

Pruebas de Mutación de APIs Web: Un Enfoque de Caja Negra

Ana B. Sánchez, Alberto Martín-Lopez, Sergio Segura, and Antonio Ruiz-Cortés

Smart Computer Systems Research and Engineering Lab (SCORE)
Research Institute of Computer Engineering (I3US), Universidad de Sevilla, Spain
{anabsanchez,alberto.martin,sergiosegura,aruiz}@us.es

Resumen Las Interfaces de Programación de Aplicaciones (APIs) web tienen un papel clave en la integración de aplicaciones, por lo que validar su correcto funcionamiento es crucial. La mayoría de técnicas de generación de casos de prueba en este ámbito son de caja negra y suelen evaluarse con APIs sin código fuente disponible. Por ello, no es posible emplear pruebas de mutación tradicional, y no podemos cuantificar la efectividad de las pruebas si la API no expone ningún error. Este artículo propone un enfoque de caja negra para evaluar la capacidad de detección de errores de las pruebas para APIs web: en lugar de crear variantes defectuosas del código del programa (pruebas de mutación tradicionales), se crean mutantes de las salidas del programa (respuestas HTTP). JSONMutator es la herramienta implementada para llevar a cabo estas mutaciones. Los oráculos de pruebas se aplican sobre dichos mutantes y, a medida que las pruebas fallan, la cobertura de mutación aumenta. Experimentos preliminares sugieren que la herramienta es efectiva para crear mutantes a partir de la salida de APIs web y que existe correlación entre la cobertura de mutación de caja negra y caja blanca.

Palabras clave: Pruebas de mutación · APIs web · JSON

1. Introducción

Las Interfaces de Programación de Aplicaciones (APIs) web permiten a los sistemas software interactuar entre ellos a través de la red, normalmente utilizando servicios web [23]. Las APIs web son una pieza fundamental para la integración de software, por lo que su uso está cada vez más extendido en la industria. Directorios de APIs populares como ProgrammableWeb [2] y RapidAPI [3] indexan actualmente más de 24 mil y 30 mil APIs web respectivamente. Dado el papel clave de las APIs web en el desarrollo y la integración de sistemas software, validar su correcto funcionamiento es crucial. Un error en una API podría afectar a decenas o cientos de otros servicios que dependan de ella. Las pruebas de APIs web suelen realizarse con un enfoque de caja negra, es decir, mediante el envío de peticiones HTTP a la API y el análisis de las respuestas HTTP obtenidas, en base a distintos oráculos como comprobaciones en el código de estado y la conformidad de la respuesta con la especificación de la API.

Las pruebas de caja negra de APIs web son un tema de investigación activo hoy en día [7,15,17,25], y la mayoría de las contribuciones muestran resultados prometedores en la generación de suites (o *bancos*) de pruebas a nivel de sistema. Sin embargo, la eficacia de estos bancos de pruebas solo puede evaluarse en función de los fallos que se pueden detectar en el sistema bajo prueba. Cuando este sistema no expone ningún fallo, la capacidad de detección de fallos del banco de pruebas resulta desconocida. A diferencia de las pruebas de caja blanca, en las que se puede utilizar la cobertura de mutación para evaluar la eficacia de detección de fallos de un banco de pruebas, los enfoques de caja negra sólo pueden recurrir a cubrir la mayor cantidad posible de funcionalidad de la API con la esperanza de provocar fallos en el sistema [18].

En este artículo, presentamos la *cobertura de mutación de caja negra* como un enfoque novedoso para evaluar la eficacia de la detección de fallos de los conjuntos de pruebas para APIs web cuando no se tiene acceso al código de dichos sistemas. En lugar de crear variantes defectuosas de la API —imposible sin acceso al código fuente— proponemos crear variantes defectuosas de sus respuestas HTTP, es decir, de los resultados devueltos por el conjunto de pruebas. Una respuesta HTTP se compone de un cuerpo, un código de estado y una serie de cabeceras. Todos estos elementos se tienen en cuenta a la hora de crear mutantes de las respuestas HTTP. Para el cuerpo de la respuesta, nos centramos en el lenguaje JSON, ya que es el formato estándar de intercambio de datos en las actuales APIs web. Los oráculos de pruebas se aplican sobre dichos mutantes de las respuestas. A medida que las pruebas fallan, los mutantes mueren demostrando una mayor capacidad del conjunto de pruebas para detectar errores.

Para dar soporte a la propuesta, hemos desarrollado una herramienta, *JSON-Mutator* [1], que integra diversos operadores de mutación inspirados en errores reales en las APIs de Spotify y YouTube. Hemos llevado a cabo una evaluación dividida en dos experimentos. En el primer experimento, evaluamos la capacidad de generación de mutantes de JSONmutator. Para ello, hemos seleccionado cuatro APIs comerciales, hemos creado un banco de 10 pruebas para cada una y hemos generado todos los mutantes posibles. En el segundo experimento, estudiamos la correlación entre la cobertura de caja negra y caja blanca. En este caso, hemos utilizado como caso de estudio una API web de código abierto, para la que hemos creado cuatro bancos de pruebas ordenados de menor a mayor exhaustividad. Para cada banco de pruebas, hemos calculado la cobertura de mutación de caja blanca (enfoque tradicional) y caja negra (nuestra propuesta). Los resultados de los experimentos indican que nuestra herramienta JSONmutator es capaz de generar miles de mutantes en cuestión de segundos, y que la cobertura de mutación de caja blanca y de caja negra muestra cierta correlación. Aspiramos a que nuestra herramienta y nuestra propuesta puedan dar soporte a la evaluación de la capacidad de detección de fallos de bancos de pruebas para APIs web cuando el código de estas no está disponible.

