Detección de Ataques Maliciosos con Aprendizaje Automatizado

Andrés Gustavo Perrone

Director: Gustavo Grieco
Codirector: Guillermo Grinblat

31 de Julio del 2018

Resumen

Año a año nuestras vidas dependen cada vez más de la tecnología, y de estar conectados a través de Internet. Progresivamente más y más objetos se conectan a Internet para facilitarnos diferentes funcionalidades. Celulares, autos, heladeras, cuentas bancarias, luces, casas, cámaras, televisores, etc. Estás conexiones nos brindan muchas ventajas y facilidades, pero a su vez aumentan la vulnerabilidad frente a ataques cibernéticos maliciosos. Estos pueden hacer caer sistemas, causar perdidas de datos, robar información privada, mover dinero, y muchos otros problemas.

En los últimos años han surgido nuevos ataques sofisticado, persistentes y con objetivos concretos. Estas nuevas amenazas son denominadas Advanced Persistent Threats (Amenazas Persistentes y Avanzadas), también llamados APT [13, 16]. Estos ataques pueden perseguir objetivos económicos (espionaje), militares (búsquedas de debilidades, revelación de información), técnicos (credenciales, código fuente) o políticos (provocar desestabilización o desorganización, debilitar misiones diplomáticas).

En vista de esta situación, y con el propósito de detectar y protegerse de estos ataques, ya no alcanza con programas tales como sistemas de detección de intrusos o antivirus que utilizan sistemas de reglas para detectar amenazas conocidas, si no que es necesario intentar prever lo desconocido. Día a día se investigan nuevas formas de detectar y prevenir amenazas en la red, generalmente utilizando técnicas de Aprendizaje Automatizado.

Desgraciadamente, la detección de estos ataques altamente dirigidos requiere de grandes cantidades de datos que no están disponibles públicamente. Es por eso que esta tesina se centra en la detección de tráfico malicioso más general.

Pero ¿qué técnicas son realmente efectivas en la práctica?, ¿son realmente implementables?, ¿qué se necesita para utilizarlas con éxito?

En este trabajo muestro los resultados de investigar, probar y analizar varios de los algoritmos publicados, comprobando si son realmente aptos para utilizarse en situaciones reales.

Índice general

Capítulo 1

Introducción

La seguridad cibernética (*cibersecurity* en ingles) es la tecnología, procesos y prácticas diseñadas para proteger redes, computadoras, programas y datos de ataques, daño o acceso no autorizado.

El mayor problema con la seguridad cibernética es la rápida y constante naturaleza evolutiva de los ataques maliciosos. Los atacantes se vuelven cada día más inventivos y hacen que cualquier sistema de seguridad basados en reglas sea insuficiente. Para lograr un buen sistema de seguridad éste debe ser capaz de detectar código malicioso nuevo y nunca vistos. Con tal objetivo en vista, en los últimos años se incrementó la investigación de algoritmos que utilizan técnicas de aprendizaje automatizado. Pero el aprendizaje automatizado no resuelve los problemas por si solo misteriosamente. Es mayormente matemática y un gran poder computacional, y su eficiencia depende totalmente de los datos utilizados para entrenar los modelos.

El mayor potencial para algoritmos de aprendizaje automatizado se encuentra en problemas con distribución de datos mayormente estable en el tiempo, y en el caso de la informática esta condición se cumple escasamente. Surgen entonces tres preguntas importantes:

1. ¿Es posible reproducirlos utilizando fuentes de datos disponibles públicamente?

2. ¿Qué tanto afecta utilizar un modelo de detección de ataque maliciosos en redes de diferentes países?

3. ¿Qué tan efectivos son estos enfoques?

Para responder estas preguntas se estudiaron 4 trabajos con diferentes enfoques. El primero es un detector de dominios generados algorítmicamentes 4.2, el segundo es un detector de URLs maliciosas 4.3, el tercero busca detectar tráfico malicioso en redes cifradas con TLS (*Transport Layer Security* en inglés) sin descifrar los datos 4.4 y finalmente un detector de anomalías 4.5.

Capítulo 2

Tráfico Malicioso

2.1. ¿Qué es?

Tal como su nombre lo indica es aquél tráfico en la red que busca causar cualquier tipo de daño. Este daño puede ser muy diverso y no siempre evidente, encontrando casos donde se busca destruir sistemas, o casos más sutiles pero generalmente más grave en donde el daño es por robo de información sensible.

Hay diversas formas de causar daño. Una de ellas es a través de *malware* o software malicioso, programas o archivos construidos con el objetivo de causar daño en las computadoras. Estos incluyen virus, worms, trojanos y spyware, programas maliciosos capaces de robar, encriptar o borrar datos sensibles, así como alterar funciones y monitorear actividades de usuario. Hay diferentes tipos de *malware*, acorde a su característica.

Los programas maliciosos pueden propagarse a menudo a través de Internet mediante "*drive-by downloads*", que son programas que se descargan automáticamente a los sistemas de los usuarios sin su aprobación o conocimiento. Estos se inician cuando un usuario visita un sitio web malicioso. Los ataques de *pishing* son otro tipo común de entrega de *malware*. Los correos electrónicos disfrazados de mensajes legítimos contienen enlaces maliciosos o los archivos adjuntos pueden entregar el archivo malicioso a usuarios desprevenidos. Ataques de *malware* sofisticados suelen incluir el uso de un servidor de comando y control que permite a los delincuentes comunicarse con los sistemas infectados, robar datos sensibles e incluso controlar remotamente el dispositivo o servidor comprometido.

Los nuevos *malwares* a menudo incluyen nuevas técnicas de evasión y ofuscación que están diseñadas no solo para engañar a los usuarios, sino también a los administradores de seguridad y los productos anti*malware*. Algunas de estas técnicas de evasión dependen de tácticas sencillas, como el uso de proxies para ocultar tráfico malicioso o la dirección IP de origen. Las amenazas más sofisticadas incluyen el *malware* polimórfico, que puede cambiar repetidamente su código para evitar la detección de las herramientas de detección basadas en reglas. Otras técnicas permiten al *malware* detectar cuando se está analizan-

do y retrasar la ejecución hasta después del análisis. Algunos *malware* no usan archivos, éstos solo residen en la RAM del sistema para evitar ser descubierto.

Hay otro tipos de programas similares a *malwares*, como es el caso de los *Adware*. Pueden tener efectos adversos en los usuarios tales como molestos anuncios no deseados y un degradamiento de rendimiento del dispositivo o sistema. El *adware* generalmente no se considera *malware*, ya que no existe una intención malintencionada de dañar a los usuarios o sus sistemas, sin embargo hay casos en que el *adware* puede contener amenazas o contener características de tipo *spyware* que recopilan información, como el historial de navegación e información personal, sin el conocimiento o consentimiento de los usuarios.

Otro tipo de programas similares al *malware* son los *Potentially Unwanted Programs* (PUP), aplicaciones que engañan a los usuarios para instalarlos en sus sistemas, como las barras de herramientas del navegador, pero no ejecutan ninguna función maliciosa una vez que se han instalado. Sin embargo, tal como en los *adwares*, estos pueden usarse como medio para trasmitir *malwares*, usualmente para recompilar información personal sin el consentimiento del usuario.

Otra forma común de causar daño son los DoS (*Denial of Service*) , que como su nombre implica, vuelven inaccesible páginas web o algún otro tipo de servicio online. En un ataque DoS, un perpetrador utiliza una conexión a Internet para explotar una vulnerabilidad de software o inundar un objetivo con solicitudes falsas, normalmente en un intento de agotar los recursos del servidor (RAM, CPU). Existe una variación de estos ataques llamada DDoS (*Distributed Denial of Service*), los cuales difieren en que en estos ataques se lanzan desde múltiples dispositivos conectados que se distribuyen a través de Internet, haciendo mas difícil desviarlos debido a su gran volumen. A diferencia de los ataques DoS de fuente única, los ataques DDoS tienden a enfocarse en la infraestructura de red en un intento de suturarla con enorme volumen de tráfico.

2.2. Consecuencias

Desde comienzos de Internet los usuarios han sufrido las consecuencias de ataques maliciosos, principalmente por medios de virus, causando perdida de sistemas, datos y en muchos casos perdidas económicas, ya sea por consecuencia de pagar el rescate de un *Ransomware* 2.3.5, por perdida de datos importantes o por el robo de información. Si bien estos problemas son suficientemente serio para tomar precauciones, en los últimos años el problema asume aún mayor importancia. Las naciones más avanzadas están bajo constante ataques cibernéticos. Estos ataques tiene objetivos económicos, políticos, militares, sociales y muchos más. Son llamados Advanced Persistent Threat (APT) y su objetivo es robar información, muchas veces sin causar daños a los sistema para así pasar desapercibido. Un APT puede infiltrarse en un sistema y permanecer inactivo e indetectable por meses, esperando el momento adecuado para robar información, y en lo posible, sin ser detectado. Para lograr permanecer en un sistema sin ser detectado en tan largos períodos, el intruso tiene que constantemente

reescribir el código del APT y emplear muchas técnicas sofisticadas de evasión. Algunos APT son tan complejos que requieren de un administrador a tiempo completo. Los APT pueden causar pérdidas millonarias, causar desorganización, manipular elecciones electorales, dar ventajas estratégicas y pueden afectar o incluso causar guerras. El problema ya involucra a estados y no solo a personas particulares.

2.3. Tipos de *malwares*

2.3.1. *Virus*

Es el tipo más común de *malware*, y es aquel programa capaz de ejecutarse por si mismo y propagarse infectando otros archivos o programas. Generalmente, el *malware* inyecta el código malicioso en algún archivo ejecutable que a partir de ese momento pasa a ser portador del virus y una nueva fuente de infección. De esta forma, ante cada ejecución el virus se propaga.

2.3.2. *Worm*

Es un tipo de *malware* que puede auto-replicarse sin un programa host. Suelen propagarse sin ninguna interacción humana o directivas de los autores del mismo.

Un *email worms* se propaga a través de mensajes de emails. Básicamente, llega un email con un adjunto y cuando el usuario lo ejecuta se crean y envían copias del email infectado. Algunos *email worms* pueden ejecutarse sin requerir ninguna intervención del usuario.

Un *internet worm* se propaga directamente a través de conexiones de internet. El *worm* busca puertos abiertos a internet y se envía a si mismo a otros sistemas. Algunos conocidos son *Morris*, *Slammer*, *CodeRed*, *Blaster*, y *Sasse*.

2.3.3. *Trojan*

Es un programa malicioso que está diseñado para aparecer como un programa legítimo. Una vez activados después de la instalación, los troyanos pueden ejecutar sus funciones maliciosas.

Un ejemplo de un troyano reciente es el *Chthonic*, que es una variación del *Zeus* un troyano bancario muy conocido. *Zeus* ganó su popularidad después de que se utilizó en un ataque de robo de credenciales dirigido al Departamento de Transporte de los Estados Unidos. Desde entonces, *Zeus* ha infectado decenas de millones de máquinas y se ha traducido en el robo de cientos de millones de dólares hasta que su creador supuestamente lo dejó en 2011, publicando el código fuente del *malware* en línea. Muchos individuos sirvieron o están cumpliendo condena en la cárcel por su participación en estafas relacionadas con *Zeus*.

Después de infectar su máquina anfitriona, *Chthonic* recopila información del sistema, roba contraseñas guardadas, registra pulsaciones de teclas, concede acceso remoto a sus controladores y también tiene la capacidad de activar

remotamente cámaras y grabadoras de audio. En última instancia, el propósito colectivo de estas características es doble: robar credenciales bancarias en línea y permitir que los atacantes tomen el control de las máquinas víctimas para realizar transacciones financieras fraudulentas.

Un sitio comprometido con éste troyano es `cavallinomotorsport.com`.

Protocol	Length	Host	Info
HTTP	549	cavallinomotorsport.com	GET / HTTP/1.1
HTTP	1395		[TCP Previous segment not captured] Continuation
HTTP	1514		Continuation
HTTP	1276		Continuation
HTTP	1395		Continuation
HTTP	1395		Continuation
HTTP	564		Continuation
HTTP	763	sjxkv.gotdirolhersheck.top	GET /?xXmNd7GYJBfICYI=l3SKfPrfJxzFGMSUb-nJDa9GP0XCRQLPh4SGhKrXCJ-
HTTP	775	sjxkv.gotdirolhersheck.top	GET /index.php?xXmNd7GYJBfICYI=l3SMfPrfJxzFGMSUb-nJDa9GP0XCRQLPh4
HTTP	204		HTTP/1.1 200 OK (text/html)
HTTP	513		HTTP/1.1 200 OK (application/x-shockwave-flash)
HTTP	533	sjxkv.gotdirolhersheck.top	GET /index.php?xXmNd7GYJBfICYI=l3SMfPrfJxzFGMSUb-nJDa9GP0XCRQLPh4
HTTP	537	sjxkv.gotdirolhersheck.top	GET /index.php?xXmNd7GYJBfICYI=l3SMfPrfJxzFGMSUb-nJDa9GP0XCRQLPh4
HTTP	506		HTTP/1.1 200 OK (application/x-msdownload)
HTTP	931	pationare.bit	POST / HTTP/1.0
HTTP	343	pationare.bit	POST /www/ HTTP/1.0

Figura 2.1: Registro de comunicación - Troyan (Wireshark)

El sitio tiene inyectado el un script malicioso que redirige el browser a el sitio `sjxkv.gotdirolhersheck.top`, en donde te ofrecen bajar el troyano disfrazado como alguna actualización, puede ser de flash o de algún otro servicio popular. Al ejecutar el archivo, infecta tu máquina, y aparecerán los siguientes archivos

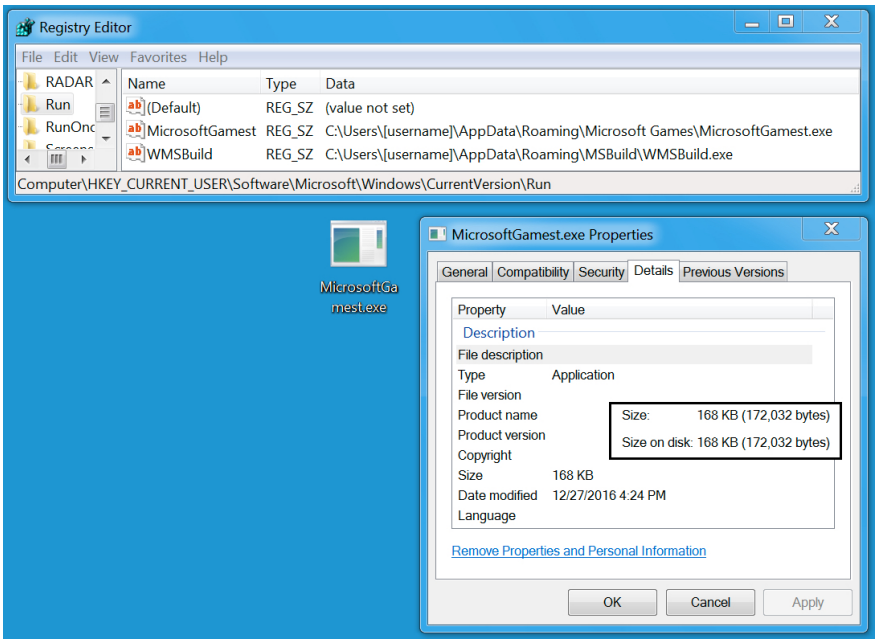


Figura 2.2: Archivos maliciosos - Troyan (Windows)

Una vez infectado se puede observar tráfico saliente hacia el sitio `pationare.bit`.

