edu.cn) is the corresponding author. This work is supported in part by NSF grants 1813487, 1854000, 1854001 and 1918757.

REFERENCES

[1] "Regular Expression." https://en.wikipedia.org/w/index.php?title=Regular_expression.

[2] H. Hosoya, J. Vouillon, and B. C. Pierce, "Regular expression types for XML," in *ACM SIGPLAN Notices*, 2000.

[3] V. Wüstholz, O. Olivo, M. J. Heule, and I. Dillig, "Static detection of DoS vulnerabilities in programs that use regular expressions," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2017.

[4] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[5] C.-A. Staicu and M. Pradel, "Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers," in *USENIX Security Symposium*, 2018.

[6] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *USENIX Security Symposium*, 2003.

[7] "safe-regex." https://github.com/substack/safe-regex.

[8] A. Rathnayake and H. Thielecke, "Static analysis for regular expression exponential runtime via substructural logics (extended)," *arXiv preprint arXiv:1405.7058*, 2014.

[9] T. Petsios, J. Zhao, A. D. Keromytis, and S. Jana, "SlowFuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities," in *ACM Conference on Computer and Communications Security*, 2017.

[10] J. Wei, J. Chen, Y. Feng, K. Ferles, and I. Dillig, "Singularity: Pattern fuzzing for worst case complexity," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018.

[11] W. Meng, C. Qian, S. Hao, K. Borgolte, G. Vigna, C. Kruegel, and W. Lee, "Rampart: Protecting web applications from CPU-exhaustion denial-of-service attacks," in *USENIX Security Symposium*, 2018.

[12] J. C. Davis, F. Servant, and D. Lee, "Using selective memoization to defeat regular expression denial of service (ReDoS)," in *IEEE Symposium on Security and Privacy*, 2021.

[13] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu, "ReScue: crafting regular expression DoS attacks," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[14] C. Câmpeanu, K. Salomaa, and S. Yu, "A formal study of practical regular expressions," *International Journal of Foundations of Computer Science*, 2003.

[15] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, 1956.

[16] K. Thompson, "Programming techniques: Regular expression search algorithm," *Communications of the ACM*, 1968.

[17] J. C. Davis, E. R. Williamson, and D. Lee, "A sense of time for JavaScript and Node.js: first-class timeouts as a cure for event handler poisoning," in *USENIX Security Symposium*, 2018.

[18] "Outage postmortem - July 20, 2016." https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[19] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy*, 2017.

[20] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *ICLR*, 2015.

[21] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," in *International Conference on Learning Representations*, 2018.

[22] "HAproxy." http://www.haproxy.org/.

[23] "Node.js." https://nodejs.org/en/.

[24] U. Naseer, L. Niccolini, U. Pant, A. Frindell, R. Dasineni, and T. A. Benson, "Zero downtime release: Disruption-free load balancing of a multi-billion user website," in *ACM SIGCOMM*, 2020.

[25] "Apache Hadoop Distributed File System (HDFS)." http://hadoop.apache.org/.

[26] "PyTorch." https://pytorch.org/.

[27] "A fully functioning Node.js shopping cart with Stripe, PayPal and Authorize.net payments.." https://github.com/mrvautin/expressCart.

[28] "Redis data structure store." https://redis.io/.

[29] "Common Vulnerabilities and Exposures." https://cve.mitre.org/.

[30] "Apache HTTP server benchmarking tool." https://httpd.apache.org/docs/2.4/programs/ab.html.

[31] N. Papernot, P. McDaniel, A. Swami, and R. Harang, "Crafting adversarial input sequences for recurrent neural networks," in *MILCOM*, 2016.

[32] F. Yu, Z. Chen, Y. Diao, T. Lakshman, and R. H. Katz, "Fast and memory-efficient regular expression matching for deep packet inspection," in *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2006.

[33] M. Firtman, *Programming the Mobile Web: Reaching Users on iPhone, Android, BlackBerry, Windows Phone, and more*. O'Reilly Media, Inc., 2013.

[34] R. J. Ray and P. Kulchenko, *Programming Web Services with Perl*. O'Reilly Media, Inc., 2002.

[35] R. Sidhu and V. K. Prasanna, "Fast regular expression matching using FPGAs," in *FCCM*, 2001.

[36] P. Barford, J. Kline, D. Plonka, and A. Ron, "A signal analysis of network traffic anomalies," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, 2002.

[37] J. Mirkovic and P. Reiher, "A taxonomy of DDoS attack and DDoS defense mechanisms," *ACM SIGCOMM Computer Communication Review*, 2004.

[38] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage, "Inferring internet denial-of-service activity," *ACM Transactions on Computer Systems*, 2006.

[39] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, and M. Karir, "Taming the 800 pound gorilla: The rise and decline of NTP DDoS attacks," in *ACM SIGCOMM Conference on Internet Measurement Conference*, 2014.

[40] C. Jin, H. Wang, and K. G. Shin, "Hop-count filtering: an effective defense against spoofed DDoS traffic," in *ACM Conference on Computer and Communications Security*, 2003.

[41] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *SIGCOMM CCR*, 2005.

[42] S. W. Shin, P. Porras, V. Yegneswara, M. Fong, G. Gu, and M. Tyson, "FRESCO: Modular composable security services for software-defined networks," in *NDSS*, 2013.

[43] S. Shin, V. Yegneswaran, P. Porras, and G. Gu, "AVANT-GUARD: Scalable and vigilant switch flow management in software-defined networks," in *ACM Conference on Computer and Communications Security*, 2013.

[44] S. K. Fayaz, Y. Tobioka, V. Sekar, and M. Bailey, "Bohatei: Flexible and elastic DDoS defense," in *USENIX Security Symposium*, 2015.

[45] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web," *arXiv preprint arXiv:1811.00918*, 2018.

[46] N. Weideman, B. van der Merwe, M. Berglund, and B. Watson, "Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA," in *International Conference on Implementation and Application of Automata*, 2016.

[47] B. Van Der Merwe, N. Weideman, and M. Berglund, "Turning evil regexes harmless," in *Proceedings of the South African Institute of Computer Scientists and Information Technologists*, 2017.

[48] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence*, 2011.

[49] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The security of machine learning," *Machine Learning*, 2010.

[50] N. Šrndic and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *IEEE Symposium on Security and Privacy*, 2014.

[51] Y. Cao, X. Pan, Y. Chen, and J. Zhuge, "JShield: Towards real-time and vulnerability-based detection of polluted drive-by download attacks," in *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[52] M. Kearns and M. Li, "Learning in the presence of malicious errors," in *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, 1988.

[53] G. Wang, T. Wang, H. Zheng, and B. Y. Zhao, "Man vs. machine: Practical adversarial detection of malicious crowdsourcing workers," in *USENIX Security Symposium*, 2014.