# Moving Target Defense for diversification of microservices on Kubernetes


1st Kenneth González Pineda
*Escuela de Ciencias de la Computación e Informática*
*Universidad de Costa Rica*
San José, Costa Rica
kenneth.gonzalez@ucr.ac.cr

2nd Francisco Rodríguez Zamora
*Escuela de Ciencias de la Computación e Informática*
*Universidad de Costa Rica*
San José, Costa Rica
francisco.rodriguezzamora@ucr.ac.cr

3rd Elena Gabriela Barrantes Sliesarieva
*Escuela de Ciencias de la Computación e Informática*
*Universidad de Costa Rica*
San José, Costa Rica
elena.barrantes@ucr.ac.cr



***Abstract*—The ease of deployment of containerized web applications and the ease of management offered by orchestrator platforms such as `Kubernetes` has driven the design of systems by microservices. Containers implementing microservices can be targets of security attacks and due to programming language homogenization said attacks are more likely to succeed. This document presents a moving target defense (MTD) that makes use of the basic load balancing and high availability features of `Kubernetes` to instantiate versions of microservices implemented in different programming languages; with the aim of mitigating the exploitation of vulnerabilities in these specific languages. An attack model is simulated and an experiment is conducted to explore the scope of the defense in terms of attacks mitigated and impact on service interruption. The results show that the proposed defense reduces the effectiveness of attacks on microservices with a minimum cost per failure (downtime, service interruption perceived by user) of around 0.235% on average.**



***Index Terms*—Moving Target Defense, MTD, `Kubernetes`, microservices, diversification.**


## I. Introducción

La tendencia de implementación de aplicaciones web usando contenedores ha estado en auge durante los últimos años. La flexibilidad y versatilidad que ofrecen las plataformas orquestadoras de contenedores, resultan atractivas para los desarrolladores ya que facilitan las actividades administrativas como creación y calendarización de contenedores que antes se debían hacer manualmente [1].

Un **contenedor** es, a grandes rasgos, una instancia de una imagen que consiste en tres capas: la capa base de sistema operativo, la capa de dependencias y la capa de aplicación, que contiene el código que implementa de la aplicación web. La arquitectura por **microservicios** consiste en la implementación de un sistema descompuesto en varias aplicaciones web [2], generalmente bajo el modelo de arquitectura de `API REST`, que funcionan independientemente unas de otras pero también consumiendo sus servicios entre ellas. El frente de acceso a los usuarios se implementa comúnmente mediante un componente llamado *ingress*, el cual se implementa mediante un servidor web contenerizado en modo proxy reverso. La figura 1 ilustra una típica arquitectura de un sistema que consiste en 3 microservicios.

`Kubernetes` es una de las plataformas de orquestación de contenedores más utilizadas en la actualidad [3].

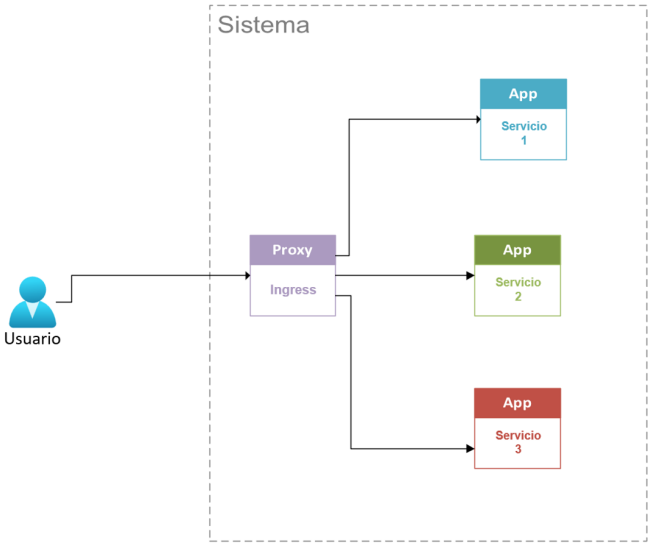


Figura 1. Arquitectura por microservicios.

`Kubernetes` estará en el corazón de nuestra defensa al hacer uso de sus funcionalidades básicas de distribución de carga y alta disponibilidad. Dos de los conceptos más fundamentales en `Kubernetes` son **pod** y **réplica**. Un pod es una unidad de agrupamiento lógico para los contenedores, los cuales representan la implementación de cada uno de los microservicios. Las réplicas son el número de instancias (pods) deseado y especificado durante el despliegue de cada microservicio (***deployment***). El *deployment* es una representación lógica de un sistema o aplicación compuesto por pods y réplicas. Cuando una réplica es eliminada, por cualquier razón, es responsabilidad de la plataforma orquestadora iniciar nuevas instancias hasta alcanzar el estado deseado, esta es la funcionalidad nativa del ***replica set*** asociado al *deployment*.

