Leader: Defense Against Exploit-Based Denial-of-Service Attacks on Web Applications

Rajat Tandon[1,2], Haoda Wang[1], Nicolaas Weideman[1], Shushan Arakelyan[1], Genevieve Bartlett[1], Christophe Hauser[1] and Jelena Mirkovic[1]
[1]University of Southern California Information Sciences Institute, Marina Del Rey, CA, USA
[2]Juniper Networks Inc., Sunnyvale, CA, USA
[1]{rajattan, haodawan, nweidema, shushana}@usc.edu, {bartlett, hauser, mirkovic}@isi.edu
[2]rajatt@juniper.net

ABSTRACT

Exploit-based denial-of-service attacks (exDoS) are challenging to detect and mitigate. Rather than flooding the network with excessive traffic, these attacks generate low rates of application requests that exploit some vulnerability and tie up a scarce key resource. It is impractical to design defenses for each variant of exDoS attacks separately. This approach does not scale, since new vulnerabilities can be discovered in existing applications, and new applications can be deployed with yet unknown vulnerabilities.

We propose Leader, an attack-agnostic defense against exDoS attacks. Leader monitors fine-grained resource usage per application on the host it protects, and per each external request to that application. Over time, Leader learns the time-based patterns of legitimate user's usage of resources for each application and models them using elliptic envelope. During attacks, Leader uses these models to identify application clients that use resources in an abnormal manner, and blocks them.

We implement and evaluate Leader for Web application's protection against exDoS attacks. Our results show that Leader correctly identifies around 99% of attack IPs, and around 99% of legitimate IPs across six different exDoS attacks used in our evaluation. On the average, Leader can identify and block an attacker after six requests. Leader has a small run time cost, adding less than 0.5% to page loading time.

CCS CONCEPTS

• **Security and privacy → Web application security**; Denial-of-service attacks.

KEYWORDS

Denial-of-service attacks, attack-agnostic defense, application-agnostic defense

ACM Reference Format:
Rajat Tandon[1,2], Haoda Wang[1], Nicolaas Weideman[1], Shushan Arakelyan[1], Genevieve Bartlett[1], Christophe Hauser[1] and Jelena Mirkovic[1]. 2023. Leader: Defense Against Exploit-Based Denial-of-Service Attacks on Web Applications. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, Hong Kong*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3607199.3607238

1 INTRODUCTION

Distributed denial-of-service (DDoS) attacks create a large disturbance to businesses and critical infrastructure services, resulting in large monetary losses [5, 12, 25, 52]. Traditionally, DDoS attacks generate a flood of traffic to deplete network resources at the target, and interfere with the target's service to its legitimate users [51]. As cloud-based defenses handle volumetric DDoS attacks, attackers shift their focus to sophisticated, attacks targeting application resources [47, 49]. Application-layer DDoS attacks can be effective at thousands of requests per second, consuming much lower bandwidth than volumetric attacks. Application-layer DDoS attacks are on the rise [21, 22, 24, 26, 50]. Recent statistics from Akamai [1, 2], show that the number of daily Web application attacks have seen a growth of more than 200% from December 2017 to October 2019. CloudFlare blog from 2022 finds a massive spike in application-layer DDoS attacks, specifically on Web servers [10], to coincide with Russia's invasion of Ukraine.

There are two broad classes of application-layer DDoS attacks: *flash-crowd attacks* [33, 53], which send high quantities of legitimate requests to the target (e.g. thousands per second), and *exploit-based attacks*, which exploit some vulnerability at the target application to bring it down with very low request rate (e.g., under 100 requests per second). This paper focuses on identifying and blocking exploit-based attacks (exDoS for short).

ExDoS attacks require very few requests to completely exhaust the application's resources and deny service to legitimate clients. In some cases the vulnerability in the application can be patched. For example, the application may use a weak hash table implementation to store user input, and the attack sends carefully crafted inputs that hash into the same slot. This leads to hash table collisions and slows down the application. Replacing the hash table implementation with a more secure one will patch the vulnerability. In other cases, the vulnerability is simply in the assumption of how an application will use a scarce resource. For example, operating systems support a limited number of simultaneously open sockets, on the order of several thousand. When a new client request arrives, the server creates a dedicated socket to process it. In case of legitimate requests this approach works well, since client/server connections are short-lived and the socket is quickly freed. Slowloris [62] exDoS attack, however, leads to a prolonged client/server communication and quickly depletes all available sockets. Since there are many applications, with known and unknown vulnerabilities, we cannot rely on patching only to address exDoS. For similar reasons, defenses against one variant of exDoS (e.g., Fitri et al. [17] and Choi et al. [9] approaches defend only against Slowloris) cannot fully address the problem.

We propose Leader, a novel *application-agnostic* and *attack-agnostic* defense against exDoS attacks. Our insight is that any exDoS attack must overuse resources of the target server to create DoS effect, regardless of the kind of vulnerability it exploits. We rely on monitoring resource use patterns per connection and per application to identify and block sources of exDoS attacks. The novelty of our approach lies in the Leader's monitoring of resource use patterns, which we call *connection life stages*. These connection life stages are built from multiple, complementary observations collected at the (1) network level, and (2) OS level as each external client request is handled. Those observations are further linked to application-level information, identifying the application's process and thread that handle the external client request.

Leader monitors all external requests for running services, and leverages connection life stages to build a fine-grained pattern of resource consumption by each service as it processes each request. During learning (in absence of attacks), Leader groups these patterns per application, and uses elliptic envelope to build the *application profile* – a model of legitimate resource usage patterns by the external requests sent by the application's legitimate clients. During classification, Leader classifies each ongoing connection as legitimate or attack, using the corresponding application profile to detect connections that consume resources in an abnormal manner. Sources of these connections (IP addresses) are blocked to mitigate the attack. Because Leader monitors all services and all resources its design is theoretically generic enough to protect various applications against various exDoS variants.

In this paper we focus on one popular class of application – Web applications. We narrowed our focus to one application class so that we could design realistic evaluation scenarios, which include mulitiple application implementations, realistic content served by the application, legitimate user's interaction with the application and a variety of exDoS attacks. Web applications come in many flavors – PHP coupled with popular servers, such as Nginx or Apache2, Javascript applications, Python-based applications, etc. Leader's design is application-agnostic, and thus can protect all these implementations. We leave exploration of Leader's use with application classes other than Web servers for future work.

We evaluate Leader on the Emulab testbed [58] using three popular Web server implementations – apache2, nginx and Flask [60], using realistic legitiimate traffic and content, and six different exDoS attacks: Slowloris attack [62], Hash Collision Attack [16], Regular Expression Denial of Service Attack (ReDoS) [54], the attack using preg_replace() PHP Function Exploitation (PHPEx) [56], Infinite recursive calls denial of service (IRC) [19] and Maliciously Crafted URL Attack (MCU).

Leader accurately identifies around 99% of attacker IP addresses, and around 99% of legitimate IP addresses. On the average, Leader can successfully identify and block an attack source after 6 requests. Leader has a minimal run-time overhead, and adds at most 0.5% to Web request processing time. Compared to related work – Rampart [30] and Finelame [13] – Leader does not require any modification to the source code of the applications it protects, and it achieves comparable or better classification accuracy on a wider range of attacks and server applications. Thus, Leader offers superior defense against exDoS.

2 EXDOS ATTACKS

Exploit-based denial-of-service attacks (exDoS) are challenging for defenses, because they exploit vulnerabilities in the target application's design or implementation. They craft legitimate-looking application requests, and are often effective at very low rates (e.g., 100s to 1000s of requests per second). Each attack variant exploits a different vulnerability through a completely different mechanism. As there are many target applications and potential vulnerabilities, it is hard to handle exDoS in a scalable manner.

We observe that many of exDoS attacks consume the resources of the target application or the underlying operating system in a manner that is different from legitimate users. For example, legitimate Web clients usually send their request to the Web server in a single packet, and then receive a reply. In a Slowloris attack [62], a variant of exDoS, the attacker starts but never finishes sending the Web request. This ties up a network socket for a long time, until all network sockets on the target server are depleted, which leads to denial of service. Similarly, when legitimate clients access a PHP page, they usually provide inputs that take a moderate time to process at the server. Conversely, attackers will craft requests to a vulnerable PHP page that will lead to lengthy processing or even an infinite loop [54]. This insight guides our approach to model how legitimate clients use server resources in a fine-grained manner, and to detect attackers as clients whose resource use departs from this model.

3 LEADER DESIGN AND IMPLEMENTATION

This section gives an overview of Leader's design and implementation. Our design and most of our implementation attempt to be generic and application-class agnostic. Some optimizations in our implementation focus on protecting Web applications from exDoS attacks.

3.1 Overview

Leader runs on the server that is being protected from exDoS attacks, and performs three distinct functionalities: (1) behavior profiling, (2) attack connection identification, and (3) attack mitigation.

