
# Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FineLame

**Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan,** ***University of Pennsylvania***

https://www.usenix.org/conference/atc19/presentation/demoulin

**This paper is included in the Proceedings of the 2019 USENIX Annual Technical Conference.**

**July 10–12, 2019 • Renton, WA, USA**

ISBN 978-1-939133-03-8

**Open access to the Proceedings of the 2019 USENIX Annual Technical Conference is sponsored by USENIX.**

# Detecting Asymmetric Application-layer Denial-of-Service Attacks In-Flight with FINELAME

Henri Maxime Demoulin Isaac Pedisich Nikos Vasilakis
Vincent Liu Boon Thau Loo Linh Thi Xuan Phan
*University of Pennsylvania*

## Abstract

Denial of service (DoS) attacks increasingly exploit algorithmic, semantic, or implementation characteristics dormant in victim applications, often with minimal attacker resources. Practical and efficient detection of these asymmetric DoS attacks requires us to (i) catch offending requests in-flight, before they consume a critical amount of resources, (ii) remain agnostic to the application internals, such as the programming language or runtime system, and (iii) introduce low overhead in terms of both performance and programmer effort.

This paper introduces FINELAME, a language-independent framework for detecting asymmetric DoS attacks. FINELAME leverages operating system visibility across the entire software stack to instrument key resource allocation and negotiation points. It leverages recent advances in the Linux extended Berkeley Packet Filter virtual machine to attach application-level interposition probes to key request processing functions, and lightweight resource monitors—user/kernel-level probes—to key resource allocation functions. The data collected is used to train a model of resource utilization that occurs throughout the lifetime of individual requests. The model parameters are then shared with the resource monitors, which use them to catch offending requests in-flight, inline with resource allocation. We demonstrate that FINELAME can be integrated with legacy applications with minimal effort, and that it is able to detect resource abuse attacks much earlier than their intended completion time while posing low performance overheads.

## 1 Introduction

Denial-of-Service (DoS) attacks aim to hinder the availability of a service from its legitimate users. They work by overwhelming one or more of the resources of the service (*e.g.*, , CPU, network, memory, or disk), causing the service to become slow or, in the limit, entirely unavailable.

Classic DoS attacks are simple in structure: attackers, in large-scale, brute-force volumetric attacks, send many requests that far exceed the service's available resources. Although potentially crippling—sometimes reaching aggregate volumes of terabits per second [24, 43]—many effective mitigation techniques have been developed over the years, including commercial services like CloudFlare, Akamai, or the intrusion detection systems of Arbor Networks.

In response to these defenses, recent attacks have become much more sophisticated in nature: rather than relying on the sheer volume, they take the form of highly specialized, application-specific *asymmetric* DoS (ADoS) attacks [11, 12, 36, 48]. These attacks contain carefully-crafted, pathological payloads that target algorithmic, semantic, or implementation characteristics of the application's internals. They require significantly lower volumes of traffic and attacker resources to compromise resource availability. With the prevalence of third-part libraries, broad swaths of applications can be vulnerable to a given attack. For instance, the Regular-Expression DoS (ReDoS) attack [12, 13, 51] affects many programs that use regular expressions by leveraging algorithmic complexity to craft a single payload of a few characters that can occupy a service for several hours.

Due to this increase in sophistication, existing defenses are becoming inadequate [10, 26–28, 31, 40, 54, 60–62]. Network-based defenses are generally ineffective against ADoS attacks because these attacks lack identifiable problematic patterns at the network level. To be successful, network tools would not only need to perform deep packet inspection, but would also need to be able to predict which requests will hog resources a priori—a challenge analogous to solving the halting problem. Similarly, existing application-level defenses are limited in their efficacy: since these attacks can target arbitrary resources and arbitrary components of the service, which may be written in different programming languages and contain multiple binary third-party packages whose source code is not available or with complex dependencies, manual instrumentation of the application is prohibitively difficult, expensive, and time-consuming.

This paper presents the design and implementation of FINELAME (Fin-Lahm), a practical framework for detect-

ing ADoS attacks. In FINELAME, users only need to annotate their own code to mark the start and end of request processing; in many cases, annotations are not even required as applications lend themselves naturally to this demarcation. Our interaction with the most recent Apache Web Server[1] and Node.js[2] versions, for example, involves tracing three and seven functions, respectively, and not a single modification in their source code. Based on the annotations, FINELAME automatically tracks CPU, memory, storage, and networking usage across the entire application (even during execution of third-party compiled binaries). It does so with low overhead and at an ultra-fine granularity, which allows us to detect divergent requests before they leave the system and while they are attempting to exhaust resources.

Enabling our approach is a recent Linux feature called extended Berkeley Packet Filter (eBPF). eBPF enables the injection of verified pieces of code at designated points in the operating system (OS) and/or application, regardless of the specific programming language used. The OS is a natural, *de facto* layer of resource arbitration, with extensive infrastructure and pluggable tooling for fine-grained resource monitoring and distribution. By interposing on key OS services, such as the network stack, the scheduler, and user-level memory management facilities, FINELAME can detect abnormal behavior in a unified fashion across the entire software stack at run time.

FINELAME consists of three synergistic components that operate at the user/kernel interface. The first component allows attaching application-level interposition probes to key functions responsible for processing requests. These probes are based on inputs from the application developers, and they are responsible for bridging the gap between application-layer semantics (*e.g.*, HTTP requests) to its underlying operating system carrier (*e.g.*, process IDs). Examples of locations where those probes are attached include event handlers in a thread pool. The second component attaches resource monitors to user or kernel-space data sources. Examples of such sources include the scheduler, TCP functions responsible for sending and receiving packets on a connection, and the memory manager used by the application. To perform anomaly detection, a third component deploys a semi-supervised learning model to construct a pattern of legitimate requests from the gathered data. The model is trained in the user space, and its parameters are shared with the resource monitors throughout the system, so that anomaly detection can be performed in-line with resource allocation.

In summary, we make the following contributions:

- A novel, backward-compatible architecture at the user/kernel interface for transparently implanting resource monitors, exposed to applications via a probe API.
- A library of resource monitors and associated probes that can be used to detect asymmetric DoS attacks.
- An eBPF-based implementation and evaluation of FINELAME on Linux.

[1] 2.4.38 at the time of this writing

[2] v12.0.0-pre, 4a6ec3bd05e2e2d3f121e0d3dea281a6ef7fa32a on the Master branch at the time of this writing

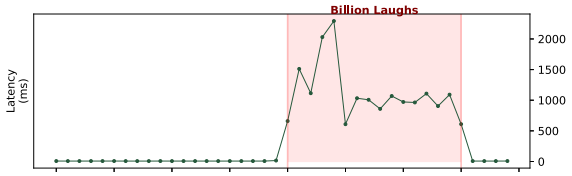



**Fig. 1: Billion Laughs (XML Bomb) Attack.** Under a normal load of about 500 requests per second, legitimate users experience a median of 6.75ms latency. After a short period of time, we start a malicious load of 10 requests per second (shaded area). XML bombs can take up to 200ms to compute (*vs.* a median of about 60*ns* for normal input). As a results, legitimate requests get serviced much slower, experiencing up to 2s latency. Setup details covered in (§6).

Our evaluation shows that FINELAME requires low additional instrumentation overhead, requiring between 4-11% additional overhead for instrumenting web applications ranging from Apache, Node.js, and DeDOS [15]. Moreover, when evaluated against real application-layer attacks such as ReDOS [5], Billion Laughs [3], and SlowLoris [46], FINELAME is able to detect the presence of these attacks in near real-time with high accuracy, based on the attack deviation from normal behavior.