El resto del artículo está estructurado como sigue. La Sección 2 describe el contexto de nuestro trabajo. La Sección 3 presenta la visión general de nuestra propuesta. Proponemos un conjunto de operadores de mutación para respuestas

HTTP de APIs web en la Sección 4. La Sección 5 presenta la herramienta de mutación JSONMutator. Evaluamos nuestro enfoque en la Sección 6. Los trabajos relacionados y las conclusiones aparecen en las Secciones 7 y 8, respectivamente.

2. Contexto

2.1. Pruebas de mutación

Las pruebas de mutación conforman una técnica ampliamente conocida para evaluar y mejorar la calidad de los conjuntos de pruebas de un sistema software [22]. Esta técnica consiste en insertar cambios sintácticos simples en el código original con la esperanza de que se asemejen a errores reales plausibles (hipótesis del programador competente [5]) y que esos cambios simples sean capaces de descubrir otros cambios más complejos (hipótesis del efecto de acoplamiento [10]). Los cambios simulando errores que son introducidos en el código original son conocidos como *mutaciones* y las nuevas versiones defectuosas del programa bajo prueba se conocen como *mutantes*. Una vez generados los mutantes, cada mutante, así como el programa original, se ejecutan contra el conjunto de pruebas. Si alguna de las pruebas falla, dicho mutante se clasifica como detectado o *muerto*. Por el contrario, si no falla ninguna prueba, el mutante se dice que permanece *vivo* y requiere un análisis más profundo, ya que puede indicar una carencia en la capacidad de detección de fallos del conjunto de pruebas. Sin embargo, también puede ocurrir que el mutante sea funcionalmente equivalente al programa original y en ese caso será nombrado como mutante *semánticamente equivalente*. Por lo tanto, el objetivo del ingeniero de pruebas debería aspirar a “eliminar” tantos mutantes como sea posible para aumentar la capacidad de detección del conjunto de pruebas. El porcentaje de mutantes eliminados respecto al total del conjunto de mutantes no equivalentes, denominado *mutation score* en inglés, determina la cobertura de mutación del conjunto de casos de prueba.

La introducción de mutaciones se suele sistematizar con el desarrollo de *herramientas de mutación*, que implementan diferentes *operadores de mutación*. Estos operadores se aplican cada vez que se encuentra un patrón en el programa (p. ej., cada aparición del operador relacional “>” se sustituye por “<”). Idealmente, un buen operador de mutación debería producir mutantes que no sean fáciles de detectar, de modo que pueda sacar a la luz las debilidades del conjunto de pruebas. Por el contrario, debería minimizar la generación de mutantes *equivalentes* (que no haya ninguna entrada que pueda detectar la mutación).

2.2. Pruebas de APIs web

Las APIs web permiten el consumo de datos y servicios a través de Internet. La mayoría de APIs web hoy en día siguen el estilo arquitectónico REST (REpresentational State Transfer) [13], siendo denominadas APIs web RESTful (o simplemente APIs REST). Una API REST permite la gestión de recursos (p. ej., un

tweet en la API de Twitter) mediante operaciones de creación, consulta, actualización o eliminación (operaciones *CRUD*), invocadas mediante peticiones HTTP (p. ej., GET y POST) a distintas URIs (p. ej., <https://api.twitter.com/2/tweets>).

Dado el papel clave de las APIs web —y más concretamente de las APIs REST— en la integración de software, probar su correcto funcionamiento resulta fundamental. Un caso de prueba para una API web está compuesto de una o varias peticiones HTTP, y un conjunto de comprobaciones en sus respuestas. Por ejemplo, para probar la funcionalidad de “añadir canciones a una lista” en la API de Spotify, tendríamos que proceder en dos pasos: 1) enviar una petición HTTP POST a la URI https://api.spotify.com/v1/playlists/{playlist_id}/tracks conteniendo la información de una canción y el identificador de la lista donde añadirla; y 2) comprobar que la respuesta HTTP es exitosa (código de estado 201), es decir, la canción se ha añadido a la lista.

Para evaluar la efectividad de las técnicas de pruebas de APIs web, en ocasiones no es posible recurrir a métricas tradicionales tales como la cobertura de código o la cobertura de mutación, puesto que no se tiene acceso al código del sistema (pensemos en APIs comerciales como Twitter o Spotify). Por este motivo, en nuestro trabajo previo [18], concebimos un total de diez criterios de cobertura de caja negra para pruebas de APIs REST. Estos criterios permiten medir la cobertura alcanzada por un banco de pruebas en función de las entradas (p. ej., parámetros) y salidas (p. ej., códigos de estado) que ejercitan. Los criterios se distribuyen en ocho niveles de cobertura (TCLs), donde TCL0 representa el mínimo nivel de cobertura, y TCL7 representa el máximo. La Tabla 1 muestra los niveles del modelo de cobertura, así como los criterios que deben ser cubiertos para considerar una suite de pruebas perteneciente a dicho nivel.

