Sync-Millibottleneck Attack on Microservices Cloud Architecture

Xuhang Gu
Louisiana State University
xgu5@lsu.edu

Qingyang Wang
Louisiana State University
qwang26@lsu.edu

Qiben Yan
Michigan State University
qyan@msu.edu

Jianshu Liu
Louisiana State University
jliu96@lsu.edu

Calton Pu
Georgia Institute of Technology
calton@cc.gatech.edu

ABSTRACT

The modern web services landscape is characterized by numerous fine-grained, loosely coupled microservices with increasingly stringent low-latency requirements. However, this architecture also brings new performance vulnerabilities. In this paper, we introduce a novel low-volume application layer DDoS attack called the Sync-Millibottleneck (SyncM) attack, specifically targeting microservices. The goal of this attack is to cause a long-tail latency problem that violates the service-level agreement (SLA) while evading state-of-the-art DDoS detection/defense mechanisms. The SyncM attack exploits two unique features of microservices architecture: (1) the shared frontend gateway that directs user requests to mid-tier/backend microservices, and (2) the co-existence of multiple logically independent execution paths, each with its own bottleneck resource. By creating synchronized millibottlenecks (i.e., sub-second duration bottlenecks) on multiple independent execution paths, SyncM attack can cause the queuing effect in each execution path to be propagated and superimposed in the shared frontend gateway. As a result, SyncM triggers surprisingly high latency spikes in the system, even when all system resources are far from saturation, making it challenging to trace the cause of performance instability.

To evaluate the practicality of the SyncM attack, we conduct extensive experiments on real cloud systems such as EC2 and Azure, which are equipped with state-of-the-art IDS/IPS systems. We also conduct a large-scale simulation using a production Alibaba trace to show the scalability of our attack. Our results demonstrate that the SyncM attack is highly effective, as it only consumes less than 15% of additional CPU resources of the target system while increasing its 95th percentile response time by more than 20 times.

CCS CONCEPTS

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

KEYWORDS

Microservices, DDoS attack, Long tail latency, SLA violations

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 the author(s) 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.
ASIA CCS '24, July 1–5, 2024, Singapore, Singapore
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0482-6/24/07…$15.00
https://doi.org/10.1145/3634737.3644991

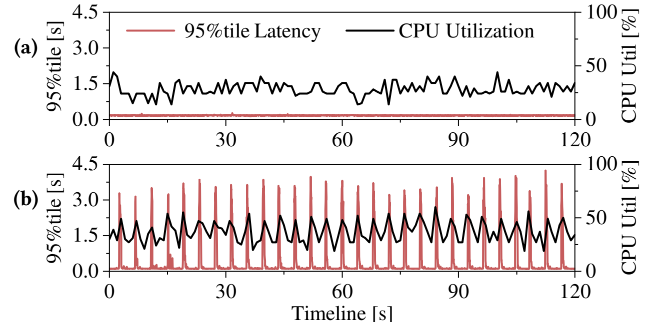


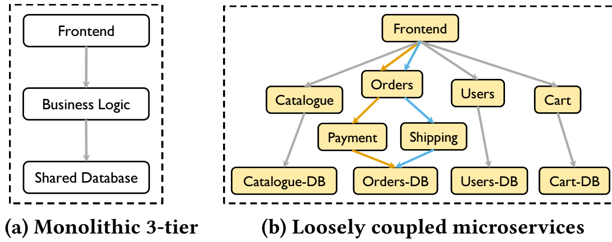
Figure 1: An illustration of system tail latency and the bottleneck service CPU util. of the target microservices SocialNetwork website. (a) Without attack. (b) Under SyncM attack.

ACM Reference Format:
Xuhang Gu, Qingyang Wang, Qiben Yan, Jianshu Liu, and Calton Pu. 2024. Sync-Millibottleneck Attack on Microservices Cloud Architecture. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3634737.3644991

1 INTRODUCTION

Fast response time is crucial for modern web applications, particularly for e-commerce that faces intense business pressure. Studies have shown that even small increases in page loading time can lead to significant sales losses. For example, Amazon [35] found that a 100ms increase in page loading time resulted in roughly a 1% reduction in sales, while Google [34] found that a 500ms delay in search results could cause up to a 20% decrease in revenue. The need for fast response times is even more critical for emerging augmented-reality devices, such as Apple Vision Pro, as these devices require extremely responsive web services to provide natural and smooth user experiences. In practice, response-time sensitive web-facing applications are particularly concerned with tail latency (e.g., 95th or 99th percentile), rather than the average. This is because studies have shown that negative experiences tend to stick in people's minds more than positive ones [26]. For instance, if the 95th percentile latency is long (e.g., > 3 seconds), and a client visits the target website with 10-page loads, there is a 41% chance that they will experience at least one slow response [65]. This means that almost half of the potential customers will have a degraded user experience. The sensitivity of tail latency for modern web applications creates new vulnerabilities for attackers to exploit.

In this paper, we introduce a new type of low-volume application layer DDoS attack called the Sync-Millibottleneck (SyncM) attack,



(a) Monolithic 3-tier **(b) Loosely coupled microservices**

Figure 2: Two typical web service architectures.

which specifically targets microservices. The objective of the SyncM attack is to cause performance instability that violates the service-level agreement (SLA) for latency-sensitive targets while evading state-of-the-art DDoS detection and defense systems. Microservices applications have two distinct characteristics that SyncM attack exploits: (1) the sharing of a frontend gateway microservice that dispatches user requests to mid-tier/backend microservices [54, 58], and (2) the co-existence of multiple logically independent execution paths, each with its own critical resource bottleneck. In a typical SyncM attack scenario, the attacker sends intermittent bursts of mixed types of legitimate HTTP requests to multiple selected execution paths of the target microservices. These requests trigger synchronized millibottlenecks on those execution paths, resulting in a queuing effect that is propagated and superimposed in the shared frontend gateway microservice. The queuing effect produces high latency spikes that violate the SLA for e-commerce (as shown in Fig. 1b). Since the created millibottlenecks in each target execution path only last for tens to hundreds of milliseconds, all the system resources are far from saturation (e.g., <50%), making it difficult to trace the root cause of the performance degradation.

Compared to the well-understood brute-force DDoS attacks, low-volume attacks pose a greater security threat to businesses since those attacks tend to go undetected, allowing the damage to persist over an extended period [11, 15, 51, 66]. Some well-known low-volume attacks including Shrew [36] and Slowloris [42] exploit protocol weaknesses, low-rate network layer DDoS attacks [31, 32, 55], and flash crowds [29, 71]. The novelty of the SyncM attack lies in the exploitation of the new architecture-level weaknesses of microservices and synchronized millibottlenecks (e.g., milliseconds saturation on CPU, memory, or disk I/O) among multiple independent execution paths in a microservices architecture. Such synchronized millibottlenecks persistently degrade the system performance while escaping the detection of the state-of-the-art IDS/IPS tools that rely on coarse-grained (in seconds or minutes) monitoring [5, 45].

The primary challenge in launching an effective SyncM attack is to identify the triggering conditions for synchronized millibottlenecks across multiple independent execution paths and quantify the performance damage to the overall system. To gain a precise understanding of the SyncM attack, we adopt a modified queuing network to model the attack scenario. The model is used to numerically analyze the impact of the attack on the target system and derive the optimal attack parameters to achieve the desired attacking goals since there is usually a trade-off between performance damage and stealthiness. To ensure that the SyncM attack is effective in real cloud settings, we develop a feedback control framework based on the Kalman filter to fit the runtime dynamics of the target. The framework enables fast adaptation of attack parameters to the variation of system state. With the help of the queuing network model and the feedback control framework, we show that the SyncM attack is able to achieve the pre-defined damage goals under various workload conditions and cloud settings while avoiding detection by state-of-the-art DDoS detection mechanisms.

In brief, our contributions include:

- The first low-volume DDoS attack that targets microservices architecture by exploiting synchronized cross-service millibottlenecks, which ensures that the attack is both harmful and stealthy;
- A mathematical model based on queuing network theory that accurately quantifies the impact of our attacks on microservices;
- A feedback control framework design/implementation that allows attackers to dynamically fit the variation of background workload and system state;
- Real-world demonstrations of the impact of our attack in production clouds (e.g., EC2 and Azure). Our experimental results show that the SyncM attack consumes less than 15% additional CPU resources of the target system while increasing the 95th percentile response time over 20 times (Table 4); and
- A large-scale simulation verifying the effectiveness of the SyncM attack on production microservices composed of thousands of component microservices (Table 7).

2 THREAT MODEL AND MOTIVATION

2.1 Performance Vulnerability of n-Tier vs. Microservices

In traditional web applications with the monolithic n-tier architecture (see Fig. 2a), the business logic is tightly-coupled, creating strong dependencies between consecutive tiers. Due to the monolithic role of each tier, if one tier experiences performance anomalies, the entire system will be impacted. The monolithic architecture leaves a severe performance vulnerability that allows low-volume DDoS attacks [42, 60] to target a single tier of the system and cause significant performance degradation to the entire system.

Unlike the monolithic n-tier architecture, the microservices architecture adopts lightweight container-based services that make it more resilient to the existing low-volume DDoS attacks. Fig. 2b shows the SockShop microservices benchmark [67]. The microservices application is split into a set of loosely-coupled microservice components, where each runs as an independent service, interacting with other components through the classic RPC style call/response [23, 41]. Once a component encounters performance anomalies, the damage is limited to its local or the execution path that the component is involved in. Thus, microservices applications usually have better performance anomaly tolerance compared to the monolithic n-tier systems [6, 37]. In particular, a production microservices (e.g., e-commerce like Alibaba) application may involve thousands of components that constitute hundreds of execution paths with complex inter-path dependencies [41]. This makes the state-of-the-art stealthy DDoS attacks targeting a single component (or tier) ineffective on microservices[3, 6]. Thus, how to cause performance instability of the target microservices application while still staying stealthy motivates our work.

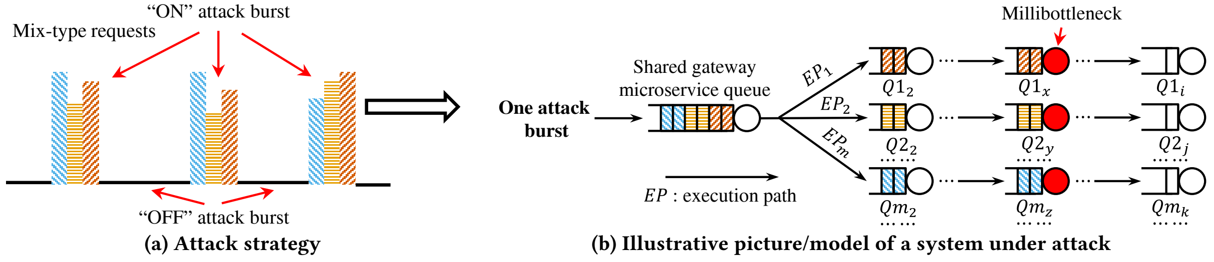


Figure 3: SyncM attack scenario and the system model. Each color means one type of attack HTTP requests, which have its own execution path through the shared frontend gateway microservice.