Si bien la arquitectura por microservicios ofrece múltiples beneficios como el aislamiento de la implementación y mantenimiento de cada componente por separado, también representa un desafío para la seguridad [4]. La tendencia en el desarrollo de microservicios es crear imágenes homogéneas,

debido a los beneficios en tiempo de desarrollo. Esto implica que los atacantes que descubran y exploten una vulnerabilidad en el ambiente de ejecución de una aplicación podrían vulnerar otros microservicios conectados a ésta. A esto se le denomina **ataque lateral por pasos** y se ilustra en el diagrama en la figura 2.

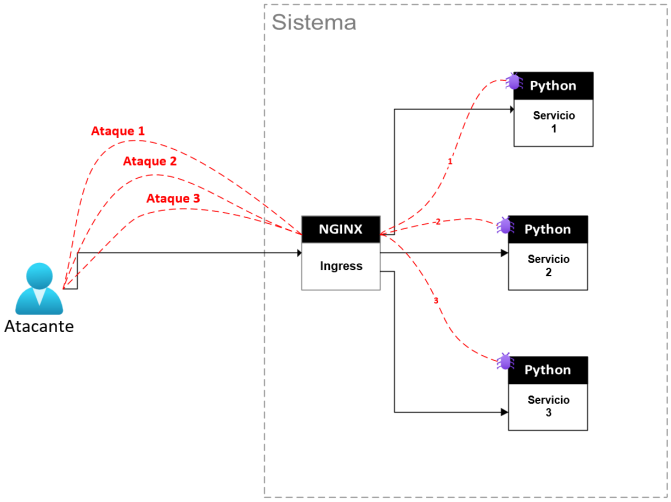


Figura 2. Ataque lateral por pasos a microservicios homogéneos.

Las defensas de blanco móvil (*Moving Target Defense*, MTD) basadas en diversificación se presentan como una opción interesante para solucionar el problema causado por la homogeneidad en microservicios [5]. Nuestro enfoque es en particular sobre microservicios homogéneos ejecutándose en `Kubernetes`. Nuestro objetivo fue determinar la efectividad de aplicar una defensa MTD para mitigar ataques laterales a microservicios en `Kubernetes` con respecto a su impacto en la interrupción del servicio.

La siguiente sección describe a detalle la defensa propuesta y su metodología de implementación. También se describen los experimentos ejecutados, así como sus factores de diseño, ruido y constantes. Exponemos los resultados obtenidos en la sección III, mencionamos trabajo relacionado en la sección IV y finalizamos con nuestras conclusiones y trabajo futuro en la sección V.

## II. Metodología

Nuestra defensa se basa en la idea de diversificación de microservicios propuesta por Torkura et al [6]. Diversificación en microservicios se refiere al cambio de lenguaje de programación que implementa la aplicación web. Esta idea ya había sido expuesta de manera abstracta y generalizada por parte de Tanguinod et al [7] en 2015.

Este tipo de defensa MTD corresponde al dominio de **diversificación de software** según Okhravi et al [8]. Se intenta mitigar las fases de *Development* (desarrollo y planificación del ataque) y *Launch* (ejecución del ataque) del *killchain*. En nuestra implementación, haciendo uso del servicio `Azure Kubernetes Service (AKS)`[1], estas dos fases de la cadena de ataque corresponden a las columnas de *Execution* y *Lateral movement* de la matriz de amenazas para `Kubernetes` desarrollada por Microsoft [9]. Estos aspectos fueron considerados para diseñar la lógica de simulación del ataque para la prueba de la defensa desarrollada.

Nuestra defensa cuenta con **2 niveles** de diversificación. El primer nivel es la creación de implementaciones paralelas en otro lenguaje de programación, almacenadas en la imagen del contenedor. El segundo nivel de diversificación se logra con la rotación de las implementaciones mediante un pod diversificador, el cual elimina réplicas aleatoriamente basado en una frecuencia determinada. La nueva réplica inicia una implementación alternativa de forma aleatoria. Con estos niveles aspiramos a reducir considerablemente la predictibilidad de que un atacante adivine el lenguaje en el que está o estará implementado un microservicio en cada ciclo de rotación, esto en concordancia con el concepto de *Timeliness* expuesto por Okhravi et al en [10].