2.3.4. *Spyware*

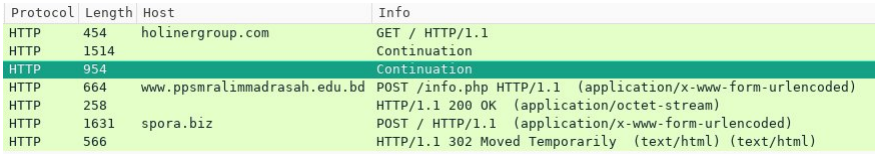
Es un tipo de *malware* que está diseñado para recopilar información y datos sobre los usuarios y observar su actividad sin su conocimiento. Pueden instalarse por si solo a través de internet o por medio de otra aplicación que lo ejecuta sin autorización del usuario. Los *spyware* suelen trabajar a "escondidas", tratando de pasar inadvertido, para que el usuario actúe con normalidad mientras se recompila la información.

Un ejemplo conocido de *spyware es Perfect Keylogger,* que graba los sitios de Internet que se visitan y las contraseñas que se ingresan.

2.3.5. *Ransomware*

Está diseñado para infectar el sistema de un usuario y cifrar los datos. Los ciber-delincuentes luego exigen un pago de "rescate" de la víctima a cambio de descifrar los datos del sistema.

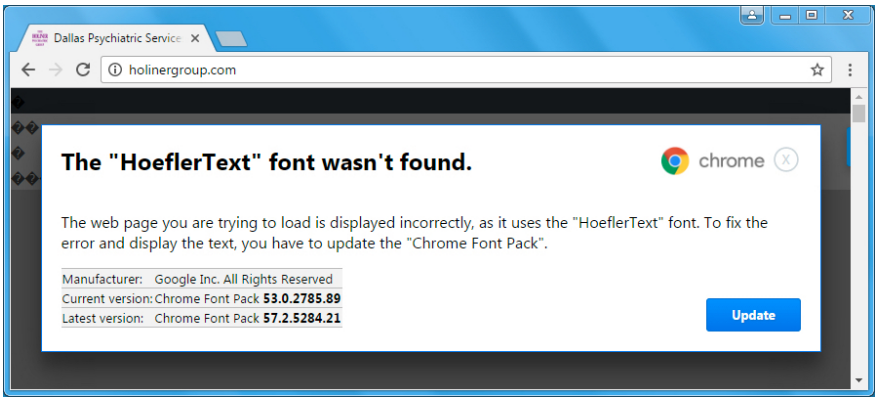
El siguiente es un ejemplo de un caso reciente de *Ransomware*, en febrero del 2015.



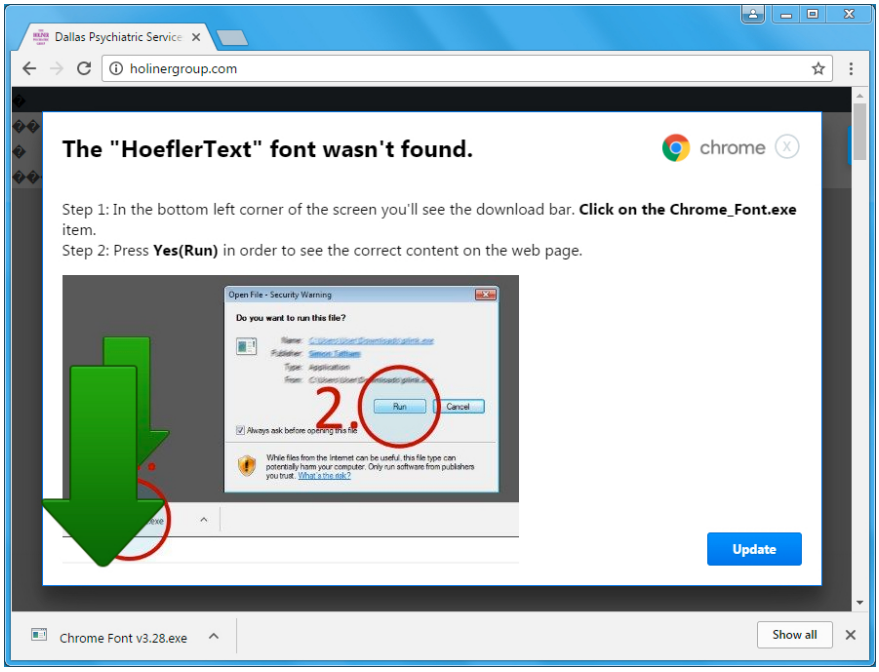
Protocol	Length	Host	Info
HTTP	454	holinergroup.com	GET / HTTP/1.1
HTTP	1514		Continuation
HTTP	954		Continuation
HTTP	664	www.ppsmralimmadrasah.edu.bd	POST /info.php HTTP/1.1 (application/x-www-form-urlencoded)
HTTP	258		HTTP/1.1 200 OK (application/octet-stream)
HTTP	1631	spora.biz	POST / HTTP/1.1 (application/x-www-form-urlencoded)
HTTP	566		HTTP/1.1 302 Moved Temporarily (text/html) (text/html)

Figura 2.3: Registro de comunicación - Ransomware (Wireshark)

El sitio comprometido es `holinergroup.com`, y en este caso es una trampa preparada para aquellos que navegan con el browser de Google, Chrome. Al ingresar al sitio, se ejecuta un script malicioso, el cual abre un popup informando que falta una supuesta fuente de texto



(a) Simulación de aviso de *Chrome* por una fuente de texto no encontrada.



(b) Al bajar y ejecutar el archivo `Chrome Font v3.28.exe` el *malware* logra su objetivo y encripta todos tus archivos

Figura 2.4: Popup Malicioso - Ransomware (Chrome)

El siguiente html aparece en tu escritorio con las instrucciones.

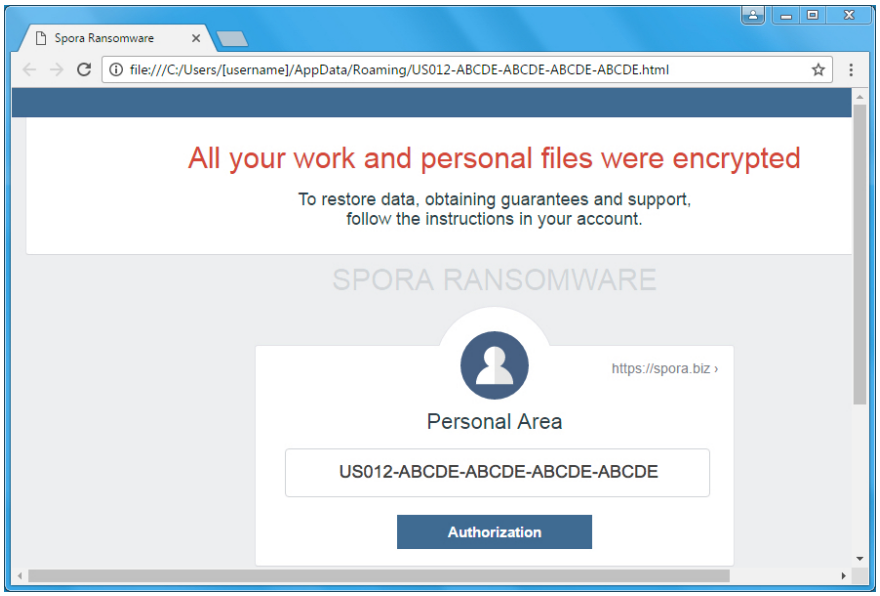


Figura 2.5: Html Malicioso - Ransomware (Chrome)

Y finalmente la página en donde se puede realizar el pago.

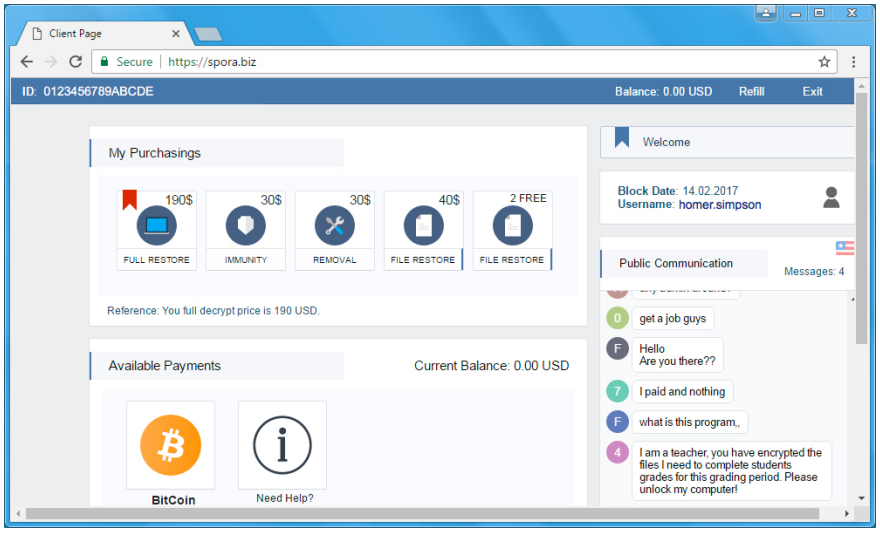


Figura 2.6: Sitio en donde pagar el "Rescate" (Chrome)

Nunca es recomendable hacer el pago, pero la eficacia de estos *malwares* reside en la desesperación de una persona que perdió todo el trabajo de años,

o datos de gran valor personal, en donde realizar un pago de rescate parece insignificante ante la pérdida de la información preciada. La mejor forma de evitar estas situaciones siempre será mantener uno o varios backups actualizados de los datos críticos, aquellos que son irrecuperables.

2.3.6. *Rootkit*

Es un tipo de *malware* diseñado para obtener acceso de administrador al sistema de la víctima. Una vez instalado, el programa da a los delincuentes acceso privilegiado al sistema.

2.3.7. *Remote Access Trojan (RAT)*

Es un programa malicioso que secretamente crea una puerta trasera en un sistema infectado que permite a los actores de la amenaza acceder a distancia sin alertar al usuario o los programas de seguridad del sistema. Suelen instalarse a través de programas instalados por los usuarios o enviados en adjuntos de emails.

2.3.8. *Phishing*

El termino *Phishing* es utilizado para referirse a uno de los métodos mas utilizados por delincuentes cibernéticos para estafar y obtener información confidencial. Esta puede ser una contraseña o información detallada sobre tarjetas de crédito u otra información de la victima.

El delincuente generalmente se hace pasar por una persona o empresa de confianza en una supuesta comunicación oficial, a través de un email o alguna red social, pidiendo al usuario que envíe información, ejecute un archivo en donde se encuentra un *malware* o acceda a un sitio preparado especialmente para engañar a los usuarios.

El siguiente es un ejemplo reportado. Se inicia con un email. En este caso simulando una respuesta de una compra en el sitio `www.hoovers.com`.

