UNIVERSIDAD DE LA REPÚBLICA URUGUAY

FACULTAD DE INGENIERÍA

Optimización de la tokenización y representación vectorial de encabezados HTTP para detección de ataques web

Informe de Proyecto de Grado presentado por

Joaquín Campo Nario, Mateo Hernán Daneri Dini

en cumplimiento parcial de los requerimientos para la graduación de la carrera de Ingeniería en Computación de Facultad de Ingeniería de la Universidad de la República

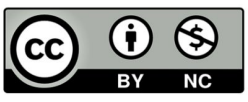
Supervisores

Gustavo Betarte
Rodrigo Martinez

Usuario experto

Amanda Riverol

Montevideo, Uruguay
9 de abril de 2026



Agradecimientos

Queremos agradecer a nuestras familias, que nos acompañaron en cada paso de este largo camino. Desde el primer día de la carrera hasta hoy, estuvieron presentes con una paciencia y una generosidad que significaron más de lo que podemos expresar en estas líneas. Cada logro de este recorrido también es de ellos. A Andrea, Fernando, Alicia, Anunciata, Mary, Plácido, Felicia, Graciela, Pablo y Mateo; Claudia, Gustavo, Camilo y Susana.

A nuestros amigos, que fueron parte esencial de este recorrido. Por las conversaciones, los mates, las distracciones necesarias y el interés genuino en cómo veníamos con esto. Por celebrar con nosotros los avances y por acompañarnos en los momentos en que todo parecía cuesta arriba. Buena parte de la energía que hay detrás de este trabajo viene de ahí, y queremos que sepan lo importante que fue contar con ellos.

A Pento y a Sabyk, las empresas donde tuvimos la oportunidad de crecer profesionalmente. Trabajar en ellas nos permitió aprender a ver las cosas desde otra perspectiva, complementando lo académico de una forma que no hubiéramos imaginado. Pero sobre todo, queremos agradecer a nuestros compañeros de trabajo, porque al final son las personas quienes hacen a las empresas, y fueron ellos quienes nos impulsaron a seguir creciendo y aprendiendo cada día.

Por último, agradecemos a nuestros tutores por su apoyo y soporte durante el desarrollo de este proyecto. Su conocimiento y experiencia fueron un aporte invaluable para este trabajo.

Resumen

Las solicitudes HTTP/1.1 constituyen una de las mayores superficies de ataque en aplicaciones web. Vulnerabilidades como *HTTP request smuggling*, inyección CRLF y suplantación de cabeceras explotan ambigüedades en la forma en que distintos componentes de la cadena de procesamiento interpretan un mismo mensaje. Los cortafuegos de aplicaciones web basados en reglas estáticas presentan limitaciones frente a la evolución constante de las técnicas de evasión y, en muchos entornos de producción, no se dispone de colecciones representativas de ataques para entrenar modelos supervisados.

Este trabajo presenta NeuralShield, un marco experimental para la detección de anomalías en solicitudes HTTP/1.1 que aborda de forma explícita la dualidad sintáctico-semántica del protocolo y opera en régimen de una sola clase, aprendiendo exclusivamente a partir de tráfico legítimo. La propuesta se articula en tres etapas. En primer lugar, se propone una taxonomía de cinco familias de anomalías estructurales observables en el paquete HTTP, derivada del análisis de las especificaciones del protocolo y de la investigación ofensiva. A partir de esta taxonomía se diseña un *pipeline* de preprocesamiento de doce pasos deterministas, organizado en tres fases (normalización, extracción de indicadores y serialización canónica), que opera bajo principios de idempotencia y no destructividad. En segundo lugar, sobre el artefacto canónico producido por el *pipeline* se construyen dos vistas complementarias: una vista sintáctica basada en TF-IDF con reducción PCA, que captura regularidades léxicas y estructurales, y una vista semántica basada en *embeddings* de SecBERT, que modela el significado funcional del tráfico. Cada vista alimenta un detector de anomalías no supervisado adaptado a sus supuestos estadísticos: LOF para la vista sintáctica y distancia de Mahalanobis para la semántica. Finalmente, las puntuaciones de ambos detectores se combinan mediante una fusión ponderada.

La evaluación experimental sobre tres conjuntos de datos de referencia (CSIC-2010, PKDD-2007 y SR_BH-2020) demuestra que el preprocesamiento estructural mejora de forma consistente la separabilidad entre tráfico legítimo y anómalo en las seis configuraciones de representación y detector evaluadas, con ganancias de hasta 0,283 en AUC. El análisis de complementariedad confirma que las vistas sintáctica y semántica capturan dimensiones distintas de la normalidad, con correlaciones entre 0,17 y 0,33. El *ensemble* ponderado supera a ambos detectores individuales en los tres conjuntos de datos, alcanzando valores de AUC

de 0,861, 0,780 y 0,928, respectivamente.

Palabras clave: detección de anomalías, HTTP, seguridad web, aprendizaje no supervisado, preprocesamiento estructural, representación dual, *ensemble*

Índice general

Capítulo 1

Introducción

1.1. Contexto, motivación y problema

El HyperText Transfer Protocol (HTTP) constituye el mecanismo de comunicación predominante en las aplicaciones web modernas. Cada solicitud HTTP transporta, además de la línea de petición y el cuerpo opcional, un conjunto de cabeceras que codifican metadatos de control: destino lógico del mensaje, credenciales de autenticación, directivas de caché, preferencias de contenido y parámetros de seguridad, entre otros. En las arquitecturas actuales, una solicitud suele atravesar una cadena de componentes intermedios (proxies, balanceadores de carga, Web Application Firewalls (WAFs), *gateways* (pasarelas) de Application Programming Interface (API) y servidores de aplicación) donde cada eslabón puede inspeccionar, reescribir o descartar cabeceras antes de entregarlas al siguiente salto. De este modo, las cabeceras no solo describen la solicitud, sino que influyen directamente en decisiones de ruteo, control de acceso, selección de *backend* y aplicación de políticas de seguridad a lo largo de todo el recorrido.

Esta centralidad convierte a las cabeceras en una superficie de ataque de primer orden. Organizaciones como Open Web Application Security Project (OWASP) documentan cómo múltiples categorías de riesgo crítico (inyección, control de acceso roto, falsificación de solicitudes del lado del servidor) se materializan con frecuencia a través de la manipulación de campos de cabecera [1]. Vulnerabilidades como *HTTP request smuggling*, inyección CRLF, *Host header injection*, suplantación de origen y contaminación de parámetros HTTP comparten un rasgo común: explotan ambigüedades o discrepancias en la forma en que distintos componentes de la cadena interpretan una misma solicitud.

La especificación de HTTP/1.1, codificada en los Request for Comments (RFC) 9110 y 9112, separa explícitamente la semántica del protocolo de su sintaxis. Sin embargo, por razones de retrocompatibilidad, admite un grado considerable de flexibilidad en la representación textual de los mensajes: caracteres obsoletos, tolerancia a codificaciones heterogéneas, mecanismos de continuación de línea y la posibilidad de que intermediarios reescriban campos en tránsi-

to. Estas "zonas grises" hacen que la interpretación efectiva de una solicitud pueda depender de la combinación concreta de componentes que intervienen en su procesamiento. El resultado es que pequeñas variaciones sintácticas (una cabecera duplicada, una codificación no canónica, un delimitador inesperado) pueden traducirse en interpretaciones semánticas radicalmente distintas según el componente que procese la solicitud.

Frente a estas amenazas, los WAFs basados en reglas estáticas constituyen la principal línea de defensa perimetral. Sin embargo, la evolución constante de las técnicas de evasión ha demostrado de forma reiterada las limitaciones de un enfoque puramente basado en firmas. A esta dificultad se suma una restricción operativa: en muchos entornos de producción no se dispone de colecciones representativas de ataques para entrenar modelos supervisados, ya que los ataques evolucionan con rapidez y la cobertura del dataset se vuelve insuficiente ante la aparición de técnicas nuevas. En consecuencia, la detección debe formularse como un problema de una sola clase (*one-class*), donde el modelo aprende la distribución del tráfico legítimo y marca como anómalas las solicitudes que se apartan significativamente de ese perfil.

La revisión de la literatura revela, además, dos vacíos relevantes. Por un lado, los enfoques de detección existentes abordan de forma aislada una de las dos dimensiones que caracterizan las solicitudes HTTP: algunos se centran en la forma estructural del mensaje, mientras que otros construyen representaciones contextuales profundas sobre el contenido. No se identifican enfoques no supervisados que aborden ambas dimensiones simultáneamente dentro de un mismo esquema de detección. Por otro lado, se identifica la ausencia de un procedimiento sistemático para trasladar los hallazgos de la investigación ofensiva (*fuzzing* diferencial, evasión de WAF, estudios de *parsing*) hacia los modelos de detección como señales explícitas.

1.2. Enfoque propuesto, objetivos y contribuciones

Para abordar los vacíos identificados, este trabajo propone *NeuralShield*, un marco experimental para la detección de anomalías en solicitudes HTTP/1.1 que se apoya explícitamente en la dualidad sintáctico–semántica del protocolo. El alcance se restringe a HTTP/1.1, que es el protocolo sobre el que se dispone de conjuntos de datos de referencia para la detección de anomalías en tráfico web; HTTP/2 y HTTP/3 se identifican como línea de trabajo futuro. NeuralShield se organiza en tres etapas. En primer lugar, un *pipeline* de preprocesamiento estructural recibe solicitudes HTTP crudas y las convierte en una representación canónica alineada con las especificaciones RFC, exponiendo mediante *flags* las desviaciones respecto de una línea base de solicitudes bien formadas. En segundo lugar, sobre el artefacto canónico se construyen dos vistas complementarias: una vista sintáctica (Term Frequency-Inverse Document Frequency (TF-IDF)/Principal Component Analysis (PCA)), que captura la forma

y las regularidades léxicas, y una vista semántica (SecBERT), que modela el significado funcional del tráfico. Finalmente, ambas vistas alimentan detectores de anomalías no supervisados (LOF y distancia de Mahalanobis, respectivamente) cuyas puntuaciones se combinan mediante una fusión ponderada.

El objetivo general es diseñar, implementar y evaluar este sistema, verificando que la modelización explícita de la dualidad sintáctico–semántica y el preprocesamiento estructural aportan valor medible en un escenario *one-class*. Los objetivos específicos son:

1. Analizar cómo las vulnerabilidades documentadas se manifiestan en la forma de las solicitudes HTTP y proponer una taxonomía de anomalías estructurales observables.

2. Diseñar e implementar un *pipeline* de preprocesamiento idempotente y no destructivo que exponga desviaciones mediante *flags* explícitas.

3. Construir una representación del artefacto canónico producido por el *pipeline*.

4. Evaluar el impacto del preprocesamiento sobre la capacidad de detección de modelos con supuestos estadísticos distintos.

5. Validar la complementariedad entre ambas vistas y la eficacia de su fusión frente a cada detector individual.

Las principales contribuciones del trabajo son: (i) una taxonomía de anomalías estructurales en solicitudes HTTP/1.1 organizada según la manifestación observable en el paquete, que identifica cinco familias de señales; (ii) un *pipeline* de preprocesamiento de 12 pasos deterministas; (iii) un modelo dual de representación y detección que articula una vista sintáctica y una vista semántica combinadas mediante fusión ponderada; y (iv) evidencia empírica, sobre tres conjuntos de datos de referencia, de que el preprocesamiento mejora consistentemente la detección, las vistas capturan anomalías de naturaleza complementaria y su fusión supera a ambos detectores individuales.

1.3. Estructura del documento

El resto de este documento se organiza de la siguiente manera. El Capítulo 2 presenta la revisión de antecedentes y el estado del arte: especificaciones de HTTP/1.1, familias de vulnerabilidades, fundamentos de aprendizaje automático para detección de anomalías y enfoques existentes de detección de tráfico HTTP malicioso. El Capítulo 3 analiza la dualidad sintáctico–semántica de las solicitudes HTTP y propone una taxonomía de anomalías estructurales observables. El Capítulo 4 presenta el diseño de NeuralShield: requisitos, *pipeline* de preprocesamiento, vistas de representación y módulo de detección. El Capítulo 5 documenta las decisiones de implementación. El Capítulo 6 describe la

evaluación experimental, incluyendo el impacto del preprocesamiento, la comparación entre vistas, los resultados del *ensemble* y la discusión de limitaciones. El Capítulo 7 concluye el trabajo y explica las líneas de trabajo futuro.

Capítulo 2

Revisión de antecedentes y estado del arte

Este capítulo presenta los antecedentes técnicos y científicos relevantes para el desarrollo de un sistema de detección de anomalías sobre tráfico HTTP basado en aprendizaje automático. En primer lugar se repasan las especificaciones del protocolo y el papel de las cabeceras como superficie de ataque. A continuación se sistematizan las principales familias de vulnerabilidades basadas en cabeceras HTTP, se revisan las tecnologías de WAFs y las técnicas de evasión, y se analizan las propuestas existentes de detección mediante *machine learning*.

2.1. Especificaciones de HTTP/1.1 y semántica básica

Siguiendo modelos como el Open Systems Interconnection (OSI) o el Protocolo de Control de Transmisión/Protocolo de Internet (TCP/IP), HTTP es un protocolo de capa de aplicación, sin estado, diseñado para sistemas de información distribuidos y orientado originalmente al intercambio de documentos de hipertexto (conjuntos de recursos enlazados mediante referencias que permiten "navegar" entre ellos interactivamente).

En lo que sigue de este documento, al referirnos a la especificación de HTTP estaremos hablando concretamente de las normativas publicadas en los RFC, propuestas de documentos donde se describen especificaciones sobre estándares técnicos de Internet. Emplearemos la noción de *estándar* en ese sentido. A su vez, en aquellas ocasiones que se mencionen aspectos técnicos del protocolo y no se provea una referencia explícita, asumiremos que dichos aspectos tienen como fuente estos documentos.

La especificación moderna separa explícitamente los aspectos de *semántica*

[2] del protocolo de los aspectos sintácticos [3], especificando cada uno en un documento individual.

En este sentido, una petición HTTP/1.1 se compone de una línea de petición (request-line), cero o más lineas de cabecera (field-line) y, opcionalmente, un cuerpo (message-body).

```
HTTP-message   = request-line CRLF
                 *( field-line CRLF )
                 CRLF
                 [ message-body ]
```

Listado 2.1: Gramática Augmented Backus-Naur Form (ABNF) de un mensaje HTTP/1.1 (RFC 9112)

En este documento nos enfocaremos en la línea de petición y las cabeceras, las cuales se representan como texto sobre un flujo de bytes en el que cada línea termina en una secuencia `CRLF`. Las estructuras definidas de cada una de las anteriores son las siguientes:

```
request-line = method SP request-target SP HTTP-version
field-line   = field-name ":" OWS field-value OWS
```

Listado 2.2: Gramática ABNF de la línea de petición y los campos de cabecera (RFC 9112)

En HTTP, las cabeceras son pares nombre-valor ("field-name" - "field-value") que transportan metadatos de control sobre la petición o la respuesta. A través de ellas se expresa, por ejemplo, el destino lógico del mensaje, las capacidades y preferencias del cliente, condiciones de autenticación, directivas de caché, o parámetros de representación del contenido.

Los RFC que se mencionan anteriormente definen HTTP/1.1, introducen una gramática detallada para la estructura de los mensajes y, en paralelo, describen la *interpretación* asociada a cada método, código de estado y campo de cabecera. Esta separación permite que distintas versiones del protocolo compartan las mismas nociones de recurso, método o cabecera, aun cuando difieran en el mecanismo de transporte subyacente.

Un aspecto relevante de la especificación de HTTP/1.1 es que, por razones de retrocompatibilidad y robustez, admite cierto grado de flexibilidad en la representación textual de los mensajes. La presencia de la categoría `obs-text` en la gramática (caracteres ASCII obsoletos a día de hoy), la tolerancia a ciertos esquemas de codificación de caracteres en los valores de cabeceras, o la posibilidad

de que agentes intermedios reescriban campos antes de entregarlos al siguiente salto son ejemplos de decisiones de diseño que, si bien facilitan la interoperabilidad, también introducen zonas grises donde la interpretación efectiva de una misma petición puede depender de la combinación concreta de componentes que intervienen en su procesamiento.

Además, la especificación distingue entre campos de cabecera de extremo a extremo (`end-to-end`) y campos salto a salto (`hop-by-hop`). Los primeros (como `Host` o `Authorization`) están destinados a ser interpretados por el servidor de origen y, en general, deben preservarse a través de proxies y otros intermediarios. Los segundos, en cambio, controlan aspectos de la conexión concreta entre dos nodos (por ejemplo, `Connection`, `TE` o campos declarados dentro de `Connection`) y no deberían reenviarse más allá del siguiente salto. Esta distinción es relevante en arquitecturas con cadenas largas de intermediarios, donde un manejo incorrecto de cabeceras *hop-by-hop* puede derivar en comportamientos inesperados o en la exposición accidental de metadatos pensados solo para enlaces internos [4], [5].

Finalmente, el conjunto de cabeceras estandarizadas está sujeto al registro y gestión centralizada por parte de la organización Internet Assigned Numbers Authority (IANA), a través del *HTTP Field Name Registry* [6], que establece qué nombres de campos están definidos, a qué especificación pertenecen y qué tipo de uso se espera de ellos. En paralelo, fuentes de documentación aplicada como Mozilla Developer Network (MDN) mantienen descripciones actualizadas de las cabeceras más utilizadas en la práctica y de su impacto en funcionalidades como autenticación, cacheo o control de políticas de seguridad [7]. Estos recursos sirven de puente entre la propuesta normativa de las RFC y los patrones de uso observables en ambientes reales.

En el resto de este trabajo nos concentraremos exclusivamente en peticiones HTTP/1.1. Si bien las RFC más recientes unifican la semántica del protocolo para distintas versiones (HTTP/1.1, HTTP/2 y HTTP/3), la presente investigación se acota a mensajes en formato texto conforme a la sintaxis de HTTP/1.1 definida en el *RFC 9112* [3]. Esto permite aislar el análisis a un único modelo de framing y representación de cabeceras, evitando introducir complicaciones adicionales.

2.2. Cabeceras HTTP como portadoras de metadatos críticos

En arquitecturas modernas, una solicitud suele atravesar una cadena de componentes que pueden inspeccionar, añadir, reescribir o eliminar cabeceras; clientes, proxies, balanceadores de carga, WAFs, `gateways` (pasarelas) de

API y servidores de aplicación. En cada eslabón, campos como `Host`, `Via`, `X-Forwarded-*`, `Authorization`, `Cookie` o `Accept-*` influyen en decisiones de ruteo, selección de `backend`, control de acceso, negociación de contenido o aplicación de cabeceras de seguridad en las respuestas. La forma en que estos componentes interpretan y transforman las cabeceras determina, en definitiva, el significado efectivo que se asigna a la petición a lo largo del recorrido.

Guías prácticas como el *Web Security Testing Guide* de OWASP sistematizan este papel central de las cabeceras al proponer conjuntos de pruebas específicas para verificar, por ejemplo, la correcta gestión de métodos HTTP, la robustez de la configuración de cabeceras de seguridad en respuestas, o la ausencia de cabeceras redundantes, conflictivas u obsoletas que puedan debilitar la seguridad de una aplicación [8]. A su vez, el documento OWASP Top 10 destaca que varias de las categorías de riesgo más críticas en aplicaciones web se materializan con frecuencia a través de la interpretación y uso que el servidor hace de determinados campos de cabecera [1].

2.3. Vulnerabilidades en el procesamiento de peticiones HTTP

Esta sección sintetiza las principales familias de vulnerabilidades asociadas al procesamiento de peticiones HTTP en arquitecturas web modernas. El foco está puesto en fallas estructurales: ambigüedades al delimitar mensajes, diferencias de interpretación entre componentes intermedios, y problemas de normalización en cabeceras, rutas y parámetros. Este marco permite conectar cada familia con sus impactos prácticos más frecuentes (evasión de controles, *cache poisoning*, secuestro de sesión, SSRF o denegación de servicio), manteniendo separadas sus causas técnicas.

2.3.1. Desincronización de mensajes HTTP (*HTTP request smuggling*)

Estas vulnerabilidades aparecen cuando un atacante explota discrepancias entre componentes consecutivos de la cadena HTTP para que asignen límites distintos al mismo flujo de bytes. En una explotación típica, el atacante envía una petición con delimitación ambigua: el *front-end* la da por finalizada y la reenvía, pero el *back-end* continúa consumiendo bytes residuales y los interpreta como una segunda petición inyectada. Ese desfase habilita acciones no previstas, por ejemplo cuando la petición residual alcanza rutas sensibles sin atravesar los mismos controles que aplicó el primer componente de la cadena. La literatura sobre *HTTP request smuggling* documenta una familia de vulnerabilidades en la que agentes intermedios HTTP (proxies, balanceadores, *gateways*) no coinciden en dónde termina una petición y comienza la siguiente [9], [10], [11]. Esta

desincronización suele surgir por ambigüedades de delimitación, por ejemplo en combinaciones conflictivas de `Content-Length` y `Transfer-Encoding`, o ante cabeceras duplicadas interpretadas de forma distinta por cada componente.

```
POST / HTTP/1.1
Host: app.example
Content-Length: 83
Transfer-Encoding: chunked

0

GET /admin HTTP/1.1
Host: app.example
X-SSL-CLIENT-CN: administrator
Foo: x
```

Listado 2.3: Ejemplo de *request smuggling* CL/TE: el front-end prioriza `Content-Length` y el back-end interpreta `Transfer-Encoding: chunked`

En el Listado 2.3, el *front-end* prioriza `Content-Length: 83` e interpreta que todo el cuerpo —incluyendo el `GET /admin` y sus cabeceras— forma parte de un único `POST`. Como consecuencia, no inspeciona ni modifica esas líneas: en particular, no aplica la política que normalmente le impide a tráfico externo incluir `X-SSL-CLIENT-CN` (cabecera interna que el propio *front-end* inyecta al verificar un certificado TLS de cliente). El *back-end*, en cambio, procesa el cuerpo con semántica `chunked`: el valor `0\r\n\r\n` es el terminador de bloque (cuerpo vacío) y los bytes restantes quedan en el *buffer* de la conexión TCP como inicio de una nueva petición. El *back-end* recibe así un `GET /admin` con `X-SSL-CLIENT-CN: administrator` intacto, y lo procesa con privilegios de administrador, sin que esa cabecera haya sido forjada a través de ningún control del *front-end* [12].

Trabajos como *HTTP Desync Attacks* [13] sistematizan variantes de este problema y muestran que la explotación práctica habilita secuestro de respuestas, *cache poisoning*, exfiltración de información y suplantación de usuarios. Las Figuras 2.1 y 2.2 ilustran este mecanismo: la primera muestra cómo el *front-end* y el *back-end* asignan límites distintos al mismo flujo de bytes, mientras que la segunda presenta el escenario en que un paquete malicioso inyectado aprovecha esa desincronización para alcanzar el servidor de origen sin haber sido inspeccionado por el componente de seguridad. Investigaciones recientes, como HTTP Garden [14] y WAFFLED [15], extienden esta línea con *fuzzing* diferencial sobre cabeceras, codificaciones y patrones de *chunking*, evidenciando que estas discrepancias de *parsing* aparecen en todo el ecosistema HTTP y reaparecen con nuevas variantes a medida que evolucionan las implementaciones.

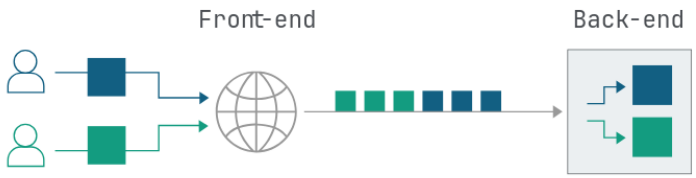


Figura 2.1: Desincronización entre front-end y back-end

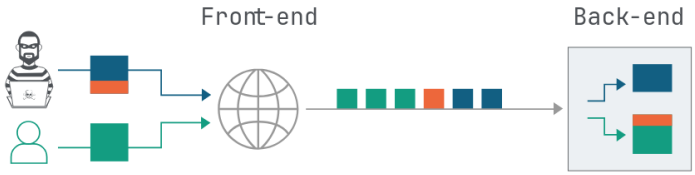


Figura 2.2: Paquete malicioso entre front-end y back-end

2.3.2. Inyección estructural en cabeceras HTTP (*CRLF Injection*)

En esta familia, el atacante logra que el servidor trate parte de una entrada de usuario como si fueran delimitadores del protocolo HTTP. Esto ocurre cuando una aplicación copia datos no confiables dentro de una cabecera de respuesta sin neutralizar `CR` y `LF`; al inyectar `%0d%0a` (los códigos asociados a CR y LF respectivamente), el adversario puede cerrar la línea actual e iniciar una cabecera nueva bajo su control [16].

```
1 GET /redir?next=/home%0d%0aSet-Cookie:%20rol=admin HTTP/1.1
2 Host: app.example
```

Listado 2.4: Ejemplo de inyección CRLF: secuencias `%0d%0a` en un parámetro permiten inyectar una cabecera `Set-Cookie` arbitraria

En el Listado 2.4, la aplicación construye `Location` con el parámetro `next` de la petición del atacante. Como el valor incluye `%0d%0a`, la respuesta termina conteniendo una cabecera adicional (`Set-Cookie`) que no fue emitida por la lógica prevista. En variantes más severas, la inyección puede incluso cerrar el bloque de cabeceras e iniciar una segunda respuesta (esto es denominado *HTTP response splitting*), lo que habilita efectos como manipulación de caché o entrega de contenido no esperado a clientes posteriores [17], [18]. OWASP encuadra este tipo de fallas dentro de A03:2021 (Injection), enfatizando que el problema de base no es el efecto final visible, sino la posibilidad de inyectar símbolos estruc-

turales del protocolo en puntos donde deberían existir únicamente datos [19].