Para probar la defensa implementamos un sistema de prueba de complejidad no trivial, denominado `Sistema TODO`. Este sistema está compuesto por 3 microservicios y un frente de acceso a usuarios provisto por `Kubernetes` conocido como *ingress*, el cual es un simple *proxy* reverso para ruteo de peticiones. La única funcionalidad de este sistema es insertar en una base de datos externa **tareas** para **proyectos** asignados a **usuarios**. La arquitectura del sistema de prueba se ilustra en el figura 3, al aplicar la defensa se aspira a obtener un momento dado un sistema totalmente diversificado como el que se muestra en la figura.

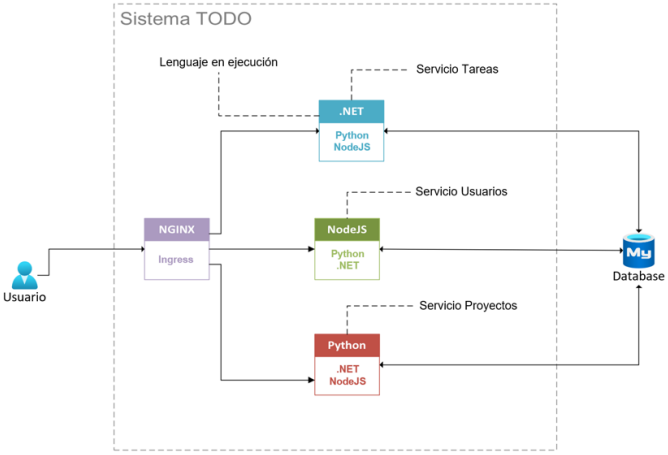


Figura 3. Diversificación de microservicios.

### A. Implementación

El diagrama que se muestra en la figura 4 resume la arquitectura del sistema implementado en `Azure`. Los siguientes recursos fueron necesarios:

- 1 máquina virtual para ejecutar el *script* de pruebas.

[1] https://azure.microsoft.com/en-us/services/kubernetes-service/

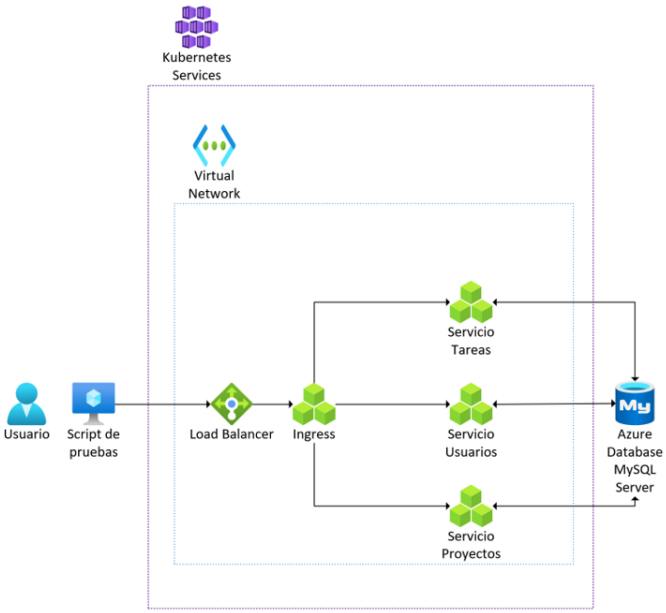


Figura 4. Implementación en Azure.

- 1 clúster de `AKS` para ejecutar los contenedores.
- 1 red virtual y un balanceador de carga para `AKS`.
- Una base de datos `MySQL` administrada por `Azure`.

Los contenedores que implementan los microservicios del sistema de prueba se crearon utilizando Docker[2]. Se escogieron `Python`, `NodeJS` y `.NET (C#)` para implementar las aplicaciones web. Se modificó el *script* de inicialización en el archivo Docker para iniciar de manera aleatoria una de las 3 implementaciones alternativas. El archivo Docker es utilizado durante la construcción de la imagen estática del contenedor.

La estrategia de rotación de las implementaciones se logra mediante el uso de **replica sets** en `Kubernetes`. Se implementó un pod diversificador para cada microservicio, el cual ejecuta un proceso calendarizado que selecciona y elimina réplicas aleatoriamente con una frecuencia parametrizada. Al eliminar una réplica la plataforma inicia otra de reemplazo, la cuál iniciará con una implementación aleatoria gracias a la configuración de la imagen del contenedor.

