Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

# Trabajo de Fin de Grado

Grado en Ingeniería Informática

---

# Operador para kubernetes para la rotación de secretos basado en regex

*Kubernetes operator for secret rotation based on regex*

Javier Herrera Serpa

---

La Laguna, *12* de *Julio* de 2019

D. **José Luis González Ávila**, con N.I.F. 78.677390-W profesor Asociado de Universidad adscrito al Departamento de Informática de la Universidad de La Laguna, como tutor

**C E R T I F I C A (N)**

Que la presente memoria titulada:

*“Operador de kubernetes para la rotación de secretos basado en regex”*

ha sido realizada bajo su dirección por D. **Javier Herrera Serpa**,

con N.I.F. 44.726.578-L.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 12 de Julio de 2024

# Agradecimientos

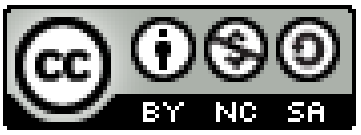
A Rebe, por todo el apoyo y ánimos que me ha dado, especialmente en momentos de estrés.

A mis padres, por la paciencia y confianza que han tenido en mi.

A Ray, por esas tardes delante de una pizarra discutiendo sobre caminos mínimos.

A mi tutor, por las correcciones y guía que me ha ofrecido.

# Licencia





**Resumen**

*Kubernetes es un software de orquestación de contenedores, diseñado originalmente por google, ahora es de código abierto bajo la licencia Apache 2.0. Permite desplegar y escalar aplicaciones basadas en contenedores, así como exponerlas a internet. Está diseñado para ser extensible, por lo que se puede añadir funcionalidad de diferentes formas.*

*Los operadores, son una de las formas que permite ampliar la funcionalidad de kubernetes. Consisten en una aplicación que, usualmente, se ejecuta dentro del propio clúster con capacidad para comunicarse con la api de Kubernetes y realizar los cambios necesarios en base al contenido de unos C.R.D. (custom resources definition). Estos C.R.D. son definidos por el operador y permiten al usuario interactuar con él.*

*Por otro lado, Kubernetes tiene soporte nativo para los secretos que usarán las aplicaciones. Sobre estos secretos se han construido muchas soluciones, ya que el secreto nativo tiene una funcionalidad muy básica. Solamente permite almacenar un secreto, pero no permite su modificación de forma dinámica ni su importación o exportación a otros sistemas.*

*Una de estas soluciones es el operador External Secrets, bajo licencia Apache 2.0. Este operador permite sincronizar los secretos de un clúster de Kubernetes con un almacén de claves (como OpenBao). Esto es una ventaja porque permite centralizar la gestión de credenciales en el almacén de claves, además soluciona el problema de la gestión de claves en entornos gitops.*

*El operador propuesto en este Trabajo Fin de Grado propone solucionar el problema de la rotación de secretos. Es una buena práctica cambiar las credenciales cada cierto tiempo. Actualmente ninguna de estas soluciones lo permiten de forma nativa, no sin modificar el código de aplicación. El operador sigue la filosofía unix, una herramienta muy simple que solo hace una cosa, pero la hace bien. El operador se encarga únicamente de rotar los secretos indicados, usando la regex indicada cada x tiempo, definido por la persona que despliega la aplicación.*

**Palabras clave:** Kubernetes, operador, secretos, rotar, regex

**Abstract**

*Kubernetes is a container orchestrator software, originally designed by google, now under the open source license Apache 2.0. It allows to deploy and scale container based applications as well as exposing it to the internet. It's designed to be extended, therefore new functionalities can be added from different approaches.*

*The operators are one of the available options to increase kubernetes functionality. An operator is an application which usually runs inside a container on the same clúster. The application usually has the ability to connect with the clúster api and, using this method, apply the required changes to achieve the desired state. All of this is based on the content of the operator C.R.D. (custom resource definition), defined by the operator and used by the user to interact with the operator.*

*Kubernetes has native support for secret storage, used by the applications. On top of that secrets many solutions has been built to improve the secrets functionality due to the simple implementation of the native secrets. The native secrets allows only to store, read and modify it, but doesn't allow to rotate, import or export to other systems.*

*One of these solutions is the external secrets operator, under Apache 2.0 license. This operator allows to sync the kubernetes clúster secrets with an external secret storage (like OpenBao). This is an advantage since it allows to have all secret management on a centralized platform. In addition it solves the problem of having the secrets hardcoded in the git repositories, especially in gitops environments.*

*The operator proposed in this end of degree project tries to solve the secret rotation problem. The rotation of the credentials after a defined period of time is considered a good practice, actually none of the current solutions allows to rotate the secrets natively and without application code changes. The operator follows unix philosophy, a simple tool that does only one thing at a time but does it well. The operator is designed to rotate secrets only, using the user defined regex and cron schedule.*

**Keywords:** kubernetes, operator, secrets, rotation, regex

# Índice general

# Índice de figuras

# Índice de tablas



# Capítulo 1 Introducción

## 1.1 Motivación

Actualmente existen múltiples formas de desplegar secretos en kubernetes, pero muy pocas permiten el uso de credenciales dinámicas. El uso de credenciales dinámicas además suele requerir de cambios en el propio código de la aplicación, lo que la mayoría de las veces no suele ser viable. Mediante el operador diseñado en este Trabajo Fin de Grado, se añade la capacidad para rotar secretos sin necesidad de modificar el código de las aplicaciones. Además permite aprovechar las características de otros operadores usados habitualmente para el despliegue de secretos, como external secrets para sincronizar dicha rotación de secretos con un almacén externo de claves.

El modelo de operadores de kubernetes permite ampliar la funcionalidad del clústers de forma sencilla, a la vez que proporciona una interfaz definida y simple para el usuario a

través de C.R.D. (custom resource definitions) por lo que se ha optado por seguir este paradigma a la hora de diseñar la solución para la rotación de secretos. Debido a que el operador modifica directamente el secreto nativo de kubernetes, no es necesario ninguna modificación en el código de las aplicaciones ni en el código de despliegue, mas allá de añadir una instancia del C.R.(custom resource). Esto también permite que mecanismos de otros operadores que también actúen sobre los secretos puedan ser compatibles con este operador.

Por último, gracias a lo sencillo que es implementar una rotación de secretos eficaz con el operador propuesto, se espera que tenga una gran adopción, mejorando el estado del manejo de credenciales a nivel general, evitando secretos estáticos, que no cambian y, normalmente, de baja entropía, compuestos por palabras fáciles de recordar, pero también de predecir por un posible atacante.

# Capítulo 2 Estado del arte

En el capítulo anterior se describe la motivación y qué problemas pretende solucionar este trabajo. En este capítulo se describe el estado del arte actual en lo referente a este trabajo, haciendo hincapié en las diferentes tecnologías aplicables en un entorno real empresarial donde se produce dichos problemas.

## 2.1 Kubernetes

En la actualidad Kubernetes tiene soporte para secretos, llamados secrets. Las aplicaciones esperan leer sus secretos a través de dicho objeto, pero Kubernetes no aporta ninguna otra funcionalidad extra, no permite la sincronización o la rotación de dichos secretos de forma automática. Hay opciones como Mozilla SOPS, que permite cifrar los secretos con criptografía asimétrica, de tal forma que solo el clúster puede leer los secretos, pero esto no permite la rotación de secretos. También hay operadores, como external secrets, que amplían la funcionalidad de los secretos de kubernetes permitiendo sincronizarlos con un almacén externo de claves. Pero la rotación de claves no está entre sus funciones, tampoco existe ningún operador con dicha funcionalidad en el momento de publicación de este Trabajo Fin de Grado.

## 2.2 Secrets

Son el objeto estándar que proporciona kubernetes para almacenar los secretos necesarios por las aplicaciones. Son similares a otro recurso llamado configmap, cada objeto es capaz de almacenar un número arbitrario de pares clave valor. La principal diferencia con respecto a los configmaps es que las claves están codificadas en base64. Además, al ser un objeto diferente, permite una mayor granularidad en cuanto al control de permisos, podemos permitir que un usuario acceda a leer o editar un configmap, pero prohibir que pueda leer o editar los secretos. Los pods desplegados que necesiten secretos para autenticarse contra servicios externos suelen esperar que exista un secret

de kubernetes desde el que leer las credenciales. Normalmente se indica en la definición del pod el nombre del secret a leer, así como las claves a leer. Los valores obtenidos de dichas claves pueden ser montados en el contenedor como variables de entorno o como fichero, dependiendo de como espere leerlo la aplicación.

## 2.3 Operadores

Los operadores permiten añadir funcionalidad a un clúster de kubernetes a través de recursos propios llamados Custom defined resources. Al instalar el operador, normalmente se instalan dichos C.R.D. los cuales permiten configurar el operador a través de manifiestos. Los operadores reciben notificaciones a través de la api de kubernetes cuando uno de sus C.R.D. es creado, editado o eliminado. En función del evento y del contenido del manifiesto, el operador realizará diferentes acciones para obtener el estado final deseado, a este proceso se le llama reconciliar. Ejemplos de operadores son External Secrets, cert-manager o external-dns.

### 2.4 External Secrets

Se trata de uno de los operadores más utilizados para desplegar secretos en kubernetes. Tiene soporte para la mayoría de almacenes de claves y además es open source (Apache 2.0). Este operador permite sincronizar secretos desde un almacén de claves hacia un secreto de kubernetes y viceversa. Está totalmente centrado en la sincronización de secretos, no está pensado para sincronizar secretos más allá de un pequeño sistema de plantillas pensado para añadir el secreto como parte de un conjunto mayor de datos. A cambio tiene una gran compatibilidad con bastante tecnologías de almacenes de claves, permitiendo reemplazarlos fácilmente en caso de ser necesario.