Leader has two running modes: learning and classification. In the *learning mode*, Leader performs behavior profiling to learn *application profiles*. Each application profile models how legitimate users consume resources when engaging with the given application. This learning occurs offline, using traces of the application's operation in the absence of attack traffic. In general, the models of per-request resource consumption should not change when the server's content changes, but only if the nature of its service changes. For example, if an application server updates its content daily, it need not retrain Leader's models. However, if an application server used to serve static content, and now it started serving dynamic content, or if it used to support text messages between users, but now it supports video messages too, Leader's models should be retrained to capture new usage patterns. During learning, Leader should ingest traces

Table 1: Resource usage by the different stages of a sample legitimate and a sample attack connection in Figure 1 .

	sample legitimate connection						sample exDoS attack connection					
call	**dur**	**#calls**	**mem**	**CPU cyc.**	**pf**	**fd.**	**dur**	**#calls**	**mem**	**CPU cyc.**	**pf**	**fd**
SyS_getsockname	6.5μs	1	0KB	0.01M	0	0	16μs	1	0KB	0.01M	0	0
sock_recvmsg	789μs	4	0KB	0.1M	0	1	22,939μs	295	0KB	44M	1	1
sock_read_iter	34μs	4	1KB	0.03M	0	2	8,750μs	295	16KB	15M	0	1
sock_sendmsg	415μs	2	1KB	0.1M	0	1	752μs	2	1KB	0.1M	0	0
sock_write_iter	9.8μs	1	1KB	0.01M	0	1	32μs	1	1KB	0.01M	0	1
sock_poll	2,491μs	3	0KB	3M	0	0	11,073,328μs	97	0KB	55M	0	0
sockfd_lookup_light	53μs	3	0KB	0.01M	0	0	120μs	3	0KB	0.01M	0	0
Sys_shutdown	62μs	1	0KB	0.01M	0	0	101μs	1	0KB	0.01M	0	0

over many time periods to ensure that we learn diverse behaviors of legitimate users.

When Leader's models are learned, it switches to *classification* mode, where it continuously performs attack connection identification. For each external request, Leader compares resource use patterns of the application that processes the request against its application profile, and classifies the request as legitimate or attack. Sources of attack requests are sent to the mitigation module for blocking.

Behavior profiling. In absence of attacks, Leader runs in *learning mode*, building models of how legitimate users consume each application's resources.

Leader observes the process of serving each connection as a sequence of *connection life stages*. A connection life stage is defined as specific resource usage (i.e., *time*, *memory*, *CPU cycles*, *page faults*, and *open file descriptors*), quantified by *amount* of resource used, as a result of serving the incoming request. Each connection life stage relates to a specific function call in (`net/socket.c`), issued in the process of serving the request.

Leader employs machine learning to build an aggregate baseline model for each application and for each connection life stage.

Attack request identification. Once Leader learns the baseline models, it switches to running in the *classification mode*. During classification, Leader works on live data, applying its models to identify attackers.

We considered running Leader in the learning mode until an attack is detected. Such on-demand engagement of classification would limit any false positives. But continuous classification had the advantage of early attack mitigation, even in the case of stealthy attacks. Extremely low rate attacks, such as PHP Infinite recursive calls denial of service [19] (discussed in detail in Section 4.1) can create load on a Web server, which is equivalent to 0.53 million requests, using just a single request. Running Leader in continuous classification mode enables us to identify and block the attacker after even a single malicious request.

Attack mitigation. Sources whose service requests consume resources in a way that deviates from Leader's models are identified as attackers. In our current prototype, we mitigate attacks by blocking IP addresses of the attackers. However, alternative mitigation approaches are possible, such as: (1) derivation of payload signatures from attack connections and their use in a firewall with deep packet inspection, (2) connection termination (e.g., via TCP RST), (3) dynamic resource replication, (4) program patching and algorithm modification. We leave exploration of other mitigation approaches as future work.

3.2 Assumptions and Limitations

As discussed in Section 2, Leader aspires to handle many variants of exDoS attacks. This includes attacks that exploit vulnerabilities at application, operating system, and protocol levels. Leader does not handle flash-crowd attacks [53] that simply send more requests per second than the server can handle, but do not consume resources in a manner that differs from legitimate users'. We consider that a powerful remote attacker (i) can send arbitrary requests to a service hosting a vulnerable application, and (ii) is aware of the application's structure and vulnerabilities. Attackers can target CPU, memory, file descriptors, bandwidth or other limited resources in the host system.

We assume that each incoming connection carries one or more application requests. Our connection life stages model how *all* requests on the given connection are being processed by the application, and how they consume resources on the server. For simplicity, we use terms "request" and "connection" interchangeably in the rest of the paper.

Leader attempts to quantify resource usage per an incoming connection to the server by measuring resource usage of processes and/or threads that serve requests received on this connection. Typically, application servers either spawn off a process or start a new thread to handle each incoming connection. This is also the case for Web server applications we tested – apache2, nginx and a web server written in Flask framework. It is possible to design applications so that they process multiple incoming requests in a single thread. This would make it harder to tease apart which resource consumption is due to which connection. We leave handling of this *shared processing scenario* for future work.

We assume an attacker model where remote attackers cannot overwrite system binaries or modify the kernel of the system running Leader, i.e, we assume that Leader process is always trustworthy and engaged.

3.3 Implementation

In this section we describe how we implemented Leader. Leader contains five modules, illustrated in Figure 2 , that are each responsible for a different functionality in exDoS attack mitigation. During the learning phase, the Prober Module collects the data needed for learning and passes it to the Builder Module. The Builder module keeps building connection life stage sequences. Each sequence is a snapshot of the associated request's resource consumption up to the given moment in time. These snapshots are sent to the Learner module to generate the baseline model of legitimate client behavior. Together, Prober, Builder and Learner modules are engaged in behavior profiling. The baseline model is used by the Scoring module in the classification phase to classify connections as legitimate or attack. Finally, source addresses of connections that are classified as attack are sent to the Mitigation module to mitigate the attack. We explain each of these modules in more detail next.

Prober module. We use SystemTap [48] to trace and log in real time in the kernel all function entries and exits in the socket library (`net/socket.c`) in the Prober module. The module also records the resource usage during each of these function calls. SystemTap dynamically inserts our selected probes into kernel code using Kprobes, and it is a reliable, widely used profiling tool. Another option would have been to use extended Berkeley Packet Filter (eBPF) [55], which is lighter-weight and safer to use. We leave exploration of eBPF for future work, but note that newer versions of SystemTap can internally use eBPF [36]. Thus we expect that our implementation could easily switch to eBPF.

We use SystemTap to probe the application's processing of incoming service requests at function call level. This does not require any modifications to applications, but only to the server's operating system to install the loadable kernel module. SystemTap provides the functionality to log the function call, entry and exit timestamps, in micro-seconds precision, and the thread and process identifiers (the task group ID of the current thread) associated with the function call. For each function call, SystemTap also allows us to log the number of CPU cycles, page faults that occurred, file descriptors opened, the amount of memory used, as well as the associated source IP address and source port number. One can use log rotation techniques [61] to limit the size of the real-time SystemTap logs.

Builder module. Using the data collected by the Prober module, the Builder module builds life stages for the given incoming connection.

In our prototype, we use the tuple <thread id, process id> to uniquely identify a given external (incoming) connection to the application at a given time. We later link this tuple to the source IP address and source port of the external client, which we obtain from the arguments of the sock_recvmsg call. We use the process table to map the <process id> to the application name. A connection's life stage corresponds to one function call of `net/socket.c` and the resource usage (CPU cycles, page faults, file descriptors and memory) measured by SystemTap for handling this call. As time progresses, recent life stages are linked to the preceding ones. Thus, each life stage pattern is actually a snapshot of the function call sequence and resource usages from the start of the given external connection up to the given moment in time. The initial call to sockfd_lookup_light marks the start of the sequence, and it ends eventually with the call to one of the following functions: SyS_shutdown, sock_destroy_inode, __sock_release or sock_close. If serving a request requires access to the database, cache, and/or any other internal services, the internal connection's life stage sequence is integrated into the main external connection's life stage sequence. We can link all these connections together if they are served by the same process/thread that accepted the external connection, or if the accept process/thread spawned the processes/threads for the internal connections. However, we currently cannot account for internal resource consumption that occurs when accept process/thread passes jobs via queues to another, already running internal thread. We leave this for future work.

After each second, we snapshot each connection's life stage sequence. If Leader is in the learning mode, we pass this snapshot to the Learner module to create the baseline model. If Leader is in the classification mode, we pass the snapshot to the Scoring module. Figure 1 illustrates the connection life stages corresponding to serving of one sample legitimate (green line) and one sample attack (red line) request. For the given incoming request, the resource usage for each life stage at that time by the given process and/or thread serving that request is shown in Table 1. The portion highlighted in red color shows the portion of the life stage sequence that differs between the attack connection and the legitimate connection. In this example, the attack connection loops many times between the calls sock_read_iter and sock_recvmsg. It occasionally departs from this loop and cycles through sock_send_msg, sock_write_iter, and sock_poll, then goes back to the loop. This pattern produces both abnormal time spent in a given life stage and abnormal number of visits to a given life stage.

Learner module. This module deploys machine learning to learn separate baseline models for the following resources spent per select connection life stage: *system time*, *memory*, *CPU cycles*, *the number of page faults* and *the number of open file descriptors*. This model also includes the number of visits for select life stages and the presence or absence of some specific life stage transitions.

Learner first groups all the snapshots for a given application. We mine the following features from each snapshot to build our baseline model. Each feature contains measures listed below, for life stages corresponding to the select function calls: sock_write_iter, sock_read_iter, sock_recvmsg and sock_poll. The features we track are: (a) total system time elapsed (in microsecond precision) in the stage, (b) total memory consumed in the stage, (c) total CPU cycles elapsed in the stage, (d) total page faults in the stage, (e) total number of unique open file descriptors in the stage, (f) number of visits to the stage and the presence or absence of two stage transition sequences (s1) sock_read_iter → sock_sendmsg, (s2) sock_read_iter → sock_poll. We learn separate models for (a) – (f) and later utilize these models for classification. While it is possible to use (a) – (f) together to learn a single baseline model, that leads to overfitting, because of high dimensionality [59].