Fue necesario simular un ataque para probar la defensa. La detección de ataques reales es un tema complejo. Optamos por diseñar un *dataset* y un **pseudoataque** para simular vulnerabilidad en el sistema de prueba, ya que nuestro enfoque primordial fue la medición de la efectividad de la defensa y no la detección de ataques. El *dataset* consiste en líneas de texto para simular peticiones de inserción para el sistema de prueba. Algunas de estas líneas contienen caracteres cirílicos para simular **datos maliciosos**. Es importante señalar que debido al diseño del sistema de prueba se obtiene una **jerarquía de datos** por dependencia en el *dataset*: usuario → proyecto → tareas. Por ejemplo, un proyecto no puede ser insertado si el usuario asignado no se encuentra en la base de datos.

[2]https://www.docker.com/

Se eligió la implementación en `Python` para ser simulada como vulnerable, al ser la única en permitir la inserción de datos maliciosos en la base de datos. Cabe destacar que la inserción de datos maliciosos **no** produce efectos adversos en el sistema de prueba o interrupción del servicio.

Utilizando este sistema de prueba se realizaron dos experimentos, cuya metodología se describe en la siguiente sección.

### B. *Experimentos*

Como referencia para el diseño de los experimentos se eligió la guía de Peisert y Bishop [11].

*1) Variable de respuesta:* el éxito de los pseudotaques es verificado de manera binaria, por lo tanto la variable de respuesta tiene el valor de 1 si la defensa fue exitosa y no se lograron insertar datos maliciosos y 0 si no.

También fue de interés medir si el servicio se interrumpe por la implementación de la defensa. Los fallos se cuantifican midiendo la cantidad de peticiones de inserción de datos que reportan un error por medio de códigos de respuesta `HTTP`.

*2) Factor de diseño:* se varió la frecuencia del reemplazo de réplicas para probar cuál es la que brinda mejor protección y menor interrupción del servicio (conocido en el mundo de `Kubernetes` como *downtime*).

Por lo tanto, se definió el factor de diseño como:

`(fv1)`: Periodicidad del reemplazo de réplicas de `Kubernetes` en segundos.

Definimos 2 experimentos de acuerdo a los posibles valores de `fv1` para comparar sus resultados:

- Experimento 1: Infinito (sólo primer nivel de diversificación).
- Experimento 2: 15, 10, 5, 3 y 1 segundo (ambos niveles de diversificación).

Se tomó como referencia el tiempo de inicio de los contenedores el cuál fue de **3.63** segundos en promedio para cualquier lenguaje de programación. Los valores de 3 y 1 segundos fueron escogidos para evaluar los resultados de frecuencias menores al tiempo promedio de los contenedores. Por razones de diseño se decidió utilizar intervalos incrementales de 5 segundos. Sin embargo, se observó que luego de 15 segundos no se observaba una varianza significativa en los resultados, por lo que el valor máximo se estableció en ese valor.

*3) Factores constantes:* se mantuvieron en un nivel estático y específico [12]:

`(fc1)`: Cantidad de réplicas por microservicio (3).
`(fc2)`: Dataset constante (líneas de texto).
`(fc3)`: Velocidad de envío de los mensajes (0.1 segundos).
`(fc4)`: Mensajes enviados en paralelo (2 hilos).

*4) Factores de ruido:* se tomaron en cuenta:

`(fr1)`: Errores `HTTP` causados por interrupción de la red.
`(fr2)`: Errores `HTTP` a causa del balanceo de carga de `Kubernetes`.
`(fr3)`:Otros errores de comunicación en la red.

*5) Ejecución:* los experimentos se realizaron mediante ejecuciones de un *script* de prueba hecho en `Python` que realiza peticiones de inserción al sistema de prueba con datos del *dataset*. Al realizar una petición de inserción con un dato

malicioso ejecutamos el pseudoataque. La figura 5 ilustra el flujo de trabajo en cada ejecución. La lógica del *script* es crear un usuario, luego proyectos para ese usuario y finalmente las tareas para esos proyectos en el ciclo más anidado en el diagrama. Nótese en la parte superior los 3 microservicios del sistema de prueba y la dependencia por jerarquía de datos en las llamadas a funciones como `GetUserByName()`, necesaria para poder insertar un proyecto. En cada ejecución se trata de insertar:

- 5 usuarios, 1 de 5 es malicioso.
- 2 proyectos por usuario (10), 1 de 2 es malicioso.
- 100 tareas por proyecto (1000), 10 de 100 son maliciosas.