The rest of the paper is structured as follows: it first motivates FINELAME's goals by providing a brief overview of asymmetric DoS attacks (§2); it then lays out our threat model and assumptions (§3); it describes the FINELAME's design and its three component parts, (i) request mapping (§4.1), (ii) resource monitoring (§4.2), and (iii) anomaly detection (§4.3); it next details several prototype implementations (§5) and evaluates the FINELAME prototypes' intrusiveness, overheads and accuracy, using a combination of micro-benchmarks and real applications (§6); finally, it compares with prior work (§7) and concludes with a discussion of limitations and possible directions for future research (§8).

## 2 Motivation

We begin by showing via an example server-side application the operation of an ADoS attack, the limitations of current detection mechanisms, and design goals for our system.

### 2.1 Background on ADoS Attacks

Fundamentally, asymmetric DoS attacks are attacks that leverage application-specific behaviors to cause disproportionate harm to the system using comparatively low amount of attacker resources. They can target any layer of the stack and

any resource within the system. ADoS vulnerabilities are widespread and often affect entire software ecosystems [41]. We detail a few of them below.

**Regular-expression DoS (ReDoS) [12, 13, 51].** ReDoS attacks target programs that use regular expressions. Attackers craft patterns that result in worst-case asymptotic behavior of a matching algorithm. An example pattern is `(a+)+`, which does not match any string of the form `a*X`, but requires the system to check $2^N$ decomposition of the pattern to reach that conclusion, where $N$ is the length of the target string.

**XML Bomb [3].** An XML bomb (or Billion-Laughs attack) is a malicious XML document that contains layers of recursive data definitions[3], resulting in quadratic resource consumption: a 10-line XML document can easily expand to a multi-gigabyte memory representation and consume an inordinate amount of CPU time and memory on the server. Fig. 1 illustrates the impact of XML bombs on the latency of requests on a susceptible server. Under normal operation, a load of 500 legitimate requests per second are served in less than 10 milliseconds each; under a low-volume attack of 10 XML bombs per second, the latency jumps up to more than two seconds. An XML bomb affects any serialization format that can encode references (*e.g.*, YAML, but not JSON).

**Improper (de-)serialization [47, 52, 53].** This class of attacks encompasses those where malicious code can be injected into running services. These vulnerabilities are, unfortunately, common in practice, and they allow malicious users to, for instance, inject a `for (;;) {}` loop to stall a process indefinitely.

**Event-handler Poisoning (EHP) [14].** Attacks like the preceding can be additionally amplified in an event-driven framework. In event-handler poisoning, attackers exploit the blocking properties of event-driven frameworks so that, when a request unfairly dominates the time spent by an event handler, other clients are further blocked from proceeding. Any slowdown, whether it is in the service itself or in its recursive layers of third-party libraries can contribute to this head-of-line blocking.

## 2.2 Design Goals

The attacks in the previous section highlight several goals that drive FINELAME's design (§4) and implementation (§5).

**In-flight Detection.** Actions often need to be taken while the offending requests are "in the work"—for example, when a single request can bring the system down (*e.g.*, cooperative scheduling) or when subsequent defenses cannot be deployed (*e.g.*, IP spoofing). DoS detection needs to catch such requests *before* they leave the system, by monitoring resource consumption at a very fine temporal and spatial granularity.

[3] For example, the first layer consists of 10 elements of the second layer, each of which consists of 10 elements of the third layer, and so on.

**Resource Independence.** ADoS attacks may target arbitrary system-level resources (CPU, memory, storage, or networking), and may even target multiple resources (*i.e.*, multi-vector attacks). A desirable solution needs to be agnostic to the resource and able to handle any instance of inordinate consumption.

**Cross-component Tracking.** Given the complex structure of modern applications, ADoS attacks can also cross the boundaries of the application's internal components or processing phases. For instance, if a request causes the triggering of a timeout to an event queue, resources consumed by the initial request parsing and the timeout should both be attributed to the same request.

**Language Independence.** Applications today combine several ready-made libraries, which are written in multiple programming languages and often available only as compiled binaries. Thus, DoS detection should remain agnostic to the application details such as the programming language, language runtime, and broader ecosystem (*e.g.*, packages, modules).

**Minimal Developer Effort.** Detection needs to impose minimal burden to developers and devops, who should benefit from DoS mitigation without having to study the application internals. Rather than presenting developers with an overabundance of configuration knobs, a DoS detection system should direct precious human labor at sprinkling applications with key semantic information utilized at runtime.

# 3 Threat Model

To be more concrete, FINELAME assumes the following about the attacker and the broader environment.

**Threats.** We consider a powerful remote attacker that (i) can send arbitrary requests to a service hosting a vulnerable application, (ii) has control over potentially all of the application's legitimate clients, and (iii) is aware of the application's structure and vulnerabilities, including exploits in its dependency tree. We do not distinguish between legitimate and malicious clients who intersperse harmful requests that attack resources with one or more benign requests. Specifically, any subset of hosts can send any number of requests that may or may not attack any subset of resources. We do not limit resources of interest to CPU; attackers can target memory, file descriptors, or any other limited resource in the host system. That means that attacks can take the form of a single client attempting to consume 100% of the CPU indefinitely, or of multiple attacks from multiple clients over many of the system's resources.

**Assumptions.** We assume (i) vulnerable but not actively malicious code, and (ii) that FINELAME sees at least some benign traffic. If all traffic is malicious from the beginning, in-flight detection and mitigation become less urgent, as anomalies become the norm, and the application owners should first

revise their deployment pipeline. We also assume that the resource utilization of request processing can be attributed to a single request by the end of each processing phase, even if the processing phases is split into multiple phases across different application components. As keeping a reference to the originating request is a natural design pattern, in all of the services we tested, a unique identifier was already available; in cases where there is no such identifier, one must be added, and we detail how to do so in section 4.

# 4 FINELAME Design

Figure 2 depicts the overall design of FINELAME. Conceptually, FINELAME consists of three main components:

- *Programmer annotations* that mark when a request is being processed. FINELAME requires only a few annotations, even for complex applications, to properly attribute resource utilization to requests.
- *Fine-grained resource monitors* that track the resource utilization of in-flight requests at the granularity of context switches, mallocs, page faults.
- A *cross-layer anomaly detection* model that learns the legitimate behavior and detects attacks as soon as they deviate from such behavior.

Programmers can use FINELAME by annotating their application with what we call *request-mappers*. These annotations delineate, for each component and processing phase, the start and end of processing, as well as the request to which resource utilization should be attributed. For example, in an event-driven framework, the beginning and the end of each iteration of the event handler loop should be marked as the start and the end of a request's processing, respectively.

At runtime, when FINELAME is installed on the host environment, FINELAME attaches small, low-overhead *resource monitors* to particular points in the application or operating system. The aforementioned request-mappers enable FINELAME to determine the request to which the resource consumed by a thread or process should be credited. In section 5, we detail our out-of-the-box FINELAME library of request-mappers and resource monitors for several popular cloud frameworks. Our library tracks the utilization of a range of key OS-level resources; however, programmers can further extend it with user-level resource monitors to track application-specific resources (*e.g.*, the occupancy of a hash table).

Finally, FINELAME's monitoring data is used to perform lightweight, inline anomaly detection. Resource monitors first feed data to a machine learning model training framework that computes a fingerprint of legitimate behavior. Parameters of the trained model are installed directly into the resource monitors, which evaluate an approximation of the model to automatically detect anomalous behavior on-the-fly. The end result of FINELAME is a system for high-accuracy, fine-grained, and general ADoS attack detection.

