Grunt Attack: Exploiting Execution Dependencies in Microservices

Xuhang Gu† Qingyang Wang† Jianshu Liu† Jinpeng Wei‡
†Louisiana State University ‡University of North Carolina at Charlotte

***Abstract*—Loosely-coupled and lightweight microservices running in containers are likely to form complex execution dependencies inside the system. The execution dependency arises when two execution paths partially share component microservices, resulting in potential runtime blocking effects. In this paper, we present Grunt Attack – a novel low-volume DDoS attack that takes advantage of the execution dependencies of microservice applications. Grunt Attack utilizes legitimate HTTP requests to accurately profile the internal pairwise dependencies of all supported execution paths in the target system. By grouping and characterizing all the execution paths based on their pairwise dependencies, the Grunt attacker can target only a few execution paths to launch a low-volume DDoS attack that achieves large performance damage to the entire system. To increase the attack stealthiness, the Grunt attacker avoids creating a persistent bottleneck by alternating the target execution paths within their dependency group.**

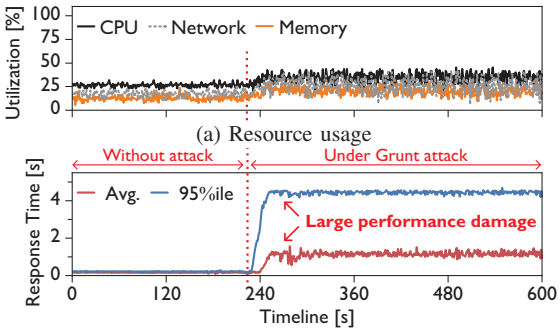
We validate the effectiveness of Grunt attack through experiments of open-source microservices benchmark applications on real clouds (e.g., EC2, Azure) equipped with state-of-the-art IDS/IPS systems and live attack scenarios. Our results show that Grunt attack consumes less than 20% additional CPU resource of the target system while increasing its average response time by over 10x.

***Index Terms*—Microservices, DDoS attack, SLA violations**

I. Introduction

Web applications are increasingly to have stringent latency requirements. Many of these applications, like those at Netflix [42], Twitter [19], and Amazon [50], are user-facing, latency-critical services that must maintain strict Service Level Agreement (SLA) [20]. For example, a study by Amazon [34] reported that every increase of 100 milliseconds in page loading time is correlated to roughly 1% loss in sales. Similarly, Google found that a 500ms additional delay in returning search results could reduce revenues by up to 20% [33].

Meanwhile, web application architecture is gradually evolving from the traditional monolithic multi-tier-based to loosely-coupled and lightweight microservices [38]. This trend is due to the special advantages of the microservice architecture in many aspects, such as fine-grained scalability, cross-team development, friendly deployment, etc. However, decomposing the originally monolithic architecture into hundreds to thousands of fine-grained microservices creates complex internal communication dependencies among component microservices, causing significant challenges for performance prediction and management [38], [46]. For example, to manage performance and reason about system behavior, Google's recent paper [22] discussed how to explicitly track and control microservice dependencies. The combination of stringent latency requirements and the complexity of internal dependencies creates new vulnerabilities for attackers to exploit for novel performance attacks.



(a) Resource usage

(b) End-to-end response time of all legitimate users

Fig. 1: System bottleneck resource utilization and response time under Grunt attack. Metrics are collected every 1 second.

In this paper, we present a novel low-volume DDoS attack—Grunt attack—on microservices. The goal of the attack is to cause significant performance damage (i.e., violate the typical service-level agreement (SLA) for e-commerce) on the target microservices application while keeping stealthy from the state-of-the-art IDS/IPS systems. Grunt attack exploits the dependencies of the runtime execution paths inside the system. A typical execution path is triggered by an incoming HTTP request, which traverses through a series of different microservices to accomplish a transaction. For example, the order execution path may involve inventory, pricing, and credit card processing component services. The dependency between execution paths arises when they share some component microservices. A recently released Alibaba trace [38] shows that 5% microservices (called "hotspot" microservices) are shared by 90% execution paths in their application, showing that execution dependency widely exists in a production system.

The pairwise dependency can be profiled through performance interference analysis by sending two types of requests simultaneously (see Section IV-C). Thus, all the execution paths can be divided into multiple performance dependency groups. There exists dependency among execution paths within a dependency group while execution paths across dependency groups have no dependency detected. Our hypothesis is that attacking a few execution paths within a dependency group will affect the performance of the entire group. By targeting only a few execution paths from each dependency group, Grunt attack can block/degrade the performance of the entire system.

2158-3927/24/$31.00 ©2024 IEEE
DOI 10.1109/DSN58291.2024.00025

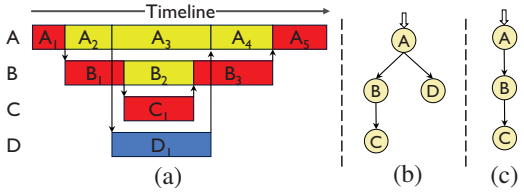


Fig. 2: (a) Execution history graph of a request with root span A, its children B and D. B further calls C. The red blocks represent the critical path through the execution. (b) Dependency graph of the request. (c) Dependency graph along the critical path.

Compared to the well-understood brute-force DDoS attacks, low-volume attacks are reported [7], [18], [45], [56] to pose a greater security threat for business since these attacks tend to go undetected while the damage persists for a long time. Some well-known low-volume attacks includes Shrew attack [35] and Slowloris [41] that exploit protocol weaknesses, low-rate network layer DDoS attacks [26], [31], [32], [35], [40], and flash crowds [29], [48]. The novelty of our Grunt attack is on the exploitation of the new architecture-level weaknesses of microservices: the complex internal dependencies among execution paths, which enable a low-volume DDoS attack (a few to tens of megabytes) that requires orders of magnitude less traffic volume compared to traditional brute-force DDoS attacks (in gigabytes [10]).

The most challenging task for launching an effective Grunt attack is to design an attacking strategy for each dependency group so that the attack can achieve the desired performance damage (e.g., average latency $>$ 1s) while keeping stealthy from typical DDoS defense tools. To achieve this goal, we first build a dependency model that characterizes the pairwise dependency between every two execution paths based on the location of their bottleneck microservice along each path: parallel dependency and sequential dependency. Based on each pairwise dependency type, we design an alternating attack strategy by sending bursts of attacking requests to each execution path in turn so that the damage of each attacking burst can be accumulated while the triggered millibottleneck length in each execution path keeps short (e.g., $\leq$ 500ms), avoiding detection by normal IDS/IPS tools. To fit the dynamics of background workload and system state in real cloud environments, we develop a feedback control framework that can fast adapt attack parameters to each dependency group. With the guide of the proposed dependency model and the feedback control framework, we validate that Grunt attack not only achieves the expected latency damage goal but also escapes the state-of-the-art DDoS detection mechanisms (Fig. 1 as an example). In brief, we made the following contributions:

- The first low-volume DDoS attack towards microservices by exploiting internal dependencies among execution paths;
- A dependency model that characterizes different dependencies between execution paths. The model helps quantify the impact of our attack based on queuing network theory;
- Practical approaches that help attackers identify the internal execution dependencies among different execution paths;
- A feedback control framework that allows attackers to dynamically fit the variation of background workload and system state; and
- Extensive real-world experiments that validate the practicality of Grunt attack in production clouds (e.g., EC2).

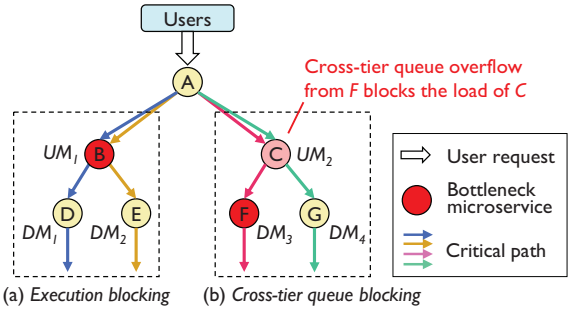


Fig. 3: Blocking effects among different critical paths. We use a dependency graph along the critical path to visualize the critical path of a user request as in Fig. 2(c).

II. Motivation & Threat Model

A. Blocking Effects in Microservice Calls

Microservice architecture introduces complex inter-service execution dependencies by decomposing the monolithic design of business logic into loosely coupled microservices, which leads to a significant challenge in system performance: the computational resource saturation in one service can block the execution of many other services, leading to *blocking effects* that this paper refers to. As a result, overloading only a few well-chosen microservices can substantially degrade the performance of an entire system. This motivates our work on designing a stealthy low-volume DDoS attack.

1) Critical path of a user request: A user request, after being sent to a gateway/entry service, triggers a series of calls between related microservices. This can be represented as either an *execution history graph* or a *dependency graph*. For example, Fig. 2(a) shows the execution history graph of a user request, and Fig 2(b) depicts its dependency graph. The *critical path* is the longest chain of dependent tasks within a dependency graph, which dominates the latency of the request. In this paper, we extract the critical path of a dependency graph to represent the execution of a request, as shown in Fig. 2(c).

