Reichman University

Efi Arazi School of Computer Science

M.Sc. program - Research Track

The Rolling Tandem of Micro-services: DDoS attack on micro-service auto-scaling mechanism

by

Michael Czeizler

M.Sc. dissertation, submitted in partial fulfillment of the requirements

for the M.Sc. degree, research track, School of Computer Science

Reichman University (The Interdisciplinary Center, Herzeliya)

May , 2023

This work was carried out under the supervision of **Prof. Anat Bremler-Barr** from the Efi Arazi School of Computer Science, Reichman University.

Abstract

Auto-scaling is a primary ability of cloud computing which allows systems to adapt to fluctuating traffic loads by dynamically increasing (scale-up) and decreasing (scale-down) the number of resources used. Today's software development landscape has witnessed a shift towards microservices architectures. Using this approach, large software systems are implemented by loosely-coupled services, each responsible for specific task and defined with separate scaling properties. We show that when microservices with separate auto-scaling mechanisms work in tandem to process traffic, they can overload each other. This overload can result in throttling (Denial of service - DoS) or the over-provisioning of resources (Economic Denial of Sustainability - EDoS). This paper demonstrates how an attacker can exploit the tandem behavior of microservices with different auto-scaling mechanisms to create an attack we denote as the *Tandem Attack*. We demonstrate the attack on a typical *Serverless* architecture and show that using managed infrastructure does not liberate from strict planning of scaling definitions.

Contents

List of Figures

List of Tables

1 Introduction

Auto-scaling is a technique widely used to automatically adapt applications to dynamic loads of tra c by increasing (scale-up) and decreasing (scale-down) the number or capacity of the handling resources. As software development shifted to micro-services architecture, large software systems are nowadays composed of many independent microservices, each has its own codebase and responsible for speci c tasks. The communication between the micro-services is usually done using standard protocols as HTTP, RPC and asynchronous messaging. The breakdown to fragmented applications is also re ected in infrastructure management, where di erent services of the same application are given di erent hardware con gurations, policies and scaling properties [14]. Even though created to accelerate software development [11] and drive e cient utilization of hardware resources the microservices approach also presents a new challenge - as systems grow larger, incoming tra c can triggers multiple calls between microservices to handle each request [16]. In this paper, we show that when microservices working in tandem to process tra c and have a separate scaling policy, they may overload each other resulting in throttling (DoS) or over-provisioning of resources (EDoS). In large applications, multiple microservices working in tandem might a ect each other in such a scenario, hence, the rolling tandem of micro-services. We demonstrate the attack, on a common and recommended Serverless architecture [3] comprised of AWS Lambda for code execution and DynamoDB. The Tandem attack on our system architecture involves an attacker that overloads the system with requests, thereby exploiting the interaction between the Lambda's fast scale-up time and the DynamoDB's relatively slow scale-up. The Lambda functions are triggered by the malicious tra c but the DB fails to keep up with the execution rate of the Lambda functions. This results in economic loss due to the increased cost of the Lambda functions, alongside a degradation in performance that impacts legitimate tra c.

We analyze the attack on di erent con gurations of the DynamoDB: Provisioned and On-demand that implement di erent auto-scaling strategies. The paper gener-

ally analyzes using well-known cyber-security and DDoS terminology (Plain DDoS and Yo-Yo attack [8]) however self-in icted Tandem attacks can occur due to faulty con g-urations and design [23] [12].

We show that using the YoYo attack pattern [8], which creates bursts of periodic high tra c, the database is forced into both situations of denial of service and severe under-utilization resulting in waste of cost. We also indicate that while on-demand mode handles scale adaptation quickly the cost of absorbing the full force of the attack is meaningful. We present a model of the Tandem attack, and experiments' results, and demonstrate the trade-o between providing high performance and limiting expen-ditures.

We also discuss the implication of the current provisioned mode policy of AWS, which limits the numbered of allowed scale-down operations per day. We show that an attacker can exploit this limitation in order to create a high nancial damage.

Unlike large-volume DDoS attacks targeting network equipment, the Tandem attack is focused on the application layers and requires the attacker to invest far less resources. The attacker also bene ts from the ease of understanding the impact of the attack simply by recording and analyzing response time metrics in real time, as our testing control plane did. This simplicity empowers crafting sophisticated attacks with auto-adjustment capabilities. The di erentiation between network and application attacks is also meaningful since the protection on the network layer is typically outsourced to external scrubbing services and are o ered by the cloud providers [4], while the application and infrastructure management are managed by the system administrators. The experiments focused on serverless managed infrastructure to suppress the notion that serverless services are fully handled by the cloud provider and do require strict planning when it comes to scaling de nitions.

We lay out possible feasible mitigation for the Tandem attack, including back-pressure mechanisms and layout future research areas to be explored to create com-prehensive solutions.

We also show that Retry strategies, which is a common way to compensate for

throttling when writing to the DB fails, is ine ective when the power of the attack is relatively high.

As far as we are aware the paper is the rst to raise awareness on the subject, and de ne the issues rising from Tandem services and with out-of-sync auto-scaling.

2 Background on Serverless services

Our work spotlights the importance of planning properly the auto-scaling properties of micro-services and the potential rami cations from exploiting their limitations, whether created by either technical limitation or by mistakes in con gurations. For demonstrating those principles, we chose fully managed popular AWS Serverless architecture: Lambda functions and DynamoDB.

Serverless computing is a derivative of cloud computing that refers to a system architecture and execution model in which the cloud provider is managing and dynamically provisions the computing power and infrastructure required to run the software. When using serverless services the system administrator has limited or no control over the underlying servers, and is responsible to con gure only the service itself.

2.1 AWS Lambda

AWS Lambda is a compute service that executes code without the user's involvement in deploying or managing the underlying infrastructure. The user writes the code in lambda functions. In fact, the code is executed inside containers that the lambda service provisions. Lambda is less suitable for long-running processes since it has an execution time limit and because of relatively high pricing compared to user-controlled compute as EC2. According to AWS docs [5], lambda can handle sudden bursts of up to 3000 executions and scale at a rate of 500 executions per minute with no maximum concurrency limitation. The lambda cost is a dominated by of the allocated hardware resource and execution time.