## 4.1 Request-mapping in FINELAME

Conceptually, there are three operations in request mapping:

- `startProcessing()`: This annotation denotes the start of a processing phase. Any resource utilization or allocations after this point are attributed to a new unique request.
- `attributeRequest(reqId)`: As soon as we can determine a unique and consistent request identifier, we map the current processing phase to that request. For instance, when reading packets from a queue, if the best consistent identifier for a packet is its 5-tuple, resource tracking would start as soon as the packet is dequeued, but would only be attributed to a consistent request ID after Layer-3 and Layer-4 processing are completed. In general, `attributeRequest(reqId)` is called directly after `startProcessing()`, and depending on the specific of the application, the two can sometimes be merged (§ 5).
- `endProcessing()`: Finally, this operation denotes the completion of processing, indicating that subsequent utilization should not be attributed to the current request.

In order for the resource monitors to properly attribute utilization to a request, FINELAME requires programmers to annotate their applications using the above three *request mapping* operations. Ideally, the annotations should cover as much of the code base as possible; however, not all resource utilization can be attributed to a single request. In such cases, programmers have flexibility in how they perform mapping: for true application overhead—rather than request processing overhead—utilization can remain unattributed, and for shared overhead (*e.g.*, garbage collection), utilization can be partitioned or otherwise assigned stochastically.

Every request is given an identifier that must be both unique and consistent across application components and processing phases. This identifier is used to maintain an internal mapping between OS entity (process or thread) and the request. Example identifiers include the address of the object representing the request in the application, a request ID generated by some application-level tracing solution [7, 20, 29, 34, 45, 49, 55], or a location in memory if the request is only processed once. From the moment a `startProcessing` annotation is called to the moment the `endProcessing` annotation is called, FINELAME will associate all the resources consumed by the OS entity to the request.

An optimization of this technique can be implemented when the application lends itself naturally to such mapping

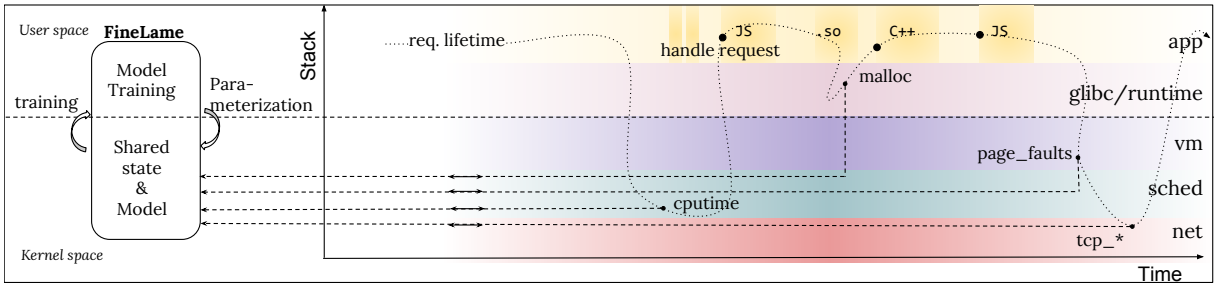



**Fig. 2: FINELAME overview.** Key elements: (1, right) user and kernel data-collection probes at points where an HTTP request interacts with resource allocation; (2, mid-left) a data structure shared between user and kernel space, that aggregates and arranges collected data; (3, left) a userspace training component that instantiates model parameters, fed back to the probes. Information flow between 1–3 is bidirectional.

between OS entity and request. For instance, event-driven frameworks or thread-pool based services usually have a single or small number of entry points for the request, to which FINELAME can readily attach request-mappers via eBPF without source code modification. We found this optimization to be the common case, and FINELAME does not require any modification to the application we explore in section 6.

## 4.2 Resource Monitoring in FINELAME

Resource tracking between `startProcessing` and `endProcessing` annotations are done via a recent Linux kernel feature called eBPF. We first provide some background on the operation of eBPF, and then discuss how we utilize it to perform extremely fine-grained resource monitoring of in-flight requests.

### 4.2.1 Background on eBPF

The original Berkeley Packet Filter (BPF) [35] has been a long-time component of the Linux kernel networking subsystem. It is a virtual machine interpreting a simple language traditionally used for filtering data generated by kernel events. Notable use cases are network packets parsing with Tcpdump [56] and filtering access to system calls in the seccomp facility. In version 3.0 a just-in-time compiler was implemented, allowing for a considerable speedup of the processing of BPF programs by optimizing them on the fly.

In version 3.15, Alexei Starovoivtov significantly extended BPF (dubbing the new system "eBPF"). The new version has access to more registers and an instruction set mimicking a native RISC ISA, can call a restricted subset of kernel functions, and can share data from kernel-space to user-space through hash-like data structures. While eBPF is a low-level language, users can write programs in higher languages such as C (and even Python with the BCC project [2]) and generate eBPF code with compilers such as GCC and LLVM.

Generated programs are verified before being accepted in the kernel. The verifier imposes a set of strict constraints to eBPF programs to guarantee the safety of the kernel. Common constraints include the absence of floating point instructions, a limit of 4096 instructions per program, a stack size capped at 512 Bytes, no signed division, and the interdiction of back-edges in the program's control flow graph (*i.e.*, no loops).

The ability of eBPF programs to be attached to both kernel and user-space functions and events, their extremely low overhead, and their ability to share data with user space without the need for any IPC or queuing mechanism make eBPF a prime candidate for implementing resource monitors in FINELAME.

### 4.2.2 Resource Monitor Architecture

FINELAME's resource monitors are attached to various user- and kernel-space data sources (*e.g.*, the scheduler or TCP stack) and use the mapping described in section 4.1 to associate resource consumption to application-level workflow (*e.g.*, HTTP requests). A resource monitor requires the following information: the type and name of the data source, and potentially the path of its binary.

Our current prototype of FINELAME uses the features listed in Table 1. When executed, most resource monitors operate under the following sequence of actions: i) verify whether a request mapping is active for the current PID and exit if not; ii) collect the metric of interest (usually through the arguments of the function triggering it) and store it, time-stamped, in a shared data structure; and iii) perform anomaly detection on the request if the model's parameters are available (see section 4.3).

The time a request spends executing instructions on a processor is represented by *cputime*. We instrument both the *scheduler_tick()* and the *finish_task_switch()* kernel functions, which are called at every timer interrupt and context switch, respectively, to either start a timer when a thread executing a registered request is scheduled for execution or collect the amount of CPU time consumed by the task swapped out. We instrument the *tcp_sendmsg()* and *tcp_rcleanbuf()* to collect *tcp_sent* and *tcp_rcvd*, the amounts of bytes sent and read from a TCP connection, respectively. To compute *tcp_idle_time*, which represents the period of inactivity from the sender on a TCP connection, we measure the time elapsed between two occurrences of *tcp_cleanup_rbuf()*. To monitor the heap memory consumption occasioned by the processing

| Name | Description | Event | Type |
|---|---|---|---|
| tcp_idle_time | Inactivity time on a TCP connection | tcp_cleanup_rbuf | kernel probe |
| tcp_sent | Bytes sent through TCP connections | tcp_sendmsg | kernel probe |
| tcp_rcvd | Bytes received through TCP connections | tcp_cleanup_rbuf | kernel probe |
| cputime | Amount of CPU time consumed | scheduler_tick, finish_task_switch | kernel probe |
| malloc_memory | Bytes allocated through the *malloc* function | glibc_malloc | user probe |
| page_faults | Number of page faults events | exceptions:page_fault_user | kernel tracepoint |

**Tab. 1: Default resource monitors in FINELAME.**

of a request, we monitor the glibc *malloc* function. Applications where memory management is partly handled by the runtime (such as in Python) can be monitored in a similar fashion. Likewise, the model can be generalized to garbage collected languages. Finally, we monitor the page fault events in the application by attaching a resource monitor to the *exception: page_fault_user* kernel tracepoint. We observed in our evaluation that CPU time was the best discriminant for CPU based attacks, while connection idle time the best for slow attacks (such as Slowloris and RUDY).

