# Automated Detection and Mitigation of Application-level Asymmetric DoS Attacks

Henri Maxime Demoulin, Isaac Pedisich, Linh Thi Xuan Phan, Boon Thau Loo
University of Pennsylvania

## ABSTRACT

This paper presents a novel integrated platform for the automatic detection and mitigation of denial-of-service (DoS) attacks in networked systems. Recently, these attacks have evolved from simple flooding at the network layer to targeted, application-specific asymmetric attacks. Because of this trend, existing techniques—which rely primarily on network classification at the edge or core routing devices—are becoming ineffective. Our platform integrates machine learning with fine-grained application-level performance metrics and monitoring statistics at the software's components to achieve precise traffic classification for detecting application-specific attacks in real time. When an attack is detected, the platform will then automatically isolate suspicious traffic by routing it to separate component instances with a fixed resource reservation, thus preventing it from interfering with the rest of the system. Our evaluation using a range of asymmetric attacks shows that our detection technique is highly effective and that the close-loop integration of real-time detection and traffic isolation can deliver substantially better quality-of-service for good users in the presence of attacks than the default mitigation using dynamic scaling of resource alone.

## CCS CONCEPTS

• **Security and privacy → Denial-of-service attacks**;

## 1 INTRODUCTION

Denial-of-service (DoS) attacks are a persistent threat in the Internet ecosystem. Recently, these attacks have evolved from the traditional volumetric attacks to complex application-specific asymmetric attacks [15, 19, 20]. Defending against the latter type of attacks is particularly challenging for at least two reasons: First, they require only a small volume of traffic to bring down a specific resource at the victim's endpoint, thus increasing their stealth potential in the midst of large volumes of legitimate traffic. Second, since fraudulent application layer packets do not necessarily translate to (detectable) problematic patterns at the network level, it is difficult to differentiate them from legitimate traffic by edge or core routing devices, which are computationally constrained and cannot perform complex intrusion detection [10]. As a result, existing DoS detection techniques—which rely primarily on traffic classification at the edge or core routing devices [10, 12, 22, 23]—are becoming ineffective against these attacks.

In this paper, we solve the above challenge based on two key insights: first, as these attacks target some resource(s) at the application level, run-time monitoring statistics about the resource use of the application and the system can provide good metrics for distinguishing potential adversarial behaviors from normal behaviors. Second, since asymmetric attacks typically target specific components in the application stack, their detection is much easier if monitoring information is available at component-level granularity. For instance, a TLS Renegotiation attack, which exploits an asymmetry in the SSL/TLS protocol, will exercise CPU time consumption by the component responsible for the TLS handshake functionality [8], which will lead to a substantial increase in the CPU resource use at the TLS component but not at others.

To enable detection using component-level run-time information, we build a platform that can be used to develop and deploy software as fine-grained components, then monitor them at runtime. Such *decomposed* design pattern is increasingly adopted for applications ranging from data analytic to edge computing [1, 2, 4, 5, 7], and are foreseen to be useful abstractions for cloud deployments [14]. Due to the modular nature of the platform, automated statistics collection can be performed at the component's boundary, thus relieving programmers from the burden of heavily instrumenting their code to identify attacks. In addition, fine-grained resource allocation and isolation policies can be applied at *individual components* instead of at the full software stack, thus enabling better resource utilization—and hence better quality of service for legitimate users—during attacks.

However, the explosion in the number of components has dire consequences for attack detection: as a set of components is often maintained by a different engineering team than the one responsible for the operation of the entire application, it is common that operators do not possess a sufficient domain expertise of each individual component to evaluate how various resources should be consumed by legitimate traffic. Accurately distinguishing attack and legitimate traffic means being able to make sense of a linearly increasing volume of monitored data, and understanding which metric discriminates the traffic best. Therefore, decomposed platforms have a critical need for automated solutions allowing feature selection and anomaly detection at the component level.