2.2 Threat Model

We assume a SyncM attacker acts as a normal user accessing the target microservices application through public HTTP requests. The attacker does not have prior knowledge about the internal implementation of the target, and the baseline workload from legitimate users. As a normal user, the attacker can profile the microservices application by sending different HTTP requests and accurately observing the end-to-end latency of the respective requests. To launch a SyncM attack, the attacker can recruit an army of bots to synchronously send attack requests to the target systems.

2.3 SyncM Attack Scenario

A typical SyncM attack scenario is illustrated in Fig. 3. The attacker adopts a Short-ON and Long-OFF pulsing style to send bursts of mixed types of legitimate HTTP requests to the target. The Short-ON period is typically on the order of milliseconds while the Long-OFF can be multiple seconds, which guarantees both harmful and stealth. The following sequence of events occurs during a Short-ON period. **(Event1)** The attacker sends a burst of mix-type requests to the target system over a short time (e.g., 50ms). **(Event2)** The frontend gateway dispatches each type of request to its independent execution path and creates a millibottleneck at the weakest component along the execution path. **(Event3)** The millibottleneck in each target execution path blocks the processing of incoming requests at the bottleneck component, causing requests to fill up the local queue (e.g., thread pool) and quickly propagate the queued requests to its upstream components. **(Event4)** Queued requests from all targeted execution paths converge at their shared gateway (See the shared queue in Fig. 3b), leading to a superimposed queuing effect in the gateway. **(Event5)** Requests from normal users encounter the extra long queuing delay in the shared gateway, leading to a long response time (e.g., order of seconds) that violates SLA.

The above attack scenario is stealthy in two ways. First, every Short-ON period is followed by a Long-OFF period in the order of seconds, in which the target system can cool down, clear up the queued requests in the system, and return to a long normal state. Second, the damage of the attack is due to the compound effect of multiple synchronized millibottlenecks in independent execution paths, where each millibottleneck itself is on a scale of milliseconds, invisible by normal monitoring tools sampling at seconds (e.g., the finest granularity of Amazon CloudWatch [5] is 1-second). By tuning attack parameters such as Short-On/Long-OFF periods and the selection of attack requests, the attacker can effectively balance the trade-off between performance damage and stealthiness.

Table 1: Measured long-tail latency problem in SocialNetowrk application under the SyncM attack.

Setting	Avg. RT (ms)		95ile RT (ms)		99ile RT (ms)		CPU (%)	
	Base.	Att.	Base.	Att.	Base.	Att.	Base.	Att.
EC2-SN-7k	144	396	148	3104	161	4847	31	41
EC2-SN-12k	143	386	154	3018	169	5132	49	54
Azure-SN-4K	155	410	161	3271	175	4757	25	40
Azure-SN-9k	157	407	163	3265	177	4871	51	61
CloudLab-SN-5k	153	411	159	3396	192	5089	23	38
CloudLab-SN-11k	159	423	161	3277	207	4958	43	54

Base.: baseline without attacks. **RT.:** end-to-end response time.
CPU.: average CPU usage of a representative bottleneck component.

2.4 Measured Damage under SyncM Attack.

Table 1 summarizes the impact of the SyncM attack through a representative microservice benchmark (SocialNetwork [19]) with production settings deployed in three cloud platforms: Amazon EC2, Microsoft Azure, NSF CloudLab. The setting EC2-SN-7k means the cloud platform, the benchmark system, and the baseline workload. The more detailed experimental setup is in Section 5.1. We compare the 95th and 99th percentile response time of the target system with and without SyncM attack, showing a significant long tail latency problem caused by our attack. Such long tail latency problem (e.g., 95th percentile response time > 3 seconds) is considered as critical performance degradation by most e-commerce web and IoT backend applications [16, 65, 76]. Meanwhile, CPU utilization of bottleneck microservice components (column 9) is increased by less than 15% under attack, which is far from saturation, creating an illusion of "normal workload" for system administrators.

3 SYNCM ATTACK MODELING

In this section, we apply a well-tuned queuing network model to numerically analyze the relationship between the attacking parameters and their impact on performance damage and stealthiness. With the model, we also conduct a boundary analysis to understand the limitations and how to address such limitations by adjusting attacking parameters, given various attacking goals.

3.1 Attack Model

Queuing network models have been widely used in complex computer systems for performance prediction [33, 76]. We use a queuing network to model microservices and analyze the performance impact of SyncM attack as shown in Fig. 3. Table 2 summarizes the model notations. As Fig. 3a shows, during the Short-ON period (L) the attacker sends a burst of mixed requests to attack multiple (m)

Table 2: Model parameters.

Param.	Description
Q_i	the queue size for the ith component
$C_{i,A}$	the capacity of the ith component serving attack requests
$C_{i,L}$	the capacity of the ith component serving legitimate requests
λ_i	the legitimate request rate for the ith component
V	the total attack volume of an attack burst
B_i	the attack rate for attacking ith execution path
L_i	the attack length for attacking ith execution path
T	the interval between every two consecutive attack bursts
l_i	the time to fill up the queue of the ith component
$Q_{gateway}$	the gateway queue caused by an attack burst
t_{damage}	the extra delay for a legitimate request caused by an attack burst
P_{MB}	the length of a millibottleneck caused by an attack burst
P_D	the period during which requests violate latency SLA
$\rho(T)$	the damage ratio during T that requests violate latency SLA
$L_{max,i}$	the maximum attack length allowed for the ith execution path, given a millibottleneck length boundary (e.g., 500ms)
$t_{max,i}$	the maximum latency that can be achieved by attacking the ith execution path, given a millibottleneck length boundary

execution paths. After each Long-OFF period ($T - L$), the attacker repeats the burst. Given a target execution path i, we use B_i to L_i to denote the attacking rate and length respectively. For each burst, the total attack volume is:

$$V = \sum_{i=1}^{m} B_i * L_i \tag{1}$$

Each burst of mixed requests is to create synchronized millibottlenecks on multiple execution paths, thereby leading to a superimposed queuing effect in the frontend gateway (Event1~Event5 in Section 2.3). We assume that the target execution paths do not overlap except for the gateway (Section 4.2 discusses how to find such execution paths through profiling). The attack can be modeled in two steps: (1) single-path attack modeling, and (2) multi-path attack modeling with a superimposed queuing effect in the gateway.

Single-path attack modeling. Tail Attack [60] introduces a queuing network to model a similar attack on a monolithic n-tier system. It assumes a limited queue size of the gateway server so that a millibottleneck in a downstream tier could degrade the performance of the entire system once the gateway queue is full. In that case, the gateway initiates request drops, leading to long TCP retransmissions. However, this assumption does not hold on microservices architecture for two reasons. Firstly, the loosely coupled microservices have better performance anomaly tolerance as introduced in Section 2.1. Secondly, the gateway of the modern microservices architecture typically adopts the asynchronous event-driven invocation, which can hold as many queued requests as the server memory allows, thus no dropped requests by the gateway. Given the change of the key assumption, we expand the queuing model from Tail Attack to align with the microservices architecture in the single-path attack modeling.

The communication between microservice components typically adopts the RPC-style call and response [23, 41], creating the inter-component dependency. Thus, one queued request in a downstream component holds a pending queue slot (e.g., a thread waiting for responses) in every upstream component along the execution path. A millibottleneck in a downstream component can cause cross-service queue propagation to its upstream components (Event3 in Section 2.3) due to the inter-component dependency. To avoid queue drop, an upstream component is usually configured to have a bigger queue size than its downstream component ($\forall i \in \{2, \ldots, n\}, Q_{i-1} > Q_i$). Then, for all microservices along the target execution path, the time needed to fill up each queue is:

$$l_n = \frac{Q_n}{\lambda_n + B - C_{n,A}} \tag{2}$$

$$\ldots$$

$$l_2 = \frac{Q_2 - Q_3}{\sum_{i=2}^{n} \lambda_i + B - C_{n,A}} \tag{3}$$

Where the queue fill-up time l_i is derived using the available queue slot of the i-th component (e.g., the numerator in Eqn. 2) divided by the queue fill-up rate for the queue of the i-th component (e.g., the denominator in Eqn. 2).

Once all the queues are filled up in the downstream components ($L > \sum_{i=2}^{n} l_i$), the gateway continues to build a queue ($Q_{gateway}$) during the remaining attack "ON" period ($l_{gateway}$).

$$l_{gateway} = L - \sum_{i=2}^{n} l_i \tag{4}$$

$$Q_{gateway} = l_{gateway} * (\sum_{i=1}^{n} \lambda_i + B - C_{n,A}) + Q_2 \tag{5}$$

Eqn. 4 shows that L needs to be long enough to push requests to the gateway if the millibottleneck occurs in the n-th component. Then we calculate the total queued requests at the gateway in Eqn. 5. Where $l_{gateway}$ is the build up time, $(\sum_{i=1}^{n} \lambda_i + B - C_{n,A})$ is the build up rate, and Q_2 is queue slot occupied by downstream components.

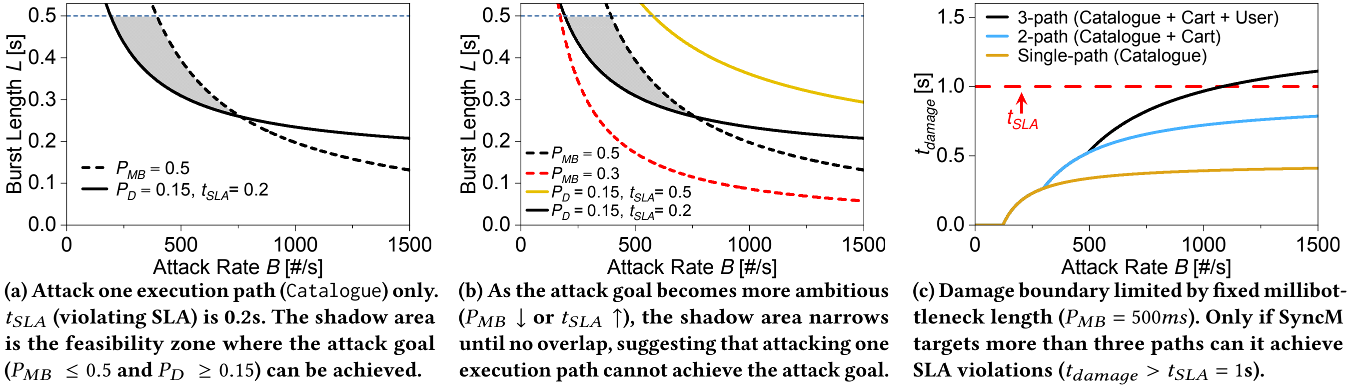
When requests are queued up in the gateway ($Q_{gateway}$), they will delay new incoming requests since the gateway is shared by all the execution paths. Thus, we refer the time to drain out the gateway queue as the damage queuing delay (or damage latency) t_{damage}, which can be calculated as:

$$t_{damage} = \frac{Q_{gateway}}{C_{n,A}} \tag{6}$$

The gateway queue drain out rate is determined by the bottleneck component service rate ($C_{n,A}$) along the execution path. And the delay time for new incoming requests from normal users is t_{damage}. We use t_{SLA} to denote the desired damage latency that violates SLA (e.g., t_{SLA}=1s). To reach the desired damage latency t_{SLA}, the minimum queue fill-up time in the gateway $l_{gateway}^{SLA}$ can be derived by combing Eqn. 5 and Eqn. 6. In this case, for each attack burst L, we can calculate the damage period P_D, during which incoming requests will have a delay longer than t_{SLA} (thus violating SLA). The *damage length* P_D for each attack burst can be calculated as:

$$P_D = L - (\sum_{i=2}^{n} l_i + l_{gateway}^{SLA}) \tag{7}$$

Furthermore, since the attack burst repeats itself every T-second on average, the overall attack damage can be estimated as $\rho(T) = P_D/T$. Here $\rho(T)$ means the damage ratio that legitimate requests violate the latency SLA during the system runtime.