The above default, general-purpose resource monitors in FINELAME are sufficient for a large set of existing applications; however, it can be extended to all the kernel events available for tracing and probing, as well as user-level functions (to monitor application-level metrics). If any application-level metrics are required (such as data structure occupancy, counters, and so on), programmers can augment our resource monitors with custom eBPF programs attached to arbitrary probe points in either kernel- or user-space.

## 4.3 Attack Detection in FINELAME

**Detection algorithm.** For fast detection, FINELAME is designed to enable anomaly detection as close as possible to the resource allocation mechanism. Without a method for in-flight anomaly detection *in addition to* mechanisms for in-flight resource tracing, detection and mitigation of in-flight requests would not be possible.

This detection problem can be reduced to quantizing the abnormality of a vector in $n$-dimensional space. Once a sufficient amount of data has been gathered to compute a fingerprint of the legitimate requests' behavior, we can train an anomaly detection model. The model can span all the metrics collected by the resource monitors, allowing us to detect abuse on any of the resources of the system as well as cross-resource (multi-vector) attacks.

For the unsupervised version of this problem, the most popular methods take one of two approaches: distance-based or prediction-based. The former family of models aims to cluster known, legitimate data points and compute the distance of new data points to those clusters—distance that is used to quantify the anomaly. The latter family assumes the existence of a set of input data points that are correct, and learns a function representing those points. When a new point enters the system, the model computes the value of the learned function; the prediction error is then used to quantify the degree of

```
Required data structures
FPAS          # FPA scaling factor
pid_to_rid    # OS carrier to request
req_points    # Request profiles
model_params  # K-means parameters
dp_dists      # Distances to centroids
thresholds    # Alerts cut-off bar

fun resource_monitor(context):
 pid = bpf_get_current_pid()
 rid = pid_to_rid.get(pid)
 if (rid):
    ts = get_timestamp()
    metric = context.get_arguments()
    dp = req_points.get(rid)
Is there a mapping?
    if (dp):
       dp.update(metric, ts)
Update request profile
    else:
       dp = init_dp(rid, metric, ts)
       req_points.insert(dp)
    μ, σ = model_params.get()
    if (μ && σ):
       metric_scaled = metric << FPAS
       metric_scaled -= μ
       if metric_scaled < 0:
           metric_scaled *= -1
           metric_scaled /= σ
           metric_scaled *= -1
       else:
           metric_scaled /= σ
If anomaly detection parameters are available, scale and standardize the data in the FPA space
       min_dist, closest_k
       #pragma loop unroll
       for k in K:
           current_dist =
             dp_dists.get(dp, k)
           new_dist =
             metric_scaled+current_dist
Update distance to clusters
           dp_dists.update(dp, new_dist)
           if (new_dist < min_dist):
               min_dist = new_dist
               closest_k = k
       t = thresholds.get(closest_k)
Perform anomaly detection
       if new_dist > t:
           report(rid, dp, s)
```

**Fig. 3: FINELAME anomaly detection.** Pseudocode for FINELAME's inline anomaly detection.

anomaly.

Because of the training complexity, prediction complexity, and required training data, many existing solutions in both distance-based and prediction-based categories are impractical to execute at fine granularity. For instance, the popular algorithm DBSCAN [18] is not suitable for FINELAME, as it requires us to evaluate the distance of new data points to all the possible "core" data points in the model. The amount of data points considered (and therefore the size of the model) is usually linearly proportional to the size of the training set. Some accurate approximations of DBSCAN have been proposed [22], but even with a small number of clusters, almost all of the training dataset still needs to be part of the model. Likewise, the performance of prediction-based models made on neural networks, such as Kitsune [38], is highly dependent on the depth and width of the model. The amount of parameters of such networks grows exponentially with the number and size of the hidden layers.

Given the above concerns, we chose to implement anomaly detection in FINELAME with $K$-means, a technique that allows us to summarize the fingerprint of legitimate requests with a small amount of data. In $K$-means, the objective function seeks to minimize the distance between points in each cluster. The model parameters are then the centroids and distribution of the trained clusters. In a typical use-case scenario, FINELAME is configured to perform only request monitoring for a certain amount of time, after which it trains $k$-means on the monitoring data gathered in user-space from the resource monitors shared maps. In practice, we found that a $K$ value equal to the number of request types in the application yields a reasonable estimation of the different behaviors adopted by legitimate requests, while being a number low enough such as to contain FINELAME's overhead.

**Model training and deployment.** Gathering the training data is done by a simple look-up from the user-space agent to the shared eBPF maps holding the requests resource consumption data. Using those profiles, the user-space agent standardizes the data (center to 0 and cast to unit standard deviation). Subsequently, the agent trains $K$-means to generate a set of centroids representing the fingerprint of the good traffic. The parameters of the model, to be shared with the performance monitors, are then the cluster centroids, as well as the mean $\mu$ and standard deviation $\sigma$ of each feature in the dataset, and a threshold value $\tau$ statistically determined for each cluster.

As described above, the performance monitors have limited computing abilities and do not have access to floating point instructions. Thus, they are designed to perform fixed point arithmetic in a configurable shifted space, and require FINELAME's to shift the model parameters in this space before sharing them. Using two precision parameters $a$ and $b$, each datapoint is transposed in a higher space $10^a$, and normalized such that the resulting value lies in an intermediate space $10^{a-b}$, retaining a precision of $a-b$ digits. This means that during the normalization operation each parameter value $x$ undergoes the following transformation: $x = \frac{(x*10^a)-(\mu*10^a)}{\sigma*10^b}$.

Once standardized, the clusters' centroids as well as each feature's mean and standard deviation are shared with the resource monitors through eBPF maps. Upon availability of those parameters, the resource monitors update not only the resource consumption of existing requests, but also their *outlier scores*, a measure we use to quantify the degree of anomaly of a request. Due to the constraints imposed on eBPF programs—specifically, taking a square root is complex as we do not have access to loops—we choose the normalized L1 distance to the closest cluster as the outlier score. While being a crude measure, the L1 is equivalent to more complex norms as resource vectors are of finite dimension. It preserves information about which resource is abused, and it lets us set statistical thresholds to determine cut-off points used for flagging abnormal requests. The algorithm for this entire process is shown in Figure 3.

Finally, we note that because FINELAME is primarily designed toward the detection of resource exhaustion attacks, we allow the anomaly detection engine to maintain signed values for outlier scores. This means that requests that have not reached their expected legitimate amounts of resource consumption, and that would look abnormal in an absolute value setting, are not flagged as such. This is important because it highlights the fact that FINELAME is not geared toward volumetric attacks that aim to bring the system down with a vast amount of low consumption requests.

| Application | Request mapping probes | SLOC |
|---|---|---|
| `Apache` | 5 | 41 |
| `Node.js` | 9 | 64 |
| `DeDoS` | 2 | 21 |

**Tab. 2: Intrusiveness of FINELAME, quantified.**

## 5 Use Cases and Implementation

To demonstrate the generality of FINELAME and the minimal developer effort required to use it, we apply FINELAME to three web platforms: Apache [1], which is estimated to serve ∼40% of all active webpages; Node.js [4] a popular server-side JavaScript-based web server; and DeDoS [15] an open source component-based framework for building web services. Our prototype of FINELAME is available on `https://github.com/maxdml/Finelame`. Table 2 quantifies the programming effort required to write request-mappers for those three applications to use FINELAME.