2.2 DynamoDB

DynamoDB is a popular serverless key-value document database service by AWS. DynamoDB's throughput and billing are de ned by virtual capacity units, measured by the size of the data written or read from the database's table, thus dictating the number of allowed parallel operations. A single write capacity unit (WCU) is consumed by one write request to the database for an item up to 1 KB in size. If the inserted item is larger than 1 KB, DynamoDB will consume additional write capacity units. As our experiments implemented only write operation we will not elaborate on reading from the DynamoDB which is measured in a similar manner.

There are two modes of auto-scaling in DynamoDB:

1. On-demand mode

   DynamoDB on-demand o ers pay-per-request pricing for read and write requests and is mostly suitable for cases were the volume of tra c to is hard to predict or volatile. DynamoDB automatically adjusts the database tables' capacity to double of the previous peak tra c in a certain time window. According to AWS [6] even in case of tra c bigger than twice of the previous peak DynamoDB will allocate enough resources to prevent throttling except if the time frame of doubling the tra c is shorter than 30 minutes.

2. Provisioned mode

   When using the provisioned mode, the user de nes the number reads and writes (in capacity units) per second that a table can endure. Pre-de ned capacity is preferred when the upper bounds of the request in known or to limit costs. When using auto-scaling with the provisioned mode the user de nes an interval of lower and upper limits for both read and write capacity. The user also de nes a target utilization percentage that DynamoDB seeks to maintain. In practice, the utilization percentage de nes the extra reserve capacity the user is willing to pay for in order to absorb a sudden spike in tra c. E.g, a table receiving tra c consuming 1000 WCUs per second with utilization target percentage set to 80%

will yield provisioned capacity of 1250 WCUs (assuming 1250 is not crossing the upper bound). The AWS documentation does not provide details on the time provisioned mode requires to detect changes in tra c and adjusting the capacity accordingly.

3 Related work

Cloud applications are resilient to many of the classic DDoS (i.e., attackers operate a at vector attacks) due to their high bandwidth pipe, built-in anti-spoo ng mitigation in the load-balancers , and the common use of a content distribution network (CDN).

However, cloud applications are vulnerable to attacks on the application level. Amazon lists the auto-scaling mechanism such as using AWS layer DDoS mitigation as one of the best practices for dealing with Distributed Denial of Service (DDoS) [1] attacks that target web applications. In at DDoS attack the auto-scaling mechanism translates a DDoS attack into an Economic Denial of Sustainability attack (EDoS) [22, 20], incurred by paying the extra resources required to process the bogus tra c of the attack. Several work have tried to mitigate EDoS attacks [21, 7, 15, 17]. Their proposed solutions are focused on the classi cation of the clients and the ability to determine whether the user is legitimate or whether it is a malicious bot.

YoYo attack [9], is a kind of burst attack, that operates against the auto-scaling mechanism in the cloud, speci cally, it was shown e ective on auto-scaling mechanisms and Kubernetes mechanisms [13].

In basic burst attack the victim is attacked with recurring high-volume tra c bursts, lasting from a few seconds up to several minutes, while the total duration of the attack can span hours and even days. In 2021, it was reported that 50% of organizations experienced such attacks [19]. B A burst attack, when carried out correctly, is cost-e ective to the attacker, who can divert the attack tra c between multiple end-point targets, leveraging a high degree of resource control. Moreover, it confused conventional DDoS mitigation and detection solution [10].

In the Yo-Yo attack, the periodic bursts of tra c loads cause the auto-scaling mechanism to oscillate between scale-up and scale-down phases. The YoYo attack causes signi cant performance degradation in addition to economic damage.

During the repetitive scale-up process, which usually takes up to a few minutes, the cloud service su ers from a substantial performance penalty. When the scale-up process nishes, the attacker stops sending tra c and waits for the scale-down process to start. When the scale-down process ends, the attacker begins the attack again, and so on. Moreover, when the scale-up process ends, there are extra machines, but no extra tra c. Thus, the victim pays unwittingly for extra machines that are not utilized. Recently it was shown that companies were attacked by Yo-Yo attack [10], and various mechanisms suggested as mitigation. All of these works ignore the e ect of the Tandem auto-scaling mechanisms, which is presented in this paper.

4 Threat Model

The suggested threat model is closely tied with the experiments conducted as part of this work and also relies on common use cases of system architecture. When an attacker wish to mount a Tandem attack on a microservices based application, we assume the following capabilities are needed:

- The attacker is able to overload requests to the application - It is likely that unauthenticated parts of applications will be a vulnerable target as they do not require login operation by a user. Such elements can be website search elds, forms or any user facing component that triggers server-side computation. Also applications containing user validation or require authentication might be a target as these mechanisms can have automated counter measures by attackers [18].
- The attacker can launch an attack with no knowledge on the internal architecture of the attacked system - When a system is lacking resources to match incoming tra c it will be detectable by changes in both of response time and success rate. An attacker recording and analyzing the impact of these

basic metrics can asses the impact of the load and created automated processes to optimize the attack. As many of modern applications are comprised of multiple tiers and services [16] the Tandem e ect can be witnessed without the insight on which exact service is causing the slowdown of response. In Tandem attack using Yo-Yo or other patterns aimed at creating scale-up actions, the attacker can optimize the attack by deriving the scale-up and down from changes in round trip time. However, an attacker can ne-grain his moves by having basic knowledge of the target's architecture. Some possible methods to gain such knowledge can be obtained is by following career posts, inducing technical sta and reviewing technical blogs. An attacker with prior knowledge can take some assumptions on a system which can assist in optimizing the attack.