2.3.3. Inyección de cabecera `Host` (*Host Header Injection*)

En *Host Header Injection*, el atacante induce a la aplicación a comportarse como si la petición hubiera sido dirigida a otra autoridad. Esto ocurre cuando el sistema usa `Host` para construir Uniform Resource Locator (URL)s absolutas, elegir *tenant* o aplicar decisiones de seguridad sin validar que el dominio recibido sea uno autorizado [20], [21], [22].

```
1 POST /forgot-password HTTP/1.1
2 Host: attacker.example
3 Content-Type: application/x-www-form-urlencoded
4
5 email=victima@empresa.com
```

Listado 2.5: Ejemplo de *Host Header Injection*: el atacante sustituye el valor de `Host` para envenenar el enlace de recuperación de contraseña

En el Listado 2.5, el servidor procesa la solicitud de recuperación para una cuenta legítima, pero al construir el enlace toma el valor de `Host` sin verificarlo. Como resultado, el correo llega a la víctima con un enlace que apunta al dominio del atacante. Si la víctima accede, el token de recuperación puede quedar expuesto y ser reutilizado para tomar control de la cuenta. La Figura 2.3 ilustra este flujo de *password reset poisoning*, mostrando los pasos mediante los cuales el atacante desvía el enlace de recuperación hacia un dominio bajo su control. El mismo patrón de confianza indebida en `Host` puede materializarse en redirecciones maliciosas, contaminación de caché compartida o disparo de solicitudes hacia recursos internos (SSRF) [23], [24], [25].

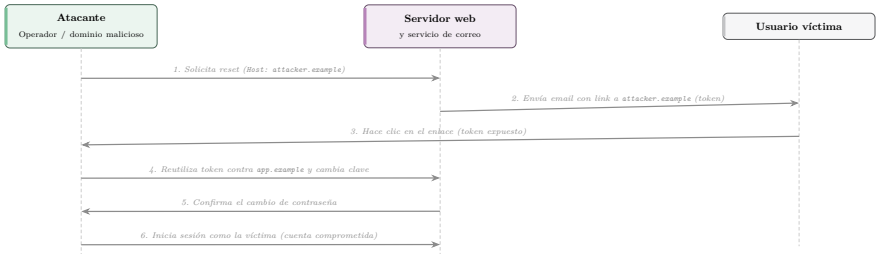


Figura 2.3: Host Header Injection: flujo de *password reset poisoning*.

2.3.4. Suplantación de origen mediante cabeceras de cliente

En esta familia, el atacante falsifica el origen aparente de la petición para, por ejemplo, influir en controles de acceso. En arquitecturas con proxies es común transportar metadatos de origen mediante cabeceras reenviadas como `X-Forwarded-For`, `Forwarded`, `Client-IP` o `True-Client-IP` [26], [27]. Estas cabeceras son útiles para observabilidad y trazabilidad, pero dejan de ser confiables si la aplicación no valida qué salto de la cadena está autorizado a establecerlas.

```
1 GET /admin HTTP/1.1
2 Host: app.example
3 X-Forwarded-For: 127.0.0.1
```

Listado 2.6: Ejemplo de suplantación de origen: la cabecera `X-Forwarded-For` falsificada simula una conexión desde `localhost`

En el Listado 2.6, el sistema interpreta `X-Forwarded-For` como prueba de que la petición proviene de un origen interno (localhost) y habilita acceso privilegiado. Como esa cabecera fue enviada directamente por el cliente y no por un proxy confiable, el valor de origen queda falsificado y se materializa el bypass. OWASP mantiene listados de cabeceras frecuentemente abusadas para *IP spoofing*, que se refiere a la suplantación del origen aparente de la petición, incluyendo variantes propietarias de la misma familia [28]. Reportes de pruebas de seguridad muestran este patrón de forma recurrente cuando aplicaciones o WAFs consumen estos campos sin una política estricta de confianza por saltos [29].

La literatura vincula esta familia con los riesgos del OWASP Top 10 A01:2021 (Broken Access Control), A05:2021 (Security Misconfiguration) y A10:2021 (SSRF), ya que la causa común es la confianza indebida en metadatos de cabecera que no están autenticados extremo a extremo [1].

2.3.5. Manipulación de rutas (*Path Traversal*)

En `Path Traversal`, el atacante consigue que la aplicación acceda a rutas del sistema de archivos fuera del directorio previsto para publicación. El mecanismo típico aparece cuando el servidor concatena una ruta base con un parámetro controlado por el cliente y no valida adecuadamente secuencias de escape como `../` [30].

```
GET /download?file=../../../../etc/passwd HTTP/1.1
Host: example.com
```

Listado 2.7: Ejemplo de *Path Traversal*: secuencias `../` en un parámetro permiten acceder a archivos fuera del directorio público

En el Listado 2.7, la aplicación pretende servir únicamente archivos bajo `/var/www/files/`, pero al resolver `../../../../etc/passwd` termina apuntando a un recurso del sistema fuera del directorio permitido. Si no existe canonicalización estricta y verificación de prefijo tras la resolución de la ruta, la petición puede exponer archivos sensibles que no forman parte del contenido público del servicio.

2.3.6. Contaminación de parámetros HTTP (*HTTP Parameter Pollution*)

En `HTTP Parameter Pollution` (HTTP Parameter Pollution (HPP)), el atacante explota diferencias de interpretación cuando un mismo parámetro aparece más de una vez o con codificaciones atípicas. La vulnerabilidad emerge cuando componentes distintos de la cadena (WAF, proxy, framework, aplicación) no coinciden sobre qué valor conservar, descartar o priorizar [31], [32].

```
GET /transfer?to=juan&monto=100&rol=user&rol=admin HTTP/1.1
Host: app.example
```

Listado 2.8: Ejemplo de *HTTP Parameter Pollution*: el parámetro `rol` aparece duplicado con valores contradictorios

En el Listado 2.8, el control de seguridad evalúa un valor distinto del que finalmente consume la lógica de negocio, por lo que la validación queda desacoplada de la ejecución real. Ese desacople habilita bypass de filtros y controles de autorización; además, combinaciones de separadores inusuales, parámetros vacíos o cadenas extensas se han asociado en la práctica a comportamientos no intencionales, incluyendo denegación de servicio y `cache poisoning` [33], [34]. Estas técnicas se relacionan con los riesgos del OWASP Top 10 A03:2021 (Injection), A01:2021 (Broken Access Control) y A05:2021 (Security Misconfiguration), según si el efecto principal recae en inyección de lógica, bypass de autorización o inconsistencias de configuración entre capas [1].

2.3.7. Ambigüedades de normalización Unicode e IDN

Unicode es el estándar que asigna a cada carácter un identificador numérico único (llamados puntos de código), de modo que distintos sistemas represen-

ten el mismo texto de forma consistente aunque usen alfabetos diferentes. Por otro lado, los Internationalized Domain Name (IDN) extienden los nombres de dominio para admitir caracteres no ASCII (por ejemplo, acentuados, griegos o cirílicos), que luego se codifican para su resolución DNS.

Sobre esa base, esta familia explota un desacople entre percepción humana y representación técnica: dos cadenas pueden verse casi idénticas, pero estar compuestas por identificadores Unicode distintos. En IDN, ese desacople permite registrar dominios visualmente similares a los legítimos y desviar tráfico hacia infraestructura controlada por el atacante [35].

```
GET /login HTTP/1.1
Host:   l.com
```

Listado 2.9: Ejemplo de ataque homógrafo IDN: caracteres cirílicos visualmente idénticos a latinos sustituyen letras en el dominio

En el Listado 2.9, el dominio `l.com` sustituye cinco caracteres latinos por sus homoglifos cirílicos (U+0440, U+0430 y U+0443), de modo que visualmente resulta indistinguible de `paypal.com`. Para una persona la URL puede parecer legítima, pero el navegador y DNS la tratan como un dominio diferente. El resultado es que el usuario entrega credenciales o tokens a un sitio controlado por el atacante, dado que la cadena es visualmente casi indistinguible. Estudios posteriores extienden esta lógica a otros identificadores (rutas, nombres de usuario) y proponen contramedidas basadas en similitud visual, normalización y análisis activo [36].

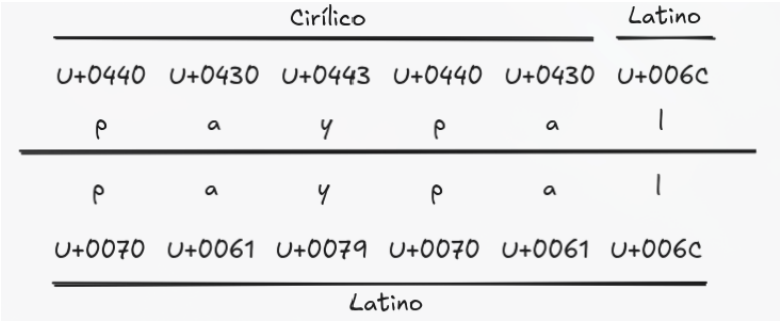


Figura 2.4: Comparación puntos de código Unicode con sus visualizaciones cirílica y latina

En el contexto HTTP, documentación sobre evasión de WAF reporta uso de homoglifos y mezcla de *scripts* en cabeceras y rutas para evadir reglas basadas en coincidencia textual [37]. Esto ha impulsado recomendaciones de normaliza-

ción Unicode, listas de caracteres permitidos y restricciones a *scripts* mixtos en campos sensibles [36].

2.3.8. Ambigüedades de parsing por espacios en blanco y `obs-fold`

Esta familia surge cuando un atacante aprovecha diferencias de *parsing* en torno a espacios en blanco y cabeceras plegadas (cabeceras expresadas en varias líneas, donde la continuación comienza con espacio o tabulación) para que cada componente del *stack* interprete algo distinto. En RFC 7230 existía la regla `obs-fold`, por la cual una línea que comenzara con espacio o tabulación se trataba como extensión de la cabecera anterior; en la especificación moderna (RFC 9112), ese comportamiento quedó desaconsejado y se exige normalización estricta o rechazo del mensaje.

```
GET /panel HTTP/1.1
Host: app.example
X-Note: ok
 X-Admin: true
```

Listado 2.10: Ejemplo de ambigüedad por `obs-fold`: la línea plegada puede interpretarse como continuación de `X-Note` o como una cabecera nueva `X-Admin`

En el Listado 2.10, un componente estricto puede desplegar la línea plegada y tratarla como valor concatenado de `X-Note` (`ok X-Admin: true`), mientras que una implementación más laxa puede recortar el espacio inicial y parsearla como una cabecera nueva `X-Admin: true`. Si el primer salto inspecciona la solicitud bajo la primera interpretación, pero el backend aplica lógica sensible bajo la segunda, se produce un `bypass` de controles. Documentos recientes enfatizan precisamente la necesidad de restringir estas variantes para minimizar interpretaciones divergentes de un mismo flujo de bytes [38]. La práctica confirma que no se trata de un detalle menor: múltiples implementaciones han corregido vulnerabilidades asociadas a manejo laxo de `obs-fold`, espacios inusuales alrededor de dos puntos o secuencias inesperadas de `CRLF` [39].

2.4. Fundamentos de aprendizaje automático para la detección de anomalías

La detección de anomalías es una de las aplicaciones clásicas del aprendizaje automático en seguridad: se trata de identificar muestras que se desvían de forma significativa del comportamiento considerado normal [40]. En el contexto de tráfico HTTP y WAF, esto se traduce en aprender un modelo del tráfico legítimo y utilizar dicho modelo para asignar una puntuación de rareza a cada

nueva solicitud. Esta perspectiva es compatible con escenarios donde apenas se dispone de ejemplos de ataque y el entrenamiento debe realizarse, en la práctica, con datos de una sola clase (tráfico benigno), posponiendo el uso de trazas maliciosas a la fase de evaluación.

Esta sección resume los conceptos básicos de aprendizaje automático que sustentan este enfoque: el paradigma de detección de anomalías de tipo *one-class*, los modelos basados en densidad local y vecindad, y los modelos estadísticos globales basados en distancias de Mahalanobis.

2.4.1. Detección de anomalías y paradigma *one-class*

Desde una perspectiva general, los algoritmos de detección de anomalías pueden agruparse en tres familias principales [40]: (i) métodos *supervisados*, que aprenden un clasificador binario a partir de ejemplos etiquetados como normales y maliciosos; (ii) métodos *semi-supervisados*, donde el modelo se ajusta sobre ejemplos normales y se utilizan unas pocas muestras etiquetadas de ataque para calibrar umbrales o refinar la frontera de decisión; y (iii) métodos *no supervisados*, que trabajan sin etiquetas y suponen que las anomalías son relativamente escasas y se manifiestan como valores atípicos respecto de la mayoría de los datos.

En entornos operativos como el que concierne a esta investigación, la obtención de conjuntos equilibrados de ataques representativos es costosa y queda rápidamente desactualizada frente a la evolución de las amenazas [41]. Por ello, gran parte de la literatura en detección de intrusiones HTTP adopta variantes del enfoque *one-class* [40]: el modelo se entrena usando solo tráfico considerado benigno y trata de estimar la región de alta densidad de la distribución normal. Las muestras que quedan fuera de esta región, según algún criterio geométrico o de densidad, se marcan como potencialmente anómalas.

Este paradigma incluye tanto métodos *kernel* (como las SVM de una clase [42]), modelos basados en aislamiento (como *Isolation Forest* [43]), técnicas de vecindad y modelos generativos más recientes (autoencoders variacionales [44], etc.).

2.4.2. Modelos basados en densidad local y vecindad

Una de las familias clásicas para detección de anomalías son los modelos basados en vecindad y densidad local. En lugar de suponer una forma global para la distribución de los datos, estos métodos comparan la densidad de cada muestra con la de su vecindario más cercano. Si la densidad local en torno a un punto es significativamente menor que la de sus vecinos, el punto se considera

un posible *outlier*.

Un representante canónico de esta familia es *Local Outlier Factor* (LOF) [45]. LOF define, para cada muestra, una *densidad de alcanzabilidad local* calculada a partir de las distancias a sus k vecinos más cercanos. El *factor de rareza* se obtiene como el cociente entre la densidad media de los vecinos y la densidad local de la muestra: valores cercanos a 1 indican que el punto reside en una región con densidad similar a la de su entorno, mientras que valores muy superiores a 1 apuntan a regiones de baja densidad local, típicas de anomalías. Esta formulación permite capturar distribuciones multimodales, es decir, configuraciones donde coexisten múltiples agrupaciones de datos normales con densidades distintas, sin necesidad de imponer una forma paramétrica global.

En el contexto de detección de anomalías, los modelos basados en vecindad resultan atractivos por varias razones: (i) no requieren una parametrización explícita de la distribución normal, (ii) se adaptan bien a estructuras no lineales y (iii) admiten entrenamiento puramente *one-class*. Sin embargo, su coste computacional escala con el tamaño del conjunto de entrenamiento y requieren fijar hiperparámetros como k (el número de vecinos considerados), lo que hace que su uso sea más natural sobre espacios de representación previamente reducidos en dimensionalidad.

2.4.3. Modelos estadísticos globales y distancias de Mahalanobis

Frente a los modelos de densidad local, otra familia de técnicas adopta una perspectiva global: suponen que las representaciones de las muestras normales se agrupan en torno a una región aproximadamente elíptica en el espacio de características y utilizan distancias ponderadas para medir la desviación de cada punto respecto de ese núcleo. La herramienta clásica en este contexto es la distancia de Mahalanobis [46], definida como

$$d_M(x) = \sqrt{(x-\mu)^\top \Sigma^{-1} (x-\mu)},$$

donde μ es la media de los datos normales y Σ su matriz de covarianza.

Esta medida tiene dos ventajas clave respecto de la distancia euclídea: (i) tiene en cuenta la varianza de cada dimensión, penalizando menos las desviaciones en direcciones de alta variabilidad y más en direcciones raras, y (ii) incorpora las correlaciones entre características, de modo que combinaciones poco frecuentes de atributos pueden producir distancias altas aunque las desviaciones individuales sean moderadas. Trabajos recientes han demostrado la eficacia de este enfoque para la detección de muestras fuera de distribución (*out-of-distribution*, OOD) y ataques adversarios en espacios de representaciones profundas [47].

En el caso de representaciones semánticas densas, como los *embeddings* generados por modelos tipo BERT, la hipótesis de una región normal aproximadamente elíptica resulta razonable: las muestras que realizan operaciones similares tienden a concentrarse en una subregión relativamente compacta del espacio de *embeddings*, y la distancia de Mahalanobis permite detectar solicitudes que se apartan de esa región de forma más expresiva que una simple distancia euclídea.

2.5. Detección de tráfico HTTP malicioso mediante aprendizaje automático

El uso de técnicas de aprendizaje automático para detectar tráfico HTTP malicioso se ha consolidado como una línea de trabajo específica dentro de la seguridad de aplicaciones web. En esta sección se presenta el estado del arte organizando la literatura en cinco ejes: clasificación de peticiones con aprendizaje profundo, detección no supervisada, modelos de lenguaje de dominio aplicados a HTTP, el ecosistema de WAFs modernos junto con las técnicas de evasión que los desafían, y los trabajos previos del grupo de investigación en detección de anomalías.

2.5.1. Clasificación de peticiones HTTP con aprendizaje profundo

Una parte significativa de la literatura aborda la detección de tráfico malicioso como un problema de clasificación supervisada a nivel de petición HTTP. En estos enfoques, cada solicitud se representa mediante un conjunto de características derivadas de su contenido y metadatos, y se entrena un modelo para distinguir entre clases (*benigno* versus *malicioso*).

DeepHTTP [48] es un referente temprano en el uso de aprendizaje profundo para este tipo de tareas. El trabajo propone un modelo que combina representaciones de bajo nivel de las peticiones (caracteres, tokens y secuencias) con componentes recurrentes y de atención para capturar patrones complejos tanto en URLs y parámetros como en cuerpos y cabeceras. Los autores muestran que esta aproximación supera a clasificadores tradicionales basados en características diseñadas manualmente en escenarios de detección de ataques web y anomalías en tráfico HTTP.

De forma más amplia, la propuesta de DeepHTTP refuerza una intuición recurrente en la literatura: la detección eficaz de anomalías en tráfico web requiere explotar simultáneamente patrones sintácticos (estructuras de *path*, combinaciones de cabeceras, codificaciones) y patrones semánticos (intención de la operación, entidad de negocio afectada, etc.). Esta dualidad sintáctico–semántica

sugiere que los enfoques más efectivos son aquellos que combinan vistas complementarias de cada solicitud.

2.5.2. Enfoques no supervisados para la detección de anomalías en tráfico web

Cuando la disponibilidad de muestras de ataque es limitada o sesgada, la detección se formula como un problema no supervisado o *one-class*: el modelo aprende el perfil del tráfico legítimo y marca como anómalas las peticiones que se apartan de ese perfil. Los fundamentos teóricos de este paradigma se discutieron en la Sección 2.4; esta subsección revisa su aplicación concreta al tráfico HTTP. El enfoque resulta especialmente relevante en seguridad web, donde los ataques cambian con rapidez y la cobertura de etiquetas suele quedar desactualizada [41].

Kruegel y Vigna introdujeron uno de los primeros sistemas de detección de anomalías centrado en ataques web [49]. Su enfoque aprende perfiles de parámetros para cada recurso (longitud, estructura, caracteres permitidos) y utiliza técnicas estadísticas para identificar desviaciones en los parámetros de las *queries*. Aunque el modelo trabaja sobre características diseñadas manualmente (*hand-crafted features*), anticipa varias ideas que siguen vigentes: la importancia de modelar cada punto de entrada por separado, el uso de distribuciones paramétricas sobre atributos de cabeceras y la correlación entre parámetros y recursos.

En una línea complementaria, Wang y Stolfo propusieron PAYL [50], un sistema que modela la distribución de bytes en los *payloads* de las peticiones para detectar anomalías a nivel de contenido. El modelo construye un perfil estadístico para cada par (puerto, longitud de *payload*) utilizando distribuciones de frecuencia de bytes, y clasifica como anómala toda petición cuyo perfil se desvíe significativamente de la referencia aprendida. Tanto este trabajo como el de Kruegel y Vigna comparten un supuesto central de la detección no supervisada en tráfico web: el tráfico benigno presenta regularidades estadísticas estables que pueden capturarse mediante modelos relativamente simples, sin necesidad de catalogar patrones de ataque.

Más recientemente, la familia de métodos basados en reconstrucción ha ganado prominencia en detección de intrusiones. En este paradigma, un autoencoder (o su variante probabilística, el autoencoder variacional descrito en la Sección 2.4.1) se entrena para reconstruir muestras normales; en inferencia, las muestras que generan un error de reconstrucción elevado se consideran anómalas. Mirsky et al. [51] aplican esta idea al tráfico de red con Kitsune, un *ensemble* de autoencoders ligeros diseñado para operar en línea. El sistema descompone cada paquete en subconjuntos de características y entrena un autoencoder independiente sobre cada subconjunto, combinando luego los errores de reconstruc-

ción para producir una puntuación de anomalía global. Aunque Kitsune opera a nivel de flujos de red y no específicamente sobre solicitudes HTTP, su diseño muestra que los métodos de reconstrucción pueden alcanzar tasas de detección competitivas en escenarios puramente *one-class* y con restricciones de latencia.

En el ámbito específico de WAF, se han aplicado algoritmos como One-Class SVM e Isolation Forest para modelar comportamiento normal de solicitudes HTTP y reducir falsos positivos [52]. Un patrón recurrente consiste en combinar una primera etapa de reglas (por ejemplo, CRS) con una segunda etapa de anomalía que reevalúa los casos sospechosos y prioriza aquellos que se desvían de forma estadísticamente significativa del tráfico habitual. Este esquema híbrido conecta directamente con los métodos de densidad local (Sección 2.4.2) y distancia estadística (Sección 2.4.3) discutidos en el capítulo de fundamentos.

Pese a sus ventajas, los enfoques no supervisados enfrentan limitaciones conocidas en entornos operativos. En primer lugar, la eficacia del modelo depende de la calidad de la línea base: si el conjunto de entrenamiento contiene tráfico malicioso no identificado, el modelo puede incorporar esos patrones como normales (*baseline poisoning*). En segundo lugar, el tráfico web evoluciona continuamente por actualizaciones de aplicaciones, cambios en las APIs, entre otros, lo que requiere mecanismos de reentrenamiento o adaptación periódica [41]. En tercer lugar, la literatura reporta de forma recurrente tasas elevadas de falsos positivos, particularmente cuando el modelo opera sobre tráfico heterogéneo proveniente de múltiples aplicaciones o servicios. Estas limitaciones motivan el interés por representaciones más expresivas que capturen regularidades de mayor nivel, como las que ofrecen los modelos de lenguaje discutidos en la subsección siguiente.

2.5.3. Modelos de lenguaje de dominio para ciberseguridad y su aplicación a HTTP

El auge de los modelos de lenguaje basados en la arquitectura Transformer [53] ha cambiado la forma de abordar la representación de datos en ciberseguridad. En lugar de diseñar manualmente características sobre cabeceras y parámetros, estos modelos se apoyan en *embeddings* distribuidos que capturan regularidades de alto nivel en los datos. En el dominio específico de ciberseguridad han surgido variantes adaptadas, entrenadas sobre grandes corpus de informes técnicos, *logs* y datos de inteligencia de amenazas. SecBERT [54], por ejemplo, ajusta una arquitectura tipo BERT [55] a texto de ciberseguridad [56], logrando mejoras respecto de BERT genérico en tareas como clasificación de informes, extracción de indicadores de compromiso o análisis de alertas. La ventaja clave de estos modelos de dominio reside en que su vocabulario y representaciones internas ya reflejan terminología y patrones propios del área, reduciendo la brecha entre el preentrenamiento y la tarea objetivo.

La combinación de estas representaciones profundas con técnicas de detección de anomalías clásicas ha dado lugar a un patrón recurrente en la literatura reciente: (i) utilizar un modelo de lenguaje para proyectar cada evento de seguridad (log, solicitud HTTP, informe) en un espacio denso de baja dimensión, y (ii) aplicar sobre ese espacio detectores *one-class* basados en densidad local o en estadística multivariante, como LOF o distancias de Mahalanobis [47]. Este esquema separa el problema de representación (delegado en el modelo profundo) del problema de detección (delegado en un modelo de rareza relativamente simple e interpretable).

En tráfico HTTP, este enfoque permite tratar las solicitudes no como vectores de características predefinidas, sino como secuencias de texto con estructura interna rica. Los modelos pueden así capturar dependencias entre componentes de la solicitud (por ejemplo, entre el método, la ruta y combinaciones específicas de cabeceras) que enfoques basados en n-gramas o en bolsas de palabras no representan adecuadamente.