**Apache web server**. Primarily written in C, Apache's request processing is implemented by Multi-Processing Modules (MPM). In the latest versions of Apache (2.x), requests are served by multiple processes which can have multiple

worker threads themselves; each thread handles one connection at a time.

When a request enters the system, an application-level (`conn`) object is created by the `core_create_conn` function to contain it before the request is dispatched to a worker thread. Subsequently, the request is processed by either the `ap_process_http_sync_connection` or the `ap_process_http_async_connection` functions, which take the `conn` object as argument. From FINELAME, we attach one request-mapper to `core_create_conn`, and two requests-mappers to the http processing functions, one over a *uprobe* called upon entering the function, the other over a *uretprobe* called when returning from it. We exploit the `conn` object to generate a unique identifier for each request and map it to the underlying thread worker, so that resource monitors can later gather resource consumption data on the request's behalf. The mapping is undone when the function returns and the request exits the system. When a worker thread executes a new request, the request-mapper updates the mapping with the new request's ID. This solution requires no modification to the Apache source code, and 41 lines of eBPF code over 5 probes.

**Node.js** required more slightly more instrumentation due to its asynchronous model, which offloads work to a worker pool (implemented with *libuv* [30]). The instrumentation required eBPF probes to be attached to seven user-space functions within the libuv library. As in Apache, we found a data structure—`struct uv_stream_t`—that could (i) be used to generate a unique identifier, and (ii) was carried consistently across the disparate components of the framework.

Request-mappers were applied to the seven libuv functions as follows:

- `uv_accept`: a new request is initialized, and is associated with the `uv_stream_t` structure that handled communication with the client.
- `uv__read` and `uv__write`: the request associated with the client's stream is assigned to the current thread for the duration of the function.
- `uv__work_submit`: the request assigned to the current thread is associated with a work-request submitted to the worker pool.
- `uv__fs_work`, and `uv__fs_done`: the request associated with the work-request is assigned to the current (worker) thread.
- `uv_async_send`: the request is unassigned from the current thread.

Again, this solution requires no changes in Node.js source code, only knowledge of which functions are processing requests. The request-mappers totalized 64 lines of eBPF code.

**DeDoS** is an event-driven framework where programmers write and deploy their application as software components that are automatically allocated and deallocated based on demand. Each of those components monitor a local event-queue from which new requests are consumed. Unifying the disparate components is a generic event-handling function (`receive()`). Programmers implement their component's functionality inside this event-handling function.

DeDOS provides request tracing and explicitly tracks the passing of requests between components. We chose DeDoS as a proof-of-concept proxy for micro-service, event-driven applications providing request tracing capability. In these types of applications, annotation is simple as FINELAME can maintain a direct mapping between the application-level unique request identifier and the event handler's thread PID in order to track resource consumption across component boundaries. FINELAME traces only the `receive()` function class with request mappers, and does not require modifications to the framework. The request-mappers require 21 lines of eBPF code.

## 6 Evaluation

In this section, we present our evaluation results of FINELAME. Our evaluation is centered around the following aspects of the system:

- **Overhead.** The overhead of FINELAME compared to no monitoring, or in-application instrumentation
- **Accuracy.** The ability of FINELAME to accurately detect real attacks never seen yet by the application

### 6.1 Experimental setup

We present the setup on which we evaluate both the overhead and accuracy aspects of FINELAME. In all cases, the server applications are running on a 12 cores Xeon Silver 4114 at 2.20GHz , while our legitimate and attack clients are running on an Intel Xeon E5-2630L v3 at 1.80GHz. Both server and client machines have a total of 62G of RAM, and have hyper-threading and DVFS disabled.

We use version 2.4.38 of Apache, and configure it to use 50 worker threads. We use version 12.0.0 – *pre* of Node.js with the default configuration of 4 worker threads for libuv. Both Apache and Node.js are configured to serve a set of Wikipedia [59] pages. Node.js parses a regular expression provided in the request's URI to find the path of the file to serve. It's parser, *liburi*, is vulnerable to the ReDoS attack. All the applications impose a timeout of 20 seconds on connections. We deploy a simple webserver in DeDoS which can process three types of requests: serve a Wikipedia article, process a randomly generated XML file uploaded in a POST request, and parse a regular expression. The server is decomposed into

several software components: socket reading, HTTP parsing, file serving, XML parsing, regular expression parsing, and response writing. The XML parser is implemented with *libxml2*, which is vulnerable to the Billion Laughs attack.

Our good traffic is generated by Tsung [6] and explores evenly all the servers' exposed endpoints; bad traffic is generated by an in-house **C** client for the ReDoS and Billion Laughs attacks, and pylorys [23] for the Slowloris attack. Tsung generates load under an exponential distribution centered on a configurable mean, while our attack client is configured to send a fixed load.

## 6.2 Overhead of FINELAME

Figures 4 presents the overheads incurred by FINELAME's instrumentation on Apache, Node.js and DeDoS. In all of our experimental setups, we evaluate the legitimate client latency experienced when the server is not instrumented, when it is instrumented by FINELAME, and when FINELAME's resource monitors are also performing anomaly detection (FINELAME +). The load is as described earlier in sec 6.1, and explore all the instrumented paths in the applications. We also evaluate the cost of instrumenting the DeDoS framework itself to evaluate FINELAME overheads compared to a traditional user-space solution. The bars plot the median of the clients latency, and all our experiments are run thrice for a period of 100 seconds. In the case of Node.js the instrumentation cost adds 8.55% overheads and adding anomaly detection 9.21%. In the case of Apache, FINELAME adds 11.38% and 11.72% overheads respectively. In the case of DeDoS, the baseline latency is higher than with the two previous services, due to the fact that the application is not only serving files but also parsing POST requests, and also the framework is less optimized than the two battle-tested Apache and Node.js. Instrumenting directly the framework comes with an overhead of 2.9%, while FINELAME comes with 4.23% overheads, 6.3% if also performing anomaly detection.

In general we observe that the overheads incurred by FINELAME are higher when the baseline processing time of the service is low, and does not grow linearly with the complexity of the application. In addition, we found that performing anomaly detection in addition to monitoring resource consumption almost comes for free.

## 6.3 Performance of FINELAME

https://www.overleaf.com/project/5c22751775031d099f528e64
Our performance evaluation of FINELAME is centered around its ability to detect attacks requests before they exit the system, while providing accuracy competitive with non-approximated user-level algorithms.

### 6.3.1 Attacks

Our experiments aim to quantify the impact of attacks on quality of service. Consequently, we tune attacks strength such that they will not bring down the server but rather degrade the quality of service provided to legitimate users.

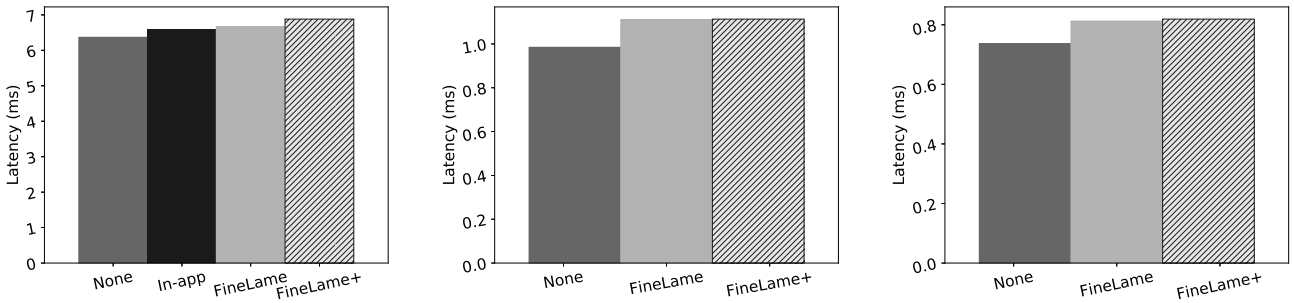
**ReDoS**: This attack consist of specially crafted regular expressions which are sent to the server for processing. The strength of the attack grows exponentially with the number of malicious characters present in the expression. Because the application processing units are busy handling those requests, legitimate requests get queued for a longer period of time, and ends-up being responded to more slowly.