When deciding which function calls from `net/socket.c` to include in our features, we first examined all possible calls. However, only the select four function calls (see Table 2), were showing differences between legitimate and attack connections. These four function calls are involved in starting the connection, receiving service requests and sending the replies back to the client. When a service request ties up resources at the server, this becomes visible in the time elapsed and the resources consumed in these selected life stages, or in the increase of the frequency of visits to these stages. This way, we abstract the details of the server and the vulnerability being exploited. We further chose to track the presence and absence of two key life stage transitions: sock_read_iter → sock_sendmsg and sock_read_iter → sock_poll. The first transition indicates receiving a remote client's request and responding to the

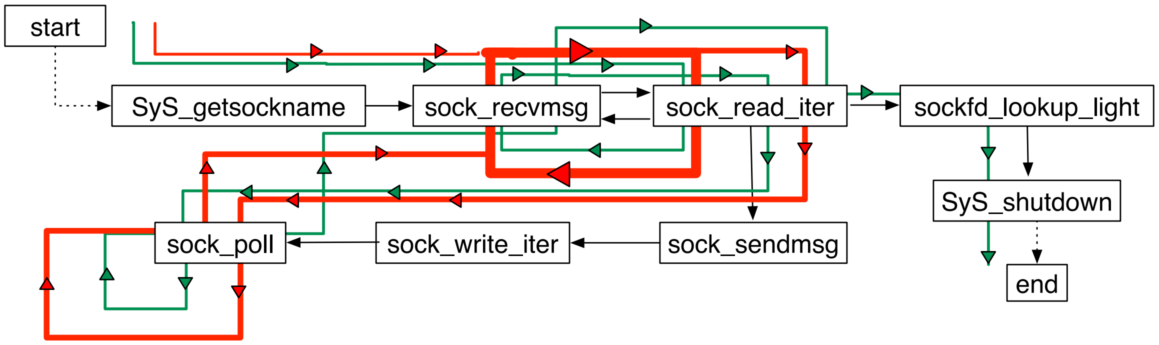

Figure 1: Sample legitimate and attack connection life stages. Table 1 shows the time of each stage, how often it is visited in the sequence, memory used, number of CPU cycles elapsed, number of page faults that occurred and the number of open file descriptors for each stage. The red portion shows the sequence of calls in an exDoS attack that are different from the calls observed on a legitimate connection. Different exDoS attacks may follow different sequences and consume different amount of resources at different calls.

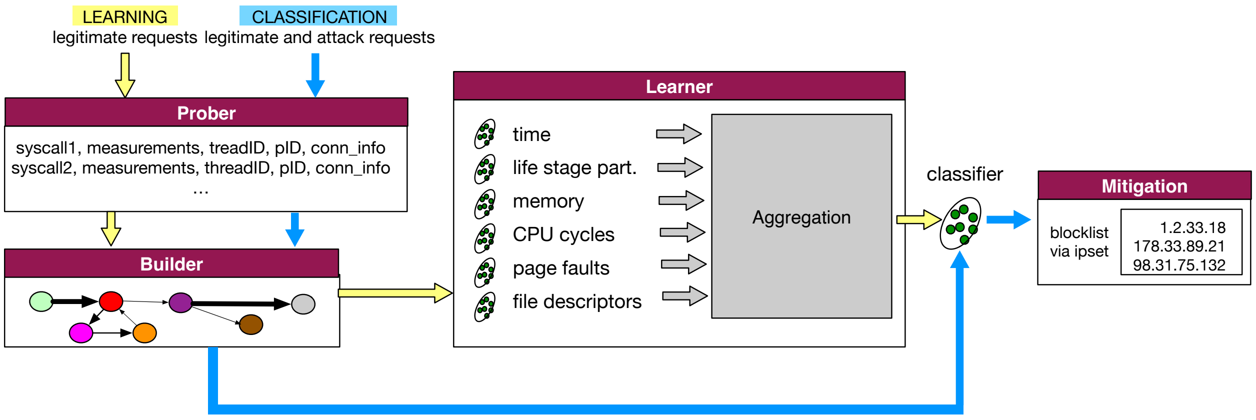

Figure 2: Leader's operation: learning and classification

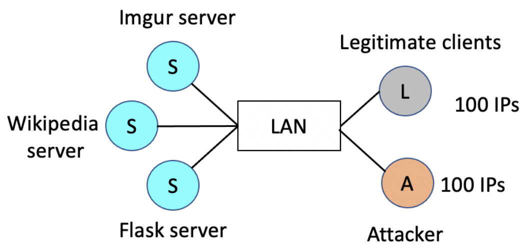

Figure 3: Experiment topology in Emulab: 5 physical attackers (100 virtual attackers), 1 legitimate client (100 virtual clients) and 3 servers on a LAN.

client. The second transition indicates continuous read from the socket to obtain a large message from the client. Tracking more sequences leads to overfitting, which makes it harder to detect new attack variants. Tracking no sequences (or just one of the two we chose) leads to misclassifications of some attack connections as legitimate.

We standardize our features (by removing the mean and scaling to unit variance). We then apply machine learning to learn the baseline model. We focus on single-class learning, because we wanted to model only legitimate traffic. Modeling only legitimate traffic enables Leader to be *attack-agnostic* and potentially effective against a variety of exDoS attacks. We evaluated 1-class SVM [27] and elliptic envelope [38, 40] for our model building, and the elliptic envelope had superior performance (see the Appendix, Table 8).

Elliptic envelope [38, 40] is an outlier detection approach, which models target features as Gaussian distributions. We verified that our features follow half-normal distribution, which is a subclass of Gaussian distribution. Elliptic envelope gives a robust co-variance estimate of the data. It defines the shape of the data, creating a frontier that delimits the contour, that is elliptical in shape. Basically, it fits the tightest Gaussian (smallest volume ellipsoid) that it can over the data points, while discarding some fixed fraction of outlier points specified by the user [37]. This Gaussian forms a decision boundary. The trained model stores the estimated co-variance and the decision boundaries for each feature.

Table 2: Leader tracks the highlighted function calls or uses them to identify a sequence of calls.

candidate function calls	
sock_lookup_light	sock_alloc_inode
sock_alloc	sock_poll
sock_alloc_file	move_addr_to_user
SYSC_getsockname	SyS_accept4
SYSC_accept4	SyS_getsockname
sock_write_iter	sock_sendmsg
sock_read_iter	sock_recvmsg
__sock_release	Sys_shutdown
sock_close	sock_destroy_inode

In current implementation, the duration of our learning stage depends on the size of our training set. In actual deployment, learning should end when the model has stabilized, and no further changes are detected with new input samples. Section 8 lists the amount of time it requires to train our model for different sizes of the training sets.

Scoring module. After learning, Leader switches to the classification mode. In classification mode, the Scoring module uses elliptic envelope to classify each connection as either a legitimate connection or an attack connection. Elliptic envelope uses the Mahalanobis distance in identifying outliers, which has been widely used in literature for identifying outliers in multivariate datasets [11]. During classification, elliptic envelope computes the Mahalanobis distance for each input, using the co-variance learned by the model. If this distance lies within the decision boundaries, the input is considered to be in line with the model. Otherwise it is considered anomalous. The same process is repeated for all six individual models ((a) – (f)) that we learned during the learning phase. We aggregate individual model classifications into a single output, by classifying a connection as anomalous if at least one model returns anomalous classification.

Each connection is classified every second so as to terminate the attack connections as soon as possible. We considered and evaluated two designs for source classification:

- *Liberal design* assumes that each anomalous connection is attack connection. This approach ensures fast decision time, but if there are any errors in classification, a legitimate source may become blocked by the module.
- *Conservative design* requires that a connection receives some portion of anomalous classifications before it is regarded as attack. This approach should reduce misclassification of legitimate connections, at the expense of longer decision time.

When scoring module identifies a connection as attack, it forwards its source IP address to the mitigation module.

Mitigation Module. Our current implementation of the mitigation module adds each IP it receives from the scoring module to the IP blocklist. We use the *ipset* utility in the Linux kernel to implement the blocklist. Blocking rules remain in place for a custom duration (e.g., 10 minutes), which can be configured by the system administrator. Long attacks would thus lead to cyclical blocking of attack IP addresses.

3.4 Deployment Considerations

We now discuss some practical considerations for deployment of Leader to protect a Web server.

User identification. While we use IP addresses for identifying attack sources, using Web cookies could improve classification accuracy, because it would more accurately identify users with dynamically assigned IP addresses, or users which may share an IP address with others (e.g., behind a network address translator). This optimization is specific to Web servers, as other applications may not have an equivalent of a cookie. We leave this optimization for future work.

Adversarial data. Leader requires training data of legitimate connections and needs to be trained per application server. Leader requires training data that covers periods of both low and high server load. This is because many servers may go through a different sequence of system calls when the load is high, engaging additional optimizations. High load may also slow down an application's processing speed, since it may have to wait for resources to be made available for use in order to finish its processing. Such disturbances, may also lead the legitimate connections' resource consumption patterns to deviate, which is important to capture in the models. The richer data we have for training, the better we can protect legitimate connections from misclassifications (Section 8). Attackers could potentially compromise the training process and introduce adversarial data to influence learned models. For example, if low-rate attacks were introduced during training this could make our models unable to accurately detect attacks during classification. One way to handle this threat is to sample training data at random, over multiple days or weeks. Another way is to exclude outliers, by setting an outlier fraction (contamination parameter) value while training the baseline model. A third approach to handle this threat is to use techniques such as machine unlearning [6]. We evaluate the effect of adversarial data injection for an attack scenario in Section 8.