To make a step toward the automated management of decomposed platforms, we build a supervised policy composition engine on top of our prototype, and evaluate how well it can perform feature selection for a representative set of application-level asymmetric DoS attacks and legitimate traffic surge. In addition, we evaluated the platform capability to enforce traffic isolation during the event of a yet unseen attack, by using an isolation policy composed with the engine. We evaluate the quality of service provided to legitimate traffic in presence of this policy compared to a default cloning policy. Overall, we present a concrete example of a closed feedback-loop decomposed platform, which automates the detection and mitigation of application level asymmetric DoS attacks and integrates human input to sharpen its accuracy.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*SelfDN 2018, August 24, 2018, Budapest, Hungary*
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5914-6/18/08…$15.00
https://doi.org/10.1145/3229584.3229589

Section 2 presents the overall architecture of our platform. Section 3 details the policy composition engine in our platform. Section 4 studies the classification abilities of the engine across a set of attacks, and demonstrates how we can use the system to perform automatic traffic isolation of a yet unseen attack. We conclude by discussing related work, our ongoing research, as well as the challenges we faced in implementing our solution.

## 2 DESIGN

### 2.1 Execution model

Figure 1 shows the overall architecture of our platform, which supports the deployment and run-time resource allocation of applications as a set of fine-grained components (shown as circles of the dataflow graph in each node). Application developers wrap their components' code into a main handler function, which consumes events from a data queue. The platform provides a default queue admission control mechanism, which can be configured for each different component (e.g., a hard limit on the number of items allowed inside the queue), as well as the ability to define customized queuing policy for components. For message passing, components can either use a default serialization protocol, or implement their own to define the structure of a component's payload.

At startup, a description of the application's dataflow is given to a centralized controller, which deploys instances of components as threads of a local runtime processes (one per node). Runtimes continually listen for requests from the controller (such as component instance addition or deletion). They are responsible for setting up and controlling the routing of messages between components (as described by the application dataflow). Local components communicate through IPC, while remote ones use a long lived TCP connection setup between each runtime process. In normal operations, packets are directed toward the next component instance in the pipeline that has the smallest buffer occupancy by default. In addition, the controller defines the resource allocation policy to be enforced at runtime. Those can be derived from rules produced by the policy composition engine (for instance, how many instances of a component should be deployed on the cluster), or manually defined heuristics (for instance a greedy resource allocation mechanism based on component's resource consumption).

Local runtimes are also in charge of collecting, aggregating, and reporting monitoring data to the controller. Table 1 describes the set of statistics collected by the system. We retrieve some of them using *getrusage()* (CPU time, page faults, etc), and other directly from the routing module of our framework (e.g. arrival rate at a component and buffer length). In addition to those, the platform's API allows programmers to register application-level statistics for collection by the runtime, and subsequently make those available for policy composition. Statistics are collected at a configurable time interval. Rather than sampling an average of this interval, the system builds a dynamic histogram of the values. More precisely, this means that for each reporting interval, we report the values of a configurable amount of percentiles of the metric distribution. For example in our experiments, we collect the minimum value, 25th percentile, median, 75th percentile, and maximum values. The controller formats and inserts collected statistics into a database for future use by the policy composition engine, which we describe in the next section.

| Name | Description |
|---|---|
| *Request* | |
| req_lifetime | Total time spent in the system |
| req_enqueues | Number of enqueues across all components |
| req_cpu | Amount of CPU time consumed |
| req_fds | Number of file descriptors opened |
| *Component* | |
| com_enqueues | Total number of enqueues |
| com_mem | Virtual memory consumed |
| com_queue | Buffer length |
| com_drops | Packets dropped from the buffer |
| com_states | Number of concurrent states maintained |
| com_throughput | Request throughput |
| com_ing_rate | Ingress rate |
| com_wall_time | Wall time for request processing |
| com_idle_time | Wall time between two executions |
| com_exec_time | Time spent in execution |
| com_total_time | Time spent in execution across all requests |
| com_usr_time | User-space fraction of CPU time |
| com_sys_time | Kernel-space fraction of CPU time |
| com_fds | Number of FDs used |
| com_max_rss | Maximum resident set size |
| com_min_faults | page faults not involving disk I/O |
| com_maj_faults | page faults involving disk I/O |
| com_vol_ctxsw | Voluntary thread yields |
| com_invol_ctxsw | Involuntary thread preemption |