(a) Attack one execution path (`Catalogue`) only. t_{SLA} (violating SLA) is 0.2s. The shadow area is the feasibility zone where the attack goal ($P_{MB} \leq 0.5$ and $P_D \geq 0.15$) can be achieved.

(b) As the attack goal becomes more ambitious ($P_{MB} \downarrow$ or $t_{SLA} \uparrow$), the shadow area narrows until no overlap, suggesting that attacking one execution path cannot achieve the attack goal.

(c) Damage boundary limited by fixed millibottleneck length ($P_{MB} = 500ms$). Only if SyncM targets more than three paths can it achieve SLA violations ($t_{damage} > t_{SLA} = 1s$).

Figure 4: Boundary analysis of feasible attack parameters. (a) and (b) show the limitations of attacking a single microservice. (c) shows that SyncM attack can achieve a more ambitious damage goal (e.g., $t_{SLA} > 1s$) by attacking more execution paths.

Table 3: Constant parameters profiled in Sock Shop experiments with 7000 normal users (baseline workload).

Component	λ_i (#/s)	Q_i (#)	$C_{i,A}$ (#/s)	$C_{i,L}$ (#/s)
Frontend	0	Infinite	5010	6531
Catalogue	0	250	3023	4439
CatalogueDB	514	100	457	1815
Cart	0	150	1969	3350
CartDB	271	100	585	1467
User	0	200	1359	2141
UserDB	213	100	2487	3101

On the other hand, to keep the attack stealthy, the millibottlenecks created should be as short as possible. The period of a millibottleneck caused by a burst is termed as P_{MB} and can be derived as follows, which is adapted from Tail Attack [60]:

$$P_{MB} = B * L * \frac{1}{C_{n,A}} * \frac{1}{(1 - (\lambda_n * \frac{1}{C_{n,L}}))} \tag{8}$$

Eqn. 7 and 8 reveal the relationship between attack parameters (L, B) and their attack impact (P_D, P_{MB}). A longer attack burst length L will cause more damage P_D to the target system (Eqn. 7). On the other hand, a longer burst length L also creates a longer millibottleneck length P_{MB} (Eqn. 8), which may violate the stealthy requirement. To address the single-path limitation and exploit the damage of synchronized millibottlenecks, we propose multi-path modeling for the microservices architecture.

Multi-path attack modeling. Assume that the target microservices application has m execution paths; except for the shared gateway, all m execution paths do not share their bottleneck component. Based on this assumption, an attacker can create at most m synchronized millibottlenecks among those execution paths. Given the restriction of each millibottleneck length (e.g.,$P_{MB} \leq 500ms$), for each target execution path i, the maximum attack burst length $L_{max,i}$ can be derived from Eqn. 8. In this case, the maximum damage latency created by the attack burst $L_{max,i}$ is $t_{max,i}$. Only when $t_{max,i} < t_{SLA}$, a multi-path attack is needed.

In a multi-path attack, the attacker sends mix-type requests in an attacking burst; each type creates a millibottleneck at the bottleneck component of its execution path. In this case, the queues of the gateway created by different types of requests are superimposed, and the delay in the gateway is also superimposed. One challenge for the attacker is to find the minimum number of execution paths EP_{min} so that the superimposed delay in the gateway can violate SLA:

$$\sum_{i=1}^{EP_{min}} t_{max,i} > t_{SLA} \tag{9}$$

In general, the more damage is desired ($t_{SLA} \uparrow$), the more execution paths ($EP_{min} \uparrow$) are needed. Since the attack burst length $L_{max,i}$ is needed to cause the delay $t_{max,i}$ on the path i, the overall damage length P_D by each multi-path burst is:

$$P_D = L - \sum_{i=1}^{EP_{min}} L_{max,i} \tag{10}$$

In a real SyncM attack scenario, how to find EP_{min} independent execution paths in the target microservices application is a challenge, which requires careful profiling of the target system (detailed discussion in Section 4.2).

3.2 Boundary Analysis

The model in the previous section allows us to quantify the damage P_D and stealthiness P_{MB} of our attack. On the other hand, if we know the model parameters, we can also validate whether a certain combination of attack goals (damage vs. stealthiness) is feasible or not through boundary analysis.

Constant Parameters Estimation. We estimate the constants of the model parameters (e.g.,$\lambda, Q, C_{i,L}, C_{i,A}$) via system profiling and configuration. For example, through profiling the service time of each component of the microservices website SockShop [67] (experimental setup in Section 5.1), we can derive the capacity of each microservice. We choose three types of attack requests, each targeting one specific execution path (see *Catalogue*, *Cart*, and *User* in Fig. 2b). Other than the shared gateway, each execution path has two microservice components (e.g.,<Cart, CartDB>). Table 3 lists the model constants profiled from the SockShop website serving 7000 legitimate users as the baseline workload.

Attack Goal and Optimal Attack Parameters. Assume our attack goals are to make the 95th percentile response time longer than 0.2 seconds ($t_{SLA} = 0.2s$) and the millibottleneck lengths shorter than 0.5 seconds. And we fix the interval as 3 seconds ($T = 3s$) between every two consecutive bursts. Then our attack goals can be

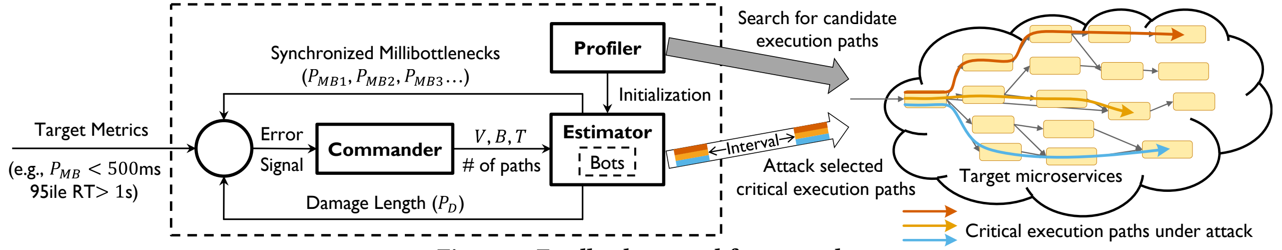


Figure 5: Feedback control framework

formalized as two inequations: $P_D \geq 0.15$s [1] and $P_{MB} \leq 0.5$s. Based on Eqns. 7 and 8, we turn the two inequations to L as a function of B, shown as follows. All other parameters are constants.

$$L >= \frac{0.2 * C_{n,A} - Q_2}{\sum_{i=1}^{n} \lambda_i + B - C_{n,A}} + \sum_{i=2}^{n} l_i \tag{11}$$

$$L <= 0.5 * (1 - \lambda_n * \frac{1}{C_{n,L}}) * \frac{C_{n,A}}{B} \tag{12}$$

Meanwhile, one obvious constraint is that the burst length should be less than the target millibottleneck length (i.e., $L < 0.5$s). Thus, how to select the optimal attack parameters (B,L) becomes a nonlinear optimization problem. Substituting the constant parameters from Table 3 in Inequation (11) and (12), we get a unique feasibility zone of the potential attack parameters (B,L) by attacking one execution path (Catalogue), shown in Fig. 4a. Fig. 4b shows that once the attack goal becomes more ambitious ($P_{MB} \downarrow$ or $t_{SLA} \uparrow$), the feasibility zone narrows until no overlap, suggesting that there is no solution to satisfy our attack goals by attacking only one execution path. Fig. 4c shows the damage latency (t_{damage}) boundary of attacking multiple execution paths while keeping $P_{MB} = 500ms$ at every target execution path. It shows we need to target at least three execution paths to achieve the damage goal $t_{damage} > t_{SLA} = 1$s.

The above boundary analysis helps a SyncM attacker determine whether a certain combination of goals can be achieved by adjusting attacking parameters, given the target system state. Next, we will show how to implement SyncM attack in real cloud settings.

4 SYNCM ATTACK IMPLEMENTATION

4.1 Overview

The analytical model in Section 3 characterizes the relationship between the attack parameters and their impact (stealthiness and damage) on the target. However, as external users, SyncM attackers do not know the internal system parameters (e.g., Table 3). Our analysis also does not consider more realistic conditions such as dependencies among execution paths, variation of baseline workloads, and drifting of system state. For example, on peak shopping days like Black Friday, online merchants see upwards of three times more traffic than usual [2]; the optimal attack parameters could become suboptimal over time in which the attacker either cannot trigger millibottlenecks or cannot meet the damage goal. Although it is impossible to get the internal system parameters or precisely predict the variation of runtime system workload, we know how the attack parameters impact attack results given the analytical model. In this section, we present a feedback control framework

[1] The damage ratio $\rho(T) = P_D/T = 0.15/3 = 0.05$ here, suggesting 95ile RT > 0.2s.

Algorithm 1: Pseudo-code for control algorithm

```
1  procedure AdaptAttackParameters
2  ReqST = EstimatedServiceTime;
3  NumTargetPath = EstimatedNumberOfTargetPath;
4  DamLen = EstimatedDamageLength;
5  MBLenSet = EsimatedMBLengthOfTargetPaths;
6  if DamLen = 0 then
7      /*not enougth queue at gateway, increase B*/
8      B = B + stepB;
9  else
10     /*find minimum number of paths to attack*/
11     if DamLen < targetDamLen then
12         NumTargetPath = NumTargetPath + 1;
13     else if DamLen > 2 * targetDamLen then
14         /*reduce target paths to keep stealthy*/
15         NumTargetPath = NumTargetPath − 1;
16     end
17 end
18 /*limit millibottleneck lengths*/
19 foreach MBLen in MBLenSet do
20     gapMBLen = |targetMBLen − MBLen|;
21     stepV = gapMBLen/ReqST;
22     if MBLen > targetMBLen then
23         V = V − stepV;
24     else if MBLen < targetMBLen then
25         V = V + stepV;
26     end
27 end
```

(using Kalman filter [30]) to dynamically tune the attack parameters to fit the dynamic of baseline workload and system state.

The control framework mainly includes three parts: Profiler, Estimator, and Commander (see Fig. 5). We first use the Profiler to profile the target system for the initialization of attack parameters. Secondly, we adopt the Estimator to estimate two critical metrics: the damage length P_D and the millibottleneck length P_{MB} of each target execution path to validate the fitness of attack parameters. Finally, we use the Commander to coordinate the bots and dynamically adjust attack parameters based on the collected feedback metrics from the Estimator. The overall control is in Algorithm 1.

4.2 Profiler

The Profiler is used by attackers to find the appropriate independent execution paths of the target system based on detailed system profiling, which involves the following three steps.