Nivel	Criterios de cobertura de entradas	Criterios de cobertura de salidas
TCL0		
TCL1	Rutas	
TCL2	Operaciones	
TCL3	Tipo de contenido	Tipo de contenido
TCL4	Parámetros	Clases de códigos de estado
TCL5	Valores de parámetros	Códigos de estado
TCL6	Valores de parámetros	Propiedades del cuerpo de la respuesta
TCL7	Secuencias de operaciones	

Tabla 1. Criterios de cobertura de APIs REST organizados por niveles.

3. Visión general de nuestra propuesta

Nuestro trabajo propone un novedoso enfoque de pruebas de mutación de caja negra para evaluar la capacidad de detección de errores de los conjuntos de pruebas de las APIs web. La Figura 1 muestra la visión general de nuestra propuesta. En lugar de crear variantes defectuosas de la API (como se haría en la mutación tradicional), proponemos la creación de mutantes de las salidas de la API, es decir, de las respuestas HTTP devueltas. Este enfoque está basado en el hecho de que cualquier error en la API debería reflejarse en menor o mayor medida en la salida de la API, ya sea con una estructura incorrecta, datos erróneos, códigos de estados inesperados, etc. Una vez mutadas las respuestas

HTTP de la API bajo prueba, utilizaremos los oráculos de los casos de prueba (generados automática o manualmente) para aplicarlos sobre dichos mutantes y evaluar así la cobertura de la mutación. A medida que las pruebas fallen, los mutantes morirán (esto es, los mutantes serán detectados), demostrando así una mayor capacidad del conjunto de pruebas para detectar errores.

Resulta importante destacar que esta propuesta permite evaluar la calidad de los oráculos de prueba, pero no de las entradas ya que no es posible medir la cantidad de código ejecutada. Así pues, este método debería emplearse de forma complementaria con otras técnicas que permitan garantizar una buena cobertura de los elementos de la API (parámetros, operaciones, códigos de salida, etc). Para ello, en este trabajo haremos uso de los criterios de cobertura de caja negra explicados en la sección anterior (2.2).

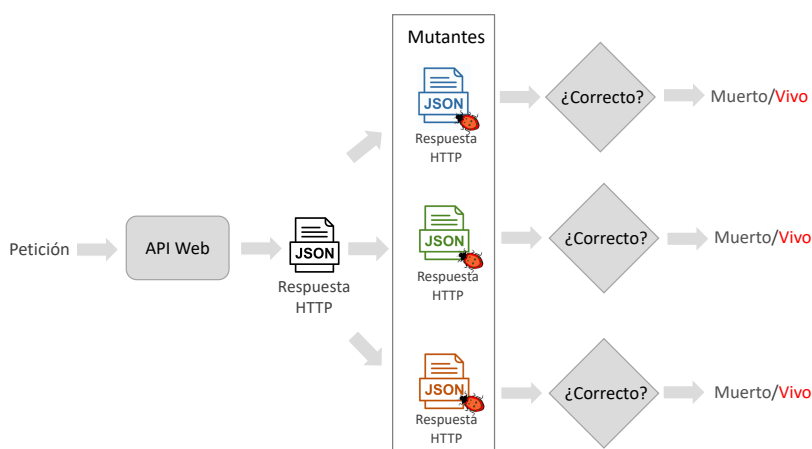


Figura 1. Enfoque de mutación de caja negra para pruebas en APIs web.

4. Operadores de mutación

En esta sección, presentamos operadores de mutación que se aplicarán a la respuesta HTTP de las APIs web. El proceso de definición de los operadores constó de dos fases. En primer lugar, estudiamos la literatura relacionada con las pruebas de mutación y perturbación en los servicios web [9,20]. Encontramos que hasta la fecha tan solo aparecen un par de trabajos donde se ha propuesto mutación para documentos JSON con objetivos diferentes al nuestro (ver Sección 7). De este modo, identificamos un vacío en cuanto a la evaluación de pruebas de caja negra en APIs web. En segundo lugar, y con el objetivo de definir operadores de mutación realistas, extrajimos y analizamos errores en los sistemas de seguimiento de errores de dos APIs web ampliamente conocidas, Spotify¹ y YouTube.² Examinamos aproximadamente 400 errores reales en estas APIs. De

¹ <https://github.com/spotify/web-api/issues>

² <https://issuetracker.google.com/issues?q=issues>

ellos, 80 y 41 errores válidos para nuestro propósito se seleccionaron a partir de Spotify y Youtube, respectivamente. Los errores fueron clasificados para facilitar su comprensión y la posterior definición de operadores de mutación.

Inspirados por la información extraída sobre los errores reales de las APIs estudiadas, proponemos un conjunto de operadores de mutación que agrupamos en tres categorías: operadores de mutación para cabeceras de respuesta HTTP, operadores sobre el código de estado HTTP, y operadores de mutación para el cuerpo de respuesta JSON. También hemos definido algunos operadores básicos, que aunque no se hayan encontrado referencias a ellos en los sistemas de seguimiento de incidencias analizados, completan la gama de operadores sobre mensajes de respuesta HTTP.

4.1. Operadores de mutación para cabeceras de respuesta HTTP

Esta sección presenta seis operadores de mutación para las cabeceras de respuesta de mensajes HTTP. Todos ellos están inspirados en errores reales. La Tabla 2 detalla el nombre y una breve descripción de cada operador.