5 Tandem Attack

We demonstrate Tandem attack on the following simple architecture, consists of a load generator, AWS API Gateway, AWS Lambda, a service that executes code functions, and AWS DynamoDB. Each lambda, triggered by an HTTP request, performs a write request to a DynamoDB table. In order to achieve high throughput the lambda functions are triggered and executed in parallel resulting parallel write requests to the DB. Our tests were crafted so each HTTP request to the system will be followed by a single lambda execution that is performing a write operation that consumes exactly 1 WCU (Write Capacity Unit). The 1-to-1 mapping was done to simplify the analysis of the attacks and model, where in actual production scenarios, the ratio will not be xed. We note that in common use-cases the microservices call graphs are more complex with longer call chains [16] and therefore making the e ect of tandem attack harder to detect and mitigate.

The experiment is comprised of multiple worker processes, generating HTTP requests, and a main control plane process that collects data and orchestrates the experiment steps.

Figure 1: Experiment architecture

In order to connect between the lambda functions and the incoming HTTP request we have used an API Gateway, de ned to trigger a lambda function per each HTTP request received.

When sending the HTTP response the lambda also sends back information on the success or fail of the write operations to the DB and insertion time. That information is received by the workers and sent to the control plane with total round trip time of each request.

The control plane aggregates the success ratio from all requests every 10 seconds and quotes from the DB the number of provisioned WCUs. Developing the control plane in that manner enabled making dynamic decision on whether to increase or decrease the power of the attack. Similarly, an attacker is able to understand the attack's e ectiveness simply by counting failed requests and measuring total execution time of the HTTP requests.

The worker processes generate tra c simulating both benign and hostile users and the control plane does not distinguish between them when assessing the system's states.

The lambda functions contain the application's logic and executing it at high-scale is aimed to exhaust DynamoDB's resources and a ect its ability to serve incoming tra c.

The lambda parallel execution was con gured so no HTTP requests will be throttled by the lambda so each lambda will execute a write operation.

The synthetic example in gure 6 shows a table created in provisioned mode with minimum capacity of 500 WCUs and maximum capacity of 1500 WCUs. The steady

state (benign) tra c of the application is 1000 requests per second and the provisioned capacity at minute 0 is also 1000 WCUs per second. The attack follows the Yo-Yo pattern with bursts of tra c peaking at 3000 requests per second. As the DB scales up to match the high load it reaches the limit of 1500 WCUs. Once the upper limit is reached the attacker halts from sending requests while the DB remains at 1500 WCUs. Following the scale-down of the DB to the steady state of 1000 WCUs the attacker resumes the attack forcing the DB again to scale-up to 3000 WCUs. The economical damage is re ected by lambdas executed while the DB is not scaled to sustain the writes, attack tra c served by the both lambdas and DB and the time the DB remains scaled during the o period of the attack. The performance damage takes place during the periods of the attack while the DB is not scaled to handle tra c resulting in dropped requests. All presented experiments in this paper follow the same method of analysis and modeling of results.

Technical notes

- DynamoDB has other types of capacity units for verifying write consistency and transactions. We did not relate to any of them since they were not used in our experiments and adhere to the same principles.
- To simplify the presentation of the synthetic examples, we assumed a target utilization of 100% even though the maximum allowed is 90%.
- AWS provides a set of con gurations to manage and limit the number of concurrent lambda function per the entire AWS account and per speci c function. We con gured the lambda in a manner that it will be able to serve all incoming tra c.
- The actual cost of lambda functions is mostly a ected by the total number of lambda requests, execution time, data transfer and storage used.

6 Modeling the Tandem Attack

In this section, we de ne the model used for analyzing the Tandem attack. The model was constructed to support comparison between di erent attacks on the two DB modes being tested while gaining insights on the e ect of the attack.

Parameter	De nition	Given by
r	Average requests rate of benign tra c	System usage
$S_{up}^{service}$ n$S_{down}^{service}$	Scale-up and scale-down time	Cloud provider of
$Cost_{service}$	Cost of one unit from the service	managed service
Service	AWS lambda or DynamoDB	Application
L_{upper}	Upper capacity limit of the DB (relative to the steady state)	System admin
R	Relative reserved capacity of DB (Only in Provisioned mode)	System admin
W_{up} nW_{down}	Relative average capacity consumed during scale up or down	System usage
k	The power of the attack	Attacker
D	The duration of the attack	
n	Number of attack cycles	Yo-Yo Attacker
t_{on} nt_{off}	Time of on-attack and o -attack phases.	
T	Cycle duration in Yo-Yo attack. $T = t_{on} + t_{off}$	

Table 1: Notation used in the model description

6.1 Model Properties

1. Scaling

   Incoming requests arrive with an average rate of r requests per time unit (seconds in our use case). Let $S_{up}^{service}$ and $S_{down}^{service}$ be the total time a service requires to nish scaling (up or down). Since our experiment used serverless services (lambda, $DB_{Provisioned}$ or $DB_{OnDemand}$) the scaling time is dictated by the system, and takes into account all operations by the service until the new capacity is available. Such operations include detection of metrics and taking decision to scale, spinning of new resources and notifying on availability of the new capacity. In other types of infrastructure some of those parameters are usually de ned by the system administrator and have better visibility.

The scaling time (up or down) of lambda is negligible and shall be accounted as zero when using the model. The scaling time of DynanmoDB depends on the de ned work mode - provisioned or on-demand.

- DynamoDB Provisioned mode - There are three con gurable parameters for auto-scaling that are de ned by the DB administrator: L_{upper} and L_{lower} de ne the interval of lower and upper limits of the DB's table capacity limits as relative to the steady-state tra c. E.g, if the DB is handling a rate of r requests and the upper limit is con gured to handle $3\ r$ then $L_{upper} = 2$. The third parameter R de nes the utilization target. E.g, if a DB is con gured to work at 80% utilization and during steady state receives r requests then it can instantly accommodate 25% more tra c ($r\ \frac{1}{0.8} = 1.25\ r$) but with increased cost of $1.25\ r\ cost_{DB\ Provisioned}$. We de ne R to be the relative reserved capacity (in our example 0.25). Since the models relate only to attack scenarios we relate only to L_{upper} in the parameters table and descriptions. We also consider R to be negligible during DDoS situation as tra c reaches the maximal capacity de ned by L_{upper}, quickly consuming R in the process. The tables for each attack in the damage analysis section present these parameters with accurate numbers calculated from the experiments data. The cost of one unit any service, is de ned by $Cost_{service}$. In the presented model the cost was considered as a constant value, but in real use cases the actual value will usually depends on multiple parameters. E.g, lambda's cost is comprised mostly of execution time and allocated memory but involve many other parameters[3].
- DynamoDB On-demand mode - As explained in Section 2, when using on-demand mode there are no capacity limits (meaning L_{upper}, L_{lower} are unde ned) and eventually the auto-scaling will adjust to handle any rate of incoming tra c. Moreover, since there is no S_{down} the cost is always proportional to the actual requests served and the attacker has no bene t from using Yo-Yo attack comparing to plain DDoS attack. Hence, our analysis

relates only to DDoS after auto-scaling was done.