**Table 1: Statistics collected at runtime**

### 2.2 Automated Mitigation

Our platform supports a variety of mitigation techniques, configured through policy tables that map entities to rules and actions. Entities for which actions can be taken are sets of application components, or requests currently sojorning in the system. Policies are systematically enforced at runtime at different points in the pipeline, either by the centralized controller, or the local runtimes. Where to enforce a policy is primarily based on which information is required to do so. For instance, determining the placement of a new component instance often requires knowledge of resource consumption across the cluster, whereas determining if a request should be admitted at a component's queue can often be done locally.

As an example, consider a default entry in the table, which specifies that the controller should attempt to clone any components whose queue length is greater than 0. Such a rule is made with the assumption that the system is provisioned to sustain an expected arrival rate — in which case components' queues should not grow. Another example is the isolation policy which we describe in our evaluation (section 4.2), which stipulates that any IP which requests have been consuming more than a certain threshold of CPU time, learned in a supervised fashion, should be flagged for temporary quarantine. This policy is systematically evaluated by local runtimes when requests are enqueued to a component.

In our current prototype, the action type and the granularity at which policies operate (for example, all components of a given type, any component, etc) are defined manually by users alongside the workload class label and the action type. Table 2 gives an example of policy table. Tables are pushed on local runtimes via the controller's

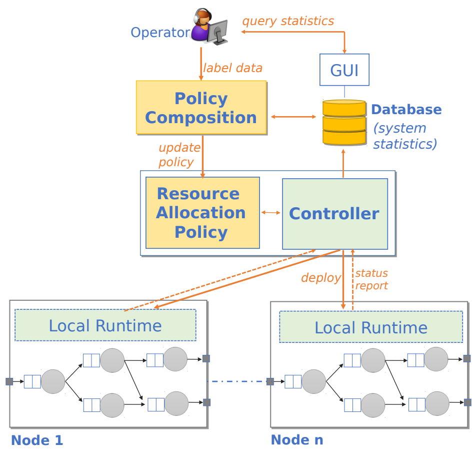

**Figure 1: Architecture of our closed-loop feedback platform**

API, either in an automated fashion by the policy composition engine, or manually by a user.

## 2.3 Supervised rule learning

The policy composition engine is in charge of generating entries in the policy table in a supervised fashion. It can integrate the operator's knowledge of the system to label training sets, and plug-in with an analytic library to train models of varying complexity. For example, if the operator is only able to recognize that an attack occurred, she would apply a binary label ("attack", "good") to the datapoints, regardless of the attack type. On the other hand, if she is able to distinguish between memory and CPU bound attacks, as well as legitimate flash crowds event, she would apply a multiclass labeling to the dataset. She can then request the engine for the most discriminant features and threshold to generate an anomaly detection rule.

Figure 1 describes the learning workflow in our prototype. (i) A human operator observes the behavior of her application over time through a standard monitoring dashboard, and has access to the entire set of features monitored by the platform. (ii) Having observed that one or many attacks occurred, she labels the period of attack to the best of her abilities, and queries the policy composition engine with the labels, a set of entities to train a model on, the entries to the policy table she wishes to fill, and a degree of precision (the $k$ features she wants to extract) (iii) The engine performs feature selection, in a fashion we describe next, and fills in all the entries in the table which workload class were in the set provided by the user, using the $k$ predicates it learned from the training set. (iv) The engine returns a confusion matrix to the user, which can either validate or refuse the automated deployment of the new entries on the platform.