Nombre	Descripción
CCT	Cambia el valor de la cabecera Content-Type.
ECT	Elimina la cabecera Content-Type.
CPC	Cambia el valor de la propiedad Charset de la cabecera Content-Type.
EPC	Elimina la propiedad Charset de la cabecera Content-Type.
CL	Cambia el valor de la cabecera Location.
EL	Elimina la cabecera Location.

Tabla 2. Operadores de mutación para cabeceras de respuesta HTTP.

Cambiar la cabecera Content-Type (CCT). El operador CCT modifica el valor de la cabecera HTTP *Content-Type*. Esta cabecera se utiliza para indicar al cliente el tipo del contenido de la respuesta, y saber así cómo procesarla correctamente. Si esta información no se especifica bien, los datos no se procesarán apropiadamente.

Eliminar la cabecera Content-Type (ECT). El operador ECT elimina la cabecera *Content-Type* de un mensaje de respuesta HTTP. Está basado en errores reales que ocurren cuando dicha cabecera está ausente en la respuesta³.

Cambiar la propiedad Charset (CPC). Este operador modifica el valor de la propiedad *charset* en la respuesta de una cabecera HTTP. La propiedad *charset*, también conocida como “conjunto de caracteres”, designa la codificación de los caracteres en el contenido de la respuesta al cliente. El operador CPC está basado en errores encontrados en la API de Spotify, como el error con ID #291 que dice así: “Incorrect charset specified in response header for audio-features (UTF8 vs utf-8)”⁴.

Eliminar la propiedad Charset (EPC). El operador EPC elimina la propiedad *charset* de la cabecera *Content-Type*.

³ <https://github.com/spotify/web-api/issues/1339>

⁴ <https://github.com/spotify/web-api/issues/291>

Cambiar la cabecera Location (CL). El operador CL modifica el valor de la cabecera *Location*. Esta cabecera indica la URL a la que debe redirigirse una página. Sólo se devuelve en las respuestas HTTP cuando se entrega con una respuesta de estado 3XX (redirección) o 201 (creado). Si la cabecera *Location* tiene un valor incorrecto, el cliente no sabrá cuál es la ruta del recurso.

Eliminación de la cabecera Location (EL). Este operador de mutación borra la cabecera *Location* del mensaje de respuesta HTTP.

4.2. Operadores de mutación para códigos de estado HTTP

Esta sección presenta tres operadores que mutan el código de estado de una respuesta HTTP. Los códigos de estado son emitidos por un servidor en respuesta a una petición de un cliente, e indican si la petición se ha resuelto de manera satisfactoria o no, entre otras cosas. La Tabla 3 presenta el nombre y una breve descripción de cada operador de mutación propuesto. Todos los operadores están inspirados en errores reales reportados en APIs web.

Nombre	Descripción
R5XX	Reemplaza el código de estado por un código de error de servidor en el rango 5XX.*
R2XX	Reemplaza el código de estado por un código de estado de éxito en el rango 2XX.◊
R4XX	Reemplaza el código de estado con un código de error en cliente en el rango 4XX.◊

* Códigos de estado de error de servidor seleccionados: 500, 502, 503, 504.

◊ Códigos de estado seleccionados: 200, 201, 204, 400, 401, 403, 404, 409.

Tabla 3. Operadores de mutación para el código de estado de respuesta HTTP.

Reemplazar por código de estado 5XX (R5XX). Este operador reemplaza el código de estado de una respuesta HTTP por un código de error en servidor en el rango 5XX: 500, 502, 503 o 504. Estos códigos de estado indican que el servidor ha fallado al completar una petición aparentemente válida. Cabe destacar que tanto este operador como los siguientes solo se aplican si el código de estado original se encuentra en un rango distinto, en este caso, 2XX o 4XX.

Reemplazar por código de estado 2XX (R2XX). El operador R2XX reemplaza el código de estado de una respuesta HTTP por un código de estado de éxito en el rango 2XX. En concreto, utilizamos los códigos más comunes: 200, 201 y 204. Los códigos de estado 2XX indican que la petición del cliente fue recibida con éxito, entendida, y aceptada por el servidor.

Reemplazar por código de estado 4XX (R4XX). Este operador de mutación reemplaza el código de estado de un mensaje de respuesta HTTP por un código de error en el rango 4XX. Concretamente, utilizamos los código de error 4XX más comunes: 400, 401, 403, 404 and 409. Los códigos 4XX representan un error en cliente debido a una sintaxis incorrecta o una petición incompleta.

4.3. Operadores de mutación para datos JSON

Esta sección presenta 14 operadores de mutación para el cuerpo de respuesta de los mensajes HTTP en formato JSON. Estos operadores modifican la respuesta mediante la edición, eliminación o inserción de propiedades, objetos o arrays.

La Tabla 4 muestra el nombre, una breve descripción y el contexto de aplicación de estos operadores de mutación, que organizamos en tres grupos de acuerdo al elemento que modifican: listas (o arrays), objetos, y propiedades y sus valores.