**Billion Laughs:** The attack consists of XML files filled with several levels of nested entities. The parsing cost is exponentially proportional to the depth of the document. The impact is similar to the ReDoS attack.

**SlowLoris:** The attack consists in maintaining open connections to the server, keeping them alive by sending individual HTTP headers at a regular interval smaller than the server's timeout, but never completing the request—we assume that the attacker is able to probe the service and discover this timeout. As a result, the server's connection pool gets exhausted, and it can't answer new requests. This technique can also implement a dormant attack which cripples the ability of the server to handle surges of legitimate traffic, by denying a fraction of the total connection pool.

### 6.3.2 Anomaly Detection Performance

**Evaluation metrics** As is common with anomaly detectors, the output of FINELAME is a score which quantifies the abnormality of a request. This score is then either used as a raw metric for mitigation algorithms, or compared against a threshold $\tau$ to be transformed into a binary variable where 0 means negative (no anomaly), and 1 means positive (attack). With $\tau$ set, and using our knowledge of the ground truth, we can determine the accuracy of each of the detector's outputs as true/false positive/negative. The choice of $\tau$ is crucial, as too low a value can result in a large amount of false positive, while too high a value can induce a large amount of false negative. For our experiments, we set $\tau$ to be the outermost point for each cluster in the training set, *i.e.*, the most consuming legitimate request we've seen so far for the cluster. The challenge associated with deriving a large $\tau$ from the training traffic is that attacks can now take longer to detect—and might not be detected at all if they are too weak. This latter case does not concern us, because to bring down the system with weaker attacks, an attacker would be forced to change its method from asymmetric to volumetric. The benefit of a higher $\tau$ is that it helps decreasing the False Positive Rate (FPR, $\frac{FP}{FP+TN}$), a desirable behavior for operators using the system. For our experiments, we present the True Positive




**(a)** FINELAME overheads with DeDoS **(b)** FINELAME overheads with Apache **(c)** FINELAME overheads with Node.js

**Fig. 4:** Overhead of FINELAME with various applications

Rate (TPR, $\frac{TP}{TP+FN}$), True Negative Rate (TNR, $\frac{TN}{TN+FP}$) and F1 ($\frac{2TP}{2TP+FP+FN}$). TPR evaluates the system's ability to detect all the attack requests. TNR evaluates its ability to evaluate legitimate requests as such. The F1 score is the harmonic mean of the TPR and the recall. It evaluates both the TPR and the precision of the system.

In addition to its post-hoc instrumentation abilities and low programmer burden, the main contribution of FINELAME is it's detection pace. We evaluate the *Detection Speedup* (DS) of the system, which we define as being the delta between the time of last detection and the time to first detection, over the lifetime of the request. We expect DS to increase as users set more strict thresholds (lower values of $\tau$), but found that even with $\tau$ set to the outermost point in each training cluster, FINELAME is able to detect attacks up to more than 97% faster.

**Experiments** All our experiments are run for a duration of 400 seconds, split into 3 phases. The first phase sees only legitimate traffic flowing through our target applications, and last 200 seconds. FINELAME is configured to only have the performance monitors gather data for the first 180 seconds, after which point it triggers the training of the anomaly detection model and share its parameters. Attacks start at time 200, and last for 150 seconds. A final period of 50 seconds sees the attack stop, and only good traffic is sent to the application. We perform two CPU exhaustion attacks, Billion Laughs and ReDoS, as well as a connection pool exhaustion attack, SlowLoris. For all experiments, we compare the TPR and TNR of FINELAME to a non approximated user-space implementation of $K$-means (that is, with floating point arithmetic) to confirm that the system is competitive with more complex user space solutions. We set $K = 3$, the maximum number of request types that the application we setup can accept, and use $a = 10$ and $b = 6$ factor to retain 4 digits in fixed point arithmetic.

Table 3 presents the detection speed and performance of FINELAME.

**ReDoS:** In our first experiment, we attack Node.js with three strengths of ReDoS requests. In the two first experiments, the workload is made of 98% of benign requests and 2% of malicious regular expressions blocking the event loop of the server (about 500 and 10 r/s, respectively). In the third experiment, with the strongest attack, we reduce the attack rate to 1 r/s, such that the attack does not bring down the server. Legitimate requests are served in about 0.8ms on average under normal conditions, but get delayed in proportion of the intensity of the ReDoS requests when the attack starts. During the first attack, bad requests are served in 23ms on average, a 28.75× increase compared to normal requests. Good requests are also penalized and are served in about 4ms. During the second attack, bad requests are served in 45.6ms on average, a 57× increase compared to normal requests. Legitimate requests are affected and incur an average latency of 13.5ms. During the third attack, bad requests are served in 90.9ms on average, a 113.6× increase. Legitimate requests incur an average latency of 6ms. Due to its ability to credit requests' resource consumption at the granularity of context switches, in both experiments, FINELAME is able to detect attack requests before they exist the system, at least 80.9% earlier for 50% of the bad traffic, and up to 95.3% earlier. The user-space, non-approximated evaluation of k-means using the L2 norm for measuring distances, perform only marginally better.

**Billion Laughs:** In this experiment, we attack DeDoS with two different strengths of Billion Laughs (XML bomb) requests. The good traffic follows a diurnal pattern, oscillating between 250 and 750 requests per second. Under normal conditions, legitimate requests are served in 6.87ms on average. In the first experiment, we send 15 malicious requests per seconds (about 2% of the peak legitimate traffic, and 6% of the lower phase), which are served in 29.28ms on average, a 4.26× increase in response time. In the second experiment, we decrease the number of bad requests to one per second (about 0.1% and 0.4% of the peak and low traffic, respectively), and increase their intensity such that they are served in 203ms in average (an order of magnitude increase compared to the first case), which represents a 29.55× increase

| Attack | Strength | TPR | | TNR | | F1 | | DS | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | FL | *K*-means L2 | FL | *K*-means L2 | FL | *K*-means L2 | median | 75*th* | max |
| ReDoS | 28.7× | 100% | 100% | 99.995% | 99.999% | 99.88% | 99.98% | 80.9% | 81.2% | 83.2% |
| | 57× | 100% | 100% | 99.993% | 99.994% | 99.81% | 99.83% | 90.4% | 90.5% | 91.0% |
| | 113.7× | 100% | 100% | 99.997% | 99.999% | 99.29% | 99.76% | 90.9% | 95.1% | 95.3% |
| Billion Laughs | 4.7× | 100% | 100% | 100% | 100% | 100% | 100% | 83.1% | 85.5% | 87.7% |
| | 34.8× | 100% | 100% | 99.998% | 99.998% | 99.53% | 99.76% | 97.0% | 97.1% | 98.2% |
| SlowLoris | 5 sockets | 100% | 100% | 100% | 100% | 100% | 100% | 75% | n/a | n/a |

**Tab. 3:** FINELAME TPR, and detection Speedup for Apache, Node.js and DeDoS.

in load compared to legitimate requests in normal conditions. For the weaker attack, FINELAME is able to detect malicious requests 78.83% faster than the user-space solution, at least 50% of the time, and up to and 97% faster for the strongest attack.