2) Blocking effects within a critical path: Each call in a dependency graph links two microservices: an *upstream microservice* (UM) and a *downstream microservice* (DM). Due to the execution of dependent tasks in a critical path, a microservice may not perform the next task (e.g., call its DM or reply to its UM) until receiving replies from DM. Therefore, resource saturation on a microservice can block the execution of itself and its UM and DM along the critical path.

3) Blocking effects across multiple critical paths: Blocking effects can propagate across multiple critical paths, because critical paths of different user requests may have overlapped microservices (i.e., microservices shared by different user

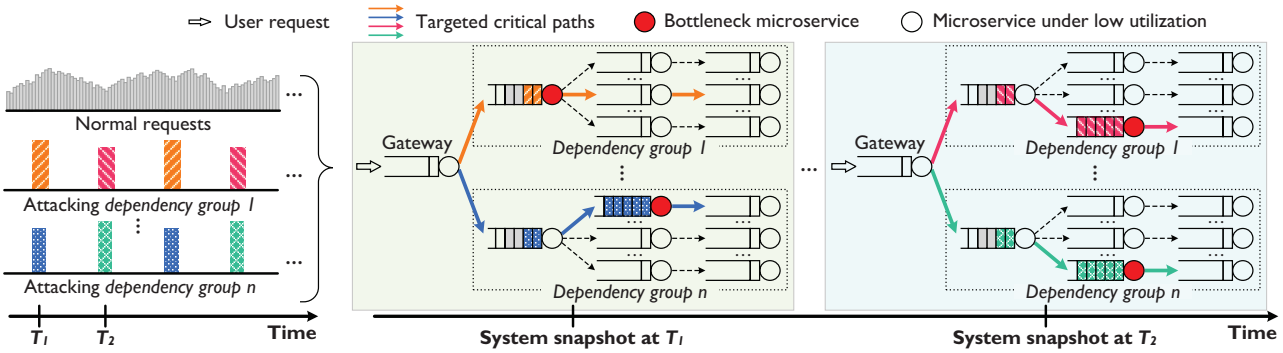


Fig. 4: Grunt attack scenario and system modeling. Attack requests trigger millibottlenecks alternatively among different critical paths in a dependency group, causing persistent blocking effects that block the execution of all other paths in the same dependency group, resulting in large response time problem.

requests), which is common in microservice architecture [38]. Firstly, when resource saturation occurs on an overlapped microservice shared by multiple critical paths, it blocks both its own request processing and the calls to its DMs. As a result, the execution of all the critical paths that share the bottleneck microservice will be blocked during resource saturation, which is called *execution blocking effect*. For instance, in Fig. 3(a), if microservice-B experiences resource saturation, it will block the load of microservice-D and microservice-E, even though they belong to different critical paths.

Additionally, when resource saturation occurs on a local microservice of a critical path (i.e., not shared by other critical paths), it can also block the execution of other critical paths. This is because the resource saturation on the bottleneck microservice will cause message queues and thread pools to fill up. Once the local queues reach capacity, requests start to overflow to the UMs, leading to a phenomenon called *Cross-Tier Queue Overflow* [58]. As a result, subsequent requests on the waiting UM will then be blocked, which is called *cross-tier queue blocking*. For instance, in Fig. 3(b), if microservice-F experiences resource saturation, it will block the load of microservice-C once its queue overflows. Moreover, the load of all DMs (e.g., microservice-G) will also be blocked due to the queued requests at the UM (microservice-C). We analyze the two blocking effects in detail in Section III.

B. Grunt Attack Scenario

A Grunt attacker behaves as a normal user who can access the target microservice application through public HTTP requests. The attacker can profile the target application by sending different types of requests and recording the end-to-end response time. To launch a Grunt attack, the attacker can recruit an bots farm and send synchronized attacking requests to the target microservice application.

Blocking effects across multiple critical paths occur when the critical paths have overlapped microservices. Therefore, we group those critical paths that can have blocking effects mutually together, which is referred to *dependency group*. A microservice application may have multiple dependency groups and any overloaded critical path can cause blocking effects on all the other critical paths within the same dependency group. Through systematic profiling (details in Section IV-C), a Grunt attacker can infer the existence of blocking effects among different critical paths (via external HTTP requests) and divide them into separate dependency groups.

We assume a Grunt attack scenario on a microservice application that adopts Remote Procedure Communication (RPC) among different microservices as illustrated in Fig. 4, where the attacker tries to create persistent blocking effects that degrade the system performance. For each dependency group, the attacker employs pulsing bursts of attacking requests consisting of mixed types to mimic legitimate HTTP requests. Each burst is used to overload one critical path, creating a very short bottleneck or millibottleneck (with sub-second duration) on the weakest microservice within the path. The purpose is to create a blocking effect that blocks the execution of all other paths in the same dependency group. Before the millibottleneck disappears and the system cools down (after an interval period), the attacker selects a different critical path from the same dependency group to launch another burst. The purpose again is to create another blocking effect that maintains the blocking of all other paths' execution. The attack pattern continues during the whole attack period, creating persistent blocking effects that block the execution of all the critical paths in the targeted dependency groups, resulting in a very long response time for all legitimate requests.

TABLE I: Measured long response time damage by Grunt

Setting	Avg. RT (ms)		95ile RT		Net. (MB/s)		CPU (%)	
	Base.	Att.	Base.	Att.	Base.	Att.	Base.	Att.
EC2-7k	106	1142	120	4231	29	41	21	36
EC2-12k	107	1256	137	4351	56	68	33	45
Azure-4K	109	1378	143	5017	19	33	23	35
Azure-9k	117	1163	159	4452	39	49	37	46
NSF-5k	110	1382	164	4323	22	33	21	36
NSF-11k	104	1249	177	4436	47	54	43	58

Base.: baseline without attacks. **Att.** with attack.
CPU: average CPU usage of a representative microservice.
Net.: average network traffic at Gateway. **RT.** end-to-end response time

C. Measured Damage under Grunt Attack.

Table I illustrates the impact of Grunt attack on SocialNetwork [23]: an open-source benchmark for cloud microservices.

TABLE II: Model parameters.

Param.	Description
Q_i	the queue size for the ith microservice
$C_{i,A}$	the capacity of the ith microservice serving attack requests
$C_{i,L}$	the capacity of the ith microservice serving legitimate requests
λ_i	the legitimate request rate for the ith microservice
B	the attack request rate during an attack burst
V	the attack volume of an attack burst
L	the attack length of an attack burst
I_i	the interval between every two consecutive attack bursts
l_i	the time to fill up the queue of the ith microservice
Q_B	the queue length caused by an attack burst in the system
t_{damage}	the time to clear up the damage queue caused by an attack burst
t_D	the time to clear up the damage queue caused by multiple bursts
t_{min}	the minimum latency persistently created by multiple bursts
P_{MB}	the length of a millibottleneck caused by an attack burst

The bechmark application is deployed in three popular cloud platforms: Amazon EC2 [13], Microsoft Azure [14], and NSF CloudLab [4]. A sample setting EC2-3k means the cloud platform and baseline workload of legitimate users. More details about the experimental setup can be found in Section V.

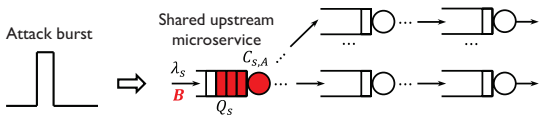
We compare the system response time perceived by normal users with and without Grunt attack, showing a significant performance degradation. For example, under all settings, without attack, the average response time of the target application is just about 100ms and the 95th percentile response time is less than 200ms. With the attack, the average and the 95th percentile response times turn to more than 1 and 4 seconds, respectively, averagely degrading more than 10 times and 20 times. On the other hand, the extra CPU and network bandwidth overhead under attack is less than 15% and 10%, respectively. Such small extra overhead will stay undetected under the radar of state-of-the-art DDoS defense systems.

III. Grunt Attack Modeling

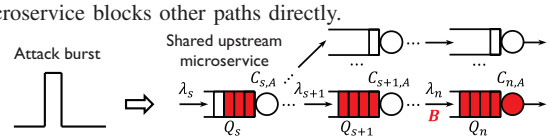
In this section, we study a theoretical queuing network model to characterize the relationship between the attacking bursts and their impact (stealthiness vs. latency damage) on the target microservices system. Such a relationship contributes to the design of the feedback control framework in Section IV.

We first model how a blocking effect could be triggered by a single burst and quantify its system impact from two perspectives (i.e., system latency and millibottleneck length). Then, we model how persistent blocking effects could be triggered when attacking a dependency group. Finally, we study how an attacker could select candidate critical paths to achieve attacking goals with minimum volume by exploiting the execution dependencies between critical paths.