Deployment cost and complexity. We envision that Leader would be deployed at an application server, as a standalone application. Leader could run at the server at all times, since it is non-intrusive and not in line with traffic.

Web servers that use a proxy or a load-balancer. Web servers that use a proxy would need to run Leader on all the backend servers, as well as on the proxy. The proxy would have to be trained separately from the backend servers, since they experience different resource use patterns when handling each connection. On identification of a malicious connection at the backend level, the backend server would have to signal to the proxy to mitigate the attack (e.g., block the attack source). Web servers that use a load balancer only need to run the Mitigation module on it, and Scorer modules from backend servers need to communicate attack sources to this Mitigation module.

Windows-based servers. Currently, our prototype supports only Linux and Unix-like operating systems, because our implementation uses Linux-specific tools (SystemTap, ipset, etc.) We leave exploration of how to port Leader to other operating systems for future work.

Attackers spoofing IP addresses. We focus on Web server exDoS attacks. The applications use TCP for communication, which requires a successful 3-way handshake for payload exchange, and exDoS attacks must use specific payload to exploit a vulnerability. Thus IP spoofing cannot be used in exDoS attacks on Web servers.

Attackers use many sources. Distribution of attack traffic across many sources does not affect Leader, because our models model per-connection behavior, and each exDoS connection will use resources in abnormal manner. Distribution could affect mitigation if the attacker can easily move on to new sources after Leader blocks the old ones. While this is possible, it is not trivial for the attacker. New sources have to be acquired (e.g., rented), set up and synchronized. Depending on attack's duration, and because our detection delay is low (around 6 attack requests), the attacker would use only a small fraction of their botnet at each time period, while spending effort and money to maintain a large infrastructure. A target protected by Leader would be far less attractive for the attacker, than a target protected by conventional DDoS defenses.

Sharing the same IP. If an attacker shares an IP with a legitimate user of the target server, which we expect to be rare, both would be blocked. This is unfortunate, but necessary to protect the server and enable it to serve any legitimate users.

Shared environment and network delays. An attacker could attempt to make Leader misclassify legitimate connections as attack by creating heavy load on the server, using non-exDoS attack vectors such as volumetric DoS attacks. Each application runs as a separate process/thread on the server and we monitor time each given process/thread spends in a given function call, as it has control of the CPU. Thus we measure an application's system time (not real time) to handle a given processing task. This time does not include network delays nor is it impacted by other applications on the same host. For this reason, an attacker cannot influence legitimate connection classification by generating heavy load on the server.

4 EVALUATION

In this section we describe our evaluation setup and our experiments, and we demonstrate how Leader defends against exDoS attacks.

4.1 Evaluation Setup

To create realistic Web application traffic, we wanted to mimic realistic diversity of static and dynamic content, the processing time of dynamic content, and the realistic user behavior. Realistic content and user interaction are important for evaluation, because this diversity of legitimate requests makes it more challenging to identify exDoS attack requests.

Realistic content. To obtain realistic content, we mirror *dynamic* content for two popular Web sites, for which the server setup is publicly available and content is copyright-free: Imgur, a picture-rich Web site, and Wikipedia, a Web site with textual content. Our selection of Web sites to replicate gave us not only content diversity, but Web server diversity as well. Imgur runs on the apache2 HTTP Server. Wikipedia could handle more requests per second with nginx compared to apache2, so we used nginx in our tests. Different content on sites and different implementations are likely to lead to different legitimate user access patterns, and processing load on the server. We download each full site and deploy the site's original configuration and scripts on our server within Emulab testbed [58]. During evaluation, all site content is generated dynamically, by pulling page information from our target server's database, using the original site's scripts. We enrich this setup with a basic, vulnerable Flask-based server to test one specific attack variant – maliciously crafted URL attack on Flask (MCU).

Realistic legitimate traffic. To generate realistic legitimate requests, we first crawl the full Web sites using the Selenium-based [41] crawler, to learn all possible legitimate request patterns. For dynamic pages, we analyze what kind of arguments they require (e.g., string vs integer) and fuzz the inputs during crawling. Additionally, we engage 350 users from Mechanical Turk service to browse our Web sites and we collect their requests to further enrich the legitimate request patterns. We have a total of over 300K legitimate requests for Imgur and over 500K legitimate requests for Wikipedia. We utilize 70% of the data for training and 30% of the data for testing. We generate legitimate requests by replaying selected records from our datasets in a controlled environment—the Emulab testbed [58]—and launch attacks on our servers on the testbed. Each site we use is vulnerable against Slowloris attack without modifications. To make sites vulnerable to other variants of exDoS attacks, we add five Web pages with vulnerabilities, one per attack variant.

Legitimate traffic generator. During each experiment, we replay the 30% of the legitimate requests that we set aside for testing, which we had collected using (a) the selenium-based [41] crawler, (b) Amazon Mechanical Turk workers. We wrote a custom traffic generator, which extracts URL sequences and their timing from our testing dataset, and then chooses when to start each full-length sequence depending on the desired number of active users. For example, if we want to have 10 active users at all times, our tool will extract 10 full-length sequences from our dataset, and replay each using a different source IP address. We adjust IP addresses assigned to machines in our experiments, and the routing, to ensure that each machine can maintain two-way communication with each server. Traffic is replayed at the application level. Thus any packet drops are properly handled by the underlying TCP protocol. When a sequence completes, another sequence is selected and another IP address becomes active. In our experiments, we maintain 100 active, simultaneous legitimate clients throughout each run, that send requests at the rate of 1 request per second.

Experiment topology and scenarios. Our experimental topology is shown in Figure 3. It has 1 physical attack node (that emulates 100 virtual attackers wherein each virtual attacker is assigned a different IP address), 1 legitimate client node (emulating 100 legitimate users at a time wherein each legitimate user is assigned a different IP address) and 1 node each for the mirrored servers Imgur, Wikipedia and Flask. All nodes were of type d820 on Emulab, with 32 cores and the Ubuntu 18.04 OS. We further fine-tuned operating system parameters on nodes to maximize the request rate that each client could generate, and to maximize the request rate that our servers could handle. We evaluate Leader by launching various exDoS attacks during classification. Each virtual attacker (one distinct source IP) sends attack traffic at a low rate of one request per second. We use the aggregate attack request rate similar to that of the aggregate legitimate request rate, i.e., close to 100 requests per second. This means that 100 virtual attackers are active at the same time in each experiment.

Attack variants. We investigate the following exDoS attacks:

(1) *Slowloris attack (SL)* [62]: This attack uses partial HTTP requests to open connections between the attacker and the targeted Web server. The attacker keeps those connections open for as long as possible, thereby overusing socket descriptors at the target, which makes the target unable to serve legitimate clients.

(2) *Hash Collision Attack (HC)* [16, 57]: A hash table is used to store a series of keys and values, and a hash function is used to convert each key to a bucket where the value will be stored. When the function maps two different keys to the same bucket, there is a *collision*, which is usually handled by employing a less efficient structure to store multiple key/value pairs, such as double chaining or linked list. Buckets with many collisions experience much slower lookup, insert and removal of values. If a Web application uses a predictable (weak) hash function and derives keys from user input, attackers can craft Web requests with colliding keys, thus dramatically slowing down the server.

(3) *Regular Expression Denial of Service Attack (ReDoS)* [54]: In a ReDoS, attackers force a situation where the regex evaluator, for example, `preg_match()` for PHP, gets stuck evaluating a string and runs for a long time. The inordinately long run time is caused by backtracking, as multiple matches are attempted. If a Web application employs regex matching on user input, and the attacker knows the specific regular expression used, they can craft inputs that take inordinately long time to process, thus causing ReDoS.

(4) *Attack using preg_replace() PHP Function Exploitation (PHPEx)* [56]: The PHP function `preg_replace()` can lead to a remote code execution vulnerability if the Web application passes user input to `preg_replace()` and if that input includes executable PHP code. This vulnerability can be used to launch an exDoS attack, if user input includes many PHP function calls. PHP versions prior to version 5.5.0 have this vulnerability. In our experiments, we used PHP 5.3.29 to reproduce this vulnerability.

(5) *Infinite recursive calls denial of service (IRC)* [19]: Passing a PHP file as an argument to itself can in some cases lead to infinite recursive call. If the Web application passes user input to a PHP file, the input containing that file's name can trigger the IRC attack. For example, `article.php? file=article.php` is an illustration of an attack request. The Web application runs until the Web server hits the maximum execution time for a script and terminates.

(6) *Maliciously Crafted URL Attack on a Flask Application (MCU)*: Flask [60] is a popular Python Web framework. It is a third-party Python library used for developing Web applications. We crafted an implementation of a vulnerable Web application in Flask to demonstrate that Leader supports a variety of Web server applications and frameworks. Our application is shown in Appendix, Section 11.1.

Attack traffic generator. For different attack scenarios, we use different attack tools. We modified the source code of the tool PySlowLoris [43], to launch the Slowloris Attack. Similarly, we modified the source code of the tool php-dos-attack [28], to launch the Hash Collision Attack. In each case we use multiple IP addresses on the same physical node to emulate multiple attackers. For other exDoS attacks, our attack traffic generator is a modified `httperf` [31] tool. We added the ability to choose source IPs from a pool, and to select requests for each IP from a given sequence, in specified order. Thus, our evaluation presents Leader's results on different exDoS attacks launched by multiple attackers, each using a different IP address. Leader's evaluation setup is comparable to, and in some cases more sophisticated than, the evaluation setup used by closely related prior work, shown in Table 3.