**SlowLoris:** In this experiment, we configure Apache to handle requests with 25 worker threads, and timeout on reading HTTP headers after 20 seconds. We configure the attack client to maintain 5 connections to the server opened at all times, refreshing it every 5 seconds. Effectively, this drives the *tcp_idle_time* of the malicious request high and makes them standout from the legitimate ones. This attack is "all or nothing", in the sense that it will not impact the legitimate requests until the connection pool gets exhausted. FINELAME's is able to detect the abnormal idle time about 75% faster than the application ($1 - \frac{5}{20} * 100$), which would have otherwise to experience the timeout before reporting the request.

## 7 Related Work

**Volumetric Attack Detection** There is a large body of work addressing volumetric DoS attacks [10, 26, 31, 40, 60–62], including attacks that target the network [27, 28, 54]. As described earlier (§1), these systems do not protect against *asymmetric* DoS attacks, a concern shared by both industry [32, 50] and academia [13, 14, 51].

**Application-based Detection** Prior works on application-layer DoS detection either depend heavily on repeated outliers, or are often deeply tied to a specific application. Techniques include comparing the entropy of offending and legitimate traffic [39, 63], sampling traffic flows [25], and sketch-based feature-dimensionality reduction [58]. While these techniques work well for volumetric attacks, they have self-assumed limitations when the attack traffic is low—the primary focus of this paper.

DSHIELD [44] is a system that assigns "suspicion scores" to user sessions based on their distance from legitimate session. While similar in nature to FINELAME's anomaly detection technique, it relies on the operator knowing all the possible classes of requests that the server can process. FINELAME anomaly detection engine learns on legitimate requests so that it does not depend on *a priori* knowledge of execution paths or vulnerabilities.

BreakApp [57] is a module-level compartmentalization system that attempts to defend against DoS attacks, among other threats stemming from third-party modules. While BreakApp's capabilities increase with more and smaller modules, FINELAME works even with monolithic applications entirely developed as a single module. BreakApp's mitigation uses simple queue metrics (*i.e.*, queue length at the module boundary *vs.* replica budget), whose cut-off parameters are statically provided by the programmer; FINELAME uses a more advanced learning model, which parameters are adjusted at runtime.

Rampart [36] focuses on asymmetric application-level CPU attacks in the context of PHP. It estimates the distribution of a PHP application function's CPU consumption, and periodically evaluates running requests to assess the likelihood they are malicious. It then builds filters to probabilistically drop offenders—repeated offenders increase their probability of being filtered out. While FINELAME profiles legitimate requests resource consumption, it is not limited to CPU-based attacks. It also works with applications with components built with many different languages.

**In-kernel Detection** Recent work has shown good results for mitigating ADoS attacks by exploiting low level system metrics. Radmin [16] and its successor Cogo [17] train Probabilistic Finite Automatas (PFAs) offline for each resource of a process they want to monitor, then perform anomaly detection by evaluating how likely the process' transition in the resource space is. Training the PFAs requires days in Radmin, and minutes in Cogo, while FINELAME can train accurate models in seconds or hundreds of microseconds. We expect this capability to be helpful in production systems where the model has to be updated, *e.g.*, to account for changes in an application's component. In addition, Cogo reports detection time in the order of seconds, while FINELAME's inline detection operates at the scale of the request's complexity—milliseconds in our experiments. Lastly, Radmin/Cogo operate at the granularity of processes/connections. FINELAME assumes a worst-case threat model where malicious requests are sent sporadically

by compromised clients, and thus operate at this granularity. Per-request detection has the added benefit to enable precise root cause analysis, further enhancing the practicality of FINELAME.

**Programmer Annotations** Prior work proposes an annotation toolkit that programmers can use in their code to specify resource consumption requirements [42]. The framework detects connections that violate the provided specification (and then attempts to mitigate by rate limiting or dropping them). Unfortunately, it requires knowledge of the application internals. Worse even, it expects developers to understand the program's expected resource consumption quite accurately. Moreover, such a hard cut does not distinguish between occasional consumption that is slightly above limits and true attackers.

**Prevention-as-a-Service** A recent vein of work proposed "Attack prevention as a Service", where security appliances are automatically provisioned at strategic locations in the network [19, 37]. Those techniques are largely dependent on attack detection (to which they do not provide a solution), and thus are orthogonal to our platform, which operates directly at the victim's endpoint.

**Performance anomaly detection** ADoS attacks are a subset of the broader topic of performance degradation, a topic that has been extensively studied. Magpie [9] instruments an application to collect events from the entire stack and obtain request profiles *post-mortem*. X-trace [21] is a tracing framework that preserves causal relationship between events, and allow the offline reconstruction of request trees. X-ray [8] builds on taint-tracking to provide record and replay system to summarize the performance of application events offline. One of FINELAME's key difference with those systems is its lightweight in-flight profiling technique, which allows us to perform anomaly detection while the request is still in the system. Retro [33] provides a tracing architecture for multi-tenant systems that enables the implementation of resource management policies. While its architecture is similar to FINELAME's, its focus is on performance degradation caused by competing workloads, rather than the detection of degradation within a single application.

While the impact can be similar, we note that for ADoS attacks, in-flight request tracking is critical to timely detection and mitigation.

## 8 Conclusion

In this paper, we describe and evaluate FINELAME, a novel fine-grained application-level DoS detection framework. FINELAME is designed for interaction with modern distributed applications, operates orders of magnitude faster than previous techniques, and is able to detect yet-unseen attacks on an application. FINELAME is enabled by recent advances in the Linux kernel, and bridges the gap between application-layer semantic and low-level resource allocation sub-systems. It is a first step toward deploying complex machine learning applications for fine grained services, in an era where the size of services is shrinking (micro/pico-services).

## 9 Acknowledgments

We would like to thank our shepherd, Mike Reiter, and the anonymous ATC reviewers for their useful feedback. This material is based upon work supported in parts by the Defense Advanced Research Projects Agency (DARPA) under Contracts No. HR0011-16-C-0056 and No. HR001117C0047, and NSF grants CNS-1513687, CNS-1513679, CNS-1563873, CNS-1703936 and CNS-1750158. 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] Apache HTTP server project. https://httpd.apache.org/.

[2] bcc on GitHub. https://github.com/iovisor/bcc.

[3] Common vulnerabilities and exposures (see cve-2003-1564). http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-1564.

[4] Node.js server project. https://nodejs.org/en/.

[5] Regular expression denial of service - ReDoS. https://www.owasp.org/index.php/Regular_expression_Denial_of_Service_-_ReDoS.

[6] Tsung. http://tsung.erlang-projects.org/.

[7] OpenTracing API. Consistent, expressive, vendor-neutral apis for distributed tracing and context propagation.

[8] Mona Attariyan, Michael Chow, and Jason Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 307–320, Berkeley, CA, USA, 2012. USENIX Association.

[9] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, HOTOS'03, pages 15–15, Berkeley, CA, USA, 2003. USENIX Association.

[10] Cristina Basescu, Raphael M Reischuk, Pawel Szalachowski, Adrian Perrig, Yao Zhang, Hsu-Chun Hsiao, Ayumu Kubota, and Jumpei Urakawa. Sibra: Scalable internet bandwidth reservation architecture. *arXiv preprint arXiv:1510.02696*, 2015.

[11] Ang Chen, Akshay Sriraman, Tavish Vaidya, Yuankai Zhang, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. Dispersing asymmetric ddos attacks with splitstack. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 197–203, New York, NY, USA, 2016. ACM.

