Tail Attacks on Web Applications

Huasong Shan†, Qingyang Wang‡, Calton Pu∗
† Computer Science and Engineering Division, Louisiana State University, Baton Rouge, LA, USA
‡ College of Computing, Georgia Institute of Technology, Atlanta, GA, USA
∗ College of Computing, Georgia Institute of Technology, Atlanta, GA, USA

ABSTRACT

As the extension of Distributed Denial-of-Service (DDoS) attacks to application layer in recent years, researchers pay much interest in these new variants due to a low-volume and intermittent pattern with a higher level of stealthiness, invalidating the state-of-the-art DDoS detection/defense mechanisms. We describe a new type of low-volume application layer DDoS attack—Tail Attacks on Web Applications. Such attack exploits a newly identified system vulnerability of n-tier web applications (millibottlenecks with sub-second duration and resource contention with strong dependencies among distributed nodes) with the goal of causing the long-tail latency problem of the target web application (e.g., 95th percentile response time > 1 second) and damaging the long-term business of the service provider, while all the system resources are far from saturation, making it difficult to trace the cause of performance degradation.

We present a modified queueing network model to analyze the impact of our attacks in n-tier architecture systems, and numerically solve the optimal attack parameters. We adopt a feedback control-theoretic (e.g., Kalman filter) framework that allows attackers to fit the dynamics of background requests or system state by dynamically adjusting attack parameters. To evaluate the practicality of such attacks, we conduct extensive validation through not only analytical, numerical, and simulation results but also real cloud production setting experiments via a representative benchmark website equipped with state-of-the-art DDoS defense tools. We further proposed a solution to detect and defense the proposed attacks, involving three stages: fine-grained monitoring, identifying bursts, and blocking bots.

CCS CONCEPTS

• Security and privacy → Distributed systems security; Web application security; Denial-of-service attacks;

KEYWORDS

Long-tail latency; milli-bottleneck; n-tier systems; pulsating attack; web attack; DDoS attack

1 INTRODUCTION

Distributed Denial-of-Service (DDoS) attacks for web applications such as e-commerce are increasing in size, scale and frequency [1, 5]. Akamai’s “quarterly security reports Q4 2016” [1] shows the spotlight on Thanksgiving Attacks, the week of Thanksgiving (involving the three biggest online shopping holidays of the year: Thanksgiving, Black Friday, and Cyber Monday) is one of the busiest times of the year for the retailers in terms of sales and attack traffic. Web applications remain the most vulnerable entrance for any enterprise and organization, so the attackers can exploit them to launch both low-volume and stealthy application-layer DDoS attacks. On the other hand, in web applications especially e-commerce websites, fast response time is critical for service providers’ business. For example, Amazon reported that an every 100ms increase in the page load is correlated to a decrease in sales by 1% [24]; Google requires 99 percentage of its queries to finish within 500ms [10]. Emerging augmented-reality devices (e.g., Google Glass) need the associated web applications with even greater responsiveness in order to guarantee smooth and natural interactivity. In practice, the tail latency, rather than the average latency, is of particular concern for response-time sensitive web-facing applications [6, 10, 11, 18, 40].

In this paper we present a new low-volume application-layer DDoS attack—Tail Attacks, significantly worsening the tail latency on web applications. Web applications typically adopt n-tier architecture in which presentation (e.g., Apache), application processing (e.g., Tomcat), and data management (e.g., MySQL) are physically separated among distributed nodes. Previous research on performance bottlenecks in n-tier systems [41–43] shows that very short bottlenecks (VSBs) or millibottlenecks (with sub-second duration) with dependencies among distributed nodes not only cause queuing delay in local tier, but also cause significant queuing delay in upstream tiers in the invocation chain, which will eventually cause the long-tail latency problem of the target system (e.g., 95th percentile response time > 1 second). More importantly, this phenomenon usually starts to appear under moderate average resource utilization (e.g., 50%) of all participating nodes, making it difficult to trace the cause of performance degradation. In the scenario of Tail Attacks, an attacker sends intermittent bursts of legitimate HTTP requests to the target web system, with the purpose of triggering millibottlenecks and cross-tier queue overflow, creating “Unsaturated DoS” and the long-tail latency problem, where denial of service can be successful for short periods of time (usually tens or hundreds of milliseconds), which will eventually damage the target website’s reputation and business in the long term.

The study of Tail Attacks complements previous research on low-rate network-layer DDoS attacks [12, 14, 21, 22, 25, 27, 39],

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.

CSC ’17, October 30–November 3, 2017, Dallas, TX, USA
© 2017 Association for Computing Machinery.
ACM ISBN 978-1-4503-4946-8/17/10...
https://doi.org/10.1145/3133956.3133968
low-volume application-layer DoS Attacks [8, 28], and flash crowds (usually tens of seconds or minutes) [19, 38, 44] which refer to the situation when thousands of legitimate users suddenly start to visit a website during tens of seconds or minutes due to a flash event (e.g., during the week of Thanksgiving). The uniqueness of Tail Attacks from previous research is that Tail Attacks aim to create very short (hundreds of milliseconds) resource contention (e.g., CPU or disk I/O) with dependencies among distributed nodes, while giving an “Unsaturated illusion” for the state-of-the-art IDS/IPS tools leading to a higher level of stealthiness.

The most challenging task for launching an effective Tail Attack is to understand the triggering conditions of millibottlenecks inside the target web system, and quantify their long-term damages on the overall system performance. To thoroughly comprehend the attack scenario, we exploit the traditional queueing network theory to model the n-tier system, and analyze the impact of our attacks to the end-users and the systems through two new proposed metrics: damage length during which the new coming requests can be dropped with highly probability, and millibottleneck length during which the bottleneck resources sustain saturation. To fit the dynamics of background requests and system state, we develop a feedback control framework. Given the guide of the proposed model and the implementation based on the feedback control algorithm, we can effectively control the attacks, and find that our attacks can not only achieve high attack efficiency, but also escape the detection mechanisms based on human-behavior models, which further increases the stealthiness of the attack.

In brief, this work makes the following contributions:

- Proposing Tail Attacks by exploiting resource contention with dependencies among distributed nodes, that can significantly cause the long-tail latency problem in web applications while the servers are far from saturation.
- Modeling the impact of our attacks on n-tier systems based on queueing network theory, which can effectively guide our attacks in an even stealthy way.
- Adopting a feedback control-theoretic (e.g., Kalman filter) framework that allows our attacks to fit the dynamics of background requests and system state by dynamically tuning the optimal attack parameters.
- Validating the practicality of our attacks through not only analytical, numerical, and simulation results but also real experimental results of a representative benchmark website equipped with state-of-the-art DDoS defense tools in real cloud production settings.
- Presenting a conceptual solution to detect and defense the proposed attacks, involving three stages: fine-grained monitoring, identifying bursts, and blocking bots.

We outline the rest of this paper as follows. Section 2 describes an attack scenario and the practical impact of Tail Attacks in real cloud production settings. Section 3 models our attack scenarios in an n-tier system using queueing network theory, and provides an effective approach to solve the potential optimal attack parameters numerically. Further, we evaluate the attack analytical model in JMT [7] simulator environment and suggest several guidelines to choose the optimal attack parameters in more complex cases (e.g., the competition for free slots of a queue between attack requests and normal requests, overloaded attack requests can be also dropped by the front-tier server). Section 4 describes our concrete implementation to launch Tail Attacks in real web applications. We adopt Kalman filter, a feedback control-theoretic tool, to automatically adjust the optimal attack parameters fitting the dynamics of target system state (e.g., dataset size change) and background workload. Section 5 shows our attack results of RUBBoS [34] benchmark website we have conducted in real cloud production settings, which further confirm the effectiveness and stealthiness of the proposed attacks, and the practicality of the control framework in more practical Web environments. Section 6 provides a “tit-for-tat” strategy to detect and defend our attacks targeting the unique scenario and feature of the proposed attack. Section 7 discusses some additional factors that may impact the effectiveness of Tail attacks. Section 8 presents the related work and Section 9 concludes the paper.