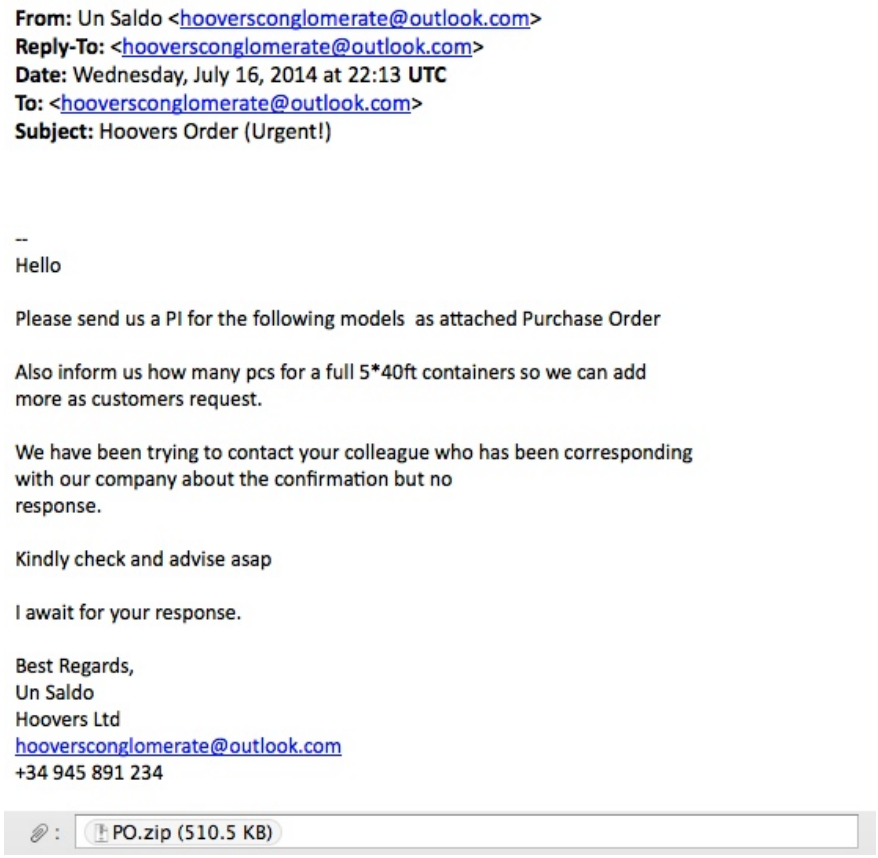
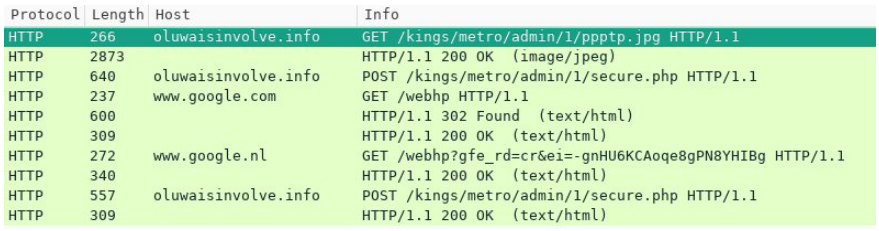


Figura 2.7: Email falso - Phishing

El email menciona que hubo algún problema con la compra y se pide que se responda con cierta información sobre la misma, y amablemente te sugiere que revises el archivo adjunto, donde claramente está el *malware*. Una vez ejecutado, la maquina está infectada y se puede observar tráfico malicioso originado por el *malware*.



Protocol	Length	Host	Info
HTTP	266	oluwaisinvolve.info	GET /kings/metro/admin/1/ppptp.jpg HTTP/1.1
HTTP	2873		HTTP/1.1 200 OK (image/jpeg)
HTTP	640	oluwaisinvolve.info	POST /kings/metro/admin/1/secure.php HTTP/1.1
HTTP	237	www.google.com	GET /webhp HTTP/1.1
HTTP	600		HTTP/1.1 302 Found (text/html)
HTTP	309		HTTP/1.1 200 OK (text/html)
HTTP	272	www.google.nl	GET /webhp?gfe_rd=cr&ei=-gnHU6KCAoqe8gPN8YHIBg HTTP/1.1
HTTP	340		HTTP/1.1 200 OK (text/html)
HTTP	557	oluwaisinvolve.info	POST /kings/metro/admin/1/secure.php HTTP/1.1
HTTP	309		HTTP/1.1 200 OK (text/html)

Figura 2.8: Registro de comunicación - Phishing (Wireshark)

Capítulo 3

Aprendizaje Automatizado

En este trabajo de tesina nos interesa la detección ataques maliciosos utilizando aprendizaje automatizado, y a lo largo del trabajo mencionaremos varias técnicas pertenecientes al área. En este capítulo entonces presentaremos un breve resumen introductorio del área y de las técnicas que vamos a utilizar.

3.1. ¿Qué es?

Aprendizaje Automatizado (*Machine Learning* en inglés) es una rama de Inteligencia Artificial en Ciencias de la Computación que estudia técnicas para lograr que las máquinas "*aprendan*". Para lograrlo se construyen algoritmos que buscan patrones o comportamientos en datos procesados durante el entrenamiento y así lograr predecir análisis, resultados durante la evaluación de nuevos datos.

Los algoritmos suelen agruparse en diferentes categorías según el estilo de aprendizaje. Estas categorías son **aprendizaje supervisado** y **aprendizaje no supervisado**.

3.2. Aprendizaje Supervisado

En esta categoría entran los algoritmos que entrenan utilizando como entrada datos sobre los que ya se conoce el resultado y utilizan esta información para generan una "*función*" que determine el resultado para nuevos y desconocidos datos. En este tipo de algoritmo es posible determinar si el resultado es correcto durante el entrenamiento, y realizar correcciones hasta alcanzar un nivel aceptable de predicción.

Los datos se representan de la siguiente forma:

$$F(X) = Y$$

donde X es una lista de las propiedades más relevantes del dato e Y podría ser un valor o una etiqueta, según el problema.

Los algoritmos de aprendizaje supervisado pueden sub-agruparse según el tipo de problema:

- Problemas de clasificación: Es cuando el conjunto de resultado es discreto, varía entre categorías. Por ejemplo, "válido" y "inválido", o "normal", "amenaza" y "desconocido".
- Problemas de regresión: Es cuando el conjunto de resultado es continuo. Por ejemplo, Edad, Salario o Peso.

Algunos algoritmos conocidos de aprendizaje supervisado son *Naive Bayes* [28] y SVM [18, 21] para problemas de clasificación, *Random Forest* [14, 44] tanto para problemas de clasificación como de regresión, y *Linear Regression* [37] exclusivamente para problemas de regresión.

Los Algoritmos Supervisados son utilizados en distintas áreas. Algunos ejemplos populares pueden encontrarse en la conducción automática de autos (Waymo [20]), en el reconocimientos de personas en fotos (Facebook), en la recomendación de música o películas, dada una preferencia (*Pandora [55]*, *Netflix [38]*) y en la detección de enfermedades ([12, 26]).

3.3. Aprendizaje No Supervisado

En esta categoría entran todos los problemas en donde no se conocen el resultado para los datos de entrada. En estos casos los algoritmos buscan deducir estructuras presente en los datos, extraer reglas generales, u organizar la información según su similitud. No hay un resultado correcto que se pueda utilizar para evaluar el resultado del algoritmo.

Los algoritmos de aprendizaje no supervisado se separan en dos subgrupos

- Clustering: Es cuando el algoritmo intenta obtener agrupaciones inherentes a los datos procesados
- Asociación: El algoritmo busca reglas de asociación entre los datos. Por ejemplo, si para un grupo A, la condición X es verdadera, también la condición Y suele serla.

Algunos algoritmos conocidos de aprendizaje no supervisado son K-Means [7] para problemas de clustering y *Apriori Algorithm* [9] para problema de asociación.

Ejemplos conocidos en donde se utiliza algoritmos de Aprendizaje No Supervisado pueden encontrarse en la clasificación de usuarios según el tipo de películas elegidas (Netflix *[38]*) y en el filtrado de correos no deseados [41, 2].

3.4. Trabajando con Datos

3.4.1. Minería de texto

La minería de texto es el análisis de datos contenidos en texto en lenguaje natural. Muchas veces nuestra fuente de datos proviene de documentos, registros, o algún otro medio de texto no estructurado o poco estructurado del cual nos interesa extraer información de valor para un posterior análisis. De esta forma, la minería de texto involucra estructurar los datos, a veces cortando o agregando información según sea necesario, para luego extraer patrones que nos permitan finalmente analizar y evaluar el resultado.

3.4.2. Bolsas de palabras

En todo trabajo de aprendizaje automatizado, lo principal es conseguir datos y saber extraer la información relevante de ellos. Solo en muy pocos y raros casos los datos se presentan en forma adecuada para utilizar en los algoritmos, en general es necesario transformar los datos.

Cuando se trabaja con formato texto el primer paso es representar el texto en vectores numéricos y la forma más intuitiva es hacerlo mediante lo que se conoce como "*bags of words*" (bolsas de palabras). Esta técnica consiste en contar las ocurrencias de las palabras en el texto, creando vectores de muy alta dimensión, en donde toda posición i del vector representa una palabra del diccionario, y guarda el número de ocurrencias en el texto.

Por ejemplo, si consideramos las frases "El aprendizaje automatizado es una rama de la inteligencia artificial" y "Inteligencia artificial contra la inteligencia humana" podemos representarlas como se muestra en la tabla 3.1. No se tienen en cuenta las palabras "el", "es", "de" y "la" ya que son muy comunes y no aportan mucha información.

	"El aprendizaje automatizado es una rama de la inteligencia artificial"	"Inteligencia artificial contra la inteligencia humana"
aprendizaje	1	0
automatizado	1	0
una	1	0
rama	1	0
inteligencia	1	2
artificial	1	1
contra	0	1
humana	0	1

Tabla 3.1: Ejemplo de "*bags of words*"

Pero contar ocurrencias no siempre es suficiente, ya que por ejemplo puede generar discrepancias en textos de diferentes tamaños, pero que sin embargo

refieran a el mismo tema. Para solucionar este problema potencial es suficiente dividir el número de ocurrencias con el número total de palabras en el texto, obteniendo la frecuencia de cada palabra. Esta técnica es llamada "tf" por " *Term Frecuencies*". También en algunas ocasiones ocurre que algunas palabras son muchas veces más frecuentes que otras, lo que hace que sean menos informativas para el análisis. Para estas situaciones, puede agregarse peso de escala reducida a estas palabras para mejorar el análisis. Esta técnica se llama "tf-idf" por " *Term Frequency times Inverse Document Frequency*".

3.4.3. Desbalanceo de datos

Cuando buscamos datos de entrenamiento para nuestros algoritmos de clasificación no es raro que obtengamos muchos más datos de una de las clase que de las otras. Lo que resulta en un conjunto de datos desbalanceado.

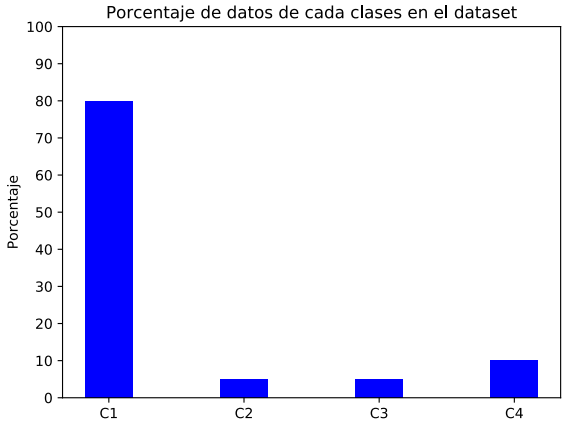


Figura 3.1: Ejemplo de desbalanceo de Datos

Clasificadores entrenados con un conjunto de datos desbalanceado típicamente son más sensibles a la detección de la clase mayoritaria y menos sensibles a las minoritarias. Estos clasificadores terminan muchas veces prediciendo siempre la clase mayoritaria. El problema es aún mayor cuando hay gran diferencia en la variedad de datos para cada categoría.

La solución obvia para este problema es conseguir más datos para las clases minoritarias, pero en la realidad muchas veces conseguir mas datos puede ser extremadamente difícil o imposible. En estos casos la solución es aplicar una de dos técnicas muy similares entre si para reducir el problema causado por el desbalanceo de datos. Estas son Sobremuestreo (*Oversampling*) de las clases minoritarias y Reducción de muestra (*Undersampling*) para la clase mayoritaria.

Las técnicas son sencillas y básicamente consisten en multiplicar las muestras de los datos minoritarios hasta alcanzar la cantidad de la mayoritaria en el caso

de Sobremuestreo, y en dividir las muestras de los datos mayoritarios hasta alcanzar la cantidad de las minoritarias en el caso de Reducción de muestra.

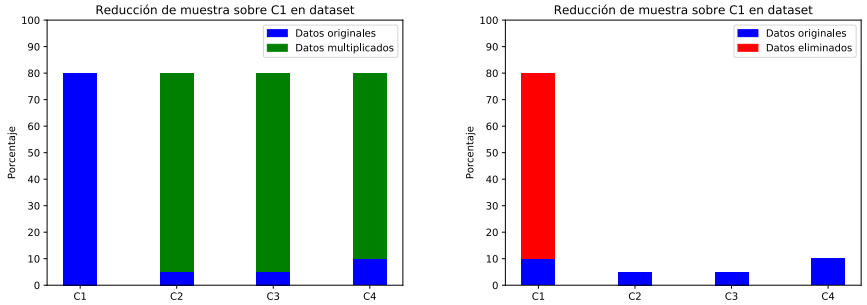


Figura 3.2: Ejemplos de Sobremuestreo y Reducción de muestra

Realizando estas técnicas se reduce considerablemente la diferencia de sensibilidad de detección entre las clases.

3.5. Sobreajuste y Ajuste Insuficiente

Durante el entrenamiento del modelo es muy importante prestar atención y evitar dos posibles situaciones, el sobreajuste (*overfitting*) y el ajuste insuficiente (*underfitting*).

El sobreajuste suele ocurrir cuando nuestro modelo es excesivamente complejo, con muchos parámetros bien característicos de nuestros datos de entrenamiento, produciendo excelentes resultados de evaluación pero muy malos resultados para nuevos y desconocidos datos que difieran incluso levemente de los de entrenamiento. También es muy común que ocurra en modelos con entrenamientos excesivamente largos o entrenado con datos muy raros en donde el modelo "memoriza" características en lugar de "aprender" de ellas.