Step 1: Profile supported execution paths via URLs. First, an attacker can retrieve supported HTTP requests of the target system by scanning the website [1, 38, 46, 70]. We adopt PhantomJS [53], which is a script-based headless browser that can automatically retrieve/analyze both static and dynamic requests supported by the

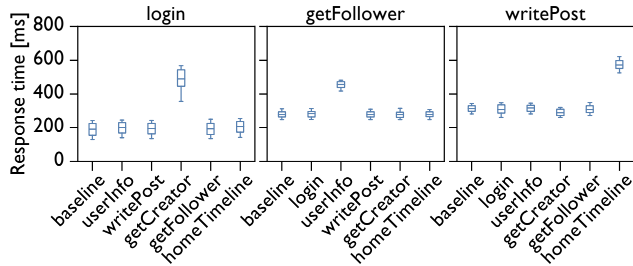
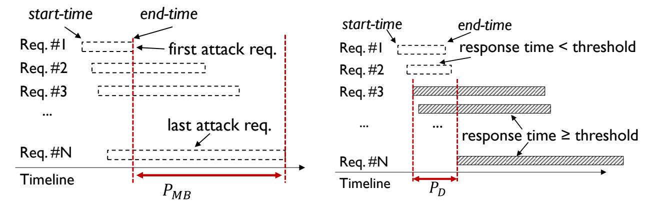


Figure 6: Boxplot of the target critical path response time when it conducts pair-wise dependency testing with other critical paths. We use SVM to determine the cases with significantly larger response time than the baseline as dependent (e.g., `login` and `getCreator` are dependent).



(a) Infer P_{MB} by end-time of the last attack request subtracting end-time of the first attack request within an attack burst.

(b) Infer P_D by the distance of start-time of the first and the last damage request (e.g., response time ≥ 1s) within an attack burst.

Figure 7: Illustration of inferring millibottleneck length P_{MB} and damage length P_D by Estimator

target website. For requests that require user input (e.g., username and password), attackers need to prepare initial values for the associate forms. By scanning supported HTTP requests, attackers can collect public URLs, each corresponding to its specific execution path on the server side. Non-valid requests (e.g., 404 Not Found) and static requests (e.g., CSS file) are lightweight and served directly by the frontend. Therefore, attackers do not consider such requests as candidates to attack backend microservices.

Step 2: Select critical paths. After profiling all the supported execution paths, the attacker needs to further identify the *critical paths* of the target system in order to launch efficient attacks. In our context, critical paths are referred to as the chain of invocations among components with the longest end-to-end latency [54]. The critical paths usually exist at those execution paths with the Critical Components (weakest components, where millibottlenecks happen) in the system and have more chance to consume bottleneck resources. Thus, attacking critical paths can achieve the desired performance damage with fewer requests (thus stealthier). To identify the critical paths, we rank all the supported execution paths according to their service time[2].

Typically, the end-to-end latency of an HTTP request involves three parts: the network latency between the client and the target, the queuing delay, and the service time of microservices along its execution path. When the target website is at low usage (e.g., midnight), the queuing time can be ignored [33]. The network latency can be measured as the Round Trip Time (RTT) using either Ping or a light HTTP request (e.g., only retrieving the header of a web page). Thus, after eliminating the network latency and queuing time from end-to-end latency, we can select critical paths based on the ranking of service time of the candidate execution paths.

Step 3: Avoid dependency among critical paths. To create synchronized but independent millibottlenecks among multiple critical paths (for stealthy purpose), SyncM attackers need to make sure that the selected critical paths are independent. However, in a production system, execution paths sharing the same bottleneck component are common, both logically and physically. The *logical dependence* is due to the shared microservice components among multiple execution paths. For example, the Alibaba trace [41] shows that 5% microservices (called "hotspot" microservices) are shared by 90% execution paths in their application. In this case, two selected critical paths may logically share the same bottleneck component. The *physical dependency* is due to the common practice of containers/VMs collocation for cost efficiency. Without careful profiling, in both dependency scenarios, attacking two critical paths may create repetitive millibottlenecks on the same resource (e.g., CPU), lengthening the original millibottlenecks with the risk of detection by normal monitor tools.

To determine whether two critical paths are independent or not, we check whether a performance interference occurs when sending attacking requests to both the critical paths in one small burst (to avoid the superimposed queuing effect in the shared gateway). Fig. 6 illustrates our pair-wise dependency testing for three sample critical paths (login, getFollower, and writePost) of SocialNetwork in three steps. (1) Baseline profiling. For each sample critical path, we profile the baseline response time without mixing with other request types in one burst. (2) Interference testing. We send a pair of mixed critical paths in one burst. (3) Response time analysis. If a pair-wise dependency exists, the response time of the sample critical path should be significantly higher than the baseline due to the performance interference by the other critical path. In our experiments, we adopted a one-class Support Vector Machine (SVM) classifier [57] to systematically detect the pair-wise dependency between all the critical paths chosen in **Step 2**.

Applying the aforementioned three steps, a SyncM attacker can evaluate every pair of candidate critical paths and select those without dependency as the real attack targets. In this way, multiple independent but synchronized millibottlenecks can be created to keep the attack both stealthy and harmful.

4.3 Estimator

We adopt and extend the Estimator from Tail Attack [60] based on the uniqueness of the microservices architecture. Tail Attack assumes that the performance damage occurs when the target n-tier system drops requests due to the limited queue size of gateway and uses a prober to detect the request dropping period. However, the target microservices of SyncM attack do not assume dropping requests (details in Section 3.1), and the performance damage is caused by superimposed queuing time in the gateway, thus requires a different way to estimate the damage length P_D.

Estimating millibottleneck length P_{MB}. To stay stealthy, we limit the millibottleneck length created by our attack (e.g., 500ms).

[2]Requests with the same URL but different parameters may have service time variations. For such requests, we choose the one with the longest service time.

Therefore, correctly estimating the millibottleneck length is important. After sending a burst of attack requests, attackers can record the start-time and end-time of each request. Assume the same type of requests flows through the same execution path, we estimate the millibottleneck length P_{MB} along the execution path by the end-time of the last attack request subtracting the end-time of the first request in an attack burst as shown in Fig. 7a. This estimation is reasonable because the burst of requests will continue to consume the bottleneck resource along the target critical path until the last one. We note that such an estimation undercharges the service time of the first request. Thus, it is a conservative estimation.

Estimating damage length P_D. A SyncM attacker needs to estimate the damage length for each attack burst to evaluate if the attack achieves the predefined damage goal. We define the requests with response time violating SLA (e.g., > 1 second) as *damage requests* and the period during which the damage requests continue to occur as *damage length*.

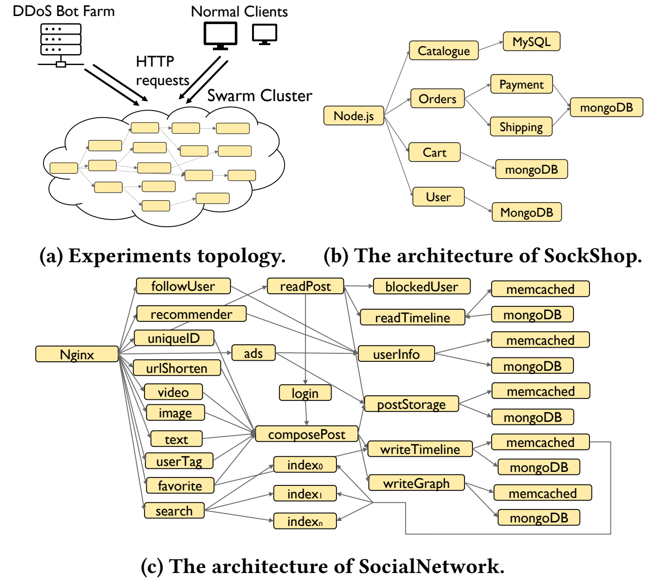
To estimate the damage length P_D of each burst, we use the *start-time* of the last damage request subtracting the *start-time* of the first damage request of the same burst as shown in Fig. 7b. This estimation is reasonable because the attacking requests and normal requests share the same frontend gateway (thus share the same queue); if the measured latency of the attack requests violates SLA, the normal requests during the same period (P_D) also violate the SLA. Once P_D of each burst is estimated, we can estimate the overall damage of the SyncM attack $\rho(T)$ (e.g., 95ile RT > 1s), given the interval T between consecutive attack bursts.

4.4 Commander

After the profiling phase, the SyncM attackers initialize the attack parameters and select independent critical paths to launch the attack. With the Estimator, attackers estimate the feedback metrics (P_{MB} and P_D) to evaluate the success of an attack. The next step is to dynamically adjust the attack parameters if the predefined goals are not met. Specifically, the Commander adjusts the attack volume V to guarantee the millibottleneck lengths P_{MB} created in all target execution paths are always within the limitation (e.g., 500ms). Meanwhile, based on the feedback metric P_D, the Commander adjusts the number of target critical paths to achieve the predefined damage goal. However, in production systems, many factors (e.g., varied network latency, baseline workload, and drifted system state) can bring uncertainties to the attack framework. All these uncertainties lead to inaccuracy in the estimation and prediction.

To mitigate the negative impact of observing/prediction inaccuracy, we adopt the feedback-based control Kalman filter [30] which is motivated by the implementation in Tail Attack [60]. It takes past measurements into account for the variation of attacking states and reduces the impact of processing noise. We apply the Kalman filter to optimize the estimated P_{MB} and P_D where the original estimation may include noise. The detailed implementation is in Appendix B. In short, using the Kalman filter, the Commander can (a) calibrate the attack parameters at the k-th burst given the historical results of all $(k-i)$-th bursts, (b) dynamically adjust the parameters in the bots, and (c) automatically perform the attack.

To effectively send HTTP requests to the target, we assume that the adversary bots are able to defeat or bypass CAPTCHAs. Many



(a) Experiments topology. **(b) The architecture of SockShop.**

(c) The architecture of SocialNetwork.

Figure 8: Topology of microservice benchmarks.

previous research efforts already provide solutions [22, 62, 68], and are orthogonal to our research. For example, Searles et al. [59] reported that ML-powered bots are faster and more accurate than humans at solving CAPTCHAs.

5 CLOUD EVALUATION

5.1 SyncM Attack in Production Environments

To evaluate the feasibility of our feedback control framework, we deploy two representative real-world open-source microservices benchmarks in three popular cloud platforms under different baseline workloads.

Experimental Setup. The two open-source microservices benchmarks are: (i) SockShop [67], and (ii) SocialNetwork in DeathStarBench [19]. SockShop is an e-commerce website that allows users to view/buy different socks. SocialNetwork implements a broadcast-style social network website with uni-directional follow relationships, where users can create, view, and comment on posts. The two websites contain 11 and 36 unique microservices, respectively, as shown in Fig. 8. We use separate containers to deploy all microservices in Docker Swarm Mode, where each is hosted by one container with dedicated vCPU.

Both websites are deployed on two commercial cloud platforms (EC2 t2.small instances [4], Azure A1V2 instances [44]) and one academic cloud platform (CloudLab c220g1 instances [49]). Each container has 1 vCPU core and 2GB memory, which is considered as the basic computing unit for commercial cloud providers. we dedicate 4 containers for the gateway microservice of both websites to avoid the gateway becoming the bottleneck.