2 SCENARIO AND MOTIVATIONS

**Attack Scenario.** Consider a scenario of a Tail attack in an n-tier system in Figure 1. The detailed model analysis and experimental
data of Tail Attacks are in the following sections. By alternating short "ON" and long "OFF" attack burst, an attacker guarantees the attack both harmful and stealthy. Short "ON" attack burst is typically on the order of milliseconds. The following sequence of causal events will lead to the long-tail problem at moderate average utilization levels during the course of Tail Attacks. (Event1) The attackers send intermittent bursts of attack but legitimate HTTP requests to the target system during the "ON" burst period; each burst of attack requests are sent out within a very short time period (e.g., 50ms). (Event2) Resource millibottlenecks occur in some node, for example, CPU or I/O saturates for a fraction of a second due to the burst of attack requests. (Event3) A millibottleneck stops the saturated tier processing for a short time (order of milliseconds), leading to fill up the message queues and thread pools of the bottleneck tier, and quickly propagating the queue overflow to all the upstream tiers of the n-tier system as shown in Figure 1b. (Event4) The further incoming packets of new requests are dropped by the front tier server once all the threads are busy and TCP buffer overflows in the front tier. (Event5) On the end-user side, the dropped packets are retransmitted several seconds later due to TCP congestion control (minimum TCP retransmission time-out is 1 second [16]), the end users with the requests encountering TCP retransmissions perceive very long response time (order of seconds).

Long "OFF" attack burst is typically on the order of seconds, in which the target system can cool down, clearing up the queued requests and returning back to a low occupied state shown in Figure 1a. Unlike the traditional flooding DDoS attacks which aim to bring down the system, our attack aims to degrade the quality of service by causing the long-tail latency problem for some legitimate users while keeping the attack highly stealthy. The alternating short "ON" and long "OFF" attack burst can effectively balance the trade-off between attack damage and elusiveness.

**Measured Long-Tail Latency.** Table 1 shows the impact of Tail Attacks through concrete benchmark web application with real production settings deployed in the most popular two commercial cloud platforms (Amazon EC2 [2], Microsoft Azure [29]) and one academic cloud platform (NSF Cloudlab[31]). We use a notation CloudPlatform-ServerTiers-BaselineWorkload to denote the cloud platform, the configuration of the n-tier system, and the background workload. For Server Tiers, We use a four-digit (or three-digit) notation #W#A#L#D to denote the number of web servers, application servers, load-balance servers (may not be configured), and database servers. More experimental details are available in Section 5.1. Table 1 compares the tail latency of the target system under attack and without attack, indicating the significant long tail latency problem under attack. Such long tail latency problem (e.g., 95th percentile response time > 1 second) is considered as severe performance degradation by most modern e-commerce web applications (e.g., Amazon) [6, 10, 11, 18, 24]. At the same time, the average response time is still in acceptable range under attack, making the illusion of "business as usual" for system administrators.

### Table 1: The measured long-tail latency under Tail Attacks in real cloud production setting.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2-111-2K</td>
<td>255</td>
<td>1347</td>
<td>267</td>
<td>1538</td>
<td>308</td>
<td>1732</td>
<td>192</td>
<td>273</td>
</tr>
<tr>
<td>EC2-1212-4K</td>
<td>247</td>
<td>1139</td>
<td>282</td>
<td>1341</td>
<td>328</td>
<td>1683</td>
<td>170</td>
<td>218</td>
</tr>
<tr>
<td>EC2-1414-6K</td>
<td>263</td>
<td>1085</td>
<td>270</td>
<td>1285</td>
<td>312</td>
<td>1628</td>
<td>160</td>
<td>206</td>
</tr>
<tr>
<td>Azure-111-1K</td>
<td>251</td>
<td>1153</td>
<td>274</td>
<td>1295</td>
<td>295</td>
<td>1821</td>
<td>176</td>
<td>290</td>
</tr>
<tr>
<td>Azure-1212-2K</td>
<td>254</td>
<td>1090</td>
<td>278</td>
<td>1284</td>
<td>295</td>
<td>1507</td>
<td>177</td>
<td>221</td>
</tr>
<tr>
<td>Azure-1414-3K</td>
<td>264</td>
<td>1090</td>
<td>297</td>
<td>1314</td>
<td>325</td>
<td>1510</td>
<td>181</td>
<td>242</td>
</tr>
<tr>
<td>NSF-111-1K</td>
<td>141</td>
<td>1217</td>
<td>151</td>
<td>2262</td>
<td>159</td>
<td>7473</td>
<td>90</td>
<td>327</td>
</tr>
<tr>
<td>NSF-1212-2K</td>
<td>128</td>
<td>1103</td>
<td>143</td>
<td>1502</td>
<td>167</td>
<td>7016</td>
<td>88</td>
<td>309</td>
</tr>
<tr>
<td>NSF-1414-3K</td>
<td>118</td>
<td>1014</td>
<td>122</td>
<td>1413</td>
<td>127</td>
<td>3152</td>
<td>71</td>
<td>268</td>
</tr>
</tbody>
</table>

W/o att: Without attacks, Att.: under Tail Attacks, RT: response time (ms)

### Table 2: Model parameters

<table>
<thead>
<tr>
<th>Param.</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_i</td>
<td>the queue size for the i-th tier</td>
</tr>
<tr>
<td>C_{LA}</td>
<td>the capacity serving attack requests for the i-th tier</td>
</tr>
<tr>
<td>C_{LL}</td>
<td>the capacity serving legitimate requests for the i-th tier</td>
</tr>
<tr>
<td>\lambda_i</td>
<td>the legitimate request rate terminating in the i-th tier</td>
</tr>
<tr>
<td>B</td>
<td>the attack request rate during a burst</td>
</tr>
<tr>
<td>L</td>
<td>the burst length during a burst</td>
</tr>
<tr>
<td>V</td>
<td>the burst volume during a burst</td>
</tr>
<tr>
<td>T</td>
<td>the interval between every two consecutive bursts</td>
</tr>
<tr>
<td>l_i</td>
<td>the time to fill up the queue of the i-th tier</td>
</tr>
<tr>
<td>P_D</td>
<td>the period of the requests dropped during a burst</td>
</tr>
<tr>
<td>P_{MB}</td>
<td>the period of a millibottleneck during a burst</td>
</tr>
<tr>
<td>\rho(L)</td>
<td>the average drop ratio during a burst</td>
</tr>
<tr>
<td>\rho(T)</td>
<td>the drop ratio during the course of an attack</td>
</tr>
</tbody>
</table>

### 3 TAIL ATTACKS MODELING

In this section, we provide a simple model to analyze the impact of our attacks to the end-users and the victim n-tier system. Based-on the simplified model we introduce an effective approach of getting the potential optimized attack parameters to achieve our attack goal. Finally, we evaluate the model via simulation experiments and suggest several approaches to tune the attack parameters.

#### 3.1 Model

Queueing network models are commonly used to analyze the performance problems in complex computer systems [23], especially for performance sizing and capacity provision. Here, we use a well-tuned queueing network to model n-tier systems, and analyze the sequence of causal events and the impact in the context of Tail Attacks shown in Figure 1. Table 2 summarizes the notation and description of the parameters used in our model. The basic attack pattern (see Event1 in Section 2) shown in Figure 1 is that during the "ON" burst period (L) the attackers send a burst of attack requests with the rate (B), after the "OFF" burst period (T-L) they send another burst again, and repeat this process during the course of a Tail attack. If all the attack requests will not be dropped by the
target system (more complex case will be discussed in Section 3.3), we can calculate the attack volume during a burst by:

\[ V = B \times L \]  \hspace{1cm} (1)

We assume that the external burst of legitimate HTTP requests (Event1 in Section 2) can cause sudden jump of resource demand flowing into the target system and cause millibottlenecks (Event2 in Section 2) in the weakest point of the system [12]. In our model analysis, we assume that the n-th tier is the bottleneck tier. For example, the bottleneck typically occurs in the database tier (the n-th tier) in web applications due to the high resource consumption of database operations.