## 2.5 Almacenes de claves

Software diseñado para almacenar y gestionar secretos o credenciales. Normalmente proveen de varios métodos de autenticación, soporte para usuarios y roles. Esto permite definir políticas restrictivas en las que un usuario o aplicación solamente tiene acceso a los secretos que necesita y no a todo el almacén de claves. Algunos almacenes de claves soportan clave efímeras, por ejemplo, cuando una aplicación solicita acceso de lectura a una base de datos, el almacén de claves se conecta a la base de datos, crea un usuario y contraseña para dicha aplicación con los permisos estrictamente necesarios y, pasado un tiempo definido, elimina el usuario. Si la aplicación sigue necesitando el acceso, el proceso se repite. Esto permite que en caso de que alguna credencial acabe comprometida se reduzca la ventana temporal durante la cual esa credencial es válida. La desventaja de esto es que la aplicación necesita tener implementada la lógica necesaria para conectarse a ese almacén de claves concreto y solicitar sus credenciales cada vez que caduquen. Otra desventaja es que el almacén de claves se convierte en un punto de fallo, si por algún motivo deja de estar disponible las aplicaciones empiezan a fallar por no poder acceder a los secretos. Esto no sucede cuando se usan solamente para sincronizar los secretos, ya que el clúster siempre tiene una copia de los secretos.

## 2.6 Gitops

Filosofía por la que todas las operaciones se realizan a través de git, modificando los manifiestos que definen el estado de una aplicación. Esto permite tener la infraestructura, aplicaciones desplegadas y configuración de las mismas versionadas, junto con un mensaje que explica que ha cambiado y quien ha sido el autor de dichos cambios. Usualmente se suele aplicar junto con tecnologías como fluxcd o argocd, pero puede ser

implementado también con una herramienta de automatización como jenkins y webhooks del repositorio.

# Capítulo 3 Tecnologías utilizadas

En el capítulo anterior se detalló el estado del arte en lo referido a este trabajo. En este capítulo se detalla qué herramientas y tecnologías han sido utilizadas durante el desarrollo del operador a la vez que se profundiza en algunos de los conceptos introducidos en el capítulo anterior.

## 3.1 Docker

Es una tecnología de contenedores, con ciertas similitudes a LXC, pero con varias diferencias importantes, como la facilidad para mover una aplicación de una máquina a otra o la tecnología sobre la que están basados. Docker permite empaquetar una aplicación, así como todas sus dependencias en un único contenedor, el resultado es una imágen de docker. Esta imágen, puede ser tratada como un artefacto inmutable que puede ser versionado. Esto facilita muchísimo el despliegue de dicha aplicación ya que al tener todas las dependencias incluídas, asegura poder desplegar la aplicación en prácticamente cualquier entorno, incluso aunque las dependencias ya no existan, debido a que están incluídas en la propia imágen. Para su funcionamiento, Docker depende principalmente de dos características del kernel de linux, los namespaces y cgroups. Los namespaces permiten aislar las aplicaciones, unas de otras, principalmente, permite aislar los UID y GID, la información sobre el hostname y dominio de la máquina, los puntos de montaje, la red, el ipc y el pid. Por otro lado, los cgroups permiten limitar los recursos asignados a cada contenedor, limitando el uso de cpu, memoria, acceso a red y el acceso a disco. Esto permite establecer cuotas para cada uno de estos recursos por cada contenedor. Por último, otra de las ventajas que aporta es que gracias a que la imagen está versionada facilita las tareas de actualización, vuelta a versiones anteriores, así como despliegues progresivos, permitiendo detectar posibles problemas en un subconjunto pequeño de usuarios.

## 3.2 Kubernetes

Se trata de un orquestador de contenedores de código abierto, ampliamente adoptado por la industria. Actualmente usa containerd como runtime de contenedores, pero en sus inicios usaba Docker. En un clúster de Kubernetes, la unidad mínima de ejecución es el pod, un pod está compuesto por uno o más contenedores.

Un clúster de kubernetes suele estar compuesto por varios nodos worker donde se ejecutan los contenedores de las aplicaciones así como dos componentes necesarios de kubernetes, el kubelet y el kube-proxy. El kubelet es el encargado de asegurarse de que los contenedores se están ejecutando asociados a un pod y que su estado es correcto. Por otro lado, el kube-proxy es un proxy de red encargado de implementar el concepto de

servicio de kubernetes,creando reglas de red que permitan o denieguen el acceso. Además estos nodos worker necesitan de otros nodos, llamados master o control plane, en ellos se ejecutan diferentes componentes críticos de kubernetes. Uno de estos componentes es el apiserver, encargado de exponer una api para configurar y consultar el estado del clúster. El scheduler es el encargado de monitorizar si es necesario crear un pod, una vez que se da esa situación, decide en que nodo worker se ejecutará el pod. El controller manager es el encargado de monitorizar si un nodo se cae, monitorizar los jobs y crear el pod asociado, crear serviceAccounts por defecto y de rellenar los datos en los endPointSlices.

Toda la configuración y despliegue de aplicaciones se define utilizando manifiestos, normalmente en formato yaml. Esto permite definir como código todo el estado deseado del clúster. A su vez esto abre la puerta a filosofías como gitops, donde todo el estado del clúster está versionado en un repositorio. Kubernetes no solo gestiona el despliegue, también resuelve problemas como el balanceo del tráfico entre múltiples instancias de la misma aplicación o políticas de red que funcionan a modo de firewall entre los diferentes pods. Además es tremendamente ampliable mediante operadores externos, prueba de ello es la gran cantidad de proyectos bajo el paraguas de la Cloud Native Compute Foundation (CNFC).

## 3.3 Kubebuilder

Es un framework diseñado para crear operadores de kubernetes, basado en el lenguaje Golang. Kubebuilder usa el patrón de operadores de kubernetes, los operadores definen sus propios C.R. para definir cómo deben actuar en el clúster. El framework proporciona una extensa documentación sobre el modelo de operadores y cómo se integra con Kubernetes, además de varias utilidades y librerías para facilitar el desarrollo de operadores. Por último, incluído en la documentación hay multitud de ejemplos que facilitan el desarrollo de operadores. Kubebuilder es parte de kubernetes-sigs (Special Interest Groups), bajo la licencia Apache 2.0.

## 3.4 External Secrets

External Secrets es un operador que permite sincronizar los secretos creados en un almacén de claves con kubernetes. Es compatible con los principales almacenes existentes en el momento. El funcionamiento de este operador consiste en dos recursos, en primer lugar configuramos el almacén de claves. Normalmente, external secrets debe autenticarse contra el almacén de claves para poder leer los secretos que creará en el clúster. Esto lo hace a través del C.R.D. SecretStore, en este objeto se define la forma de acceso al almacén de claves (normalmente una url), credenciales, tipo de almacén de claves, etc. Una vez definido el secretStore, cuando una aplicación es desplegada, suele incluir un objeto del tipo ExternalSecret. En este objeto se definen todos los secretos que deben ser leídos del almacén de claves, su ubicación dentro del almacén de claves, nombre del secreto de kubernetes que contendrá dichos secretos, una referencia al secretStore a utilizar, y varios campos más opcionales.

Al desplegar una aplicación que incluye un objeto del tipo ExternalSecret, el operador creará los secretos en el clúster y la aplicación podrá utilizarlos. Con esto se consigue que los secretos nunca sean parte de la aplicación, solo un parámetro más de configuración

que además no existe en ningún repositorio de código.

## 3.5 Flux

Flux es uno de los principales operadores que existen en la actualidad para la implementación del principio gitops en kubernetes, junto con su principal competidor, argocd. Se trata de software libre, bajo la licencia Apache 2.0 creado por la empresa weaveworks. Flux se encarga de, usando sus C.R.D., mantener el estado del clúster tal y como ha sido especificado en uno o varios repositorios. Esto permite realizar cambios en el clúster a base de commits en el repositorio, poder deshacer cambios fácilmente, volviendo a un estado anterior y tener versionada la configuración.

## 3.6 Kind

Es la abreviatura de Kubernetes in docker, esta herramienta, también bajo la licencia Apache 2.0, permite desplegar un clúster de kubernetes dentro de contenedores docker. Está ideado principalmente para pruebas, ya que permite tener un clúster funcionando en cuestión de segundos sin necesitar grandes cantidades de recursos hardware, así como destruirlo con la misma facilidad ya que se trata simplemente de contenedores docker. Es lo suficientemente versátil para permitir desplegar el clúster en diferentes configuraciones, con diferente número de nodos, pero tiene algunas ligeras diferencias con un clúster desplegado en condiciones normales, principalmente en el apartado de red.

# Capítulo 4 Título del capítulo cuarto

En el capítulo anterior se trataron todas las herramientas y tecnologías utilizadas durante el desarrollo del operador, en este capítulo se detalla el proceso de desarrollo seguido, así como los problemas encontrados durante el desarrollo y las soluciones aplicadas.

## 4.1 Desarrollo

Para el desarrollo se ha utilizado Neovim como herramienta de edición.

En primer lugar se ha desarrollado una aplicación de test, también en lenguaje Golang. Dicha aplicación espera leer un secreto desde un fichero o variable de entorno, luego levanta un servidor web que responde con el secreto leído cada vez que es consultada.

Posteriormente se ha empezado con el desarrollo del operador propiamente dicho. En una primera versión el operador solamente era capaz de monitorizar su custom resource y de leer secretos. Esta primera parte fue bastante lenta debido a la poca familiarización del

autor con el ecosistema de operadores y con la implementación de la api de kubernetes en Golang, por lo que fue necesario consultar la documentación con bastante frecuencia.