Table II shows the notations of the parameters. The queue size of each microservice represents the number of server threads. Because of the inter-service communication dependency (call/response RPC), one queued request in a DM holds a queue slot in every UM. Fig 4 shows that the attacker sends bursts of mixed requests (each with a rate B) to launch a Grunt attack to every dependency group. We use L to represent the burst length and I to represent the interval between two successive bursts.



(a) Execution blocking effect. Millibottleneck on shared upstream microservice blocks other paths directly.



(b) Cross-tier queue blocking effect. Millibottleneck needs to fill up downstream queues to block other paths.

Fig. 5: Blocking effects by a single burst.

A. Blocking effects by a single burst

When an attacking burst overloads a critical path, it blocks the execution of other critical paths once a blocking effect is created. We use the latency and millibottleneck length caused by the burst to quantify the impact of the burst.

Latency created by a burst. An execution blocking effect exists when the millibottleneck occurs on a shared UM, as Fig. 5a shows. The millibottleneck can block all critical paths that share the bottleneck microservice. Given the burst length L and rate B, we can calculate the total queue that will be created by the burst as:

$$Q_B = (L) * (\lambda_i + B - C_{s,A}) \tag{1}$$

where $(\lambda_i + B - C_{s,A})$ is the queue build-up rate, which is the sum of the incoming request rate from normal users (λ_i) and attacker (B) subtract the service rate of the bottleneck microservice $(C_{s,A})$.

On the other hand, a cross-tier queue blocking effect exists when the millibottleneck causes cross-tier queue overflow from the bottleneck DM to the shared UM (see Fig. 5b). Therefore, we calculate the total time needed to fill up all the DMs before we derive the total queue length that can be created. If the n-th microservice is the bottleneck, then the time needed to fill up $n-$th microservice is:

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

where Q_n denotes the queue size of the n-th microservice and $(\lambda_n + B - C_{n,A})$ is the queue fill-up rate, which is the sum of incoming request rate from normal user (λ_n) and attacker (B) subtract the service rate of the bottleneck microservice $(C_{n,A})$. Similarly, we can calculate the time needed to fill up other DMs. Given the attacking parameters (L and B) of a burst, we can calculate the total queue length that is created after filling up all the DMs as:

$$Q_B = (L - \sum_{i=s+1}^{n} l_i) * (\sum_{i=s}^{n} \lambda_i + B - C_{n,A}) \tag{3}$$

where $(L - \sum_{i=s+1}^{n} l_i)$ is the total time to build up queues and $(\sum_{i=s}^{n} \lambda_i + B - C_{n,A})$ is the build up rate.

Finally, we calculate the overall latency that can be created by the burst based on the queue length. We assume the overall

service rate to process the queued requests is limited by the bottleneck microservice during the bottleneck period, which is ($C_{n,A}$). Thus, we consider the time to process the total queued requests as the damaged latency by the burst:

$$t_{damage} = \frac{Q_B}{C_{n,A}} \tag{4}$$

Then all the requests that access the target dependency group during the millibottleneck period will have a response time greater than (t_{damage}).

Millibottleneck length caused by the burst. During the period of processing the attacking burst, a millibottleneck occurs on the bottleneck microservice. The period of a millibottleneck is termed as *millibottleneck length* (P_{MB}) and can be derived as follows, which is adapted from Tail Attack [51].

$$P_{MB} = B * L * \frac{1}{C_{n,A}} * \frac{1}{(1 - (\lambda_n * \frac{1}{C_{n,L}}))} \tag{5}$$

B. Persistent blocking effects in a dependency group

A single burst only causes limited damage since the blocking effect will disappear once the burst has been processed. Therefore, Grunt attack tries to create persistent blocking effects at all the targeted dependency groups. Here we discuss the trigger condition of persistent blocking effects within a dependency group, which can be extended to attacking multiple dependency groups.

Assume that the targeted dependency group has m critical paths. We first use a mixed burst targeting all m critical paths to create multiple blocking effects and quickly build up queues at the shared UM. After that, each time we target one critical path to maintain the blocking effects.

Given m targeted critical paths ($i = 1, 2, .., m$) with attacking parameters (L_i, B_i), we can calculate the maximum damage latency t_D that can be created by multiple bursts:

$$t_D = \sum_{i=1}^{m} t_{damage,i} \tag{6}$$

where $t_{damage,i}$ is the damage latency created by millibottleneck at i-th execution path, which is calculated from Eqn. 4.

After the interval I_0, additional bursts are used to maintain blocking effects and cause persistent latency delay. Then the remaining damage latency after the interval I_0 is:

$$t_{min} = t_D - I_0 \tag{7}$$

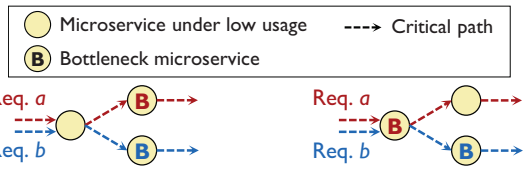
For any burst ($i = 1, 2, .., m$) with an interval time I_i to maintain the blocking effects, the remaining damage latency to be the same after each interval:

$$t_{min} = t_{min} + t_{damage,i} - I_i \tag{8}$$

Then, during the process, all the requests accessing the target dependency group will have a response time greater than t_{min}. The required interval of each burst to maintain the latency can be calculated with:

$$I_i = t_{damage,i} \tag{9}$$

where the interval I_i for i-th burst serves as the condition of maintaining the blocking effect created by the previous burst.



(a) Parallel dependency. The requests a and b have different bottleneck microservices while they share an upstream microservice.

(b) Sequential dependency. The bottleneck microservice of a is upstream of the bottleneck microservice of b.

Fig. 6: Pairwise execution dependencies

C. Attacking a dependency group with minimum volume

Given all the critical paths within a target dependency group, critical paths may have very different damage impacts with the same attacking volume ($L*B$). For example, attacking a critical path can block the execution of other critical paths directly if an execution blocking effect is triggered, however, it needs to fill up all DMs to block other critical paths if a cross-tier queue blocking is triggered. Therefore, to attack a dependency group with minimum volume (e.g., choosing those critical paths can cause more damage), an attacker needs to know the blocking effects that can be triggered by the critical paths. To help an attacker understand such dependency relationships, we first define the pairwise execution dependencies between critical paths. Then we theoretically discuss the priority of choosing critical paths based on the pairwise execution dependency.

Definition I. Two critical paths have a *parallel dependency* if they have different bottleneck microservices while they share one or more UMs (see Fig. 6a). In such a dependency, a critical path can trigger a cross-tier queue blocking effect to block the execution of the other critical path.

Definition II. Two critical paths have *sequential dependency* if the bottleneck microservice of one critical path is upstream of the bottleneck microservice of the other critical path (see Fig. 6b). In this dependency, the "upstream" critical path (i.e., req.a in Fig. 6b) can trigger an execution blocking effect. In contrast, the "downstream" critical path (i.e., req.b in Fig. 6b) can trigger a cross-tier queue blocking effect to block the execution of the other path.

Priority of choosing candidate critical paths. Given the pairwise dependencies in a dependency group, an attacker can rank the critical paths when choosing candidate critical paths for attacking. Based on the definition of pairwise execution dependency, the "upstream" critical path in a sequential dependency can always trigger an execution blocking effect over the other critical path, which can block the execution of other critical paths directly without filling up DMs. Therefore, such "upstream" critical paths have a higher priority when choosing the candidate critical paths. On the other hand, those critical paths that trigger cross-tier queue blocking effects need to fill up DMs to block the execution of other critical paths, we rank them by the volume ($L * B$) to trigger the same length of millibottleneck (i.e., $P_{MB} = 500$ms). The request with a lower volume has a higher priority at the same level (for stealthy). Here we discuss how an attacker could rank those

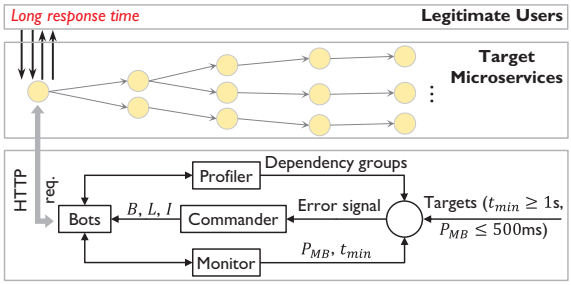


Fig. 7: Feedback control Framework

critical paths in a dependency group based on the pairwise execution dependencies. We introduce how an attacker can estimate the pairwise execution dependencies in practice using a blackbox profiling approach in Section IV.

In summary, the theoretical model gives us a firm foundation to design a feedback control framework that dynamically tunes the attacking parameters with the drift of system states. Specifically, t_{damage} and P_{MB} have a linear relationship with the attacking burst length L when we fix the burst rate B. The linear relationship model can be used by the Kalman filter control foundation to tune the attacking parameters in real attacking scenarios. In the next section, we introduce how an attacker can estimate millibottleneck length and damage latency and use them as the feedback control input to tune attacking parameters in real attacking scenarios.