Due to the inter-tier dependency (call/response RPC style communication) in the n-tier system, one queued request in a downstream server holds a thread in every upstream server. Thus, the system administrator typically configures the queue size of upstream tiers bigger than the queue size of downstream tiers. In this case, millibottlenecks (Event2 in Section 2) caused by overloaded attack bursts can lead to cross tier queue overflow from downstream tiers to upstream tiers (Event3 in Section 2) due to the strong dependency among n-tier nodes. If the queue size satisfies

(C1) \( Q_i > Q_2 > ... > Q_{n-1} > Q_n \)

and the burst rate satisfies

(C2) \( \lambda_n + B > C_n \)

for all \( i = 1, ..., n \), then the time needed to fill up the queue for the n-th server is approximately

\[ l_n = \frac{Q_n}{(\lambda_n + B - C_{n,A})} \]  \hspace{1cm} (2)

\[ l_{n-1} = \frac{(Q_{n-1} - Q_n)}{(\lambda_{n-1} + \lambda_n + B - C_{n,A})} \]  \hspace{1cm} (3)

\[ l_1 = \frac{(Q_2 - Q_1)}{(\sum_{i=1}^{n} \lambda_i + B - C_{n,A})} \]  \hspace{1cm} (4)

When millibottlenecks occur in the n-th server, firstly the queue in the n-th tier is overflown during \( l_n \), which equals the available queue size of the n-th tier divided by the newly-occupied rate for the queue of the n-th tier in Equation 2. The available queue size equals the queue size of each tier subtracting the queue size of its directly upstream tier. The newly-occupied rate equals the incoming rate of each tier subtracting the outgoing rate of the total system \( (C_{n,A}) \), the incoming rate of the n-th tier includes the requests going through the n-th tier and terminating in the n-th tier (B for the attack requests and \( \lambda_n \) for the normal requests). We carefully choose the attack requests [40] guaranteeing the attack requests go through every tier and terminate in the last bottleneck tier (detailed implementation in Section 4.2). Equation 3 represents the time to fill up the queue in the n-1-th tier. Because one queued request in a downstream server holds a position in the queue of every upstream server, after the n-th tier is full, the available queue size of the n-1-th tier should be \( Q_{n-1} - Q_n \). All the requests arriving to a downstream tier need to go through every upstream tier, thus the incoming rate of the n-1-th tier includes the request rate of terminating in the n-1-th tier \( (\lambda_{n-1}) \) and the request rate of going through the n-1-th tier \( (\lambda_n + B) \). Similarly, we can calculate the time to fill up every queue in the n-tier system during the process of propagating the queue overflowed. Finally, the required time to overflow all the queues in the n-tier system is the sum of \( l_i \). Here, \( Q, C, \) and \( \lambda \) are constants. \( l_i \) is a function of \( B \).

Once all the queues are overflown (Event3 in Section 2) in the n-tier system, the new incoming requests may be dropped by the front tier (Event4 in Section 2). We term the period of the requests dropped during a burst as damage length. If the attackers continue to send attack requests to the system with overflown queues, and we assume that attack requests can always occupy the free position of the queue in the system (more complicated case will be discussed in Section 3.3), then we can approximately infer damage length by:

\[ P_D = L - \sum_{i=1}^{n} l_i \]  \hspace{1cm} (5)

Further, the end-users with the dropped requests perceive very long response time (Event5 in Section 2), leading to the long-tail latency problem caused by our attacks (Event1 in Section 2), which can be approximately estimated as follows,

\[ \rho(T) = \frac{P_D}{L} \]  \hspace{1cm} (6)

During a burst, the servers need to provide all the required computing resources (including bottleneck resources) to serve both attack requests and normal requests. We term the period of a millibottleneck during a burst as millibottleneck length, during which bottleneck resources sustain saturation. Thus, millibottleneck length should involve the resource consumption for both attack and normal requests during a burst. Equation (7) represents millibottleneck length derived through the geometric progression in mathematics (more detailed derivation of this equation in Appendix A).

\[ P_{MB} = V \times \left( \frac{1}{C_{n,A}} \times \frac{1}{(1 - (\lambda_n \times \frac{1}{C_{n,L}}))} \right) \]  \hspace{1cm} (7)

where \( 1/C_{n,A} \) and \( 1/C_{n,L} \) are the service time for attack and normal requests in the bottleneck tier, respectively.

### 3.2 Numerically Solve Attack Parameters

Based on the model, we can infer the damage and elusiveness of our attacks through damage length and millibottleneck length. Further, if we assign the attack goal and know system parameters, we can calculate the optimal attack parameters mathematically.

**Constant Parameters Estimation.** To get some reasonable constant parameters \( (\lambda, C_{L,A}, C_{L,N}, Q) \) in the model, we estimate these constants via profiling the service time of each type of request of each component tier in the benchmark web-site RUBBoS [34] (more details in Section 5.1), the capacity of each tier \( C_i \) can be calculated from the service time. We choose heavy requests (e.g., long service time by consuming more system bottleneck resource, detailed explanation in Section 4.2) as attack requests. Table 3 lists a group of reasonable values of the constants for our model profiled in RUBBoS. During the profiling, we choose 2000 legitimate users with 7-second think time as our baseline experiment. All the transactions supported by RUBBoS are terminated in MySQL, each transaction follows a static page-load terminated in Apache, and no transaction terminates in Tomcat. Thus, the request rate of each tier \( \lambda_i \) is 280, 0, and 280, respectively. We set the queue size of each server \( Q_i \) satisfying the condition \( CI \) in Equation (2).
Suppose that we set our attack goal as $P_{MB}$ and above the solid line $P_D$. Then there exists an overlapped feasible zone below the dash line $P_{MB}$ and above the solid line $P_D$. As $Q$ increases, the feasible zone narrows down until no solution when $Q = 130$ (red line).

Table 3: Constant parameters estimation profiled in RUB-BoS experiment with 2000 concurrent users.

<table>
<thead>
<tr>
<th>Server</th>
<th>Tier</th>
<th>$\lambda$</th>
<th>$C_{n,A}$</th>
<th>$C_{n,L}$</th>
<th>$Q_{c}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache</td>
<td>1</td>
<td>280</td>
<td>3443</td>
<td>3657</td>
<td>55</td>
</tr>
<tr>
<td>Tomcat</td>
<td>2</td>
<td>0</td>
<td>1300</td>
<td>1987</td>
<td>25</td>
</tr>
<tr>
<td>MySQL</td>
<td>3</td>
<td>280</td>
<td>280</td>
<td>725</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 2: Numerically solve the optimal attack parameters. Solid line depicts damage length, the target zone is above the solid line; and dash line depicts millibottleneck length, the target zone is below the dash line.

**Attack Goal and Solver.** Suppose that we set our attack goal as 95th percentile response time longer than 1 second which is a severe long-tail latency problem for most e-commerce websites [6, 10, 11, 18], and the duration of a millibottleneck less than 0.5 seconds in the bottleneck tier such that the average utilization can be at moderate level (e.g., 50-60%) to bypass the defense mechanisms. If we assume the burst interval $T$ is 2 seconds, then the input to the model can be two inequations: damage length $P_D$ is bigger than 0.1 seconds and millibottleneck length $P_{MB}$ is less than 0.5 seconds. Further, we turn these two inequations to $L$ as a function of $B$, the others parameters are all constants (Inequation (8) and (9)).

$$L > \sum_{i=1}^{n} l_i + 0.1 = \frac{Q_n}{(\lambda_n + B - C_{n,A})} + \frac{Q_{n-1} - Q_n}{(\lambda_{n-1} + \lambda_n + B - C_{n,A})} + \frac{Q_1 - Q_2}{(\sum_{i=1}^{n} \lambda_i + B - C_{n,A})} + 0.1$$

(8)

$$L \leq 0.5 + (1 - \lambda_n) * \frac{1}{C_{n,L}} * \frac{C_{n,A}}{B}$$

(9)

Thus, the problem of selecting a set of optimal attack parameters $(B, L, V, T)$ can become a nonlinear optimization problem. Although nonlinear optimization problem is hard to solve since there exist multiple feasible regions and multiple locally optimal points in those regions, we can add more constraints to narrow the range of feasible regions. For instance, the burst length obviously should be less than target millibottleneck length(e.g., $L < 0.5$).