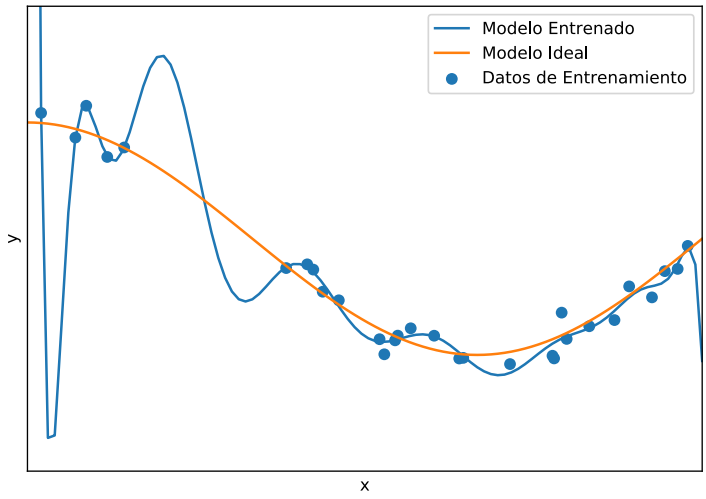


Figura 3.3: Ejemplo de Sobreajuste

Para evitar el sobreajuste existen varias técnicas (Por ejemplo: *cross-validation [43], regularization, early stopping [63], pruning* [42]), pero también se puede controlar realizando controles con datos bien distintos a los de entrenamiento.

La otra situación que puede ocurrir es el ajuste insuficiente, que es cuando el modelo entrenado no captura la tendencia de los datos. Un ejemplo sería un modelo de resultado lineal para datos no lineales.

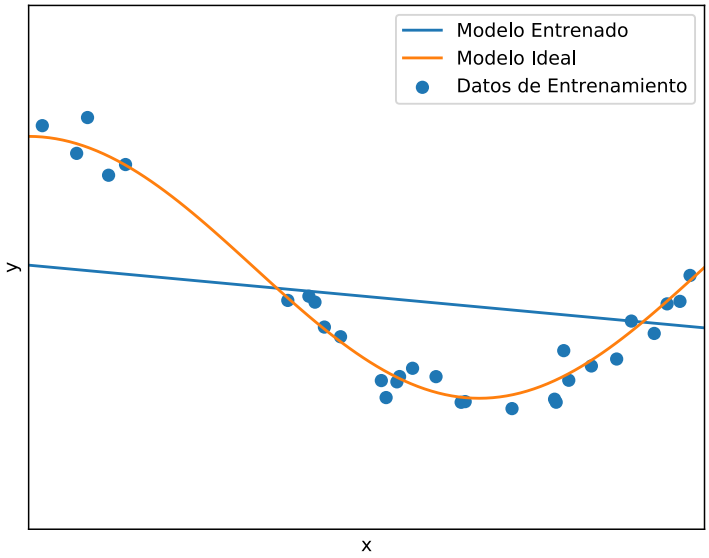


Figura 3.4: Ejemplo de Ajuste Insuficiente

Un modelo con ajuste insuficiente es usualmente fácil de detectar ya que suele dar como resultado evaluaciones muy malas.

3.6. Valores Atípicos

Los valores atípicos (Outliers en inglés) son aquellos datos de entrenamiento cuyo valores distan mucho del resto del conjunto. Estos valores pueden surgir por diferentes razones, ya sea por utilizar diferentes métodos para obtener los datos, por simple error humano de lectura, o pueden ser valores legítimos pero que ocurren solo en ocasiones extremadamente raras.

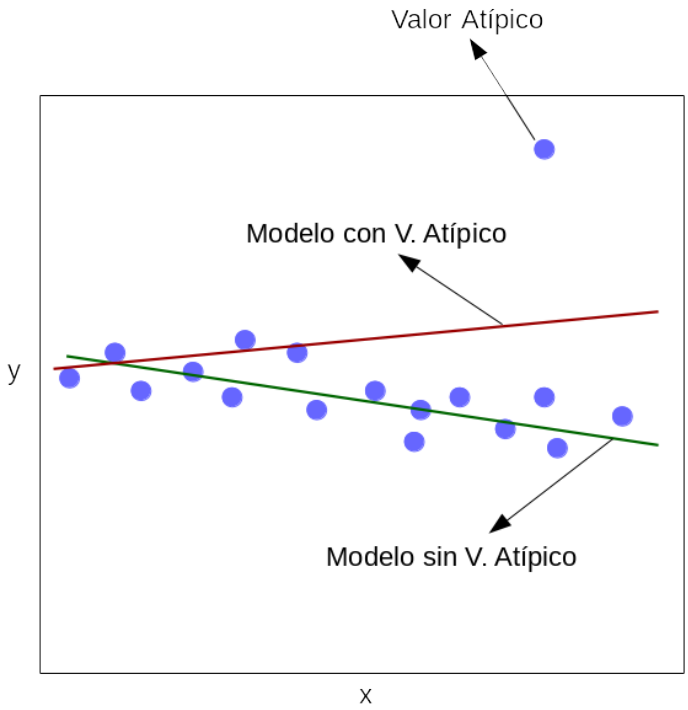


Figura 3.5: Ejemplo Valor Atípico

Estos valores atípicos suelen tener un efecto importante en el modelo, cambiando mucho el resultado según se incluyan o no en el entrenamiento. Cómo tratar estos valores depende mucho del problema. Cuando se tratan de errores humanos u otro tipo de problema que no se espera en el funcionamiento normal, lo mejor suele ser eliminarlos, pero en otros casos el valor atípico puede ser de gran importancia y eliminarlos sería un gran error. También podría ocurrir que tales valores atípicos pertenezcan a un grupo de datos del que no se espera que nuestro modelo pueda interpretar correctamente, en tal caso puede ser conveniente también no tenerlos en cuenta al entrenar.

3.7. Métricas de Evaluación

Una vez entrenado nuestro modelo, es importante saber como medir y comparar su rendimiento. Existen varias formas de hacerlo.

Para problemas de clasificación las métricas más utilizadas son *Classification Accuracy* (Precisión) [35], *Logarithmic Loss* (Pérdida) [35] , *Area Under ROC Curve* (AUC) [6] y *Confusion Matrix* (Matriz de confusión) [35].

Para problemas de Regresión existen otras métricas, tales como *Mean Absolute Error (Error Absoluto)*, *Mean Squared Error* y R^2, pero que no profun-

dizaremos sobre estas ya que no se utilizaron en nuestros experimentos.

3.7.1. Precisión de la clasificación

La precisión es el número de predicciones correctas en relación a todas las predicciones hechas. Es la métrica de evaluación más común para los problemas de clasificación, pero no siempre es la más adecuada para usar. Es principalmente útil cuando la cantidad de datos de cada clase utilizados para entrenar es similar y cuando las predicciones correctas y erróneas tienen la misma importancia. Estas condiciones no suelen darse muy seguido, por lo que muchas veces es necesario realizar algún otro tipo de métrica.

La precisión será un valor entre 0 y 1, siendo en los extremos los mejores resultados. Una precisión mayor a 0.9 es generalmente un buen resultado, aunque dependiendo del problema y de lo que espera el modelo, podría no ser suficientemente buena. Una precisión cercana a 0.5 implica que el modelo se comporta de modo casi en forma aleatoria, siendo un resultado de poca utilidad. Una precisión baja, cercana a 0.1, para problemas donde se clasifique unicamente en dos categorías, puede ser en realidad un buen resultado, ya que si bien el clasificador está dando el resultado opuesto al deseado, basta invertir el resultado o las etiquetas para obtener una precisión de 0.9. Vale aclarar que en una precisión de 0.1 suele indicar algún error en la configuración del modelo. Por otro lado, un valor idéntico de medición que deriva de la precisión es el "*Error*", siendo este (1 - acc). Por ejemplo, si se obtuvo una precisión de 0.9, se puede decir que el modelo tiene un error de 0.1.

3.7.2. Umbral de la clasificación

Muchos modelos de clasificación devuelven una probabilidad. Esta probabilidad es la que se utiliza para determinar a que clase pertenece el dato evaluado. Miremos por ejemplo un clasificador que determina si un email es spam o no. Si al evaluar un email en particular nos devuelve 0.95 es muy probable que sea en efecto spam. Pero ¿qué pasa si nos devuelve una predicción de 0.6? Cuando queremos clasificar entre dos categorías, es necesario definir un umbral de clasificación, el cual es un valor límite que determina a partir de que valor de predicción se considera de una clase o de la otra. Por ejemplo, si para el clasificador de spam definimos un umbral de 0.7, una predicción de 0.6 no lo consideraremos spam, ya que solo las predicciones mayores al umbral son consideradas como spam.

3.7.3. Matriz de Confusión

La matriz de confusión es un tabla útil para clasificadores de dos o más categorías, en donde se cruzan los valores reales con los que el modelo predijo. Por definición, en la posición i, j de la matriz de confusión podemos encontrar el número de observaciones que son de la clase i pero que el modelo predijo como

j. De esta forma, en la diagonal, cuando i es igual a j, podemos encontrar la cantidad de aciertos en la predicción del modelo.

		real		
		clase 1	clase 2	clase 3
predicción	clase 1	5	2	0
	clase 2	3	3	2
	clase 3	0	1	11

Tabla 3.2: Ejemplo de Matriz de Confusión

Por otro lado, de la matriz de confusión de un problema binario se pueden obtener fácilmente cuatro valores importantes; Verdaderos Positivos (VP), Falsos Positivos (FP), Verdaderos Negativos (VN) y Falsos Negativos (FN).

	clase 1	no clase 1
clase 1	VP	FP
no clase 1	FN	VN

Tabla 3.3: Matriz de Confusión

Veamos un ejemplo sencillo para un detector de alguna enfermedad en particular. Supongamos que se utilizó el detector sobre 100 personas, de los cuales 20 tienen la enfermedad a detectar. El detector marcó como enfermas a 25 personas, pero después de analizar cada caso se encuentra la matriz de confusión del cuadro 3.4

n = 100		enfermo	no enfermo
		20	80
enfermo	25	15	10
no enfermo	75	5	70

Tabla 3.4: Ejemplo de Matriz de Confusión

La matriz nos permite observar que de las 25 personas que se detectaron como enfermas solo 15 están realmente enfermas y de las 75 personas detectadas como sano, hay 5 que tienen la enfermedad.

3.7.4. Area Under ROC Curve

Cuando se realizan predicciones con un clasificador sobre un conjunto de datos se obtienen 4 tipos de resultados :

Verdaderos Positivos (VP), Falsos Positivos (FP), Verdaderos Negativos (VN), Falsos Negativos (FN). El conjunto de datos tiene una Población Positiva (PP), *Población Negativa* (PN).

De estos datos se desprenden métricas útiles.

$$Accuracy\,(Precisión) = (1 - Error) = (VP + VN)\,/\,(PP + PN)$$

Mencionado anteriormente, es la probabilidad de una correcta clasificación.

$$Sensitivity\,(Sensibilidad) = VP/\,(VP + FN) = VP/PP$$

La habilidad del modelo para detectar positivo en la población positiva.

$$Specificity\,(Especificidad) = VN/\,(VN + FP) = VN/PN$$

La habilidad del modelo para detectar negativo en la población positiva.

Considerando solo la Precisión o la Sensibilidad se pierde información potencialmente importante. Al considerar los resultados erróneos, así como los correctos se obtiene una mayor comprensión del rendimiento del clasificador. Por ejemplo, en el caso del detector de enfermedades utilizado en el ejemplo de la subsección 3.7.3, el modelo detecta erróneamente como enfermo a 10 personas, pero sin embargo, al tratarse de una enfermedad, es mucho mas costoso el error que comete al marcar 5 personas realmente enfermas como sanas.

Una manera de superar el problema de tener que elegir un corte es comenzar con un umbral de 0, de modo que cada caso se considera positivo. Clasificamos correctamente todos los casos positivos y clasificamos incorrectamente todos los casos negativos. Luego se mueve el umbral sobre cada valor entre 0 y 1, disminuyendo progresivamente el número de FP y aumentando el número de VP.

$VP\,(Sensitivity)$ se puede trazar contra $FP\,(1 - Specificity)$ para cada umbral utilizado. El gráfico resultante se denomina *Receiver Operating Characteristic (ROC)* curve.

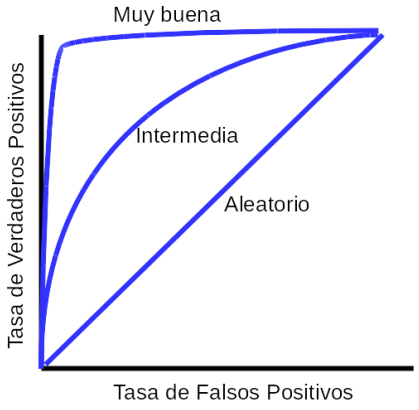


Figura 3.6: Ejemplo de curva ROC

Para un clasificador perfecto, la curva ROC irá directamente hacia arriba por el eje Y y luego por el eje X. Un clasificador sin poder se posicionara en la diagonal, mientras que la mayoría de los clasificadores caen en algún punto intermedio.

La curva ROC se puede utilizar para encontrar un umbral que permita un *tradeoff* entre VP y FP apropiado para el problema concreto que se busque resolver.

La métrica más usada sobre la curva es el (AUC), literalmente el Área debajo de la curva ROC. Un AUC de valor 0,5 significa un modelo de comportamiento aleatorio, mientras que un AOC de valor 1 implica un modelo perfecto. Usualmente el valor está en el medio y se busca maximizarlo.