2.5.4. WAFs modernos, evasión de WAF y aprendizaje automático

Los WAF constituyen la principal línea de defensa perimetral contra ataques a aplicaciones web. Tradicionalmente basados en reglas estáticas, los WAFs han evolucionado hacia arquitecturas que incorporan componentes de aprendizaje automático con distintos niveles de integración. Esta subsección revisa las propuestas defensivas, las técnicas de evasión que las desafían y el estado de adopción en la industria.

En el plano defensivo, propuestas recientes exploran esquemas híbridos donde el conocimiento codificado en reglas se reutiliza como entrada para modelos de Machine Learning (ML). ModSec-Learn [57] utiliza activaciones de reglas de ModSecurity/CRS como características para aprender una frontera de decisión más precisa que la suma heurística de severidades, con el objetivo de conservar cobertura y reducir falsos positivos. Sobre esa base, ModSec-AdvLearn [58] incorpora entrenamiento adversarial para robustecer el clasificador frente a variantes evasivas de SQLi. En la misma dirección, Betarte et al. [59], [60] proponen combinar clasificadores entrenados sobre características de cabeceras y cuerpo con reglas preexistentes de WAF, y Montes et al. [61] evalúan arquitecturas de redes profundas para detectar ataques web. En conjunto, esta línea de trabajo muestra que la capa de ML no reemplaza al WAF clásico, sino que lo calibra y endurece frente a ataques que explotan la rigidez de reglas estáticas.

En paralelo a las mejoras defensivas, la literatura ofensiva documenta métodos sistemáticos para evadir WAFs. WAF-A-MoLE [62] mostró que es posible generar *payloads* semánticamente equivalentes mediante mutaciones adversariales hasta encontrar variantes que el WAF clasifica como benignas. AutoSpear [63]

continúa esa línea con generación automática de SQLi guiada por gramáticas y búsqueda heurística, orientada a escenarios de evaluación en caja negra. WAFFLED [15] extiende el problema al nivel de *parsing* HTTP: en lugar de ofuscar el *payload*, muta la estructura de la solicitud para inducir interpretaciones distintas entre WAF y *backend*. Estas investigaciones ofensivas resultan valiosas desde una perspectiva defensiva: exponen debilidades que los WAFs continúan presentando y permiten identificar qué patrones estructurales resulta prioritario detectar.

Los WAF comerciales han incorporado ML de manera heterogénea. Cloudflare [64], [65] reporta un esquema de *attack scoring* en línea, entrenado de forma supervisada, que asigna una puntuación de riesgo por solicitud para ser consumida por reglas de bloqueo definidas por política. AWS documenta módulos específicos con ML para detección de bots y fraude, mientras mantiene la lógica central del WAF apoyada en reglas administradas [66]. Akamai [67] describe arquitecturas multicapa que combinan señales de contenido y contexto, mientras que Imperva [68] reporta capacidades de analítica de ataques y correlación posterior a la detección en línea. En conjunto, estos despliegues muestran que la industria converge hacia arquitecturas híbridas: reglas para cobertura determinística de amenazas conocidas y ML para priorización de riesgo, detección de patrones emergentes y ajuste operativo.

2.5.5. Trabajos previos del grupo de investigación

Trabajos previos del grupo de investigación al que se adscribe este proyecto han explorado la detección de anomalías en tráfico HTTP de aplicaciones productivas. La mayor parte de esta línea se enmarca en técnicas de detección de anomalías con aprendizaje automático clásico. Betarte et al. [69] analizan distintas familias de clasificadores (árboles de decisión, *random forests*, SVM y redes neuronales) sobre conjuntos de características extraídas de logs y peticiones HTTP, incluyendo información de cabeceras, parámetros y patrones de acceso, para detectar peticiones maliciosas en aplicaciones web. Martínez [70] extiende esta línea, evaluando modelos supervisados sobre trazas de tráfico reales y comparando el impacto de diferentes estrategias de ingeniería de atributos en la capacidad de detección. Montes de Marco [71] es el único antecedente del grupo que incorpora modelos de aprendizaje profundo, experimentando con representaciones secuenciales de las peticiones para mejorar la detección de ataques y anomalías en tráfico HTTP.

Capítulo 3

Análisis

Este capítulo analiza la estructura de las solicitudes HTTP/1.1 con el objetivo de identificar dónde y cómo se manifiestan las anomalías explotadas por las familias de ataques descritas en el Capítulo 2. El análisis se sustenta en las especificaciones del protocolo (RFC 9110 y 9112) y en la evidencia empírica recogida en la literatura sobre técnicas de evasión y vulnerabilidades estructurales. En primer lugar, se examina la dualidad sintáctico-semántica de las peticiones HTTP, es decir, la relación (y posible discrepancia) entre la forma textual de una solicitud y su interpretación efectiva. A continuación, se realiza un análisis exploratorio de anomalías estructurales que permite abstraer los ataques conocidos en familias de desviaciones respecto de una línea base de solicitudes bien formadas. Los resultados de este análisis constituyen la base conceptual sobre la cual se construye el enfoque de preprocesamiento y detección presentado en capítulos posteriores.

3.1. Dualidad sintáctico–semántica en las solicitudes HTTP

En el contexto de este trabajo, la dimensión sintáctica de una solicitud HTTP refiere a su forma textual a nivel de línea de petición, cabeceras, delimitadores y codificación, mientras que la dimensión semántica refiere al efecto operativo que esa solicitud produce al ser procesada (por ejemplo, decisiones de ruteo, autenticación, autorización o creación de lineas de caché). A partir de los trabajos revisados en el Capítulo 2, esta sección analiza la relación entre ambas dimensiones y, en particular, los casos en que su correspondencia deja de ser unívoca. A esa discrepancia entre forma e interpretación la denominamos, en este documento, dualidad sintáctico–semántica. Sobre esa base, se discuten los patrones de desalineación más recurrentes en la literatura y el rol específico de las cabeceras HTTP como punto de concentración de dichas desalineaciones.

3.1.1. Patrones de dualidad sintactico–semántico observados en la literatura

Los trabajos revisados en la Sección 2.3 muestran que las vulnerabilidades asociadas al procesamiento de peticiones HTTP pueden concentrarse en situaciones donde la relación entre la forma de la solicitud y su interpretación efectiva deja de ser unívoca. En otras palabras, cómo pequeñas variaciones en la sintaxis pueden corresponder a significados semánticos o interpretaciones muy distintas. Considerados en conjunto, estos resultados permiten identificar una serie de patrones recurrentes de desalineación entre la dimensión sintáctica y la dimensión semántica de las peticiones. En esta subsección se discuten dos de estos patrones como un marco conceptual para el resto del análisis.

Zonas grises y flexibilidad de las especificaciones. Un primer patrón está asociado a la existencia de zonas grises en las especificaciones y en las implementaciones. La flexibilidad del protocolo abre zonas donde la sintaxis no determina una semántica única (ver Sección 2.1). En consecuencia, formatos marginales o poco habituales, aun siendo sintácticamente aceptables, constituyen una señal potencial de desalineación.

Complejidad acumulada en los mecanismos de normalización. Un segundo patrón tiene que ver con la *normalización* que distintos componentes aplican sobre las solicitudes en tránsito. Las reescrituras y *canonicalizaciones* pueden alterar la semántica efectiva de una solicitud válida (ver Sección 2.3.8). Por ello, este patrón sugiere privilegiar señales robustas frente a transformaciones superficiales (p. ej., espacios, duplicados, reordenamientos).

En conjunto, estos factores explican cómo pequeñas variaciones sintácticas pueden traducirse en interpretaciones semánticas distintas según el componente que procese la solicitud, delimitando un espacio de ellas donde coexisten múlti-

ples lecturas posibles para el mismo paquete.

3.1.2. Vacíos relevantes encontrados

A partir de lo discutido en el Capítulo 2 (la separación entre semántica y sintaxis del protocolo y las discrepancias prácticas de *parsing*) es posible identificar algunos vacíos relevantes para este trabajo.

Desde la perspectiva de este trabajo, lo discutido anteriormente deja al descubierto algunos vacíos relevantes. En primer lugar, los enfoques de detección pueden clasificarse según la dimensión que priorizan. Algunos se centran en la forma estructural de las solicitudes sin modelar su significado funcional [69], [72]. Otros construyen representaciones contextuales profundas que capturan el contenido y su significado, ignorando en gran medida las propiedades estructurales del mensaje [71]. Trabajos como DeepHTTP [48] han reconocido la utilidad de combinar ambas perspectivas, y proponen arquitecturas que las integran simultáneamente, obteniendo mejoras en la capacidad de generalización. Sin embargo, estos enfoques son supervisados: requieren ejemplos etiquetados de tráfico malicioso para el entrenamiento. En el escenario *one-class*, donde solo se dispone de tráfico legítimo, no se identifica un enfoque que combine sistemáticamente ambas dimensiones de representación.

En segundo lugar, la literatura aparece fragmentada en dos líneas que avanzan en paralelo. Por un lado, las investigaciones orientadas al ataque y a la evasión exploran solicitudes de frontera y muestran casos en los que distintos componentes (por ejemplo, intermediarios y *backend*) asignan interpretaciones distintas al mismo flujo de bytes (Sección 2.3). Por otro lado, las investigaciones orientadas a la defensa se centran en modelar tráfico real para discriminar entre solicitudes legítimas y anómalas (Sección 2.5). Sin embargo, no se identifican trabajos que relacionen explícitamente ambos enfoques, es decir, que conviertan los hallazgos ofensivos (variantes de *parsing*, vectores de evasión) en señales o transformaciones reproducibles para un detector.

3.2. Análisis exploratorio de anomalías estructurales en solicitudes HTTP

En esta sección se realiza un análisis conceptual de vulnerabilidades basadas en solicitudes HTTP/1.1, con foco en cómo la intención maliciosa se manifiesta en la *forma* de la petición. El punto de partida lo constituyen las especificaciones del protocolo y su separación entre sintaxis y semántica [2], [3], las técnicas de evasión de WAF, y las familias de ataques descritas en el Capítulo 2.

En lugar de organizar el análisis por familia de vulnerabilidades (inyección, *smuggling*, SSRF, etc.), se propone abstraer estos ejemplos en términos de desviaciones respecto de una línea base de solicitudes bien formadas. Los objetivos son identificar qué patrones de anomalía estructural se repiten de manera consistente, ya sea en ataques conceptualmente similares o distintos, y cómo pueden convertirse en señales explícitas para mecanismos de detección.

3.2.1. Objetivo y metodología del análisis estructural

El objetivo de este análisis es entender de qué manera la intención maliciosa se manifiesta en la *forma* de las peticiones HTTP, independientemente de la familia de vulnerabilidad a la que se asocie cada caso. En particular, se buscan patrones recurrentes en la estructura de la solicitud: cómo se delimitan los componentes del mensaje, cómo se codifican y combinan los campos de cabecera, cómo se estructura la ruta y la cadena de consulta, y en qué medida ligeras variaciones sintácticas alteran el comportamiento de los componentes que procesan la solicitud.

Metodológicamente, el análisis toma como "corpus conceptual" el conjunto de vulnerabilidades discutidas en el Capítulo 2. A partir de esta codificación se agrupan los casos en familias de anomalías. El alcance del análisis se restringe a peticiones HTTP/1.1 y abarca tanto las cabeceras como la línea de petición (*request line*), incluyendo método, ruta y cadena de consulta, dejando fuera tanto las particularidades de HTTP/2 y HTTP/3 como el cuerpo del mensaje. En ese sentido, los resultados deben entenderse como un marco conceptual y no como una caracterización exhaustiva del tráfico HTTP.

3.2.2. Línea base: solicitudes HTTP/1.1 bien formadas

Como línea base para el análisis se adopta la caracterización de HTTP/1.1 presentada en la Sección 2.1. A partir de esa descripción normativa, este trabajo utiliza una noción de solicitud "bien formada" que no pretende abarcar todas las variantes admitidas por el protocolo, sino capturar el perfil de tráfico que se espera observar con mayor frecuencia en peticiones legítimas. Esta expectativa se fundamenta en las restricciones de conformidad que las propias especificaciones

del protocolo [2], [3] imponen a las implementaciones: los requisitos expresados como MUST y SHOULD en los RFC delimitan un subconjunto de representaciones que los clientes conformes están obligados o incentivados a emitir, lo que en la práctica las convierte en las formas predominantes del tráfico legítimo.

Consideramos como línea base aquellas solicitudes que cumplen simultáneamente que: (i) la línea de petición contiene un método estándar, un *request target* con una estructura de ruta y parámetros convencional y una versión de protocolo válida; (ii) la sección de cabeceras utiliza nombres de campo válidos y coherentes con los registros estándar, no recurre a mecanismos obsoletos como el *line folding* ni introduce caracteres de control incrustados en los valores; (iii) se respeta la unicidad de cabeceras críticas como `Host` y `Content-Length`, evitando combinaciones incompatibles entre conjuntos de cabeceras; (iv) el número y tamaño de cabeceras es acotado, con valores mayoritariamente legibles y sin codificaciones exóticas; y (v) ruta y cadena de consulta se expresan en ASCII, con codificaciones *percent-encoded* estándar y sin patrones de ofuscación.

En las subsecciones siguientes se considera como *anomalía estructural* cualquier desviación relevante respecto de estas propiedades, y se muestra cómo estas desviaciones aparecen de forma recurrente en los ataques descritos en el Capítulo 2.

3.2.3. Familias de anomalías estructurales en solicitudes HTTP

El recorrido por las vulnerabilidades de la Sección 2.3 sugiere que muchas fallas de seguridad no dependen tanto del contenido semántico de la petición como de la forma concreta en que la solicitud se representa a nivel de bytes y líneas. A partir del análisis de las familias de ataques conocidas en la literatura y de las discrepancias de *parsing* documentadas en dichas fuentes, esta sección propone una taxonomía de *anomalías estructurales* en solicitudes HTTP/1.1, ilustrada con ejemplos representativos.

El criterio de agrupación es estrictamente la *manifestación observable en el paquete*: cómo se ve la anomalía en el flujo de bytes, independientemente de la familia de vulnerabilidad a la que pertenezca, del impacto que produzca o de la técnica de explotación involucrada. Así, dos ataques conceptualmente distintos que se manifiestan de la misma forma en la solicitud pertenecen a la misma familia, y dos variantes de un mismo ataque que se manifiestan de forma distinta pertenecen a familias separadas. Las cinco familias resultantes son: (i) duplicación, conflicto o ambigüedad de selección entre campos, (ii) inyección o presencia anómala de delimitadores estructurales, (iii) codificación, normalización y repertorio de caracteres, (iv) estructura anómala del *request target*, y (v) valores extremos o perfil estadístico implausible.

En cada familia se incluye un ejemplo anotado de solicitud HTTP/1.1 y se enumeran señales estructurales observables asociadas. La formalización de estas señales como criterios computables (*flags*) se presenta en el capítulo de diseño (Secciones 4.1 y 4.2).

Duplicación, conflicto y ambigüedad de selección

Vulnerabilidades tan diversas como *request smuggling*, *host header injection*, suplantación de origen y HPP (Sección 2.3) convergen en una misma manifestación observable: el mensaje ofrece más de una fuente de información para una misma decisión y no hay unicidad. Ya sea mediante cabeceras incompatibles (`Content-Length` junto a `Transfer-Encoding`), cabeceras críticas duplicadas con valores contradictorios (`Host`, `Authorization`), múltiples cabeceras alternativas para el mismo dato (`X-Forwarded-For`, `Client-IP`, `True-Client-IP`) o parámetros repetidos en la cadena de consulta (`a=1&a=2`), el patrón observable es el mismo: el paquete contiene información redundante o contradictoria donde debería existir un único valor autoritativo.

Ejemplo (múltiples fuentes contradictorias para delimitación, identidad y origen).

```
POST /transfer?to=juan&monto=100&rol=user&rol=admin HTTP/1.1
host: victim.example
Host: attacker.example
Content-Length: 50
Transfer-Encoding: chunked
X-Forwarded-For: 127.0.0.1
Client-IP: 127.0.0.1
Authorization: Bearer eyJhbGciOiJIUzI1NiJ9...
authorization: Basic dXNlcjpwYXNz
```

Listado 3.1: Múltiples fuentes contradictorias para delimitación, identidad y origen

La solicitud acumula varias instancias de duplicación y conflicto: el parámetro `rol` aparece dos veces con valores contradictorios (`user` vs. `admin`); `Host` está duplicado con capitalización inconsistente y valores distintos; `Content-Length` y `Transfer-Encoding` coexisten, generando ambigüedad de delimitación; dos cabeceras alternativas de origen (`X-Forwarded-For` y `Client-IP`) transportan la misma dirección *loopback*; y `Authorization` aparece dos veces con esquemas incompatibles (`Bearer` vs. `Basic`).

Señales observables.

- coexistencia de `Content-Length` y `Transfer-Encoding`;

- duplicación de cabeceras que en tráfico legítimo suelen ser únicas (`Host`, `Content-Length`, `Authorization`);
- presencia simultánea de múltiples cabeceras alternativas para el mismo dato (`X-Forwarded-For`, `Forwarded`, `Client-IP`, `True-Client-IP`);
- parámetros duplicados en la cadena de consulta (múltiples instancias del mismo nombre con valores distintos).

Lo que unifica a esta familia es que la anomalía se manifiesta como *multiplicidad* donde debería haber unicidad: el paquete presenta más de un candidato para una misma decisión de procesamiento. Desde la perspectiva de detección, estas señales pueden identificarse sin interpretar el significado de los valores; basta con verificar la unicidad de campos y parámetros críticos y detectar combinaciones mutuamente excluyentes. La ambigüedad resultante introduce grados de libertad que un atacante puede explotar para desalinear la interpretación de la petición a lo largo de la cadena de componentes.

Presencia anómala de delimitadores estructurales

Las vulnerabilidades de inyección CRLF, *HTTP response splitting* y las ambigüedades de *obs-fold* descritas en la Sección 2.3 comparten una manifestación distinta de la anterior: la anomalía no consiste en que haya múltiples fuentes para una misma decisión, sino en que *datos* que deberían ser contenido pasan a ser interpretados como *delimitadores* del protocolo. Ya sea mediante secuencias `CRLF` reales o introducidas tras decodificar `%0d%0a`, líneas que comienzan con espacio o tabulador (*obs-fold*), o separadores de cabecera mal ubicados, el efecto observable es siempre el mismo: los límites entre componentes del mensaje se desplazan respecto de lo previsto por el emisor legítimo.

Ejemplo (CRLF embebido y continuación *obs-fold*).

```
1 GET /redir?next=/home%0d%0aSet-Cookie:%20rol=admin HTTP/1.1
2 Host: app.example
3 X-Note: ok
4  X-Admin: true
```

Listado 3.2: CRLF embebido y continuación *obs-fold*

La cadena de consulta incluye una secuencia `%0d%0a` que, si la aplicación la decodifica antes de emitir la respuesta, inyecta una cabecera `Set-Cookie` no prevista. Además, la cuarta línea comienza con un espacio: un componente estricto la trata como continuación del valor de `X-Note` (`ok X-Admin: true`), mientras que una implementación laxa puede recortar el espacio y parsearla como una cabecera independiente `X-Admin: true`.

Señales observables.

- presencia de bytes de control inesperados en valores de cabeceras o en la cadena de consulta (secuencias `\r\n` embebidas, `%0d%0a`);
- líneas que comienzan con espacio o tabulador en la sección de cabeceras (*obs-fold*);
- continuaciones huérfanas (sin cabecera previa a la que anexarse);
- nombres de campo con caracteres inválidos o separadores `:` mal ubicados.

Lo que unifica a esta familia es que la anomalía consiste en la *aparición de delimitadores estructurales donde deberían existir únicamente datos*. A diferencia de la familia anterior, donde el conflicto surge de la multiplicidad de campos legítimos, aquí el problema es que un fragmento de contenido adquiere función de separador, desplazando los límites del mensaje. Desde la perspectiva de detección, estos indicadores pueden identificarse inspeccionando la presencia de bytes de control, patrones de continuación de línea y anomalías de formato en la frontera entre nombres y valores de cabecera.

Codificación, normalización y repertorio de caracteres

Las técnicas de evasión mediante codificaciones dobles y las ambigüedades de normalización Unicode descritas en las Secciones 2.3 y 2.5.4 comparten un rasgo en común: la anomalía no reside en *qué* dice la solicitud, sino en *cómo lo representa*. Esta familia agrupa las desviaciones en la capa de codificación, un aspecto que atraviesa todos los componentes de la solicitud (ruta, cadena de consulta y valores de cabeceras) y que incluye tanto esquemas de *percent-encoding* redundantes como el uso de caracteres Unicode no habituales o la mezcla de sistemas de escritura. El patrón observable común es que el mismo contenido lógico se expresa mediante una representación textual o de bytes no canónica.

Ejemplo (doble codificación y Unicode fullwidth).

```
GET /%252e%252e/%EF%BC%8Fadmin HTTP/1.1
Host: xn--exmple-qta.com
```

Listado 3.3: Doble codificación y Unicode fullwidth

Decodificación: `%252e%252e` → `%2e%2e` → `..`; `%EF%BC%8F` → *fullwidth solidus* (U+FF0F). La ruta efectiva tras doble decodificación es `../../admin`. El campo `Host` utiliza la representación Punycode (`xn--`) de un nombre de dominio internacionalizado.

Señales observables.

- secuencias de doble *percent-encoding* (`%25XX`);
- presencia de caracteres *fullwidth* o de formato Unicode en ruta o consulta;
- mezcla de scripts (latino, cirílico, griego, etc.) dentro de un mismo token;
- entidades HTML en contextos de URL o cadena de consulta;
- presencia del prefijo `xn--` en `Host` (IDN en punycode);
- proporciones inusuales de caracteres no ASCII en campos donde se espera mayoritariamente ASCII.

Desde la perspectiva de un análisis estructural, estos casos pueden detectarse sin apelar al significado de las cadenas: basta con inspeccionar el repertorio de scripts utilizados, la proporción de caracteres no ASCII, la presencia de capas de codificación redundantes y las propiedades de normalización de los campos relevantes. Lo que unifica a esta familia es que las anomalías residen en *cómo se representa* el contenido, no en el contenido mismo.

Estructura anómala del *request target*

Los ataques de *path traversal* descritos en la Sección 2.3 se manifiestan en la estructura de la ruta y la cadena de consulta mediante patrones observables que no requieren interpretar el significado de cada recurso o parámetro. La desviación se concentra en la *forma* del *request target*: segmentos de navegación anómalos, barras consecutivas, segmentos de directorio actual y uso mixto de delimitadores en la *query string*.

Ejemplo (*traversal*, barras múltiples y delimitadores mixtos).

```
GET /app/../admin/.//config?debug;verbose&format=json HTTP/1.1
Host: example.com
```

Listado 3.4: *Traversal*, barras múltiples y delimitadores mixtos

La ruta contiene un segmento de *traversal* (`../`), un segmento de directorio actual (`.`), barras consecutivas (`//`); la cadena de consulta mezcla los delimitadores `;` y `&`, e incluye parámetros sin valor (`debug`, `verbose`).

Señales observables.

- patrones de *path traversal* (`../`) o segmentos de ruta atípicos (`./`);
- barras múltiples consecutivas (`//`) en la ruta;

- uso mixto de separadores en la cadena de consulta (`&`, `;`) o parámetros sin valor explícito.

Estructuralmente, esta familia se describe mediante rasgos independientes del recurso concreto al que se accede: la presencia de segmentos de navegación de directorio, barras redundantes y delimitadores no convencionales. Estos elementos afectan directamente la estructura del *request target* y, en combinación con políticas de *parsing* heterogéneas en proxies, *frameworks* y aplicaciones, pueden derivar en discrepancias sobre qué recurso se está solicitando efectivamente [33]. Para un mecanismo de detección, la relevancia de esta familia reside en que todos estos rasgos pueden caracterizarse sin necesidad de interpretar el significado semántico del recurso o los parámetros.

Valores atípicos y outliers cuantitativos

Una última familia agrupa anomalías que no se manifiestan como conflictos, delimitadores desplazados, codificaciones anómalas ni estructuras de ruta irregulares, sino como *valores atípicos* respecto del perfil habitual del tráfico. Se trata de desviaciones en magnitud, cantidad o distribución: campos cuyo contenido es sintácticamente válido y no ambiguo, pero cuyas propiedades cuantitativas se sitúan fuera del rango esperable.

Ejemplo (longitud extrema, entropía y cantidad de cabeceras).

```
POST /api/data HTTP/1.1
Host: example.com
Content-Length: 104857600
Connection: close, X-Real-IP
X-Custom-1: aaa...
X-Custom-2: bbb...
...
X-Custom-48: zzz...
Cookie: s=9f3a1b7c4e... (2048 caracteres de alta entropia)
```

Listado 3.5: Longitud extrema, entropía y cantidad de cabeceras

La solicitud presenta un `Content-Length` de ∼100 MB, muy por encima del perfil habitual; un campo `Connection` que declara `X-Real-IP` como cabecera *hop-by-hop*, forzando su descarte en el siguiente salto; 48 cabeceras `X-Custom-*`, un número inusualmente elevado; y un valor de `Cookie` de 2048 caracteres con alta entropía.