Substituting the constant parameters in Inequation (8) and (9), we can get an unique feasible region as the potential attack parameters shown in Figure 2a. In real cloud production settings, the queue size must be diverse according to the capacity of the websites. In Figure 2b, we can see that as the queue of the front tier (e.g., Apache) increases from 30 to 80, the feasible region reduces; when it increases to 130 (the red line), the two inequations do not overlap, which implies that there is no solution to satisfy our predefined attack goal. The fundamental reason is that our attack goal is too strict, which seems to be an impossible mission. Note that when the queue of the front tier is 80, in the strictest attack target cases ($P_D > 0.2$ seconds, the red line in Figure 2c; or $P_{MB} < 0.3$ seconds, the red line in Figure 2d), there is also no solution that can solve the attack parameters of our attacks. We will further discuss how to deal with the non-solution cases in Section 4.1.

### 3.3 Simulation Experiments

The numerical solver in the previous section does not consider many aspects of the real system (e.g., the competition of the free position in the queue between attack requests and normal requests, overloaded attack requests can be dropped, etc.). To further validate the simple model, we present results from Java Model Tools(JMT) [7] in which such limitations are absent. JMT is an open source suite for modeling Queueing Network computer systems. It is widely used in the research area of performance evaluation, capacity planning in n-tier systems. Thus, it is a natural choice to evaluate the impact of our attacks in n-tier systems. We modify the JMT code and simulate the bursts of attack requests for our attacks with the configurable attack parameters in our model.

Given the proposed model and the idea of solving the nonlinear optimization problem, we can get the feasible region of attack parameters. We initialize the parameters in JMT similar to the setting of our numerical solver, and choose a potential optimal point (400,0.215) in Figure 2a as our attack parameters, the attack rate $B$ is 400 requests per second and the burst length $L$ is 0.215 seconds, so the attack volume per burst $V$ is 86 (see equation (1)) if all the attack requests will not be dropped by the target system.

**Results in JMT** Figure 3 shows the results of one burst during 1 second time period using fine-grained monitoring (e.g., 50 milliseconds) in JMT experiment. Figure 3a illustrates the process of filling up all the queues in the n-tier system. Note that the queue of MySQL, Tomcat, and Apache is overflow from down-stream tiers.
Due to the existence of the competition between the attackers and the legitimate users, the attackers may fail to get the predefined attack goal (e.g., 95th percentile response time > 1 second) by using the recommended attack parameters of the proposed model. We further investigate how to increase the competition ability of attack requests, and the drop ratio of the requests from the legitimate users during damage length, namely \( \rho(L) \). Finally, the attackers can choose the optimal attack parameters to achieve high damage with low cost and high stealthiness. For simplicity, we assign attack interval \( T \) as fixed value (say, 2 seconds), since our focus on this paper is to investigate how to effectively trigger the millibottlenecks which is predominantly determined by the other three parameters \((B, L, V)\). Due to interdependent relationship of these three parameters in Equation 1, we fix one parameter \((L \text{ or } V)\), then observe the impact with various attack rate \(B\). We still consider the marked potential optimal point \((400, 0.215)\) in Figure 2a as our baseline attack parameters.

First, we fix burst volume \(V\) as 86, and select a set of attack rate (from 300 to 800) to conduct our attack experiments using JMT. Table 4 shows that as the attack rate increases, the drop rate \(\rho(T)\) accordingly increases, which confirms that higher attack request rate, compared to normal request rate, can achieve higher competition ability to seize the available position in the queue.

Next, we fix burst length \(L\) as 0.215 seconds, and select another set of attack rate to conduct our JMT experiments. Figure 4 depicts \(\rho(T)\) and \(\rho(L)\) as a function of the ratio of attack rate and service rate for the bottleneck tier. We mark two vertical lines to split up into three zones with various attack rate \(B\). (1) In zone \(a\), \(B\) is less than \(C_{n,A}\), the drop ratios are all zero, since the attack rate is too low to trigger an effective millibottleneck to lead to cross tier queue overflow in the target system, which violates the condition \(C_2\) in Equation 2. (2) In zone \(b\), \(B\) is bigger than \(C_{n,A}\), the drop ratio increases non-linearly as \(B\) increases. Observe that \(\rho(L)\) of Normal Users is a little bit bigger than \(\rho(L)\) of Attackers, because \(B\) is bigger than \(\lambda_a\). In this case, the requests from the attackers can seize the available position in the queue with a more highly probability than ones from the legitimate users. (3) In zone \(c\), \(B\) is bigger than \(C_{l,A}\), the attack requests are directly dropped by the most front tier, thus the increase of \(B\) does not contribute to the drop ratio of

Figure 3: Results of one burst during 1 second period in JMT experiment

<table>
<thead>
<tr>
<th>Attack Para.</th>
<th>Dropped Req.</th>
<th>Total Req.</th>
<th>(\rho(T))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(B)</td>
<td>(L)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>0.285</td>
<td>1096</td>
<td>55998</td>
</tr>
<tr>
<td>400</td>
<td>0.215</td>
<td>1099</td>
<td>54901</td>
</tr>
<tr>
<td>500</td>
<td>0.173</td>
<td>1705</td>
<td>54292</td>
</tr>
<tr>
<td>600</td>
<td>0.144</td>
<td>2082</td>
<td>53915</td>
</tr>
<tr>
<td>700</td>
<td>0.125</td>
<td>2326</td>
<td>53671</td>
</tr>
<tr>
<td>800</td>
<td>0.108</td>
<td>2399</td>
<td>53598</td>
</tr>
</tbody>
</table>

Table 4: Fix burst volume \(V=86\). Bigger \(B\) Higher \(\rho(T)\), implying higher competition ability for the burst with larger \(B\).
the legitimate users, it only increases the drop ratio itself. Given this observation, we should choose a moderate $B$ until the attack goal is achieved (e.g., the green horizontal line in Figure 4).

4 TAIL ATTACKS IMPLEMENTATION

4.1 Overview

As mentioned before, the analytical model used in the numerical solver analyzes the simple scenario skipping many aspects of the system reality (e.g., the competition for the free position of the queue between attack requests and normal requests, overloaded attack requests can be dropped, etc.), and the simulation experiments show more complicated cases involving the absent system reality. However, our attacks do not consider more realistic case with dynamics of baseline workload or system state (e.g., dataset size change). For example, the peak workload occurs at approximately 1:00 p.m. during the week of Thanksgiving [1]. A set of effective attack parameters of Tail Attacks may become failed ones over time, either it can not trigger millibottlenecks (not enough attack requests) or it might trigger the defense alarm of the target system (too frequent or massive attack requests). How can we dynamically adjust the attack parameters catering to instant system state and baseline workload is a big challenge for Tail Attacks. The static attack parameters in the dynamical environment may make the attack either invalid like a "mosquito bite" or easily exposed to the detection mechanisms. In this section, we implement a feedback control-theoretic (e.g., Kalman filter) framework that allows attackers to fit the dynamics of background requests or system state by dynamically adjusting the optimal attack parameters in Figure 5.

Via our best practice, we find that the attack rate should not be invariable to maximize the attack effectiveness and stealthiness. [25] suggests a double-rate DoS steam to minimize the attack cost. We design a three-stage transmitting strategy to send one burst: Quickly-Start, Steadily-Hold and Covertly-Retreat. In Quickly-Start stage, the attacker sends the burst of requests at a high rate to quickly fill up all the queues in the n-tier system, heavy requests (detailed explanation in Section 4.2) are preferred because it can consume more bottleneck resources and occupy the queue longer with low cost and high stealthiness, the amount of heavy

\footnote{For e-commerce applications, the baseline workload during the day time is usually significantly higher than that during the mid-night period.}
requests during this stage should be large enough to temporarily saturate the bottleneck resource in the target system [40]. In Steadily-Hold Stage, the attacker should guarantee the queue can be overflown during this stage and attack requests can seize the free position in the queue with highly probability. Heavy requests are not necessary to serve as attack requests during this stage. We prefer light requests to hold on queue, such that in the last Covertly-Retreat stage, the attack requests can quickly and covertly leave the systems. In Covertly-Retreat stage, there is no attack request to be sent out. Figure 6 demonstrates the queue shifts during a three-stage burst. Through the strategy of variable attack rate and various attack requests, we can solve the insolvable cases in Section 3.2 by carefully choosing less heavy requests as attack requests.