## 3 POLICY COMPOSITION ENGINE

The engine has access to all the events which occurred in the application since its startup. It does expose an API which allows an operator to label slices of the data, and query the best set of discriminant for the labels she provided. We implemented the engine using python scikit [18]. Our prototype first remove features with zero variance from the training set, then performs a two level cross-validation method exploiting the degree of knowledge the user have on her application. The $k$ features requested and their threshold are then used to complete the policy which is then pushed on the controller and/or local runtimes. We explain our choice for Decision Tree and our validation method next.

### 3.1 Decision Tree

We select Decision Tree in our prototype because of several reasons. (i) Its ability, through information gain, to reveal how informative each feature is for attack characterization. (ii) The cost of execution of a Decision Tree is logarithmic in the number of samples used for training (which we expect to be quite large for any long-lived application. In our current setup, an hour worth of data accounts for about 600 MB). (iii) It has the ability to perform multi-class classification. Indeed, we expect that over time, the human operator will discover more and more workload classes she will want to incorporate in her labels, and deploy increasingly refined mitigation policies. (iv) Its ease of interpretation: analyzing the selected features can be easily done by a human, contrarily to other non linear feature selection techniques such as auto-encoders. Interpretability is particularly important to understand the nature of the vulnerability in the target application component. (v) Its extensibility to Random Forests and Boosted Forests, which can help increase the robustness of the engine in the future.

### 3.2 Feature selection

First, the engine automatically removes features with zero variance across all the training sets. Because such features do not vary during the execution, they will not be informative for supervised classification.

The learning engine then performs a grid search over various depths of decision trees as well as increasing complexity of cross-validations depending on the user's query. If the user simply asks for a binary classification, the engine immediately performs a 10-fold cross- validation, making sure that for each fold each class sample is properly weighted to avoid class imbalance. If the request is for multi-class, the engine trains a classifier for each class and select the one with the best average training error after 10-fold cross validation, following the one-versus-all method. We downsample with elimination the "rest" classes to avoid class imbalance with the "one" class.

The engine then returns a confusion matrix summarizing the performance of the best tree, which depth is always set the tree's depth to be $k$, the number of features requested by the user. In addition, the engine returns a set of rules predicate corresponding to the requested class labels. For each entry to be filled in the policy table, there are $k$ predicates which represent a path from the root of the best tree to its leaves. The learning engine then uses the platform's controller API to deploy the learned policy table's entries to the runtimes.

## 4 EVALUATION

We performed a series of experiments to evaluate the effectiveness of our platform. Our evaluation aims to investigate (i) how insightful is a traffic isolation policy composed with the policy engine, and (ii)

| Entity type | Entity set | Workload class | Rule | Action | Enforcement Point |
|---|---|---|---|---|---|
| REQUEST | * | Slow attacks | Flow Lifetime $> \epsilon$ | ISOLATE (remote) | PACKET ENTERS A QUEUE |
| REQUEST | * | CPU asym | Packet CPU Time $> \epsilon$ | ISOLATE (remote or local) | PACKET LEAVES THE SYSTEM |
| COMPONENT | * HTTP | Flash crowd | Arrival Rate $> \epsilon$ | CLONE | CONTROLLER PARSE STATS |
| COMPONENT | * | Normal | Queue Length $> 0$ | CLONE | CONTROLLER PARSE STATS |

**Table 2: An example of Policy Table. Rules' predicate can be drawn from domain expertise are learned in a supervised fashion. Entries are systematically enforced at runtime. Actions are automatically applied when a predicate holds.**