Una vez solventados los problemas encontrados en la fase anterior, se implementó una versión con capacidad para escribir secretos, aunque en este momento, el operador solo era capaz de escribir un valor definido en tiempo de compilación, ignorando el campo Regex del C.R.

En este punto se hizo evidente que el esquema elegido inicialmente para el C.R.D. era insuficiente. Los cambios realizados fueron añadir un campo booleano que permite habilitar o deshabilitar la rotación del secreto. Además, debido a que un secreto puede contener un número indeterminado de pares clave valor, se hace necesario poder especificar de alguna forma que pares deben ser rotados y cuáles no. Es frecuente que un secreto contenga un par clave valor para indicar un usuario y otro par clave valor para indicar la contraseña. Para solucionar este problema se introdujeron dos nuevos campos, IncludeKeys y ExcludeKeys. En caso de que ninguno de los dos campos sea indicado, el operador rotará todos los pares clave valor que existan en el secreto. Si el campo IncludeKeys está definido, el operador solamente rotará los valores de aquellas claves indicadas en IncludeKeys y que existan en el secreto, ignorando el resto de pares. Por otro lado, el campo ExcludeKeys, define todas las claves que no deben ser rotadas, el operador modificará todas las claves que existan en el secreto a excepción de las indicadas en el campo ExcludeKeys. Además, si el campo IncludeKeys y el campo ExcludeKeys están definidos el operador ignorará el campo ExcludeKeys y solo usará el campo IncludeKeys.

Con esta última modificación surgió un nuevo problema, la complejidad de comparar si una clave definida en ExcludeKeys o IncludeKeys existe en el secreto. Se podría iterar por todos las claves de ExcludeKeys y por todas las claves del secreto, pero esto tiene una complejidad muy elevada ($N^2$). La otra alternativa contemplada es almacenar uno de los sets de datos en un hashmap, de esta forma el tiempo de acceso es constante, pero construir el hashmap lleva una sobrecarga de tiempo que podría no justificar su uso.

Para resolver este problema se ha implementado código para generar de forma aleatoria sets de datos. Este código genera dos arrays con elementos de forma aleatoria y cuyo porcentaje de intersección (que porcentaje de elementos comunes hay entre ambos arrays) puede ser definido. En base a este código se ha generado un conjunto de arrays, desde 2 elementos hasta 8192 elementos, con un porcentaje de intersección variable desde el 0% hasta el 100% en pasos de un 10%. Posteriormente se han creado 3 algoritmos encargados de comprobar si un elemento de un array existe en otro. El primero es un bucle anidado, que recorre el primer array y para cada elemento del primer array, intenta localizarlo en el segundo, su complejidad es $N^2$.
El segundo algoritmo es muy similar al anterior, con la diferencia de que en cuanto encuentra el elemento detiene la ejecución del segundo array, esto hace que sea

ligeramente más rápido, especialmente cuando hay un alto índice de correlación, también con complejidad N² pero ligeramente más eficiente en ciertas ocasiones.

El último algoritmo implementado es el hashmap, usando la implementación nativa de golang, cuya complejidad es N log N. En el caso del hashmap, en primer lugar se busca el array más pequeño y es ese el que se transforma en hashmap.

Por último se han ejecutado todos los algoritmos 1000 veces sobre el mismo set de datos, compuesto por un tamaño definido de array y un porcentaje de intersección también definido. Los tres algoritmos han usado los mismos arrays, con el mismo contenido para cada combinación de tamaño y porcentaje de intersección. El tiempo medido incluye el tiempo necesario para reservar en memoria las variables necesarias para cada algoritmo.

```go
func intersectionNestedFor(keys1 []string, keys2 []string) (time.Duration, []string) {
    start := time.Now()
    var result []string
    for _, element1 := range keys1 {
        for _, element2 := range keys2 {
            if element1 == element2 {
                result = append(result, element1)
            }
        }
    }
    finish := time.Now()
    return finish.Sub(start), result
}
```

**Figura 4.1.1:** Algoritmo de bucle anidado

```go
func intersectionNestedForBreak(keys1 []string, keys2 []string) (time.Duration, []string) {
    start := time.Now()
    var result []string
    for _, element1 := range keys1 {
        for _, element2 := range keys2 {
            if element1 == element2 {
                result = append(result, element1)
                break
            }
        }
    }
    finish := time.Now()
    return finish.Sub(start), result
}
```

**Figura 4.1.2:** Algoritmo de bucle anidado con “break”

```go
func intersectionHashmap(keys1 []string, keys2[]string) (time.Duration, []string) {
    start := time.Now()
    var result []string
    keysMap := make(map[string]bool)
    if len(keys2) > len(keys1) {
        for _, element := range keys1 {
            keysMap[element] = true
        }
        for _, element := range keys2 {
            if keysMap[element] {
                result = append(result, element)
            }
        }
    } else {
        for _, element := range keys2 {
            keysMap[element] = true
        }
        for _, element := range keys1 {
            if keysMap[element] {
                result = append(result, element)
            }
        }
    }
    finish := time.Now()
    return finish.Sub(start), result
}
```

**Figura 4.1.3:** Algoritmo de tabla hash

Como vemos en la siguiente gráfica, para dos elementos no se justifica el tiempo que tarda en construirse el hashmap, tanto el algoritmo 1 como el 2 dan mejores resultados, especialmente el 2, como era de esperar, a medida que aumenta la correlación.

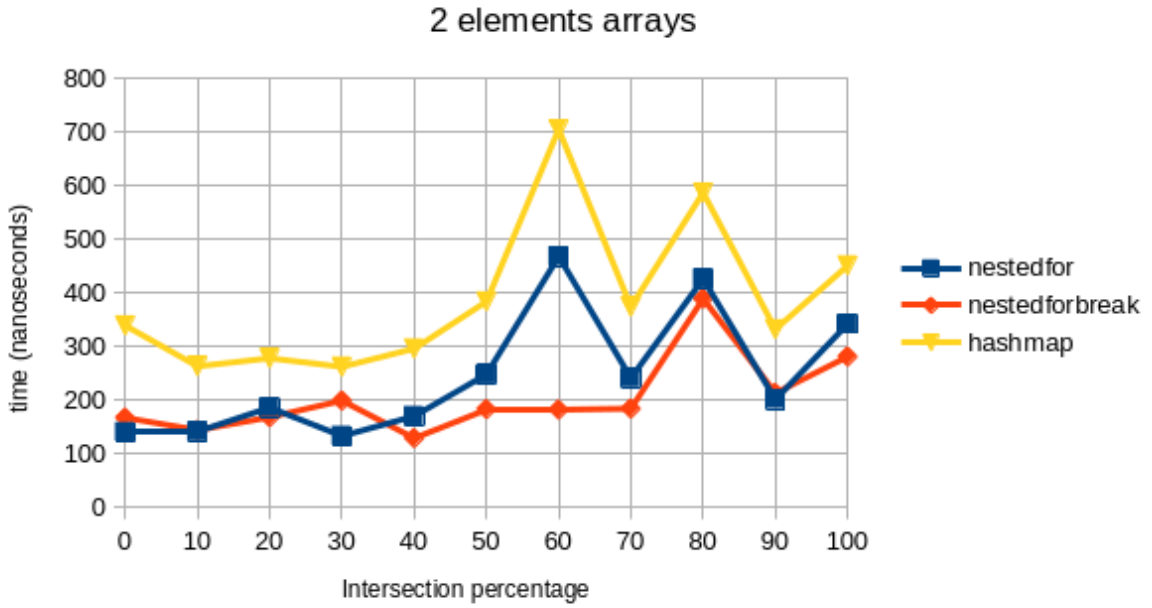



**Figura 4.1.4:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 2 elementos

Tanto para 4 elementos, como para 8 y 16 elementos tenemos una situación muy similar a la anterior, siendo en este último caso y para porcentajes de intersección muy bajos el hashmap ligeramente más eficiente.