Señales observables.

- valores numéricos fuera de rango (`Content-Length` extremo);

- número de cabeceras o longitud total del bloque de cabeceras muy por encima del perfil benigno;
- patrones de alta entropía en campos donde se esperan valores acotados;
- cabeceras *hop-by-hop* inesperadas (campos declarados en `Connection` para forzar su descarte).

Lo que unifica a esta familia es que la anomalía no es un conflicto estructural ni un error de formato, sino un *outlier de forma*: el paquete se desvía del perfil estadístico habitual sin violar la gramática del protocolo. Desde la perspectiva de detección, estas señales requieren establecer una línea base cuantitativa del tráfico legítimo y marcar como sospechosos los valores que se sitúan en los extremos de la distribución.

Capítulo 4

Diseño del enfoque propuesto

En este capítulo se presenta el diseño del enfoque propuesto, al que denominamos *NeuralShield*. Se describen los requisitos de diseño, el *pipeline* de preprocesamiento, las vistas de representación sintáctica y semántica, y el módulo de detección de anomalías.

4.1. Visión general y requisitos de diseño

En este trabajo se propone estudiar qué decisiones de preprocesamiento, representación y modelado resultan efectivas para la detección de anomalías en solicitudes HTTP. Denominamos *NeuralShield* al marco experimental que utilizamos para responder esas preguntas: un *pipeline* completo que nos permite formular hipótesis, instanciar distintas combinaciones de técnicas y medir su impacto de forma controlada. Como se discutió en la Sección 3.1, el diseño se apoya explícitamente en la dualidad sintáctico–semántica de las peticiones HTTP y busca operacionalizarla a nivel de cabeceras.

Conceptualmente, NeuralShield se organiza en tres etapas. En primer lugar, un preprocesamiento estructural que toma mensajes HTTP crudos y los convierte en una representación canónica alineada con los RFC, exponiendo desviaciones de *framing*, codificación y estructura mediante *flags*. En segundo lugar, sobre esa representación se construyen dos vistas complementarias que materializan la dualidad analizada previamente: una vista sintáctica, que captura la forma y las regularidades léxicas de las solicitudes, y una vista semántica, que modela el contexto y el significado funcional del tráfico. Finalmente, ambas vistas alimentan detectores de anomalías no supervisados cuyas puntuaciones se combinan en una decisión única.

A lo largo de los capítulos siguientes se utiliza este marco para evaluar el

aporte de cada etapa y la complementariedad entre ambas vistas. Los vacíos identificados en la Sección 3.1.2, junto con las restricciones operativas del escenario *one-class*, se traducen en los siguientes requisitos para el enfoque propuesto:

- **Modelo dual explícito.** La arquitectura debe separar y articular claramente una vista sintáctica y una vista semántica de las solicitudes, de modo que la dualidad identificada en el análisis no sea sólo una intuición conceptual, sino un componente observable del diseño.
- **Exposición sistemática de anomalías estructurales.** El *pipeline* de preprocesamiento debe implementar una línea base de cabeceras bien formadas y una taxonomía de *flags* que cubra las familias de anomalías descritas en el análisis, permitiendo que el comportamiento descubierto por herramientas de *fuzzing* y estudios de parsing se incorpore como señal en los modelos de detección.
- **Equilibrio entre interpretabilidad y capacidad predictiva.** La representación debe ser lo suficientemente rica como para alimentar detectores basados en *embeddings* densos, pero conservar, a través de las *flags* y de la vista sintáctica, un nivel de interpretabilidad que permita auditar las decisiones y relacionar las anomalías detectadas con patrones estructurales concretos.
- **Compatibilidad con escenarios de una sola clase.** Dado que en muchos despliegues reales sólo se dispone de tráfico benigno para entrenar modelos, el diseño debe ser compatible con detectores de tipo *one-class* capaces de modelar la distribución del tráfico legítimo y estimar rareza sin depender de grandes colecciones de ejemplos de ataque.

Las secciones siguientes describen cómo el *pipeline* de preprocesamiento, las vistas de representación y el módulo de detección materializan estos requisitos. La Figura 4.1 presenta una visión general de esta arquitectura. Como puede observarse, la solicitud HTTP cruda se procesa mediante un *pipeline* de preprocesamiento que produce un artefacto canónico. Este alimenta dos vistas complementarias (sintáctica y semántica), cuyos detectores independientes generan puntuaciones de rareza que se combinan mediante una fusión ponderada. La decisión final compara la puntuación fusionada con un umbral τ.

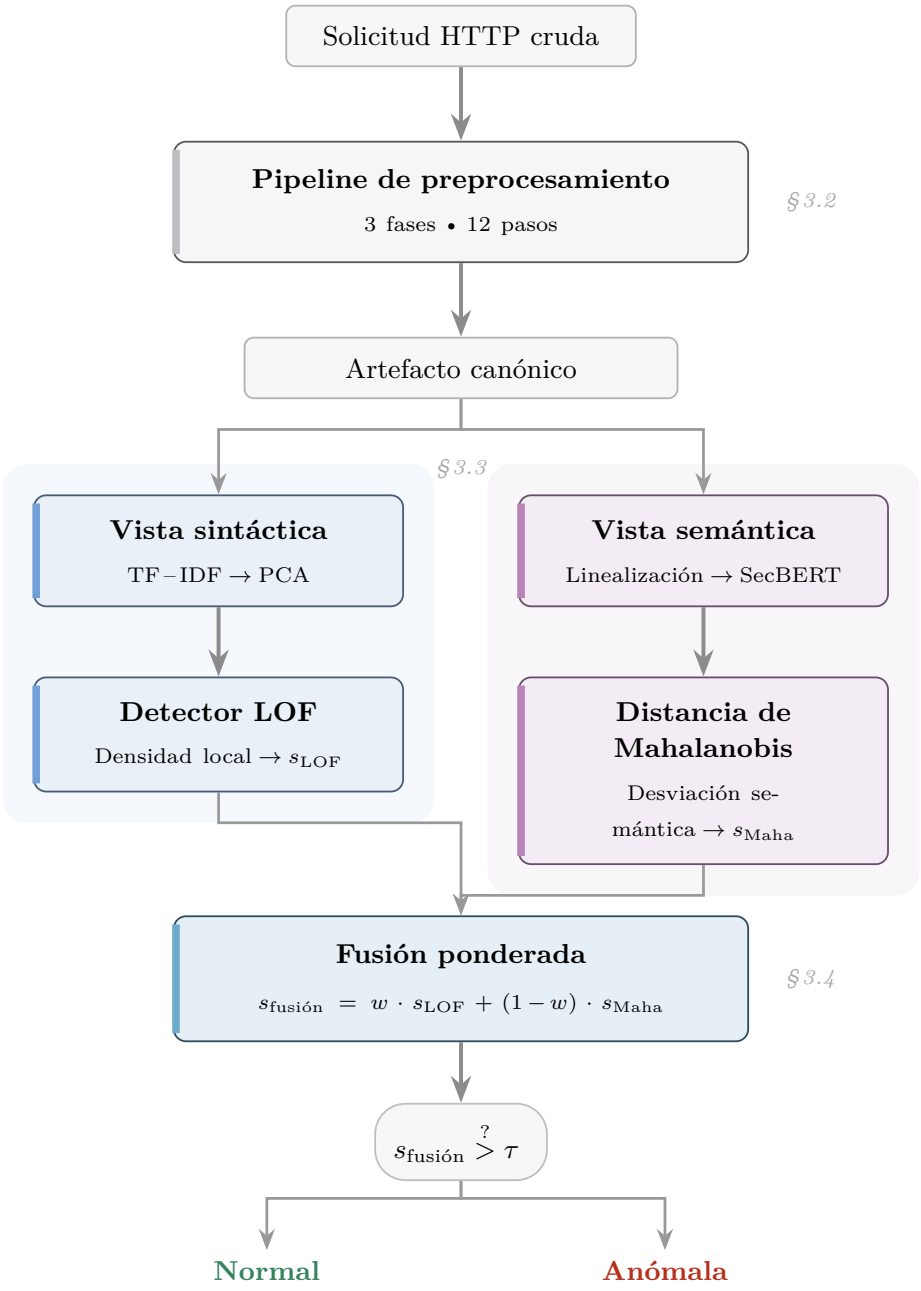


Figura 4.1: Arquitectura general de NeuralShield.

4.2. Diseño del pipeline de preprocesamiento

El *pipeline* de preprocesamiento es el primer componente operativo de NeuralShield. Recibe solicitudes HTTP y entrega a las etapas de representación y detección un artefacto estructurado donde: (i) la solicitud se descompone en elementos explícitos (método, *request target*, cabeceras, *path*, *query*); (ii) se fija una convención estable para nombres de campos, delimitadores y codificaciones; y (iii) se adjunta un conjunto de *flags* que registran las desviaciones estructurales observadas.

4.2.1. Operaciones y principios del preprocesamiento

Cada paso del *pipeline* puede realizar dos tipos de operaciones:

- **Normalización estructural:** transformar la representación de la solicitud hacia una forma canónica, interpretable de mejor forma tanto por el/los modelos que realizarán el procesamiento como por los humanos que monitorean el proceso.
- **Exposición de evidencia:** emitir una *flag* o bloque de metadatos estructurales que registre una desviación observada respecto de la línea base definida en la Sección 3.2.2.

La relación entre ambas depende de la naturaleza de la desviación observada. En general, la normalización se utiliza para fijar una estructura estable del artefacto (segmentación, etiquetas y agregados) que facilite su consumo por modelos de aprendizaje automático y su inspección por analistas. En cambio, cuando la desviación constituye una manifestación potencialmente explotable (por ejemplo, duplicación de cabeceras críticas o combinaciones ambiguas como `Content-Length` y `Transfer-Encoding`), se busca evidenciar el fragmento anómalo de la petición. Un punto importante respecto a esto es el hecho de que el *pipeline* evita "corregir" esos fragmentos potencialmente anómalos, limitándose a registrarlos mediante la *flag* correspondiente (`DUPHDR`, etc.), de modo que la evidencia permanezca explícita en la representación. Es válido notar que ambos tipos de operación no son exclusivos entre sí: un paso del *pipeline* puede ser encasillado en ambos tipos simultáneamente.

Estas operaciones satisfacen dos propiedades. La primera es la idempotencia: cada paso se define como una transformación pura; aplicarlo dos veces produce el mismo resultado que aplicarlo una sola vez, ya que las normalizaciones actúan únicamente sobre estados no canónicos y dejan inalterado lo que ya cumple la forma alineada con las RFC. Esta propiedad facilita el razonamiento sobre el efecto acumulado de las transformaciones y reduce el riesgo de introducir artefactos al combinar varias etapas. La segunda es la no destructividad respecto de la evidencia: ninguna normalización elimina información estructural sin que una *flag* la registre. Concretamente, se dan dos casos: (1) la información se registra y se elimina (el fragmento se normaliza hacia la forma canónica), y

(2) la información se registra y se mantiene (la *flag* señala la anomalía pero el contenido original permanece en el artefacto). En conjunto, ambas propiedades garantizan una representación estable y reproducible sin sacrificar información relevante para la seguridad.

Las operaciones de normalización implementan operativamente la línea base de solicitudes "bien formadas" (Sección 3.2.2): segmentan la petición, despliegan mecanismos obsoletos, adoptan convenciones estables para nombres, codificaciones y delimitadores (implementadas mediante *flags* inline, concepto definido en la Sección 5.1.1). Las operaciones de exposición de evidencia, por su parte, recorren las familias de anomalías descritas en la Sección 3.2.3 y traducen cada aparición en una *flag* o bloque de metadatos estructurales con nombre y significado precisos.

En conjunto, estas operaciones producen tres tipos de elementos en el artefacto canónico: tags de línea, marcadores estructurales que segmentan la solicitud en campos explícitos y fijan el formato del artefacto; *flags* de anomalía, anotaciones inline o consolidadas que registran desviaciones concretas observadas durante el procesamiento; y metadatos del bloque, líneas de resumen que condensan propiedades agregadas de cada bloque, es decir, métricas observables únicamente al considerar el bloque en su conjunto.

4.2.2. Especificación del pipeline

Los principios de idempotencia y no-destructividad descritos en la sección anterior se materializan en una secuencia fija de pasos deterministas. A continuación se agrupan estos pasos en tres fases secuenciales, cada una orientada a un subconjunto de las familias de anomalías identificadas en la Sección 3.2.3.

1. **Saneamiento y segmentación.** Sobre el texto crudo de la solicitud se eliminan artefactos en los bordes y se estructura la petición en una representación etiquetada explícita, registrando mediante una *flag* si el método utilizado no forma parte del conjunto de métodos estándar definidos en las especificaciones.

2. **Estructuración y normalización de cabeceras.** Se detectan cabeceras con continuación de línea (*obs-fold*) y se anota la presencia del mecanismo. Luego se canonizan nombres de cabecera, se preserva la multiplicidad de campos, se registran duplicados y condiciones estructuralmente anómalas, y se estabiliza el espaciado de valores sin borrar evidencia.

3. **Procesamiento de URL, *path* y *query*.** Se enriquecen y estabilizan las partes de la solicitud ligadas al *request target*: se detectan patrones po-

tencialmente ofensivos, se reconstruye una URL absoluta a partir de `Host` y *request target*, y se aplican normalizaciones Unicode conservadoras y decodificaciones controladas, registrando las desviaciones mediante *flags*. De manera análoga, se canoniza la *query* y se normaliza la estructura del *path* sin resolver indicadores de *traversal*.

Además de las *flags* inline y los bloques de resumen generados por las fases anteriores, el *pipeline* produce líneas de metadatos estructurales que condensan propiedades *agregadas* de sus bloques respectivos, es decir, métricas que caracterizan al bloque en su conjunto y que no se observan examinando elementos individuales. El criterio de selección responde a un principio común: incluir únicamente conteos o indicadores que *emergen del bloque completo* y que son relevantes para distinguir tráfico anómalo del benigno. Por ejemplo, que una solicitud tenga 2 cabeceras duplicadas frente a 15 constituye una diferencia estructural significativa que solo se manifiesta al realizar la agregación, ninguna cabecera individual la expone. De forma análoga, la cantidad de parámetros repetidos en la *query* es una propiedad del bloque que desaparece al examinar parámetros uno a uno. Estas propiedades son declaradas en las líneas `[HSUM]` y `[QSUM]`, y su especificación concreta se presenta en la Sección 5.1.1.

El *pipeline* incorpora también una línea de bloque `[HGF]`, destinada a consolidar las *flags* cuyo alcance es el bloque de cabeceras completo, no una cabecera individual. Investigación reciente sobre evasión de WAF muestra que resulta efectivo el distribuir mutaciones entre múltiples componentes del paquete HTTP [15], lo que hace que ciertas propiedades relevantes para la detección solo sean observables al nivel del bloque. En etapas iniciales del diseño se evaluó adjuntar estas *flags* a cada línea `[HEADER]` que las disparaba; sin embargo, esto presentó un problema: la repetición superaba el límite de tokens del codificador en una gran cantidad de solicitudes, provocando truncamiento y pérdida de señal. `[HGF]` consolida estas *flags* en una única línea al final del bloque de cabeceras, solución semánticamente correcta y eficiente en el uso de tokens. Su especificación concreta se presenta en la Sección 5.1.1.

Como se observa en la Figura 4.2, cada fase combina normalización estructural con exposición de evidencia. Las familias de anomalías abordadas por cada fase se indican a la izquierda del diagrama. El artefacto resultante alimenta las dos vistas del modelo dual de representación.

El resultado de este flujo es un artefacto único por solicitud que combina una representación canónica estable con *flags* y metadatos estructurales que capturan las anomalías relevantes para seguridad, sin borrar las manifestaciones que motivan dichas anotaciones. La especificación detallada de cada paso se presenta en el Capítulo 5.1. Este artefacto constituye la entrada común de las dos vistas de representación descritas en la Sección 4.3.

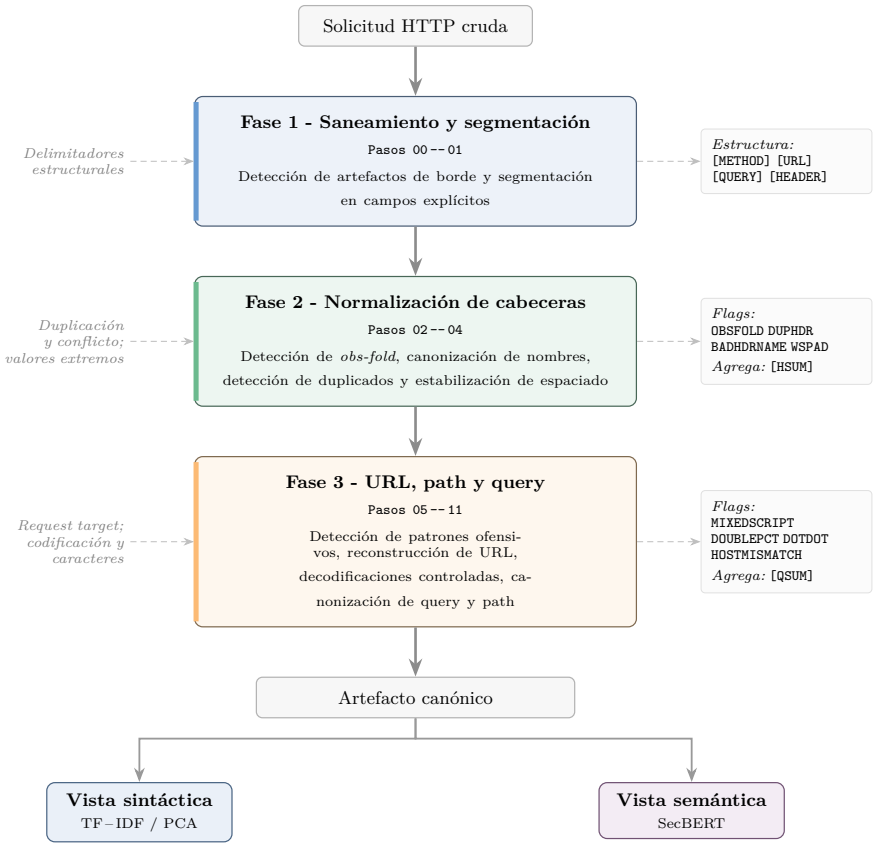


Figura 4.2: Visión general del *pipeline* de preprocesamiento de NeuralShield.

Para ilustrar el efecto concreto del *pipeline*, considérese la siguiente solicitud HTTP que combina varias anomalías: cabecera `Host` duplicada con valores distintos, coexistencia de `Content-Length` y `Transfer-Encoding`, *path traversal*, doble *percent-encoding* y espaciado irregular en valores de cabecera.

```
GET /app/%252e%252e/admin/../config?debug;verbose&format=json HTTP/1.1
host: api.example.com
Host: evil.example.com
Content-Length: 0
Transfer-Encoding:  chunked
X-Custom:   valor-con-espacios
Cookie: session=abc123
```

Listado 4.1: Solicitud HTTP cruda con múltiples anomalías estructurales

Tras recorrer las tres fases del *pipeline*, la solicitud se transforma en el artefacto canónico que se muestra a continuación. Obsérvese cómo cada campo queda segmentado con los metadatos asociados (`[METHOD]`, `[URL]`, `[HEADER]`, etc.), los nombres de cabecera se canonizan a minúsculas, y las anomalías detectadas se registran como *flags* (`DUPHDR`, `DOUBLEPCT`, `DOTDOT`, `WSPAD`) sin eliminar la evidencia original. Los metadatos de cabeceras (`[HSUM]`) y de *query* (`[QSUM]`) resumen propiedades agregadas del artefacto. Es válido resaltar que la versión del protocolo (HTTP/1.1) no es incluida en el artefacto final dado que el trabajo aborda únicamente dicha especificación, por las razones que fueron explicadas en secciones anteriores.

```
[METHOD] GET
[URL] /app/%252e%252e/admin/../config
[QUERY] debug
[QUERY] verbose
[QUERY] format=json
[QSUM] count=3 MIXEDSEP
[HEADER] host: api.example.com DUPHDR
[HEADER] host: evil.example.com DUPHDR
[HEADER] content-length: 0
[HEADER] transfer-encoding: chunked WSPAD
[HEADER] x-custom: valor-con-espacios WSPAD
[HEADER] cookie: session=abc123
[HSUM] h_count=6 dup_names=1 hopbyhop=0 bad_names=0 total_bytes=182
[FLAGS] DUPHDR DOUBLEPCT DOTDOT WSPAD
```

Listado 4.2: Artefacto canónico resultante con *flags* de anomalía

4.2.3. Justificación de la estrategia de inyección de conocimiento

Al diseñar el *pipeline*, se evaluaron dos alternativas para incorporar las *flags* y metadatos estructurales al proceso de representación: inyectarlos directamente en el texto del artefacto canónico que consumen los vectorizadores, o extraerlos como un vector numérico independiente que se concatena al *embedding* textual. Se adoptó la primera alternativa, y a continuación se exponen las razones que fundamentan esta elección.

En primer lugar, debemos notar que nuestro procesamiento no reduce la petición a únicamente las flag y metadatos obtenidos, sino que mantiene los elementos de la petición original, haciendo modificaciones a los mismos según los criterios explicados anteriormente. Esta decisión está relacionada al principio de no-destructividad establecido en la Sección 4.2.1: el *pipeline* agrega anotaciones al artefacto sin eliminar el contenido original de la solicitud. Las *flags* no reemplazan la señal presente en el texto, sino que la complementan. El artefacto canónico conserva íntegramente el método, la URL, los parámetros, los valores de cabecera y la estructura del mensaje, elementos que contienen patrones semánticos y léxicos que ningún conjunto finito de reglas deterministas puede capturar por completo. Por ejemplo, combinaciones inusuales de *User-Agent* y *Path*, valores atípicos en cabeceras de control o secuencias de parámetros implausibles constituyen señales que solo emergen de la representación aprendida sobre el texto completo.

En segundo lugar, al mantener las *flags* como tokens dentro del mismo documento, se preserva la relación contextual entre la anotación y el fragmento donde se observa la desviación. Un indicador `DUPHDR` adjunto a una línea `[HEADER] host:` tiene un significado distinto al mismo indicador junto a `[HEADER] accept:`: el primero está asociado a *Host Header Injection*, mientras que el segundo es frecuentemente benigno. En un vector plano, esta distinción se pierde a menos que se diseñen *features* cruzadas adicionales para cada combinación de flag y campo, lo que escala de forma combinatoria. Los modelos de representación textual (TF-IDF con n-gramas, o modelos contextuales como SecBERT, descritos en la Sección 4.3) capturan estas relaciones posicionales de forma natural a través de la co-ocurrencia de tokens o la atención contextual.

Finalmente, la estrategia de inyectar anotaciones en el texto antes de la vectorización responde al concepto de inyección de conocimiento (*knowledge injection*) en procesamiento de lenguaje natural. Liu et al. [73] proponen K-BERT, un modelo que inyecta tripletas de un grafo de conocimiento directamente como tokens adicionales en la secuencia de entrada de BERT, demostrando que el enriquecimiento textual a nivel de *input* mejora la representación sin requerir modificaciones arquitectónicas. De forma análoga, Ke et al. [74] inyectan etiquetas de categoría gramatical y polaridad de sentimiento en la capa de *embeddings* de un modelo preentrenado, estableciendo que las anotaciones lingüísticas ex-

traídas de forma determinista pueden mejorar el desempeño cuando se integran directamente en la entrada del vectorizador. Estos antecedentes respaldan que la inyección de *features* como texto es una estrategia válida cuando se busca preservar el contexto original y permitir que el vectorizador aprenda patrones de co-ocurrencia entre las anotaciones y el contenido, interacciones que un esquema de concatenación tardía no podría capturar. Esta hipótesis fue posteriormente validada en el dominio específico de detección de ataques web por Luo et al. [75], quienes proponen WADBERT, un sistema que combina la extracción de características estructurales de solicitudes HTTP con codificadores basados en SecBERT (la misma familia de modelos utilizada en este trabajo), fusionando ambas fuentes mediante *multi-head attention*.

Es importante reconocer que la estrategia adoptada implica un *trade-off*: las *flags* se representan como tokens y, por tanto, su contribución al vector final depende de cómo el vectorizador las pondera dentro del documento completo. Un esquema alternativo, donde las *flags* alimenten un clasificador como vector numérico separado (posiblemente concatenado al *embedding* textual), podría otorgarles un peso más directo y controlable. Esta alternativa se identifica como una línea de trabajo futuro en la Sección 7.2.

En síntesis, el artefacto canónico producido por el *pipeline* alimenta dos vistas de representación, descritas en la Sección 4.3: una sintáctica basada en TF-IDF/PCA y una semántica basada en SecBERT. Ambas se ajustan exclusivamente sobre tráfico considerado benigno. La complementariedad de ambas perspectivas se explora en la etapa de detección descrita en la Sección 4.4.