Capítulo 4

Trabajos Anteriores

4.1. Conceptos previos

4.1.1. URL

URL es una sigla correspondiente a Localizador Uniforme de Recursos (*Uniform Resource Locator*) y determina una dirección a un recurso en Internet. Una URL usualmente consiste en un protocolo, un nombre de dominio y un camino a un archivo o recurso en el servidor. Tiene la siguiente forma

```
protocolo://dominio/camino
```

Algunos protocolos conocidos son *http, https* y *ftp*. El siguiente es un ejemplo de URL

```
https://www.google.com.ar/search?q=machine+learning
```

4.2. Algoritmo de Generación de Dominio (DGA)

Muchos *malwares* requieren conectarse con un servidor remoto para recibir nuevos comandos y para transferir datos robados. Estos *malwares* mantienen una lista de dominios a los que intentan conectarse hasta encontrar un servidor que esté activo. Pero una lista estática de servidores sería un gran problema para el criminal, ya que no pasaría mucho tiempo antes de que los servidores sean rastreados por las autoridades. Para evitar este problema, la mayoría de *malwares* actuales están configurados para que continuamente actualicen su archivos de configuración con nuevos candidatos de servidores C&C (Comando y Control). Pero con este método aún existe el riesgo de que eventualmente, con tiempo y análisis, el servidor usado sea rastreado. En respuesta a esto, los criminales cibernéticos desarrollaron algoritmos que dado una fecha en particular, un tiempo y un valor inicial como semilla, produce una serie de dominios candidatos. Luego testea estos dominios y determina si el servidor C&C está

o no escuchando. Estos algoritmos pueden ser muy variados, después de todo, existen muchas formas de generar cadenas de caracteres aleatorios. El propósito de estos algoritmos es:

- Hacer imposible mantener una lista estática y precisa de servidores de C&C.
- Evadir las tecnologías de filtrado de red basadas en el perímetro.
- Mantener una pequeña pero ágil infraestructura C&C física que solo necesita ser configurada y activada durante cortos períodos de tiempo.
- Proporcionar registros temporales de nombres de dominio para evitar contramedidas reactivas y la aplicación de la ley.
- Permitir que los agentes de *crimeware* (software utilizado para llevar a cabo el crimen) se propaguen y establezcan una gran base de infecciones sin exponer la infraestructura C&C.

Por ejemplo, una forma de fácil de generar dominios en Python es utilizar la librería de cifrado *sha256*. Pasando como entrada la fecha y número, genera una larga cadena de caracteres de las cuales se puede extraer el dominio buscado. Utilizando este simple algoritmo y aplicando varias trasformaciones, podemos generar varias URLs como las siguientes.

Dando los números [0, 1, 2, 3, 4] como raíz y para el día "2017-02-12" se generan estos dominios

- `a1e300f89bb2a3ac2844debd66e8b0967e.cc`
- `b60ae928b57a369ad74f63e385e0fb9fe9.ws`
- `c63371b460a7b29ff52b5c9c61401151cb.to`
- `d78b336bd524071b28133bdb17fc06836c.hk`
- `e583d8757dc1ebfa4bceb4cec4124c5763.cn:443`

De esta forma un *malware* puede generar cientos de dominios en un determinado día, con unas raíces conocidas por el criminal, y probar conectarse con cada una de ellas hasta que encuentra aquella que fue dada de alta en ese mismo día. Una vez lograda la comunicación, el dominio es dado de baja. De estar forma es extremadamente difícil rastrear el dominio, ya que sólo se puede conocer el mismo día en el que se usa y sólo está activo por un corto tiempo. Por supuesto, la mayoría de *malwares* actuales utilizan algoritmos aún más sofisticados.

En vista de esta situación se han investigado varios algoritmos para generar modelos que puedan detectar cuando un dominio es generado por un algoritmo o cuando es legítimo. El detectar los dominios generados no es con el objetivo de rastrear los servidores C&C, ya que estos solo están en funcionamiento un corto tiempo, si no que sirve para descubrir si hay algún *malware* infiltrado en el sistema.

Para lograr distinguir los dominios generados de los legítimos generalmente se suele utilizar varios tipos de medidas de distancias entre los dominios o grupos de estos. Algunas de las utilizadas son:

- *Kullback–Leibler divergence [24]* : Es una medida de diferencia entre dos distribuciones de probabilidad. También suele ser llamada entropía relativa. $KL\_divergence\,(p,k) = \sum_i (p(i) * log(p(i)/q(i)))$, donde p y q son distribuciones de probabilidad discretas, para los i estados existentes. Solamente se define si p y q suman 1 y si $q(i) > 0$ para cualquier i tal que $p(i) > 0$.
- *Shannon entropy [30]* : Mide el grado de incertidumbre que existe en un sistema. $entropy = -\sum_i (p_i * log_2\,(p_i))$donde p_i es la frecuencia con la que el carácter i aparece en el string, para los $i$estados existentes en la distribución.

Jacobs [23] plantea el problema con un modelo de aprendizaje supervisado, en particular como un problema de clasificación, ya que su objetivo es distinguir entre dos clases, "legítimo" y "dga".

El primer y mayor paso de todo problema de clasificación es obtener datos de entrenamiento. Usualmente es bastante complicado obtener datos confiables, pero en este caso al autor le resultó bastante más simple, ya que le bastó con obtener una lista de dominios legítimos y una lista de dominios generados. Para el conjunto de dominios legítimos los obtuvo de la lista que publica Alexa [1] con el top de visitas de URLs y para los dominios generados le bastó con usar alguno de los algoritmos de generación. La lista obtenida de Alexa es muy útil, pero puede no ser suficiente, ya que hay muchos dominios legítimos que no estarán en Alexa, por lo que para mejorar la eficiencia del algoritmo, el autor utilizó también una lista de palabras obtenidas del diccionario.

El siguiente paso en el trabajo estudiado fue la selección de las carasterísticas que representan a los datos. Para esto utilizó 4 carasterísticas:

- La longitud del dominio
- El valor de entropía (*Shannon entropy*)
- El número de coincidencias de ngramas entre el dominio y el conjunto de URLs de alexa
- El número de coincidencias de ngramas entre el dominio y el conjunto de palabras del diccionario

Finalmente para entrenar el modelo el autor utilizó el clasificador *Random Forest* obteniendo muy buenos resultados de evaluación.

Matriz de confusión

	dga	legítimo
dga	84.27 % (225/267)	15.73 % (42/267)
legítimo	0.49 % (33/6746)	99.51 % (6713/6746)

Tabla 4.1: Matriz de confusión - Detector de DGA

4.3. URL Maliciosas

Si bien hay muchos tipos de *malwares* con distintos objetivos y formas, la mayoría de ellos tienen algo en común: requieren de algún usuario distraído que acceda a una URL maliciosa, en donde ya sea por "*drive-by-downloads*", "*pishing*" o algún otro método el delincuente logra su objetivo. La mejor y obvia solución sería extender el conocimiento adecuado para distinguir que URLs deberían evitarse. Pero es imposible pensar que todos los usuarios estarán preparados para detectar que URL tiene aspecto malicioso, sin mencionar que los delincuentes intentan crear URLs lo suficientemente "normales" para engañar incluso a alguien con cierta experiencia.

Veamos algunos ejemplos de posibles URLs maliciosas. Una forma muy común de engañar a usuarios es utilizar una variación de una URL muy usada, como por ejemplo

```
htpp://www.faceb00k.com/login.exe
htpp://www.goog1e.com/getPassword.do
```

Usualmente estos links se ofrecen en mails, o otros medios de comunicación, donde muchos usuarios acceden sin prestar atención al aspecto extraño de las URLs. A simple vista podemos observar indicios que levantan sospechas sobre la validez de estas URLs, siendo uno el reemplazo de letras por números similares en el dominio, junto a la extraña composición final de la URL.

Es por todo esto que el poder tener un modelo que pueda distinguir URLs maliciosas de las inofensivas resulta muy útil. El objetivo es encontrar un modelo que pueda clasificar cada URL accedida entre "buena" y "mala".

Hay varios trabajos que proponen clasificadores capaces de detectar las URLs maliciosas con un bajo porcentaje de error [32, 17], con diferentes métodos, utilizando diferentes datos y características, pero uno ellos [17] además ofrece los datos de entrenamiento, algo esencial para poder reproducir su trabajo. El trabajo en particular presenta un algoritmo de la categoría Aprendizaje Supervisado, del tipo clasificador, y utiliza cerca de 400000 URLs para entrenar, de las cuales 80000 son maliciosas.

Para obtener las características se extrajo tokens de las URLs utilizando una función que sigue el siguiente procesamiento de ejemplo. Tomando como ejemplo la siguiente URL:

```
www.google.com.ar/#q=machine-learning
```

Primero crea secuencias de caracteres cortando la URL en las '/'.

[www.google.com.ar, #q=machine-learning]

Luego, corta cada ítem por '-'.

[www.google.com.ar, #q=machine, learning]

Por último, separa cada ítem por '.'

[www, google, com, ar, #q=machine, learning]

Elimina los 'com', ya que aparecen mucha veces y no aportan nada a la detección. Y finalmente se define el conjunto uniendo los últimos tokens generados con los generados al cortar con '/'

[www.google.com.ar, #q=machine-learning, www, google, ar,
#q=machine, learning]

Utilizando esta función obtiene un "*bag of words*", y utiliza *tf-idf* para representarla como vector. Separa 80 % de los datos para entrenar el modelo y el 20 % restante para validación. El autor asegura una precisión de 98 % .

Y ofrece unos ejemplos de funcionamiento:

wikipedia.com	buena
google.com/search=faizanahad	buena
pakistanifacebookforever.com/getpassword.php/	mala
www.radsport-voggel.de/wp-admin/includes/log.exe	mala
ahrenhei.without-transfer.ru/nethost.exe	mala
www.itidea.it/centroesteticosothys/img/_notes/gum.exe	mala

Tabla 4.2: Ejemplos de funcionamiento - Detector de URLs maliciosas

El resultado es muy bueno, con predicciones acordes a lo que un humano determinaría, al menos sobre el dataset diseñado por el autor.

4.4. Transport Layer Security (TLS)

El cifrado es necesario para proteger la privacidad de los usuarios finales. En una configuración de red, *Transport Layer Security* (TLS) es el dominante para proporcionar cifrado para el tráfico de red. El tráfico legítimo ha visto una rápida adopción del TLS en la última década, con algunos estudios mostrando que aproximadamente el 60 % del tráfico de red utiliza TLS. Desafortunadamente, los *malwares* también han adoptado TLS para asegurar sus comunicaciones. Si bien se estima que solo alrededor del 14 % de los *malwares* utilizan TLS, es lógico creer que este valor crezca acorde pasen los años. Es por esto que es importante

determinar si el tráfico de la red cifrada es benigno o malicioso, y hacerlo de una manera que preserve la integridad del cifrado. Si bien no suele ser práctico, se pueden configurar dispositivos especiales para descifrar el cifrado HTTPS e inspeccionarlo para detectar *malwares*. Sin embargo, existen varias limitaciones a este enfoque, incluyendo una sobrecarga técnica relativamente alta del proceso de descifrado, junto con la amplia gama de problemas burocráticos que puede desencadenar en toda organización. Es por esto que algunos trabajos buscan lograr detectar cuando la tráfico cifrada es maligna sin tener que descifrar los paquetes.

En [8] se presenta un estudio enfocado en el descifrado, siguiendo varias reglas e incluyendo listas de excepción para preservar datos privados, pero no es el enfoque que buscamos en este trabajo.

Por otro lado en [3] se describe un método para la detección de *malwares* en TLS sin necesidad de descifrar el contenido. Los autores recolectaron más de 5.000 muestras maliciosas y varias horas de tráfico de una gran red empresarial, de las cuales extrajeron diferentes tipos de datos. Entre ellos, la cantidad de *bytes* de entrada y de salida, la distribución de *bytes*, los números de puertos usados en la red (443 el más popular), la secuencia de longitudes de paquetes y la información de cabecera de TLS. Utilizando la información extraída los investigadores afirman haber creado un detector con precisión mayor al 93 % en la detección de varias familias importantes de *malware*.

Lamentablemente estos trabajos no ofrecen el algoritmo, y más importante, ninguno ofrece los datos de entrenamiento, haciendo prácticamente imposible su reproducción. Vista esta situación nos propusimos intentar coleccionar datos por nuestra cuenta, y crear un algoritmo clasificador y analizar los resultados.

4.5. Detección de Anomalías - Anagram

Hay dos tipos de enfoques para detectar tráfico malicioso. El primero es analizar y buscar aquella parte que hace malicioso al tráfico, tales como el dominio o alguna cadena particular en el *header* o *payload* del paquete. En general, este método requiere conocer qué es "malo", para así poder detectarlo. Pero hay un segundo enfoque, el cual es pensar el problema de forma inversa, o sea, no se necesita saber que es "malo", alcanza con saber que es "bueno", y todo aquello que no entre en esta categoría se considera malicioso. Este método se suele llamar Detección de Anomalías (*Anomaly Detection*, en inglés) y viene siendo estudiado desde hace muchos años.

Hay muchos trabajos sobre detección de anomalías [27, 10, 19], que utilizan diferentes técnicas y prometen buenos resultados. Entre ellos uno resulta particularmente interesante, llamado "*Anagram: A Content Anomaly Detector Resistant to Mimicry Attack*" [58]. Es un trabajo que fue realizado con el objetivo de crear un modelo que sea capaz de detectar incluso ataques polimórficos, *malwares* bastante avanzados capaces de cambiar y adaptarse a la red a la que pretenden infectar.

Anagram funciona en modo entrenamiento y en modo detección. El enfoque

de *Anagram* para la detección de anomalías en el *payload* (cuerpo de los mensajes, sin considerar la cabecera) de la red utiliza una mezcla de n-gramas de orden superior para modelar y probar el contenido del tráfico de la red. Los n-gramas son básicamente conjuntos formados por las palabras (o caracteres) que ocurren dentro de una ventana de longitud n sobre una cadena de caracteres. Por ejemplo, los 2-gramas sobre la frase "Esto es un ejemplo" serían

[(Esto, es), (es, un), (un, ejemplo)]

En el caso de *Anagram* los n-gramas son generados por ventanas de tamaño arbitrario sobre la secuencia de *bytes*. Durante el entrenamiento el algoritmo observa todos los distintos n-gramas y los almacena en un *Bloom Filter*.