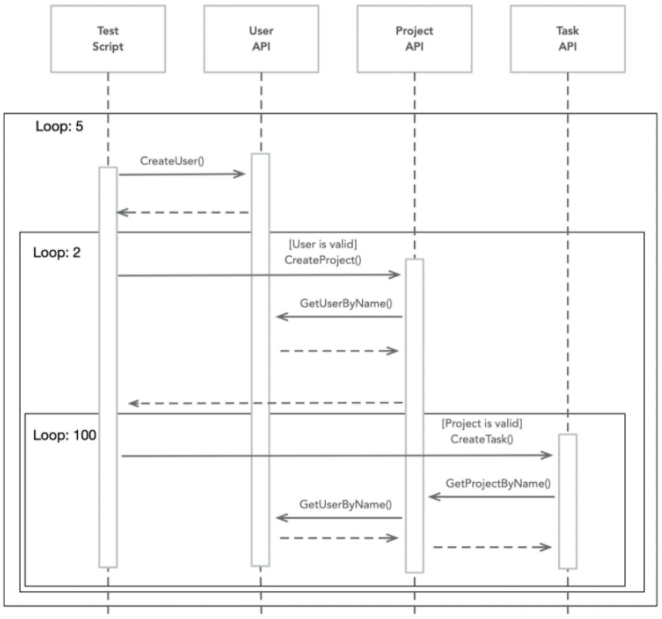


Figura 5. Diagrama de flujo para el *script* de prueba.

## III. Resultados

A continuación presentamos los resultados de la ejecución de los experimentos. El primer enfoque es por tipo de dato, presentando una serie de gráficos de pastel para cada una de las frecuencias de reemplazo de réplicas. La intención de análisis de estos resultados es determinar en qué medida nuestra defensa afecta la inserción y la jerarquía de datos. Los porcentajes en estos gráficos son:

- Porcentaje **del total del *dataset*** insertado (total de peticiones de inserción al sistema de prueba).
- Porcentaje **del total de datos insertados que contienen** datos maliciosos (pseudoataques no mitigados).
- Porcentaje **del total de datos insertados** que **no contienen** datos maliciosos.
- Porcentaje **del total del *dataset* rechazado** (por la defensa).
- Porcentaje **del total del *dataset* no insertado** (por fallos o dependencia por jerarquía de datos).

Comenzando con los resultados para usuarios en la figura 6, se puede observar que el porcentaje de datos maliciosos insertados baja conforme incrementa la frecuencia de reemplazo de réplicas. En contraste, sin reemplazo de réplicas se obtiene casi el doble de datos maliciosos. Además, debido a que los usuarios no sufren de dependencia de datos se pueden observar pequeños porcentajes de no inserción solo con frecuencias de reemplazo altas de 3 y 1 segundos.

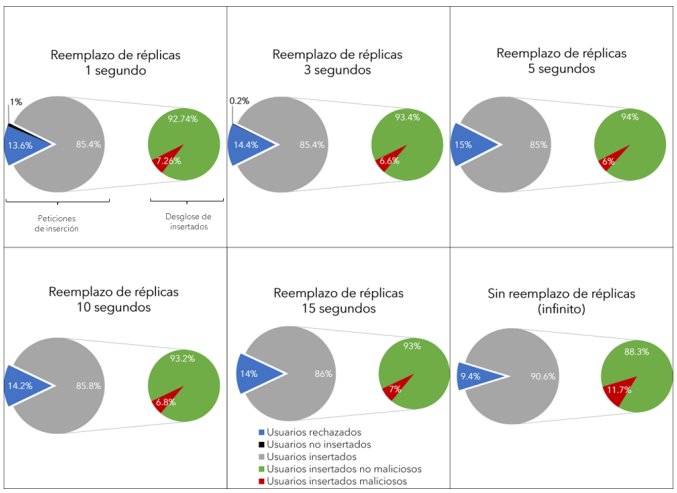


Figura 6. Comparativa de porcentajes de inserción y no inserción de usuarios.

Al observar la figura 7, se puede notar que para proyectos el porcentaje de datos no insertados crece debido a la jerarquía de datos. También se nota que en las ejecuciones sin reemplazo de réplicas se producen más datos maliciosos insertados. Esto quiere decir que utilizar 2 niveles de diversificación en la defensa afecta positivamente.