Nombre	Descripción	Contexto
AEA	Añade un elemento a un array.	<i>Arrays</i>
EEA	Elimina un elemento de un array.	<i>Arrays</i>
IEA	Intercambia un elemento por otro en un array.	<i>Arrays</i>
ANA	Asigna el valor nulo a un array.	<i>Arrays</i>
APO	Añade una propiedad a un objeto.	<i>Objetos</i>
EPSO	Elimina una propiedad simple de un objeto.	<i>Objetos</i>
EPOO	Elimina una propiedad de tipo objeto de un objeto.	<i>Objetos</i>
CTP	Cambia el tipo de una propiedad.	<i>Propiedades</i>
IPB	Invierte el valor de una propiedad de tipo booleano.	<i>Propiedades</i>
ANP	Asigna el valor nulo a una propiedad.	<i>Propiedades</i>
RPN	Reemplaza el valor de una propiedad de tipo numérico con otro valor.	<i>Propiedades</i>
ACSP	Añade caracteres especiales a una propiedad de tipo cadena.	<i>Propiedades</i>
RCP	Reemplaza la cadena completa de una propiedad por otra cadena.	<i>Propiedades</i>
RECP	Reduce/Extiende la cadena de una propiedad a un longitud límite.	<i>Propiedades</i>

Tabla 4. Operadores de mutación para el cuerpo de respuesta en formato JSON.

Añadir un elemento a un array (AEA). Este operador inserta un elemento en una lista o array. El elemento puede ser de cualquier tipo permitido en un array, es decir, una cadena, un número, un objeto, etc. Este operador está basado en errores reales reportados en respuestas JSON con elementos incorrectos en una lista, tales como el del error con ID #960⁵ encontrado en el sistema de seguimiento de errores de Spotify.

Eliminar un elemento de un array (EEA). Este operador de mutación elimina un elemento de un array. Se encontraron errores reales relacionados con arrays y listas de elementos incompletas tanto en la API de Spotify como en la de Youtube, como por ejemplo, las incidencias con IDs #1096 y 130331391⁶ de Spotify y Youtube.

Intercambiar un elemento por otro en un array (IEA). El operador IEA intercambia un elemento por otro en una lista o array. Está basado en errores reales encontrados en respuestas JSON que contenían listas de elementos desordenados o en una posición errónea. Por ejemplo, podemos mencionar los errores con IDs #305 y #529⁷ de la API de Spotify.

Asignar valor nulo a un array (ANA). Este operador convierte un array a vacío. El operador ANA está basado en errores reales encontrados en respuestas JSON que contenían listas sin elementos, como por ejemplo, los errores con IDs #650 y 141100401⁸ de las APIs de Spotify y Youtube, respectivamente.

Añadir una propiedad a un objeto (APO). El operador APO añade una propiedad a un objeto. No hemos encontrado referencias a errores reales relacionados con este operador en los dos sistemas de seguimientos de errores analiza-

⁵ <https://github.com/spotify/web-api/issues/960>

⁶ <https://issuetracker.google.com/issues/130331391>

⁷ <https://github.com/spotify/web-api/issues/529>

⁸ <https://issuetracker.google.com/issues/141100401>

dos, sin embargo, haciendo un esfuerzo por cubrir las operaciones básicas sobre todos los elementos del cuerpo de respuesta, hemos considerado añadir también este operador.

Eliminar una propiedad simple de un objeto (EPSO). El operador EPSO borra una propiedad simple (es decir, de tipo número, cadena o booleana) de un objeto JSON. Este operador está basado en errores reales encontrados en mensajes de respuesta JSON que contienen objetos con alguna propiedad desaparecida, como por ejemplo, el error con ID #856⁹ encontrado en el sistema de seguimiento de errores de Spotify.

Eliminar una propiedad de tipo objeto de un objeto (EPOO). Este operador elimina una propiedad de tipo objeto dentro de un objeto JSON en un mensaje de respuesta. Este operador de mutación está inspirado en errores reales reportados en el sistema de seguimiento de errores de Spotify, tales como el error con ID #632¹⁰ que surge cuando un objeto JSON no contiene una propiedad de tipo objeto esperada.

Cambiar tipo de una propiedad (CTP). El operador CTP cambia el tipo de una propiedad en un objeto (p. ej., convirtiéndola de número a cadena). No hemos encontrado referencias a errores reales relacionados con este operador en los sistemas de seguimiento de errores estudiados, sin embargo, para intentar cubrir todas las operaciones básicas sobre los elementos del cuerpo de respuesta, hemos añadido también este operador.

Invertir propiedad booleana (IPB). Este operador invierte el valor de una propiedad booleana dentro de un objeto. Está basado en errores reales encontrados en la API de Spotify, como por ejemplo el error con ID #623¹¹ que reporta un problema con la propiedad booleana *is_playing*; mientras que se espera que la propiedad devuelva *false* cuando no hay música siendo reproducida, *is_playing* devuelve siempre *true*.

Asignar valor nulo a una propiedad (ANP). Este operador pone a *null* una propiedad de un objeto. Está basado en errores reales encontrados en la API de Spotify, donde se reportaron respuestas JSON conteniendo objetos con propiedades a *null* (error con ID #930¹²).

Reemplazar el valor de una propiedad numérica por otro valor (RPN). El operador RPN reemplaza el valor de una propiedad numérica dentro de un objeto por otro valor numérico. Está inspirado en errores reales reportados en respuestas con objetos conteniendo una propiedad con un valor incorrecto, como por ejemplo los errores con ID #152¹³ y #14 de la API de Spotify.