IV. Attack Implementation

A. Overview

The numerical analysis shows the theoretical relationship between the attack parameters and their impact on the target dependency groups. However, as external users, Grunt attackers do not know the internal system parameters and microservices architecture. In addition, the previous analysis does not consider more realistic conditions, such as variations of system state and baseline workloads. In this section, we present a feedback control framework to dynamically tune the attacking parameters with the drift of system states as illustrated in Fig. 7. To evaluate the fitness of attacking parameters, we design a **Monitor Module** which estimates the system impact (e.g., damage latency t_{min} and millibottleneck length P_{MB}) caused by Grunt attacks in Section IV-B. To infer the dependency groups, we design a blackbox **Profiler Module** to profile the pairwise execution dependencies between critical paths from the perspective of external users. Based on the pairwise execution dependencies, we can construct the full dependency groups to help attacks select candidate critical paths for attacking in Section IV-C. Finally, to dynamically adapt the attacking parameters with the dynamic of system states, we design a control **Commander Module** with feedback control tools (i.e., Kalman filer [30]) in Section IV-D.

B. Monitor Module

Estimating millibottleneck length. To avoid being detected by normal resource monitoring tools (i.e., with 1s granularity),

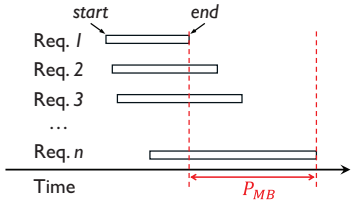


Fig. 8: Infer P_{MB} by the last attack request's end time subtracting the first attack request's end time within a burst.

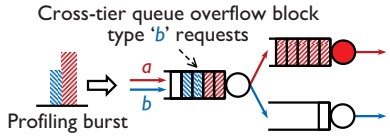
Grunt attacks limit the millibottleneck length (i.e., $P_{MB} <$ 500ms) created in each critical path. Therefore, we need to correctly estimate the millibottleneck length practically from the perspective of external users. After sending a burst of attacking requests to the target, the attacker can record the start-time and end-time of each request. Assume the same type of requests flow through the same critical path and consume the same bottleneck resource, we estimate the millibottleneck length P_{MB} along the critical path by the end-time of the last attack request subtracting the end-time of the first request in an attack burst that targets the path, as shown in Fig. 8. This estimation is reasonable because the burst of attack requests will continue to consume the bottleneck resource until the last one. We note that such a way is a conservative estimation since we undercharge the service of the first request. The real P_{MB} could be shorter than the estimation.

Estimating damage latency. To quantify the damaging impact, we estimate the damage latency (t_{min}) by the average end-to-end response time of requests in each burst. Through the modeling analysis in Section III, each burst can cause a blocking effect on all the critical paths in the dependency group. Persistent damage happens when additional bursts can maintain the blocking effects during the attacking period. We update t_{min} after each burst to estimate the success of persistent damage dynamically.

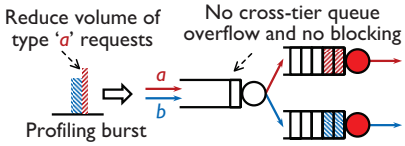
C. Profiler Module

The profiler first finds all supported critical paths of the target system, then identifies the pairwise execution dependency (i.e., parallel and sequential) among them, and finally constructs the dependency groups.

Extracting supported critical paths via public URLs. In a microservices web application, each type of user request traverses among multiple microservices and triggers one critical path. Hence, we can crawl the public URLs of the target application to retrieve all supported user requests, and we consider that each user request corresponds to a specific critical path. Website crawler and scraping tools [1], [3] can profile all supported HTTP requests of a target system. In our implementation, we use similar approaches introduced in the work [52] by using the script-based web browser PhantomJS to automatically retrieve all public requests of a target website. For some dynamic requests that require input, an attacker may provide some initial values for associate input forms (e.g., user name and password). Non-valid or static requests are



(a) Millibottleneck triggered by type 'a' request burst (*red*) causes cross-tier queue overflow to the shared UM and blocks type 'b' requests (*blue*). Performance interference exists.



(b) Reducing the volume of type 'a' request burst (*red*) to avoid cross-tier queue overflow. No performance interference.

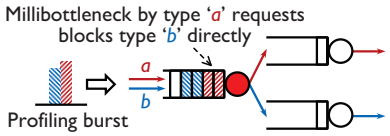
Fig. 9: Performance interference testing in parallel dependency scenario.

lightweight and served directly by the gateway or cache server. Therefore, we do not consider such requests as candidate.

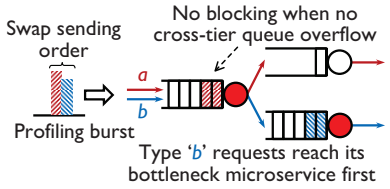
Identifying pairwise dependencies. Given any two critical paths, we identify the dependency type by analyzing how performance interference could happen among them. To test the performance interference, we send bursts consisting of two types (namely req. a and b) of requests with different volumes. Then we check whether the changes in volumes would affect the existence of performance interference in the following.

Parallel Dependency exists when the two critical paths have different bottleneck DMs while sharing an UM (see Fig 6a), which means that type a requests can block type b requests only when the cross-tier queue overflow occurs on the shared UM (cross-tier queue blocking). For example, Fig 9a shows the case when sending a burst of profiling requests successively. The millibottleneck in critical path a causes the local queue to fill up and further causes cross-tier queue blocking. Then the next incoming requests (type b) would be blocked at the shared UM (performance interference exists). However, if the millibottleneck in the path a does not cause cross-tier queue overflow due to a low volume (no cross-tier queue blocking), the incoming requests (type b) would directly reach its DMs (see Fig 9b). Then we could not observe performance interference. Hence, to profile parallel dependency, we send a series of bursts of requests a and b. Then we gradually increase the volume of requests (type a) from low to high until we reach the maximum volume (e.g., $P_{MB} = 500$ms). During the process, (1) if no performance interference is observed at any volume, we consider that the two execution paths have no dependency; (2) if the existence of performance interference varies with the volume, we consider that the two execution paths have a parallel dependency relationship; (3) if the performance interference exists persistently (does not vary with the volume), we move to the next profiling step.

Sequential Dependency exists when the bottleneck microservice of one critical path is an UM of the other critical path (see Fig 6b). In contrast to the parallel dependency, the upstream path a would always have performance interference (execution

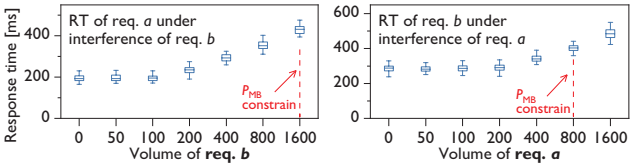


(a) Type 'a' request burst (*red*) triggers the millibottleneck at the shared UM and thus blocks type 'b' requests (*blue*). Performance interference exists.

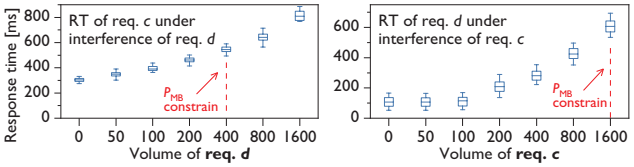


(b) Switch the order of type 'a' and 'b' bursts. Type 'b' request burst (*blue*) triggers the millibottleneck at a non-shared DM and thus does not block type 'a' requests (*red*). No performance interference.

Fig. 10: Performance interference testing in sequential dependency scenario.



(a) Existences of performance interference vary with burst volume, indicating a parallel dependency between `req. a` and `req. b`.



(b) `req. d` always have performance interference to `req. c`, indicating a sequential dependency between `req. c` and `req. d`.

Fig. 11: Pairwise dependency profiling. We consider the cases with significantly larger response time than the baseline as the existence of performance interference.

blocking effect) with path b no matter how much volume was sent (see Fig. 10a). This is because the millibottleneck triggered by path a happens on the shared UM, which can block the next incoming requests (type b) directly. However, if we change the order of profiling bursts (i.e., type b first and a second), the first arrived requests (type b) would reach its DMs, and we may not observe the performance interference unless queue overflow occurs in path b (see Fig 10b). Hence, to profile sequential dependency, again we send a series of bursts of requests a and b with gradually increased volume. If we observe one type of request (a) have persistent performance interference over another (b), then it suggests a sequential dependency among the pair of requests.

Fig. 11 illustrates the procedure of pairwise dependency profiling. We use profiling bursts that consist of a pair of requests to test the performance interferece. If performance interference exists, the response time of the sample requests should be

significantly higher than no performance interference. Fig. 11a shows the existence of performance interference between requests a and b varies with profiling volume. It suggests a parallel dependency between requests a and b. Fig. 11b shows req. d always has performance interference to req. c, while req. c needs a certain volume to have performance interference. It suggests a sequential dependency (d is upstream).