Return back to our model in Equation 5, we can see the relationship between $P_D$ and the attack rate $B$ were nonlinear. We can transfer $P_D$ as a function of $V$ and $B$ using equation 1:

$$P_D = \frac{V}{B} - \sum_{i=1}^{n} I_i$$  

(11)

If we fix the attack rate $B$, mathematically, $P_D$ and the attack volume $V$ have a linear relationship; the same as $P_{MB}$ (see Equation 7). These linear relationships provide us with a firm theoretical foundation to dynamically adapt the optimal attack parameters fitting the changes of system state and baseline workload. The overall control algorithm can be described in Algorithm 1.

### 4.2 Estimator

The Estimator, as illustrated in Figure 5, estimates three critical metrics in the control algorithm implementing the proposed model: service time of the requests, damage length $P_D$, and millibottleneck length $P_{MB}$. We use a prober to monitor attacks and infer $P_D$, coordinate and synchronize bots to launch attacks and infer $P_{MB}$.

**Estimating Service Time.** Service time of a HTTP request is the time that the target web system needs to process the request without any queuing delay. It is easy to calculate the end-to-end response time of a request using two time stamp of sending requests and receiving responses [26], we term them start-time and end-time of a HTTP request. The end-to-end response time of a request equals the difference between end-time and start-time. Typically, the end-to-end response time of a HTTP request involves three parts: the network latency between the client and the target web application, the queuing delay in the n-tier system, and the service time of each server. The network latency can be measured using the ping command. When the target system is at low utilization, the queuing effect inside the target system can be ignored. Thus, we can approximately estimate the service time of any HTTP request supported by the target web system as the end-to-end response time subtracting the network latency when the target system is in the time block with a low workload. Since the service time of the estimated request may drift over time (e.g., due to changes in the data selectivity and the network latency variation) in real applications, we measure the service time of a HTTP request multiple times and take the average.

Previous research results [41] show that the predominant part of the service time of a request is spent on the bottleneck resource in the system. We call the requests that heavily consume the bottleneck resource as heavy requests with long service time (e.g., the request querying multi-tables in the database) while those consume no or little bottleneck resource as light requests with short service time (e.g., static requests) [40]. Thus, the prober naturally exploits light requests to monitor the impact of the attacks since it can be more elusive under the radar without causing any alert of the target web

---

**Algorithm 1** Pseudo-code for the control algorithm

```plaintext
1: procedure ADAPTATTACKPARAMETERS
2:   AttackReqST ← EstimateServiceTime
3:   DamLen ← EstimateDamageLenByProber
4:   MBLen ← EstimateMilliBottleneckLenByBots
5:   if DamLen = 0 then
6:      /* can not fill up queue, increase B */
7:      B ← B + stepB
8:   else
9:      gapDamLen ← Abs(DamLen - targetDamLen)
10:     stepV ← gapDamLen/AttackReqST
11:     if DamLen > targetDamLen then
12:        /* reduce damage length by decreasing V */
13:        V ← V - stepV
14:     else if DamLen < targetDamLen then
15:        /* increase damage length by increasing V */
16:        V ← V + stepV
17:     else
18:        /* current values are the optimal parameters */
19:     end if
20:   end if
21:   if MBLen > targetMBLen then
22:      /* set max V */
23:      Vmax ← targetMBLen/AttackReqST
24:      V ← Vmax
25:      /* choose less heavy requests as attack requests */
26:   end if
27: end procedure
```

---

![Queue Usage Graph](image-url)

**Figure 6:** Queue shifts during a three-stage burst (quickly-start, steadily-hold, covertly-retreat)
which usually try to fully saturate the target network bandwidth.

Figure 7: Demonstration of inferring damage length and millibottleneck length by Estimator.

(a) Estimate $P_D$ by start-time of the last dropped probing request subtracting start-time of the 1st dropped probing request during an attack burst.

(b) Infer $P_{MB}$ by end-time of the last non-dropped attack request subtracting end-time of the 1st non-dropped attack request during a interval.

Through profiling and exploiting heavy requests [40], Tail Attacks can transiently saturating the critical bottleneck resource (e.g., database CPU) of the target systems, which can saturate the critical resource of the system with much lower volume (thus less bots are needed) compared to that of traditional flooding DDoS attacks which usually try to fully saturate the target network bandwidth.

Estimating damage length $P_D$. To estimate $P_D$, the prober needs to send probing requests (e.g., light requests) to the target web system at a predefined rate and record start-time and end-time of a probing request. The recommended sending interval of probing requests is less than the target damage length, in the case, any probing request may not miss the period of overflowed queue caused by an attack burst and the prober can sense $P_D$ [22]. Figure 7a illustrates the implementation approach to estimate $P_D$ as start-time of the last dropped probing request subtracting start-time of the first dropped probing request during a burst. Since some probing requests may probably seize the free position in the queue and are not dropped during the damage length, we can calibrate $P_D$ by multiplying $\rho_L$ (see equation 10). Some websites may send some alarm to the users if they send the requests at a very high rate. In this case, the prober can send the probing requests at an acceptable rate for the target web system and estimate the drop ratio during a sampling period, then our control algorithm can exploit drop ratio as the target criterion to dynamically adjust the attack parameters.

Estimating millibottleneck length $P_{MB}$. After sending a burst of attack requests (e.g., heavy requests) to the target web system, the bots can record start-time and end-time of an attack request and estimate $P_{MB}$. We only count the non-dropped attack requests, since the dropped request involves TCP retransmission time-out. There are two options to infer $P_{MB}$. One way is end-time of the last non-dropped attack request subtracting start-time of the first non-dropped attack request during one attack interval. As we mention before, the end-to-end response time of a HTTP request involves three parts: the network latency, the queuing delay, and the service time. In this way, the result overcharges a length of the network latency. The other option of inferring $P_{MB}$ is end-time of the last non-dropped attack request subtracting end-time of the first non-dropped attack request during one attack interval shown in Figure 7b. In this way, it undercharges a length of the service time.

For the concrete environments of a special website, the attacker can choose any approach to estimate millibottleneck length. In our evaluation section, we choose the second one, since the network latency is much longer than the service time in our RUBBoS environment.

4.3 Controller

So far, we discuss many aspects that can influence the effectiveness of our attacks. In the model, the competition for the free slot of the queue between attack requests and normal requests may impact precision of damage length, and the dropped attack requests may decrease millibottleneck length. For Estimator, the network latency variation and the drifted target system state might reduce the accuracy of inferring damage length and millibottleneck length in our implementation. All of aspects lead to the observing and measuring inaccuracy, and result in the invalidation of launching an effective attacks using our control algorithm. To mitigate these negative impacts for our control algorithm, we adapt a popular feedback-based control tool, Kalman filter [20]. On the one hand, it can take past measurements into account for implementing our feedback control algorithm, reducing the impact of the process noise (e.g., baseline workload and system state). On the other hand, it can mitigate the measurement noise due to inaccuracy of the estimator.

Let $z(k)$ be the measurement of $P_D$ in $k$-th burst by Estimator. Since $P_D$ is a linear function of burst volume $V$ (see equ. (11)), we can define $x(k)$ using a linear dynamical system model:

$$SystemDynamics : x(k) = x(k-1) + U(k) + v(k)$$  (12)

$$MeasurementDomain : z(k) = x(k) + w(k)$$  (13)

where the variables $v(k)$ and $w(k)$ are the process noise (e.g., dynamics of baseline workload) and the measurement noise (e.g., imperfect estimation by Estimator), respectively. $U(k)$ is the expected control result, a linear function of burst volume $V$.