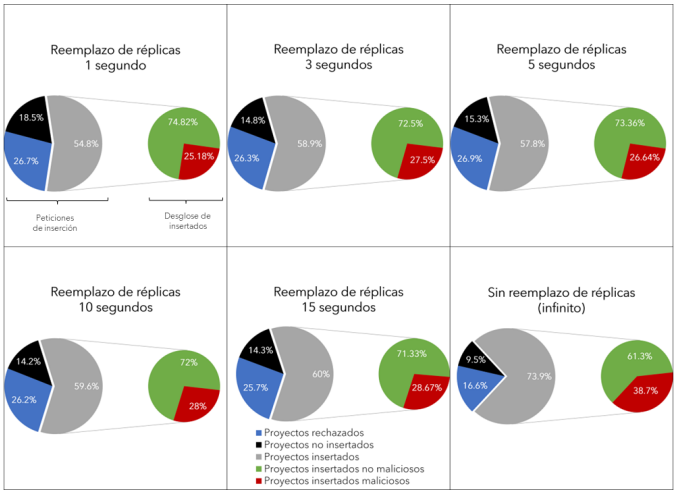


Figura 7. Comparativa de porcentajes de inserción y no inserción de proyectos.

Con respecto a los resultados para tareas en la figura 8, observamos que el porcentaje de datos no insertados crece considerablemente por el nivel más profundo en la jerarquía de datos. Es decir, este es el dato más afectado negativamente

por el segundo nivel de diversificación de la defensa. Sin embargo, en lo positivo, podemos observar que entre más datos se intenten insertar menos datos maliciosos insertados son obtenidos; esto significa una mayor efectividad de la defensa.

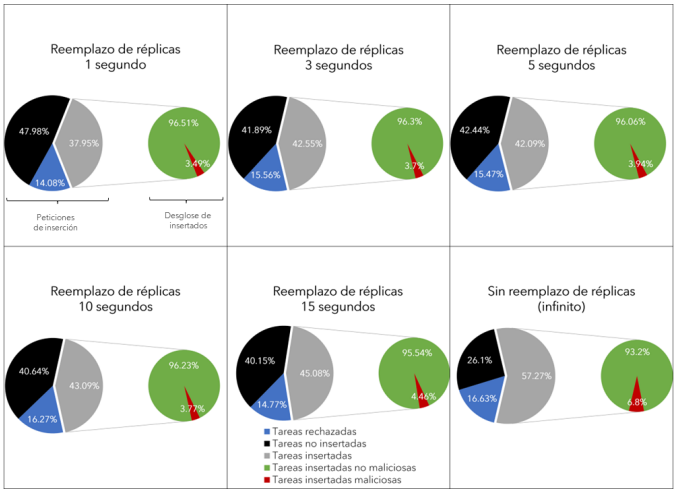


Figura 8. Comparativa de porcentajes de inserción y no inserción de tareas.

El siguiente enfoque para los resultados son los fallos globales. Estos resultados nos dan una idea de cuánto afecta nuestra defensa la funcionalidad del sistema de prueba. En el gráfico en la figura 9 se observa una linea base con el promedio total de fallos para todas las frecuencias de reemplazo de réplicas, el cual fue de **0.698%**. Como era de esperarse, la tendencia de los fallos es incremental a medida que se reemplazan réplicas más frecuentemente. Sin embargo, no siempre así es el caso. Por ejemplo, la frecuencia de 5 segundos obtuvo un 0.001% más que la frecuencia de 10 segundos. Otro dato interesante es que las frecuencias de 3 y 1 segundo no generan un 100% de fallos a pesar de ser menores al tiempo promedio de inicio de los contenedores de 3.63 segundos. Esto se debe al factor constante de réplicas por microservicio (3) y al diseño del pod diversificador, el cual selecciona réplicas aleatoriamente para su reemplazo. En `Kubernetes` cuando un pod es eliminado entra en un período de gracia de 30 segundos en el cual no se le envían más peticiones, pero sigue existiendo. Esta réplica podría ser seleccionada para ser eliminada nuevamente, si esto sucede la plataforma simplemente ignora esta petición.

Finalmente procedemos a medir la efectividad de nuestra defensa. Un posible enfoque podría ser medir el total de las tareas insertadas con respecto al total de tareas que serían insertadas en una **ejecución óptima**. En una ejecución óptima se asume que todos los datos maliciosos son rechazados, dando como resultado **360 tareas** (90 tareas * 1 proyecto * 4 usuarios). El gráfico en la figura 10 compara el total de tareas insertadas para cada frecuencia de reemplazo de réplicas con respecto a las 360 tareas insertadas en una ejecución óptima. Se puede observar que bajo este criterio la frecuencia más efectiva es 1 segundo. Sin embargo, como se vio en los resultados para fallos, esta frecuencia es la que más genera interrupción del servicio. Entonces, para cumplir con nuestro