4.3. Diseño de las vistas de representación

Sobre el artefacto (representación canónica de texto estructurado producida por el *pipeline*, Sección 4.2), NeuralShield proyecta cada solicitud en espacios vectoriales aptos para el aprendizaje automático. Esta sección describe cómo se instancian las dos vistas del modelo dual establecido en los requisitos de diseño (Sección 4.1): una representación sintáctica basada en TF-IDF/PCA y una representación semántica basada en SecBERT. El objetivo del modelo dual no es duplicar información, sino permitir que cada vista se especialice en un tipo de señal: la vista sintáctica es especialmente sensible a desviaciones estructurales ligadas al *parsing* y al *framing*; la vista semántica es más adecuada para capturar rarezas contextuales que se mantienen dentro de la gramática del protocolo [48], [71]. Ambas vistas reciben como entrada el mismo artefacto estructurado (incluyendo *flags* y metadatos), pero lo procesan de formas distintas.

4.3.1. Representación sintáctica de solicitudes HTTP

La vista sintáctica tiene por objetivo capturar la *forma* de la solicitud HTTP, es decir, cómo está escrita dentro del protocolo, sin intentar modelar directa-

mente su intención funcional. Esta vista parte de la representación canónica producida por el *pipeline* de preprocesamiento, donde cada petición se ha convertido ya en un documento de texto estructurado con metadatos que describen método, destino, cabeceras, parámetros de *query* y anomalías estructurales expuestas mediante *flags*.

Sobre estos documentos se aplica un modelo TF-IDF con n-gramas de caracteres de longitud 1 a 3, que asigna a cada término (o subsecuencia de caracteres) un peso $tf \times idf$: proporcional a su frecuencia en la solicitud (tf) e inversamente proporcional a su frecuencia en el corpus de tráfico legítimo (idf). El uso de n-gramas de caracteres permite capturar fragmentos sub-léxicos recurrentes en las solicitudes (prefijos de rutas, esquemas de codificación, secuencias de escape), lo que refuerza la sensibilidad de esta vista a las desviaciones estructurales. El resultado es un vector disperso de alta dimensión en el que cada coordenada refleja la relevancia de un rasgo sintáctico observable: presencia o ausencia de determinadas cabeceras, combinaciones particulares de parámetros, aparición de patrones estructurales raros, etc.

Dado que esta representación es de muy alta dimensionalidad, se aplica posteriormente una reducción de dimensionalidad mediante Análisis de Componentes Principales (PCA). Esta transformación proyecta los vectores dispersos a un espacio denso de tamaño fijo, preservando las direcciones de mayor varianza. Desde la perspectiva de la detección, este paso condensa las combinaciones de rasgos sintácticos que aparecen con más frecuencia en el tráfico legítimo y facilita que los detectores delimiten la región de comportamiento normal.

En conjunto, la vista sintáctica puede interpretarse como una proyección numérica de la estructura de la petición: solicitudes que siguen los patrones habituales de cabeceras y parámetros quedan cerca del núcleo de la distribución, mientras que aquellas que activan *flags* poco frecuentes o combinan cabeceras y rutas de forma inusual se sitúan en regiones de baja densidad. Esta representación proporciona una referencia sintáctica concreta para NeuralShield y complementa la perspectiva contextual de la vista semántica descrita a continuación.

4.3.2. Representación semántica de solicitudes HTTP

La vista semántica aborda la representación desde una perspectiva complementaria. En lugar de centrarse en la forma exacta de la petición, busca capturar su significado funcional: qué tipo de operación se está intentando realizar, qué recursos y parámetros se combinan, qué patrones contextuales aparecen en las cabeceras de control y en los valores relevantes para la lógica de la aplicación.

Para ello se recurre a SecBERT, un modelo de lenguaje basado en la arquitectura BERT y preentrenado específicamente sobre corpus de ciberseguridad [54]. La elección de un modelo de dominio responde a que su vocabulario y sus re-

presentaciones internas ya incorporan familias de tokens frecuentes en tráfico HTTP, *logs* y textos técnicos de seguridad, lo que reduce la distancia de transferencia respecto de modelos de propósito general.

El artefacto canónico producido por el *pipeline* se procesa con SecBERT, que produce un *embedding* contextual denso a partir del token especial de clasificación (`[CLS]`). Este token es prepuesto por BERT al inicio de cada secuencia de entrada; tras el preentrenamiento, su representación agrega información de toda la secuencia mediante atención y constituye la forma estándar de obtener un *embedding* a nivel de documento. La variante base usa exclusivamente `[CLS]`, mientras que variantes alternativas aplican *pooling* sobre capas ocultas; los detalles de dimensionalidad y las variantes de codificación se describen en la Sección 5.2.3.

A diferencia de la vista sintáctica, donde cada coordenada del vector TF-IDF tiene una correspondencia directa con un rasgo léxico observable, aquí cada dimensión codifica patrones distribuidos aprendidos durante el preentrenamiento. Dos peticiones con resultados funcionalmente similares tienden a proyectarse cerca en este espacio, aunque difieran en detalles de codificación o en el conjunto exacto de cabeceras. Simétricamente, solicitudes cuya forma superficial respeta la gramática del protocolo pero cuyo contenido es anómalo, un patrón alineado con la desalineación sintaxis–semántica discutida en la Sección 3.1.1, se alejan de la región que ocupa el tráfico legítimo.

4.4. Diseño del módulo de detección de anomalías

Definidas las vistas de representación sintáctica y semántica de las solicitudes HTTP (Sección 4.3), el siguiente componente es el módulo encargado de modelar el tráfico legítimo y asignar a cada nueva petición un score de anomalía. En esta sección se describe el objetivo general de detección, el paradigma de *one-class* adoptado y la arquitectura concreta del módulo de anomalías: un detector sintáctico basado en LOF sobre TF-IDF/PCA, un detector semántico basado en distancia de Mahalanobis sobre *embeddings* de SecBERT y un criterio de fusión que combina ambas vistas.

4.4.1. Objetivo de detección y paradigma one-class

El módulo de detección adopta el paradigma *one-class* descrito en la Sección 2.4.1: los detectores se entrenan exclusivamente con tráfico benigno, construyen una estimación de la región de alta densidad del tráfico legítimo y estiman la rareza de cada nueva muestra. Las etiquetas de ataque se reservan para la fase de evaluación. En este marco, una *anomalía* se define como una solicitud cuya proyección vectorial se aleja significativamente del núcleo de la distribución

legítima en al menos una de las vistas.

4.4.2. Detector sintáctico: LOF sobre TF–IDF/PCA

Sobre la vista sintáctica, NeuralShield emplea *Local Outlier Factor* (LOF) como detector de anomalías (véase la Sección 2.4.2 para la definición formal del algoritmo). LOF resulta especialmente adecuado para esta representación porque se adapta a la geometría local del espacio TF-IDF/PCA: no supone una distribución global esférica o gaussiana, sino que captura irregularidades locales, lo que permite representar configuraciones donde coexisten múltiples subnubes de tráfico legítimo. Las solicitudes anómalas se manifiestan como vectores aislados o incrustados en regiones de baja densidad, con patrones estructurales poco frecuentes ligados a las familias de anomalías descritas en la Sección 3.2.3.

4.4.3. Detector semántico: distancia de Mahalanobis sobre SecBERT

Sobre la vista semántica, NeuralShield utiliza la distancia de Mahalanobis como criterio de desviación (véase la Sección 2.4.3 para la definición formal). En los *embeddings* densos generados por SecBERT, el tráfico legítimo tiende a agruparse en torno a una región aproximadamente elíptica, lo que hace que la distancia de Mahalanobis sea una medida natural de rareza: pondera cada dimensión por la estructura de covarianza del tráfico normal, reforzando la detección de combinaciones inusuales sin sobrepenalizar desviaciones en direcciones de alta varianza.

4.4.4. Fusión de detectores y criterio de decisión

Los dos detectores anteriores producen puntuaciones independientes que reflejan rareza en dominios distintos: el score de LOF s_{LOF} mide hasta qué punto la forma de la solicitud resulta inusual en el espacio TF-IDF/PCA, mientras que la distancia de Mahalanobis $s_{\text{Mahalanobis}}$ cuantifica la desviación semántica en el espacio de *embeddings* de SecBERT. Para integrar ambas fuentes de evidencia, NeuralShield aplica una estrategia de fusión ponderada.

Dado que ambos detectores operan en escalas distintas, las puntuaciones se normalizan para que contribuyan en una escala comparable. Las puntuaciones normalizadas se combinan mediante una fusión lineal:

$$s_{\text{fusion}} = w \cdot s_{\text{LOF}} + (1 - w) \cdot s_{\text{Mahalanobis}},$$

donde $w \in [0, 1]$ controla la influencia relativa de cada vista. Valores de w cercanos a 1 privilegian la sensibilidad a desviaciones estructurales visibles en

la sintaxis; valores cercanos a 0 enfatizan las anomalías contextuales capturadas por la vista semántica. La selección del peso se realiza mediante búsqueda exhaustiva sobre un conjunto discreto de valores candidatos, evaluando cada configuración en la partición de validación de cada dataset y seleccionando el valor que maximiza el Area Under the Curve (AUC) ROC (véase la Sección 6.5 para los resultados detallados).

Finalmente, el criterio de decisión se define fijando un umbral τ sobre s_{fusion}: las solicitudes cuya puntuación de fusión supera τ se marcan como anómalas. El umbral τ se fija en el percentil $(1-\alpha)$ de los scores de anomalía calculados sobre las muestras normales del conjunto de entrenamiento, donde α corresponde a la tasa de falsos positivos objetivo (*false positive rate*, FPR). En este trabajo se utiliza $\alpha = 0{,}05$, de modo que τ coincide con el percentil 95 de la distribución de scores del tráfico legítimo.

Capítulo 5

Implementación de NeuralShield

Este capítulo documenta las decisiones concretas que materializan el diseño presentado en el Capítulo 4.1. Se organiza en tres secciones: la implementación del *pipeline* de preprocesamiento (Sección 5.1), la construcción de las vistas de representación vectorial (Sección 5.2) y la implementación de los detectores de anomalías junto con su lógica de fusión (Sección 5.3).

5.1. Implementación del pipeline de preprocesamiento

Siguiendo los principios de diseño establecidos en la Sección 4.2, esta sección se organiza en cuatro subsecciones: (i) el contrato de entrada/salida del artefacto canónico, (ii) el significado operativo de las *flags*, (iii) una matriz explícita paso–flags, y (iv) ejemplos completos de entrada/salida. El objetivo es que cada flag, tag de línea o metadato de bloque mencionado en el resto del documento tenga una definición clara, una condición de activación concreta y un emisor identificable dentro del flujo.

5.1.1. Contrato de Entrada y Salida

Cada paso del *pipeline* implementa una función pura que recibe una solicitud estructurada como texto y devuelve una nueva versión del mismo artefacto. La composición se define externamente en un archivo de configuración, por lo que el orden de ejecución es explícito y reproducible sin modificar código fuente.

El contrato de salida del artefacto canónico se basa en los tags de línea introducidos en la Sección 4.2. Los tags principales son:

- `[METHOD]`: método HTTP observado en la línea de petición.
- `[URL]`: *request target* original.
- `[URL_ABS]`: URL absoluta, conteniendo schema, autoridad y ruta accedida.
- `[QUERY]`: parámetros de la consulta realizada.
- `[HEADER]`: líneas de cabecera observadas.
- `[FLAGS]`: línea que agrega las *flag* emitidas a lo largo de todo el preprocesamiento.
- `[HGF]`: *flags* globales de cabeceras no atribuibles a una única línea `[HEADER]`.
- `[QSEP]`: separador dominante en la consulta (`&` o `;`).
- `[HSUM]`: resumen agregado del bloque de cabeceras.
- `[QSUM]`: resumen global de la consulta.

La implementación mantiene las propiedades definidas en diseño. En particular, la idempotencia exige que volver a ejecutar un paso no introduzca cambios nuevos sobre entradas ya procesadas por ese paso, y la no-destructividad respecto de evidencia exige que las desviaciones relevantes queden observables mediante texto canónico y/o *flags*.

Composición de las líneas resumen

Los criterios de selección de métricas para las líneas de metadatos estructurales `[HSUM]` y `[QSUM]` se establecen en la Sección 4.2.2. A continuación se detalla su composición concreta.

`[HSUM]` registra la cardinalidad del bloque de cabeceras, la cantidad de nombres duplicados, cabeceras *hop-by-hop*, nombres malformados y el tamaño total en bytes. `[QSUM]` registra la cantidad de parámetros junto con *flags* estructurales que aplican al bloque como un todo: repeticiones de clave (`QREPEAT`), patrones de arreglo (`QARRAY`), parámetros sin valor (`QBARE`, `QEMPTYVAL`) y presencia de bytes nulos (`QNUL`). El desglose campo por campo de ambas líneas se presenta en el Anexo A.

La línea `[HGF]` consolida las *flags* de bloque de cabeceras cuya repetición por línea comprometería el límite de tokens del codificador; su motivación se discute en la Sección 4.2.3. El catálogo completo de *flags* y sus condiciones de activación se presenta en el Anexo A.

5.1.2. Semántica de flags

Una *flag* es una anotación estructural con dos componentes: un `trigger` (condición sintáctica concreta que la activa) y una indicación (lectura de por qué esa desviación importa). En esta implementación, las *flags* no atribuyen un ataque ni constituyen una clasificación final: registran evidencia intermedia y vinculan la desviación con patrones de anomalía estructural discutidos en secciones anteriores.

El sistema usa tres formas de posicionamiento en el artefacto:

- **Tags de línea:** prefijos estructurales que segmentan la solicitud en campos explícitos (`[METHOD]`, `[URL]`, `[QUERY]`, `[HEADER]`).
- **En la misma línea (inline):** la *flag* se adjunta al tag de línea donde se observa la desviación.
- **En una línea resumen del bloque:** la *flag* no se puede atribuir a una única línea, así que se registra como resumen del bloque: cabeceras en `[HSUM]`, *flags* globales de cabeceras en `[HGF]`, consulta en `[QSUM]` y resumen global del artefacto en `[FLAGS]`.

Además, algunas *flags* incluyen un identificador con el formato `FLAG:<clave>` para indicar explícitamente a qué parámetro aplica (por ejemplo, `QREPEAT:page` o `QARRAY:items[]`).

Esta distinción evita ambigüedades: dos solicitudes pueden compartir una misma *flag* global y, sin embargo, diferir en los triggers locales que la originaron.

Duplicación intencional de flags entre líneas y resumen

Ciertas *flags* aparecen simultáneamente en la línea donde se detecta la desviación (inline) y en la línea resumen `[FLAGS]` al final del artefacto. Esta duplicación es intencional porque las *flags* aportan información a dos escalas complementarias. En posición inline, la *flag* forma n-gramas con el contenido que la disparó (por ejemplo, `keep-alive HOPBYHOP` o `host: evil.example DUPHDR`), lo que permite a los vectorizadores capturar *qué elemento* de la solicitud activó la anotación. En la línea `[FLAGS]`, las *flags* coinciden entre sí, exponiendo el *perfil de anomalías* de la solicitud como un todo: una petición con `DUPHDR`, `HOPBYHOP` y `OBSFOLD` tiene una firma distinta a una con solo `DUPHDR`. Sin la duplicación habría que renunciar a una de estas dos escalas: solo inline se pierde la coincidencia entre *flags* dispersas por el artefacto; solo en `[FLAGS]` se pierde la asociación entre cada *flag* y el fragmento concreto que la originó. Como se discute en la Sección 4.2.3, la separación de las *flags* en un canal independiente se identifica como línea de trabajo futuro.

El catálogo completo de *flags*, con sus condiciones de activación, indicaciones de seguridad y ejemplos representativos, se presenta en el Anexo A.

5.1.3. Descripción a Alto Nivel de los Pasos de Preprocesamiento

Esta sección resume, paso a paso, qué hace cada etapa del preprocesamiento sobre el artefacto canónico: qué líneas consume, qué transformaciones aplica y qué líneas (tags y *flags*) agrega a la salida.

Paso `FramingCleanup`. Elimina BOM (`\ufeff`) y recorta caracteres de control en los bordes del mensaje.

Paso `RequestStructurer`. Valida la *request-line* (método, *request target*, versión HTTP), separa URL y query en tags de línea, y emite cada cabecera dentro de un tag de línea `[HEADER]`.

Paso `HeaderUnfoldObsFold`. Analiza continuaciones de cabecera y bytes de control incrustados. Su objetivo es exponer anomalías de delimitación.

Paso `HeaderNormalizer`. Valida nombres de cabecera, detecta duplicados y cabeceras *hop-by-hop*, y emite `[HSUM]` y `[HGF]` como metadatos del bloque de cabeceras. Las líneas `[HEADER]` se reordenan alfabéticamente para reducir la varianza de entrada de los modelos; esta normalización descarta el orden original, cuya preservación se identifica como trabajo futuro (Sección 7.2).

Paso `WhitespaceCollapse`. Colapsa secuencias de espacios y tabulaciones en valores de cabecera a un único espacio y recorta los extremos; emite `WSPAD` cuando la normalización modifica el valor original.

Paso `DangerousCharacters`. Escanea `[URL]`, `[QUERY]` y `[HEADER]` en busca de caracteres estructuralmente sensibles (literales y codificados) y mezcla de alfabetos.

Paso `AbsoluteUrlBuilder`. Reconstruye `[URL_ABS]` en la forma canónica (esquema//host[puerto]/ruta[?query]) detectando el tipo de *request-target*. Extrae y valida `Host`, aplica tratamiento IDNA cuando corresponde y expone inconsistencias mediante `HOSTMISMATCH`.

Paso `UnicodeSanitizer`. Evalúa `[URL]` y `[QUERY]` para detectar anomalías Unicode (control, formato, fullwidth, entre otras).

Paso `PercentDecodeNormalizer`. Inspecciona `[URL]` y `[QUERY]` en busca de secuencias *percent-encoded* sospechosas y patrones de doble codificación.

Paso `HtmlEntityDetector`. Detecta entidades HTML en URL/query comparando la cadena observada con una versión desescapada (es decir, la misma cadena tras decodificar entidades como `<`, `&` o `&#x2F;`), en caso de encontrar diferencias, anota `HTMLENT`.

Paso `QueryParser`. Reprocesa bloques de query, detecta su separador dominante y emite `[QSEP]` con dicho valor. Cada parámetro es representado en una línea independiente, agregando *flags* en los que corresponda. Además sintetiza anomalías globales en `[QSUM]`.

Paso `PathNormalizer`. Detecta anomalías de forma del path y expone evidencia de anomalías con *flags* cuando corresponda.

5.1.4. Orden de ejecución

El orden de ejecución de los pasos mencionados anteriormente se aplica de forma determinista:

1. `FramingCleanup`
2. `RequestStructurer`
3. `HeaderUnfoldObsFold`
4. `HeaderNormalizer`
5. `WhitespaceCollapse`
6. `DangerousCharacters`
7. `AbsoluteUrlBuilder`
8. `UnicodeSanitizer`
9. `PercentDecodeNormalizer`
10. `HtmlEntityDetector`
11. `QueryParser`
12. `PathNormalizer`

5.1.5. Ejemplos end-to-end del artefacto canónico

Para ilustrar el funcionamiento del *pipeline*, se presentan tres ejemplos que cubren familias de anomalías distintas: manipulación de cabeceras, evasión por codificación en rutas, e inyección en la cadena de consulta. En cada caso se muestra la solicitud HTTP original y el artefacto canónico resultante.

Ejemplo 1: *request smuggling* mediante cabeceras.

```
POST /api/v2/users HTTP/1.1
 smuggled-orphan-value
Host: api.example.com
Host: evil.example.com
Connection: keep-alive
X-Note: ok
 X-Admin: true
X-Pad:   internal
```

Listado 5.1: Solicitud con anomalías de cabeceras orientadas a *request smuggling*

El artefacto canónico resultante es:

```
[METHOD] POST
[URL] /api/v2/users
[URL_ABS] http://api.example.com/api/v2/users
[HEADER] connection: keep-alive HOPBYHOP
[HEADER] host: api.example.com DUPHDR
[HEADER] host: evil.example.com DUPHDR
[HEADER] x-note: ok X-Admin: true OBSFOLD
[HEADER] x-pad:   internal    WSPAD
[HSUM] h_count=5 dup_names=1 hopbyhop=1 bad_names=0
       total_bytes=132
[FLAGS] BADHDRCONT DUPHDR HOPBYHOP OBSFOLD
        WSPAD
```

Listado 5.2: Artefacto canónico: evidencia de *smuggling* en cabeceras

Ejemplo 2: *path traversal* con evasión por codificación.

```
GET /static/./css//../%252e%252e/etc/passwd HTTP/1.1
Host:  pp.café.example
Accept: text/html
X-Token: secret
```

Listado 5.3: Solicitud con *path traversal* y evasión por doble codificación

El artefacto canónico resultante es:

```
[METHOD] GET
[URL] /static/css/../%2e%2e/etc/passwd DOTCUR DOTDOT
       MULTIPLESLASH DOUBLEPCT
[URL_ABS] http://xn--pp-zja.xn--caf-dma.example
          /static/css/../%2e%2e/etc/passwd IDNA
[HEADER] accept: text/html
[HEADER] host:  pp.café.example MIXEDSCRIPT IDNA
[HEADER] x-token: secret
[HSUM] h_count=3 dup_names=0 hopbyhop=0 bad_names=0
       total_bytes=72
[FLAGS] DOTCUR DOTDOT DOUBLEPCT IDNA
        MIXEDSCRIPT MULTIPLESLASH
```

Listado 5.4: Artefacto canónico: evidencia de *traversal* y evasión de codificación

Ejemplo 3: inyección en cadena de consulta (*XSS/HPP*).

```
GET /search?q=&#x3c;script&#x3e;alert(1)
    &id=1;DROP&role=user&role=admin
    &items[]=a&items[]=b&debug
    &empty=&tok=ab%00cd HTTP/1.1
Host: shop.example.com
Cookie: session=xyz
```

Listado 5.5: Solicitud con inyección en cadena de consulta

El artefacto canónico resultante es:

```
[METHOD] GET
[URL] /search
[URL_ABS] http://shop.example.com/search
[QUERY] q=<script>alert(1) HTMLENT ANGLE PAREN
[QUERY] id=1;DROP QRAWSEMI SEMICOLON
[QUERY] role=user
[QUERY] role=admin
[QUERY] items[]=a
[QUERY] items[]=b
[QUERY] debug QBARE
[QUERY] empty= QEMPTYVAL
[QUERY] tok=ab%00cd QNUL PCTNULL
[HEADER] cookie: session=xyz
[HEADER] host: shop.example.com
[HSUM] h_count=2 dup_names=0 hopbyhop=0 bad_names=0
       total_bytes=44
[QSEP] &
[QSUM] count=9 QARRAY:items[] QREPEAT:role QBARE
       QEMPTYVAL QNUL
[FLAGS] ANGLE HTMLENT PAREN PCTNULL
        QARRAY:items[] QBARE QEMPTYVAL QNUL QRAWSEMI
        QREPEAT:role SEMICOLON
```

Listado 5.6: Artefacto canónico: evidencia de inyección en *query*

5.1.6. Matriz paso–flags

La Tabla 5.1 sintetiza qué produce cada paso del *pipeline* y permite trazar cada *flag* hasta su emisor. En la columna “Líneas/Meta”, las entradas entre corchetes corresponden a tags de línea estructurales o metadatos de bloque emitidos por el paso; las entradas en la columna “Flags emitidas” son *flags* inline o globales que anotan desviaciones puntuales.

Paso	Líneas/Meta	Flags emitidas
Fase 1 — Saneamiento y segmentación		
FramingCleanup	Sin líneas nuevas	Sin flags nuevas.
RequestStructurer	[METHOD], [URL], [QUERY], [HEADER], [FLAGS]	UNUSUAL_METHOD.
Fase 2 — Estructuración y normalización de cabeceras		
HeaderUnfoldObsFold	Sin líneas nuevas	OBSFOLD, BADHDRCONT, BADCRLF.
HeaderNormalizer	[HSUM], [HGF]	BADHDRNAME, DUPHDR, HDRMERGE, HOPBYHOP
WhitespaceCollapse	Sin líneas nuevas	WSPAD.
Fase 3 — Procesamiento de URL, path y query		
DangerousCharacters	Sin líneas nuevas	ANGLE, QUOTE, SEMICOLON, PAREN, BRACE, PIPE, BACKSLASH, SPACE, NUL, QNUL, MIXEDSCRIPT.
AbsoluteUrlBuilder	[URL_ABS]	HOSTMISMATCH, IDNA, BADHOST .
UnicodeSanitizer	Sin líneas nuevas	FULLWIDTH, CONTROL, UNICODE_FORMAT, MATH_UNICODE, INVALID_UNICODE.
PercentDecode Normalizer	Sin líneas nuevas	DOUBLEPCT, PCTSLASH, PCTBACKSLASH, PCTSPACE, PCTCONTROL, PCTNULL, PCTSUSPICIOUS.
HtmlEntityDetector	Sin líneas nuevas	HTMLENT.
QueryParser	[QSEP], [QSUM]	QSEMISEP, QRAWSEMI, QBARE, QEMPTYVAL, QNONASCII, QLONG, QARRAY:<key>, QREPEAT:<key>, QNUL.
PathNormalizer	Sin líneas nuevas	HOME, MULTIPLESLASH, DOTCUR, DOTDOT.