⁹ <https://github.com/spotify/web-api/issues/856>

¹⁰ <https://github.com/spotify/web-api/issues/632>

¹¹ <https://github.com/spotify/web-api/issues/623>

¹² <https://github.com/spotify/web-api/issues/930>

¹³ <https://github.com/spotify/web-api/issues/152>

Añadir caracteres especiales a una propiedad de cadena (ACSP). Este operador de mutación introduce caracteres especiales, tales como “/” o “*” en una propiedad de tipo cadena dentro de un objeto. Está basado en errores reales reportados en los dos sistemas de seguimiento de errores estudiados. Por ejemplo, podemos mencionar el error con ID #725¹⁴ de Spotify conteniendo una playlist para “AC/DC”.

Reemplazar el valor de propiedad de tipo cadena por otro (RCP). El operador de mutación RCP reemplaza el valor de una propiedad de tipo cadena en un objeto por otra cadena. Está inspirado en errores reales encontrados en Spotify, como por ejemplo el error con ID #124¹⁵, en el que las propiedades *next* y *href* devuelven una url con el valor “/users/###” en lugar de “/me”.

Reducir/Extender la cadena de una propiedad a una longitud límite (RECP). Este operador modifica el tamaño del valor de una propiedad de tipo cadena a una longitud límite. RECP está inspirado en errores reales encontrados en las APIs de Spotify y Youtube que reportaban propiedades conteniendo valores truncados o excediendo el número de caracteres permitidos. Podemos mencionar el error con ID 141319505¹⁶ de Youtube que dice así: “I am currently experiencing the problem that the descriptions of the videos coming from the Youtube API are being truncated to 150-160 characters”.

5. Herramienta de soporte

Para dar soporte a nuestra propuesta, hemos creado JSONMutator [1], una herramienta que implementa todos los operadores de mutación para datos JSON definidos en la sección 4, además de otros operadores adicionales. JSONMutator es *open source*, el código fuente está disponible en GitHub, y puede ser importada directamente en proyectos Java como una librería Maven.

La Figura 2 muestra un diagrama de clases simplificado de JSONMutator. Como se puede observar, JSONMutator está integrada por varios *mutadores* (clase *AbstractMutator*), los cuales a su vez están compuestos de varios *operadores* (clase *AbstractOperator*). Los operadores deben implementar el método *mutate*, el cual recibe un objeto de un tipo concreto (p. ej., una cadena) y le aplica una mutación concreta (p. ej., añadir caracteres especiales a dicha cadena). Los mutadores aglutinan todos los operadores posibles para cada tipo de propiedad JSON (objeto, array, null, cadena, booleano, número entero y número decimal). Los mutadores incluyen dos métodos principales: *mutate* y *getOperator*. El método *mutate* es simplemente una llamada al método *mutate* de un operador concreto, el cual es elegido de entre todos los operadores asociados a ese mutador mediante el método *getOperator*. Es posible realizar mutaciones de orden simple (una) o de orden compuesto (varias). Por último, JSONMutator

¹⁴ <https://github.com/spotify/web-api/issues/725>

¹⁵ <https://github.com/spotify/web-api/issues/124>

¹⁶ <https://issuetracker.google.com/issues/141319505>

tiene la capacidad de crear un solo mutante (método *mutate*) o todos los mutantes posibles (método *getAllMutants*), pudiendo limitar los mutantes generados a un número concreto.

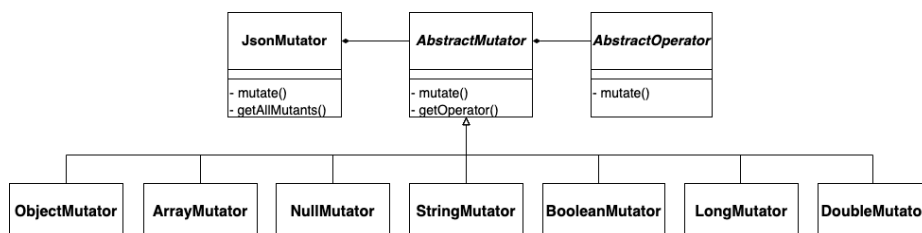


Figura 2. Diagrama de clases de JSONMutator.

6. Evaluación

La evaluación de nuestra propuesta está dividida en dos experimentos. En el primer experimento, evaluamos la capacidad de generación de mutantes de nuestra herramienta (total y clasificados por tipo). En el segundo experimento, evaluamos la relación entre las pruebas de mutación de caja blanca y caja negra, comparando la cobertura de mutación y de líneas de código.

Para el primer experimento, se han utilizado cuatro suites de pruebas de un tamaño de 10 casos de prueba (generados con RESTest [17]) cada una, pertenecientes a cuatro APIs REST distintas. Cada caso de prueba incluye únicamente una llamada HTTP, por lo que solo se generan mutantes de una respuesta HTTP. Después, se ha ejecutado el método *getAllMutants()* de JSONMutator sobre cada respuesta. La Tabla 5 muestra los resultados obtenidos. Para cada suite de pruebas, se muestran el tamaño medio de las respuestas HTTP (en bytes) y el número de mutantes generados, incluyendo el total y clasificados por tipo. El tamaño medio de las respuestas es relevante puesto que determina el número de mutantes que se podrán generar (a mayor tamaño, más mutantes). Los operadores de mutación utilizados son los descritos en la Sección 4 relativos a objetos JSON (cuerpo de la respuesta). Tal como se ilustra en la tabla, nuestra herramienta es capaz de generar un conjunto amplio y diverso de mutantes—entre 350 (Stripe) y 2516 (LanguageTool). Conviene recordar que en la práctica nuestra herramienta puede ser configurada para limitar el número de mutantes generados haciendo la propuesta eficiente. La mayoría de mutaciones se realizan sobre propiedades (p. ej., cadenas), seguidas de objetos y arrays. El tiempo de generación de mutantes fue inferior a 10 segundos en todos los casos (con un ordenador de 16GB RAM y procesador Intel i5, ejecutando Windows 11).