2. Attack

Let k be the power of the attack de ned as the extra power of tra c sent by the attacker. I.e., When attacker sendsk times more malicious requests than the rate of the steady state the total rate of requests is(k + 1) r. In plain DDoS attack, over owing the available resources of a system, the power is assumed to be constant and is (k + 1) r for the entire span of the attack. When usingDB $_{Provisioned}$ we reasonably assume thatk > R + 1 since otherwise no performance damage will be made making any scale-up redundant.

We denote the duration of the attack asD. When implementing the YoYo pattern [8], the attack is created by repetitive bursts of powerk. We denote the adaptive YoYo as a variation of YoYo where the attacker waits for the scale down operation to be complete before sending the next burst. Formally the Yo-Yo attack is composed ofn cycles where each cycle duration isT. T is comprised of anon-attack period, denoted ast_{on}, where the attacker sendsk r requests, and an o -attack period, denoted ast_{off} where the attacker halts. Thus T is equal to $t_{on} + t_{off}$, and the total duration of the attack is D = n T.

7 Damage Analysis

In this section, we derive closed formulas to quantify the damage in icted by attacks with power of k for both plain DDoS and YoYo.

We de ne $Damage_e^{service}(k)$ as the economic damage caused by the attack. We analyze $Damage_e^{service}(k)$ per each service,Lambda and DynamoDB (under the two modes), and assess it as the average extra cost of the service for the span of the entire attack.

We de ne $Damage_p(k)$ to be the performance damage caused by the attack and assess it as the number of incoming requests that failed during the attack. In our use case those are requests that failed to write to the DB.

For each of damage types described above, we derive relative damage formula that de nes the ratio between the damage created by the attack and the corresponding value during steady state. We note that the relative performance damage from the de nition is equal to the failure rate.

7.1 DDoS attack on DynamoDB in Provisioned Mode

Figure 2: DB in provisioned mode under DDoS attack economic damage use case model

Since the lambda is scaled immediately the extra cost of dealing with the attacker's requests is:

$$Damage_e^{lambda}(k) = r \quad k \quad Cost_{lambda} \quad D \tag{1}$$

And hence the Relative damage to the steady state of lambda is exactly

$$RD_e^{lambda}(k) = \frac{r \quad k \quad Cost_{lambda} \quad D}{r \quad Cost_{lambda} \quad D} = k \tag{2}$$

$DB_{Provisioned}$ can adapt immediately only by using the spare capacity R (assum-

Figure 3: DB in provisioned mode under DDoS attack performance damage use case model

ing $1 + R < k$) hence it can sustain a burst of $r \quad (1 + R)$. As we approximate the damage when sustaining a long attack we disregard the time $S_{up}^{DB\ Provisioned}$ to perform the initial scale-up and let D determine the damage. We also assume that $k > 1 + L_{upper}$, as otherwise no meaningful damage will be made. $Cost_{DB}$ is dominated by the cost of the consumed WCU and treated as a constant value.

$$Damage_e^{DB\ Provisioned}\ (k) = r \quad L_{upper} \quad Cost_{DB} \quad D \tag{3}$$

And hence the Relative damage to the steady state of lambda is exactly

$$RD_e^{DB\ Provisioned}\ (k) = \frac{r \quad L_{upper} \quad Cost_{DB} \quad D}{r \quad Cost_{DB} \quad D} = L_{upper} \tag{4}$$

The performance damage is counted as legitimate tra c's failed requests as a result of the attack. We consider a failed request as one that failed to perform a write to the DB. For simplicity, we assume that in steady state the number of failed requests is 0.

As before, we disregard the spare capacity dictated by R and the initial time $S_{up}^{DB\ Provisioned}$ as they are negligent when assessing the entire DDoS attack.

$$Damage_p^{DDOS\ Provisioned}(k) = (r \quad r \quad \frac{1 + L_{upper}}{1 + k}) \quad D \tag{5}$$

$$RD_p^{DDOS\ Provisioned}(k) = \frac{(r \quad r \quad \frac{1+L_{upper})}{1+k} \quad D}{r \quad D} = 1 \quad \frac{1 + L_{upper}}{1 + k} \tag{6}$$

7.2 YoYo attack on DB in Provisioned mode

Figure 4: DB provisioned mode under YoYo attack economic damage use case model

We de ne the model under the assumption that the attacker's main goal is to optimize the attack ,meaning, maximize the economic damage while minimizing the resources used for the attack. Therefore, the attack should be active long enough to scale up the DB with performance damage created in the process is collateral damage.

Figure 5: DB provisioned mode under YoYo attack performance damage model

Figure 6: DB provisioned mode under Adaptive YoYo attack economic damage use case model

Hence, we de ne the optimal attack duration to be:

$$t_{on} = S_{up}^{DB\ provisioned} \tag{7}$$

Figure 7: DB provisioned mode under Adaptive YoYo attack performance damage model

Figure 8: DB provisioned mode under Adaptive YoYo attack economic damage - Experiment results

$$t_{off} = S_{down}^{DB\ provisioned} \tag{8}$$

The model describes the adaptive YoYo pattern as featured in gure 6 but can easily be transformed to xed pulses.