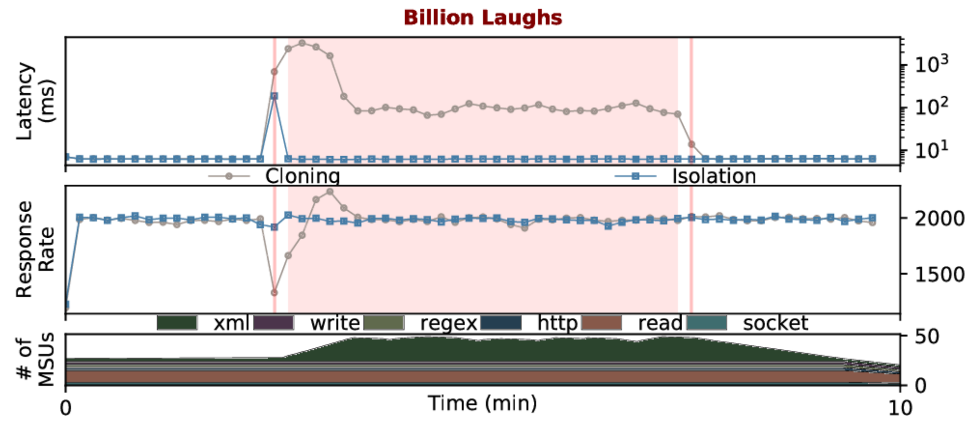

**Figure 2: Legitimate traffic latency with and without traffic isolation. Attack period is delimited with a red background. Without traffic isolation, the platform deploys up to 24 more XML component instances to withstand the load.**

how well the system can maintain quality of service for legitimate clients under attacks, including attacks *unseen* in the policy training set, using the learned traffic isolation policy. For comparison, we also evaluated the performance of the system under a default policy where the system clones components as and when their resource demands increase, without performing any traffic isolation.

**Scenario:** A user wants to fill in a rule in the policy table to perform isolation of suspicious traffic. Her application (a simple API service she uses as the front-end for her business backend) has previously been under some asymmetric CPU bound attacks, the exact type of which she has not been able to precisely determine. She identifies two attack periods by analyzing her logs, and queries the policy composition engine for the single best traffic feature over which to perform isolation. We analyzed the query's results produced by the learning engine, as well as the effects on the application with and without the learned rule enabled in the controller.

**Application:** In all the experiments, we deployed a simple decomposed web server stack. The webserver is made of six components: an *I/O* component, which is responsible for handling events on the webserver's socket; a *read* component, which is responsible for reading raw bytes from the socket, performing TLS handshake, deciphering the message, and handing the plain text request to the *HTTP* component. The HTTP component parses requests headers, and forwards the request to either a *RegEX* component, responsible for parsing a regular expression with the PCRE engine, or an *XML* component, responsible for parsing an XML retrieved from the HTML payload. Finally, a *write* component is responsible for encrypting the HTTP response and sending it to the client.

**Training traffic description:** The actual attacks the user identified are an occurrence of TLS renegotiation attacks [8] and one of Redos [6] (regular expression attack). During the TLS renegotiation attack, the attacker repetitively triggers TLS handshakes on a single connection. In our setup, a single handshake required approximately 2.1 milliseconds of computation time (we used a 2048-bits RSA key), and every malicious request triggers 50 renegotiations before closing. We sent the attack traffic using a house-made C HTTPS client for 5 minutes at a rate of 25 requests per second. The Redos attack exploits a flaw in the PCRE engine by sending a specifically crafted regular expression to the parser. In our setup, each request required approximately 100 milliseconds of computation time. We sent the Redos traffic at a rate of 25 requests per second for 5 minutes, using an open-source distributed benchmarker named Tsung [9]. During both attacks, as well as 5 minutes before and after each attack, the application receives legitimate traffic, which was generated by Tsung under an exponential distribution with a mean of 2000 requests per second, split over two client nodes. Overall, the dataset spans a period of about 20 minutes.

**Testbed:** Our testbed is a cluster of 10 computers connected via a 10 Gbps switch in a star topology. Eight of the computers were used for processing the traffic; each of them has 8 1.80 GHz cores (with hyperthreading and DVFS disabled), 64 GB of memory, and runs Linux kernel 4.4.0-62. The remaining two, which we used as traffic generators, have 24 2.4 GHz cores and 64 GB of memory each, and both run Linux kernel 4.13.5-200.