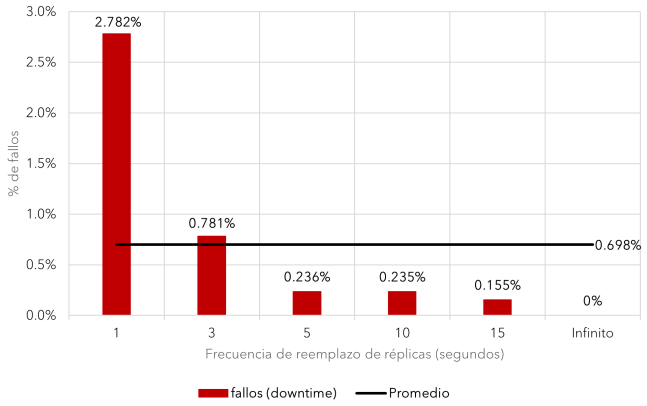


Figura 9. Comparativa de porcentajes de fallos (*downtime*).

objetivo, un enfoque más completo para la medición de la efectividad de la defensa es comparar las tareas insertadas (y su desglose en maliciosas y no maliciosas) contra la cantidad de fallos generados por cada una de las frecuencias de reemplazo de réplicas.

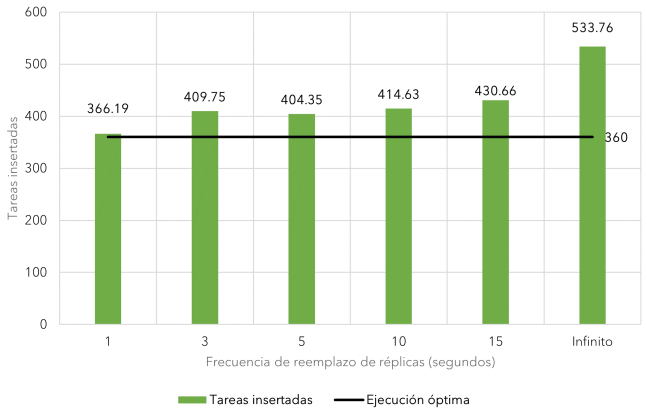


Figura 10. Comparación del total de tareas insertadas con respecto a la ejecución óptima (360 tareas insertadas).

El gráfico en la figura 11 nos permite determinar que la frecuencia para el reemplazo de réplicas más efectiva es **10 segundos**, causando solo un **0.235%** de **fallos** y generando solo **3.77%** de **tareas maliciosas** insertadas. Otra muy buena candidata es una frecuencia de 5 segundos, sin embargo, causa un 0.001% más de fallos y genera 0.24% más de tareas maliciosas insertadas.

## IV. Trabajo Relacionado

La idea de diversificar microservicios en la cual tomamos inspiración fue identificada en el trabajo de Torkura et al [6] del año 2018. La defensa diversifica la capa de OS y el lenguaje de programación de forma dinámica, utilizando la libería Swagger Codegen[3] para la migración de código en formato OpenAPI [4] y un identificador de vulnerabilidades de

[3] https://swagger.io/tools/swagger-codegen/
[4] https://swagger.io/specification/

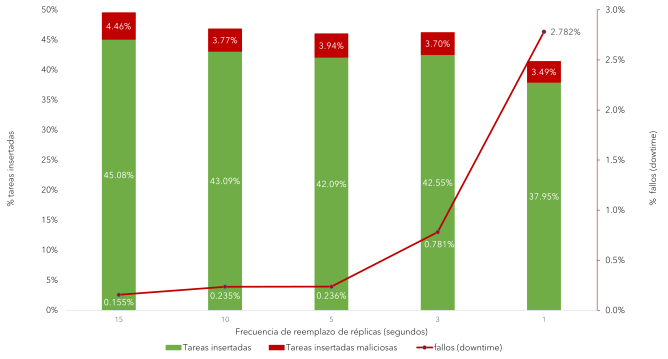


Figura 11. Comparación del total de tareas insertadas con respecto a los porcentajes de fallo.

un trabajo previo. Nuestra defensa se diferencia principalmente en que nosotros proponemos una construcción manual y estática de las implementaciones alternativas del código del microservicio. Si bien esto conlleva cierta penalización en la eficiencia del uso del almacenamiento a nivel de imagen encontramos que esto en `Kubernetes` tiene un impacto menos significativo comparado al incremento del tiempo de inicio de un microservicio debido a su diversificación de forma dinámica. El uso de generadores de código y formatos de programación garantiza en gran medida la equivalencia funcional. Sin embargo, la estrategia de implementación manual permite más flexibilidad para la toma de decisiones para lograr una total equivalencia en otros aspectos, como por ejemplo el desempeño.