Since thelambda is scaled up and down immediately the amortized extra cost of

Figure 9: DB provisioned mode under Adaptive YoYo attack performance damage - Experiment results

dealing with the attacker requests is:

$$Damage_e^{lambda}(k) = n \; r \; k \; Cost_{lambda} \; t_{on} = n \; r \; k \; Cost_{lambda} \; S_{up}^{DB\ provisioned} \quad (9)$$

And hence the Relative damage to the steady state of lambda is exactly:

$$RD_e^{lambda}(k) = \frac{n \; r \; k \; Cost_{lambda} \; S_{up}^{DB\ provisioned}}{n \; r \; Cost_{lambda} \; T} = k \; \frac{S_{up}^{DB\ provisioned}}{T} \quad (10)$$

When considering the economic damage working under provisioned mode we account both $S_{up}^{DB\ provisioned}$ and $S_{down}^{DB\ provisioned}$. When scaling down $Cost_{DB}$ comes from over provisioned WCUs, denoted as W_{down}, that are serving no tra c. When scaling up the cost is a ected from actual writes done by the attacker's tra c with the average relative tra c served denoted as W_{up}.

Hence, the total economic damage related to the DB is:

$$Damage_e^{DB\ Prov:}(k) = \quad (11)$$

$$= n \; r \; Cost_{DB} \; ((1 + W_{up}) \; S_{up}^{DB\ Prov:} + (1 + W_{down}) \; S_{down}^{DB\ Prov:})$$

with relative damage de ned by:

$$RD_e^{DB\ Prov:}(k) = \tag{12}$$

$$= \frac{n\ r\ Cost_{DB}\ ((1+W_{up})\ S_{up}^{DB\ Prov:} + (1+W_{down})\ S_{down}^{DB\ Prov:})}{n\ r\ (1+R)\ Cost_{DB}\ T}$$

$$= \frac{((1+W_{up})\ S_{up}^{DB\ Prov:} + (1+W_{down})\ S_{down}^{DB\ Prov:})}{(1+R)\ T}$$

The performance damage is estimated here as the average failure requests as a result of the attack. Note that there is a failure rate only during T_{on} time which is equal to S_{up}.

Hence the failure rate is:

$$Damage_p^{YOYO_{Prov:}}(k) = n\ (r\ \ r\ \frac{1+L_{upper}}{1+k})\ S_{up}^{DB\ prov:} \tag{13}$$

$$RD_p^{YOYO_{prov:}}(k) = \frac{n\ (r\ \ r\ \frac{1+L_{upper}}{1+k})\ S_{up}}{n\ T\ r} = (1\ \ \frac{1+L_{upper}}{1+k})\ \frac{S_{up}^{DB\ prov:}}{T} \tag{14}$$

7.3 DDoS attack on DynamoDB in On-demand mode

We analyze the On-demand after the auto-scaling has ended. Hence, we can trivialy deduct that the scaling of the lambda and the DynamodDB is by a factor of k and there is no performance damage. Therefore,

$$RD_e^{lambda}(k) = k \tag{15}$$

$$RD_e^{DB\ OnDemand}(k) = k \tag{16}$$

Figure 10: DB On-Demand mode under DDoS attack economic damage use case model

Figure 11: DB On-Demand mode under DDoS attack performance damage use case model

$$RD_{p}^{DDOS\ OnDemand}(k) = 0 \quad (17)$$

Figure 12: DB On-Demand Scale-up behaviour

Figure 13: DB On-Demand Scale-up behaviour performance

7.4 Yo-Yo attack on DB in On-demand mode

After enough scaling of Yo-Yo attack, the scale up in On-demand will be almost immediately and hence:

$$RD_e^{lambda}(k) = k \quad \frac{t_{on}}{T} \tag{18}$$

Figure 14: DB On-Demand under DDoS attack economic damage - Experiment results

Figure 15: DB On-Demand under DDoS attack performance damage - Experiment results

$$RD_e^{DB\ OnDemand}(k) = k\frac{t_{on}}{T} \quad (19)$$

$$RD_p^{Y\ OY\ O_O\ nDemand}(k) = 0 \quad (20)$$

Figure 16: DB On-Demand mode under Yo-Yo attack economic damage model

Figure 17: DB On-Demand mode under Yo-Yo attack performance damage model

8 Cost and Potency Analysis

Potency is a metric de ned to measure the e ectiveness of DDoS attacks [add qoute]. The e ectiveness is de ned as the ratio between the in icted damage and the resources invested in creating it. In order to compare the di erent featured

variations of Tandem attack, we de ne $P_e^{attack}(k)$, the attack's economic potency, to be the ratio between the relative economic damage $RD_e^{attack}(k)$ and $C^{attack}(k)$, the cost of mounting such an attack:

$$P_e^{attack}(k) = \frac{RD_e^{attack}(k)}{C^{attack}(k)} \tag{21}$$

Similarly we de ne the potency for performance damage:

$$P_p^{attack}(k) = \frac{RD_p^{attack}(k)}{C^{attack}(k)} \tag{22}$$

Clearly, an attacker would be interested in maximizing the damage per unit cost, i.e., maximizing the attack potency.

In Yo-Yo attack, the attack cost is directly a ected by the power of attack k and t_{on} period relative to attack cycle length:

$$C^{YoYo}(k) = k \frac{t_{on}}{T} \tag{23}$$

While in DDoS the cost of the attack is exactly k as the attacker's tra c is continuous.