4.2 Limitations

There are several limitations of our evaluation scenarios. First, our topology is small and this limits the scale of our tests (e.g., number of requests per second and how many separate IPs we can emulate). Since we focus on low-rate, exDoS attacks, this limitation does not impact realism of our evaluation.

Second, we replicate only two Web sites' contents and the Web server applications – apache2, nginx and a Flask-based application. While we would have liked to replicate other popular sites, like Facebook or Google, this impossible as their source code is private, very complex and supported by large datacenters. Our evaluation setup is comparable to other published work on this topic (e.g., Rampart [30], Cogo [15] and Finelame [13]). Evaluation on more applications would require us to identify more flavors of exDoS attacks for the specific target application, and collect a diverse, realistic set of legitimate traffic that interacts with the realistic content of the target application using remote volunteers. We investigated this direction, but could not identify sufficient exDoS attack variants for non-Web applications, and thus we leave it for future work.

Third, a portion of our legitimate users' traffic comes from our Amazon MTurk study, and it may not be realistic. However, Leader models do not rely on a user's request sequence, but on the diversity of processing times and system calls for the pages visited by legitimate users. In our study users explored many of the pages on the replicated sites, and thus provided a good blend of request variety. Our full crawl of the Web sites provided a comprehensive dataset to complement Amazon MTurk dataset.

5 RESULTS

In this Section we show results for Imgur running on apache2 (and Flask for MCU attack). Results for the Wikipedia server running on nginx show similar findings.

5.1 The *Liberal* Design Scenario

In the liberal scenario, Leader has an aggregate accuracy of 99% to 100% in identifying the attacker IP addresses and an aggregate accuracy of 96.6% to 99.9% in identifying the legitimate IP addresses, across the six exDoS attack scenarios used in our evaluation. We summarize the results in Table 4. On the average, Leader can identify and block an attacker after 1–2 requests in the liberal design scenario. Attack variants ReDoS, PHPEx and IRC are more challenging for Leader, resulting in false positives (legitimate connection identified as attack) higher than 1%. Since such high false positives are undesirable, we conclude that liberal design scenario is impractical for real Web server deployment.

Table 3: Comparing Leader's evaluation setup with closely related prior work.

defense	applications tested	machines used	leg. requests	attacks
Rampart [30]	Apache2 on Drupal, Wordpress	2 (1 for server; 1 for generating both legitimate and attack traffic)	Crawled all the endpoints of each web application requests for replay, combined with humans generated requests. Legitimate traffic involves up to 128 user instances, with at least 0.1s pause between any 2 requests	Tested attack variants: XML-RPC and PHPass. It maintains up to 30 attacker sessions, with 0.17 requests/second to 1 requests/second
Finelame [13]	Apache2, Node.js, DeDoS	2 (1 for server; 1 for generating both legitimate and attack traffic)	Generated using Tsung [32] (250 requests/second to 750 requests/second)	Tested attack variants: ReDoS, Billion Laughs and SlowLoris. Attack Rate: 1 to 15 malicious requests/second
Cogo [15]	Apache, OpenSIPS	3 for Apache (1 for server; 1 for generating legitimate and 1 for generating attack traffic)	Generated using Choi-Limb [8] (100 benign clients (up to 100 requests/second)	Tested attack variants for Apache: Slowloris and Slowread. Attack Rate: 1 to 100 connections/second
Leader	Apache2 on mirrored server Imgur, Nginx on mirrored server Wikipedia and a Flask application	3 (1 for server; 1 for generating legitimate and 1 for generating attack traffic). We use multiple IP addresses on the same physical node to emulate multiple attackers and legitimate clients.	Crawled the full Web-sites of each web application to collect legitimate requests, combined with humans generated requests fetched using Amazon Mechanical Turk workers. 100 active, simultaneous legitimate clients send requests at the rate of 1 request/second.	Tested attack variants: SL, HC, Re-DoS, PHPEx, IRC and MCU. Attack Rate: 100 requests/second.

5.2 The *Conservative* Design Scenario

Our conservative classification approach reduces false positives, by only classifying a source as attack if its connections receive more than a certain percentage of anomalous connection classifications. We define a *classification threshold* as the percentage of *additional* anomalous connections (after the first anomalous connection) from the same source, required for the source to be labeled as attack. Further, we must observe some minimum number of connections from a given source before we apply our classification threshold to classify the source. We call this value *minimum sequence length*. For example, if our minimum sequence length is 3 and our classification threshold is 0.5 we must observe at least 3 connections from a given source before we attempt to classify it. Assuming that there is one anomalous classification, we need another one (1 out of additional 2 connections) to label this source as attack. We plot ROC curves for the different sequence lengths and different values of classification threshold in the Appendix, Figure 6 for sequence lengths 3, 4, 5, 6 and 7. The area under the ROC curves is the largest for sequence length 6 and stays the same for sequence length 7. Figure 4 in the Appendix, shows the optimal classification threshold versus sequence length, based on the cutoff points. While the optimal classification threshold is high for shorter sequences, it dips as the sequence length increases. Since the area under the ROC curves is the largest for sequence length 6, we use the corresponding classification threshold of 20% in our evaluation. Leader achieves around 99% accuracy for both legitimate and attack source identification, but it takes about 6 requests per source for classification. This is expected as we trade off detection delay for higher accuracy.

5.3 Feature Selection

To understand the importance of our selected features (see Section 3.3), we further investigated if only system time (just feature (a)) could be enough to detect and block attack connections. We repeated our experiments with liberal design and trained and tested using only feature (a). Time-only detection was indeed sufficient to detect all attackers in our scenarios, but detection was delayed around 1–2 seconds (11% of time to detect an attacker), when compared to the all-feature model. We then hypothesized that time-based detection may struggle when durations of legitimate and attack connections were similar. We modified the Slowloris attack to terminate its connections after five seconds, thus putting the total connection duration in range with some computationally-expensive legitimate connection durations in our dataset. We also increased the frequency of Slowloris attack connections to keep the load on the server comparable to our original attack scenario. Time-only detection failed to detect any attack connections, as expected, while all-feature detection was successful in identifying and blocking all attackers.

6 COMPARISON WITH RELATED WORK

We evaluate two of the closely related works using the same experimental setup and attack scenarios that we use for Leader's evaluation.

***Rampart* [30]**: Meng et al. present the defense mechanism Rampart, that protects PHP-based Web applications from sophisticated CPU-exhaustion DoS attacks. Rampart uses statistical methods and function-level program profiling. It builds a statistical model of the CPU time consumed by each function call of the application process, and uses Chebyshev inequality to detect when a function spends more CPU time than its model indicates. These function calls are specific to the protected application, unlike our function calls that capture any invocation of functions in `net/socket.c`, i.e., any socket communication on the server, and thus protect a variety of Web applications in our evaluation. Rampart probabilistically terminates suspicious connections after they overspend CPU time, and can also devise source-IP blocklists and payload signatures for future attack filtering. Leader differs from Rampart in the way it models legitimate connections (system function calls and wall-clock time and call frequency vs Rampart's CPU time and application function calls), and in the way it detects anomalies (elliptic envelope vs Rampart's statistical detection). Because Leader models system function calls it is more broadly applicable to protect any Web application on the server.

We evaluate Rampart [29] using the same settings as for Leader's evaluation. Table 6 shows the accuracy of Rampart over different attack scenarios. While Rampart can handle HC, Re-DoS and IRC very well, it fails to identify attackers in the SL attack. Rampart only identifies the anomalies related to the CPU time spent inside each PHP function, but exDoS attacks do not have to overspend CPU time to deny service. For example, in the SL attack scenario, each attack connection to the server remains idle waiting for the complete request to arrive in order for it to be processed. Moreover, Rampart works only on PHP-based servers and the current code [29] supports PHP 5.6 and PHP 7.0. This is the reason why Rampart fails to detect our PHPEx and MCU attacks. Leader does not have such restrictions and dependencies. To summarize, while Leader and Rampart have a comparable false positive and false negative rates, Leader detects and mitigates a wider range of exDoS attacks using more flexible models.

***Finelame* [13]**: Demoulin et al. propose Finelame [13], which leverages the operating system's visibility across the entire software stack to instrument key resource allocation and negotiation points. It leverages extended Berkeley Packet Filter to attach application-level probes to some of the key request processing functions. Finelame trains a K-means-based model of resource utilization, which includes models of time spent in each application-level function, memory use, file descriptor use, page faults and total time to serve a request. The model parameters are then shared with the resource monitors, which use them to detect exDoS attacks and produce alerts containing process and thread IDs. Leader compares favorably to Finelame. Leader is easily portable to different applications, while Finelame requires instrumentation of each specific application (e.g., it would instrument apache2 separately from nginx and Flask in our experiments), and adds 4–11% instrumentation overhead. Further, Finelame does not provide mitigation, while Leader has an effective mitigation.

We evaluate Finelame [13] using the same settings that we use for Leader's evaluation, and using an already instrumented apache2 binary, shared by Demoulin et al. Table 6 shows the accuracy of Finelame over different attack scenarios. Finelame can correctly distinguish legitimate from attack connections only during SL and HC attacks. In other cases it labels all connections (legitimate and attack) as attack. The reason for this are unsophisticated Finelame models, which cannot distinguish legitimate from attack connections under heavy server load. We verified with Finelame's authors that their models suffer from this deficiency.