D. Commander Module

Initialization of attacking parameters. Given the stealthiness requirement (i.e., $P_{MB} \leq$ 500ms), we initialize B and L for each path in two steps. (1) Find the minimum burst rate B. We fix the burst length L and use testing bursts with the gradually increased B to find the minimum rate that just triggers a millibottleneck. Since the response time of requests barely changes when there is no resource saturation in the system [57], we use the average response time of each burst to determine whether a millibottleneck happens. (2) Find the maximum burst length L. We fix B to the value from step (1) and use testing bursts with gradually increased L to find the maximum B that meets the stealthiness requirement. (3) Find the minimum number of paths m to attack. We fix B and L to the value from steps (1) and (2) for each path and use testing bursts with gradually increased m to find the minimum m that meets the damage requirement (i.e., $t_{min} \geq$ 1s).

Adapting attacking parameters. During the attacking process, we dynamically adapt the attacking parameters with the variation of system states based on the feedback metrics (t_{min} and P_{MB}) from the Monitor module. For each burst that attacks a dependency group, we adapt L to always trigger the desired millibottleneck length (i.e., $P_{MB} \leq$ 500ms). Given a dependency group with m critical paths, we adapt the interval time I to always maintain the damage goal (i.e., $t_{min} =$ 1s). To mitigate the negative impact of observing/prediction inaccuracy, we adopt the feedback-based control Kalman filter [30] algorithm to optimize the estimation of P_{MB} and t_{min}.

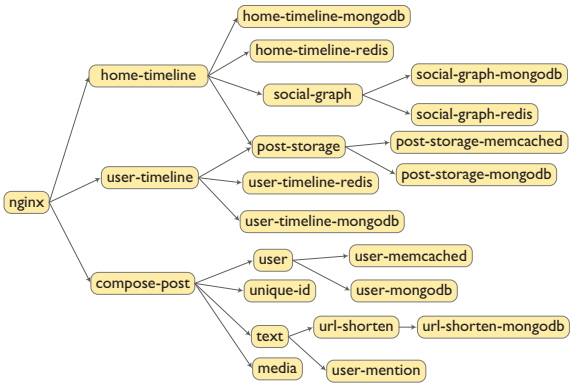
V. Attack Evaluation

A. Ethical Considerations

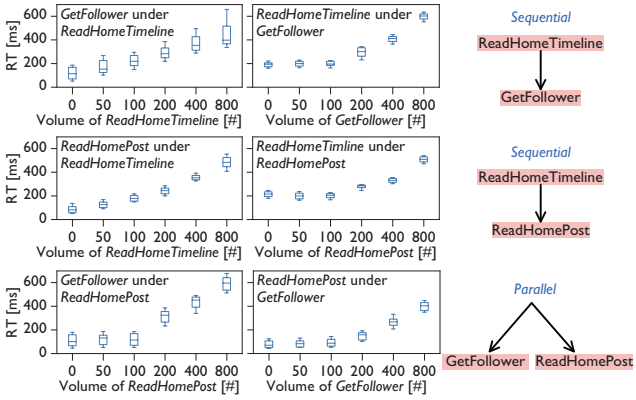
To understand the impact of Grunt without raising ethical issues, we deploy the attack in controlled environments. Specifically, we implement microservice applications on our instances in different cloud platforms. All the offensive traffic (e.g., HTTP requests) will only be sent to our applications. And we only create resource contention on our instances without affecting other cloud users due to the instance isolation mechanisms provided by the Cloud platforms [47], [55].

B. Grunt Attack in Cloud Production Environments

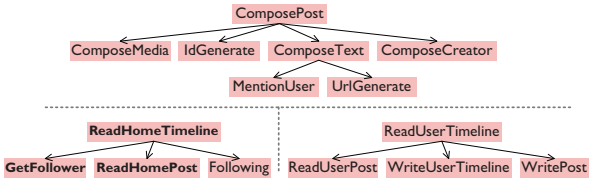
We first deploy an open-source microservice application SocialNetwork [23] in a Docker swarm cluster on two popular commercial cloud platforms (Amazon EC2 [13], Microsoft Azure [14]) and one academic cloud platform (CloudLab [4]). SocialNetwork implements a broadcast type of social network application, where users can post messages and follow other users. All the cloud platforms are equipped with auto-scaling features to handle workload variations. We also deploy one of the state-of-the-art IDS/IPS systems Snort [8].



(a) **Administrator's view.** Service Dependency graph of SocialNetwork microservices [23]. Each element represents a microservice.



(b) **Attacker's view.** Representative pairwise profiling results.



(c) **Attacker's view.** Dependency groups constructed after pairwise profiling. Each element represents a user request.

Fig. 12: Experimental results for SocialNetwork microservices.

Experimental Setup. Fig. 12a illustrates the representative call graph architecture of SocialNetwork. To emulate legitimate user behavior, we employ a closed-loop workload generator. Each user progresses through a Markov chain to navigate web pages, with an average 7-second thinking time. Each microservice is hosted within a container running on a VM. With auto-scaling features enabled, these VMs are initialized with 1 vCPU core and 2GB memory, representing the basic computing unit for commercial cloud providers [51].

To implement auto-scaling for the backend cluster on cloud platforms, we use CloudWatch [9] and Azure Monitor [43] with a granularity of 1 second to monitor microservice resource utilization in EC2 and Azure, respectively. CloudWatch and Azure Monitor are the default monitoring tools that enable

TABLE III: Attacking results as we set the attacking goals to be $P_{MB} \leq 500ms$ and average response time $>$ 1s.

Setting	Bot (#)	P_{MB} (ms)	avg. RT (ms)		Net. (MB/s)		CPU (%)	
			Base.	Att.	Base.	Att.	Base.	Att.
EC2-7K	269	482	106	1142	29	41	21	36
EC2-12K	196	499	107	1256	56	68	33	45
Azure-4K	243	452	109	1378	19	33	23	35
Azure-9K	178	458	117	1163	39	49	37	46
CloudLab-5K	314	473	110	1382	22	33	21	36
CloudLab-11K	229	443	104	1249	47	54	43	58

Bot: number of bots used to avoid rate-based detection.
P_{MB}: average millibottleneck length created. **RT:** response time.
Net.: average network traffic measured at frontend. **Att.:** with attack.
CPU: average CPU usage of a representative bottleneck microservice.

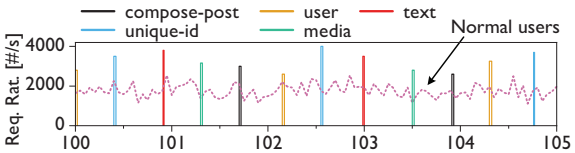
the auto-scaling features in the two cloud platforms [11], [12]. In CloudLab, we use customized scripts to scale the backend microservices with the Docker default resource monitor tool Docker stats [6]. We configure the auto-scaling policy to scale up if the CPU utilization exceeds 70% for 30 seconds and scale down if the CPU utilization is less than 30% for 30 seconds on all the cloud platforms.

To evaluate whether the attackers' behavior deviates from normal users, we deploy Snort [8], a rule-based IDS/IPS system, at the gateway. We follow a popular user-behavior model [44] to set alert rules. The user-behavior model analyzed the distribution of normal users' interaction with a production website from a long-term log data. To set an appropriate threshold for the alert rule for normal clients, we calculate the 95% confidence interval of the inter-request interval of the legitimate users, which is 2.8 to 14.4. We round the lower bound to 3 to reduce the false positive rate, which means if a user session sends two consecutive requests with an interval of less than 3 seconds, it will be considered as abnormal behavior.

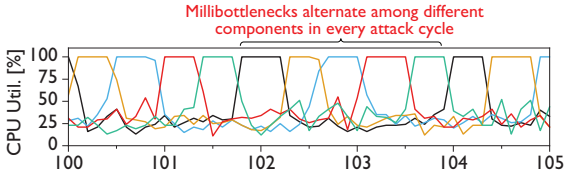
We use a centralized way to coordinate and synchronize a bot farm [26], which can achieve millisecond-level synchronization. During an attacking burst, we use multiple bots to send HTTP requests (each bot sends one request).

Overall results. We first use the Profiler to construct the dependency groups of the target system. Fig. 12b shows three representative pairwise profiling. Based on the pairwise profiling, we construct three dependency groups for SocialNetwork as shown in Fig. 12c. We then conduct 20-minute attacks under different settings. Table III shows the corresponding attack parameters and system status in various production settings. The setting "EC2-7K" means the cloud platform (AWS EC2) and the baseline workload (7000 concurrent normal users). We set the damage goal as "avg. RT $\geq$ 1s" and stealthy goal as "$P_{MB} \leq$ 500ms" to illustrate the flexibility of Grunt attack. Columns 3 to 5 show that we achieve the predefined attacking goals under different system settings. Column 2 shows the number of bots used per setting. Overall, the results show that Grunt attack consumes less than 20% additional CPU while increasing its average response time by 10 times.