Tabla 5.1: Matriz de trazabilidad entre pasos del pipeline y evidencia emitida. El catálogo completo de *flags*, con ejemplos de activación y descripciones detalladas, se presenta en el Anexo A.

5.2. Implementación de las vistas de representación

Esta sección documenta cómo se construyen las representaciones vectoriales a partir del artefacto canónico, concretando el marco dual descrito en la Sección 4.3.

5.2.1. Flujo de datos y orquestación

La extracción de representaciones se organiza como un pipeline modular que separa el acceso a datos, el preprocesamiento y la codificación. El sistema consume colecciones de solicitudes HTTP almacenadas en formato JSONL (*JSON Lines*), donde cada línea es un objeto JSON independiente con las claves `"request"` (texto crudo de la solicitud HTTP) y `"label"` (etiqueta de clase). Un componente lector recorre estos archivos de forma secuencial, agrupa las peticiones en lotes y, cuando la configuración así lo indica, aplica el pipeline de preprocesamiento descrito en la Sección 4.2 para obtener una representación canónica.

Sobre esta base, un módulo de orquestación instancia el codificador sintáctico o semántico correspondiente y gestiona la iteración sobre el conjunto de datos. Este mismo flujo se emplea tanto en la fase de entrenamiento como en la fase

de detección, lo que garantiza que las solicitudes nuevas se procesan con la misma configuración y en las mismas condiciones que el tráfico legítimo utilizado durante el ajuste de los modelos.

5.2.2. Vista sintáctica: TF–IDF y PCA

La vista sintáctica representa cada solicitud HTTP como un documento que combina, en una misma secuencia, los elementos canónicos producidos por el pipeline de preprocesamiento: método, URL, parámetros de consulta, cabeceras y *flags* estructurales, de acuerdo con la definición presentada en la Sección 4.3.1.

Sobre este corpus se aplica el modelo TF-IDF descrito en la Sección 4.3.1, exponiendo como hiperparámetros el tamaño máximo de vocabulario, el filtrado por frecuencia y el rango de n-gramas, lo que permite ajustar la granularidad léxica en los experimentos. La reducción de dimensionalidad mediante PCA se aplica a continuación sobre los vectores resultantes. Tanto el modelo TF-IDF como la transformación PCA se entrenan una sola vez sobre tráfico legítimo y se reutilizan sin modificaciones durante la inferencia, de modo que cualquier solicitud nueva se proyecta en el mismo espacio sintáctico que se utilizó para entrenar los detectores.

5.2.3. Vista semántica: SecBERT

La vista semántica se implementa con una familia de codificadores basados en SecBERT (Sección 4.3.2). Cada solicitud preprocesada se linealiza como una secuencia de tokens etiquetados (método, URL, cabeceras, flags) y se tokeniza con el esquema del modelo, aplicando padding y truncado hasta una longitud máxima fija. Se implementan tres variantes de codificación:

- **Variante base.** Ejecuta el modelo en modo inferencia y toma la representación contextual del token especial `[CLS]` como embedding global de la petición, generando vectores densos de 768 dimensiones. Esta es la variante por defecto.
- **Variante adaptada.** Carga un *checkpoint* afinado sobre tráfico HTTP y combina las dos últimas capas ocultas, aplicando *pooling* medio y máximo sobre los tokens de contenido para obtener embeddings de mayor expresividad.
- **Variante ponderada por flags.** Introduce ponderaciones específicas para tokens asociados a *flags* estructurales, de modo que las anomalías expuestas por el pipeline tengan mayor influencia en el espacio semántico.

Las alternativas se evalúan empíricamente en el capítulo de evaluación experimental. En todos los casos, los codificadores reciben secuencias canónicas y producen matrices densas consumibles directamente por los detectores.

5.2.4. Persistencia y reutilización de *embeddings*

Para desacoplar la extracción de representaciones del entrenamiento de detectores, NeuralShield persiste los *embeddings* generados como artefactos reutilizables. La configuración de cada corrida agrupa parámetros como el tamaño de lote, el tipo de codificador, la activación del pipeline y el dispositivo de ejecución.

Cada lote de *embeddings* se almacena junto con sus metadatos (índices, etiquetas, identificador de modelo y configuración), de manera que los entrenamientos posteriores de los detectores puedan operar directamente sobre estos artefactos sin repetir la fase de codificación. Este patrón facilita la experimentación controlada: diferentes detectores y estrategias de fusión pueden evaluarse sobre el mismo conjunto de representaciones sintácticas y semánticas, preservando reproducibilidad entre corridas.

5.3. Implementación del módulo de detección

Siguiendo la arquitectura de detección descrita en la Sección 4.4, esta sección documenta cómo se implementan los detectores y la lógica de fusión que combina sus puntuaciones de anomalía.

5.3.1. Detector sintáctico basado en LOF

El detector LOF se ajusta sobre los vectores producidos por la vista TF-IDF/PCA (Sección 5.2.2), utilizando únicamente tráfico legítimo durante el entrenamiento. Los hiperparámetros expuestos son el número de vecinos k, la tasa de contaminación esperada y la métrica de distancia. La puntuación bruta se invierte de signo para que valores mayores indiquen mayor anomalía, manteniendo así una convención uniforme con el detector semántico.

5.3.2. Detector semántico basado en Mahalanobis

El detector de Mahalanobis se ajusta sobre los *embeddings* producidos por SecBERT (Sección 5.2.3), estimando la media y la matriz de covarianza del tráfico legítimo. Durante la inferencia, cada solicitud recibe como puntuación su distancia de Mahalanobis al centroide aprendido: valores no negativos donde puntuaciones mayores indican mayor desviación semántica respecto del tráfico normal.

En ambos detectores, el modelo entrenado se serializa junto con los parámetros de normalización aplicados a los *embeddings* y la dimensionalidad del espacio de entrada, de modo que la inferencia reproduzca exactamente las condiciones de entrenamiento.

5.3.3. Calibración de umbrales y lógica de *ensemble*

Siguiendo la estrategia de fusión descrita en la Sección 4.4.4, la calibración se realiza en tres pasos. Primero, se reserva un subconjunto de tráfico legítimo

y se evalúan las puntuaciones de cada detector sobre él para seleccionar un umbral τ por cuantil, fijando una tasa de falsos positivos objetivo. Segundo, se normalizan las puntuaciones individuales mediante estandarización *z-score* con las estadísticas de ese mismo conjunto. Tercero, se aplica la combinación lineal con peso $w \in [0, 1]$ que controla la influencia relativa de cada vista.

Todos los artefactos (detectores, parámetros de normalización, umbrales y peso de *ensemble*) se serializan de forma conjunta, permitiendo reconstruir el módulo de detección completo sin repetir el entrenamiento.

En este capítulo se han documentado las decisiones concretas que materializan el diseño de NeuralShield: la composición configurable del pipeline, la construcción de las representaciones sintáctica y semántica, y la implementación de los detectores con su lógica de fusión. El capítulo siguiente evalúa empíricamente el aporte de cada componente y la complementariedad entre ambas vistas.

Capítulo 6

Evaluación experimental

6.1. Objetivos y metodología

El objetivo de esta sección es evaluar empíricamente el enfoque propuesto, verificando si las decisiones de diseño adoptadas, en particular el preprocesamiento estructural y si la representación dual sintáctico-semántica se traducen en mejoras tangibles en la detección de anomalías sobre tráfico HTTP. El interés no reside únicamente en alcanzar altos valores de desempeño, sino en analizar cómo cada componente aporta al desempeño y a la capacidad de generalización del sistema.

La evaluación busca responder dos preguntas fundamentales. En primer lugar, **¿el preprocesamiento estructural aporta valor medible**? Es decir, si al normalizar y etiquetar las solicitudes según los principios de idempotencia y preservación de evidencia, los modelos aprenden distribuciones más coherentes del tráfico legítimo. En segundo lugar, **¿la combinación de ambas perspectivas mejora la robustez del sistema?** Esta última hipótesis se evalúa mediante un esquema de fusión ponderada que integra las puntuaciones de los detectores asociados a cada vista.

Desde el punto de vista metodológico, el diseño experimental se orienta a medir la contribución individual y conjunta de cada capa del sistema. Se incluyen, por tanto, tres niveles de comparación:

1. **Modelos sin preprocesamiento:** aplican los detectores directamente sobre las solicitudes crudas, permitiendo estimar el impacto neto de la etapa de normalización estructural.

2. **Modelos con vistas separadas:** evalúan por separado la detección basada en la representación sintáctica y en la representación semántica, identificando sus fortalezas y limitaciones relativas.

3. **Modelo combinado:** fusiona ambas vistas mediante una combinación

ponderada de scores con el fin de validar la hipótesis de complementariedad entre forma y significado.

6.1.1. Métricas de evaluación

Antes de presentar los experimentos, se definen formalmente las métricas empleadas a lo largo de este capítulo. Todas ellas parten de los cuatro contadores fundamentales de una matriz de confusión binaria: verdaderos positivos (TP), falsos positivos (FP), verdaderos negativos (TN) y falsos negativos (FN).

Precision y Recall. La *precisión* mide la proporción de detecciones positivas que son correctas, mientras que el *recall* mide la proporción de anomalías reales que el modelo identifica:

$$\text{Precision} = \frac{TP}{TP + FP}, \qquad \text{Recall} = \frac{TP}{TP + FN}.$$

F1-score. El *F1-score* es la media armónica de Precision y Recall, y ofrece un indicador único que penaliza el desequilibrio entre ambas:

$$F_1 = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}.$$

Tasa de falsos positivos (FPR). La tasa de falsos positivos (*False Positive Rate*) cuantifica la fracción de muestras legítimas que el modelo clasifica erróneamente como anómalas:

$$\text{FPR} = \frac{FP}{FP + TN}.$$

En contextos operativos de detección de intrusiones, controlar el False Positive Rate (FPR) es crítico: una tasa elevada incrementa el volumen de alertas falsas, eleva la carga de *triage* y puede degradar los tiempos de respuesta ante incidentes reales.

Área bajo la curva ROC (AUC). La curva ROC (*Receiver Operating Characteristic*) representa el Recall en función del FPR al variar el umbral de decisión. El *AUC* (*Area Under the Curve*) resume esta curva en un escalar entre 0 y 1:

$$\text{AUC} = \int_0^1 \text{Recall}(\text{FPR}) \, \mathrm{d}\,\text{FPR}.$$

Un AUC de 1 indica separación perfecta entre clases; un valor de 0.5 equivale a un clasificador aleatorio. A diferencia de métricas puntuales como F1 o Recall, el AUC evalúa el desempeño del modelo a lo largo de *todos* los umbrales posibles, lo que lo convierte en un indicador especialmente adecuado para comparar detectores de anomalías cuyo punto de operación se fija a posteriori.

A lo largo de los experimentos se reportan principalmente **ROC AUC** como indicador global de capacidad discriminativa y **Recall @ 5 % FPR** como indicador operativo que refleja la cobertura de detección bajo una cota de falsos positivos aceptable. El **F1-score** se emplea como métrica complementaria en la evaluación del ensemble.

6.2. Datasets y protocolo de evaluación

Para la evaluación se emplearon tres conjuntos de datos de referencia en detección de anomalías HTTP, seleccionados por su diversidad de origen, escala y tipo de etiquetado.

El conjunto **CSIC-2010** [76] fue generado por el Instituto de Seguridad de la Información del CSIC (España) mediante herramientas automatizadas contra una aplicación de comercio electrónico simulada. Contiene aproximadamente 36 000 solicitudes normales de entrenamiento, 36 000 solicitudes normales de prueba y 25 000 solicitudes anómalas que cubren inyección Structured Query Language (SQL), Cross-Site Scripting (XSS), *path traversal*, desbordamiento de búfer, CRLF y manipulación de parámetros, entre otras categorías. Su origen sintético lo convierte en un entorno controlado, útil para aislar el efecto de cada componente del pipeline sin la variabilidad de un entorno de producción.

El conjunto **SR_BH-2020** [77] fue recopilado por Sureda Riera et al. a partir de un *honeypot* WordPress expuesto a Internet durante doce días, con ModSecurity en modo de detección. Comprende más de 900 000 solicitudes (525 000 normales y 382 000 anómalas) clasificadas con un esquema *multi-label* que asigna a cada petición una o más categorías CAPEC (inyección de código, inyección SQL, *path traversal*, *command injection*, *verb tampering*, entre otras). A efectos de esta evaluación, las etiquetas se reducen a una clasificación binaria normal/anómalo.

Finalmente, **PKDD-2007** [78] proviene del *Discovery Challenge* de ECML/PKDD 2007, con tráfico web real recolectado durante cuatro semanas por la agencia Gemius SA. El conjunto contiene aproximadamente 50 000 solicitudes (80 % normales, 20 % anómalas) etiquetadas en siete categorías de ataque (XSS, inyección SQL, LDAP, XPATH, *path traversal*, ejecución de comandos y SSI). Un 10 % de los ataques está marcado como "fuera de contexto": solicitudes sintácticamente maliciosas dirigidas a entidades incorrectas, lo que añade una dimensión de ruido realista.

La Tabla 6.1 resume las características principales de los tres conjuntos.

Tabla 6.1: Resumen de los conjuntos de datos empleados en la evaluación.

Característica	**CSIC-2010**	**SR_BH-2020**	**PKDD-2007**
Origen del tráfico	Sintético	Real (honeypot)	Real (agencia web)
Solicitudes normales	∼72 000	∼525 000	∼40 000
Solicitudes anómalas	∼25 000	∼382 000	∼10 000
Etiquetado original	Binario	Multi-label (CAPEC)	Multi-clase (7 cat.)

Para garantizar imparcialidad y consistencia, en todos los casos se mantuvo la misma política de particionado:

- **Entrenamiento:** únicamente tráfico etiquetado como normal, utilizado para estimar la distribución base y calibrar los detectores.
- **Validación:** subconjunto de tráfico normal empleado para ajustar hiperparámetros (número de vecinos en LOF, varianza retenida en PCA, umbrales de fusión).
- **Prueba:** mezcla de tráfico normal y anómalo, utilizada exclusivamente para evaluar métricas de desempeño (AUC, FPR, F1, etc.).

6.3. Impacto del preprocesamiento estructural

Esta sección examina de forma sistemática el impacto del preprocesamiento estructural dentro de un entorno controlado, utilizando exclusivamente el conjunto **CSIC-2010** como banco de evaluación. El objetivo es aislar y cuantificar el efecto que tiene esta etapa sobre la coherencia estadística del tráfico normal y la capacidad de los detectores para distinguir solicitudes legítimas de anómalas, independientemente del modelo o la representación empleada.

Para garantizar la validez de la comparación, todas las configuraciones experimentales comparten la misma partición de datos y el mismo esquema de validación. Cada combinación de codificador y detector se evalúa en dos condiciones (con y sin preprocesamiento estructural), manteniendo el resto de los factores constantes. De esta forma, las diferencias observadas dentro de cada par pueden atribuirse exclusivamente al efecto del preprocesamiento sobre la geometría del espacio de representación, mientras que la variación entre pares permite verificar que dicho efecto es consistente a través de paradigmas de representación y detección distintos.

El análisis de resultados se orienta a dos preguntas:

1. **Separabilidad efectiva:** ¿aumenta la distancia entre muestras normales y anómalas, reflejada en una mejora sistemática de las métricas de desempeño?

2. **Robustez cruzada:** ¿el beneficio del preprocesamiento se mantiene al

variar la representación o el detector, o depende de combinaciones particulares?

La comparación simultánea de múltiples modelos permite evaluar la *robustez estructural* del enfoque: si la mejora se mantiene consistente a lo largo de representaciones y detectores distintos, puede concluirse que el preprocesamiento no actúa como una optimización local sino como una transformación de valor general. En ese sentido, esta etapa constituye una validación crítica del diseño del pipeline antes de extender los experimentos a escenarios más complejos o vistas combinadas.

6.3.1. Modelos evaluados

Para evaluar de manera amplia el efecto del preprocesamiento estructural, se seleccionó un conjunto heterogéneo de modelos que combinan distintos enfoques de representación y detección. El objetivo no es comparar arquitecturas entre sí, sino verificar que el impacto del pipeline sea consistente a través de paradigmas con supuestos estadísticos diferentes. Se utiliza exclusivamente **CSIC-2010** como banco de pruebas porque su origen controlado (solicitudes simuladas hacia una aplicación de comercio electrónico) permite atribuir las diferencias observadas al preprocesamiento y no a variaciones de dominio o distribución. De las dieciséis combinaciones posibles entre los cuatro codificadores y los cuatro detectores, se seleccionaron seis configuraciones representativas: al menos una por familia de codificador y una por familia de detector, priorizando las que mejor ilustran los extremos del espectro (representación léxica pura, byte-level, embeddings de dominio, fine-tuning) y los distintos supuestos estadísticos de los detectores (densidad local, aislamiento, covarianza, mezcla gaussiana).

Codificadores y representaciones. Se incluyeron cuatro familias de encoders que cubren un espectro progresivo de expresividad y granularidad:

- **TF-IDF:** representa las solicitudes como bolsas de tokens normalizados. Proporciona una vista puramente léxica y altamente interpretable, adecuada para medir el efecto del preprocesamiento sobre la estabilidad del vocabulario.
- **BGE-small:** embedding denso preentrenado de propósito general, empleado aquí como un punto intermedio entre representaciones léxicas y semánticas. Permite observar cómo las normalizaciones estructurales afectan a modelos que no fueron diseñados específicamente para tráfico HTTP.
- **ByT5:** un modelo que toma el texto byte por byte. Esta aproximación elimina los sesgos del tokenizador, pero vuelve al modelo mucho más sensible a diferencias de codificación o formato. Evaluarlo junto al preprocesamiento permite observar si la estructuración del mensaje y la exposición de evidencia mediante *flags* (limitando las transformaciones a aspectos no

asociados a manifestaciones de ataque) ayudan a que incluso representaciones tan crudas capten regularidades estables del tráfico HTTP.

- **SecBERT:** modelo contextual especializado en texto técnico de ciberseguridad. Su inclusión permite analizar la interacción del preprocesamiento con embeddings semánticos que capturan dependencias funcionales entre componentes del request.

Detectores de anomalías. Para cada tipo de embedding se ensayaron detectores con fundamentos estadísticos complementarios:

- **Isolation Forest (IF):** enfoque basado en árboles que identifica anomalías por su facilidad de aislamiento. Es robusto ante distribuciones no gaussianas y sensible a outliers puntuales.
- **Local Outlier Factor (LOF):** mide densidad relativa entre vecinos cercanos, útil para detectar regiones dispersas o poco pobladas del espacio. El hiperparámetro K (número de vecinos) se fijó en K=100, valor seleccionado mediante búsqueda exhaustiva sobre el conjunto $\{50, 100, 150, 200, 300, 500\}$ en un experimento preliminar de calibración con el subconjunto de validación.
- **Mahalanobis Distance (Maha):** cuantifica desviación estadística respecto a la distribución global, capturando anomalías contextuales bajo supuestos de covarianza elíptica.
- **Gaussian Mixture Model (GMM):** modela la distribución normal como combinación de componentes gaussianos, apropiado cuando el tráfico legítimo presenta modos múltiples.

6.3.2. Resultados cuantitativos

Tabla 6.2: Impacto del preprocesamiento estructural sobre CSIC-2010. Cada combinación de codificador y detector se evalúa con y sin el pipeline de preprocesamiento, manteniendo idéntica la partición de datos. La mejora es consistente en todas las configuraciones, lo que indica que el efecto es independiente del modelo de detección empleado.

Modelo (Encoder + Detector)	**ROC AUC**			**Recall @ 5 % FPR**		
	Sin Prep.	**Con Prep.**	**Δ**	**Sin Prep.**	**Con Prep.**	**Δ (pp)**
TF-IDF + PCA (300) + Mahalanobis	0.504	0.787	**+0.283**	5.81	27.09	**+21.28**
TF-IDF + PCA + Local Outlier Factor	0.686	0.707	**+0.021**	61.87	64.20	**+2.44**
BGE-small + Isolation Forest	0.676	0.717	**+0.041**	15.07	28.48	**+13.41**
ByT5 + Mahalanobis	0.789	0.883	**+0.094**	37.43	44.09	**+6.66**
SecBERT + Mahalanobis	0.735	0.768	**+0.033**	43.69	49.26	**+5.57**
SecBERT (finetuned) + GMM	0.743	0.763	**+0.020**	51.21	52.92	**+1.71**

Los resultados de la Tabla 6.2 muestran una mejora clara y consistente tras aplicar el preprocesamiento. En todos los modelos evaluados, la separación entre tráfico normal y anómalo aumenta, reflejada en incrementos sostenidos de ROC

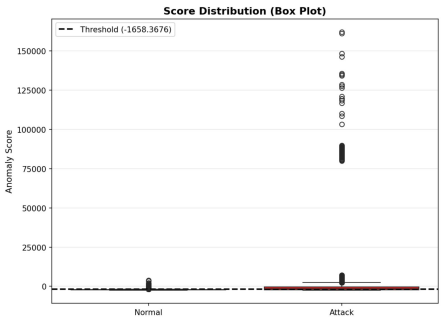
AUC y Recall@5 %FPR.

El efecto es más pronunciado en representaciones léxicas como TF-IDF, donde el pipeline estructura el mensaje, explicita delimitadores y agrega *flags* que registran desviaciones, limitando las transformaciones a aspectos no asociados a manifestaciones de ataque. En modelos densos (BGE, SecBERT, ByT5), la ganancia es menor pero constante, lo que indica que el preprocesamiento aporta estabilidad incluso cuando el encoder ya incorpora contexto.

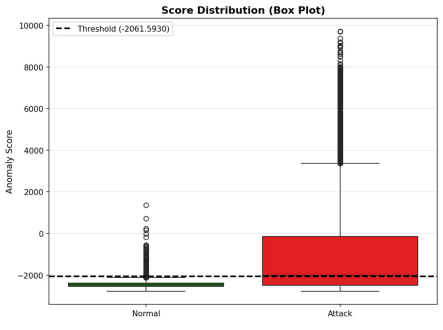
Entre los detectores, la distancia de Mahalanobis obtiene las mayores mejoras, sugiriendo que la estructuración del mensaje y la exposición sistemática de evidencia mediante *flags* contribuyen a distribuciones más gaussianas y con menor dispersión interna. Los métodos basados en densidad (LOF) o aislamiento (IF) también mejoran, aunque de forma más moderada.

6.3.3. Análisis de las distribuciones de scores

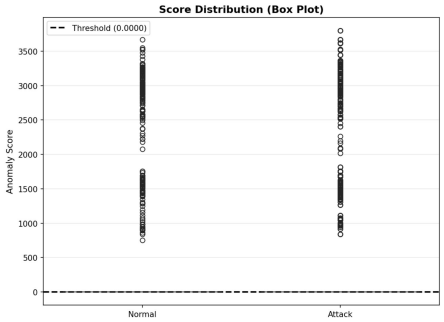
Para profundizar en los efectos observados, se analizaron las distribuciones de puntuaciones de anomalía mediante diagramas de boxplots. Estas visualizaciones permiten observar la estructura interna del espacio de decisión, cómo se separan las muestras normales y anómalas tras aplicar el preprocesamiento, y de qué forma se modifican las varianzas y los umbrales entre ambas clases. En los ejemplos que siguen se comparan dos configuraciones representativas evaluadas con y sin el pipeline estructural.



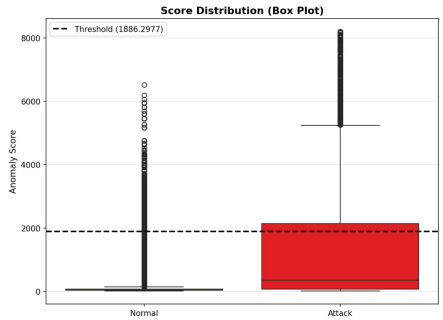
(a) SecBERT + GMM (sin preprocesamiento)



(b) SecBERT + GMM (con preprocesamiento)



(c) TF-IDF + PCA + Mahalanobis (sin preprocesamiento)



(d) TF-IDF + PCA + Mahalanobis (con preprocesamiento)

Figura 6.1: Comparación visual de la distribución de scores entre modelos y condiciones de preprocesamiento.

Los gráficos de la Figura 6.1 evidencian de forma clara que el principal efecto del preprocesamiento estructural es aumentar la separación entre las distribuciones de puntuaciones de solicitudes normales y anómalas. En los escenarios sin preprocesamiento (a, c), ambas clases presentan un rango de valores parcialmente superpuesto, con umbrales poco definidos y alta dispersión en los scores de ataque. Tras aplicar el pipeline estructural (b, d), la distribución de las puntuaciones asociadas a ataques se desplaza hacia valores significativamente más altos, mientras que la de las solicitudes legítimas se mantiene estable. El resultado es una brecha más amplia entre ambas curvas, que facilita la calibración del umbral y reduce la probabilidad de falsos positivos.

En **SecBERT + GMM**, esta separación se refleja en una clara divergencia entre los centroides de ambas clases, lo que sugiere que el preprocesamiento refuerza la capacidad del modelo gaussiano para capturar las regularidades del tráfico legítimo y distinguir desviaciones semánticas.