We adopt the RUBBoS workload generator [56] to simulate normal user behaviors as the baseline workload for both websites. Each user follows a Markov chain model to navigate among web pages, with an average 7-second Poisson distributed thinking time between every two consecutive requests. We control the baseline

Table 4: Targeting "SockShop" with the attacking goals $P_{MB} \leq 500ms$ and 95th percentile > 1s; Targeting "SocialNetwork" with the attacking goals $P_{MB} \leq 500ms$ and 95th percentile > 3s.

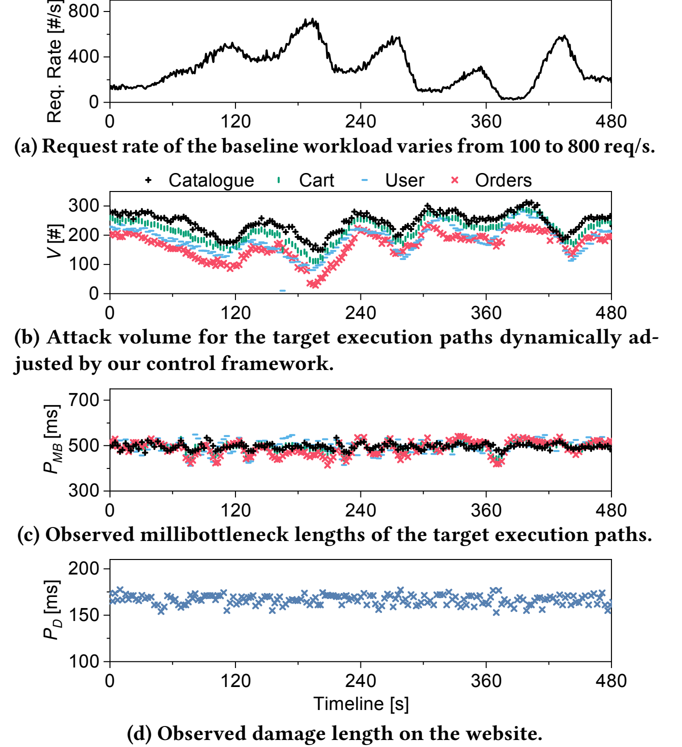
Setting	# of EP.	V (#)	P_{MB} (ms)	Avg. RT (ms)		95ile RT (ms)		99ile RT (ms)		Net. (MB/s)		CPU (%)	
				Base.	Att.	Base.	Att.	Base.	Att.	Base.	Att.	Base.	Att.
EC2-**SockShop**-7K	4	269	479	102	245	131	1415	152	3197	22	28	13	29
EC2-**SockShop**-15K	4	136	473	114	273	137	1561	149	3547	46	51	26	41
Azure-**SockShop**-6K	4	221	489	91	259	124	1433	136	2924	8	13	21	33
Azure-**SockShop**-13K	4	109	515	108	243	131	1429	155	3141	21	26	48	57
CloudLab-**SockShop**-3K	4	213	507	118	276	134	1354	146	3871	7	12	22	35
CloudLab-**SockShop**-9K	4	121	493	127	283	141	1418	151	4041	21	25	49	60
EC2-**SocialNetwork**-7K	6	201	478	144	396	148	3104	161	4847	29	37	31	41
EC2-**SocialNetwork**-12K	6	111	491	143	386	154	3018	169	5132	56	63	49	54
Azure-**SocialNetwork**-4K	6	238	501	155	410	161	3271	175	4757	19	29	25	40
Azure-**SocialNetwork**-9K	6	112	489	157	407	163	3265	177	4871	39	47	51	61
CloudLab-**SocialNetwork**-5K	6	171	491	153	411	159	3396	192	5089	22	30	23	38
CloudLab-**SocialNetwork**-11K	5	90	492	159	423	161	3277	207	4958	47	52	43	54

# of EP.: number of execution path under attack. ***V***: average attack volume per EP.
P_{MB}: average millibottleneck length created by SyncM. **Base.** baseline without attacks. **RT:** response time.
Net.: average network traffic measured at gateway. **CPU:** average CPU usage of a representative bottleneck component.

workload by specifying the number of normal users with two criteria: (1) all the bottleneck resource usage is less than 50% and (2) no long-tail latency problem exists without attack. Meanwhile, in our experiments we use a centralized way [24, 77] to coordinate and synchronize a bot farm on 20 machines (one serves as a centralized controller), synchronized by the NTP service and can achieve millisecond-level synchronization [27][3]. Each bot uses a customized httperf [28] to send attack bursts of HTTP requests, controlled by our framework in Fig. 5.

Overall Results. Tables 4 show the corresponding attack parameters and system resource utilization in real production settings for SockShop and SocialNetwork, respectively. Given that SockShop has fewer independent execution paths available (through profiling) and also the e-commerce websites are typically more latency-sensitive, we set a more strict requirement of system latency that violates SLA for SockShop (95ile RT > 1s) than for SocialNetwork (95ile RT > 3s). At the same time, we keep the same stealthy requirement ($P_{MB} \leq 500ms$) for both websites. Columns 4 and 8 show that the attacker achieves the predefined stealthiness and damage goals (Column 1). Columns 2 and 3 show the attack parameters controlled by our attack framework in each setting. Both the number of execution paths and their corresponding attack volumes need to be adapted according to each cloud setting. Comparing the two applications, attacking SocialNetwork requires more execution paths (6 on average) than Sockshop (4 on average) since the former case requires higher damage latency that violates SLA than the latter. Columns 11 to 14 show SyncM attack consumes less than 10 MB/s additional network bandwidth and 15% additional CPU of the target website to achieve the damage goal. For example, the 95th percentile response time increases over 6 times for Sockshop and 20 times for SocialNetwork under attack. We also conduct a white box analysis with the internal metrics in Appendix C.

Evaluation under bursty baseline workload. To further evaluate the effectiveness of the control framework under workload variations, we conduct experiments of the SockShop deployed in the CloudLab [49], using a real-world "Large Variation" bursty workload trace collected and categorized by Gandhi [20], as shown in Fig. 9a. The baseline workload from normal users varies from 100 to 800 req/s during an 8-minute runtime experiment. Our attack goal keeps the same as those in Table 4: achieving 95ile RT > 1s while P_{MB} < 500ms. Fig. 9b shows the required volume for the four target execution paths dynamically adjusted by the control framework. By changing the volume according to the variation of



(a) Request rate of the baseline workload varies from 100 to 800 req/s.

(b) Attack volume for the target execution paths dynamically adjusted by our control framework.

(c) Observed millibottleneck lengths of the target execution paths.

(d) Observed damage length on the website.

Figure 9: Results of SyncM Attack on SockShop under a real-world "Large Variation" workload trace.

[3]More sophisticated decentralized synchronization approaches are available in [31, 32].

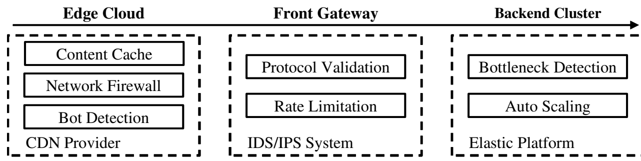


Figure 10: A typical 3-layer DDoS defense strategy

Table 5: Cache rules for SocialNetwork in Cloudflare

Req. Name	Description	Cached
Static contents	HTML, images, etc.	Yes
Register/Login	User credentials	No
Compose post	Create user post	No
Read post	Read user post	Yes
Follow/unfollow	Social connection	No

baseline workload, the observed millibottleneck length is limited to around 500ms as shown in Fig. 9c. Fig. 9d shows the damage length P_D caused by each attack burst observed by the framework, which is about 160ms. Given that the average attack burst interval T is 3 seconds, the overall damage ratio $\rho(T)$ is $0.16/3 \approx 5\%$, thus the 95th percentile response time of normal users is longer than 1 second, reaching the predefined damage goal.

5.2 SyncM Attack under DDoS-Resilient Systems

In this section, we evaluate the stealthiness of SyncM attack on SocialNetwork under the radar of state-of-the-art DDoS-resilient systems as shown in Fig. 10. This figure shows a typical 3-layer DDoS defense strategy adopted by industry practitioners [21, 69], ranging from Edge Cloud CDNs to typical IDS/IPS systems in the gateway, and to auto-scaling strategies adopted by backend systems.

Experimental Setup. In the first defense layer, we adopt the Cloudflare [12] free-tier which aims to filter out potential attack traffic to the backend. We set the caching level to standard, which tries to cache all the static web contents. Cloudflare also provides protection to mitigate DDoS attacks, such as bot detection and network-layer protections (e.g., ACK floods, SYN-ACK amplification).

In the second layer defense, we deploy Snort [63], a rule-based Open-Source IDS/IPS tool, at the gateway to identify abnormal user behavior. We set alert rules following a popular user-behavior model [50] to evaluate whether our attack behavior deviates from it. The user-behavior model analyzed the distribution of normal users' interaction with a web server from a collection of web server logs. Typically, the average inter-request interval per session is less than 10 seconds. The RUBBoS workload generator also models the inter-request interval per session as a Poisson process with an average 7-second thinking time. To set an appropriate threshold for normal clients, we calculate the 95% confidence interval to be (2.8, 14.4). We round the lower bound to 3 to reduce the false-positive, which means that if a client sends two consecutive requests in less than 3 seconds, the client is considered abnormal (i.e., bot).

In the third layer defense, we deploy the backend with AWS EC2 Auto Scaling. We use Amazon CloudWatch (the finest granularity is 1-second [5]) to monitor the instances and manage auto-scale rules on high or low CPU utilization. Specifically, we configure the system to scale up if CPU utilization exceeds 60% for one minute and scale down if CPU utilization drops below 10% for one minute.

Table 6: Rule-based alerts. As the sampling period decreases, legitimate users triggered more false positive errors while bots didn't trigger any alert under all sampling periods.

Alert Rule Parameters		# Alerts Triggered	
Threshold	Sampling Period	Bots	Legitimate Users
1	3s	0	5832
5	15s	0	3417
10	30s	0	928
100*	300s	0	0

*: supported by AWS Shield [61], not effective here

We conduct a 5-minute attack on SocialNetwork with 10k baseline normal users by attacking six independent execution paths during each burst. The millibottleneck lengths are limited to 500ms and the interval time between bursts is set to 3s.

Results. By creating synchronized millibottlenecks at the six execution paths, we successfully penetrate the 3-layer defense shown in Fig 10 and cause 95ile RT of normal users > 1 second. We discuss the results under the 3-layer defense here.

First, Cloudflare CDN does not prevent SyncM attack from achieving its damage goal. This is because CDN is mainly used to cache static content. Table 5 shows CDN cache rules for 5 representative requests in SocialNetwork. Our profiling module in the control framework (Fig. 5) can detect and rank all the requests for dynamic content (e.g., POST requests) as candidate attack requests. However, adopting CDN indeed increases the difficulty to achieve our predefined damage goal since part of normal users' requests are directly served by CDN regardless of the attack. For example, in our SocialNetwork experiment, the overall web requests of normal users served by CDN cache is 68%, suggesting that our attack can only affect the performance of the rest 32% of dynamic requests.