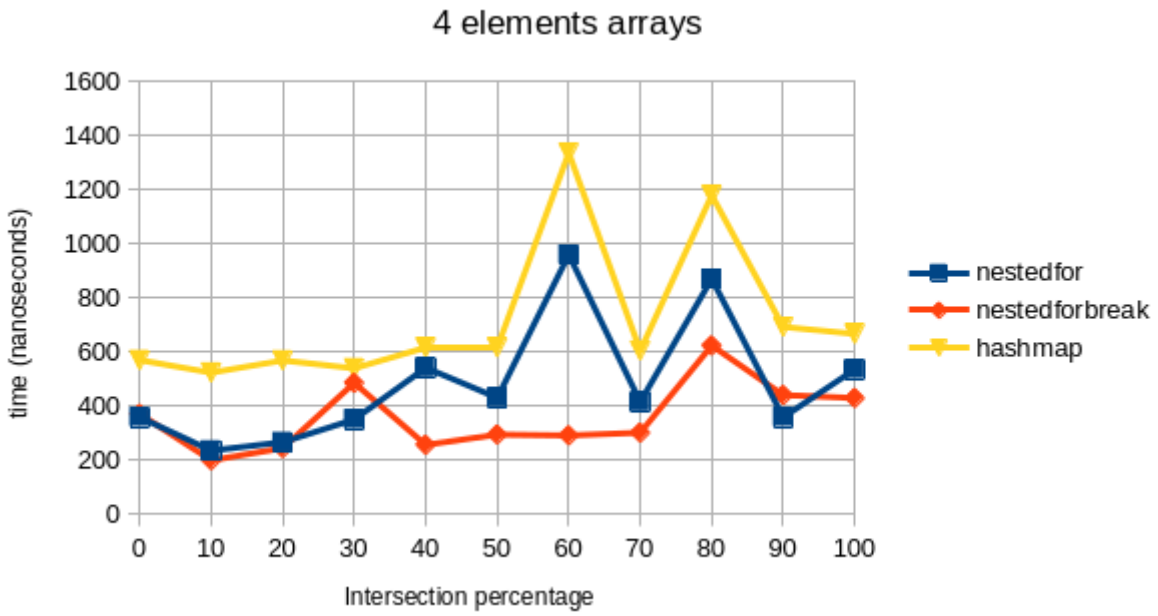



**Figura 4.1.5:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 4 elementos

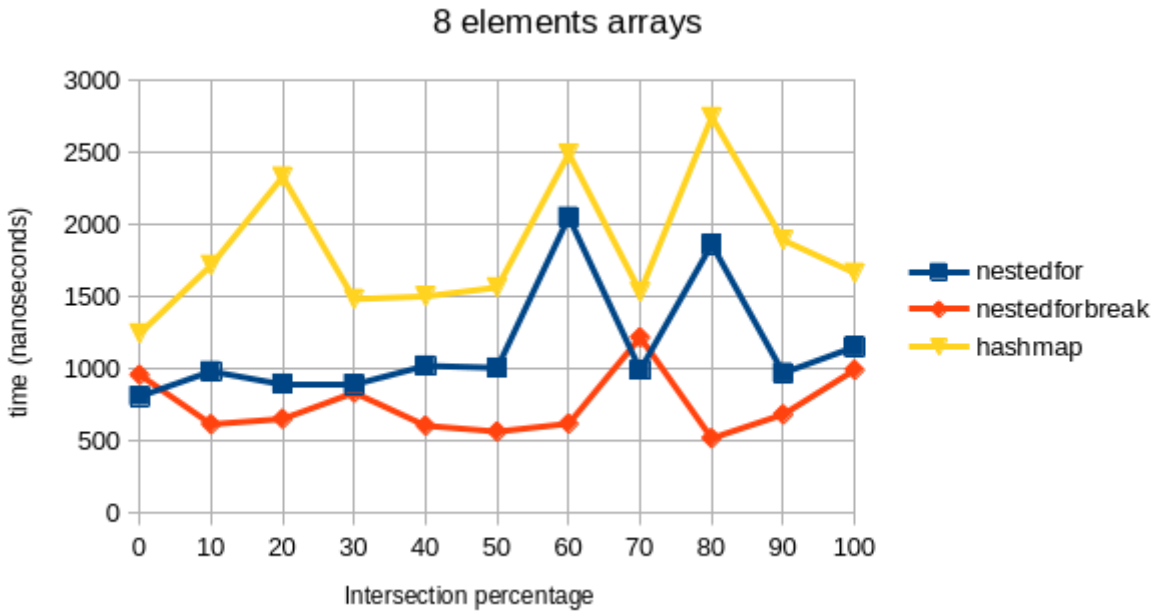



**Figura 4.1.6:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 8 elementos

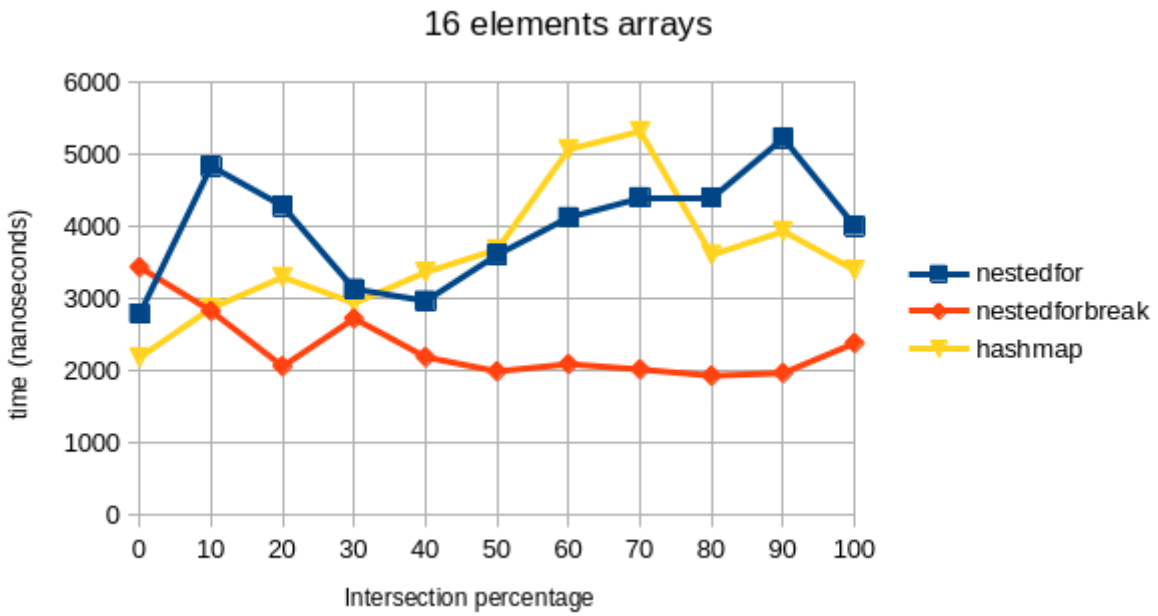



**Figura 4.1.7:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 16 elementos

A partir de los 32 elementos vemos como el hashmap empieza a tener un mejor rendimiento, especialmente cuando el porcentaje de intersección es bajo y siendo ligeramente más lento que el algoritmo 2 cuando el porcentaje de intersección es mayor.

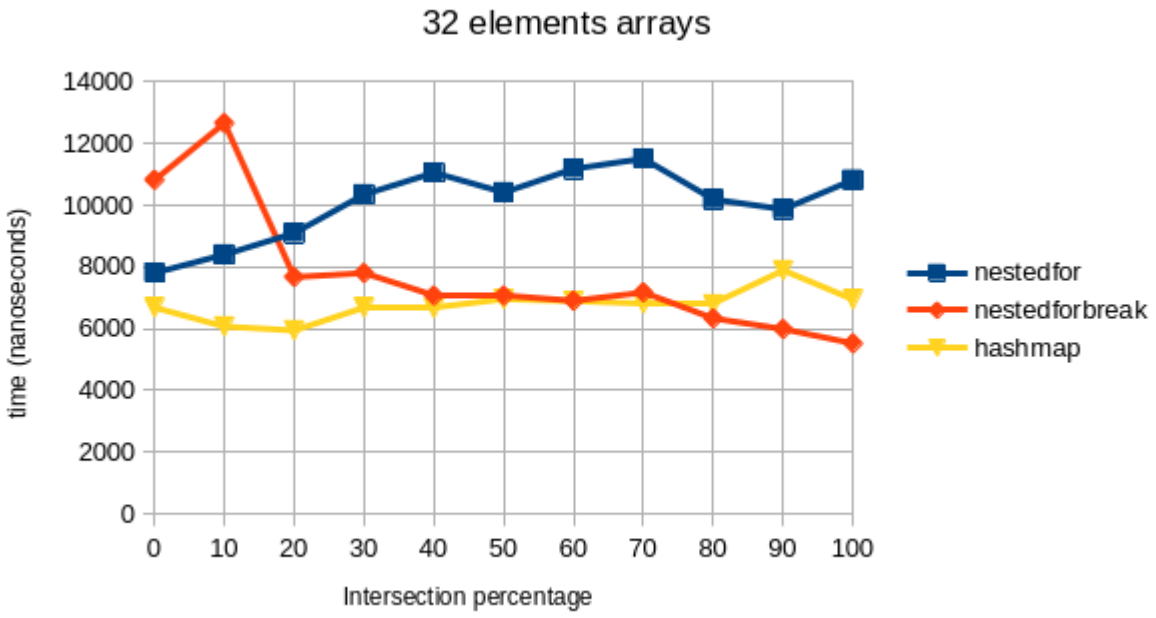



**Figura 4.1.8:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 32 elementos

A partir de este punto, en todos los casos el algoritmo 3 empieza a ser más rápido que los otros dos algoritmos, siendo cada vez más rentable el tiempo perdido en construirlo a medida que aumentan el número de elementos del array.