En **TF-IDF + PCA + Mahalanobis**, el cambio es aún más pronuncia-

do: la distribución de scores de ataque se aleja de la masa principal del tráfico normal, evidenciando que la homogenización léxica y estructural estabiliza el espacio y mejora la discriminación estadística.

En conjunto, las visualizaciones confirman que el aporte del preprocesamiento es geométrico, reorganiza el espacio de representación de modo que los ataques se proyectan más lejos del soporte de la normalidad, ampliando la distancia entre clases.

Dado que el preprocesamiento mejora consistentemente todas las configuraciones, el resto de la evaluación utiliza siempre solicitudes preprocesadas. Para la comparación de vistas y el ensemble se seleccionan las dos combinaciones que mejor representan cada perspectiva del modelo dual definido en la Sección 4.3: **TF-IDF/PCA + LOF** como detector sintáctico, por ser el que mayor Recall@5 %FPR obtiene entre las configuraciones léxicas, y **SecBERT + Mahalanobis** como detector semántico, por su mayor AUC entre los modelos contextuales. Esta elección no busca el mejor modelo absoluto, sino instanciar fielmente la dualidad forma–significado con los representantes más competitivos de cada familia.

6.4. Comparación de vistas sintáctica y semántica

Con el fin de evaluar la contribución específica de cada dimensión de representación, se compararon los dos enfoques seleccionados en la sección anterior: uno sintáctico, centrado en la forma textual y estructura del mensaje (TF-IDF/PCA + LOF), y otro semántico, orientado al contexto funcional del tráfico (SecBERT + Mahalanobis). El objetivo fue determinar no solo cuál obtiene mejor desempeño individual, sino si ambos capturan anomalías de naturaleza diferente y, por tanto, complementarias.

Además de las métricas de desempeño habituales, se emplean tres indicadores de complementariedad calculados sobre las predicciones binarias de ambos modelos al umbral de 5 % FPR:

- **Agreement (%):** proporción de muestras del conjunto de prueba en las que ambos detectores coinciden en su veredicto (ambos clasifican como normal o ambos como anómalo) [79]. Un acuerdo alto indicaría redundancia; un acuerdo moderado sugiere que cada modelo cubre zonas distintas del espacio de decisión.
- **Jaccard:** índice de Jaccard [80] entre los conjuntos de detecciones positivas de cada modelo, es decir, $|D_{\text{LOF}} \cap D_{\text{Maha}}| \,/\, |D_{\text{LOF}} \cup D_{\text{Maha}}|$. Valores bajos indican que los ataques detectados por cada vista se solapan poco.
- **Correlación:** coeficiente de correlación de Pearson [81] entre los vectores de *scores* continuos (antes de umbralizar) de ambos detectores. Mide la

asociación lineal global entre las puntuaciones de anomalía asignadas por cada vista.

Tabla 6.3: Comparación de desempeño y complementariedad entre modelos en los conjuntos **CSIC-2010**, **PKDD-2007** y **SRBH-2020**.

Dataset	Modelo	ROC AUC	Recall @ 5 % FPR	Precision @ 5 % FPR	Agreement (%)	Jaccard	Correlación
CSIC-2010	TF-IDF + PCA + LOF	0.707	64.20	92.95	71.10	0.3623	0.3323
	SecBERT + Mahalanobis	0.768	49.26	90.81			
PKDD-2007	TF-IDF + PCA + LOF	0.630	13.03	71.97	76.55	0.1337	0.1718
	SecBERT + Mahalanobis	0.740	34.12	91.13			
SR_BH-2020	TF-IDF + PCA + LOF	0.728	23.80	81.08	71.99	0.2018	0.2022
	SecBERT + Mahalanobis	0.803	48.18	90.07			

Los resultados comparativos de la Tabla 6.3 muestran un patrón coherente a través de los tres conjuntos de datos. En todos los casos, los modelos sintáctico y semántico exhiben comportamientos complementarios, con niveles moderados de acuerdo y baja correlación entre sus predicciones. Esta independencia relativa sugiere que cada uno detecta anomalías de naturaleza distinta, el primero responde a irregularidades en la estructura y composición textual de la solicitud, mientras que el segundo captura desviaciones de significado o contexto funcional.

La consistencia de estos resultados en **CSIC-2010**, **SR_BH-2020** y **PKDD-2007** refuerza la hipótesis de que la dualidad sintáctico-semántica es una propiedad estructural del tráfico HTTP, no un artefacto de un dataset particular.

6.5. *Ensemble* sintáctico-semántico

Tras validar la complementariedad entre los modelos sintáctico y semántico, el siguiente paso fue evaluar la efectividad de una estrategia de fusión de vistas que combine ambas perspectivas. El objetivo de este ensemble es aprovechar la diversidad de representaciones para incrementar la cobertura de detección sin sacrificar precisión.

Estrategia de combinación

El esquema de ensemble propuesto opera sobre las puntuaciones normalizadas de ambos detectores. Cada modelo produce un vector de *scores de anomalía* $\mathbf{s}_{\text{lof}}$ y $\mathbf{s}_{\text{maha}}$, los cuales se estandarizan mediante z-score sobre el tráfico legítimo:

$$\hat{s}_i = \frac{s_i - \mu_{\text{normal}}}{\sigma_{\text{normal}}}$$

Posteriormente, se combinan de forma convexa según un peso $w \in [0, 1]$:

$$\mathbf{s}_{\text{fused}} = w \cdot \hat{\mathbf{s}}_{\text{lof}} + (1 - w) \cdot \hat{\mathbf{s}}_{\text{maha}}$$

Se evalúan los valores $w \in \{0{,}1,\ 0{,}2,\ \ldots,\ 0{,}9\}$; para cada uno se recalibra el umbral de decisión manteniendo constante el límite de $\mathbf{FPR = 5\,\%}$. El mejor

modelo se selecciona según el máximo **AUC ROC** obtenido sobre el conjunto de validación.

La Tabla 6.4 resume los resultados comparativos entre los detectores individuales y el ensemble en los tres conjuntos de evaluación.

Tabla 6.4: Desempeño comparativo del ensemble frente a los modelos individuales en los tres datasets.

Dataset	Modelo	ROC AUC	Recall @ 5 % FPR	F1-score
CSIC-2010	TF-IDF + LOF	0.707	64.20	75.95
	SecBERT + Mahalanobis	0.768	49.26	63.87
	Ensemble (w=0.75)	**0.861 (+12.1 %)**	**68.82 (+4.6pp)**	**79.19 (+4.3 %)**
PKDD-2007	TF-IDF + LOF	0.630	13.03	22.41
	SecBERT + Mahalanobis	0.740	34.12	49.84
	Ensemble (w=0.2)	**0.780 (+5.4 %)**	**39.26 (+5.1pp)**	**53.77 (+7.9 %)**
SR_BH-2020	TF-IDF + LOF	0.728	23.80	36.80
	SecBERT + Mahalanobis	0.803	48.18	62.77
	Ensemble (w=0.5)	**0.928 (+15.6 %)**	**54.33 (+6.2pp)**	**67.83 (+8.1 %)**

6.6. Discusión de resultados

Los experimentos realizados permiten extraer varias conclusiones sobre el comportamiento y el valor de las distintas etapas del sistema.

6.6.1. Preprocesamiento como transformación de valor general

Los resultados confirman que el preprocesamiento estructural tiene un impacto positivo y transversal. La separación entre tráfico legítimo y anómalo aumenta de forma sistemática en las seis configuraciones evaluadas, sin comprometer la estabilidad de las muestras normales. Este efecto geométrico, observable tanto en las métricas globales como en la distribución de los *scores* (Figura 6.1), demuestra que la normalización estructural no actúa como una simple limpieza textual, sino que reorganiza el espacio de representación de modo que las desviaciones se vuelven más detectables.

La magnitud de la mejora varía según el tipo de representación. Las configuraciones léxicas (TF-IDF) obtienen las ganancias más pronunciadas (+0.283 AUC en el caso de Mahalanobis), lo cual es esperable: el preprocesamiento estabiliza el vocabulario y reduce la varianza debida a codificaciones y formatos heterogéneos. En modelos densos preentrenados (BGE, SecBERT, ByT5), la mejora es menor pero consistente, lo que sugiere que el pipeline aporta señal complementaria incluso cuando el codificador ya incorpora conocimiento contextual.

6.6.2. Complementariedad de las vistas

El análisis comparativo entre las representaciones sintáctica y semántica evidencia que ambas capturan dimensiones complementarias del tráfico HTTP. Mientras la vista sintáctica es más sensible a irregularidades formales y patrones de estructura, la vista semántica identifica anomalías ligadas al contexto funcional de las solicitudes. La baja correlación entre sus puntuaciones (entre 0.17 y 0.33 según el dataset) y el reducido solapamiento de sus detecciones (Jaccard entre 0.13 y 0.36) confirman que estas perspectivas no son redundantes, sino que modelan aspectos distintos de la normalidad.

6.6.3. Eficacia y comportamiento del *ensemble*

Los resultados del *ensemble* sintáctico–semántico validan la hipótesis de complementariedad. Al combinar las puntuaciones de ambos detectores, el sistema mejora el *recall* sin degradar la precisión en los tres conjuntos de datos.

Un aspecto notable es la variabilidad del peso óptimo w entre datasets: 0.75 en CSIC-2010, 0.2 en PKDD-2007 y 0.5 en SR_BH-2020. Esta variación refleja diferencias en la composición de los ataques de cada conjunto. En CSIC-2010, donde predominan ataques con patrones estructurales visibles (inyecciones SQL, XSS), la vista sintáctica recibe mayor peso. En PKDD-2007, cuyo tráfico presenta mayor heterogeneidad y ataques más sutiles, la vista semántica resulta más discriminativa. SR_BH-2020 ocupa una posición intermedia. La ganancia del ensemble es consistente en todos los casos, lo que sugiere que la fusión de vistas actúa como un mecanismo de defensa en profundidad frente a ataques de naturaleza diversa.

6.6.4. Limitaciones

Los resultados deben interpretarse considerando varias limitaciones del diseño experimental.

En primer lugar, los tres conjuntos de datos empleados, aunque ampliamente utilizados en la literatura, presentan sesgos conocidos. CSIC-2010 contiene tráfico simulado que no reproduce la variabilidad de un entorno de producción real. PKDD-2007 y SR_BH-2020 amplían el dominio, pero sus distribuciones de ataque no necesariamente reflejan las amenazas contemporáneas.

En segundo lugar, la evaluación no incluye una comparación directa con otros sistemas de detección publicados sobre los mismos datasets. Si bien las métricas obtenidas son coherentes con los rangos reportados en la literatura para estos conjuntos, una comparación rigurosa requeriría garantizar idénticas condiciones de particionado y preprocesamiento, lo cual excede el alcance de este trabajo orientado a evaluar el aporte de cada componente del pipeline.

En tercer lugar, la selección del peso w del ensemble se realiza por búsqueda exhaustiva sobre el conjunto de validación de cada dataset. En un escenario de despliegue real, donde no se conoce a priori la composición de los ataques,

sería necesario un mecanismo de calibración adaptativa o un valor por defecto robusto.

Finalmente, la evaluación se centra en métricas agregadas (AUC, Recall@FPR fijo, F1) y no incluye un desglose por tipo de ataque. Un análisis más granular permitiría identificar qué familias de amenazas son mejor cubiertas por cada vista y dónde persisten puntos ciegos del sistema.

Capítulo 7

Conclusiones y Trabajo Futuro

7.1. Conclusiones

Este trabajo presentó NeuralShield, un marco experimental para la detección de anomalías en solicitudes HTTP/1.1 que explota de forma explícita la dualidad sintáctico-semántica del protocolo. A continuación se recapitula el grado de cumplimiento de cada objetivo específico planteado en la introducción.

1. **Taxonomía de anomalías estructurales.** Se analizaron las principales familias de vulnerabilidades HTTP documentadas en la literatura y en la investigación ofensiva, y se propuso una taxonomía de cinco familias de anomalías observables en el paquete: duplicación, conflicto y ambigüedad de selección; inyección o presencia anómala de delimitadores estructurales; codificación, normalización y repertorio de caracteres; estructura anómala del *request target*, y valores extremos o perfil estadístico implausible. Esta taxonomía proporcionó la base conceptual para el diseño del preprocesamiento.

2. ***Pipeline* de preprocesamiento estructural.** Se diseñó e implementó un *pipeline* de 12 pasos deterministas, organizado en tres fases (normalización, extracción de *flags* y serialización canónica), que opera bajo principios de idempotencia y no-destructividad. La evaluación experimental confirmó que este preprocesamiento mejora la separabilidad entre tráfico legítimo y anómalo de forma consistente en las seis configuraciones de representación y detector evaluadas.

3. **Representaciones duales.** Sobre el artefacto canónico producido por el *pipeline* se construyeron dos vistas complementarias: una vista sintáctica basada en TF-IDF con reducción PCA, y una vista semántica basada en *embeddings* de SecBERT. Cada vista alimenta un detector de anomalías no

supervisado (LOF y distancia de Mahalanobis, respectivamente) adaptado a sus supuestos estadísticos.

4. **Impacto del preprocesamiento.** La comparación controlada entre modelos con y sin preprocesamiento demostró que la normalización estructural aporta valor medible en todos los casos. Las ganancias son especialmente pronunciadas en representaciones léxicas (hasta +0.283 AUC), pero también consistentes en codificadores densos preentrenados, lo que indica que el *pipeline* genera señal complementaria al conocimiento incorporado por estos modelos.

5. **Complementariedad y fusión.** El análisis de correlación entre las puntuaciones de ambas vistas (entre 0.17 y 0.33 según el dataset) y el bajo solapamiento de sus detecciones confirmaron que las perspectivas sintáctica y semántica capturan dimensiones distintas de la normalidad. El *ensemble* ponderado superó a ambos detectores individuales en los tres conjuntos de datos evaluados, alcanzando valores de AUC de 0.861, 0.780 y 0.928 en CSIC-2010, PKDD-2007 y SR_BH-2020, respectivamente.

En conjunto, los resultados validan la hipótesis central del trabajo: la modelización explícita de la dualidad sintáctico-semántica, combinada con un preprocesamiento estructural guiado por las especificaciones del protocolo, permite construir un sistema de detección de anomalías que opera en régimen *one-class* y cuyas vistas se refuerzan mutuamente al capturar aspectos complementarios del tráfico HTTP.

7.2. Trabajo futuro

Los resultados obtenidos y las limitaciones identificadas durante la evaluación abren varias líneas de investigación.

- **Análisis por tipo de ataque.** La evaluación actual se basa en métricas agregadas. Un desglose por categoría de amenaza (inyección SQL, XSS, *smuggling*, entre otras) permitiría identificar qué familias son mejor cubiertas por cada vista y dónde persisten puntos ciegos del sistema.

- **Calibración adaptativa del peso de fusión.** El peso w del *ensemble* se seleccionó por búsqueda exhaustiva sobre cada conjunto de validación. En un escenario de despliegue real, sería necesario un mecanismo que ajuste este parámetro sin conocimiento previo de la distribución de ataques.

- **Evaluación sobre tráfico contemporáneo.** Los conjuntos de datos empleados, aunque ampliamente utilizados, no reflejan necesariamente las amenazas actuales. La evaluación sobre tráfico HTTP reciente y sobre protocolos HTTP/2 y HTTP/3 constituye un paso natural para validar la generalización del enfoque.

- **Comparación directa con sistemas publicados.** Si bien las métricas obtenidas son coherentes con los rangos reportados en la literatura, una comparación rigurosa bajo condiciones idénticas de particionado y preprocesamiento fortalecería la evidencia sobre el valor del modelo dual.

- **Extensiones del modelo de representación.** La arquitectura dual admite la incorporación de vistas adicionales (por ejemplo, modelos a nivel de byte como ByT5) o el ajuste fino de los codificadores semánticos sobre corpus de tráfico HTTP, lo que podría ampliar la cobertura de detección sin modificar el esquema de fusión.

- **Separación de *flags* como canal independiente de *features*.** En el diseño actual, las *flags* y metadatos estructurales se integran en el texto del artefacto canónico y se vectorizan junto con el contenido original de la solicitud (véase la justificación en la Sección 4.2.3). Una alternativa a explorar consiste en extraer las *flags* como un vector numérico independiente (por ejemplo, codificación *one-hot* o conteo) y concatenarlo al *embedding* textual antes de la etapa de detección. Este esquema permitiría otorgar a las *flags* un peso más directo y controlable, sin depender de cómo el vectorizador las pondera dentro del documento completo. De forma complementaria, podría evaluarse un procesamiento diferenciado donde las *flags* de estructura y formato alimenten un modelo distinto al que procesa el contenido semántico, combinando ambas salidas en la etapa de fusión.

- **Preservación del orden original de cabeceras.** El *pipeline* actual reordena las cabeceras alfabéticamente para reducir la varianza de la representación. Si bien esto simplifica el aprendizaje con conjuntos de datos limitados, descarta información potencialmente útil: el orden original de las cabeceras puede servir como señal de *fingerprinting* del cliente HTTP o como indicador de anomalía cuando difiere del patrón habitual. Con conjuntos de datos de mayor escala, sería interesante evaluar si preservar el orden original mejora la capacidad de detección.

- **Ponderación explícita de *flags* en el espacio de representación.** Actualmente todas las *flags* reciben el mismo tratamiento que cualquier otro token del artefacto. Una extensión natural sería asignar pesos diferenciados a los tokens de tipo *flag* para amplificar su influencia en el vector resultante, bien mediante mecanismos de atención ponderada en el codificador semántico, bien mediante esquemas de *term weighting* ajustados en la vista sintáctica.

Anexo A

Catálogo completo de *flags* y ejemplos

Este anexo presenta el catálogo completo de *flags* emitidas por el *pipeline* de preprocesamiento de NeuralShield, junto con ejemplos representativos que ilustran su activación. Para cada *flag* se indica su nombre, la condición que la dispara (*trigger*), su indicación (lectura de seguridad) y un fragmento de solicitud que la activa. La agrupación sigue las tres fases operativas del *pipeline* descritas en la Sección 5.1.4.

A.1. Fase 1: Saneamiento y segmentación

`UNUSUAL_METHOD`

Trigger: el método de la línea de petición no pertenece al conjunto estándar HTTP (`GET`, `POST`, `PUT`, `DELETE`, `HEAD`, `OPTIONS`, `PATCH`, `CONNECT`, `TRACE`).

Indicación: uso de un método poco frecuente o no estándar, que puede indicar una implementación personalizada o un intento de invocar funcionalidades no previstas.

Ejemplo: `PROPFIND /webdav/ HTTP/1.1`

A.2. Fase 2: Estructuración y normalización de cabeceras

`OBSFOLD`

Trigger: una línea de cabecera comienza con espacio o tabulador, indicando continuación plegada (*obs-fold*).

Indicación: mecanismo obsoleto que puede inducir interpretaciones distintas entre intermediarios y *backend*.

Ejemplo:

```
X-Custom: value
 continuation-value
```

`BADHDRCONT`
: **Trigger:** aparece una continuación de cabecera sin línea base previa.
Indicación: estructura de cabeceras inconsistente con riesgo de parseo divergente.

`BADCRLF`
: **Trigger:** caracteres `CR`/`LF` insertados dentro de una línea de cabecera.
Indicación: presencia anómala de delimitadores, relevante para desincronización y *request smuggling*.
Ejemplo: `X-Injected: value\r\nEvil-Header: payload`

`BADHDRNAME`
: **Trigger:** el nombre de cabecera contiene caracteres fuera de lo permitido por el estándar.
Indicación: riesgo de que distintos intermediarios rechacen o normalicen el campo de forma diferente.
Ejemplo: `X Bad: value` (espacio en el nombre)

`DUPHDR`
: **Trigger:** un nombre de cabecera aparece más de una vez en el bloque.
Indicación: ambigüedad de selección que puede inducir precedencia distinta entre componentes.
Ejemplo:

```
Host: legitimate.com
Host: evil.com
```

`HDRMERGE`
: **Trigger:** cabeceras duplicadas mergeables se consolidan en una sola línea.
Indicación: idea análoga a `DUPHDR`; indica que se ejecutó una consolidación.

`HOPBYHOP`
: **Trigger:** presencia de cabeceras de tipo *hop-by-hop* (`Connection`, `TE`, `Transfer-Encoding`, `Proxy-Authorization`, etc.) en contexto de *request*.
Indicación: metadatos que pueden alterar el reenvío y generar interpretaciones distintas entre componentes.
Ejemplo: `TE: trailers`

`HDRNORM`
: **Trigger:** al menos un nombre de cabecera requirió normalización de forma.

Indicación: variantes de representación útiles como evidencia frente a evasiones basadas en normalización. Se emite como *flag* global en `[HGF]` porque la normalización de *case* se aplica uniformemente a todos los nombres.
Ejemplo: `Accept-Language: en` → `accept-language: en`

`WSPAD`
Trigger: padding anómalo de espacios o tabuladores en valores de cabecera.
Indicación: whitespace no canónico que puede cambiar el resultado de parseo en implementaciones distintas.
Ejemplo: `X-Custom:  padded  value`

A.3. Fase 3: Procesamiento de URL, path y query

`HOSTMISMATCH`
Trigger: el `Host` y el destino derivado de la URL no son coherentes.
Indicación: conflicto de autoridad asociado a *Host Header Injection*.
Ejemplo:

```
GET http://evil.com/path HTTP/1.1
Host: legitimate.com
```

`IDNA`
Trigger: se aplica tratamiento IDN/IDNA al host.
Indicación: dominio internacionalizado con potencial de parseo divergente.

`BADHOST`
Trigger: el host no supera validaciones de formato.
Indicación: host malformado con potencial de bypass.

`MIXEDSCRIPT`
Trigger: un token mezcla alfabetos (latino/cirílico/griego).
Indicación: posible homógrafo para ofuscación visual.
Ejemplo: `Host: p` `ypal.com` (la "a" es cirílica U+0430)

`DOUBLEPCT`
Trigger: patrón de doble *percent-encoding* (`%25..`).
Indicación: capa redundante de codificación usada para inducir decodificaciones distintas.
Ejemplo: `/admin/%252e%252e/config`

`DOTDOT`
Trigger: segmentos `../` en el path.

Indicación: patrón asociado a *path traversal*.
Ejemplo: `GET /app/../../../etc/passwd HTTP/1.1`

`QREPEAT:<key>`
Trigger: una misma clave de parámetro aparece repetida.
Indicación: HPP y ambigüedad de precedencia sobre qué valor consume la aplicación.
Ejemplo: `?role=user&role=admin`

`ANGLE`
Trigger: presencia de `<` o `>` (literal o codificado) en URL, *query* o cabecera.
Indicación: metacarácter frecuente en payloads de inyección HTML/XSS.
Ejemplo: `?q=<script>alert(1)`

`QUOTE`
Trigger: presencia de comillas simples o dobles en contexto estructural.
Indicación: metacarácter asociado a inyección SQL y XSS.

`SEMICOLON`
Trigger: presencia de punto y coma literal en valor de parámetro o URL.
Indicación: delimitador alternativo que puede inducir parseo divergente entre componentes.

`PAREN`
Trigger: presencia de paréntesis en contexto estructural.
Indicación: metacarácter frecuente en llamadas a funciones dentro de payloads de inyección.

`BRACE`
Trigger: presencia de llaves (`{`, `}`) en contexto estructural.
Indicación: metacarácter asociado a plantillas, inyección de expresiones y *template injection*.

`PIPE`
Trigger: presencia de `|` en contexto estructural.
Indicación: metacarácter de *command injection* en entornos shell.

`BACKSLASH`
Trigger: presencia de `\` en contexto estructural.
Indicación: metacarácter de escape, asociado a evasión de filtros y *path traversal* en Windows.

`SPACE`
Trigger: presencia de espacio literal (no codificado como `%20`) en URL o *query*.
Indicación: delimitador no codificado que puede inducir parseo divergente.

`NUL`

Trigger: presencia de byte nulo literal en URL, *query* o cabecera.
Indicación: terminador de cadena en C/C++ que puede truncar valores o inducir comportamientos inesperados.

`QNUL`

Trigger: byte nulo detectado específicamente en valor de parámetro de *query*.
Indicación: análogo a `NUL` pero en contexto de *query*; impacto potencial por truncación entre capas.
Ejemplo: `?tok=ab%00cd`

`MIXEDSCRIPT`

Trigger: un token mezcla alfabetos (latino/cirílico/griego).
Indicación: posible homógrafo para ofuscación visual.
Ejemplo: `Host: p` `ypal.com` (la "a" es cirílica U+0430)

`FULLWIDTH`

Trigger: presencia de caracteres Unicode fullwidth.
Indicación: variantes usadas para evadir reglas ASCII o comparaciones literales.
Ejemplo: `?fw=%EF%BC%A1%EF%BC%A2` (fullwidth)

`CONTROL`

Trigger: caracteres Unicode de control (no imprimibles) en URL o *query*.
Indicación: controles que pueden romper parsing o esconder delimitadores.

`UNICODE_FORMAT`

Trigger: caracteres de formato Unicode (separadores invisibles, marcas bidireccionales, *zero-width*).
Indicación: ambigüedad de renderizado y segmentación con potencial de evasión visual.