Para el segundo experimento, hemos utilizado como caso de estudio la API REST YouTubeMock¹⁷, una implementación *open source* de la operación de búsqueda de la API de YouTube¹⁸. Se han generado cuatro bancos de pruebas,

¹⁷ <https://github.com/opensourcingsapis/YouTubeMock>

¹⁸ <https://developers.google.com/youtube/v3/docs/search/list>

ordenados por tamaño y exhaustividad, de acuerdo a los niveles de cobertura propuestos en [18] y descritos en la Sección 2.2. En concreto, los oráculos del banco de pruebas TCL3 (nivel 3 de cobertura) se limitan a comprobar la ausencia de errores de servidor (es decir, de códigos de estado 5XX). En el banco TCL4 se comprueba, además, que las respuestas de la API siguen el formato definido en la especificación. El banco TCL5 incluye otro oráculo: se comprueba el rango del código de estado (es decir, 2XX para llamadas exitosas y 4XX para llamadas erróneas). Por último, en el banco TCL6 se comprueba el valor exacto del código de estado y se incluyen aserciones específicas sobre el cuerpo de la respuesta (p. ej., debe existir una propiedad x con valor “ y ”). En relación a los operadores de mutación utilizados, para el enfoque de caja negra se utilizan los mismos que en el experimento anterior, y para el enfoque de caja blanca usamos los operadores habilitados por defecto en la herramienta PIT¹⁹. La Tabla 6 muestra los resultados de la evaluación, donde se puede apreciar que el banco de pruebas TCL3 (menor tamaño y exhaustividad) consigue un 12% de cobertura de caja blanca y un 0% de caja negra, ya que los oráculos utilizados no consiguen detectar los cambios introducidos en las salidas del programa (respuestas HTTP mutadas). Por contra, el banco de pruebas TCL6 (mayor tamaño y exhaustividad) consigue un 57% y 51% de cobertura de caja blanca y negra, respectivamente, lo que sugiere que nuestra propuesta es efectiva para evaluar la capacidad de detección de errores cuando no se dispone del código fuente del programa.

API	Tamaño medio de respuestas HTTP en bytes	Mutantes generados			
		Objetos	Arrays	Propiedades	Totales
Amadeus	334.8	69	55	485	609
LanguageTool	1571.4	584	93	1839	2516
Stripe	238	58	7	285	350
Yelp	1217.4	250	133	1443	1826

Tabla 5. Mutantes generados para suites de 10 pruebas en cuatro APIs comerciales.

Test suite	Casos de prueba	Caja blanca			Caja negra	
		Mutantes	Cobertura de código	Cobertura de mutación	Mutantes	Cobertura de mutación
TCL3	1	723	69%	12%	852	0%
TCL4	6	723	81%	14%	2797	45%
TCL5	18	723	86%	32%	6878	45%
TCL6	19	723	87%	57%	7082	51%

Tabla 6. Cobertura de caja blanca y caja negra en función del tamaño y exhaustividad del banco de pruebas.

7. Trabajos relacionados

Nuestro trabajo está relacionado con herramientas de mutación y perturbación de datos, así como con enfoques de pruebas de caja negra de APIs web.

En cuanto a herramientas de mutación, las principales alternativas actuales se centran en mutación de código fuente en distintos lenguajes tales como Java (PIT [8]), Python (MutPy [11]) o JavaScript (Mutandis [19]), entre otros. Estas herramientas se diferencian en gran medida de nuestro enfoque puesto que se

¹⁹ <https://pitest.org/quickstart/mutators/>

suelen utilizar en pruebas de mutación de caja blanca donde el código del sistema está disponible. Algunos autores proponen operadores de mutación para distintos formatos de datos de intercambio en aplicaciones web, principalmente XML [12,16,20,24]. En los últimos años, también han surgido herramientas y trabajos de perturbación de datos en otros formatos, incluyendo JSON [4,9,14,21]. Sin embargo, estas propuestas se centran principalmente en la generación de *entradas* que puedan ser utilizadas en casos de pruebas. Por contra, nosotros proponemos un enfoque de pruebas de mutación, donde se mutan las *salidas* del programa para evaluar la capacidad de detección de errores de un banco de pruebas. Además, proponemos una herramienta de código abierto sobre la que es sencillo añadir nuevos operadores de mutación.

Las técnicas de generación automática de pruebas de APIs web, y especialmente las de APIs REST, son un tema de investigación muy activo en la actualidad. Algunas técnicas utilizan un enfoque de caja blanca [6], por lo que pueden recurrir a métricas como la cobertura de código o la cobertura de mutación para determinar la efectividad de los bancos de pruebas generados. Sin embargo, la mayoría de técnicas se basan en enfoques de caja negra [7,15,17,25], por lo que no pueden utilizar dichas métricas. Como un primer paso para abordar este problema, en nuestro trabajo previo [18], propusimos diez criterios de cobertura de APIs REST. Sin embargo, la cobertura de dichos criterios no implica que las pruebas sean efectivas para detectar errores, lo cual depende de los oráculos de pruebas utilizados —este problema es análogo a lo que ocurre con la cobertura de código. Este trabajo supone un paso adelante para la evaluación de la capacidad de detección de errores de técnicas de pruebas de caja negra.