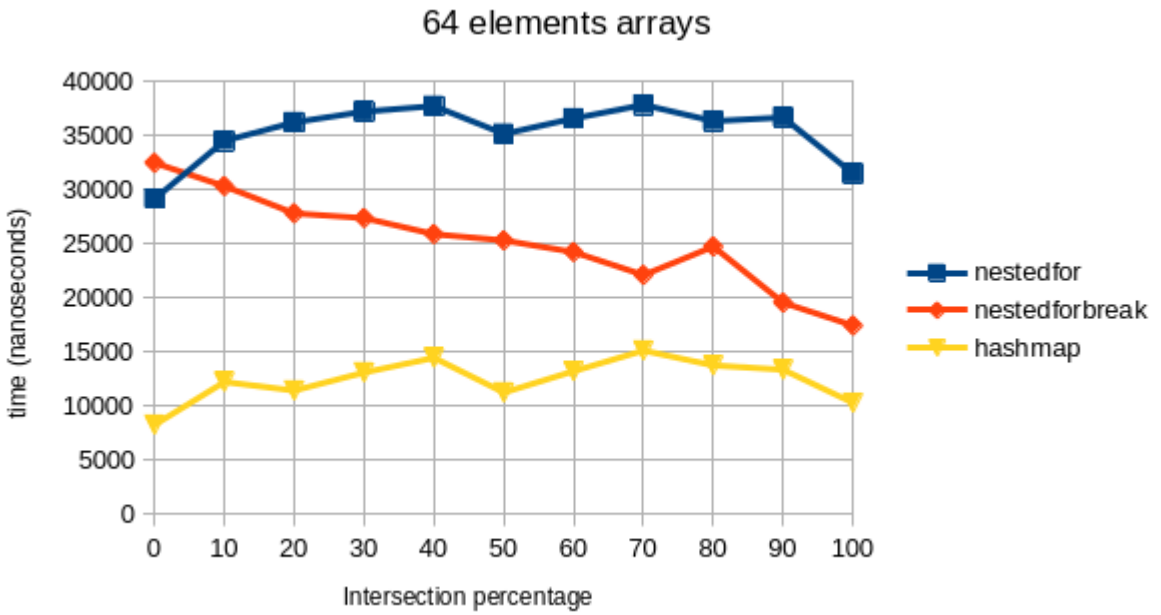



**Figura 4.1.9:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 64 elementos

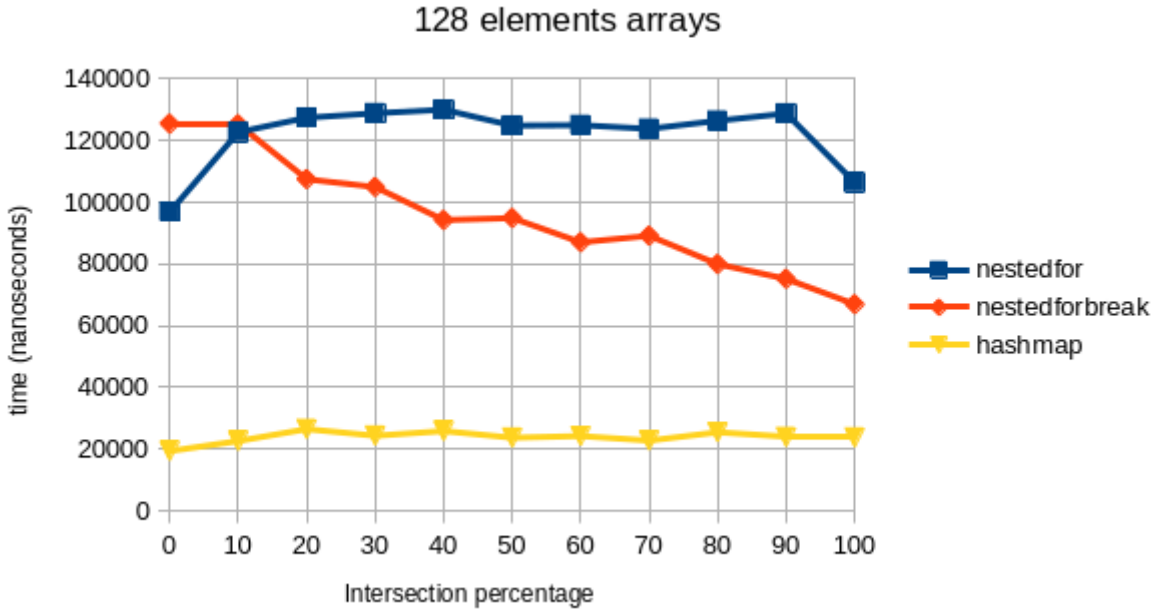



**Figura 4.1.10:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 128 elementos

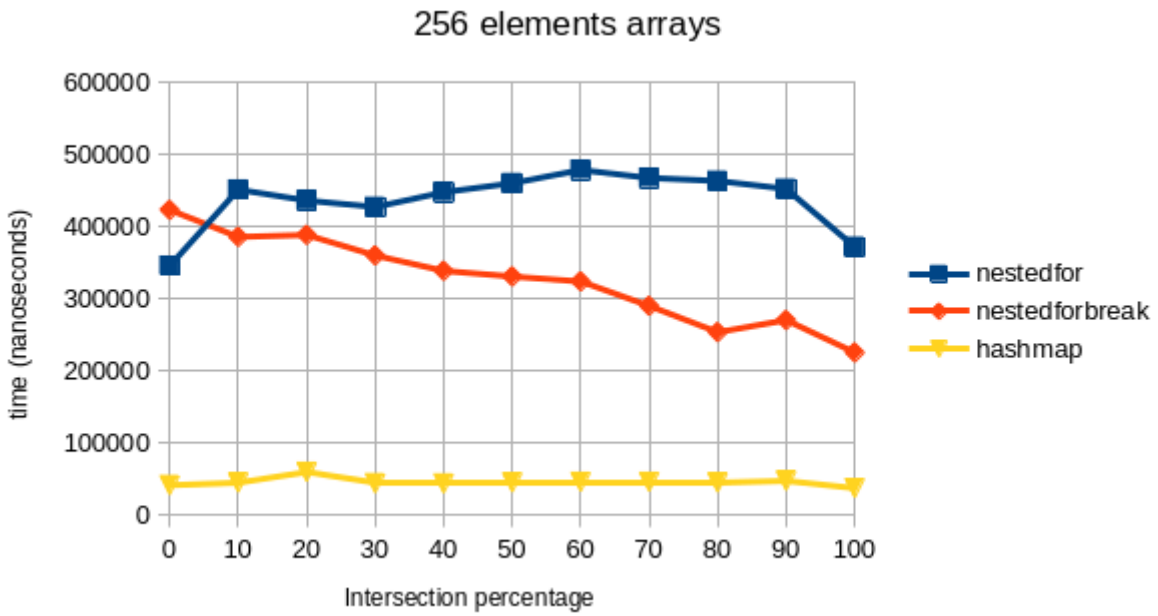



**Figura 4.1.11:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 256 elementos

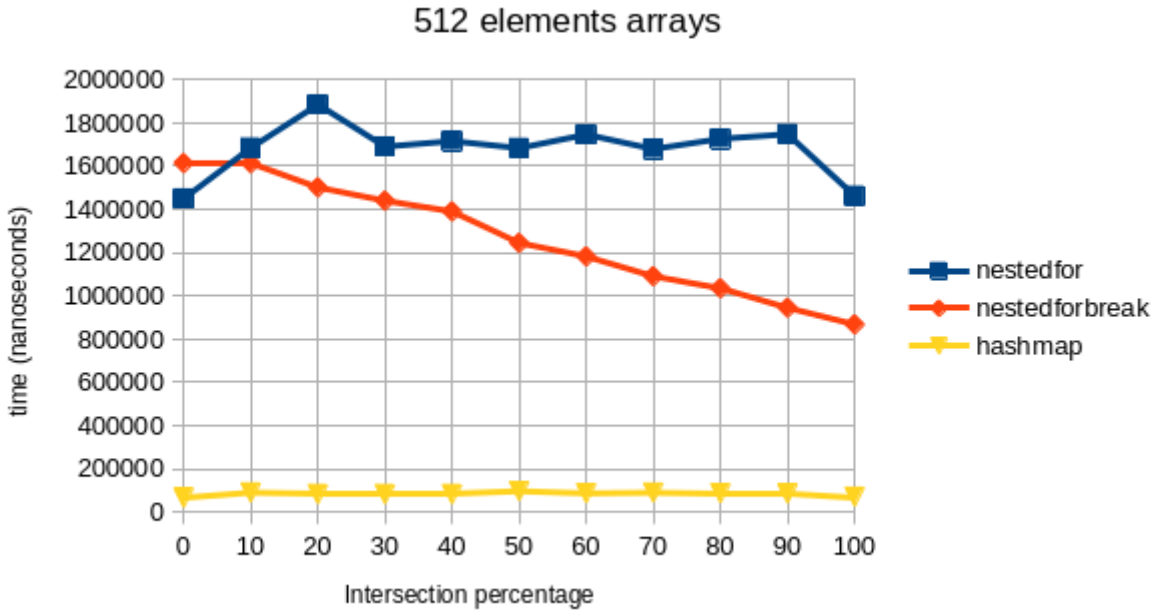



**Figura 4.1.12:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 512 elementos

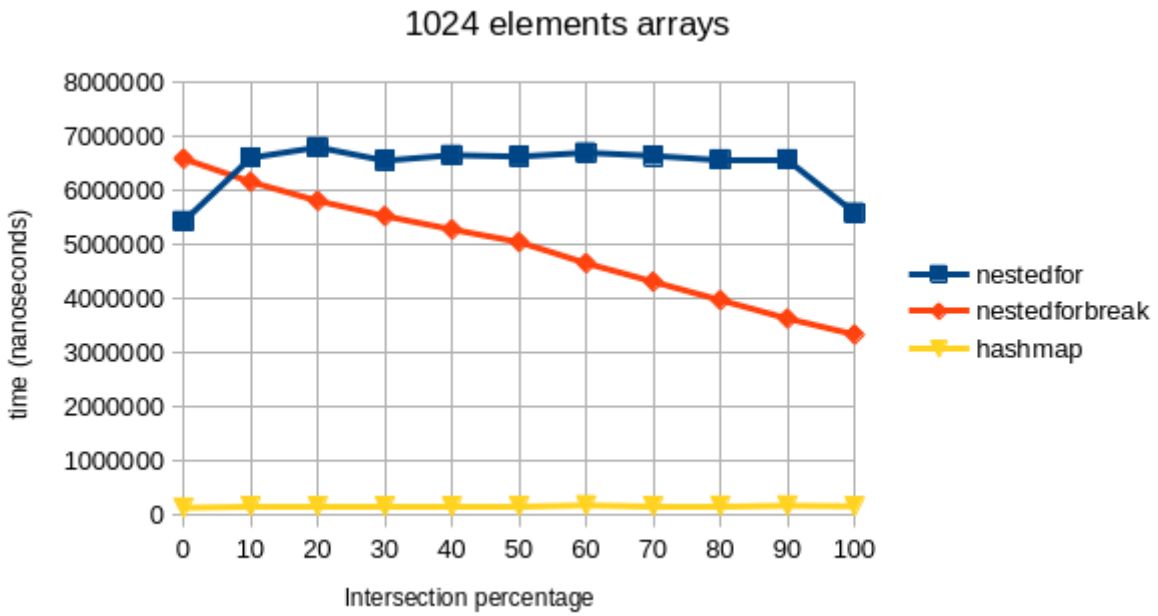



**Figura 4.1.13:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 1024 elementos

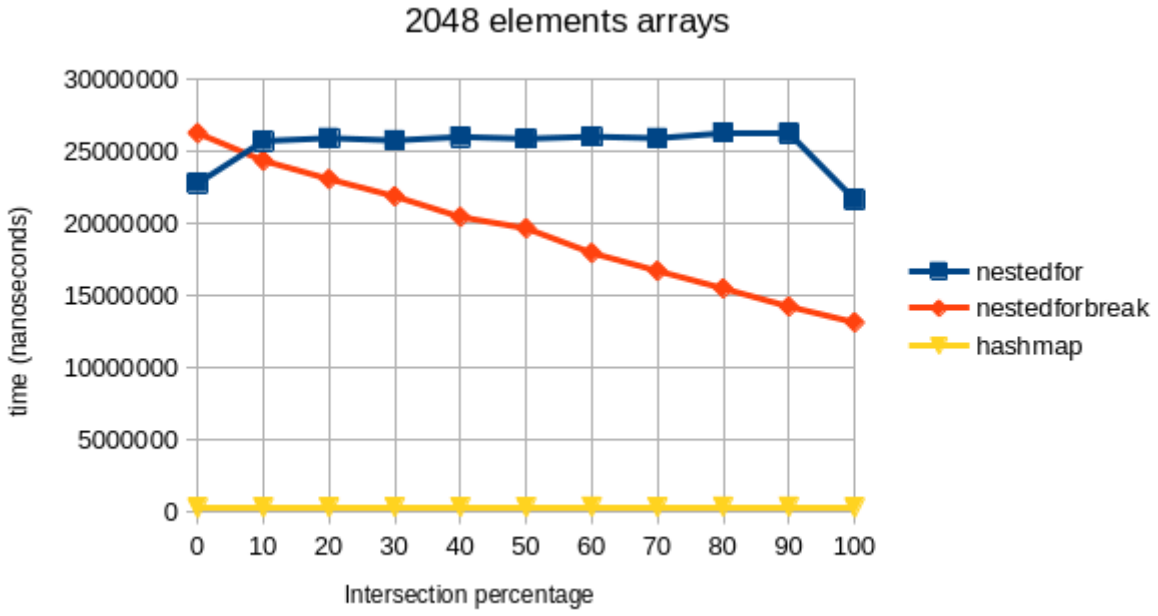



**Figura 4.1.14:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 2048 elementos

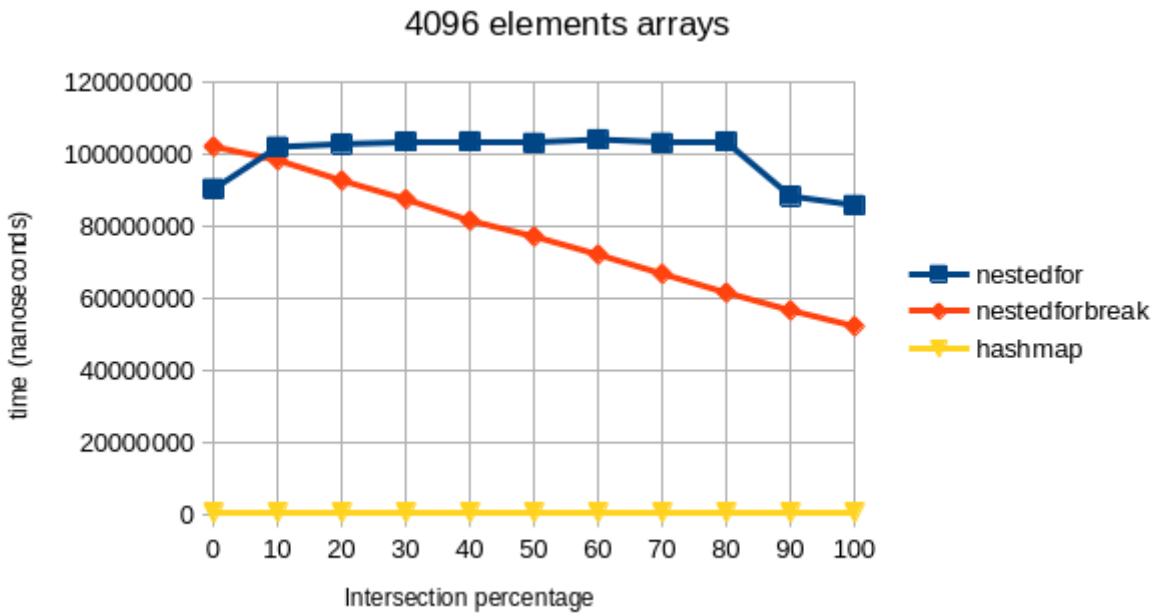



**Figura 4.1.15:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 4096 elementos

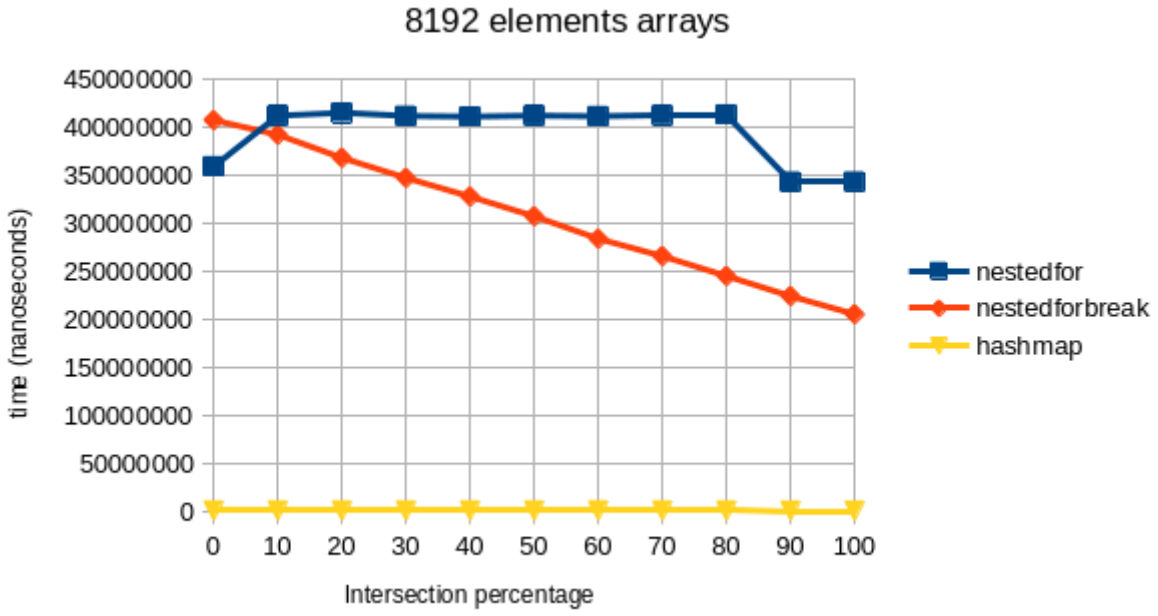



**Figura 4.1.16:** Gráfica del tiempo de ejecución de cada algoritmo para arrays de 8192 elementos

En base a estos datos, se ha optado por una estrategia híbrida, en la que si los arrays tienen menos de 32 elementos, se usará el algoritmo 2. En caso de que los arrays tengan 32 o más elementos, se usará un hashmap. Con esto se aprovechan las ventajas de los hashmaps para arrays grandes, mientras que no tienen la desventaja del tiempo necesario para su construcción con arrays pequeños.

## 4.2 Problemas encontrados

Durante el desarrollo del proyecto se encontraron varios problemas, a continuación se detallan cada uno de ellos y la solución utilizada.