Un *Bloom Filter [5]* es esencialmente un arreglo de m bits en donde cada bit i es seteado en 1 si el hash del valor de entrada, mod m, es igual a i. Un *Bloom Filter* no contiene falsos negativos pero puede contener falsos positivos, ocurriendo cuando hay colisiones entre dos valores. Para reducir los falsos positivos se utiliza un valor alto de m y se utilizan varias funciones hash, considerando que un valor está presente en el *Bloom Filter* solo si todas las funciones hash se cumplen.

Luego durante el modo de detección, cada paquete es puntuado acorde a la cantidad de n-gramas no vistos durante el entrenamiento que contenga

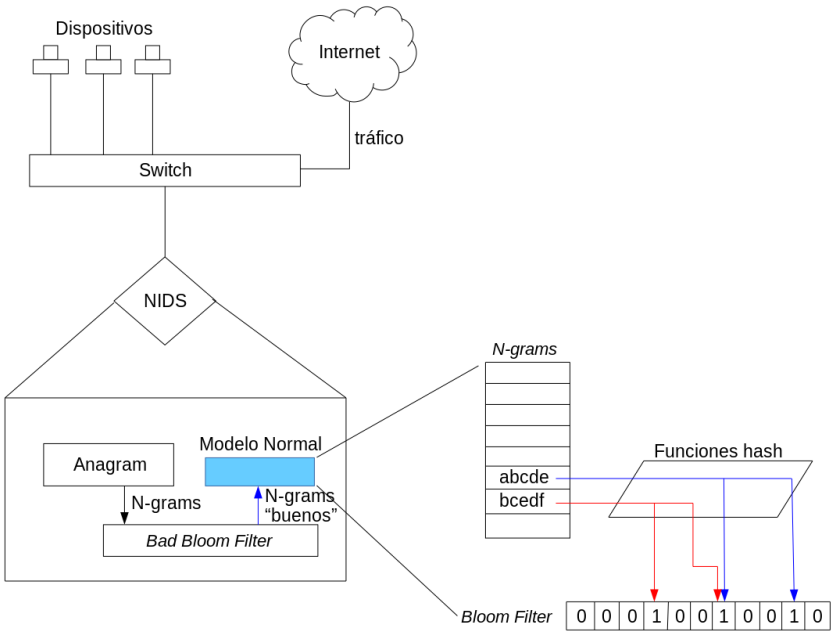
$$Score = N_{new} / \, T \in [0, 1]$$

donde N_{new}es la cantidad de n-gramas nuevos y T la cantidad total de n-gramas en el paquete. Entonces, si el $Score$ es mayor a un límite definido, el paquete se considera sospechoso y se genera una alerta.

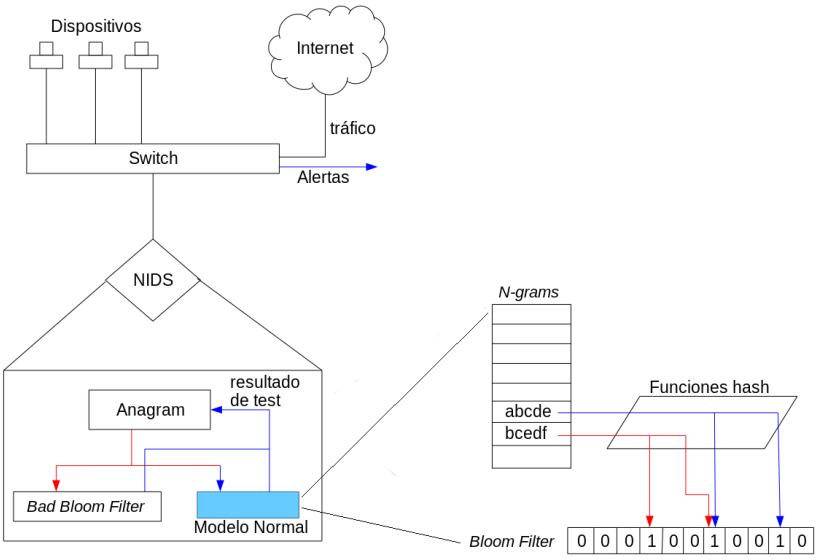
La duración del entrenamiento de *Anagram* variará según el tráfico de la red en la que se utilice, siendo más largo cuando la variedad de tráfico sea mayor. Se considerará entrenado cuando la cantidad de n-gramas nuevos en un período sea menor que un límite determinado. Se supone que durante el entrenamiento la red está libre de ataques. Si ocurre un ataque durante el entrenamiento el modelo lo procesará como normal y Anagram ya no será capaz de detectarlo. Esto hace que el entrenamiento sea muy sensible. Una forma de reducir este riesgo es con un entrenamiento semi-supervisado, en donde se requiere otro *Bloom Filter* que contenga los n-gramas de múltiples ataques conocidos, y entonces al entrenar no se consideren los paquetes que contengan una alta cantidad de n-gramas en este "*Bad Bloom Filter*".

Otra técnica que propone el trabajo para mejorar la detección de ataques que imitan la frecuencia de *bytes* de una red (*Mimicry Attacks*) es utilizar "*Randomization Testing*". La técnica consiste en dividir la cadena de *bytes* en subcadenas generadas de forma aleatoria y entonces testear y evaluar cada subcadena de *bytes* generada en lugar de toda la cadena completa. Como la partición de la cadena es aleatoria y secreta, se supone que el atacante no tendrá forma de saberlo y por lo tanto no podrá modificar el ataque para evitar ser detectado.

En la figura 4.1 podemos observar una representación gráfica del funcionamiento de Anagram.



(b) Modelo de Entrenamiento



(c) Modelo de Testing

Figura 4.1: Representación gráfica de Anagram

En el trabajo se reportan unos muy buenos resultados, sugiriendo que *Anagram* obtuvo menos de 0.1 % de tasa de falsos positivos para una tasa de detección de *malwares* de 100 % en el tráfico de la red en donde realizaron las pruebas. También reporta que *Anagram* fue capaz de detectar los existentes *malwares* que utilizan la técnica *Mimicry Attack*.

Capítulo 5

Overview

5.1. Proceso

La intención de este trabajo fue implementar los trabajos investigados, experimentar con ellos, evaluarlos y analizarlos con el objetivo de ver su utilidad en una red real. De hecho, la idea fue aprovechar el resultado de este trabajo para decidir qué algoritmos integrar en DAPA (Detector de Amenazas Persistentes y Avanzadas). DAPA es un proyecto en desarrollo en el cual se trabajó desde la **Cooperativa de Trabajo Tecso Ltda** con colaboración del **CIFASIS-CONICET**. El objetivo del proyecto es proporcionar un sistema *open-source* capaz de detectar incluso las más avanzadas amenazas combinando detección tradicional basada en reglas con algoritmos de aprendizaje automatizado. En consecuencia, se integraron los algoritmos en una estructura modular acorde a la arquitectura de DAPA.

El procedimiento que se siguió para cada trabajo investigado fue el siguiente: primero se intentó reproducir el trabajo tal cual se describe y analizar su eficacia. Luego, se realizaron distintas medidas de evaluación y pruebas con datos de redes reales. Posteriormente, se experimentó buscando obtener la mejor versión del modelo y finalmente se analizaron los resultados.

Aquellos modelos en los que se obtuvieron buenos resultados se adaptaron e integraron en DAPA, mientras que en los casos donde los resultados no fueron sastifactorios se estudió la razones por las que no fueron exitosos.

5.2. Arquitectura de DAPA

Un sistema de seguridad cibernética (*cibersecurity*) sigue usualmente un proceso similar al de la Figura 5.1.

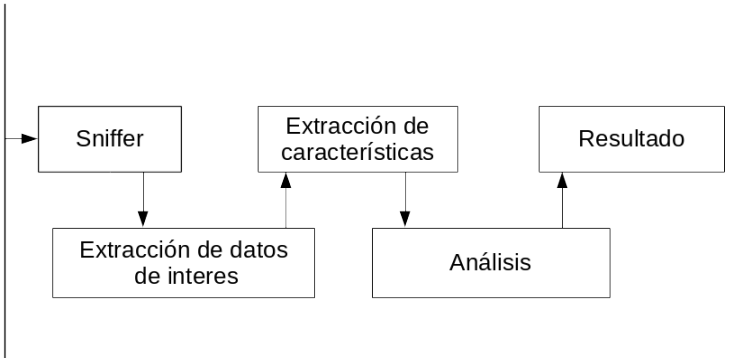


Figura 5.1: Sistema genérico de defensa cibernética

DAPA fue diseñado en dos módulos, DAPA-BASE y DAPA-ML, separando los dos niveles de detección. Por un lado, DAPA-BASE utiliza un Sistema de Detección de Intrusos (IDS *o Intrusion Detection System* en inglés) tradicional basado en reglas, mientras DAPA-ML utiliza los algoritmos de aprendizaje automatizado que resulten aceptables para integrar. DAPA utiliza *Suricata* [39], un motor libre y *open source* capaz de funcionar como IDS y a su vez capturar el tráfico para un posterior análisis mediante los algoritmos de DAPA-ML. *Suricata* detecta amenazas a través de un conjunto de reglas que son continuamente actualizadas por una gran comunidad, que le permiten estar al día con las últimas amenazas conocidas.

Para almacenar y consultar las amenazas detectadas, DAPA utiliza *Solr* [4], una plataforma de almacenamiento y búsqueda de código abierto y de gran alcance. *Solr* permite indexar documentos vía *JSON*, *XML*, *CSV* y realizar consultas o modificaciones a través de HTTP GET. Para ver y analizar las amenazas, DAPA utiliza *Banana* [31], un derivado de *Kibana* [15], capaz de trabajar con *Solr* ofreciendo potentes *dashboards* fácilmente configurables, mostrando la información almacenada en *Solr* en tiempo real.

La arquitectura general de DAPA se presenta en la Figura 5.2.

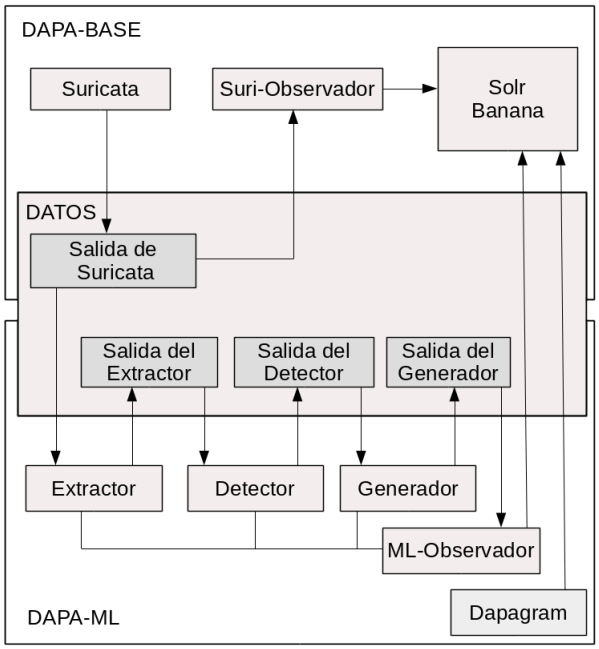


Figura 5.2: Arquitectura General de DAPA

Como se puede ver en el diagrama de la Figura 5.2, DAPA-ML está diseñado en varios módulos conectados entre ellos. El primero es el Extractor, que es quien se encarga de extraer de los pcaps (Subsección 5.3) capturados toda la información requerida para cada detector. Para lograrlo se vale de diferentes scripts en *bash*, utilizando principalmente *tshark* [60], un conocido analizador de protocolo de red. Una vez extraída la información, sigue el turno de el módulo Detector, que es quien corre cada detector que esté configurado. Cada detector creará registros de aquellos paquetes que se consideren sospechosos. Finalmente el módulo Generador se encargará de crear las alertas que serán almacenadas en *Solr*.

Para implementar tanto DAPA como los algoritmos utilizados durante los experimentos, fue necesario seleccionar distintas herramientas. Las más importantes son listadas en la subsección 5.3.

5.3. Herramientas

Todo el trabajo para ésta tesina fue realizado utilizando *software* libre. El principal lenguaje utilizado fue Python, en su versión 3, aprovechando el enorme potencial de las librerías libres *scikit-learn* [46] y Pandas [40]. *Scikit-learn* es una librería específicamente de Aprendizaje Automatizado para Python. Contiene una enorme cantidad de simples y eficientes herramientas para la minería y aná-

lisis de datos, e implementaciones de la mayoría de algoritmos conocidos (*Naive Bayes*, SVM, *Random Forest*, K-Means, etc). Pandas (*Python Data Analysis Library*) es una librería libre que provee estructuras y herramientas de fácil uso para el manejo y análisis de datos.

Durante la recolección de datos fue necesario en muchas ocasiones almacenar tráfico de red y para hacerlo se utilizó principalmente *Suricata* [39], que permite almacenar el tráfico en archivos *pcap [59].* El formato de archivo *pcap* es el formato de archivo de captura más utilizado por las diferentes herramientas de red conocidas. El archivo tiene un encabezado global que contiene información general, seguido de cero o más registros para cada paquete capturado, con el siguiente aspecto:

EG	EP	DP	EP	DP	EP	DP	...

donde EG es Encabezado General, EP es Encabezado de Paquete y DP es Datos de Paquete. Para interpretar los archivos se utilizó principalmente el analizador de protocolo de red *tshark [60].*

Capítulo 6

Experimentos

En esta sección se presentan todos los experimentos y análisis que se hicieron para los trabajos estudiados. En general, como se mencionó en el Sección 5.3, se trabajó en el lenguaje de programación Python en su versión 3, aprovechando el gran potencial de las librerías conocidas *de scikit-learn [46]*, en donde se puede encontrar tanto implementaciones de los algoritmos más conocidos de aprendizaje automatizado como las técnicas de evaluación necesarias.