**Training set:** We configured the platform to report monitoring data for all 6 components of the application every second. In addition, it also reported traffic information and runtime node-wide statistics. As explained in Section 2, the data for each reporting

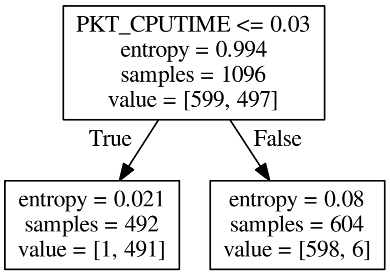

**Figure 3: Training output of the decision tree**

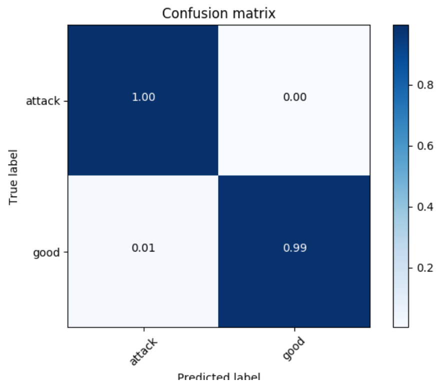

**Figure 4: Training confusion matrix for the decision tree**

interval were distributed into a histogram which we configured to have 5 bins (mininum, 25th percentile, median, 75th percentile, and maximum values). The specific metrics we collected are listed in Table 1. In total, each sample account for about 590 raw features (6 components time 19 component features time 5 bins, and 4 request features time 5 bins), however, we usually average each reported histogram across component types' to reduce the volume of data. (Note that our engine allows queries to be made over an arbitrary set of components, and thus it is possible to consider data at the instance granularity; however, we found that, for our setup, this granularity was not too practical and does not provide much additional usefulness.) For this specific experiment, as we intent to create a policy targeting suspicious requests, we only use the requests features.

## 4.1 Feature selection

We queried the learning engine with labels for "attack" and "good", and we requested the three best discriminant among all 75 features, such that we can fill an entry in the policy table enforcing packet isolation (using a mechanism described in Section 4.2) for those flows which break the learned rule. Upon reception of the query, the engine performed the training described in Section 3.2. The query's results are shown in Figures 3 and 4. For the sake of space, we only display the first layer of the decision tree: indeed, we found that the flow's request CPU time, with values greater than 30 milliseconds the demarcation for good and attack traffic, was the single best discriminant feature, classifying with 99.36% precision and 99.83% recall the points where the system was under attack. Of course those values are only an encouragement, in the sense that they are obtained from a training set and not a tangible clue of the generalization capability of the classifier. In addition, we do expect that more complex application will have more than one discriminant. We evaluate how well the model generalize to a testing set in the next section.

## 4.2 Online policy enforcement

We now evaluate the quality of service provided to the legitimate traffic with and without the enforcement of the learned policy. Specifically, we attacked the application with a Billion Laughs attack [3], generated by Tsung; this attack was *not seen yet* by the learning engine, and thus not present in the policy training dataset.

**Traffic isolation:** The isolation policy is evaluated by local runtimes whenever a packet is enqueued to a component, or leaves the system. When the rule is verified, the runtime flags the origin IP of the packet as suspicious for a configurable amount of time (10 seconds in our experiment), and spawns a quarantine worker thread on which it deploys the full library of functions in the application[1], and updates its routing module such that only the suspicious traffic is redirected to that component.

**Default cloning policy:** This default heuristic, introduced in section 2.2, monitors the minimum buffer length of each component from the centralized controller, and if this value exceeds 0, greedily picks an unoccupied core on which to deploy a new instance of the overloaded component.