Uno de los primeros problemas encontrados fue a la hora de ejecutar el comando make manifests, este comando debe generar una serie ficheros yaml en base al código golang proporcionado. El error no era demasiado descriptivo, así que la solución fue ir ejecutando go build, el comando encargado de compilar los ficheros en golang sobre cada fichero en busca de errores. Una vez encontrados y subsanados todos los errores make manifests funcionaba correctamente, esto es debido a que la compilación de esos ficheros era uno de los pasos de este comando.

Otra de las dificultades encontradas fue el entendimiento sobre como funcionan las librerías de golang para comunicarse con la api de kubernetes debido a la poca familiaridad del autor con estas. La solución fue leer la documentación oficial, la cual es bastante descriptiva, para ir solucionando las dudas al respecto.

El esquema definido originalmente para el C.R.D. no contemplaba todos los posibles casos de uso, casos como pausar momentáneamente la rotación o tener la capacidad para elegir qué claves del secreto se rotan y cuáles no, no estaban contemplados en el esquema original. La solución en este caso fue sencilla, se añadió un nuevo campo booleano que permite al usuario indicar que ese secreto no debe ser rotado. Para poder indicar que claves deben ser rotadas y cuales no, se añadieron los campos IncludeKeys y ExcludeKeys, que permiten indicar una lista de las claves que deben ser rotadas o, de forma alternativa, la lista de claves que no deben ser rotadas.

La solución a este problema introdujo un nuevo problema, que el usuario indique claves que deban ser rotadas que no existan realmente o claves que no deben ser rotadas que no existan. La solución a este problema en concreto está detallada en el apartado de desarrollo.