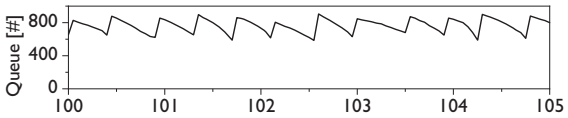
Zoom-in white-box analysis. Fig. 13 shows the zoom-in timeline of the system status of a dependency group in "EC2-12K" in Table III. Fig. 13a shows the request rate from attackers and normal users. Fig. 13b shows the millibottlenecks alternate among different bottleneck microservices in every attacking



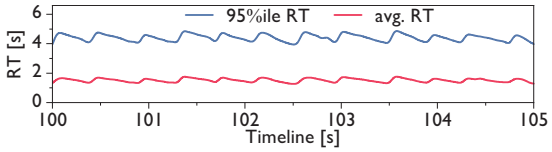
(a) Request rate from attackers and normal users. Each burst creates a millibottleneck at its corresponding bottleneck microservice.



(b) Millibottlenecks alternate among different bottleneck microservices within the target dependency group. Only fine-grained monitoring (e.g., 100ms) can detect the millibottlenecks.



(c) Persistent queue at the shared upstream microservice (i.e., `compose-post`) due to the persistent blocking effects.



(d) Large performance damage caused by the persistent queue at the shared upstream microservice.

Fig. 13: Fine-grained runtime analysis of system resource usage and response time for a dependency group under attack in "EC2-12K". Metrics are collected every 100 milliseconds.

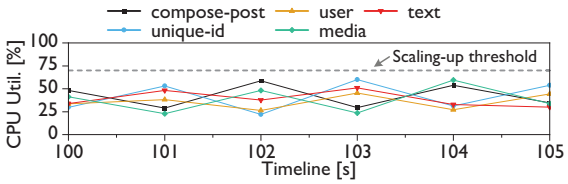
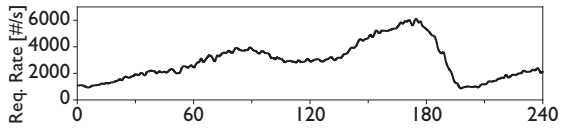


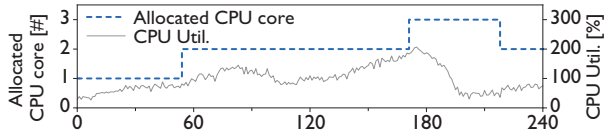
Fig. 14: CPU utilization monitored by CloudWatch with a granularity of 1s in the same experiment as shown in Fig. 13. No scaling actions were triggered.

cycle, monitored with 100-ms granularity. Fig. 13c shows the queued requests at the shared UM (`compose-post`). We observe that multiple queued requests from different bottleneck microservices accumulate with each other at the shared UM and cause persistent queuing delays. Fig. 13d shows the average end-to-end response time of the legitimate users.

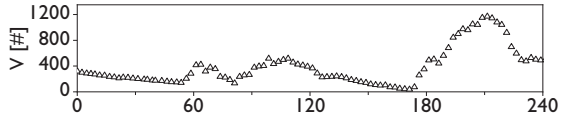
Stealthiness evaluation. We first find that the auto-scaling mechanisms in all cloud platforms do not take any scaling action during the attacking period. This is because the coarse-grained cloud monitoring tools could not detect any millibottlenecks since their finest supported granularity is 1s [2], while the triggered millibottleneck length among different microservices is less than 500ms. Fig. 14 shows the CPU utilization of corresponding microservices monitored by CloudWatch



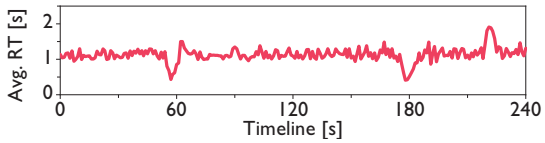
(a) Request rate of baseline workload varies from 1000 to 6000 req/s.



(b) CPU utilization of `compose-post` microservice. The system dynamically scales up and down to handle the workload variations.



(c) Attack volume for targeting `compose-post` microservice dynamically adjusted by our Framework.



(d) Measured average end-to-end latency of normal requests.

Fig. 15: Results of Grunt attack on EC2 Cloud with Auto-scaling under a real-world "Large Variation" workload trace. We show the auto-scaling actions and corresponding attacking volume on a representative microservice (`compose-post`) in (b) and (c) as illustration.

within the same experiment in Fig. 13. The average CPU utilization of the most heavily loaded microservice is less than 60%. Therefore, the current auto-scaling mechanisms could not mitigate the performance damage caused by our Grunt attack. Second, Snort could detect the denial of service (e.g., long response time) while not tracking the root causes. First, Grunt utilizes low-volume legitimate HTTP requests. It neither modifies the content of requests (e.g., header manipulation) nor violates transaction protocols (e.g., TCP split handshake). Thus, no content-based and protocol-based alerts are triggered. Second, no resource saturation (e.g., CPU, network bandwidth) is observed with normal granularity (i.e., 1s interval) monitor tools. Thus, no resource-based alert is triggered. Finally, IPS tools monitor the request rate per IP and quantify the interval between two successive requests to detect bots. For example, the AWS Shield limits the total number of requests per IP every 5 minutes to block bots [13]. In our experiment, each virtual bot only sends one request in a burst, and we tune the interval of requests sent per bot to avoid bot blocking. In practice, attackers can use lightweight requests to estimate the threshold value before attacking and use conservative values (e.g., use more bots) to avoid the rate-based rules [32], [49].

Evaluation under bursty baseline workload. To verify the effectiveness of the Commander under the variation of baseline workload, we conducted an experiment on EC2 under "Large Variation" bursty workload traces from Gandhi [24] as shown in Fig. 15a. The baseline workload from normal users varies from 1000 to 6000 req/s. Fig. 15 shows the timeline of the attacking parameters tuning under the "Large Variation" workload. The system dynamically scales up and down to handle the workload variations. Here, we use `compose-post` microservice as an illustration since other targeted microservices have a similar pattern. Fig. 15b captures 2 scale-up actions and 1 scale-down action that happens on `compose-post` with the variations of CPU utilization. Fig. 15c tracks the attacking volume adjusted by the Commander, and Fig. 15d monitors the average end-to-end latency of normal requests. The Commander dynamically adjusts the attacking volume to fulfill the damage goals with baseline workload variation and system resource scaling.

Overall, the results substantiate the efficacy of our framework in executing Grunt attack, successfully accomplishing the predefined attacking goals in the cloud environments with realistic workloads.

C. Grunt Attack in Live Attack Scenarios

We further design three live attack scenarios with unknown microservice architectures to evaluate the profiling accuracy and effectiveness of Grunt attacks. We conduct the experiments with μBench [21], a factory of benchmarking microservices tool. The tool can create customized microservice applications running on the Kubernetes cluster. The experiments are conducted on CloudLab [4] subject to the following rules. (1) We have engaged 6 independent volunteers from an academic institution (LSU), where 3 volunteers behave as administrators for microservices applications and the other 3 volunteers behave as baseline workload managers to simulate legitimate users. We provide each volunteer with access to the CloudLab cluster of 30 m510 nodes. Each node has 8 CPUs and 64GB of memory. (2) To simulate microservices applications at different scales, the three administrators were instructed to have different total numbers of unique microservices in their applications. (3) Each administrator designs and deploys their customized microservices. (4) The workload managers use the same closed-loop workload generator as Section V-B to emulate the behavior of legitimate users. (5) For each application, we conduct eight 20-minute attacks. During each attack, the workload managers choose a number of baseline legitimate users to emulate the system under different workloads.

To provide the ground truth of pairwise dependency, we track the execution path of user requests with Jaeger Tracing [28] and find the bottleneck microservice along the execution path with Collectl [5], where the two tools have been integrated into the μBench. Following each live attack, we conduct an offline analysis to identify the dependency relationship between each pair of user requests, establishing them as the ground truth.

Results. Fig. 16 shows the Precision, Recall, and F-score of the Profiler under 8 different baseline workloads for each attack scenario. The lowest Recall occurs when the baseline workload is very low, showing false negatives by the Profiler. This is because we constrain the profiling volume to meet

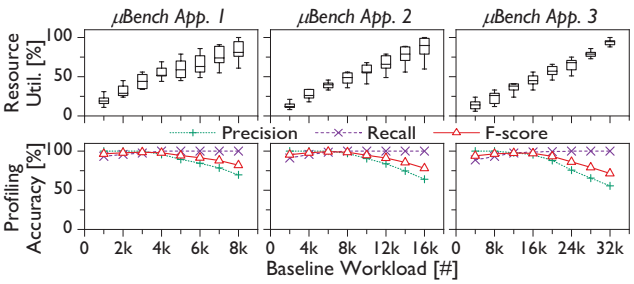


Fig. 16: Precision, Recall, and F-score of pairwise requests profiling under different baseline workloads. The three applications deployed by different volunteers have 62, 118, and 196 unique microservices, respectively.

stealthy requirements (i.e., $P_{MB} < 500$ms). A pair of critical paths with parallel dependency may not have performance interference when the profiling bursts cannot trigger cross-tier queue blocking (not enough queued requests). Thus, we may miss some parallel dependencies when the baseline workload is low. On the other hand, the lowest Precision occurs when the baseline workload is very high, showing false positives by the Profiler. This is because the performance under the high baseline workload is already unstable. A pair of critical paths without dependency may have performance fluctuations caused by high resource usage. Thus, we may wrongly consider some non-dependent critical paths as dependent when the baseline workload is high. In reality, the average resource usage of modern cloud applications is usually less than 50% [57], where our Profiler has high accuracy (i.e., F-score$>$ 90%). Overall, we confirm that the Profiler can achieve high accuracy when the baseline workload is moderate within a realistic range.