Let $\hat{x}(k | k-1)$ be a priori estimate of state parameter $x$ at burst $k$-th given the history of all $k-1$ bursts, and let $\hat{x}(k | k)$ be a posteriori estimate of state parameter $x$ at $k$-th burst. Further, let $P(k | k)$ be a posteriori error covariance matrix which quantifies the accuracy of the estimate $\hat{x}(k | k)$. The Kalman filter executes recursively for each new observation including two phases: Predict in which a priori estimate of state and error matrix are calculated, and Correct in which a posteriori estimate of state and error matrix are refined using the current measurement. The Kalman filter model for our control framework is given by:

Predict (Time Update)

$$\hat{x}(k | k-1) = \hat{x}(k-1 | k-1) + U(k)$$  (14)

$$P(k | k-1) = P(k-1 | k-1) + V(k)$$  (15)

Correct (Measurement Update)

$$Kg(k) = \frac{P(k | k-1)}{P(k | k-1) + W(k)}$$  (16)
$\hat{x}(k | k) = \hat{x}(k | k - 1) + Kg(k)(z(k) - \hat{x}(k | k - 1))$ \hspace{1cm} (17)

$P(k | k) = (1 - Kg(k))P(k | k - 1)$ \hspace{1cm} (18)

where $W(k)$ and $V(k)$ are the covariances of $w(k)$ and $v(k)$, respectively. In practice, we can estimate these two noise covariances using automatic mathematical tools (e.g., autocovariance least-squares method [32]) or manual observation to tune the optimal value. $Kg(k)$ is termed the Kalman gain which represents the confidence index of the new measurement ($z(k)$) over the current estimate ($\hat{x}(k | k - 1)$). If $Kg(k)$ equals 1, it implies that the attacker totally trust the measurement, the effectiveness of adjusting the attack parameters in our attacks is totally depending on the accuracy of the estimated value by Estimator.

Using the Kalman filter, the Controller in Figure 5 can predict the required attack parameters at $k$-th burst given the historical results of all $k-1$ bursts, dynamically command the new parameters to the bots, and automatically and effectively launch Tail Attacks.

5 REAL CLOUD PRODUCTION EVALUATION

5.1 Tail Attacks in Real Production Settings

To evaluate the practicality of our feedback control attack framework in the real cloud production settings, we deploy a representative benchmark website in the most popular two commercial cloud platforms (Amazon EC2, Microsoft Azure) and one academic cloud platform (Cloudlab[31]).

Experiment Methodology. We adopt RUBBoS [34], a representative n-tier web application benchmark modeled after the popular news website Slashdot. We configure RUBBoS using the typical 3-tier or 4-tier architecture. A sample setting EC2-111-2K in Table 5 is 1 Apache web server, 4 Tomcat application servers, 1 C-JDBC clustering load-balance server, 4 MySQL database servers deployed in Amazon EC2, and 6000 concurrent legitimate users. RUBBoS has a workload generator to emulate the behavior of legitimate users to interact with the target benchmark website. Each user follows a Markov chain model to navigate among different webpages, with averagely 7-second think time between every two consecutive requests. Through modifying the RUBBoS source code, we simulate various baseline workload (e.g., variable concurrent users during the experiment). Meanwhile, in our experiments we adopt a centralized strategy to coordinate and synchronize bots [12, 37, 49] (more discussion about distributed bots coordination and synchronization in Section 7). All the bots are in the same location to rule out the impact of the shift of network-latency. We control a small bot farm of 10 machines (one of which serves as a centralized controller), synchronized by NTP services, which can achieve millisecond precision [15]. Each bot uses Apache Bench to send intermittent bursts of attack HTTP requests, commanded by our control framework.

All the VMs we run are 1 vCPU core and 2GB memory, which is the basic computing unit for the commercial cloud providers. We select HDD disk since our experimental workloads are browse-only CPU intensive transactions. We select t2.small instance ($0.023$ per hour) in Amazon EC2 us-east-1a zone, and A1 ($0.024$ per hour) instance in Microsoft Azure East US zone. They have similar prices and hardware configurations. However, the CPU core in EC2 (2.40GHz Intel Xeon E5-2676 v3) is more powerful than the one in Azure (2.10GHz AMD Opteron 4171 HE or 2.40GHz Intel Xeon E5-2673 v3). The worst one is in NSF Cloud (2.10GHz Intel Xeon E5-2450), where we run the VMs in Apt Cluster in the University of Utah’s Downtown Data Center.

For the baseline workload, we have two chosen criteria: the bottleneck resource utilization (e.g., CPU utilization or Network bandwidth) is less than %50 (Column 6 and 8 of Table 5), and no long-tail latency problem exists in without-attacks cases shown in the Table 1. Because EC2 has more powerful CPU that we used in our experiments, it can serve higher baseline workload than the other two. The network overhead can be the bottleneck resource in EC2 platform even though CPU is at low utilizations [17]. In our experiments, MySQL CPU is the bottleneck tier in Azure and NSF Cloud due to the high resource consumption of database operations. We pre-define our attack goal as the 95th percentile response time longer than 1 second and the utilization of the bottleneck resource less than 50%, and fix the attack interval as 2 seconds. Thus, we need to control attack length $P_\text{D}$ longer than 100 milliseconds, millibottleneck length $P_\text{M} < 500$ milliseconds.

Results. Column 2 to 5 of Table 5 show the corresponding model parameters in our real cloud production setting experiments controlled by our attack framework. It clearly shows that our attacks controlled by our algorithm can achieve the predefined targets (5% drop ratio $\rho(T)$, damage length $P_\text{D} < 100ms$, millibottleneck length $P_\text{M} < 500ms$). EC2 has more powerful CPU, so it requires more attack requests per burst $V$ to launch a successful attack. As the servers scale out (from 111 to 1212, 1414), the n-tier system can service more legitimate users, at the same time, in order to launch Tail Attacks it requires higher $V$ due to their higher capacity, which means that bigger websites need larger botnet to attack. Column 7

<table>
<thead>
<tr>
<th>Setting</th>
<th>V</th>
<th>$P_\text{D}$</th>
<th>$\rho(T)$</th>
<th>$P_\text{M}$</th>
<th>CPU (%)</th>
<th>NW (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(%)</td>
<td>(ms)</td>
<td>(%)</td>
<td>(ms)</td>
<td>W/o att.</td>
<td>Att.</td>
</tr>
<tr>
<td>EC2-111-2K</td>
<td>202</td>
<td>96.4</td>
<td>5.64</td>
<td>287</td>
<td>18.8</td>
<td>27.3</td>
</tr>
<tr>
<td>EC2-1212-4K</td>
<td>217</td>
<td>96.9</td>
<td>5.47</td>
<td>297</td>
<td>16.0</td>
<td>20.2</td>
</tr>
<tr>
<td>EC2-1414-6K</td>
<td>234</td>
<td>105.3</td>
<td>5.89</td>
<td>271</td>
<td>15.2</td>
<td>18.1</td>
</tr>
<tr>
<td>Azure-111-1K</td>
<td>113</td>
<td>100.8</td>
<td>5.26</td>
<td>346</td>
<td>41.2</td>
<td>70.6</td>
</tr>
<tr>
<td>Azure-1212-2K</td>
<td>137</td>
<td>99.1</td>
<td>5.36</td>
<td>479</td>
<td>34.6</td>
<td>58.3</td>
</tr>
<tr>
<td>Azure-1414-3K</td>
<td>156</td>
<td>99.9</td>
<td>5.57</td>
<td>408</td>
<td>19.5</td>
<td>25.6</td>
</tr>
<tr>
<td>NSF-111-1K</td>
<td>97</td>
<td>101.0</td>
<td>5.40</td>
<td>453</td>
<td>49.1</td>
<td>76.8</td>
</tr>
<tr>
<td>NSF-1212-2K</td>
<td>112</td>
<td>96.7</td>
<td>5.75</td>
<td>490</td>
<td>48.4</td>
<td>62.4</td>
</tr>
<tr>
<td>NSF-1414-3K</td>
<td>131</td>
<td>99.4</td>
<td>5.22</td>
<td>470</td>
<td>34.6</td>
<td>51.0</td>
</tr>
</tbody>
</table>

W/o att.: Without attacks, Att.: under Tail Attacks.

CPU: MySQL CPU usage, NW: Apache network traffic.

Table 5: The corresponding parameters and bottleneck resource utilization of Tail Attacks in real production settings.
and 9 of Table 5 show the average CPU utilization of MySQL tier and the average network traffic of Apache tier, which are the bottleneck resource in our experimental environment. Under our attacks in the real cloud production settings, the end users encounter the long-tail latency problem (see Table 1). However, all the bottleneck resources are under moderate average resource utilization even in large scale case (e.g., CPU is 25.6% in Azure-1414-3K, and NW is 401MB/s in EC2-1414-6K). These experimental results show that our attacks can cause significant long tail latency problem, while the servers are far from saturation, guaranteeing a high level stealthiness.