Los errores poco descriptivos, aparte de los encontrados, mencionados anteriormente a la hora de ejecutar el comando make manifests, también fueron un problema:

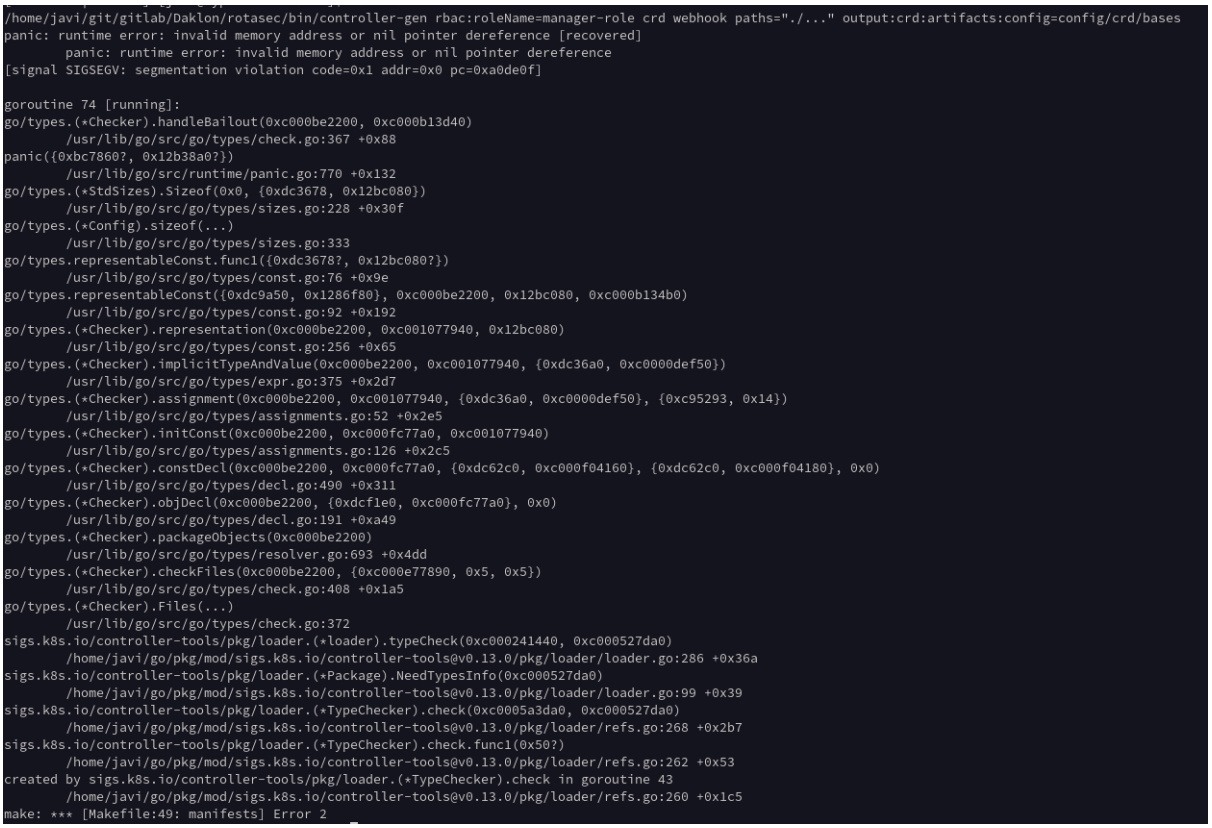



**Figura 4.2.1:** Captura de pantalla con una de las trazas de error

En este caso todos los ficheros compilaban correctamente, tras investigar a que podría deberse el error, se hizo evidente que se trataba de un bug (https://github.com/kubernetes-sigs/controller-tools/issues/888). La versión utilizada (0.13.0) estaba afectada por dicho bug. La solución fue actualizar a la última versión, la 0.14.0 ya que dicha versión corregía este bug y añadía otras mejoras. Tras la actualización no volvió a manifestarse dicho bug.

# Capítulo 5 Conclusiones y líneas futuras

## 5.1 Conclusiones

Este trabajo ha servido para diseñar, implementar y evaluar un operador para clústers de kubernetes encargado de la rotación de secretos nativos de kubernetes. Con su diseño se buscaba además que fuera compatible con otras soluciones, también implementadas sobre secretos de kubernetes.

En un primer momento se diseñó una api que debería permitir la interacción con el operador. A medida que se fue desarrollando el proyecto, se hizo evidente que la api definida inicialmente era insuficiente para permitir toda la funcionalidad deseada por lo que fue necesario ampliarla. Una vez resuelto este problema, apareció un problema nuevo, debido a la forma en la que está implementada la api de kubernetes, era necesario obtener la intersección de la lista de pares clave valor que proporcionaba el usuario a través del C.R.D. y la lista de pares clave valor que realmente existen en el secreto en cuestión.

Para resolver este problema, se implementó el código necesario para generar listas aleatorias, con porcentajes de intersección definidos y de longitud variable. Una vez generadas las listas, entre 2 elementos y 8192, con porcentajes que variaron entre el 0% y el 100%, se desarrollaron varios algoritmos para obtener la intersección. Los algoritmos probados fueron un bucle for anidado, un bucle for anidado con un “break” cuando encuentra una coincidencia y una tabla hash. Luego se probaron los algoritmos sobre todos los sets de datos, en múltiples iteraciones y promediando los resultados. Finalmente los datos fueron graficados para obtener una mejor representación de los resultados. Con estos gráficos se observó que para listas con menos de 32 elementos, el bucle for anidado con el “break” era la opción más rápida, pero a partir de ahí, debido a la complejidad del bucle for anidado, la tabla hash era más rápida, manteniéndose su tiempo de ejecución relativamente estable en comparación con el bucle for anidado.

En base a estas observaciones se decidió implementar un algoritmo dual, en el que si alguno de las listas es menor o igual a 32 elementos se calcula intersección utilizando el bucle anidado con “break”. En cambio si ambas listas son mayores de 32 elementos, se elige la menor de ellas, se transforma en una tabla hash y se recorre la otra buscando

intersecciones contra la tabla hash. De esta forma se consigue un buen rendimiento tanto en listas pequeñas como en listas grandes.

Este trabajo ha servido para resolver un problema real en el ámbito profesional a la vez que formaba al autor sobre el desarrollo de operadores para kubernetes y sobre cómo funciona la api de kubernetes.

## 5.2 Líneas futuras

Uno de los grandes determinantes en la adopción de una nueva tecnología es la documentación disponible. Durante el desarrollo de este tfg no ha sido posible abordarla debido a restricciones de tiempo, los problemas encontrados retrasaron el proyecto, por lo que hubo que reducir en áreas como la documentación. Una línea de trabajo futuro es precisamente generar dicha documentación, para facilitar la adopción del operador.

También se debería definir un código de conducta y una serie de normas para los que deseen colaborar con el proyecto, corrigiendo bugs o incorporando mejoras, así como el mantenimiento del propio operador a medida que van surgiendo nuevas versiones de kubernetes.

Otro de los posibles trabajos está relacionado con los propios CDR, es costumbre que las primeras versiones sean designadas v1alpha1, ya que en este punto se consideran inestables, con gran cantidad de posibles modificaciones. A medida que dichas versiones son probadas y se demuestra que no sufrirán grandes cambios pasan a v1beta1, en esta versión aún pueden sufrir cambios, pero no se esperan cambios mayores. Si finalmente se considera que no se introducirán más cambios en dicha versión, se considera que es un C.R.D. estable y se pasa a la versión v1. Todo este proceso requiere de cierto tiempo y de muchas pruebas, por lo que ha quedado fuera del ámbito de este trabajo.

Por otro lado, los proyectos relacionados con kubernetes, de cierto calado, suelen adherirse a la CNCF (Cloud Native Computing Foundation). Esto suele aumentar significativamente la visibilidad del proyecto, a su vez, esto provoca un aumento en las colaboraciones recibidas en los proyectos open source debido al aumento de popularidad. Probablemente el operador desarrollado en este trabajo, sea un buen candidato como proyecto adherido a la CNCF, pero es un proceso que requiere tiempo y que también ha quedado fuera del alcance de este trabajo.

Por último, se podría facilitar la instalación del operador con tecnologías como kustomize, helm o con un manifiesto de ejemplo aplicable con un curl. El primero permite, partiendo de unos manifiestos genéricos, personalizar solamente lo necesario. De esta forma, a la hora de instalar se podrían definir una serie de valores por defecto, modificando solamente algunos de ellos con particularidades del entorno y el despliegue en cuestión. Usando Helm también se facilita el despliegue del operador, pero helm trabaja de una forma diferente a kustomize, en lugar de parchear un manifiesto por defecto con los

cambios deseados, se define una serie de valores configurables y el usuario los modifica en un fichero, con valores para cada entorno y despliegue. El ejemplo aplicable con un curl, no suele permitir una gran modificación, a no ser que sea descargado y modificado, pero permite que el operador pueda ser instalado con una única línea ejecutada en una terminal con acceso al clúster. Esto último es muy útil para pruebas y como primera forma de despliegue a modo de prueba de concepto.

# Capítulo 6 Summary and Conclusions

## 6.1 Summary

There are currently a large number of integrations with kubernetes in various areas, including secret management. Many of these integrations have been developed to improve efficiency and security in managing sensitive information within kubernetes clusters. However, among all the available solutions, only one allows automatic rotation of secrets. This solution, although effective, has an essential requirement, modifying the application in a way that it can request credentials directly from the keystore. This can be a significant challenge for many organizations looking to integrate solutions without modifying their existing applications, either due to lack of time or because certifying their applications requires a lengthy process.

In this context, the operator designed and implemented in this project offers an innovative solution. This operator not only solves the problem of secret rotation, but also ensures compatibility with other operators that could also interact with kubernetes secrets. In this way, the integrity and functionality of the Kubernetes ecosystem is maintained without the need to make drastic changes to existing applications or infrastructure.