Table IV shows the setting and corresponding system status when attacking after the profiling. For each application, we select two representative baseline workloads (low and medium) to show the results. The setting "App.1-1K" means the application name and the baseline workload. Column 2 shows the satisfaction of triggered millibottleneck lengths. Columns 3 and 4 show the response time of all normal users, which validates the success of the damage goal. Columns 5 to 8 show only small additional traffic and low extra resource overhead to achieve the system-level damage, which are considered as low costs compared to traditional DDoS attacks. Comparing different workloads under the same application, the results show that it is easier to achieve the attacking goal when the baseline workload is high. This is because the bottleneck resource is already under high usage during the high baseline workload, which requires less effort from attacking requests to trigger the same blocking effects on those critical paths.

In summary, the live attack experiments validate that Grunt attack is also effective on different scale microservices. Compared with small-scale applications, large-scale ones have even less bottleneck resource overhead. This is because large-scale applications usually have more critical paths (thus more different bottleneck microservices) within a dependency group. Alternating among more bottleneck microservices results in a longer gap between two successive millibottlenecks on the same microservice, thus less overhead on bottleneck resources.

TABLE IV: Results of live attack experiments.

Setting	P_{MB} (#)	avg. RT (ms)		Norm. traffic		CPU (%)	
		Base.	Att.	Base.	Att.	Base.	Att.
App.1-1K	478	69	1441	1	1.23	22	38
App.1-3K	484	76	1533	1	1.22	41	53
App.2-4K	455	79	1356	1	1.32	17	29
App.2-8K	483	77	1249	1	1.31	39	49
App.3-8K	489	83	1233	1	1.38	21	29
App.3-16K	470	91	1317	1	1.37	44	51

P_{MB}: average millibottleneck length created. **Att.:** with attack.
Avg. RT: average response time (ms). **Norm. traffic:** normalized traffic.
CPU: average CPU usage of a representative bottleneck microservice.

VI. Discussions

Possible defense and mitigation. Detecting and defending Grunt attacks is challenging since it is difficult to accurately distinguish attack requests from legitimate ones. We discuss two directions to explore the possible defense.

Detection of millibottlenecks and suspicious requests. Grunt attack triggers alternating millibottlenecks in the system. To detect millibottlenecks, the monitoring granularity should be less than the millibottleneck length with fine-grained monitoring tools. With the detection of millibottlenecks, an administrator can consider the requests with a high correlation to millibottleneck as suspicious. Usually, the normal requests follow the user behavior and have no statistical correlation with millibottlenecks while suspicious requests may only appear when the millibottlenecks occur. For example, Tail attack [51] introduces a statistic-based solution to address this millibottleneck challenge by correlating the resource usage spikes with suspicious clients. However, detecting millibottlenecks requires fine-grained resource monitoring at a microservice component level, which is well-known to bring non-trivial overhead. System administrators need to consider the trade-off between monitoring granularity and the performance overhead. Designing appropriate resource monitors and statistical analysis of user behaviors represents a potential solution to defend against Grunt attack.

Reduce sharing of hot-spot bottleneck microservices. Grunt attack exploits the runtime dependencies among different critical paths. The dependencies arise when critical paths partially share microservices. Thus, an administrator can mitigate the runtime dependencies by reducing/avoiding sharing microservices between different paths. For example, Alibaba Trace [38] analyzes call graphs to identify runtime dependencies and suggests that coupling the microservices with strong dependency can reduce the propagation of blocking effects. However, identifying runtime dependencies among different critical paths requires fine-grained tracing tools, and reducing the overlapped microservices requires redesigning the microservices architecture, which may increase the difficulty of development and deployment. Finding new performance dependency analysis approaches and designing new resource management tools to handle resource contention by such dependency represents a potential mitigation.

Benchmark vs. real-world microservices. We evaluate the effectiveness of Grunt attack with benchmarks instead of real-world microservices to avoid ethical issues. The benchmark applications (SocialNetwork and μBench) employ the same RPC communication between component microservices as real-world microservices [27], [38], [61]. The number of unique microservices in those benchmarks ranges from 36 to 196, representing different scales from small to large systems. In addition, even though some very large-scale microservices deployments may consist of hundreds to thousands of services across various containers, our Grunt attack still remains harmful due to its ability to tune attack parameters to achieve different attacking goals flexibly. For example, the attacker can choose to attack only a subset of dependency groups to damage some vulnerable critical paths in a large-scale system without overloading its network bandwidth.

Impact of microservice's queue size. The cross-tier queue blocking exploits the limited queue size of each microservice. It happens when the queue slots of DMs are occupied by requests. The queue size of each microservice represents the number of server threads that can be concurrently processed. A larger queue size may cause more attacking requests needed to fulfill the queue but also increase the system's hardware overhead to handle the concurrent threads. Sub-optimal queue size can lead to large performance fluctuations and even system instability [37]. Thus using very large queue sizes in microservices could not address Grunt attack.

Limitations of Grunt attack. Grunt attack has several limitations that we plan to address in future work. First, Grunt attack currently focuses on disrupting the performance of the backend microservices. However, static requests are usually cached and served directly by frontend/edge servers (e.g., CDN) without accessing the backend microservices, which may escape the impact of Grunt attacks. Second, profiling accuracy for pairwise dependencies varies with the baseline workload and achieves the highest F-score when the system is under moderate resource utilization (Fig. 16). Third, some dynamic requests require input parameters, attackers may not be able to cover all possible valid parameter combinations, which may leave some critical paths undiscovered.

VII. Related Work

In this section, we review the most relevant work on performance vulnerabilities of microservice applications and low-volume application layer DDoS attacks.

Performance vulnerabilities of microservices. Several existing studies have explored the cross-service dependency and runtime performance [23], [27], [39], [53], [61] in microservices. Luo el al. [38] conduct anatomy of microservice call graphs to compare the dependencies between them to traditional DAGs using large-scale tracing data. FIRM [46] analyzes the runtime performance interference caused by hardware resource contention and presents a resource management framework using reinforcement learning. However, all of them explore performance vulnerabilities from the perspective of system managers. Our work presents a thorough analysis of performance vulnerabilities of microservices from the perspective of external users and utilizes such vulnerabilities to launch stealthy low-volume DDoS attacks that achieve large performance damage.

Low-volume application layer DDoS attacks. Low-volume application layer DDoS attacks focus on disrupting legitimate user's services by a small number of attack requests targeting application services, as an extension of network-layer low-volume attacks [17], [25], [26], [60]. Warmonger [59] exploits the limited IPs in serverless platforms to induce IP blockages of benign functions. ReDoS [54] uses malicious input to match regular expressions that cause long delays. Shan et al. [51] introduce a low-rate DoS attack that targets a single path against a monolithic n-tier system with wisely strike ON/OFF attack waveforms to bypass defense mechanisms, which share similar waveforms patterns with us. However, microservices architecture decomposes the monolithic design of business logic into loosely coupled microservices, thus having better performance anomaly tolerance [16]. The attacks that target single path may become ineffective on microservices [15], [36]. This is because attacking a single path will only affect a few dependent execution paths in the system as shown in this paper, which may not meet either the damage goal or stealthiness requirements. Grunt attack profiles the runtime dependencies between multiple paths and exploits multiple very short bottlenecks alternating among different microservices to achieve large performance problems with only small resource overhead, which has not been explored before.

VIII. Conclusion

In this paper, we introduce Grunt attack, a novel form of low-volume DDoS attack specifically targeting microservice applications. The attack exploits the runtime dependencies among execution paths within the system. We demonstrate that by strategically introducing multiple millibottlenecks across a few critical paths, accumulated blocking effects occur due to these dependencies, resulting in severe system response time degradation. We adopt a theoretical queuing network model to characterize the relationship between the attacking bursts and their impact (stealthiness vs. latency damage) on the target microservices system. To assess the practicality of our approach, we conducted extensive experiments using open-source microservices benchmark applications on real cloud environments and also evaluated live attacking scenarios. Our findings contribute to the understanding of internal runtime dependencies in microservices and shed light on novel and stealthy application layer DDoS attacks.

IX. Acknowledgements

We thank the anonymous reviewers and our shepherd Dr. Xing Gao for helping us improve the paper. This work has been partially funded by the NSF grant CNS-2000681 and contracts from Fujitsu Limited. Any opinions, findings, and conclusions are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] "20 best open source automation testing tools in 2022," https://www.softwaretestinghelp.com/open-source-testing-tools/, accessed, 2022.