Second, our deployed IDS/IPS system Snort does not trigger any alarm. Columns 1 and 2 in Table 6 depict the threshold and the sampling period configuration, which means that during a sampling period, at most the threshold number of requests can be sent from the same IP. Columns 3 and 4 show the triggered alarms by bots and legitimate users, respectively. The results show that the bots do not trigger any alarms while legitimate users trigger 5832 alarms when the sampling period is 3s. As the sampling period increases, the alerts triggered by legitimate users decrease, indicating that a reasonable sampling period can reduce false positive errors. In practice, to reduce the false positive, the sampling periods are usually set in the order of minutes. For example, the state-of-the-art Cloud provider AWS Shield [61] adopts the rate-based rule with a minimum sampling period of 5 minutes (300s). However, such a long sampling period also gives a SyncM attacker more flexibility to dynamically adjust its attack burst interval to keep stealthy.

Third, the auto-scaling mechanism in EC2 does not take any scaling action during our SyncM attack. This is because CloudWatch (also apply to Azure Monitor [45]) could not detect any millibottlenecks since its finest supported monitor granularity is 1s [5] while the triggered millibottleneck length is within 500ms. AWS Simple Queue Service (SQS) [7] even collects and publishes queue samples of the target system per minute by default. Furthermore, the average CPU utilization of the most heavily used microservice

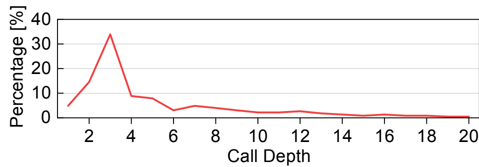


Figure 11: The distribution of call depth in all simulated invocation traces, which contain 1190 unique component microservices and 343 unique execution paths.

under our SyncM attack is less than 50% (over 1-minute), shown in Fig. 1, thus it does not trigger any auto-scaling action based on the pre-defined rules. Therefore, the current auto-scaling mechanisms do not work for our low-volume SyncM attack.

5.3 Large-scale Simulation

To further evaluate the scalability of SyncM attack on large-scale microservices and avoid the ethics problem, we conduct a large-scale simulation using the JMT simulator [8][4] based on a recently released Alibaba invocation trace dataset [41]. The dataset contains over ten billion call traces of production microservice applications in 7 days from the Alibaba cluster. To limit the evaluation time, we analyze a 30-min dataset to build the call graphs among 1190 unique microservices constituting 343 execution paths. The call depths of the execution paths still keep a similar distribution as that in the original dataset, shown in Fig. 11.

For the baseline workload, we use the same workload generator as that in our experiments to simulate a Poisson arrival process (Section 5.1). We use the same attack control framework (Fig. 5) to profile all the independent execution paths and generate the attacking bursts. The stealth goal also keeps the same: $P_{MB} < 500ms$, while we evaluate the effectiveness of our attack under three damage goals (95ile RT > 1s, 3s, and 90ile RT > 3s).

Results. Table 7 shows the damage goals and the attack parameters under two representative baseline workloads: 200k and 600k normal users, under which the average CPU usage of the bottleneck microservices is 13% and 37%, respectively. Compared to the experimental results using small/medium-scale benchmark applications, Column 3 shows that more execution paths are needed to achieve the same damage goal for large microservices applications. For example, 39 independent execution paths are needed to achieve 95ile RT > 3s while only 6 are needed for the "SocialNetwork" application (Table 4). This is because the large-scale application separates the backend microservices into multiple clusters and different clusters have dedicated gateways. The results show the required number of execution paths that achieve corresponding damage goals to the entire system (all clusters). Large-scale production applications typically distribute user traffic geographically based on DNS services to different clusters [13]. In such cases, a SyncM attacker may choose to attack fewer execution paths and cause regional damage or attack more execution paths in all the support regions and cause full damage to the entire target.

On the other hand, Columns 8 and 9 show that only small attack traffic (3% ∼ 24% of baseline) and extra CPU overhead (< 15%) are needed to achieve various damage goals, suggesting significantly

[4]JMT is an open-source suite for modeling Queuing Network computer systems. It is widely used in performance evaluation in distributed systems.

Table 7: Simulation results on large-scale microservices

Normal users (#)	Damage Goal	# of EPs to attack	P_{MB} (ms)	Tail latency (ms) 90ile	95ile	99ile	Norm. traffic	CPU (%)
200k	0 (baseline)	0	0	109	173	211	1	13%
	95ile RT > 1s	33	501	511	1043	2961	1.17	26%
	95ile RT > 3s	43	511	2698	3136	3874	1.21	28%
	90ile RT > 3s	47	509	3027	3440	4127	1.23	29%
600k	0 (baseline)	0	0	121	239	349	1	37%
	95ile RT > 1s	29	491	561	1168	3163	1.05	46%
	95ile RT > 3s	37	512	2719	3034	4077	1.09	48%
	90ile RT > 3s	41	504	3104	3589	4157	1.12	49%

RT: response time. **EP:** execution path. P_{MB} : millibottleneck length.
Norm. traffic: normalized traffic. **CPU:** average CPU usage of victim components.

low attack cost compared to the traditional brute-force DDoS attacks. In addition, comparing the two baseline workload cases (200k vs. 600k), the simulation results also show that it is easier to achieve the damage goal when the baseline workload is high. This is because high baseline workloads require less attack traffic to trigger the same length millibottlenecks. This underscores the vulnerability for potential attackers to target prominent e-commerce websites during peak shopping events, such as Black Friday.

Overall, our simulation results validate that SyncM attack is practical and scalable on large-scale microservices. To cause more damage, the attackers need to select more independent execution paths. By adjusting and coordinating a number of execution paths to attack, attackers can dynamically set different damage goals while maintaining a high degree of stealthiness.

6 POSSIBLE DETECTION AND DEFENSE

Detecting and defending SyncM attack is challenging because it is difficult to accurately separate the attack requests from normal ones. Tail Attack [60] proposes a statistic-based solution to address this challenge by correlating critical resource usage spikes with suspicious requests. Here, we adopt their detection and defense strategies and apply them in microservices based on the unique characteristics of SyncM attacks. We focus more on discussing the limitations of the proposed strategies, with the hope to shed light on more sophisticated solutions in the future.

Step 1: Millibottleneck cluster detection. The stealthiness of SyncM attack relies on the very short duration of millibottlenecks on the target bottleneck components. Since each burst may cause multiple synchronized millibottlenecks (referred as millibottleneck clusters), the first step is to detect the system-wide millibottleneck clusters using fine-grained monitoring tools. Fig. 12a shows the detection of synchronized millibottleneck clusters at four microservice components in one SockShop experiment with 3500 legitimate users, using 100ms monitoring granularity.

Step 2: Suspicious request identification. Once we detect a synchronized millibottleneck cluster using fine-grained monitoring tools, the millibottleneck cluster should match back to an attacking burst. Fig. 12b shows the real-time throughput measured from the gateway microservice. By correlating the occurrence of millibottleneck clusters in Fig. 12a with the throughput spikes in Fig. 12b, we can infer the potential attack requests sent by SyncM attacker.

Step 3: Bots confirmation. After identifying *suspicious requests*, the next step is to trace their IPs and discriminate the bots from

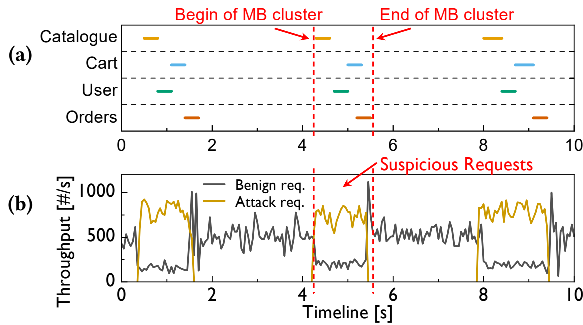


Figure 12: Correlation between millibottleneck clusters and suspicious requests.

normal users. In our attack scenario, the attacker aims to send attack requests during the short millibottleneck cluster period while keeping quiet during other periods to maintain a low access rate for stealthy. Following this pattern, if we find IPs specifically send requests during the millibottleneck cluster periods while keeping quiet during other periods, we suspect such IPs are owned by bots.

Limitations. While promising, the above defense workflow mainly has three limitations. ① It heavily relies on the capability of fine-grained monitoring tools that can reliably detect millibottlenecks caused by SyncM attack. However, it is well-known that fine-grained resource monitoring (especially millisecond level) brings non-trivial overhead, which explains why the default coarse granularity of modern cloud monitoring tools such as Amazon CloudWatch [5], AWS Simple Queue Service (SQS) [7], and Azure Monitor [45] is from 10-second to 1-minute. ② While CPU contention is a typical source of millibottlenecks in microservices [54], millibottlenecks may result from many other sources [17] that may not be detectable by existing fine-grained resource monitoring tools (e.g., collectl [14]). Thus, designing appropriate new resource monitors (e.g., lock contention) for identifying new causes of millibottlenecks represents a potential solution to defend against SyncM attack targeting different resources. ③ One hidden assumption of the defense workflow is that the bots only send attack requests during the short "ON" periods while keeping quiet during the long "OFF" periods. However, bots may attempt to send requests outside the "ON" periods to hide their intention in practice. Thus, we need to further investigate the bots detection techniques that can identify the bots' behavior patterns, which is for our future research.

7 RELATED WORK

In this section, we review the most relevant work on low-volume application layer attacks and defense techniques.

Low-volume Application layer DDoS Attacks. DDoS attacks in this category focus on disrupting normal users' services by a small number of application-level attack requests [9, 10, 18, 24, 25, 47, 52, 55]. Some representative work includes: Regular Expression Denial of Service (ReDoS) [64] searches for malicious input to match a regular expression that takes unexpectedly long. Warmonger [72] exploits limited IPs in serverless platforms to cause IP-blockages of benign functions.

The closest to ours are LoRDAS [42] and Tail Attack [60], which attack the target web service with regular ON/OFF strike waveform. LoRDAS exploits the limited queue size of the frontend web server. It aims to always occupy all the available queue slots (e.g., server threads) of the target web server by sending bursts of attacking requests at carefully crafted times to seize newly released queue slots. Tail Attack only targets the monolithic n-tier architecture. The hidden assumption of Tail Attack is that the performance anomalies of one tier will degrade the entire system performance. With this assumption, the Tail Attack only targets a single tier/component of the system, which does not work on microservices. This is because a microservices application is split into a large number of loosely-coupled microservice components; attacking any individual does not cause any significant damage to the overall system performance. Instead, our SyncM attack exploits the vulnerability of multiple synchronized cross-service millibottlenecks on carefully chosen execution paths, which balances well between the attack damage and the attack stealthiness.

Detecting/Defending Application DDoS Attacks. Most recent research efforts focus on filtering out the malicious traffic from the normal traffic [6, 39, 40, 73–75]. Some representative work includes: Rampart [43] detects CPU-exhaustion DoS attacks using statistical methods and function-level program profiling. Li et al. [37] propose a mathematical model based on queuing theory to scrutinize the behavior of suspicious requests and prioritize the resource allocation for benign requests. However, all these solutions mainly target DDoS aiming for a shutdown of the target and monitor resource usage using normal granularity (e.g., in seconds) tools, thus, they are likely to fail to detect our SyncM attack which exploits millibottlenecks (in milliseconds) in microservices.