Figure 8 further illustrates the 3-minute detailed experimental results with various baseline workload under our attacks launched by our control framework in NSF-1212 setting. Figure 8a shows baseline workload changes from 2000 to 500 concurrent users at the middle point of the experiment (each user has 7-second think time between two webpages). Note that the shift of baseline workload in this case is different from the previous cases in which we scale out the servers. Figure 8d shows the required burst volume dynamically adjusted by our framework, a sudden increase at the middle due to the decrease of baseline workload. Figure 8b and Figure 8c depict damage length and millibottleneck length estimated by the prober and the bots, respectively. The measured average value of two critical metrics of our model are successfully controlled in the target range. Figure 8e and Figure 8f depict the drop ratio during every attack interval as (2.814, 14.423). We can set the minimal boundary of the interval (e.g., rounded to 3, the statistical granularity is seconds in

5.2 Tail Attacks under IDS/IPS systems

Next, in order to evaluate the stealthiness of Tail Attacks under the radar of state-of-the-art IDS/IPS systems, we deploy some popular defense tools in the web tier in our RUBBoS environments to evaluate whether our attacks can be detected by them. **Experiment Methodology.** Typically, the popular solution to mitigate the application layer DDoS attacks is identifying abnormal user-behavior [38, 44–47]. Snort [9] is a signature rules based Open-Source IDS/IPS tool that is widely used in practice for DDoS defense, the users can customize the alert rules by setting reasonable thresholds in the specific systems. We set alert rules following some user-behavior models in Snort to evaluate whether our attacks deviate from the model (judging based on predefined thresholds). In the following experiments, we configure 2000 concurrent legitimate users, and our attack goal is to achieve the 95th percentile response time longer than 1s for these legitimate users. **Results.** To validate the user behavior model, we take request dynamics model [33] as an example. The authors analyzed the distribution of a user’s interaction with a Web server from a collection of Web server logs, categorized four session types (searching, browsing, relaxed and long session), and modeled the features for each type of session based on average pause between sessions. Typically, average inter-request interval for searching and browsing sessions is less than 10 seconds. RUBBoS [34] also models the inter-request interval per session as a Poisson process with the mean as 7, which means an average 7-second think time between two webpages for each session. In this case, we can calculate the 95% confidence interval as (2.814, 14.423). We can set the minimal boundary of the interval (e.g., rounded to 3, the statistical granularity is seconds in
Table 6: The attacker can successfully predict the inter-request model. Less the sampling period, higher alerts from the legitimate users, indicating higher false positive error. However, the attacker can trigger no alert by carefully controlling the request pattern.

<table>
<thead>
<tr>
<th>Alert Rule Parameters</th>
<th>Triggered Alert Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threshold</td>
<td>Sampling Period</td>
</tr>
<tr>
<td>1</td>
<td>3s</td>
</tr>
<tr>
<td>5</td>
<td>15s</td>
</tr>
<tr>
<td>10</td>
<td>30s</td>
</tr>
<tr>
<td>15</td>
<td>45s</td>
</tr>
<tr>
<td>20</td>
<td>60s</td>
</tr>
<tr>
<td>50</td>
<td>150s</td>
</tr>
<tr>
<td>100</td>
<td>300s</td>
</tr>
</tbody>
</table>

Table 7: The attacker conservatively predicts the inter-request model. To achieve a burst with 300 requests in every 3 seconds (V=300, L=50, I=3), more conservative predict needs bigger botnet.

<table>
<thead>
<tr>
<th>Inter-request Model</th>
<th>Botnet</th>
<th>Designed Attack Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual</td>
<td>Predicted</td>
<td>minS</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>300</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>600</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>1200</td>
</tr>
<tr>
<td>3</td>
<td>24</td>
<td>2400</td>
</tr>
</tbody>
</table>

6 DETECTION AND DEFENSE

Here, we consider a solution to detect and defend against Tail Attacks. There exists no easy approach to accurately distinguish the attack requests from legitimate requests. Instead, we can identify the attack requests by detecting the burst and matching the bursty arrivals to milli-bottlenecks and cross tier queue overflow (Event2 and Event3 in Section 2). We present a workflow to mitigate our attacks involving three stages: fine-grained monitoring, burst detection, and bots blocking.

Fine-grained monitoring. The unique feature of our attacks is that we exploit the new-found vulnerability of milli-bottlenecks (with subsecond duration) in recent performance studies of web applications deployed in cloud [41–43]. In order to capture milli-bottlenecks, the monitoring granularity should be less than the milli-bottlenecks period in millisecond level. For example, if the monitoring granularity is 50ms, it can definitely pinpoint the milli-bottlenecks period in millisecond level. Another important guide to our attacks is that as the sampling intervals increase, the threshold of the rules also has to increase, which can give us more flexible options to send the attack requests using different intervals (e.g., in the above case, less than 10 requests every 30s interval per session, or less than 20 requests every 60s interval per session). To choose which sending pattern, we can further learn from the other user-behavior models to make our attacks even more stealthy.

Someone may argue that the "no alert" results are got from the assumption that the attackers comprehensively know the alert thresholds of the user-behavior model, such that they can design a corresponding attack pattern to avoid being detected. Most user-behavior models are public to both the defenders and the attackers, the only gap is the specific alert threshold and rules, which typically are learned from the server’s past logs for their anomaly detection systems by the defenders of specific websites [33]. However, the attackers can use the questionnaire approach to similarly estimate the real users’ behaviors, and use a conservative value as the potential threshold to design the attack pattern with the price of increasing more bots shown in Table 7. The four rows in Table 7 show four cases with different predicted values (3, 6, 12, 24). Obviously, as the predicted value is more conservative, it requires a bigger botnet (minimal size is 2400 when the prediction is 24). In the ‘24’ case, the attacker must split the 2400 bots into 8 groups, each group takes turn to send a burst of 300 requests in every 24-second interval.
bursts from them. Based on the unique scenario of our attacks in Section 2, we can define our attack bursts in which all the following events occur simultaneously: very long response time requests (dropped requests), cross tier queue overflowed, millibottlenecks (e.g., CPU utilization, I/O waiting), and burst of requests. If all the events are observed in the same spike duration, we can regard the spike duration as a potential attack burst.

**IP-based statistical analysis defense.** Once we identify the bursts of Tail Attacks, the next task is to distinguish the requests of the bots from the requests of the legitimate users during the burst and block them. The attacker, in our attack scenario, aims to coordinate and synchronize the bots to sending bursts of attack requests during short “ON” burst period and repeat the process after long “OFF” burst period as introduced in Section 2, we can ideally introduce a new request metric that quantifies the suspicion index of the incoming requests by aggregating the requests statistics during “ON” burst and “OFF” burst for further analysis. Specially, we define the suspicion index for each IP address as follows:

\[ SI_{IP} = \frac{NB_{IP}}{N_{IP}} \]  

(19)

where \( NB_{IP} \) and \( N_{IP} \) are the number of requests for each IP during “ON” burst and the attack interval \( T \) (including “ON” and “OFF” burst), respectively. If \( SI_{IP} \) for a IP is close to 1, the IP is likely to be a bot; on the other hand, \( SI_{IP} \) of the legitimate user can be approximately the target attack drop ratio (e.g., 0.5 if the target is 95th percentile response time longer than 1 second). Figure 9 shows Probability Density Function of \( SI_{IP} \) in the RUBBoS experiment in Section 5.1. The red bar represents the bots and the green bars represent the 2000 legitimate users. In this way to identify bots, the false positive and false negative error can be close to 0 with 100% high precision.

7 DISCUSSIONS

**Impact of load balancing.** Some web applications adopt load balancing (e.g., Amazon Elastic Load Balancing [4]) in front of the system to distribute load across multiple web servers. This type of load balancing works well for stateless servers such as web servers, in which incoming traffic are supposed to evenly distribute among them. However, whether or not load balancing can mitigate the effectiveness of Tail attacks depends on the location of the bottleneck resource in the target web system. For example, if the bottleneck resource is in the web server tier, load balancing indeed increases the bar of an effective Tail attack since each web server only receives a portion of total attacking requests, thus higher volume of attacking requests per burst are needed to create millibottlenecks in the system. On the other hand, if the bottleneck resource is in a non-scalable tier such as the relational database tier, the load balancing in front tiers does not help mitigating the effectiveness of Tail Attacks. This is because no matter which web server an attacking HTTP request arrives at, the database queries sent out by the HTTP request may eventually converge to the same database server, and create millibottlenecks there.