6.1. Detector de DGA

El objetivo de este detector es identificar aquellos dominios que fueron generados algorítmicamente, diferenciando entre dos clases, "legítimo" y "dga", por lo que se plantea el problema con un modelo de aprendizaje supervisado. Como la idea es, en caso de encontrar buenos resultados, integrarlo en la plataforma de detección DAPA 5.2, se adaptó el algoritmo a la estructura correspondiente. Por un lado se definió una funcionalidad para el modulo Extractor, que se encarga de extraer todos los dominios de las URLs accedidas capturadas en un archivo pcap [59]. Luego se implementó el Detector, que tomando como entrada una lista de dominios, categoriza cada uno como "legítimo" o "dga" según su análisis y reporta de qué paquetes dentro del pcap se obtuvieron estos últimos. Por último, se implementó la funcionalidad correspondiente en el módulo Generador, encargada de obtener la información de interés de los paquetes reportados en el pcap y de generar las alertas correspondientes para luego ser enviadas y almacenadas en *Solr* (Figura *5.2)*. Como veremos a continuación, bastó con reproducir el trabajo estudiado en la Sección 4.2 para alcanzar muy buenos resultados.

6.1.1. Datos

En el caso del detector de dominios generados algorítmicamente, el trabajo presentaba los datos de entrenamiento, siendo estos de muy buena calidad, lo que facilitó en gran manera el desarrollo. Aún así, tanto para testear como para

integrar el detector a la plataforma de detección fue necesario realizar un script que obtenga los dominios de cada comunicación dentro de los pcaps [59]. Esto se logra fácilmente utilizando *tshark* [60] y algunos pocos comandos en bash.

Los datos proporcionados en el trabajo y utilizados durante el entrenamiento son

1. Una lista de los primeros 100000 dominios de la lista proporcionada por Alexa [1]
2. Una lista de más de 400000 palabras del diccionario Inglés
3. Una lista de 2670 dominios generados algorítmicamente

Por otro lado, se capturó semanas de tráfico de la red de la oficina de la *Cooperativa de Trabajo Tecso Lta* en Rosario para realizar pruebas en un ambiente real.

6.1.2. Implementación

Reproducir el algoritmo fue una tarea sencilla, ya que es presentado de forma ordenada y proporcionando los datos necesarios para entrenar.

Primero cargamos los dominios de Alexa y los clasificamos como "legítimo". Seguimos cargando los dominios generados y los clasificamos como "dga". Juntamos ambos grupos, calculamos la longitud de cada dominio y descartamos aquellos dominios de longitud menor a 6, ya que de menor longitud son muy aleatorios y muy difícil de clasificar como "legítimo" o "dga".

A continuación, como segunda característica, calculamos la entropía para cada dato. La siguiente característica calculada es la coincidencia de n-gramas con el conjunto de dominios de Alexa. Y la última característica a utilizar es la coincidencia de n-gramas con el conjunto de palabras de diccionario.

Aquellos dominios que son legítimos pero que no alcanzan a 2 o 3 coincidencias (muy pocos dominios de 3 o 4 caracteres de largo) son considerados URL raras y actuarían como *outliers*, por lo que no se utilizan para entrenar el modelo. Estos fueron

Ya con los datos procesados, se arman los vectores de características y sus etiquetas. Dividimos los datos en dos, 80 % para entrenar y reservamos el 20 % para testear el modelo entrenado. También se realizaron testing con datos obtenidos de la red de la oficina de la *Cooperativa de Trabajo Tecso Lta* en Rosario, con el objetivo de analizar la cantidad de falsos positivos en una red considerablemente "limpia".

Finalmente utilizamos el algoritmo *Random Forest* para entrenar el clasificador. Se utilizó la selección de hiperparámetros definida por defecto en la librería de *scikit-learn.*

6.1.3. Resultados

Con el modelo ya entrenado lo validamos con el 20 % de datos reservados y calculamos la precisión, obteniendo 0.989 .

La matriz de confusión resultante se muestra en la Tabla 6.1.

	dga	legítimo
dga	82.692 %	17.307 %
legítimo	0.379 %	99.621 %

Tabla 6.1: Matriz de confusión - Detector de DGA

Y la curva ROC en la Figura 6.1

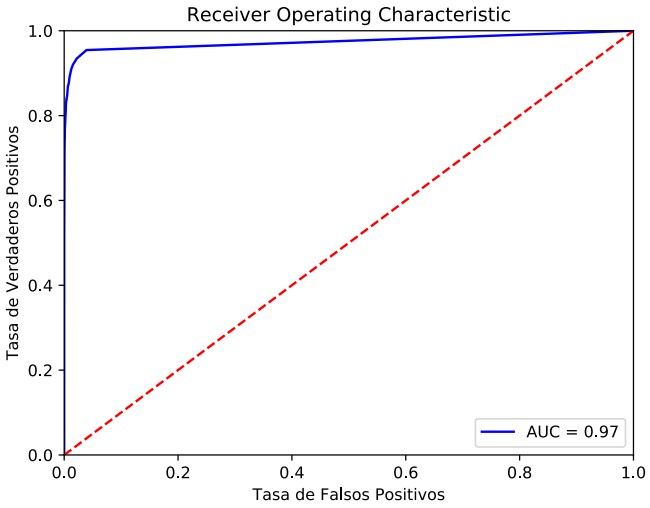


Figura 6.1: Curva ROC - Detector de DGA

Sin dudas los resultados son muy buenos. Probando con algunas URLs genéricas

`www.google.com`	legítimo
`www.google1234.com`	legítimo
`www.1cb8a5f36f.com`	dga
`www.f2c3b9oK.com`	legítimo
`www.facebook.com`	legítimo
`www.lacapital.com.ar`	legítimo
`stackoverflow.com`	legítimo
`sr34c4ov4t333fl3ow.com`	dga
`www.tecso.coop`	legítimo
`www.argen2dwef23rgrt43.com`	dga
`www.thisisafakeurl.com`	legítimo
`www.thi4i4afa9e5u2r1.com`	dga

La evaluación es buena, quizá con la excepción del dominio `www.f2c3b9oK.com`, que uno podría considerar generado. Para hacer una evaluación más realista, se realizó la prueba sobre 236369 URLs obtenidas en aproximadamente una semana de tráfico en la oficina de *Tecso* en Rosario. Entre todas las URLs se encontraron 8492 distintas. El modelo detectó solo 8 dominios como "dga".

```
2ch | 6sc | dlvr | dpgr | srcsmrtgs | ns02 | ss2 | tpo
```

Estos dominios son raros, especialmente porque son de corta longitud (excepto `srcsmrtgs`, pero es suficientemente raro como para que sea detectado). Para casos como estos es probable que debería crearse una lista de excepciones, ya que siendo tan cortos, es muy difícil que el detector los pueda clasificar correctamente como legítimo o generado.

Si bien no se observó cada una de las 236369 URLs para ver si hay alguna generada por algoritmo, se puede hacer una suposición relativamente segura de que generalmente en efecto no hay acceso a URLs generadas en la red de *Tecso*, por lo que el modelo estaría funcionando bastante bien. Pero para mejorar la prueba, se realizó la misma prueba pero esta vez agregando 2 dominios que podrían considerarse generados. Y se obtuvo muy buen resultado. El modelo clasificó las mismas que antes, pero también los 2 dominios que se agregaron.

```
zn3vgs8zfh | zdsre3d3t3jf34r3d
```

Esta prueba fue suficiente para saber que el modelo es considerablemente bueno para usarse en una red real tal como se presenta, pero se decidió probar algunas variantes para ver si puede mejorarse aún más.

Viendo los datos de entrada se puede observar un claro desbalanceo entre las muestras de dominios legítimos y las muestras de dominios generados. Hay 100000 dominios legítimos y 2669 dominios generados. En vista de los resultados vistos, en este caso el desbalanceo no afecta demasiado en la eficiencia del modelo. Pero aún así se realizó una nueva prueba aplicando sobremuestreo para equilibrar los grupos.

Como resultado se obtuvo una precisión = 0.990. La matriz de confusión obtenida se muestra en la Tabla 6.2.

	dga	legítimo
dga	86.796	13.204
legítimo	0.470	99.530

Tabla 6.2: Matriz de confusión - DGA Detector

Y la curva ROC en la Figura 6.2.

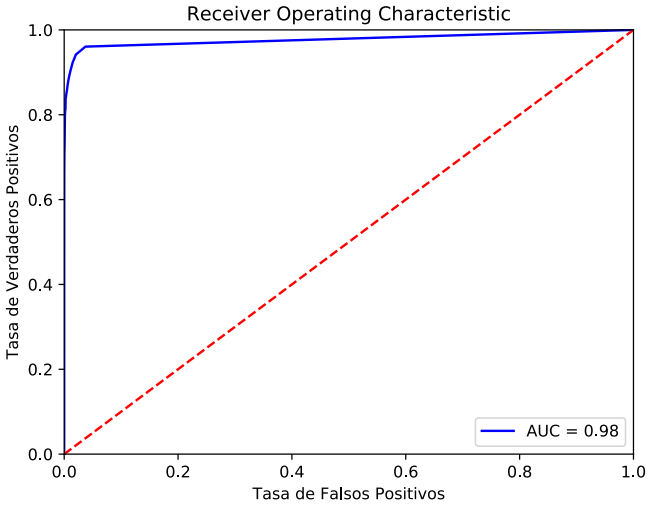


Figura 6.2: Curva ROC - Sobremuestreo - DGA Detector

Probando con los mismos dominios obtuvimos los siguientes resultados.

`www.google.com`	legítimo
`www.google1234.com`	legítimo
`www.1cb8a5f36f.com`	dga
`www.f2c3b9oK.com`	legítimo
`www.facebook.com`	legítimo
`www.lacapital.com.ar`	legítimo
`stackoverflow.com`	legítimo
`sr34c4ov4t333fl3ow.com`	dga
`www.tecso.coop`	legítimo
`www.argen2dwef23rgrt43.com`	dga
`www.thisisafakeurl.com`	legítimo
`www.thi4i4afa9e5u2r1.com`	dga

Y con las URLs obtenidas de la red de *Tecso* detectó 3 dominios más que antes pero que entran en la misma categoría de dominios de muy corta longitud difícil de clasificar correctamente.

`2ch | bit | 6sc | dlvr | dpgr | goo | srcsmrtgs | kn3 | ns02 | ss2 | tpo | zn3vgs8zfh | zdsre3d3t3jf34r3d`

En general puede observarse una leve mejora en la evaluación y resultados muy similares en las pruebas.

Los resultados son realmente muy buenos y sin dudas el modelo es capaz de detectar cualquier comunicación con dominios generados con muy baja cantidad de falsos positivos y menor aún falsos negativos.

6.1.4. Limitaciones

Como podemos ver en los resultados el modelo resultante no es capaz de diferenciar dominios legítimos de generados algorítmicamente en dominios cortos, de menos de 5 o 6 caracteres. Pero esta limitación resulta esperable, ya que tales dominios son muy difíciles de distinguir incluso para el ojo humano. Se podría programar el detector para que ignore el análisis en dominios cortos y así reducir la cantidad de falsos positivos, pero para mayor seguridad posiblemente sería conveniente agregar un mecanismo que permita al usuario marcar entre los dominios detectados aquellos que son legítimos y agregarlos a una lista de excepciones.

Un aspecto a tener en cuenta es que las palabras de diccionario usadas durante el entrenamiento son del lenguaje inglés. Esto hace que el modelo esté pensado para clasificar dominios en lenguajes derivados del latín. Pero considerando que la mayor cantidad de dominios que se acceden cumplen está condición no resulta un problema.

Otro aspecto a estudiar cuando se quiere utilizar en una red es el tiempo que requiere el modelo para clasificar los dominios. Se realizaron algunas mediciones de ejecución, considerando el proceso de formateo del dato y el tiempo de clasificación. Para una única URL, el tiempo fue de 0.32 s, para 1.000 URLs, el tiempo también fue de 0.32 s. Ya para 2600 URLs el tiempo subió a 0.60 s.

Las pruebas muestran que para una cantidad de URLs entre 1 y 1000, el algoritmo tarda prácticamente lo mismo, por lo que el modelo funciona de forma más eficiente al evaluar muchas URLs por iteración. También se puede ver que para una cantidad masiva de URLs el tiempo aumenta considerablemente. Estos resultados muestran que la mejor forma de utilizar el modelo es realizar evaluaciones cada cierta cantidad de tráfico. Por ejemplo, almacenar todos los dominios de las comunicaciones en la red durante 5, 10 o 15 minutos y clasificar todas juntas. Si bien de esta forma uno obtiene un retraso en la detección de varios minutos, no aparenta ser un gran problema, ya que los *malwares* que se comunican con servidores C&C suelen hacerlo varias veces y generalmente durante largos periodos de tiempo, a veces días o meses [13].

6.2. Detector de URL Maliciosas

Para implementar el detector de URLs maliciosas se siguió inicialmente un procedimiento similar al del detector de dominios generados, solo que en este caso en el módulo Extractor se definió la funcionalidad que recolecta todas las URLs completas encontradas en el archivo "pcap". Luego el detector implementado toma como entrada la lista de URLs y clasifica cada una como "buena" o "mala" reportando los paquetes correspondientes. Nuevamente nos encontramos con un problema de aprendizaje supervisado. Para este detector fue necesario realizar varios pasos de experimentación. Primero reproducimos el trabajo estudiado en la Sección 4.3. Al no encontrar resultados satisfactorios, se analizó e intentó mejorar el modelo siguiendo dos enfoques, primero modificando la selección de *features* y luego modificando los datos utilizados para entrenar.