***Cogo* [15]**: Elsabagh et al. propose Cogo, which builds behavioral models of network I/O events. It employs Probabilistic Finite Automata (PFA) over these events to recognize future resource exhaustion states. Like Leader, Cogo's tracing of events spans the entire code stack from userland to kernel. For attack mitigation, Cogo kills the process whose network I/O events depart from the PFA model. We evaluated Cogo on the mini_httpd server, based on the instructions provided by the Cogo's authors.

The rest of our evaluation settings were the same as for Leader. Cogo was successful in detecting the start of the attack, and identifying process and thread ID for the mini_httpd process. Unfortunately, killing that process also denies service to all the legitimate users of the Web server. Unlike Cogo, Leader is able to identify attack sources and block them, allowing legitimate traffic to proceed unharmed. Due to Cogo software's current implementation limitations, we could not deploy Cogo in our evaluation scenarios to perform a detailed comparative analysis.

7 SENSITIVITY

During the learning phase, elliptic envelope uses multiple parameters to learn the baseline model. The key parameter is *contamination*, which denotes the proportion of outliers allowed in the input dataset. Another parameter is *assume_centered*, which determines whether the covariance estimates are to be directly computed with the FastMCD algorithm [39] or not. The parameter *support_fraction* determines the proportion of points to be included in the support of the raw MCD (minimum covariance determinant) estimate. If set to False, the minimum value of *support_fraction* will be used within the algorithm which is: (*number of features* + *number of samples* + 1) * 0.5. The parameters and values we used for learning the model are: *contamination* = 0.002, *assume_centered* = False and *support_fraction* = False. Table 7 in the Appendix, shows the precision and recall values obtained for Leader for the Slowloris attack experiment for various parameter values, and for liberal classification approach. By setting the contamination to be 0.002, Leader could achieve the best overall accuracy. Additionally, *assume_centered* and *support_fraction* set as True as well as False do not affect the accuracy. We set them as False and directly compute the covariance estimates using the FastMCD algorithm [38].

We measure the sensitivity of Leader's model by varying the number and type of legitimate connections used for training the model. Table 5 shows the training time and model's sensitivity for the MCU attack scenario on varying the number of connections. The results indicate that: (a) the size of the training data is directly proportional to Leader's classification accuracy. (b) capturing and including the data of legitimate users' connections when the application is under load, helps prevent the misclassification of legitimate users. The time required to train the model is lower than 10 seconds for all the scenarios we evaluated. We observe similar trends for other exDoS attack scenarios.

We further test resilience to model poisoning (adversarial data) during training, assuming that we use the contamination parameter value of 0.1 (10%), and vary the portion of the training data that is poisoned. Results are shown in the Appendix, Figure 5. There is almost no effect on the classification accuracy if the percentage of adversarial data injection does not surpass the contamination parameter value. However, once it exceeds the contamination parameter value, the model's accuracy for legitimate user classification declines sharply. Machine learning with adversarial data is an open research problem. We expect that we will be able to use solutions proposed by machine learning researchers to improve Leader's training and resilience to adversarial data injection, such as [6].

Table 4: Leader's classification accuracy, avg over 10 trials, for the liberal and the conservative scenario.

measure/scenario	liberal						conservative					
	SL	HC	Re-DoS	PHPEx	IRC	MCU	SL	HC	Re-DoS	PHPEx	IRC	MCU
true positive	99.9%	99.8%	99.6%	99.7%	100%	99.9%	100%	100%	100%	99.9%	100%	100%
true negative	99.1%	98.1%	98.7%	99.4%	96.6%	99.9%	99.4%	98.9%	99.4%	99.8%	99.2%	99.9%
false positive	0.9%	1.9%	1.3%	0.6%	3.4%	0.1%	0.6%	1.1%	0.6%	0.2%	0.8%	0.1%
false negative	0.1%	0.2%	0.4%	0.3%	0%	0.1%	0%	0%	0%	0.1%	0%	0%
att. req. before block	1.65	1.92	1.27	1.25	1.44	1.18	5.17	5.32	5.50	5.02	5.84	5.07

Table 5: Training time and model's sensitivity for an MCU attack scenario on varying the number of connections.

Learning			Classification	
leg. traffic (100 rps) conns	**leg. traffic under load (1,000 rps) conns**	**training time**	**true positives**	**true negatives**
100	0	0.05s	100%	82.7%
1,000	0	0.05s	100%	82.1%
10,000	0	1.5s	100%	84.3%
50,000	0	4.9s	99.9%	87.4%
100	100	0.09s	100%	86.3%
1,000	1,000	0.58s	99.8%	97.2%
10,000	10,000	2.1s	99.4%	98.2%
50,000	50,000	7.5s	99.6%	98.7%

8 OPERATIONAL COST AND SCALABILITY

Leader's operational cost is modest and scales sub-linearly with the number of incoming concurrent connections. Its core engine is written in C++. We use the tool SystemTap [48] for tracing and logging. Gebai et al. [18] show that the average latency for a SystemTap's tracepoint in Kernel Space is 130 nanoseconds, and these are the tracepoints we use in Leader.

We measured the time (in seconds) that the Web server takes to serve each of the requests to the homepage of the mirrored Imgur Web server both with and without SystemTap probes and Leader, under a heavy request load of 1,000 requests per second. The average time to process a request was 0.3239 seconds with SystemTap probes and Leader on, and 0.3223 seconds without SystemTap probes and without Leader. Thus, SystemTap and Leader jointly add less than 0.5% overhead to the server's latency.

Leader's elliptic envelope model requires a training time of 7–10 seconds for training on 100,000 legitimate users' connections. One could train the baseline model for millions of legitimate users' connections within a few minutes. On the average across 100,000 connections, Leader took around 0.5 ms for classifying a connection. Since Leader does not run on the request-processing path of an application, the classification delay is invisible to the user. Leader stores some state per connection and per source of incoming request.

The aggregate of total RAM usage by Leader, including SystemTap and in-memory elliptic envelope model was 2.73 GBs. Maintaining a short-lived state (until the connection finished or its source is blocked) about each active connection is thus affordable even for real-time deployment of Leader on popular sites that serve millions of users daily. Leader's state consists of several measures of resource consumption, i.e. a few integers per call, added cumulatively. Once a connection finishes, we clear its state. Our current model requires less than 0.5 MB per connection, thus a low-end server with 16GB of memory could accommodate tens of thousands of connection life stage sequences. This is in line with the amount of connections a typical Web server can serve simultaneously.

Extremely active Web sites, like Amazon can see about 4 M active clients per hour [45, 46], thus a few GBs would suffice to hold statistics for all active clients.

We evaluate scalability of Leader's mitigation using `iptables` and `ipset`. We artificially insert a diverse set of IP-rules and send packets matching these rules at a high rate. This emulates the situation when a server is under attack by numerous bots. We issue Web page requests and measure the time it takes to receive the reply. If the blacklisting places a heavy burden on the server, we would notice a slowdown in Web replies as the firewall's rule table grows. For space reasons we summarize our results. `iptables` only add 5% of overhead when there are more than 10 K rules. However, when the number of rules exceeds 1 M IPs the processing time explodes. On the other hand, `ipset` adds 5–8% to the processing time, as the rules table grows from 100 K to 1 M, and no measurable delay for fewer than 100 K rules. Thus, Leader can block at least one million IPs using `ipset`.

9 RELATED WORK

Most defenses handle exDoS attacks piecemeal, focusing on just a single variant, e.g., [3, 3, 4, 9, 17, 42, 44]. This piecemeal handling results in numerous application-specific and attack-specific defenses, which are then slow to transition to practice, because they only handle one attack variant. Another way to handle exDoS attacks is to treat them as application-level attacks and look for anomalies in the payload of the incoming service requests. This

Table 6: Comparison with related works over the same attack scenarios.

	Rampart				**Finelame**				**Leader (conservative)**			
scenario	TP	TN	FP	FN	TP	TN	FP	FN	TP	TN	FP	FN
SL	0%	100%	0%	100%	94.4%	99.9%	0.1%	5.6%	100%	99.4%	0.6%	0%
HC	100%	100%	0%	0%	100%	100%	0%	0%	100%	98.9%	1.1%	0%
Re-DoS	100%	100%	0%	0%	100%	0.02%	99.98%	0%	100%	99.4%	0.6%	0%
PHPEx	current code does not support vulnerable PHP versions				99.9%	0.04%	99.96%	0.1%	99.9%	99.8%	0.2%	0.1%
IRC	99.7%	100%	0%	0.3%	99.9%	0.02%	99.98%	0.1%	100%	99.2%	0.8%	0%
MCU	Rampart supports PHP applications only				100%	0%	100%	0%	100%	99.9%	0.1%	0%

is the approach taken by DDoS defense providers [23]. It requires costly deep-packet inspection and it is incomplete; it will detect some attacks that require malformed content, but will miss others that rely on timing and send well-formed payloads (e.g., Slowloris).

Several defenses take a more general approach to DDoS attack handling. For example, Hoque et al. [20] develop a statistical measure called Feature Feature Score (FFSc) for multivariate data analysis, to distinguish DDoS attack traffic from normal traffic. They use three traffic features: entropy of source IPs, variation of source IPs and packet rate. They build profiles of these features during normal operation and detect attack traffic as the traffic that deviates from these profiles. While this approach will detect flooding attacks, it may not be so effective against exDoS attacks, because they can be launched at a much lower rate. Such a low rate may not significantly change traffic features monitored by FFSc.