8 CONCLUSION

We introduce the SyncM Attack, a novel low-volume performance attack that targets microservices cloud architecture with high stealthiness. We show that multiple synchronized millibottlenecks along different execution paths can cause a superimposed queuing effect in the shared gateway, leading to severe performance damage that violates the SLA for typical e-commerce. To numerically analyze the SyncM attack scenario, we model the attack based on a well-tuned queuing network which helps derive optimal attack parameters given pre-defined damage and stealthiness goals. We also designed a feedback control framework that allows attackers to dynamically optimize attack parameters to fit the baseline workload and system state variations. Our experiments, both real cloud-based experiments and large-scale simulations, demonstrate that the SyncM Attack can achieve a wide range of damaging goals (e.g., 95ile response time > 1 or 3 seconds) while remaining undetected by state-of-the-art DDoS defense tools. Overall, our research provides a valuable contribution by advancing our understanding of novel and stealthy application layer DDoS attacks.

9 ACKNOWLEDGEMENTS

This research has been partially funded by the National Science Foundation by CNS (2000681), CNS (2235231), and contracts from Fujitsu Limited, Cisco Research Funding. Any opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] Gunes Acar, Marc Juarez, Nick Nikiforakis, Claudia Diaz, Seda Gürses, Frank Piessens, and Bart Preneel. 2013. FPDetective: dusting the web for fingerprinters. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1129–1140.

[2] Akamai. 2020. Credential Stuffing in the Media Industry. https://www.akamai.com/newsroom/press-release/state-of-the-internet-security-credential-stuffing-in-the-media-industry.

[3] Firas Al-Doghman, Nour Moustafa, Ibrahim Khalil, Zahir Tari, and Albert Zomaya. 2022. AI-enabled secure microservices in edge computing: Opportunities and challenges. *IEEE Transactions on Services Computing* (2022).

[4] AWS. 2022. Amazon EC2. https://aws.amazon.com/ec2/.

[5] AWS. 2022. *AWS CloudWatch Custom Metrics*. Amazon AWS. https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/publishingMetrics.html.

[6] Ataollah Fatahi Baarzi, George Kesidis, Dan Fleck, and Angelos Stavrou. 2020. Microservices made attack-resilient using unsupervised service fissioning. In *Proceedings of the 13th European workshop on Systems Security*. 31–36.

[7] Jeff Barr. 2014. Rapid Auto Scaling with Amazon SQS. https://aws.amazon.com/blogs/aws/auto-scaling-with-sqs/.

[8] Marco Bertoli, Giuliano Casale, and Giuseppe Serazzi. 2009. JMT: performance engineering tools for system modeling. *SIGMETRICS Perform. Eval. Rev.* 36, 4 (2009), 10–15. https://doi.org/10.1145/1530873.1530877

[9] Enrico Cambiaso, Gianluca Papaleo, and Maurizio Aiello. 2012. Taxonomy of slow DoS attacks to web applications. In *International Conference on Security in Computer Networks and Distributed Systems*. Springer, 195–204.

[10] Jiahao Cao, Qi Li, Renjie Xie, Kun Sun, Guofei Gu, Mingwei Xu, and Yuan Yang. 2019. The {CrossPath} Attack: Disrupting the {SDN} Control Channel via Shared Links. In *28th USENIX Security Symposium (USENIX Security 19)*. 19–36.

[11] Ericka Chickowski. 2022. Why haven't DDoS attacks gone away? https://www.hpe.com/us/en/insights/articles/why-havent-ddos-attacks-gone-away-2202.html.

[12] Cloudflare. 2022. Cloudflare CDN: A fast, agile, and secure global network. https://www.cloudflare.com/cdn/.

[13] CloudFlare. 2023. What is DNS-based load balancing? https://www.cloudflare.com/learning/performance/what-is-dns-load-balancing/.

[14] Collectl. 2022. Collectl. https://collectl.sourceforge.net/.

[15] corero. 2017. Short, Stealthy, Sub-Saturating DDoS Attacks Pose Greatest Security Threat to Businesses. https://www.businesswire.com/news/home/20170605005149/en/Short-Stealthy-Sub-Saturating-DDoS-Attacks-Pose-Greatest.

[16] J. Dean and L. Barroso. 2013. The tail at scale. *Commun. ACM* 56 (2013), 74–80.

[17] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56 (2013), 74–80. http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[18] Massimo Ficco and Massimiliano Rak. 2014. Stealthy denial of service strategy in cloud computing. *IEEE transactions on cloud computing* 3, 1 (2014), 80–94.

[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 3–18.

[20] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. 2012. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (2012), 1–26.

[21] Yossi Gilad, Amir Herzberg, Michael Sudkovitch, and Michael Goberman. 2016. CDN-on-Demand: An affordable DDoS Defense via Untrusted Clouds.. In *NDSS*.

[22] Ian J Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. 2013. Multi-digit number recognition from street view imagery using deep convolutional neural networks. *arXiv preprint arXiv:1312.6082* (2013).

[23] gRPC. 2023. gRPC: A high performance, open source universal RPC framework. https://grpc.io/.

[24] Mina Guirguis, Azer Bestavros, and Ibrahim Matta. 2004. Exploiting the transients of adaptation for RoQ attacks on Internet resources. In *Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004.* IEEE, 184–195.

[25] Robert Hansen and John Kinsella. 2009. Slowloris HTTP DoS. https://web.archive.org/web/20090822001255/http://ha.ckers.org/slowloris/.

[26] Caroline Harting. 2022. How Can We Break the Cycle of Focusing on Negative Experiences? https://news.columbia.edu/news/how-can-we-break-cycle-focusing-negative-experiences.

[27] Sabrina Hiller. 2015. Precise to the millisecond: NTP services in the "Internet of Things". https://www.retarus.com/blog/en/precise-to-the-millisecond-ntp-services-in-the-internet-of-things/.

[28] httperf. 2022. The httperf HTTP load generator. https://github.com/httperf/httperf.

[29] Jaeyeon Jung, Balachander Krishnamurthy, and Michael Rabinovich. 2002. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *Proceedings of the 11th International Conference on World Wide Web*. ACM, 293–304.

[30] Rudolph Emil Kalman. 1960. A new approach to linear filtering and prediction problems. (1960).

[31] Min Suk Kang, Soo Bum Lee, and Virgil D Gligor. 2013. The crossfire attack. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'13)*. IEEE, San Francisco, CA, USA, 127–141.

[32] Yu-Ming Ke, Chih-Wei Chen, Hsu-Chun Hsiao, Adrian Perrig, and Vyas Sekar. 2016. CICADAS: Congesting the Internet with Coordinated and Decentralized Pulsating Attacks. In *Proceedings of the 11th ACM on ASIACCS*. 699–710.

[33] Leonard Kleinrock. 1975. Theory, volume 1, Queueing systems.

[34] Ron Kohavi, Randal M Henne, and Dan Sommerfield. 2007. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *Proceedings of the ACM international conference on Knowledge discovery and data mining*. 959–967.

[35] Ron Kohavi and Roger Longbotham. 2007. Online experiments: Lessons learned. *Computer* 40, 9 (2007), 103–105.

[36] Aleksandar Kuzmanovic and Edward W Knightly. 2003. Low-rate TCP-targeted denial of service attacks: the shrew vs. the mice and elephants. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. 75–86.

[37] Zhi Li, Hai Jin, Deqing Zou, and Bin Yuan. 2019. Exploring new opportunities to defeat low-rate DDoS attack in container-based cloud environment. *IEEE Transactions on Parallel and Distributed Systems* 31, 3 (2019), 695–706.

[38] Timothy Libert. 2015. Exposing the hidden web: An analysis of third-party HTTP requests on 1 million websites. *arXiv preprint arXiv:1511.00619* (2015).

[39] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: detecting and exploiting regular expression denial-of-service vulnerabilities. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1468–1484.

[40] Zaoxing Liu, Hun Namkung, Georgios Nikolaidis, Jeongkeun Lee, Changhoon Kim, Xin Jin, Vladimir Braverman, Minlan Yu, and Vyas Sekar. 2021. Jaqen: A {High-Performance}{Switch-Native} Approach for Detecting and Mitigating Volumetric {DDoS} Attacks with Programmable Switches. In *30th USENIX Security Symposium (USENIX Security 21)*. 3829–3846.

[41] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing*. 412–426.

[42] Gabriel Maciá-Fernández, Jesús E Díaz-Verdejo, Pedro García-Teodoro, and Francisco de Toro-Negro. 2007. LoRDAS: A low-rate DoS attack against application servers. In *International Workshop on Critical Information Infrastructures Security*. Springer, 197–209.

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

[44] Microsoft. 2022. Microsoft Azure. https://azure.microsoft.com/en-us/.

[45] Microsoft. 2023. Collect Windows and Linux performance data sources with Log Analytics agent. https://learn.microsoft.com/en-us/azure/azure-monitor/agents/data-sources-performance-counters.

[46] Seyed Ali Mirheidari, Sajjad Arshad, Kaan Onarlioglu, Bruno Crispo, Engin Kirda, and William Robertson. 2020. Cached and confused: Web cache deception in the wild. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 665–682.

[47] Hoai Viet Nguyen, Luigi Lo Iacono, and Hannes Federrath. 2019. Your cache has fallen: Cache-poisoned denial-of-service attack. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1915–1936.

[48] Brian J Odelson, Murali R Rajamani, and James B Rawlings. 2006. A new autocovariance least-squares method for estimating noise covariances. *Automatica* 42, 2 (2006), 303–308.

[49] The University of Utah. 2023. CloudLab. https://www.cloudlab.us/.

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

[51] Nelli Klepfish Pamela Weaver. 2021. Shorter, sharper DDoS attacks are on the rise – and attackers are sidestepping traditional mitigation approaches. https://www.imperva.com/blog/shorter-sharper-ddos-attacks-are-on-the-rise-and-attackers-are-sidestepping-traditional-mitigation-approaches/.

[52] Junhan Park, Keisuke Iwai, Hidema Tanaka, and Takakazu Kurokawa. 2014. Analysis of slow read DoS attack. In *2014 International Symposium on Information Theory and its Applications*. IEEE, 60–64.

[53] PhantomJS. 2021. PhantomJS. http://phantomjs.org/.

[54] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 805–825.