[2] "Amazon cloudwatch introduces high-resolution custom metrics and alarms," https://aws.amazon.com/about-aws/whats-new/2017/07/amazon-cloudwatch-introduces-high-resolution-custom-metrics-and-alarms/, accessed: 2017.

[3] "Automate software testing — testim," https://go.testim.io/testim-automate-enterprises, accessed, 2022.

[4] "Cloudlab," https://www.cloudlab.us/, accessed: 2021.

[5] "Collectl," http://collectl.sourceforge.net/, accessed: 2022.

[6] "Docker container stats," https://docs.docker.com/reference/cli/docker/container/stats/, accessed: 2023.

[7] "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, accessed: 2022.

[8] "Snort - network intrusion detection," https://www.snort.org/, accessed: 2021.

[9] "Amazon cloudwatch," https://aws.amazon.com/cloudwatch/, 2017.

[10] "Cloudflare ddos threat report for 2022 q4," https://blog.cloudflare.com/ddos-threat-report-2022-q4/, 2022.

[11] "Autoscale common metrics," https://learn.microsoft.com/en-us/azure/azure-monitor/autoscale/autoscale-common-metrics, 2023.

[12] "Monitor cloudwatch metrics for your auto scaling," https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-cloudwatch-monitoring.html, 2023.

[13] "Amazon ec2," https://aws.amazon.com/ec2/, Accessed: 2021.

[14] "Microsoft azure," https://azure.microsoft.com/en-us/, Accessed: 2021.

[15] F. Al-Doghman, N. Moustafa, I. Khalil, Z. Tari, and A. Zomaya, "Ai-enabled secure microservices in edge computing: Opportunities and challenges," *IEEE Transactions on Services Computing*, 2022.

[16] A. F. Baarzi, G. Kesidis, D. Fleck, and A. Stavrou, "Microservices made attack-resilient using unsupervised service fissioning," in *Proceedings of the 13th European workshop on Systems Security*, 2020, pp. 31–36.

[17] E. Cambiaso, G. Papaleo, and M. Aiello, "Taxonomy of slow dos attacks to web applications," in *International Conference on Security in Computer Networks and Distributed Systems*. Springer, 2012, pp. 195–204.

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

[19] J. Cloud, "Decomposing twitter: Adventures in service-oriented architecture," https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture, 2013.

[20] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of ACM*, vol. 61, no. 8, p. 65–72, July 2018.

[21] A. Detti, L. Funari, and L. Petrucci, "μbench: an open-source factory of benchmark microservice applications," *IEEE Transactions on Parallel and Distributed Systems*, 2023.

[22] S. Esparrachiari, T. Reilly, and A. Rentz, "Tracking and controlling microservice dependencies: Dependency management is a crucial part of system and software design." *Queue*, vol. 16, no. 4, pp. 44–65, 2018.

[23] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi *et al.*, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *ASPLOS*, 2019, pp. 3–18.

[24] A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *TOCS*, vol. 30, no. 4, pp. 1–26, 2012.

[25] X. Gu, Q. Wang, Q. Yan, J. Liu, and C. Pu, "Sync-millibottleneck attack on microservices cloud architecture." in *Proceedings of the 19th ACM ASIA Conference on Computer and Communications Security*, 2024.

[26] M. Guirguis, A. Bestavros, and I. Matta, "Exploiting the transients of adaptation for roq attacks on internet resources," in *ICNP*. IEEE, 2004, pp. 184–195.

[27] D. Huye, Y. Shkuro, and R. R. Sambasivan, "Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, 2023, pp. 419–432.

[28] Jaeger, "Jaeger: open source, end-to-end distributed tracing," https://www.jaegertracing.io/, 2022.

[29] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites," in *International Conference on World Wide Web*. ACM, 2002.

[30] R. E. Kalman, "A new approach to linear filtering and prediction problems," 1960.

[31] M. S. Kang, S. B. Lee, and V. D. Gligor, "The crossfire attack," in *(S&P'13)*. San Francisco, CA, USA: IEEE, May 2013, pp. 127–141.

[32] Y.-M. Ke, C.-W. Chen, H.-C. Hsiao, A. Perrig, and V. Sekar, "Cicadas: Congesting the internet with coordinated and decentralized pulsating attacks," in *AsiaCCS*. Xi'an, China: ACM, Nov. 2016, pp. 699–710.

[33] R. Kohavi, R. M. Henne, and D. Sommerfield, "Practical guide to controlled experiments on the web: listen to your customers not to the hippo," in *13th ACM SIGKDD*, 2007, pp. 959–967.

[34] R. Kohavi and R. Longbotham, "Online experiments: Lessons learned," *Computer*, vol. 40, no. 9, pp. 103–105, 2007.

[35] A. Kuzmanovic and E. W. Knightly, "Low-rate tcp-targeted denial of service attacks: The shrew vs. the mice and elephants," ser. SIGCOMM '03, p. 75–86.

[36] Z. Li, H. Jin, D. Zou, and B. Yuan, "Exploring new opportunities to defeat low-rate ddos attack in container-based cloud environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 695–706, 2019.

[37] J. Liu, Q. Wang, S. Zhang, L. Hu, and D. Da Silva, "Sora: A latency sensitive approach for microservice soft resource adaptation," in *Proceedings of the 24th International Middleware Conference*, 2023, pp. 43–56.

[38] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM SoCC*, 2021, pp. 412–426.

[39] S. Luo, H. Xu, K. Ye, G. Xu, L. Zhang, J. He, G. Yang, and C. Xu, "Erms: Efficient resource management for shared microservices with sla guarantees," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, 2022, pp. 62–77.

[40] X. Luo and R. K. Chang, "On a new class of pulsing denial-of-service attacks and the defense," in *NDSS*, San Diego, CA, USA, Feb. 2005.

[41] G. Maciá-Fernández, J. E. Díaz-Verdejo, P. García-Teodoro, and F. de Toro-Negro, "Lordas: A low-rate dos attack against application servers," in *CRITIS*. Springer, 2007, pp. 197–209.

[42] T. Mauro, "Adopting microservices at netflix: Lessons for architectural design," https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, 2015.

[43] Microsoft, "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, 2023.

[44] G. Oikonomou and J. Mirkovic, "Modeling human behavior for defense against flash-crowd attacks," in *2009 IEEE International Conference on Communications*. IEEE, 2009, pp. 1–6.

[45] N. K. Pamela Weaver, "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/, 2021.

[46] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "{FIRM}: An intelligent fine-grained resource management framework for slo-oriented microservices," in *14th ({OSDI} 20)*, 2020, pp. 805–825.

[47] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009, pp. 77–84.

[48] S. Ranjan, R. Swaminathan, M. Uysal, A. Nucci, and E. Knightly, "Ddos-shield: Ddos-resilient scheduling to counter application layer attacks," *TON*, vol. 17, no. 1, pp. 26–39, 2009.

[49] M. A. Sánchez, J. S. Otto, Z. S. Bischof, D. R. Choffnes, F. E. Bustamante, B. Krishnamurthy, and W. Willinger, "Dasu: Pushing experiments to the internet's edge." in *NSDI*, 2013, pp. 487–499.

[50] C. Satnic, "Amazon, microservices and the birth of aws cloud computing," https://www.linkedin.com/pulse/amazon-microservices-birth-aws-cloud-computing-cristian-satnic/, 2013.

[51] H. Shan, Q. Wang, and C. Pu, "Tail attacks on web applications," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1725–1739.

[52] H. Shan, Q. Wang, and Q. Yan, "Very short intermittent ddos attacks in an unsaturated system," in *International Conference on Security and Privacy in Communication Systems*. Springer, 2017, pp. 45–66.

[53] A. Sriraman and T. F. Wenisch, "{μTune}:{Auto-Tuned} threading for {OLDI} microservices," in *OSDI 18*, 2018, pp. 177–194.

[54] C.-A. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 361–376.

[55] S. Subashini and V. Kavitha, "A survey on security issues in service delivery models of cloud computing," *Journal of network and computer applications*, vol. 34, no. 1, pp. 1–11, 2011.

[56] A. Toh, "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/, 2022.

[57] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu, "Detecting transient bottlenecks in n-tier applications through fine-grained analysis," in *ICDCS*. IEEE, 2013, pp. 31–40.

[58] Q. Wang, C.-A. Lai, Y. Kanemasa, S. Zhang, and C. Pu, "A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 207–217.

[59] J. Xiong, M. Wei, Z. Lu, and Y. Liu, "Warmonger: Inflicting denial-of-service via serverless functions in the cloud," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 955–969.

[60] S. T. Zargar, J. Joshi, and D. Tipper, "A survey of defense mechanisms against distributed denial of service (ddos) flooding attacks," *IEEE communications surveys & tutorials*, vol. 15, no. 4, pp. 2046–2069, 2013.

[61] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "{CRISP}: Critical path analysis of {Large-Scale} microservice architectures," in *USENIX ATC 22*, 2022, pp. 655–672.