## V. Conclusiones y Trabajo Futuro

La defensa se implementó utilizando herramientas disponibles en la nube y probadas en la industria. Se puede evidenciar una reducción tangible del porcentaje total de datos maliciosos insertados al aplicar ambos niveles de diversificación en la defensa, en comparación con los resultados obtenidos al utilizar el mismo número de réplicas por microservicio y recaer totalmente en la entropía del balanceo de carga de `Kubernetes`, es decir, un solo nivel de diversificación. El uso de 3 implementaciones por cada microservicio, 3 réplicas por contenedor y el reemplazo de réplicas cada **10 segundos** genera la suficiente entropía para obtener resultados satisfactorios en la mitigación de los ataques con respecto a fallos generados.

El porcentaje de fallos se puede catalogar como una penalización baja y manejable en comparación con los beneficios ofrecidos por la diversificación, siempre y cuando los microservicios sean relativamente simples y no sensitivos a la persistencia de sesiones. Sistemas compuestos de microservicios que ejecutan transacciones individuales de inserción y lectura de datos (como el sistema de prueba) son buenos candidatos. Otro aspecto importante a considerar es la escalabilidad, en el sentido del esfuerzo que conlleva crear implementaciones alternativas para muchos microservicios. Sin embargo, se podría también tomar la decisión de diversificar solo ciertos microservicios sensibles.

El pequeño porcentaje de **0.235%** de fallos podría ser fácilmente mitigable mediante la implementación de una lógica de reintento en el código. Además, si bien la frecuencia de reemplazo de réplicas se manejó con el mismo valor para todos los microservicios, el diseño de la defensa permite personalizar este valor en cada pod diversificador. De esta manera se podría configurar el valor deseado según tiempos distintos de inicio de contenedores en cada uno de los microservicios para así evitar más fallos.

Nos planteamos algunos puntos de mejora. Sería deseable realizar este mismo ejercicio simulando un ataque real, sobre aplicaciones reales que implementan microservicios de un sistema real. Así mismo, se plantean algunos cambios al experimento que podrían generar nuevas conclusiones. Por ejemplo, la inclusión de una nueva variable de diseño para el número de réplicas por microservicio, eliminándola de los factores constantes. Lo mismo podría implementarse para el valor constante que representa la velocidad de envío de las peticiones por parte del *script* de pruebas, asemejándose aún más a un escenario real.

## Referencias


[1] E. Casalicchio, *Container Orchestration: A Survey*. Cham: Springer International Publishing, 2019, pp. 221–235. [Online]. Available: https://doi.org/10.1007/978-3-319-92378-9_14

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*. Cham: Springer International Publishing, 2017, pp. 195–216. [Online]. Available: https://doi.org/10.1007/978-3-319-67425-4_12

[3] K. Hightower, B. Burns, and J. Beda, *Kubernetes: Up and Running Dive into the Future of Infrastructure*, 1st ed. O'Reilly Media, Inc., 2017.

[4] T. Y. B and C. Otterstad, *A Game of Microservices: Automated Intrusion Response*. Springer International Publishing, January 2018. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-93767-0_12

[5] H. Alavizadeh, J. B. Hong, J. Jang-Jaccard, and D. S. Kim, "Comprehensive security assessment of combined MTD techniques for the cloud," *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 11–20, 2018.

[6] K. A. Torkura, M. I. Sukmana, A. V. Kayem, F. Cheng, and C. Meinel, "A cyber risk based moving target defense mechanism for microservice architectures," in *2018 IEEE Intl Conf on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, 2018, pp. 932–939.

[7] M. Taguinod, A. Doupé, Z. Zhao, and G.-J. Ahn, "Toward a moving target defense for web applications," in *2015 IEEE International Conference on Information Reuse and Integration*, 2015, pp. 510–517.

[8] H. Okhravi, T. Hobson, D. Bigelow, and W. Streilein, "Finding focus in the blur of moving-target techniques," *IEEE Security Privacy*, vol. 12, no. 2, pp. 16–26, 2014.

[9] "Threat matrix for kubernetes," May 2021, https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/. [Online]. Available: https://www.microsoft.com/security/blog/2020/04/02/attack-matrix-kubernetes/

[10] T. Hobson, H. Okhravi, D. Bigelow, R. Rudd, and W. Streilein, "On the challenges of effective movement," *Proceedings of the ACM Conference on Computer and Communications Security*, vol. 2014-November, no. November, pp. 41–50, 2014.

[11] S. Peisert and M. Bishop, "How to design computer security experiments," *IFIP International Federation for Information Processing*, vol. 237, pp. 141–148, 2007.

[12] D. C. Montgomery, *Design and Analysis of Experiments*. Hoboken, NJ, USA: John Wiley & Sons, Inc., 2006.