[55] Ryan Rasti, Mukul Murthy, Nicholas Weaver, and Vern Paxson. 2015. Temporal lensing and its application in pulsing denial-of-service attacks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P'15)*. IEEE, San Jose, CA, USA, 187–198.
[56] rubbos. 2021. RUBBoS: Bulletin Board Benchmark. https://projects.ow2.org/view/rubbos/.
[57] Scikit-learn. 2023. Scikit learn one class SVM. https://scikit-learn.org/stable/modules/generated/sklearn.svm.OneClassSVM.html.
[58] Bryan Payne Scott Behrens. 2017. Starting the Avalanche Application DDoS In Microservice Architectures. https://netflixtechblog.com/starting-the-avalanche-640e69b14a06.
[59] Andrew Searles, Yoshimichi Nakatsuka, Ercan Ozturk, Andrew Paverd, Gene Tsudik, and Ai Enkoji. 2023. An Empirical Study & Evaluation of Modern {CAPTCHAs}. In *32nd USENIX Security Symposium (USENIX Security 23)*. 3081–3097.
[60] Huasong Shan, Qingyang Wang, and Calton Pu. 2017. Tail attacks on web applications. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 1725–1739.
[61] AWS Shield. 2019. Lower Threshold for AWS WAF Rate-based Rules. https://aws.amazon.com/about-aws/whats-new/2019/08/lower-threshold-for-aws-waf-rate-based-rules/. Accessed: 2021.
[62] Suphannee Sivakorn, Iasonas Polakis, and Angelos D Keromytis. 2016. I am robot:(deep) learning to break semantic image captchas. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 388–403.
[63] snort. 2023. Snort - Network Intrusion Detection. https://www.snort.org/.
[64] Cristian-Alexandru Staicu and Michael Pradel. 2018. Freezing the web: A study of redos vulnerabilities in javascript-based web servers. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 361–376.
[65] Tyler Terat. 2023. Everything You Know About Latency Is Wrong. https://bravenewgeek.com/everything-you-know-about-latency-is-wrong/.
[66] Alethea Toh. 2022. Azure DDoS Protection—2021 Q3 and Q4 DDoS attack trends. https://azure.microsoft.com/en-us/blog/azure-ddos-protection-2021-q3-and-q4-ddos-attack-trends/.
[67] Weaveworks. 2022. Sock Shop: A Microservices Demo Application. https://microservices-demo.github.io/.
[68] Haiqin Weng, Binbin Zhao, Shouling Ji, Jianhai Chen, Ting Wang, Qinming He, and Raheem Beyah. 2019. Towards understanding the security of modern image captchas and underground captcha-solving services. *Big Data Mining and Analytics* 2, 2 (2019), 118–144.
[69] Holly Willey. 2017. How to Help Protect Dynamic Web Applications. https://aws.amazon.com/blogs/security/how-to-protect-dynamic-web-applications-against-ddos-attacks-by-using-amazon-cloudfront-and-amazon-route-53/.
[70] Guowu Xie, Huy Hang, and Michalis Faloutsos. 2014. Scanner hunter: Understanding http scanning traffic. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 27–38.
[71] Yi Xie and Shunzheng Yu. 2009. Monitoring the Application-Layer DDoS Attacks for Popular Websites. *IEEE/ACM Transactions on Networking* 17, 1 (2009), 15–25. https://doi.org/10.1109/TNET.2008.925628
[72] Junjie Xiong, Mingkui Wei, Zhuo Lu, and Yao Liu. 2021. Warmonger: Inflicting Denial-of-Service via Serverless Functions in the Cloud. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 955–969.
[73] Guangliang Yang, Jeff Huang, Guofei Gu, and Abner Mendoza. 2018. Study and mitigation of origin stripping vulnerabilities in hybrid-postmessage enabled mobile applications. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 742–755.
[74] Chengxu Ye and Kesong Zheng. 2011. Detection of application layer distributed denial of service. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, Vol. 1. IEEE, 310–314.
[75] Menghao Zhang, Guanyu Li, Shicheng Wang, Chang Liu, Ang Chen, Hongxin Hu, Guofei Gu, Qianqian Li, Mingwei Xu, and Jianping Wu. 2020. Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*.
[76] Shungeng Zhang, Huasong Shan, Qingyang Wang, Jianshu Liu, Qiben Yan, and Jinpeng Wei. 2019. Tail amplification in n-tier systems: a study of transient cross-resource contention attacks. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 1527–1538.
[77] Ying Zhang, Zhuoqing Morley Mao, and Jia Wang. 2007. Low-Rate TCP-Targeted DoS Attack Disrupts Internet Routing.. In *NDSS*. Citeseer.

A DERIVATION OF QUEUE OVERFLOW

Here we introduce the detailed derivation of queue overflow in a single path during an attacking burst. For all microservices ($i = 2, ..., n$) along the execution path under attack, the time needed to fill up each queue is:

$$l_n = \frac{Q_n}{\lambda_n + B - C_{n,A}} \tag{13}$$

$$l_{n-1} = \frac{Q_{n-1} - Q_n}{\lambda_{n-1} + \lambda_n + B - C_{n,A}} \tag{14}$$

$$\dots$$

$$l_2 = \frac{Q_2 - Q_3}{\sum_{i=2}^{n} \lambda_i + B - C_{n,A}} \tag{15}$$

L is the total attack length during a burst. l_n denotes the time needed to fill up the queue in the n-th component where the millibottleneck occurs. $(\lambda_n + B - C_{n,A})$ is the queue fill-up rate. Once the queue of n-th component fills up, requests start to queue in its direct upstream (n-1)-th component, and so on to cause queue propagation to the frontend gateway. (l_{n-1}) denotes the time to fill up the queue in the (n-1)-th component. Remember every queued request in the n-th component holds a pending request queue slot of its upstream component due to the RPC style call/response. Thus, when the n-th component is full, the available queue slots of the (n-1)-th component is $(Q_{n-1} - Q_n)$. The queue fill-up rate in (n-1)-th component is $(\lambda_{n-1} + \lambda_n + B - C_{n,A})$. This is because of two factors: (1) all the requests arriving at a downstream component need to go through every upstream component; (2) the overall service rate of the entire execution path is determined by the capacity service rate of the bottleneck component, which is $C_{n,A}$.

B THE IMPLEMENTATION OF KALMAN FILTER

As introduced in Section 3, both P_D and P_{MB} have a linear relationship with the attack volume V when we fix the attack rate B. The linear relationships provide us with a firm theoretical foundation for our control framework. Therefore, in control algorithm 1, once we find the appropriate attacking rate B, we can tune the other attacking parameters with the Kalman filter. For each group of attack requests attacking one component, there is one measurement of millibottleneck length. Let Z_k be the measurement of millibottleneck length P_{MB} in k-th burst. In Eqn. 8, P_{MB} is a linear function of the volume V of the group of requests. We define the process model of the evolution of the state from time k-1 to time k as

$$x_k = Fx_{k-1} + Bu_k + v_k \tag{16}$$

where F is the state transition matrix applied to the previous state vector x_{k-1}, B is the control-input matrix applied to the control vector u_k, and v_k is the process noise vector that is assumed to be zero-mean Gaussian with covariance Q_1. The measurement model that describes the relationship between the state and the measurement at the current time step k is

$$z_k = Hx_k + w_k \tag{17}$$

where H is the measurement matrix, and w_k is the measurement noise vector that is assumed to be another zero-mean Gaussian distribution with covariance Q_2. Given the series of measurement $z_1, z_2, ...$, and the information described by F, B, H, and Qs, Kalman Filter can provide estimation of x_k at time k. Denoting $\widehat{x}(k|k-1)$ as an estimate of x at time k given the history observations to time $k-1$. We show the key steps in Kalman Filter as

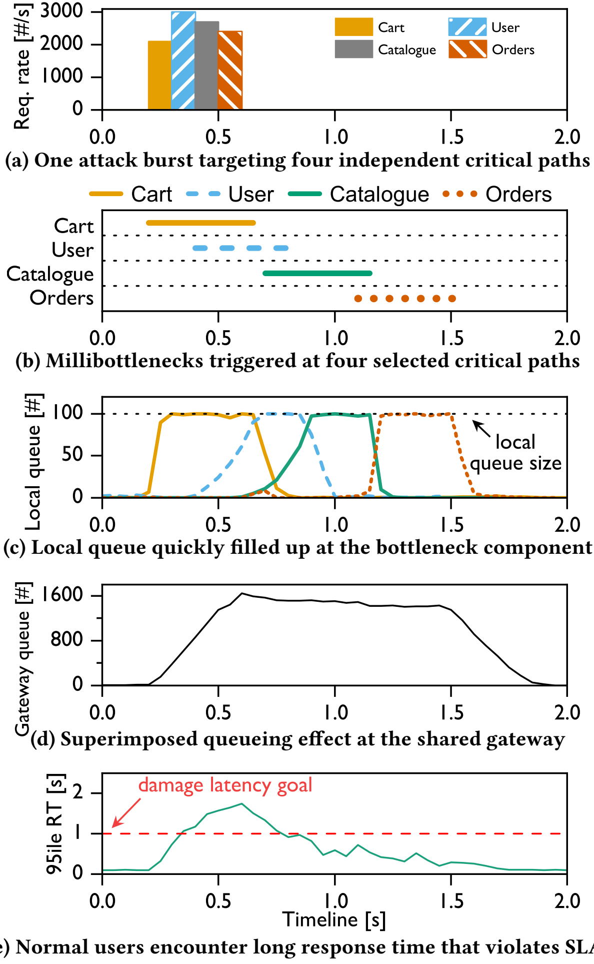


Figure 13: White box timeline analysis illustrating the sequence of events when the attacker attacks SockShop with the setting "EC2-SS-15K" from Table 4.

Predict:

$$\widehat{x}(k|k-1) = F\widehat{x}(k-1|k-1) + Bu_{k-1} \tag{18}$$

$$P(k|k-1) = FP(k-1|k-1)F^T + Q_1 \tag{19}$$

Update:

$$K_k = P(k|k-1)H^T(Q_2 + HP(k|k-1)H^T)^{-1} \tag{20}$$

$$\widehat{x}(k|k) = \widehat{x}(k|k-1) + K_k(z_k - H\widehat{x}(k|k-1)) \tag{21}$$

$$P(k|k) = (1 - K_k H)P(k|k-1) \tag{22}$$

where $\widehat{x}(k|k-1)$ is the priori estimate and $\widehat{x}(k|k)$ is the posteriori estimate, $P(k|k-1)$ is the priori error covariance and $P(k|k)$ is the posteriori error covariance. The noise covariance $Q-1$ and Q_2 are the tuning parameters that attackers can adjust to get the desired performance. In practice, attackers can estimate the two noise covariances with mathematical tools (e.g., autocovariance least-squares method [48]).

C WHITE BOX ANALYSIS UNDER ATTACK.

To better understand the internal impact of the SyncM attack on the target system, we conduct a white box analysis to illustrate the sequence of events that occurs during a Short-ON period as described in Section 2.3 (Event1~Event5). Fig 13 illustrates the sequence of events when the attacker attacks SockShop with the setting "EC2-SocialNetwork-15K" from Table 4. **(Event1)** Fig. 13a shows the attacker sends a burst of 4-type attack requests to the target system over a very short period of time. **(Event2)** Fig. 13b shows each type of attack requests creates a millibottleneck on its corresponding execution path of the target system. **(Event3)** Fig. 13c shows the millibottleneck in each target execution path quickly fills up the local queue (we set the size to be 100) at the bottleneck component. **(Event4)** Fig. 13d shows queued requests from all target execution paths converge at their shared frontend gateway microservice, leading to a superimposed queuing effect in the gateway (e.g., queued requests over 1600). **(Event5)** Fig. 13e shows requests from normal users encounter long response time that violates SLA ($t_{SLA} = 1s$ here), due to the interference of the superimposed queuing effect in the gateway. The white box analysis clearly shows the attack burst is well controlled by our framework, and both our attack goals (P_D, P_{MB}) are achieved.