**Experimental results:** For both experiments, we started the platform with 3 machines, each of which is instantiated with one I/O, 4 read, 1 HTTP, 3 RegEX, 3 XML, and 1 write components. We ran good traffic only for 2.5 minutes, then we started the attack which lasted for 5 minutes. We left good traffic running for 2.5 additional minutes after the attack. Figure 2 shows the latency and success rate of the good clients in with the isolation policy and with the cloning policy. The bottom chart displays the number of components deployed using the default cloning without the traffic isolation policy (up to 24 new XML components on the three free servers in our cluster, one per available CPU). When the policy was enabled, no cloning happens and the platform isolated traffic based on the learned rule.

The results show that under default cloning, the additional resource deployed to handle the increase in traffic load can help to reduce the latency (from more than a second latency to hundreds of milliseconds). However, the latency only began to reduce after one minute into the attack, and it still remained at hundreds of milliseconds during the rest of the attack duration.

In contrast, when isolation was enabled, because offending traffic was confined to quarantine components, good clients suffered only a brief increase in latency (up to about 200 milliseconds), and this happened only at the beginning of the attack period. The latency returned to the same latency as when there is no attack (about two

[1]We are working on optimizing this such that only the target component is replicated.

milliseconds) as soon as within in a couple of seconds. We observe no false positive event where one of the two legitimate client's IP is isolated. The above results demonstrate that the traffic isolation enabled by the learned rule is highly effective in isolating attacks, even for unseen attacks, and that it can maintain desirable quality of service for legitimate users.

## 5 RELATED WORK

Previous work on application-layer DoS attacks relies mostly on the processing and analyzing of traffic generated by volumetric attacks, by comparing the entropy of offending and legitimate traffic [17, 24], using flow sampling [13] or sketch-based solutions [22]. While these techniques work well for volumetric attacks, they have been known to decrease in their effectiveness when the amount of attack traffic is low [13], such as asymmetric attacks.

A popular defense against asymmetric attacks is based on interventions from the end-users, either by solving a CAPTCHA, or by having its clients computing a computational puzzle. This technique has known limitations [10]. Our platform takes advantage of the fact that asymmetric application-level DoS attacks often target a vulnerability in a single component of the system, thus making it possible to apply precise mitigation policy at the component level, adding to the usual traffic information the enormous amount of system statistics collected at runtime, which are ultimately the closest representatives of the resource under attack. Recent work have also shown good results for mitigating low rate DoS attacks using Probabilistic Finite Automata [11], but using only a small amount of features, and at the cost of heavy kernel instrumentation. Our platform reduces instrumentation needs by providing a general decomposed programming framework. In addition, it can perform characterization for a potentially large range of workloads.

Lastly, we note that our work can be integrated with distributed monitoring and tracing platforms such as Retro [16] and Dapper [21]. Once an attack is detected, those advanced tracing tools can allow for a finer profiling of the vulnerable execution path at low cost, as well as precise forensic to identify the component under attack.

## 6 DISCUSSION

This paper is one step towards our longer term vision of having self-driving data centers that can identify and mitigate security attacks using machine-learning techniques. We view the advent of microservices and component-based cloud application stack as an important piece of the puzzle.

In the immediate future, we are working on augmenting the number of attack type (targeting other resources than CPU) to study how one can compose policies mitigating the exhaustion of multiple resource type — potentially interleaved — while distinguishing malicious traffic from legitimate flash crowds. In addition, we are working on enhancing the scalability of our platform, for example the ability to store large volumes of monitored data in high-performance data stores. As we collect more data, we also plan to use summarization techniques, e.g. the use of LSTM (Long Short-Term Memory) to summarize temporal time series data. We also plan to scale up our learning agent, through the use of distributed controllers and distributed learning agents.

We also plan to incorporate more sophisticated learning techniques, e.g. the use of reinforcement learning to reward good mitigation techniques, while deemphasizing poor ones. In the longer term, we plan to explore mitigation techniques beyond traffic isolation. For example, rather than isolating the traffic in functional components, one could redirect the traffic to specific traffic analysis functions, which could perform more complex analysis. This is particularly useful when application developer are aware of hard-to-hide vulnerabilities in their protocol, and want to verify where they are victim of an attack or if the workload is circumstantial. In the same vein, with the intention to save space and compute time, it would be a good idea to start monitoring more advanced statistics only when an alert has been raised, or a flow flagged suspicious.