8.1 Discussion

Table 2 shows the summary of the model analysis:

	Tandem: DDoS DB provisioned	Tandem: Yo-Yo DB Provisioned	Tandem: DDoS DB on-demand	Tandem: YoYo DB on-demand
Attack Cost	k	$k \frac{S_{up}}{T}$	k	$k \frac{t_{on}}{T}$
Relative Economic Damage lambda	k	$k \frac{S_{up}^{DB\ prov:}}{T}$	k	$k \frac{t_{on}}{T}$
Relative Economic Damage DB	L_{upper}	$\frac{(1+W_{up})\, S_{up}^{DB\ Prov:} + (1+W_{down})\, S_{down}^{DB\ Prov:}}{(1+R)\, T}$	k	$k \frac{t_{on}}{T}$
Relative Performance Damage	$1 \frac{1+L_{upper}}{1+k}$	$(1 \frac{1+L_{upper}}{1+k}) \frac{S_{up}}{T}$	0	0
Economic damage Potency lambda	1	1	1	1
Economic damage Potency DB	$\frac{L_{upper}}{k}$	$\frac{(1+W) S_{up}^{DB\ Prov:} + S_{down}^{DB\ Prov:}}{k\,(1+R)\, S_{up}^{DB\ prov:}}$	1	1
Performance Potency	$(1 \frac{1+L_{upper}}{1+k}) \frac{1}{k}$	$(1 \frac{1+L_{upper}}{1+k}) \frac{1}{k}$	0	0

Table 2: Model Summary

Parameter	DDoS Prov. Model (Figure 2)	YoYo Prov. Model (Figure 6)	YoYo Prov. Exp. (Figure 8)	DDoS On-Demand Model (Figure 10)	DDoS On-Demand Exp. (Figure 14)	YoYo On-Demand Model (Figure 16)
r	1000	1000	1300	1000	4000	1000
k	2	2	2:6	2	2	2
L_{upper}	0.5	0.5	2:6	-	-	-
R	0.1	0.1	0.1	-	-	-
W_{up}	0.25	1	1:5	-	-	2
W_{down}	-	1	2:23=2:5	-		-
S_{up}/t_{on}	36	5	7/7	36	31	5
S_{down}/t_{off}	-	21	26/60	-	-	7
T/D	36	26	33/67	36	31	12

Table 3: Parameters values of use case example

	Tandem: DDoS DB provisioned Model	Tandem: Yo-Yo DB Provisioned Model	Tandem: Yo-Yo DB Provisioned Experiment	Tandem: DDoS DB on-demand Model/Experiment	Tandem: YoYo Yo-Yo on-demand Model
Attack Cost	2	0:38	0:55=0:27	2	0:83
Relative Economic Damage lambda	2	0:38	0:55=0:27	2	0:83
Relative Economic Damage DB	0:5	1:81	2:79=3:08	2	0:83
Relative Performance Damage	0:5	0:27	0/0	0	0
Economic damage lambda Potency	1	1	1	1	1
Economic damage DB Potency	0:25	4:73	5.06/11.36	1	1
Performance Potency	0:25	0:25	0/0	0	0

Table 4: Comparison of all models and experiments. In Yo-Yo on provisioned mode we analyzed based on the 2 separate scaling properties due to DynamoDB's scale-down limit which is detailed in the discussion.

The experiments conducted to evaluate the Tandem attack model with meaningful tra c are brought alongside the modeled attack scenarios as described in tables 3 and 4. When usingProvisioned modeunder DDoS pattern we can observe that the damage is both economic and performance. In such situation, the DB is running out of available resources as soon as the initial scale up is nished while 50% of benign tra c fail to write to the DB. The economic damage is meaningful as all triggered lambda function are billed while most of the DB resources(66%) are serving malicious tra c. As DDoS attacks are continuous the potency to the attack is higher comparing to intermittent attack pattern, making it less lucrative

for long periods of time.

We evaluate the Yo-Yo attack on the DB in Provisioned mode using both synthetic data and experiment results. The model, described in gure 6, di ers from the experiment by creating tra c that exceeds the upper bound for scaling and therefore yields more damage - performance and economic. When comparing the synthetic model to at DDOS model we can witness that even though less performance damage is in icted (27% vs 50%), the attack is more e cient when evaluating the economic damage and potency as the attacker is required to invest fewer resources in creating the bursts xating the DB to maximal capacity.

The experiment described in gure 8 shows Tandem attack using the Yo-Yo pattern on lambda and DynamoDB in Provisioned mode. The data collected during the experiment included both the actual WCUs consumed, calculated by the number of successful writes to the DB and the reported WCUs capacity by AWS.

From the chart we can see that the actual capacity was reported in delay of approximately 6 minutes. This can be explained by the fact the DynamoDB's scaling mechanism is based on sampling metrics every 6 minutes as will be discussed in detail when analyzing the attack. Another observed behavior of the DB was a similar delay in actual time WCU capacity was restricted after scaling-down was reported. This required to ne tune the attack by giving a grace period of 6 minutes after the DB reported scaling down as shown in the gure 8.

Lastly, DynamoDB in provisioned mode contains a limitation on the number of scale-down actions that can be performed per day [2]: On a given day, 4 decreases can be done in the 1 hour, as long as no previous decreases happened on that day, those can be followed by an additional single decrease in any hour that follows. This brings that total decreases to 27 per single day (4 for the rst hour followed by a single decrease in the others). As validated with AWS support this number can be increased to 50 decreases per day.

Our experiment (gure 8) presents adaptive YoYo attack up to minute 105, in which DynamoDB's limit for decreases per one hour was exhausted. As a result,

between minutes [105,177] we can observe that the DB fails to scale-down 7 times while being at peak capacity without any malicious tra c in action. Table 4 analyze both parts of the experiment separately as utilizing the speci c limit yielded much severe results, application wise, when evaluating the economical potency (5.06 vs. 11.36).

When evaluating On-demand modewe present two experiments. Figure 12 describes an experiment created to showcase the scaling behavior of the DB by performing a long DDoS attack starting and remaining at around 8300 requests per second. The DB's table initial capacity start at 4000 and we can observe 2 meaningful scale-up actions performed with the latter doubling the initial number of WCUs. As described by the AWS documentation [6], when required to scale-up to more than double of the previous peak tra c under 30 minutes, we should expect throttling. This can be observed in gure 13.

The experiment described in gure 14 presents a more realistic tra c pattern, where we provide the DB more time to perform the initial scaling action. As observed, even under 30 minutes, the DB was able to adapt quickly to the increased tra c. When quantifying the attack we disregard the initial drops of requests for On-demand modeas on the long-term they would become minor.

Since On demand mode does not scale-down resources and rapidly adapts to increased tra c it dismantles Yo-Yo attack's (gure 16) ability to trick scaling mechanism but with the price for serving both hostile and friendly tra c. We discuss and elaborate on this trade-o and possible mitigation's in the next sections.