Xiang et al. [63] propose using the generalized entropy metric and the information distance metric to detect low-rate DDoS attacks, by measuring the difference in these measures between legitimate traffic and attack traffic. They then engage IP traceback to trace the attack close to the sources, and filter it there. Like Hoque et al. [20], they consider a limited set of features and thus may miss attacks when attack and legitimate traffic are very similar, or may mistakenly drop legitimate traffic when it changes due to normal fluctuations. Their response strategy is also limited, as attack often comes from far away and traceback is not supported on the Internet.

There has been research work done in the area of finding vulnerabilities [7, 34, 35], that can be exploited to launch sophisticated exDoS attacks. This work is complementary to Leader. Patching helps prevent exDoS attacks, while Leader helps detect and neutralize those attacks that exploit new or yet unpatched vulnerabilities. In additon to Cogo [15], Finelame [13] and Rampart [30], which we already discussed in Section 4, another general exDoS defense approache is DeDoS by Demoulin et al. [14]. DeDoS is a platform for mitigating asymmetric DoS attacks. DeDoS offers a framework to deploy code in a modular fashion. If part of the application stack is experiencing a DoS attack, DeDoS can massively replicate only the affected component, potentially across many machines. This allows scaling of the impacted resource separately from the rest of the application stack, so that resources can be precisely added where needed to combat the attack. Leader is complementary to DeDoS, as it looks to filter attack traffic rather than increase available resources.

10 CONCLUSIONS

Exploit-based DDoS (exDoS) attacks exploit some existing vulnerability at the target application, and can be effective at very low rates. Volumetric DDoS defenses fail to handle exDoS attacks, and current exDoS defenses only handle a few variants. In this paper, we introduced Leader, a novel approach for application-agnostic and attack-agnostic detection and mitigation of exDoS attacks. Leader operates by learning normal patterns of network and system-level resource usage when an application serves legitimate external requests. These baseline models are then used to detect external connections that use resources in anomalous manner. Leader blocks sources of such anomalous connections. We implement and evaluate Leader for protection of Web applications. Our results show that Leader has an aggregate accuracy of around 99% for both legitimate and attack connections, across six different exDoS variants used in our evaluation. Leader also has modest operating cost and adds only 0.5% of delay to Web request processing. Leader compares favorably to related work (Rampart and Cogo), and is more portable to different applications and attack variants.

Our future work includes evaluating Leader for protection of other popular applications, such as DNS servers, VPN proxy servers and mail servers. We would also like to explore if Leader can be used to protect against OS-based exDoS variants, such as TCP SYN floods or IP fragmentation attacks. It is likely that we would need to enrich Leader's monitoring with other system function calls, in addition to `net/socket.c` to make it effective against OS-level exDoS attacks.

ACKNOWLEDGMENTS

This material is based on research sponsored by the National Science Foundation via contract number 1815495. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation.

REFERENCES

[1] Akamai. 2020. State of the Internet Reports. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-a-year-in-review-report-2019.pdf, Accessed: July 6th, 2021.

[2] Akamai. 2021. State of the Internet Reports. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-a-year-in-review-report-2020.pdf, Accessed: July 6th, 2021.

[3] Zhihao Bai, Ke Wang, Hang Zhu, Yinzhi Cao, and Xin Jin. 2021. Runtime Recovery of Web Applications under Zero-Day ReDoS Attacks. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1575–1588.

[4] Efe Barlas, Xin Du, and James C Davis. 2022. Exploiting input sanitization for regex denial of service. In *Proceedings of the 44th International Conference on Software Engineering*. 883–895.

[5] Andreea Bendovschi. 2015. Cyber-attacks–trends, patterns and security countermeasures. *Procedia Economics and Finance* 28 (2015), 24–31.

[6] Yinzhi Cao and Junfeng Yang. 2015. Towards making systems forget with machine unlearning. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 463–480.

[7] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. 2009. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 186–199.

[8] Hyoung-Kee Choi and John O Limb. 1999. A behavioral model of web traffic. In *Proceedings. Seventh International Conference on Network Protocols*. IEEE, 327–334.

[9] Jongseok Choi, Jong-gyu Park, Shinwook Heo, Namje Park, and Howon Kim. 2016. Slowloris DoS Countermeasure over WebSocket. In *International Workshop on Information Security Applications*. Springer, 42–53.

[10] CloudFlare. 2022. DDoS Attack Trends for 2022 Q1. https://blog.cloudflare.com/ddos-attack-trends-for-2022-q1/, Accessed: July 6th, 2022.

[11] Roy De Maesschalck, Delphine Jouan-Rimbaud, and Désiré L Massart. 2000. The mahalanobis distance. *Chemometrics and intelligent laboratory systems* 50, 1 (2000), 1–18.

[12] Anderson Bergamini de Neira, Burak Kantarci, and Michele Nogueira. 2023. Distributed denial of service attack prediction: Challenges, open issues and opportunities. *Computer Networks* 222 (2023), 109553.

[13] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. 2019. Detecting asymmetric application-layer denial-of-service attacks in-flight with finelame. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. 693–708.

[14] Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Bob DiMaiolo, Jingyu Qian, Chirag Shah, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, et al. 2018. Dedos: Defusing dos with dispersion oriented software. In *Proceedings of the 34th Annual Computer Security Applications Conference*. 712–722.

[15] Mohamed Elsabagh, Dan Fleck, Angelos Stavrou, Michael Kaplan, and Thomas Bowen. 2017. Practical and accurate runtime application protection against dos attacks. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 450–471.

[16] Exploit Database. 2012. Hashtables Denial of Service. https://www.exploit-db.com/exploits/18296, Accessed: July 6th, 2021.

[17] NR Fitri, AHS Budi, I Kustiawan, and SE Suwono. 2020. Low interaction honeypot as the defense mechanism against Slowloris attack on the web server. In *IOP Conference Series: Materials Science and Engineering*, Vol. 850. IOP Publishing, 012037.

[18] Mohamad Gebai and Michel R Dagenais. 2018. Survey and analysis of kernel and userspace tracers on Linux: Design, implementation, and overhead. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–33.

[19] Hacking with PHP. 2015. Denial of service. http://www.hackingwithphp.com/17/1/9/denial-of-service, Accessed: July 6th, 2021.

[20] Nazrul Hoque, Dhruba K Bhattacharyya, and Jugal K Kalita. 2016. A novel measure for low-rate and high-rate DDoS attack detection using multivariate data analysis. In *2016 8th International Conference on Communication Systems and Networks (COMSNETS)*. IEEE, 1–2.

[21] Imperva. 2020. https://tinyurl.com/y5jmjuzv, Accessed: July 6th, 2021.

[22] INDUSFACE. 2019. https://tinyurl.com/y4c3ywry, Accessed: July 6th, 2021.

[23] Mattijs Jonker, Anna Sperotto, Roland van Rijswijk-Deij, Ramin Sadre, and Aiko Pras. 2016. Measuring the adoption of DDoS protection services. In *Proceedings of the 2016 Internet Measurement Conference*. 279–285.

[24] Kaspersky. 2019. https://tinyurl.com/y258rnpm, Accessed: July 6th, 2021.

[25] Kaspersky lab. 2018. Denial of service: How Businesses Evaluate the threat of DDoS attacks. https://tinyurl.com/ybnmogg3, Accessed: July 6th, 2021.

[26] The Security Ledger. 2018. https://tinyurl.com/yysvu859, Accessed: July 6th, 2021.

[27] Kun-Lun Li, Hou-Kuan Huang, Sheng-Feng Tian, and Wei Xu. 2003. Improving one-class SVM for anomaly detection. In *Proceedings of the 2003 international conference on machine learning and cybernetics (IEEE Cat. No. 03EX693)*, Vol. 5. IEEE, 3077–3081.

[28] Lukas Martinelli. [n. d.]. Simulate Hash Collision Attack on a PHP Server. https://github.com/lukasmartinelli/php-dos-attack.

[29] Wei Meng. 2018. Rampart's code. https://github.com/cuhk-seclab/rampart, Accessed: July 6th, 2021. (2018).

[30] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. 2018. Rampart: protecting web applications from CPU-exhaustion denial-of-service attacks. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 393–410.

[31] David Mosberger and Tai Jin. 1998. Httperf&Mdash;a Tool for Measuring Web Server Performance. *SIGMETRICS Perform. Eval. Rev.* 26, 3 (Dec. 1998), 31–37. https://doi.org/10.1145/306225.306235

[32] Nicolas Niclausse. 2017. Tsung 1.7.0 released. http://tsung.erlang-projects.org/, Accessed: July 6th, 2021.

[33] Georgios Oikonomou and Jelena Mirkovic. 2009. Modeling human behavior for defense against flash-crowd attacks. In *2009 IEEE International Conference on Communications*. IEEE, 1–6.

[34] Oswaldo Olivo, Isil Dillig, and Calvin Lin. 2015. Detecting and exploiting second order denial-of-service vulnerabilities in web applications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. 616–628.

[35] Theofilos Petsios, Jason Zhao, Angelos D Keromytis, and Suman Jana. 2017. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2155–2168.

[36] Red Hat. 2019. Introduction to eBPF in Red Hat Enterprise Linux 7. https://www.redhat.com/en/blog/introduction-ebpf-red-hat-enterprise-linux-7, Accessed: July 6th, 2021.

[37] Marc Roig, Marisa Catalan, and Bernat Gastón. 2019. Ensembled Outlier Detection using Multi-Variable Correlation in WSN through Unsupervised Learning Techniques.. In *IoTBDS*. 38–48.

[38] Peter J Rousseeuw and Katrien Van Driessen. 1999. A fast algorithm for the minimum covariance determinant estimator. *Technometrics* 41, 3 (1999), 212–223.