**Impact of cloud scaling.** Large-scale web applications usually adopt dynamic scaling strategy (e.g., Amazon Auto Scaling [2]) for better load balancing and resource efficiency, however, Tail Attacks can easily bypass current dynamic scaling techniques, since the control window of the state-of-the-art scaling mechanism is usually in minute-level (e.g., Amazon CloudWatch monitoring in a minute granularity), while Tail Attacks are too short (sub-second duration) for them to catch and take any scaling actions [40]. The main advantage of Tail attacks is that it is invisible to most monitoring programs and can remain hidden for a long time because of the low volume characteristic and the sub-second duration of millibottlenecks as shown in our experimental results.

**Browser compatibility check.** Some state-of-the-art defense tools may check the header information of each HTTP request to determine whether it is sent from a real browser or from a bot. A HTTP request sent from a real browser usually has completed header information such as “User-Agent”, while such information may not appear or be difficult to be generated by a bot which only uses a script language to generate HTTP requests. In addition, some websites such as Facebook need a legitimate user to login first before any following transactions, especially heavy query requests (detailed explanations in Section 4.2). They may track the cookies stored in the client side browser in order to keep an active session in the server side; a bot may not be able to interact with such websites due to the lack of support of a real browser. We can address these challenges by using PhantomJS [36] to generate attacking HTTP requests. PhantomJS is a headless web browser without a heavy graphical user interface. It is built on top of WebKit, the engine behind Chrome and Safari. So PhantomJS can behave the same as a real browser does. Therefore, an attacker can launch browser-based Tail Attacks using heavy requests as attack requests by PhantomJS, and the generated requests will be extremely difficult to distinguish from the requests sent by legitimate users.

**Distributed bots coordination and synchronization.** One precondition of Tail attacks is that bots could be coordinated and synchronized so that the generated HTTP requests are able to reach the target web system within a short time window. Many previous research efforts already provide solutions, using either centralized [12, 37, 49] or decentralized methods [22], to coordinate bots to send synchronized traffic to cause network congestion at a specific network link. Centralized control can achieve higher level of bots coordination and synchronization, which enables a more effective Tail Attack compared to decentralized methods. In this paper we adopt the centralized control method to do experiments. On the other hand, a decentralized method in general is able to coordinate and synchronize more bots than a centralized one, thus making it possible to target large-scale/high-capacity websites.
However, a decentralized method is more challenging to control the length of each burst of attacking requests arrived at the target website, thus mitigating the effectiveness of Tail Attacks.

8 RELATED WORK

In this section, we review the most relevant work with regard to low-volume application-layer attacks, which is even stealthier to avoid traditional network-layer based detection mechanisms [30, 35, 48]. Low-volume Application Layer DDoS attacks. Low-volume DDoS application attacks are characterized by a small number of attack requests transmitted strategically to the target application servers, as an extension of network-layer low-volume attacks [12, 14, 21, 22, 25, 27, 39]. Macia-Fernandez et al. initially proposed low-rate attacks against application servers (LoRDAS) [28] that send traffic in periodic short-time pulses at a low rate, sharing the similar on-off attack pattern with our attacks. Slow-rate attacks [8] deplete system resources on the server’s side by sending (e.g., slow send/Slowloris [13]) or receiving (e.g., slow read) traffic at a slow rate. Our attacks share the similar features of low attack volume with low-rate and slow-rate attacks.

However, compared to these two attacks, our attacks dig more deeply into n-tier architecture applications while LoRDAS attacks only in 1-tier application server. Our analytical model for n-tier systems can guide and guarantee our attacks dynamically controlled in a more effective and elusive way by accurately estimating damage length and millibottleneck length. In addition, slow-rate attacks need to develop well-crafted HTTP header (Slow Headers) or body (Slow Body) thus expose obvious attack patterns to the defense tools, while Tail Attacks utilize the legitimate and normal heavy requests as our attack requests thus hide deeper. More importantly, we exploit the ubiquity of millibottlenecks (with sub-second duration) and strong dependencies among distributed nodes for web applications, leading to long-tail latency problem with a higher level of stealthiness than LoRDAS and Slow-rate attacks.

Detecting and Defending against Application DDoS attacks. To mitigate application DDoS attacks, existing solutions typically focus on distinguishing application-layer DDoS traffic from the traffic created by legitimate users, such as abnormal user-behavior in high-level features [44, 46, 47] of surging web pages, in session-level [38], or in request-level [45]. To be stealthy, for the features of navigating web pages, our attacks can learn from the user behaviors of a legitimate user; as for session or requests level of our attacks, we can calculate the required optimized attack volume and botnet size discussed in Section 5. Compared to existing solutions, our proposed countermeasure can be tied to the unique feature of our attacks and accurately capture the bots.

9 CONCLUSION

We described Tail Attacks, a new type of low-volume application-layer DDoS attack in which an attacker exploits a newly identified system vulnerability (millibottlenecks and resource contention with dependencies among distributed nodes) of n-tier web applications to cause the long-tail latency problem of the target web application with a higher level of stealthiness. To thoroughly comprehend the attack scenario (Section 2), we formulated the impact of our attacks in n-tier systems based-on queueing network model, which can effectively guide our attacks in a stealthy way (Section 3). We implemented a feedback control-theoretic (e.g., Kalman filter) framework that allows attackers to fit the dynamics of background requests or system state by dynamically adjusting the optimal attack parameters (Section 4). To validate the practicality of our attacks, we evaluated our attacks through not only analytical, numerical, and simulation results but also benchmark web applications equipped with state-of-the-art DDoS defense tools in real cloud production settings (Section 3 and Section 5). We further proposed a solution to detect and defend the proposed attacks, involving three stages: fine-grained monitoring, identifying bursts, and blocking bots (Section 6). More generally, our work is an important contribution towards a comprehensive understanding of emerging low-volume application DDoS attacks.

A DERIVATION OF MILLIBOTTLENECK LENGTH

During millibottleneck length, the bottleneck resources sustain saturation. Millibottleneck length should involve the serving time for both attack and normal requests during a burst. The attackers only send requests within the short “ON” period, thus the amount of attack requests is the burst volume \(V\). Meanwhile, the legitimate users always send requests during millibottleneck length, thus the saturation length is an infinite recursive process until it converges to zero exponentially. Equation (20) represents millibottleneck length derived through the geometric progression in mathematics, here, \(m\) limits to infinity.

\[
P_{MB} = V \cdot \frac{1}{C_{n,A}} + V \cdot \frac{1}{C_{n,A}} \cdot \lambda_n \cdot \frac{1}{C_{n,L}} + \ldots
\]

\[
= \lim_{m \to \infty} \sum_{k=0}^{m} V \cdot \frac{1}{C_{n,A}} \cdot (\frac{\lambda_n}{C_{n,L}})^k
\]

\[
= V \cdot \frac{1}{C_{n,A}} \cdot \lim_{m \to \infty} \frac{(1 - (\lambda_n \cdot \frac{1}{C_{n,L}})^{m+1})}{(1 - (\lambda_n \cdot \frac{1}{C_{n,L}}))}
\]

\[
= V \cdot \frac{1}{C_{n,A}} \cdot \frac{1}{1 - (\lambda_n \cdot \frac{1}{C_{n,L}})}
\]

where \(1/C_{n,A}\) and \(1/C_{n,L}\) are the service time for attack and normal requests in the bottleneck tier, respectively.

ACKNOWLEDGMENTS

This research has been partially funded by National Science Foundation by CISE’s CNS (1566443, 1421561), SAV/RCN (1402266, 1550379), CRISP (1541074), SaTC (1564097) programs, an REU supplement (1545173), Louisiana Board of Regents under grant LEQSF (2015-18)-RD-A-11, and gifts, grants, or contracts from Fujitsu, HP, Intel, and Georgia Tech Foundation through the John P. Imlay, Jr.
Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

REFERENCES