9 Mitigation Strategies

In this section we outline a number of possible solution to issues raised in the article. The suggested mitigation relate to the general use-case of Tandem behavior of micro-service and not particularly to the architecture referred throughout the work.

3. Limit traffic as close to the origin as possible - limiting the incoming requests closer to their origin is a simplest solution to be implemented that can limit the economic damage in Tandem attack. As demonstrated in our work, the lack of sync between the scale of lambda and DynamoDB created an operational vulnerability that was used to damage the entire chain of services. By limiting the parallel execution of the lambda to match the DB's ability to serve traffic would have eased the economic damage significantly. Implementing similar approach in large system pose a challenge as, in practice, micro-service architectures can implement complex service graphs [16]. In such system services receive traffic from multiple others and might be located in deeper layers of the service call-graph making the understanding of the cross-correlations between services cumbersome. A possible, yet not full solution, would be to of identify the sensitive services that scale slowly and map the service chains that affect them rather than the entire service graph. Such services will likely be IO or storage bounded as databases or have unique scaling traits as DynamoDB.

4. Consume cloud services that scale quickly - Denoting the difference results comparing both modes of DynamoDB we can see that On-demand mode yielded superior results handling shifting load and if combined with limiting the lambda would have served as a relatively solid solution. By using infrastructure services that can scale quickly a system can become robust and sustain peaks in traffic.

5. Implementing retries - Figures 18 and 20 present two similar experiments and the effect of implementing a retry mechanism. It shows that retry can compensate for the lack on synchronicity of connected services. This mitigation should be

considered under three caveats. First, it might result a signi cant decrease in latency. In gures 19 and 21 we can see a signi cant increase of in both insertion time and as a derivative in the response latency. In real-time system such a drop is likely not possible. Second, when k is large, the retry mechanism might prolong the e ect of the attack making a short burst by an attacker to span for a longer period of time increasing the attacker's potency. Third, increasing the execution time of code by retry attempts will results in higher economic damage as run-time is the common billing factor in compute resources over in public cloud providers. As retry mechanism tend to be exponential they might result exponential growth in costs.

6. Implementing back-pressure control planes - For cases as the lambda-DynamoDB architecture, where both services reside in the same cloud provider and report execution metrics using a mutual service (e.g. Cloudwatch) it is possible to implement a dynamic control plane that given throttling in the receiving service will slowdown or limit the emitting service. Even tough such a solution will likely be limited to small systems, the simplicity of implementation makes it quite feasible solution.

7. Decoy using noise in response time - Our experiments showed that measuring response time and failures are enough to gain understanding attack's impact. By adding random delays to when sending responses during the attack period the defender might be able to mask the system's true state. Without reliable metrics an attacker's automated code performing sub-optimal.

10 Conclusions and Future Work

In this work we de ned and presented the potential issues arising from the Tandem behavior of micro-services. We analyzed the impact of both performance and economic damage under YoYo and DDoS tra c patterns using accurate experiments with signi cant tra c.

Figure 18: DynamoDB On-Demand mode under DDOS attack without retry

Figure 19: DynamoDB On-Demand mode under DDOS attack without retry - RTT and Write insertion time

We have also provided solid evidence of the trade-o between limiting costs and providing better performance. As microservices o er exibility in development and evolution of large software system it is likely to remain the dominant architecture.

With systems becoming larger and more complex the Tandem behavior of microservices will remain an issue to be addressed both by cloud providers , creating new services, and system architects, designing complex applications.

As this work was primarily focused in de ning the problem, implementing feasible

Figure 20: DynamoDB On-Demand mode under DDOS attack with retry

Figure 21: DynamoDB On-Demand mode under DDOS attack with retry - RTT and Write insertion time

mitigation remain a topic to be explored. Key aspect that future work should be focused on is developing generic models that can evaluate many connected services and will be powerful enough to assess systems capacity in real time. As our work focused solely on real-time requests, future analysis should relate also to applications with services connected to each other by asynchronous queues and messaging brokers. Such implementation are extremely popular when implementing microservices architectures and will expose other rami cation from the Tandem e ect.

References

[1] AWS best practices for DDoS resiliency, 2015, https://d0.awsstatic.com/whitepapers/DDoS_White_Paper_June2015.pdf.

[2] AWS, Service, account, and table quotas in amazon dynamodb. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/ServiceQuotas.html

[3] , Aws lambda, 2023. [Online]. Available: https://aws.amazon.com/lambda/

[4] , Aws shield, 2023. [Online]. Available: https://aws.amazon.com/shield/

[5] , Lambda function scaling, 2023. [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html

[6] , Read/write capacity mode, 2023. [Online]. Available: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadWriteCapacityMode.html

[7] Z. A. Baig and F. Binbeshr, Controlled virtual resource access to mitigate economic denial of sustainability (EDoS) attacks against cloud infrastructures, in Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on, pp. 346 353.

[8] A. Bremler-Barr, E. Brosh, and M. Sides, Ddos attack on cloud auto-scaling mechanisms, in IEEE INFOCOM 2017 - IEEE Conference on Computer Communications, May 2017, pp. 1 9.

[9] , Ddos attack on cloud auto-scaling mechanisms, in IEEE INFOCOM 2017 - IEEE Conference on Computer Communications, May 2017, pp. 1 9.

[10] E. Chai, Ddos attacks: Hit and run ddos attack, https://www.imperva.com/blog/hit-and-run-ddos-attack, 2013.

[11] G. cloud, What is microservices architecture? 2023. [Online]. Available: https://cloud.google.com/learn/whatismicroservicesarchitecture?

[12] A. H. Dave Rensin, How to avoid a self-in icted ddos attack cre life lessons, 2023. [Online]. Available: https://cloud.google.com/blog/products/gcp/how-to-avoid-a-self-in icted-ddos-attack-cre-life-lessons

[13] R. B. David and A. Bremler- Barr, Kubernetes autoscaling: Yoyo attack vulnerability and mitigation, in CLOSER 2021 pages 34-44 vol. abs/2105.00542, 2021.