[39] Peter J Rousseeuw and Katrien Van Driessen. 1999. A fast algorithm for the minimum covariance determinant estimator. *Technometrics* 41, 3 (1999), 212–223.

[40] Scikit learn. 2018. EllipticEnvelope. https://scikit-learn.org/stable/modules/generated/sklearn.covariance.EllipticEnvelope.html, Accessed: July 6th, 2021. (2018).

[41] Selenium. 2012. Selenium Webdriver. https://tinyurl.com/y6a4czhe, Accessed: July 6th, 2021.

[42] Mark Shtern, Roni Sandel, Marin Litoiu, Chris Bachalo, and Vasileios Theodorou. 2014. Towards mitigation of low and slow application ddos attacks. In *2014 IEEE International Conference on Cloud Engineering*. IEEE, 604–609.

[43] JacobMisirian SplittyDev. [n. d.]. Python implementation of a slowloris DoS tool. https://github.com/ProjectMayhem/PySlowLoris.

[44] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *27th USENIX Security Symposium (USENIX Security 18)*. 361–376.

[45] Statista. 2018. Most popular retail websites in the United States as of December 2017, ranked by visitors (in millions). https://www.statista.com/statistics/271450/monthly-unique-visitors-to-us-retail-websites/, Accessed: July 6th, 2021.

[46] Statista. 2019. Combined desktop and mobile visits to Amazon.com from February 2018 to April 2019 (in millions). https://www.statista.com/statistics/623566/web-visits-to-amazoncom/, Accessed: July 6th, 2021.

[47] Stack Status. 2016. Outage postmortem. https://stackstatus.tumblr.com/post/147710624694/outage-postmortem-july-20-2016, Accessed: July 6th, 2021.

[48] SystemTap. [n. d.]. SystemTap. https://sourceware.org/systemtap/.

[49] Liran Tal. 2019. The state of open source security report. https://res.cloudinary.com/snyk/image/upload/v1551172581/The-State-Of-Open-Source-Security-Report-2019-Snyk.pdf.

[50] Rajat Tandon. 2020. A survey of distributed denial of service attacks and defenses. *arXiv preprint arXiv:2008.01345* (2020).

[51] Rajat Tandon, Pithayuth Charnsethikul, Michalis Kallitsis, and Jelena Mirkovic. 2022. AMON-SENSS: Scalable and Accurate Detection of Volumetric DDoS Attacks at ISPs. In *GLOBECOM 2022-2022 IEEE Global Communications Conference*. IEEE, 3399–3404.

[52] Rajat Tandon, Jelena Mirkovic, and Pithayuth Charnsethikul. 2020. Quantifying cloud misbehavior. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 1–8.

[53] Rajat Tandon, Abhinav Palia, Jaydeep Ramani, Brandon Paulsen, Genevieve Bartlett, and Jelena Mirkovic. 2021. Defending web servers against flash crowd attacks. In *International Conference on Applied Cryptography and Network Security*. Springer, 338–361.

[54] The Open Web Application Security Project (OWASP). 2018. Regular expression Denial of Service - ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS, Accessed: July 6th, 2021.

[55] Marino Urso. 2020. *High performance eBPF probe for Alternate Marking performance monitoring*. Ph. D. Dissertation. Politecnico di Torino.

Table 7: Precision and recall values when varying the contamination parameter of elliptic envelope.

Contamination	Precision	Recall
0.000	0.999	0.936
0.002	**0.990**	**0.992**
0.005	0.982	0.999
0.01	0.969	0.999
0.03	0.941	0.999
0.05	0.911	1

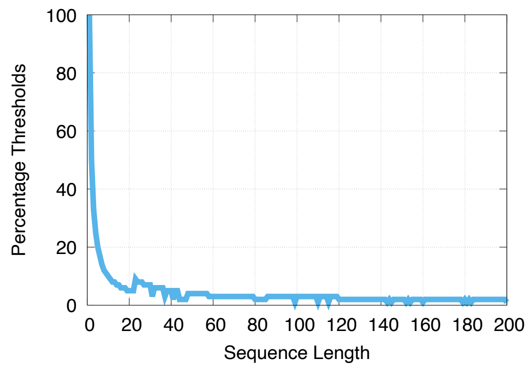

Figure 4: The classification threshold values for the connections that are outliers for a given source IP for different sequence lengths that lead to around 99% true positives and around 1% false positives.

[56] Vickie Li. 2018. Preg_replace() PHP Function Exploitation. https://www.yeahhub.com/code-execution-preg_replace-php-function-exploitation/, Accessed: July 6th, 2021.

[57] Nicolaas Weideman, Haoda Wang, Tyler Kann, Spencer Zahabizadeh, Wei-Cheng Wu, Rajat Tandon, Jelena Mirkovic, and Christophe Hauser. 2022. Harm-DoS: Hash Algorithm Replacement for Mitigating Denial-of-Service Vulnerabilities in Binary Executables. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses.* 276–291.

[58] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 255–270.

[59] Wikipedia. 2018. Curse of dimensionality. https://en.wikipedia.org/wiki/Curse_of_dimensionality, Accessed: July 6th, 2021.

[60] Wikipedia. 2018. Flask. https://en.wikipedia.org/wiki/Flask_(web_framework), Accessed: July 6th, 2021.

[61] Wikipedia. 2018. Log rotation. https://en.wikipedia.org/wiki/Log_rotation/, Accessed: July 6th, 2021.

[62] Wikipedia. 2018. Slowloris. https://en.wikipedia.org/wiki/slowloris_(computer_security), Accessed: July 6th, 2021. (2018).

[63] Yang Xiang, Ke Li, and Wanlei Zhou. 2011. Low-rate DDoS attacks detection and traceback by using new information metrics. *IEEE transactions on information forensics and security* 6, 2 (2011), 426–437.

11 APPENDIX

11.1 Maliciously Crafted URL Attack on a Flask Application (MCU)

We crafted an implementation of a vulnerable Web application in Flask to demonstrate that Leader supports a variety of Web server applications and frameworks. Our application is shown below:

```
from flask import Flask, request, render_template_string, render-
_template
```

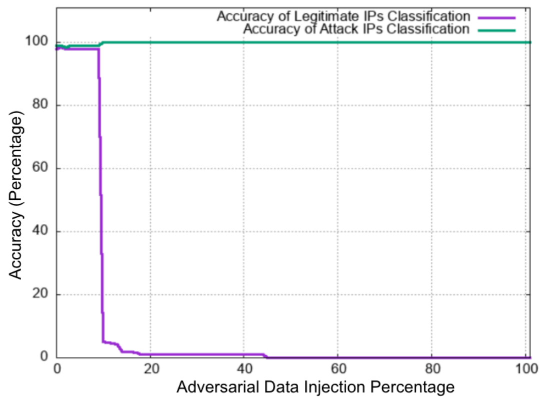

Figure 5: Effect of adversarial data injection for an MCU attack scenario using the contamination parameter as 0.1.

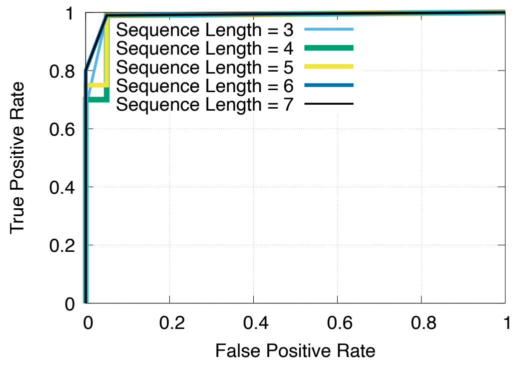

Figure 6: ROC curves using the percentage thresholds for different sequence lengths

```
app = Flask(__name__)
@app.route('/')
def sample():
    person = {'name':"XYZ",'details':"==dfdgg␣......␣cG5lcnJlZg=="}
    if request.args.get('name'):
        person['name'] = request.args.get('name')
    template = '''<h2>The details are: %s!</h2>''' % person['name']
    return render_template_string(template,person=person)
```

A legitimate user's request may look as follows:

```
Webpage.com?name=XYZ{{person.details}}
```

An attack request may look as follows:

```
Webpage.com?name=XYZ{{person.details}}{{person.details}}-
......{{person.details}}{{person.details}}
```

; where {{*person.details*}} can be appended to the URL hundreds of times, until the limit for the allowed maximum length for a URL is reached. If {{*person.details*}} resolves to a large text or string, the call to render_template_string() would lead to hundreds of occurrences of {{*person.details*}} being resolved. The return value of the function will be hundreds of times larger than the return values of legitimate requests. If multiple concurrent malicious requests are made to the server this can overload its outgoing bandwidth and deny service to legitimate clients.

Table 8: Classification accuracy for the liberal design scenario for 1-class SVM and elliptic envelope.

liberal design	**1-class SVM**				**elliptic envelope**			
scenario	**TP**	**TN**	**FP**	**FN**	**TP**	**TN**	**FP**	**FN**
SL	93.2%	92.4%	7.6%	6.8%	99.9%	99.1%	0.9%	0.1%
HC	93.6%	91.7%	8.3%	6.4%	99.8%	98.1%	1.9%	0.2%
Re-DoS	93.9%	92.1%	7.9%	6.1%	99.6%	98.7%	1.3%	0.4%
PHPEx	93.1%	92.4%	7.6%	6.9%	99.7%	99.4%	0.6%	0.3%
IRC	93.7%	92.1%	7.9%	6.3%	100%	96.6%	3.4%	0%
MCU	94.1%	92.5%	7.5%	5.9%	99.9%	99.9%	0.1%	0.1%