The development of this operator required extensive research and documentation in various technologies, all of them widely used in the professional field. These technologies include Kubernetes and the way operators interact with its API. Deep understanding of these interactions was crucial to effective operator design. In addition, it was necessary to carry out a series of tests and performance measurements of the different algorithms used to compare the current keys existent inte secret and the ones that are requested to be modified. These tests evaluated algorithms with different complexities, to ensure that the implemented solution offered the best possible performance in different scenarios.

This approach ensures that the solution is efficient, while solving a critical problem for an organization that wants to implement best practices in secret management while maintaining compatibility with other tools and without the need for major modifications of their applications.

## 6.2 Conclusions

This work has served to design, implement and evaluate an operator for kubernetes clusters in charge of rotating native kubernetes secrets. Its design also sought to be compatible with other solutions, also implemented on top of Kubernetes secrets.

Initially, an API was designed to allow the interaction with the operator. As the project developed, it became evident that the initially defined API was insufficient to allow all the desired functionality, so it was necessary to extend it. Once this problem was solved, a new problem appeared, due to the way in which the kubernetes API is implemented, it was necessary to obtain the intersection of the list of key-value pairs provided by the user through the C.R.D. and the list of key value pairs that actually exist in the secret in question.

To solve this problem, the necessary code to generate random lists was implemented, with defined intersection percentages and variable length. Once the lists were generated, between 2 elements and 8192, with percentages that varied between 0% and 100%, several algorithms were developed to obtain the intersection. The algorithms tested were a nested for loop, a nested for loop with a “break” when it finds a match, and a hash table. The algorithms were then tested on all data sets, in multiple iterations and averaging the results. Finally the data were graphed to obtain a better representation of the results. With these graphs it was observed that for lists with less than 32 elements, the nested for loop with the “break” was the fastest option, but from there, due to the complexity of the nested for loop, the hash table was faster , keeping its execution time relatively stable compared to the nested for loop.

Based on these observations, it was decided to implement a dual algorithm, in which if any of the lists is less than or equal to 32 elements, intersection is calculated using the nested loop with “break”. On the other hand, if both lists are greater than 32 elements, the smallest of them is chosen, transformed into a hash table and the other is traversed looking for intersections against the hash table. In this way, good performance is achieved in both small lists and large lists.

This work has served to solve a real problem in the professional field while training the author on the development of operators for kubernetes and how the kubernetes API works.

# Capítulo 7 Presupuesto

En la primera tabla se detalla el presupuesto de los costes relacionados con la producción y desarrollo del operador, incluyendo las horas invertidas y recursos utilizados. Por otro lado, en la segunda tabla se detallan los posibles costes para la implementación del operador en una empresa, asumiendo que ya poseen uno o más clústers de kubernetes ya operativos y que el personal de la empresa tiene los conocimientos y habilidades para gestionarlos.

## 7.1 Desarrollo

Para el desarrollo del operador solo fue necesario el uso de un ordenador personal junto con software libre y gratuito, por lo que el mayor gasto es debido al tiempo empleado para el desarrollo.

| Tipo | Descripción | Cantidad | Coste Unitario | Total |
|---|---|---|---|---|
| Infraestructuras | clúster de kubernetes, funcionando en cualquier máquina de desarrollo, implementado con Kind | 1 | 0,00 € | 0,00 € |
| Licencias de software | Todo el software utilizado para el desarrollo es de código abierto y gratuito | - | 0,00 € | 0,00 € |
| Formación | Formación para las nuevas tecnologías utilizadas | 30 h | 15 €/h | 450,00 € |
| Investigación | Resolución de problemas e investigación de diferentes alternativas para cada uno | 40 h | 15 €/h | 600,00 € |
| Desarrollo | Implementación del operador y su api | 70 h | 15 €/h | 1050,00 € |
| Depuración | Detección y corrección de errores | 40 h | 15 €/h | 600,00 € |
| | | | | 2700,00 € |

**Tabla 7.1:** Presupuesto de desarrollo

## 7.2 Despliegue

Este presupuesto contempla el tiempo necesario para que el personal de la empresa se documente acerca del funcionamiento del operador. Una pequeña prueba de concepto a modo demostrativo que permita descubrir problemas, incompatibilidades o necesidades no cubiertas por el operador. El despliegue del operador, primero en un entorno de pruebas, para confirmar que funciona de la forma esperada en un entorno muy parecido a un entorno productivo de la empresa. Posteriormente, el despliegue del operador en un entorno productivo, con las pruebas necesarias asociadas. Por último se incluye una ventana de 10 horas necesarias para mantenimiento, como podría ser la actualización del operador o la creación o modificación de Custom resources de las aplicaciones que lo utilizan. Este último apartado es una aproximación ya que depende enormemente del uso que se le de al operador, número de aplicaciones que lo utilicen y frecuencia de la necesidad de cambios en los Custom Resources.

| Tipo | Descripción | Cantidad | Coste Unitario | Total |
|---|---|---|---|---|
| Documentación | Recopilación y lectura de la documentación del proyecto | 10 h | 15 €/h | 150,00 € |
| Prueba de concepto | Pequeño despliegue a modo de prueba de concepto para detectar posibles incompatibilidades o problemas | 4 h | 15 €/h | 60,00 € |
| Despliegue en entorno de pruebas | Diseño del despliegue automatizado del operador en un entorno de pruebas. | 8 h | 15 €/h | 120,00 € |
| Despliegue en entorno productivo | Diseño e implementación del despliegue del operador en un entorno productivo | 16 h | 15 €/h | 240,00 € |
| Mantenimiento | Actualización del operador y creación de nuevos Custom Resources para nuevas aplicaciones | 10 h | 15 €/h | 150,00 € |
| | | | | 720,00 € |

**Tabla 7.2:** presupuesto de despliegue

## 7.3 Aplicaciones

La implementación del operador se realiza a través de manifiestos de kubernetes, dichos manifiestos incluyen la definición del pod del operador así como los Custom

Resource Definition. Una vez aplicados dichos manifiestos, kubernetes, usando sus mecanismos para acabar en el estado deseado indicado, descargará la imágen del contenedor del operador del repositorio indicado en el manifiesto y la ejecutará en uno de los nodos del clúster. Una vez el pod esté funcionando, en cuanto el usuario cree un custom resource del pod en el clúster, el clúster notificará al pod del cambio. El pod según los datos indicados en el Custom Resource creado rotará las credenciales indicadas en cuanto se cree el Custom Resource y, posteriormente, en intervalos de tiempo definidos en el propio Custom Resource.

Las aplicaciones que usen secretos pueden llevar un Custom Resource del operador, con la configuración específica para esa aplicación y ese entorno, de tal forma que cuando se despliegue la aplicación, se despliegue también la configuración necesaria para rotar los secretos de esa aplicación, además al eliminar la aplicación, se eliminaría también la configuración asociada, evitando acumular Custom Resources que ya no tienen uso.

# Capítulo 8 Título del Apéndice 1

## 8.1 Algoritmo 1 Intersección con bucle anidado

```
**********************************************************************************
*
* Javier Herrera Serpa
*
*
* 12/04/2024
*
*
* Este algoritmo recibe dos arrays y devuelve la intersección de ambos arrays, para ello por cada elemento de la primera lista, recorre toda la segunda lista en busca de una coincidencia. Tiene una complejidad O(N²)
*
*
**********************************************************************************/
func intersectionNestedFor(keys1 []string, keys2 []string) []string {
    var result []string
    for _, element1 := range keys1 {
        for _, element2 := range keys2 {
            if element1 == element2 {
                result = append(result, element1)
```

```
                }
        }
    }
    return result
}
```

## 8.2 Algoritmo 2 Intersección con bucle anidado con "break"

```
*****************************************************************************
*
* Javier Herrera Serpa
*
*
* 12/04/2024
*
*
* Este algoritmo recibe dos arrays y devuelve la intersección de ambos arrays, en caso de encontrar una coincidencia detiene el bucle y pasa al siguiente elemento, por lo que en un caso ideal es más eficiente que el caso anterior. Su complejidad es O(N²)
*
*
*****************************************************************************/
func differenceNestedForBreak(keys1 []string, keys2 []string) []string {
    var result []string
    for _, element1 := range keys1 {
        pair_found := false
        for _, element2 := range keys2 {
            if element1 == element2 {
                pair_found = true
                break
            }
        }
        if !pair_found {
            result = append(result, element1)
        }
    }
    return result
}
```

## 8.3 Algoritmo 3 Intersección con tabla hash

```
*****************************************************************************
*
* Javier Herrera Serpa
*
*
```

```
* 12/04/2024
*
*
* Este algoritmo recibe dos arrays y devuelve la intersección de ambos arrays, para ello obtiene el
array de menor tamaño, lo transforma a una tabla hash, luego recorre una sola vez el array mayor
buscando coincidencias con la tabla hash generada. Su complejidad es O(log N))
*
*
*******************************************************************************/
func intersectionHashmap(keys1 []string, keys2 []string) []string {
    var result []string
    var keysIter1 []string
    var keysIter2 []string
    keysMap := make(map[string]bool)
    if len(keys2) > len(keys1) {
            keysIter1 = keys1[:]
            keysIter2 = keys2[:]
    } else {
            keysIter1 = keys2[:]
            keysIter2 = keys1[:]
    }
    for _, element := range keysIter1 {
            keysMap[element] = true
    }
    for _, element := range keysIter2 {
            if keysMap[element] {
                    result = append(result, element)
            }
    }
    return result
}
```

# Bibliografía

https://kubernetes.io/docs/concepts/overview/components/
https://book.kubebuilder.io/
https://pkg.go.dev/regexp/syntax
https://pkg.go.dev/k8s.io/api/core/v1#Secret
https://github.com/getsops/sops
https://github.com/external-secrets/external-secrets