[14] Kubernetes, Running multiple instances of your app, 2023. [Online]. Available: https://kubernetes.io/docs/tutorials/kubernetes-basics/scale/scale-intro/

[15] P. S. M. K. M. Kumar, R. Korra, P. Sujatha, and M. Kumar, Mitigation of economic distributed denial of sustainability (EDDoS) in cloud computing, in Proc. of the Intl'Conf. on Advances in Engineering and Technology, 2011

[16] 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 Symposium on Cloud Computing 2021, pp. 412 426.

[17] I. M. Mary, P. Kavitha, M. Priyadharshini, and V. S. Ramana, Secure cloud computing environment against DDoS and EDoS attacks. Citeseer, 2014.

[18] Packetlabs, Attackers are hacking your system using recaptcha domain hacks, 2021. [Online]. Available: https://www.packetlabs.net/posts/recaptcha-domain-hacks/

[19] 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-mitigatior 2021.

[20] G. Somani, M. S. Gaur, and D. Sanghi, "DDoS/EDoS attack in cloud: a ecting everyone out there!" in *Proceedings of the 8th International Conference on Security of Information and Networks, 2015*. ACM, pp. 169–176.

[21] M. H. Sqalli, F. Al-Haidari, and K. Salah, "EDoS-shield-a two-steps mitigation technique against edos attacks in cloud computing," in *Utility and Cloud Computing (UCC), 2011*, pp. 49–56.

[22] S. VivinSandar and S. Shenai, "Economic denial of sustainability (EDoS) in cloud services using http and xml based DDoS attacks," *International Journal of Computer Application 2012*, vol. 41, no. 20.

[23] B. Ziniman, "Rate limiting in a modern app world," 2022. [Online]. Available: https://youtu.be/w03bRbgPUNc

תקציר

מנגנוני התאמת משאבים באופן אוטומטי (auto-scaling) הוא חלק מרכזי מיכולות בענן ובעזרתם ניתן להתאים את כמות משאבי המחשוב המנוצלים בהתאם לדפוסי תעבורה שונים ודינמיים.
על פי פרדיגמת פיתוח בארכיטקטורת מיקרו-שירותים, מערכות תוכנה מחולקות לרכיבי תוכנה קטנים שיש ביניהם תלות נמוכה כאשר לכל מיקרו-שירות יש הגדרות scaling נפרדות.

בעבודה זו אנו מגדירים ומדגימים את ה-Tandem attack המנצלת את העובדה שמיקרו-שירותים המקיימים מערכת גומלין אינם מסונכרנים במנגנוני ה- auto-scaling שלהם. ניצול חולשה זו יכול להתבטא בגרימת נזק כלכלי (EDOS) או בזמינות המערכת (DOS) ע״י גורמים עוינים אך גם ע״י מנהלי מערכת הטועים בהגדרתה. אנו מדגימים את מתקפת ה-Tandem על גבי ארכיטקטורת Serverless פופולרית המייצגת שני מיקרו-שירותים. אנו מראים שהוצאת ניהול השרתים מאחריותו של המשתמש לא רק שאינה פותרת את הבעיה אלא אף עלולה להציג מורכבויות נוספות.

ארכיטקטורת המערכת המוצגת בניסויים בתזה זו מורכבת משני רכיבי תשתית מנוהלים בענן של AWS ומהווים ארכיטקטורה נפוצה מאד : Lambda function ו-DynamoDB.
DynamoDB הוא בסיס נתונים פופולרי מכיוון שאינו דורש ממפעיליו לנהל נפח אחסון או משאבי מחשוב ונדרש רק להגדיר תצורות לטבלאות השונות בו ע״מ שיוכלו לתמוך בתעבורה אליהן. הגדרת התצורה והחיוב של DynamoDB מוגדרים ע״י יחידות וירטואליות שנגזרות מנפח התעבורה לטבלאות השונות.

אנו בוחנים את המערכת בשתי תצורות עבודה מוכרות של DynamoDB : On-demand ו- Provisioned ותחת תעבורה המדמה מתקפת DDOS ומתקפת Yo-Yo. העבודה מציגה מודל השוואתי המאפשר להעריך את התנהגות המערכת בזמן המתקפות השונות. תוצאות הניסויים והשימוש במודל מדגימים היטב את היחס ההפוך בין מערכת זמינה ויקרה למערכת זולה יותר אך פגיעה לתנודות בתעבורה הנכנסת. בניגוד למתקפת DDOS המוכרות מתחום הרשתות, המתקפות המתוארות בעבודה זו אינן דורשות משאבים בסדר גודל משמעותי ע״מ לגרום לנזק במערכת תוכנה. כמו כן, אנו מנתחים מה נדרש מתוקף ע״מ לקיים מתקפות מהצורה המוצגת.

לאחר הגדרת המודל והצגת תוצאות הניסויים אנו מציעים מספר דרכי התמודדות ישימים עם התופעות המוצגות וכיווני מחקר עתידיים.

לטובת הניסוי פותחה תשתית ניסוי ייעודית המאפשרת איסוף נתונים בקנה מידה רחב והערכת תוצאות בזמן אמת ע״מ לייצר ניסויים מורכבים ורבי שלבים המאתגרים את התשתית הנבדקת.

!

!

!

- , +$$&*) ($ ' &%$#"! *

"10 , /!#.- , +!#*( )!#')!('&%$#"!

&%*653&'('43&2&&1 - ./0.,&%$+&*)(" '&"'%$#"!

&

&

&

!0/! , , - .!)+*) (!"#$%& '

83@*?(!0>!#56789:#;<$=!43121(

! A?)3@?>

$) ,!

)%(""'&%$#"!&

!

&& 67897&5)4#)&03#%&%."/&2+.&%#+10$()&/.-,&%+*#)(&('%&%$#"!

&>) ,110&%=140"1<#3&;"+-) (&1!$) .&1'03&1:3&0:4&%1""&10/-) (&.#.4)"

!

@A@@&?0)&