`MATH_UNICODE`

Trigger: símbolos matemáticos Unicode en contexto estructural.
Indicación: representación fuera del perfil textual esperado, evidencia de posible ofuscación.

`INVALID_UNICODE`

Trigger: puntos de código inválidos o *noncharacter*.
Indicación: secuencias anómalas que pueden activar fallbacks o tratamientos inconsistentes.

`DOUBLEPCT`

Trigger: patrón de doble *percent-encoding* (`%25..`).
Indicación: capa redundante de codificación usada para inducir decodificaciones distintas.
Ejemplo: `/admin/%252e%252e/config`

`PCTSLASH`

Trigger: secuencia `%2F` (barra codificada) en URL.
Indicación: separador de path codificado, asociado a evasión de reglas de ruta.

`PCTBACKSLASH`

Trigger: secuencia `%5C` (barra invertida codificada).
Indicación: carácter de escape codificado, relevante en entornos Windows.

`PCTSPACE`

Trigger: secuencia `%20` en posición donde un espacio literal sería anómalo.
Indicación: codificación de delimitador que puede alterar el parseo.

`PCTCONTROL`

Trigger: secuencia `%XX` que decodifica a un carácter de control.
Indicación: codificación de bytes de control, potencialmente usada para inyección o evasión.

`PCTNULL`

Trigger: secuencia `%00` (byte nulo codificado).
Indicación: terminador codificado con riesgo de truncación.
Ejemplo: `?file=secret.txt%00.jpg`

`PCTSUSPICIOUS`

Trigger: otras secuencias *percent-encoded* que codifican valores sospechosos.
Indicación: codificación de bytes sensibles no cubiertos por las *flags* específicas anteriores.

`HTMLENT`

Trigger: presencia de entidades HTML (`<`, `&`, `&#x3c;`, etc.) en URL o *query*.
Indicación: capa adicional de codificación textual utilizada para evasión de filtros.
Ejemplo: `?q=&#x3c;script&#x3e;alert(1)`

`QSEMISEP`

Trigger: predomina `;` como separador de parámetros en la *query*.
Indicación: separador alternativo con potencial de parseo divergente entre componentes.

`QRAWSEMI`

Trigger: `;` aparece en la *query* sin adoptar separación plena.
Indicación: delimitador potencialmente ambiguo entre parsers.
Ejemplo: `?id=1;DROP`

`QBARE`

Trigger: parámetro sin signo `=` ni valor.

Indicación: token cuyo parseo varía entre *frameworks*.
Ejemplo: `?debug&verbose`

`QEMPTYVAL`
Trigger: parámetro con `=` pero valor vacío.
Indicación: valor vacío explícito que puede afectar validaciones y control de flujo.
Ejemplo: `?token=&action=login`

`QNONASCII`
Trigger: valor de parámetro contiene caracteres fuera del repertorio ASCII.
Indicación: riesgo de normalizaciones o *encodings* divergentes entre componentes.

`QLONG`
Trigger: valor de parámetro supera un umbral de longitud configurable.
Indicación: valores extremos con impacto potencial en denegación de servicio o evasión por *padding*.

`QARRAY:<key>`
Trigger: nombre de parámetro sigue patrón de arreglo (por ejemplo, `items[]`).
Indicación: sintaxis cuya interpretación depende del *framework*, con riesgo de diferencias de parseo.
Ejemplo: `?items[]=a&items[]=b`

`HOME`
Trigger: el path apunta a raíz (`/`) o estructura equivalente.
Indicación: señal contextual (común y benigna) registrada para caracterización del tráfico.

`MULTIPLESLASH`
Trigger: barras consecutivas en el path (`//`).
Indicación: ruta no canónica con potencial de normalización divergente y bypass de reglas.
Ejemplo: `GET /static//css///style.css HTTP/1.1`

`DOTCUR`
Trigger: segmentos `./` en el path.
Indicación: forma no canónica que puede afectar comparaciones de rutas y validaciones.
Ejemplo: `GET /app/./config HTTP/1.1`

`DOTDOT`
Trigger: segmentos `../` en el path.
Indicación: patrón asociado a *path traversal*.
Ejemplo: `GET /app/../../../etc/passwd HTTP/1.1`

A.4. Composición de las líneas resumen

Las líneas `[HSUM]` y `[QSUM]` condensan propiedades agregadas de sus bloques respectivos. A continuación se detalla cada campo.

Campos de `[HSUM]`

`h_count` Cantidad total de líneas `[HEADER]` observadas. Permite detectar solicitudes con un número inusualmente alto o bajo de cabeceras respecto del perfil aprendido.

`dup_names` Cantidad de nombres de cabecera que aparecen más de una vez. Un valor distinto de cero señala ambigüedad de precedencia, relevante para ataques de tipo *Host Header Injection* y HPP.

`hopbyhop` Cantidad de cabeceras de tipo *hop-by-hop* (`Connection`, `TE`, `Transfer-Encoding`, etc.) presentes. Estas cabeceras son legítimas pero su presencia en cantidad o combinación inusual puede indicar intentos de *request smuggling*.

`bad_names` Cantidad de cabeceras cuyo nombre contiene caracteres fuera de lo permitido por el estándar (RFC 9110). Indica malformación estructural que puede inducir parseo divergente entre componentes.

`total_bytes` Tamaño total en bytes del bloque de cabeceras. Captura el volumen global del bloque, útil para detectar *padding* excesivo o cabeceras sobredimensionadas que podrían buscar desbordamientos de búfer o evasión de límites.

Campos de `[QSUM]`

`count` Cantidad total de parámetros en la *query*. Un número anormalmente alto puede indicar HPP o intentos de desbordamiento.

`QARRAY:<key>` Aparece por cada clave que sigue patrón de arreglo (por ejemplo, `items[]`). Indica sintaxis cuya interpretación depende del *framework* y que puede generar diferencias de parseo entre componentes.

`QREPEAT:<key>` Aparece por cada clave que se repite con valores distintos (por ejemplo, `role=user&role=admin`). Señala ambigüedad de precedencia, vector clásico de HPP.

`QBARE` Presente si algún parámetro carece de signo `=` y valor. Indica tokens cuyo parseo puede variar entre *frameworks* (algunos los interpretan como clave sin valor, otros como valor sin clave).

`QEMPTYVAL` Presente si algún parámetro tiene `=` pero valor vacío. Relevante porque puede afectar validaciones, normalización y control de flujo en la aplicación.

QNUL Presente si algún valor de parámetro contiene bytes nulos (`%00`). Indica presencia de terminadores que pueden truncar cadenas en implementaciones basadas en C/C++ o inducir comportamientos inesperados.

QNONASCII Presente si algún valor de parámetro contiene caracteres fuera del repertorio ASCII. Indica repertorio fuera del perfil base, con riesgo de normalizaciones o *encodings* divergentes entre componentes.

QLONG Presente si algún valor de parámetro supera un umbral de longitud configurable. Indica valores extremos o implausibles con impacto potencial en denegación de servicio o evasión por *padding*.

SEMICOLON Presente si algún valor de parámetro contiene punto y coma literal. Indica presencia de un delimitador alternativo que puede inducir parseo divergente entre componentes que interpretan `;` como separador de parámetros.

Bibliografía

[1] OWASP Foundation, *OWASP Top 10:2021 – The Ten Most Critical Web Application Security Risks*, Accessed: 14 November 2025, 2021. dirección: https://owasp.org/Top10/

[2] R. Fielding, M. Nottingham, J. Reschke et al., "HTTP Semantics," IETF, inf. téc. RFC 9110, 2022. dirección: https://www.rfc-editor.org/rfc/rfc9110

[3] R. Fielding, M. Nottingham, J. Reschke et al., "HTTP/1.1 Message Syntax and Routing," IETF, inf. téc. RFC 9112, 2022. dirección: https://www.rfc-editor.org/rfc/rfc9112

[4] National Institute of Standards and Technology, *CVE-2022-31813 Detail*, https://nvd.nist.gov/vuln/detail/CVE-2022-31813, National Vulnerability Database, 2022. visitado 16 de nov. de 2025.

[5] K. Onarlioglu, *Akamai Mitigates Hop-by-Hop Header Abuse Leading to Request Smuggling*, https://www.akamai.com/blog/security/akamai-mitigates-hop-by-hop-header-abuse, Akamai Security Blog, oct. de 2022. visitado 16 de nov. de 2025.

[6] Internet Assigned Numbers Authority, *Hypertext Transfer Protocol (HTTP) Field Name Registry*, Accessed: 14 November 2025, 2021. dirección: https://www.iana.org/assignments/http-fields

[7] Mozilla Developer Network, *HTTP headers*, Accessed: 14 November 2025, 2025. dirección: https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers

[8] OWASP Foundation, *OWASP Web Security Testing Guide*, Accessed: 14 November 2025, 2024. dirección: https://owasp.org/www-project-web-security-testing-guide/

[9] C. Linhart, A. Klein, R. Heled y S. Orrin, *HTTP Request Smuggling*, Watchfire white paper, Describe una técnica de ataque basada en discrepancias de parsing entre dispositivos HTTP intermedios y servidores de origen, 2005. dirección: https://www.cgisecurity.com/lib/HTTP-Request-Smuggling.pdf

[10] TheHacker Recipes, *HTTP Request Smuggling*, https://legacy.thehacker.recipes/web/config/http-request-smuggling, Accedido: nov. 2025, 2025.

[11] MITRE Corporation, *CWE-444: Inconsistent Interpretation of HTTP Requests ('HTTP Request Smuggling')*, https://www.cvedetails.com/cwe-details/444/Inconsistent-Interpretation-of-HTTP-Requests-HTTP-Request-.html, Accedido: nov. 2025, 2025.

[12] PortSwigger Web Security Academy, *HTTP Request Smuggling*, https://portswigger.net/web-security/request-smuggling, Accedido: nov. 2025, 2025.

[13] J. Kettle, "HTTP Desync Attacks: Request Smuggling Reborn," en *Black Hat USA*, Actualiza y amplía los ataques de HTTP request smuggling en infraestructuras modernas, 2019. dirección: https://portswigger.net/research/http-desync-attacks-request-smuggling-reborn

[14] B. Kallus, P. Anantharaman, M. Locasto y S. W. Smith, "The HTTP Garden: Discovering Parsing Vulnerabilities in HTTP/1.1 Implementations by Differential Fuzzing of Request Streams," *arXiv preprint*, 2024. arXiv: 2405.17737 [cs.CR]. dirección: https://arxiv.org/abs/2405.17737

[15] S. A. Akhavani, B. Jabiyev, B. Kallus, C. Topcuoglu, S. Bratus y E. Kirda, "WAFFLED: Exploiting Parsing Discrepancies to Bypass Web Application Firewalls," *arXiv preprint arXiv:2503.10846*, 2025. DOI: 10.48550/arXiv.2503.10846 dirección: https://arxiv.org/abs/2503.10846

[16] OWASP Foundation, *CRLF Injection*, https://owasp.org/www-community/vulnerabilities/CRLF_Injection, Accedido: nov. 2025, 2025.

[17] OWASP Foundation, *HTTP Response Splitting*, https://owasp.org/www-community/attacks/HTTP_Response_Splitting, Accedido: nov. 2025, 2025.

[18] Acunetix, *What Are CRLF Injection Attacks*, https://www.acunetix.com/websitesecurity/crlf-injection/, Accedido: nov. 2025, 2025.

[19] OWASP Foundation, *A03:2021 – Injection*, Accessed: 14 November 2025, 2021. dirección: https://owasp.org/Top10/A03_2021-Injection/

[20] MDN Web Docs, *Host - HTTP Header*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Host, Accessed: 2025-11-17, 2025.

[21] OWASP Web Security Testing Guide Project, *Testing for Host Header Injection*, https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/17-Testing_for_Host_Header_Injection, Guía de pruebas para identificar vulnerabilidades asociadas a la cabecera Host, 2023.

[22] PortSwigger Web Security Academy, *Host header attacks*, Accessed: 14 November 2025, 2025. dirección: https://portswigger.net/web-security/host-header

[23] Fastly, Inc., *What are HTTP host header attacks?* Accessed: 14 November 2025, 2025. dirección: https://www.fastly.com/learning/security/what-are-http-host-header-attacks

[24] OWASP Foundation, *A10:2021 – Server-Side Request Forgery (SSRF)*, Accessed: 14 November 2025, 2021. dirección: https://owasp.org/Top10/A10_2021-Server-Side_Request_Forgery_%28SSRF%29/

[25] YesWeHack, *HTTP Header Exploitation: Cache-Poisoned Denial of Service*, https://www.yeswehack.com/learn-bug-bounty/http-header-exploitation, Accedido: nov. 2025, 2024.

[26] MDN Web Docs, *Forwarded - HTTP Header*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Forwarded, Accessed: 2025-11-17, 2025.

[27] MDN Web Docs, *X-Forwarded-For*, https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/X-Forwarded-For, Referencia de la cabecera X-Forwarded-For y advertencias de seguridad asociadas, 2025.

[28] OWASP Foundation, *IP Spoofing via HTTP Headers*, https://owasp.org/www-community/pages/attacks/ip_spoofing_via_http_headers, Accedido: nov. 2025, 2025.

[29] Skudonet, *What Is X-Forwarded-For Spoofing?* https://www.skudonet.com/blog/what-is-x-forwarded-for-spoofing/, Explica cómo la cabecera X-Forwarded-For puede manipularse para falsear la dirección IP de origen, 2025.

[30] OWASP Foundation, *Path Traversal*, https://owasp.org/www-community/attacks/Path_Traversal, Accedido: nov. 2025, 2025.

[31] OWASP Foundation, *Testing for HTTP Parameter Pollution*, https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/07-Input_Validation_Testing/04-Testing_for_HTTP_Parameter_Pollution, Accedido: nov. 2025, 2025.

[32] C. Natarajan, *Unveiling HTTP Parameter Pollution (HPP): A Simple Explanation with a Real-Life Example*, https://medium.com/@natarajanck2/unveiling-http-parameter-pollution-hpp-a-simple-explanation-with-a-real-life-example-422dfcac7895, Accedido: nov. 2025, 2024.

[33] CQR Security. "HTTP parameter pollution (HPP) attacks." Describe HPP y menciona consecuencias como robo de datos, ejecución remota y ataques de denegación de servicio; consultado: 17 nov. 2025. dirección: https://cqr.company/web-vulnerabilities/http-parameter-pollution-hpp-attacks/

[34] Invicti Security. "Web Cache Poisoning via JSONP and UTM_ parameter." Caso de estudio donde un mismo parámetro GET, oculto dentro de un parámetro UTM, permite envenenar la caché con contenido controlado por el atacante; consultado: 17 nov. 2025. dirección: https://www.invicti.com/web-application-vulnerabilities/web-cache-poisoning-via-jsonp-and-utm-parameter

[35] Wikipedia contributors, *IDN Homograph Attack*, https://en.wikipedia.org/wiki/IDN_homograph_attack, Accedido: nov. 2025, 2025.

[36] InstaTunnel, *Unicode Normalization Attacks: When "admin" ≠ "admin"*, https://medium.com/@instatunnel/unicode-normalization-attacks-when-admin-admin-32477c36db7f, Accedido: nov. 2025, 2025.

[37] DeepStrike, *What Is a Homoglyph Attack? (IDN Homograph Attack)*, https://deepstrike.io/blog/what-is-a-homoglyph-attack, Accedido: nov. 2025, 2025.

[38] H. Frystyk et al., *HTTP/1.1, Part 1: URIs, Connections, and Message Parsing (Internet-Draft)*, https://www.ietf.org/archive/id/draft-ietf-httpbis-p1-messaging-11.html, Accedido: nov. 2025, 2012.

[39] Netty Project, *Obsolete Line Folding in HTTP Headers Enables CRLF Injection*, https://github.com/netty/netty/issues/10574, Accedido: nov. 2025, 2020.

[40] V. Chandola, A. Banerjee y V. Kumar, "Anomaly Detection: A Survey," *ACM Computing Surveys*, vol. 41, n.º 3, 15:1-15:58, 2009. DOI: 10.1145/1541880.1541882

[41] R. Sommer y V. Paxson, "Outside the Closed World: On Using Machine Learning for Network Intrusion Detection," en *IEEE Symposium on Security and Privacy (S&P)*, IEEE, 2010, págs. 305-316.

[42] B. Schölkopf, J. C. Platt, J. Shawe-Taylor, A. J. Smola y R. C. Williamson, "Estimating the Support of a High-Dimensional Distribution," *Neural Computation*, vol. 13, n.º 7, págs. 1443-1471, 2001.

[43] F. T. Liu, K. M. Ting y Z.-H. Zhou, "Isolation Forest," en *IEEE International Conference on Data Mining (ICDM)*, IEEE, 2008, págs. 413-422.

[44] D. P. Kingma y M. Welling, "Auto-Encoding Variational Bayes," en *International Conference on Learning Representations (ICLR)*, 2014.

[45] M. M. Breunig, H. Kriegel, R. T. Ng y J. Sander, "LOF: Identifying Density-based Local Outliers," en *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, ACM, 2000, págs. 93-104. DOI: 10.1145/342009.335388

[46] P. C. Mahalanobis, "On the Generalized Distance in Statistics," *Proceedings of the National Institute of Sciences of India*, vol. 2, n.º 1, págs. 49-55, 1936.

[47] K. Lee, K. Lee, H. Lee y J. Shin, "A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks," en *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[48] Y. Yu, H. Yan, H. Guan y H. Zhou, "DeepHTTP: Semantics-Structure Model with Attention for Anomalous HTTP Traffic Detection and Pattern Mining," en *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018. arXiv: 1810.12751 [cs.CR]. dirección: https://arxiv.org/abs/1810.12751

[49] C. Kruegel y G. Vigna, "Anomaly Detection of Web-based Attacks," en *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS)*, ACM, 2003, págs. 251-261. DOI: 10.1145/948109.948144

[50] K. Wang y S. J. Stolfo, "Anomalous Payload-Based Network Intrusion Detection," en *Recent Advances in Intrusion Detection (RAID)*, ép. Lecture Notes in Computer Science, vol. 3224, Springer, 2004, págs. 203-222. DOI: 10.1007/978-3-540-30143-1_11

[51] Y. Mirsky, T. Doitshman, Y. Elovici y A. Shabtai, "Kitsune: An Ensemble of Autoencoders for Online Network Intrusion Detection," en *Network and Distributed Systems Security Symposium (NDSS)*, Internet Society, 2018. DOI: 10.14722/ndss.2018.23204

[52] C. Folini y F. Gilliéron, *A New Attempt to Combine the CRS with Machine Learning*, https://coreruleset.org/20210519/a-new-attempt-to-combine-the-crs-with-machine-learning/, OWASP Core Rule Set Blog, 2021.

[53] A. Vaswani et al., "Attention Is All You Need," en *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 30, 2017, págs. 5998-6008.

[54] J. Duma y contributors, *SecBERT: A Pretrained Language Model for Cyber Security Text*, GitHub/Hugging Face, 2020. dirección: https://github.com/jackaduma/SecBERT

[55] J. Devlin, M.-W. Chang, K. Lee y K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," en *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT)*, 2019, págs. 4171-4186.

[56] M. Liberato, "SecBERT: Analyzing Reports Using BERT-like Models," Tesis de mtría., University of Twente, 2022. dirección: https://essay.utwente.nl/93906/

[57] C. Scano, G. Floris, B. Montaruli, L. Demetrio et al., "ModSec-Learn: Boosting ModSecurity with Machine Learning," *arXiv preprint arXiv:2406.13547*, 2024. dirección: https://arxiv.org/abs/2406.13547

[58] L. Demetrio, G. Lagorio, A. Valenza y G. Costa, "ModSec-AdvLearn: Towards Adversarially Trained Machine Learning for Web Application Firewalls," en *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2022, págs. 115-122. DOI: 10.1109/CSR54599.2022.00025

[59] G. Betarte, R. Martínez y Á. Pardo, "Web Application Attacks Detection Using Machine Learning Techniques," en *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2018, págs. 1065-1072. DOI: 10.1109/ICMLA.2018.00169

[60] G. Betarte, R. Martínez y Á. Pardo, "Improving Web Application Firewalls Through Anomaly Detection," en *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2018, págs. 779-784.

[61] N. Montes, G. Betarte, R. Martínez y Á. Pardo, "Web Application Attacks Detection Using Deep Learning," Universidad de la República, inf. téc., 2021, Disponible como MBMP21.pdf en el repositorio institucional.

[62] L. Demetrio, A. Valenza, G. Costa y G. Lagorio, "WAF-A-MoLE: Evading Web Application Firewalls through Adversarial Machine Learning," en *Proceedings of the 35th ACM Symposium on Applied Computing*, 2020, págs. 1745-1752. DOI: 10.1145/3341105.3373928 dirección: https://arxiv.org/abs/2001.01952

[63] X. Zhang, J. Xu y P. Wang, "AutoSpear: Grammar-Guided Automatic SQL Injection Payload Generation for WAF Evaluation," en *Black Hat Asia*, 2023. dirección: https://i.blackhat.com/asia-22/Thursday-Materials/asia-22-Zhang-Challenging-WAF-By-Compress-Automated-SQL-Injection-Payload-Synthesis-Design.pdf

[64] Cloudflare, *Improving the WAF with Machine Learning*, https://blog.cloudflare.com/waf-ml/, 2022.

[65] A. Bocharov y Cloudflare, *Making WAF ML Models Go BRRR: Optimizing the Cloudflare Attack Score*, https://blog.cloudflare.com/making-waf-ml-models-go-brrr-saving-decades-of-processing-time/, 2024.

[66] Amazon Web Services, *AWS WAF Bot Control rule group*, https://docs.aws.amazon.com/waf/latest/developerguide/aws-managed-rule-groups-bot.html, AWS WAF Developer Guide, 2024. visitado 17 de nov. de 2025.

[67] Akamai Technologies, *Applying Machine Learning in Web App and API Protection*, https://www.akamai.com/blog/security/machine-learning-waf, 2022.

[68] Imperva Inc., *Attack Analytics: Machine Learning for Security Event Correlation*, https://www.imperva.com/resources/datasheets/attack-analytics/, 2021.

[69] G. Betarte, Á. Pardo y R. Martínez, "Web Application Attacks Detection Using Machine Learning Techniques," en *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, IEEE, 2018, págs. 1065-1072. DOI: 10.1109/ICMLA.2018.00174 dirección: https://ieeexplore.ieee.org/document/8614199

[70] R. Martínez, "Enhancing Web Application Attack Detection Using Machine Learning," Tesis de mtría., Universidad de la República, Pedeciba Informática, Montevideo, Uruguay, 2019. dirección: https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/29285

[71] N. Montes De Marco, "Web Application Attacks Detection Using Deep Learning," Utiliza modelos de lenguaje profundo sobre tráfico HTTP para detectar ataques, Tesis de mtría., Universidad de la República, Montevideo, Uruguay, 2021. dirección: https://www.colibri.udelar.edu.uy/jspui/handle/20.500.12008/50829

[72] R. Bortolameotti et al., "HeadPrint: Detecting Anomalous Communications through Header-Based Application Fingerprinting," en *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, ACM, 2020. DOI: 10.1145/3341105.3373862

[73] W. Liu et al., "K-BERT: Enabling Language Representation with Knowledge Graph," en *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, págs. 2901-2908. DOI: 10.1609/aaai.v34i03.5681

[74] P. Ke, H. Ji, S. Liu, X. Zhu y M. Huang, "SentiLARE: Sentiment-Aware Language Representation Learning with Linguistic Knowledge," en *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, Association for Computational Linguistics, 2020, págs. 6975-6988. DOI: 10.18653/v1/2020.emnlp-main.567

[75] K. Luo, Y. Xie, S. Zhao y J. Pan, "WADBERT: Dual-channel Web Attack Detection Based on BERT Models," *arXiv preprint arXiv:2601.21893*, 2026.

[76] C. Torrano Giménez, A. Pérez Villegas y G. Álvarez Marañón, *HTTP Dataset CSIC 2010*, Spanish National Research Council (CSIC) – Information Security Institute, 2010. dirección: https://www.tic.itefi.csic.es/dataset/

[77] T. Sureda Riera, J. R. Bermejo Higuera, J. Bermejo Higuera, J. A. Sicilia Montalvo y J. J. Martínez Herráiz, "A new multi-label dataset for Web attacks CAPEC classification using machine learning techniques," *Computers & Security*, vol. 120, pág. 102 788, 2022. DOI: 10.1016/j.cose.2022.102788

[78] C. Raïssi, F. Masseglia y P. Poncelet, "Web Analyzing Traffic Challenge: Description and Results," en *ECML/PKDD 2007 Discovery Challenge*, Warsaw, Poland, 2007. dirección: https://www.lirmm.fr/pkdd2007-challenge/

[79] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and Psychological Measurement*, vol. 20, n.º 1, págs. 37-46, 1960. DOI: 10.1177/001316446002000104

[80] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et du Jura," *Bulletin de la Société Vaudoise des Sciences Naturelles*, vol. 37, págs. 547-579, 1901.

[81] K. Pearson, "Notes on Regression and Inheritance in the Case of Two Parents," *Proceedings of the Royal Society of London*, vol. 58, págs. 240-242, 1895. DOI: 10.1098/rspl.1895.0041