[12] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[13] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (redos) in practice: An empirical study at the ecosystem scale. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 246–256, New York, NY, USA, 2018. ACM.

[14] James C. Davis, Eric R. Williamson, and Dongyoon Lee. A sense of time for javascript and node.js: First-class timeouts as a cure for event handler poisoning. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 343–359, Berkeley, CA, USA, 2018. USENIX Association.

[15] Henri Maxime Demoulin, Tavish Vaidya, Isaac Pedisich, Bob DiMaiolo, Jingyu Qian, Chirag Shah, Yuankai Zhang, Ang Chen, Andreas Haeberlen, Boon Thau Loo, Linh Thi Xuan Phan, Micah Sherr, Clay Shields, and Wenchao Zhou. Dedos: Defusing dos with dispersion oriented software. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 712–722, New York, NY, USA, 2018. ACM.

[16] Mohamed Elsabagh, Daniel Barbará, Dan Fleck, and Angelos Stavrou. Radmin: Early detection of application-level resource exhaustion and starvation attacks. In *Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404*, RAID 2015, pages 515–537, New York, NY, USA, 2015. Springer-Verlag New York, Inc.

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

[18] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, pages 226–231. AAAI Press, 1996.

[19] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and elastic ddos defense. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 817–832, Berkeley, CA, USA, 2015. USENIX Association.

[20] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[21] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 20–20, Berkeley, CA, USA, 2007. USENIX Association.

[22] Junhao Gan and Yufei Tao. Dbscan revisited: Mis-claim, un-fixability, and approximation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 519–530, New York, NY, USA, 2015. ACM.

[23] Gkbrk. SlowLoris attack tool. https://github.com/gkbrk/slowloris.

[24] Dan Goodin. US service provider survives the biggest recorded ddos in history, 2018.

[25] Hossein Hadian Jazi, Hugo Gonzalez, Natalia Stakhanova, and Ali A. Ghorbani. Detecting http-based application layer dos attacks on web servers in the presence of sampling. *Comput. Netw.*, 121(C):25–36, July 2017.

[26] Srikanth Kandula, Dina Katabi, Matthias Jacob, and Arthur Berger. Botz-4-sale: Surviving organized ddos attacks that mimic flash crowds. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 287–300, Berkeley, CA, USA, 2005. USENIX Association.

[27] Min Suk Kang and Virgil D. Gligor. Routing bottlenecks in the internet: Causes, exploits, and countermeasures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 321–333, New York, NY, USA, 2014. ACM.

[28] Min Suk Kang, Soo Bum Lee, and Virgil D. Gligor. The crossfire attack. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, SP '13, pages 127–141, Washington, DC, USA, 2013. IEEE Computer Society.

[29] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[30] libuv. A multi-platform support library with a focus on asynchronous i/o.

[31] Xin Liu, Xiaowei Yang, and Yong Xia. Netfence: preventing internet denial of service from inside out. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010.

[32] SS Jeremy Long. Owasp dependency check, 2015. Accessed: 2017-06-11.

[33] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. Retro: Targeted resource management in multi-tenant distributed systems. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 589–603, Berkeley, CA, USA, 2015. USENIX Association.

[34] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Trans. Comput. Syst.*, 35(4):11:1–11:28, December 2018.

[35] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2–2, Berkeley, CA, USA, 1993. USENIX Association.

[36] Wei Meng, Chenxiong Qian, Shuang Hao, Kevin Borgolte, Giovanni Vigna, Christopher Kruegel, and Wenke Lee. Rampart: Protecting web applications from cpu-exhaustion denial-of-service attacks. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 393–410, Berkeley, CA, USA, 2018. USENIX Association.

[37] Rui Miao, Minlan Yu, and Navendu Jain. Nimbus: Cloud-scale attack detection and mitigation. *SIGCOMM Comput. Commun. Rev.*, 44(4):121–122, August 2014.

[38] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[39] Tongguang Ni, Xiaoqing Gu, Hongyuan Wang, and Yu Li. Real-time detection of application-layer ddos attack using time series analysis. *J. Control Sci. Eng.*, 2013:4:4–4:4, January 2013.

[40] Tao Peng, Christopher Leckie, and Kotagiri Ramamohanarao. Survey of network-based defense mechanisms countering the dos and ddos problems. *ACM Comput. Surv.*, 39(1), April 2007.

[41] Open Web Application Security Project. Owasp top ten project'17, 2018. Accessed: 2018-09-27.

[42] Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build dos-resistant software. *SIGOPS Oper. Syst. Rev.*, 36(SI):45–60, December 2002.

[43] Steve Ranger. Github hit with the largest ddos attack ever seen, 2018.

[44] Supranamaya Ranjan, Ram Swaminathan, Mustafa Uysal, Antonio Nucci, and Edward Knightly. Ddos-shield: Ddos-resilient scheduling to counter application layer attacks. *IEEE/ACM Trans. Netw.*, 17(1):26–39, February 2009.

[45] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 9–9, Berkeley, CA, USA, 2006. USENIX Association.

[46] David Senecal. Slow DoS on the rise. https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html.

[47] N. Seriot. http://seriot.ch/parsing_json.php, 2016.

[48] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. Rescue: Crafting regular expression dos attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 225–235, New York, NY, USA, 2018. ACM.

[49] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Technical report, Google, Inc, 2010.

[50] Snyk. Find, fix and monitor for known vulnerabilities in node.js and ruby packages, 2016.

[51] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *Proceedings of the 27th USENIX Conference on Security Symposium*, SEC'18, pages 361–376, Berkeley, CA, USA, 2018. USENIX Association.

[52] Cristian-Alexandru Staicu, Michael Pradel, and Benjamin Livshits. Synode: Understanding and automatically preventing injection attacks on node.js. In *Networked and Distributed Systems Security*, NDSS'18, 2018.

[53] Michael Stepankin. [demo.paypal.com] node.js code injection (rce), 2016. Accessed: 2018-10-05.

[54] Ahren Studer and Adrian Perrig. The coremelt attack. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 37–52, Berlin, Heidelberg, 2009. Springer-Verlag.

[55] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 3–14. ACM, 2006.

[56] Vern Paxson Steven McCanne Van Jacobson, Sally Floyd. Tcpdump, a command-line packet analyzer.

[57] Nikos Vasilakis, Ben Karel, Nick Roessler, Nathan Dautenhahn, André DeHon, and Jonathan M. Smith. Breakapp: Automated, flexible application compartmentalization. In *Proceedings of the 25th Networked and Distributed Systems Security Symposium*, NDSS'18, 2018.

[58] Chenxu Wang, Tony TN Miu, Xiapu Luo, and Jinhe Wang. Skyshield: A sketch-based defense system against application layer ddos attacks. *IEEE Transactions on Information Forensics and Security*, 13(3):559–573, 2018.

[59] wikipedia. Wikipedia, the free encyclopedia.

[60] Yang Xu and Yong Liu. Ddos attack detection under sdn context. In *INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications, IEEE*, pages 1–9. IEEE, 2016.

[61] Xiaowei Yang, David Wetherall, and Thomas Anderson. A dos-limiting network architecture. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 241–252, New York, NY, USA, 2005. ACM.

[62] Saman Taghavi Zargar, James Joshi, and David Tipper. A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks. *IEEE communications surveys & tutorials*, 15(4):2046–2069, 2013.

[63] Wei Zhou, Weijia Jia, Sheng Wen, Yang Xiang, and Wanlei Zhou. Detection and defense of application-layer ddos attacks in backbone web traffic. *Future Generation Computer Systems*, 38:36–46, 2014.