## 7 ACKNOWLEDGEMENT

This material is based upon work supported in parts by NSF Grants CNS-1703936 and CNS-1563873, and by the the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-16-C-0056, No. HR001117C0047 and No. HR0011-16-C0061. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or NSF.

## REFERENCES

[1] AWS lambda. https://aws.amazon.com/lambda.
[2] CloudFlare Workers. https://developers.cloudflare.com/workers/about/.
[3] Common vulnerabilities and exposures (see cve-2003-1564). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564.
[4] Google Cloud Functions. https://cloud.google.com/functions.
[5] OpenWhisk. https://developer.ibm.com/openwhisk.
[6] Regular expression denial of service - ReDoS. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.
[7] Spark Streaming. https://spark.apache.org/streaming/.
[8] SSL renegotiation DoS. https://www.ietf.org/mail-archive/web/tls/current/msg07553.html.
[9] Tsung. http://tsung.erlang-projects.org/.
[10] Beitollahi, H., and Deconinck, G. Analyzing well-known countermeasures against distributed denial of service attacks. *Computer Communications 35*, 11 (2012), 1312–1332.
[11] Elsabagh, M., Fleck, D., Stavrou, A., Kaplan, M., and Bowen, T. Practical and accurate runtime application protection against dos attacks. In *International Symposium on Research in Attacks, Intrusions, and Defenses* (2017), Springer, pp. 450–471.
[12] Idhammad, M., Afdel, K., and Belouch, M. Semi-supervised machine learning approach for ddos detection. *Applied Intelligence* (2018), 1–16.
[13] Jazi, H. H., Gonzalez, H., Stakhanova, N., and Ghorbani, A. A. Detecting http-based application layer dos attacks on web servers in the presence of sampling. *Computer Networks 121* (2017), 25–36.
[14] Jonas, E., Pu, Q., Venkataraman, S., Stoica, I., and Recht, B. Occupy the cloud: distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing* (2017), ACM, pp. 445–451.
[15] Krupp, J., Backes, M., and Rossow, C. Identifying the scan and attack infrastructures behind amplification DDoS attacks. In *Proc. CCS* (2016).
[16] Mace, J., Bodik, P., Fonseca, R., and Musuvathi, M. Retro: Targeted resource management in multi-tenant distributed systems. In *NSDI* (2015), pp. 589–603.
[17] Ni, T., Gu, X., Wang, H., and Li, Y. Real-time detection of application-layer ddos attack using time series analysis. *Journal of Control Science and Engineering 2013* (2013), 4.
[18] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.
[19] Pescatore, J. DDoS attacks advancing and enduring: A SANS survey. Tech. rep., SANS Institute, 2014.
[20] Ryba, F. J., Orlinski, M., Wählisch, M., Rossow, C., and Schmidt, T. C. Amplification and DRDoS attack defense – a survey and new perspectives. *CoRR abs/1505.07892* (2015).

[21] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a large-scale distributed systems tracing infrastructure. Tech. rep., Technical report, Google, Inc, 2010.

[22] Wang, C., Miu, T. T., Luo, X., and Wang, J. Skyshield: A sketch-based defense system against application layer ddos attacks. *IEEE Transactions on Information Forensics and Security 13*, 3 (2018), 559–573.

[23] Yu, S., Zhou, W., Jia, W., Guo, S., Xiang, Y., and Tang, F. Discriminating ddos attacks from flash crowds using flow correlation coefficient. *IEEE Transactions on Parallel and Distributed Systems 23*, 6 (2012), 1073–1080.

[24] Zhou, W., Jia, W., Wen, S., Xiang, Y., and Zhou, W. Detection and defense of application-layer ddos attacks in backbone web traffic. *Future Generation Computer Systems 38* (2014), 36–46.