8. Conclusiones

En este trabajo hemos presentado una novedosa aplicación de las pruebas de mutación sobre servicios web. Nuestro enfoque propone 23 operadores de mutación realistas sobre el mensaje de respuesta HTTP que devuelven las APIs web. Estos operadores están implementados en la herramienta JSONMutator y han sido evaluados utilizando cuatro APIs comerciales y una API de código abierto. Los resultados de la evaluación preliminar muestran que nuestra propuesta es efectiva tanto para generar mutantes como para evaluar la capacidad de detección de errores cuando no se dispone del código fuente del programa. Nuestro trabajo futuro perseguirá una evaluación más exhaustiva y con más APIs web que nos permita confirmar la validez de la propuesta y estudiar más a fondo la correlación entre cobertura de mutación de caja negra y caja blanca.

9. Agradecimientos

Trabajo parcialmente financiado por la Comisión Europea (FEDER), la Junta de Andalucía bajo los proyectos APOLO (US-1264651) y EKIPMENT-PLUS (P18-FR-2895), el Gobierno de España bajo el proyecto HORATIO (RTI2018-101204-B-C21), financiado por: FEDER/Ministerio de Ciencia e Innovación – Agencia Estatal de Investigación y el Programa de becas FPU, concedido por Ministerio de Educación y Formación Profesional de España (FPU17/04077).

Referencias

1. JSONmutator. <https://github.com/isa-group/JSONmutator>, accessed April 2022
2. ProgrammableWeb, <http://www.programmableweb.com/>, accessed May 2022
3. RapidAPI, <https://rapidapi.com>, accessed April 2022
4. Snodge. <https://github.com/npryce/snodge>, accessed May 2022
5. Acree, A.T., Budd, T.A., DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Mutation analysis. Tech. Rep. GIT-ICS-79/08, Georgia Institute of Technology (1979)
6. Arcuri, A.: RESTful API Automated Test Case Generation with EvoMaster. ACM TOSEM **28**(1), 1–37 (2019)
7. Atlidakis, V., Godefroid, P., Polishchuk, M.: RESTler: Stateful REST API Fuzzing. In: International Conference on Software Engineering. pp. 748–758 (2019)
8. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: PIT: A Practical Mutation Testing Tool for Java (demo). In: ISSTA. p. 449–452 (2016)
9. de Melo, A.C., Silveira, P.: Improving data perturbation testing techniques for web services. Information Sciences **181**(3), 600–619 (2011)
10. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: Help for the practicing programmer. Computer **11**(4), 34–41 (1978)
11. Derezińska, A., Hałas, K.: Analysis of mutation operators for the python language. In: International Conference on Dependability and Complex Systems DepCoS-RELCOMEX. pp. 155–164 (2014)
12. Emer, M., Vergilio, S., Jino, M.: A testing approach for XML schemas. In: International Computer Software and Applications Conference (COMPSAC’05). vol. 2, pp. 57–62 Vol. 1 (2005)
13. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. Ph.D. thesis, University of California, Irvine (2000)
14. Huang, X., Zhou, A., Jia, P., Liu, L., Liu, L.: Fuzzing the android applications with http/https network data. IEEE Access **7**, 59951–59962 (2019)
15. Karlsson, S., Causevic, A., Sundmark, D.: QuickREST: Property-based Test Generation of OpenAPI Described RESTful APIs. In: ICST. pp. 131–141 (2020)
16. Li, J.B., Miller, J.: Testing the semantics of w3c xml schema. In: 29th Annual International Computer Software and Applications Conference (COMPSAC’05). vol. 1, pp. 443–448 Vol. 2 (2005)
17. Martín-Lopez, A., Segura, S., Ruiz-Cortés, A.: RESTest: Black-Box Constraint-Based Testing of RESTful Web APIs. In: ICSOC. pp. 459–475 (2020)
18. Martín-Lopez, A., Segura, S., Ruiz-Cortés, A.: Test Coverage Criteria for RESTful Web APIs. In: A-TEST. pp. 15–21 (2019)
19. Mirshokraie, S., Mesbah, A., Pattabiraman, K.: Guided mutation testing for javascript web applications. IEEE TSE **41**(5), 429–444 (2015)
20. Offutt, J., Xu, W.: Generating test cases for web services using data perturbation. SIGSOFT Softw. Eng. Notes **29**(5), 1–10 (2004)
21. Olsthoorn, M., van Deursen, A., Panichella, A.: Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In: International Conference on Automated Software Engineering. p. 1224–1228 (2020)
22. Papadakis, M., Kintis, M., Zhang, J., Le Traon, Y., Harman, M.: Mutation testing advances: An analysis and survey. Advances in Computers (2017)
23. Richardson, L., Amundsen, M., Ruby, S.: RESTful Web APIs. O’Reilly Media, Inc. (2013)
24. Siblini, R., Mansour, N.: Testing web services. In: The 3rd ACS/IEEE International Conference on Computer Systems and Applications, 2005. pp. 135– (2005)
25. Viglianisi, E., Dallago, M., Ceccato, M.: RestTestGen: Automated Black-Box Testing of RESTful APIs. In: ICST. pp. 142–152 (2020)