6.2.1. Datos

Para entrenar el detector se consiguió una larga lista de URLs, tanto normales como maliciosas o al menos sospechosas. La primera fuente de URLs se obtuvo del mismo trabajo investigado [17], ya que el autor presentaba los datos con los que entrenó su modelo. Esta fuente de URLs fue también la utilizada para reproducir el modelo.

La segunda fuente de URLs "buenas" se obtuvo de los datos publicados por Alexa [1]. También se utilizó los datos que obtenidos de la red de *Tecso*. Para el caso de tráfico malicioso, la principal fuente se obtuvo del sitio "http://malware-traffic-analysis.net/" [33], sitio web en donde desde 2013 se han publicado más de 1000 archivos pcaps capturados que contienen el tráfico específico de diferentes *malwares*. También se tuvo la oportunidad de aislar una red privada en donde intencionalmente se ejecutaron diferentes *malwares* conocidos capturando el tráfico resultante. Pero con este método no se obtuvo demasiados datos, ya que no tuvimos acceso a una gran cantidad de *malwares*. Estas listas de URLs fueron utilizadas para los posteriores experimentos.

6.2.2. Implementación

El primer paso fue reproducir el algoritmo tal cual se presenta y analizar los resultados. Primero cargamos los datos. En este caso una lista de "buenas" y "malas" URLs. Luego mezclamos los datos y formateamos los vectores con *Term Frequency times Inverse Document Frequency*. Finalmente dividimos los datos, 80 % para entrenamiento, 20 % para evaluación.

Para entrenar el clasificador utilizamos el algoritmo *Logistic Regression*. Se comenzó utilizando selección de hiperparámetros definida por defecto en la librería de *scikit-learn*.

6.2.3. Primeros Resultados

Utilizando los datos reservados evaluamos el modelo y obtuvimos una precisión de 0.986, con una matriz de confusión mostrada en la Tabla 6.3.

	buena	mala
buena	99.490 %	0.510 %
mala	3.075 %	96.924 %

Tabla 6.3: Matriz de confusión - Detector de URL Maliciosas

La curva ROC en la Figura 6.3.

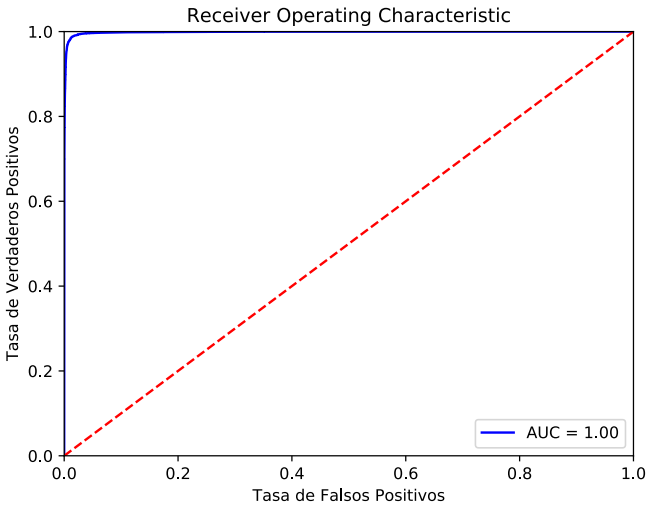


Figura 6.3: Curva ROC - Detector de URL Maliciosas

Y probando con las mismas URLs presentadas en el trabajo obtenemos exactamente el mismo resultado publicado.

`wikipedia.com`	buena
`google.com/search=faizanahad`	buena
`pakistanifacebookforever.com/getpassword.php/`	mala
`www.radsport-voggel.de/wp-admin/includes/log.exe`	mala
`ahrenhei.without-transfer.ru/nethost.exe`	mala
`www.itidea.it/centroesteticosothys/img/_notes/gum.exe`	mala

Viendo los valores de evaluación y particularmente el valor de AUC = 1.00, parecería que el autor encontró un modelo prácticamente perfecto, ya que no es posible obtener mejor evaluación. Pero la realidad es que es imposible obtener un modelo sin un mínimo de error, por lo que algo inusual está pasando. Solo

con una simple prueba más se pudo ver que en efecto el modelo está lejos de ser perfecto. Se probó 4 URLs comunes, no maliciosas, y el modelo clasificó las 4 como "mala".

`https://www.google.com`	mala
`http://www.cifasis-conicet.gov.ar/index.html`	mala
`www.google.com.ar/#q=maching-learning`	mala
`www.tecso.coop`	mala

Si el modelo no es capaz de detectar a `https://www.google.com` como buena, realmente no sirve de mucho. El siguiente paso fue investigar por qué se obtiene tan buenos valores de evaluación y tan malos resultados al clasificar. Lo primero que notamos es que en los datos hay un gran desbalanceo en las clases, teniendo 344821 URLs de clase "buena" y 42767 URLs de clase "mala", lo que significa que 89 % de los datos utilizados para entrenar pertenece a la clase "buena". Esta diferencia podría ser la causa de los resultados de tan alto valor de precisión en la evaluación. Así que lo primero que se probó fue balancear el conjunto de datos aplicando sobremuestreo. También para analizar mejor los resultados se separó según su clase ("buena" y "mala") y se evaluó el modelo en cada grupo por separado. En esta ocasión se obtuvo una precisión de 0.934, con la matriz de confusión de la Tabla 6.4.

	buena	mala
buena	98.937 %	1.062 %
mala	12.102 %	87.897 %

Tabla 6.4: Matriz de confusión - Detector de URL Maliciosas

Y la curva ROC en la Figura 6.4.

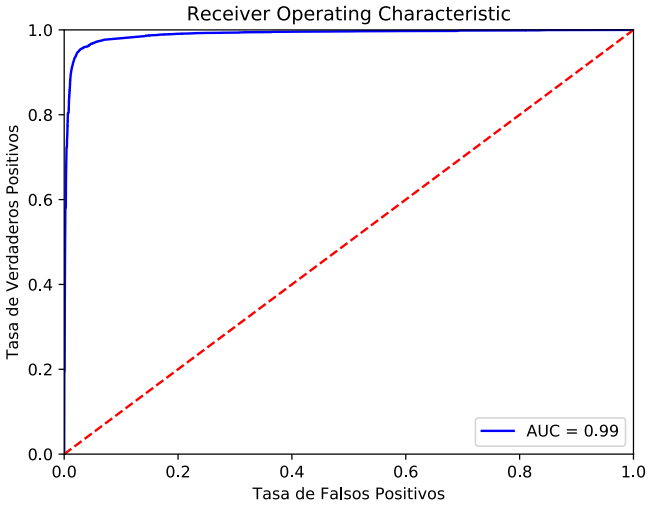


Figura 6.4: Curva ROC - Sobreajuste - Detector de URL Maliciosas

La precisión bajo levemente, pero aún se obtienen excelentes valores de evaluación. Y sin embargo al clasificar URLs comunes seguimos obteniendo un mal resultado. Probamos con las mismas 4 URLs y con una extra de aspecto claramente "malo".

`https://www.google.com`	mala
`http://www.cifasis-conicet.gov.ar/index.html`	mala
`www.google.com.ar/#q=maching-learning`	mala
`www.tecso.coop`	buena
`www.goog1e.com/getPassword.exe`	mala

La evaluación sigue siendo engañosa. Con los datos reservados se obtiene muy buen resultado pero con otras URLs normales se obtiene mal resultado, por lo que decidimos utilizar otra lista de URLs para la evaluación. Armamos una lista de URLs "normales", recolectadas del tráfico de la red privada de *Tecso* y otra con URLs de aspecto sospechoso, obtenidas de los pcaps de tráfico malicioso [33] y las utilizamos para la evaluación del modelo. El resultado resultó realmente decepcionante, pero bastante más realista de acuerdo a los resultados del test, funcionando prácticamente aleatoreamente. Se obtuvo una precisión de 0.420, con la matriz de confusión de la Tabla 6.5.

	buena	mala
buena	28.373 %	71.626 %
mala	44.285 %	55.714 %

Tabla 6.5: Matriz de confusión - Detector de URL Maliciosas

Y la curva ROC de la Figura 6.5.

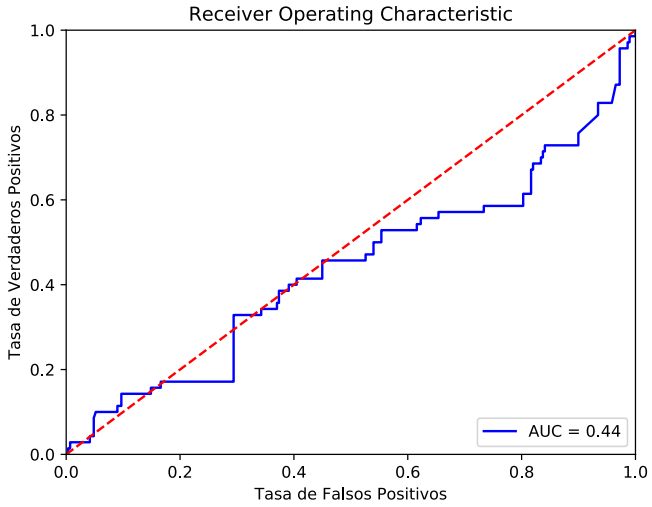


Figura 6.5: Curva ROC - Sobreajuste - Validando con datos nuevos - Detector de URL Maliciosas

6.2.4. Nuevos Experimentos

Dado estos resultados poco satisfactorios, se propuso realizar diferentes experimentos, probando variantes, intentando obtener una versión aceptable del clasificador. Se consideraron dos posibles problemas. Es posible que la función encargada de extraer los *features* no sea suficientemente buena. Por otro lado, también es posible que los datos de entrenamiento no sean adecuados.

Primero nos enfocamos en el primer problema. Por lo que creamos diferentes versiones de funciones para vectorizar las URLs.

```
1  def getTokens(input):
2
3      allTokens = re.split(r'[/]',input)
4      allTokens = allTokens + re.split(r'[/.-]',input)
5
6      allTokens = list(set(allTokens))
7
8      if ('com' in allTokens):
9          allTokens.remove('com')
10     if ('www' in allTokens):
11         allTokens.remove('www')
12     if ('' in allTokens):
13         allTokens.remove('')
14
15     return allTokens
```

La primera *getTokens* es la misma que presenta el autor, solo que realizando los cortes utilizando expresiones regulares para aumentar la eficiencia y además eliminando también los *tokens* "www" ya que no aportan valor para diferenciar las URLs. En la segunda variante, *getTokens2,* la función solo utiliza el conjunto de *tokens* divido por "/","." y "-", mientras que en la original también incluye los *tokens* obtenidos por solo dividir por "/". La tercera versión*, getTokens*3*,* es también una variante de la primera, solo que en esta se generan los *tokens* dividiendo según los caracteres "/" "." "?" "&" ";" "-" . En la cuarta versión*, getTokens*4, se modificó la segunda de la misma forma que la tercera modifica la primera, dividiendo los *tokens* según los caracteres "/" "." "?" "&" ";" "-". Finalmente, las funciones *getTokens*5, *getTokens*6, *getTokens*7 y *getTokens*8 son variaciones respectivas de las funciones *getTokens*1, *getTokens*2, *getTokens*3 y *getTokens*4, en donde se marca aquellos *tokens* que pertenecen al dominio agregando adelante "dom/" y al resto "url/".

Aplicando la funciones a la siguiente URL de ejemplo

`espec.com.ar/proy/index_cob.html?chId=pymnro0&initWidth=960`

getTokens(url)	"espec" "ar" "index_cob" "proy" "html?chId=pymnro0&initWidth=960" "espec.com.ar," "index_cob.html?chId=pymnro0&initWidth=960"
getTokens2(url)	"espec" "ar" "index_cob" "proy" "html?chId=pymnro0&initWidth=960"
getTokens3(url)	"chId=pymnro0" "html" "espec" "ar" "initWidth=960" "espec.com.ar" "index_cob" "proy" "index_cob.html?chId=pymnro0&initWidth=960"

getTokens4(url)	“chId=pymnro0” “html” “espec” “ar” “initWidth=960” “index_cob” “proy”
getTokens5(url)	“dom/espec” “dom/ar” “dom/espec.com.ar” “url/html?chId=pymnro0&initWidth=960” “url/proy” “url/index_cob” “url/proy” “url/index_cob.html?chId=pymnro0&initWidth=960”
getTokens6(url)	“dom/espec” “dom/ar” “url/proy” “url/index_cob” “url/html?chId=pymnro0&initWidth=960”
getTokens7(url)	“dom/espec” “dom/ar” “dom/espec.com.ar” “url/index_cob” “url/chId=pymnro0” “url/proy” “url/html” “url/proy” “url/initWidth=960” “url/index_cob.html?chId=pymnro0&initWidth=960”
getTokens8(url)	“dom/espec” “dom/ar” “url/html” “url/proy” “url/index_cob” “url/initWidth=960“ “url/chId=pymnro0”

Para realizar las pruebas también se probó ajustando algunos parámetros del algoritmo *Logistic Regression*. En total, fueron 280 experimentos, en donde en solo unos pocos de ellos resultó alguna leve mejora, pero aún lejos de ser buen resultado. Tras varias iteraciones la combinación que resultó más estable y con mejor resultado fue usando la función `getToken2` y con los parámetros “C” y

"tol", 1000 y 0.0001 respectivamente. Se obtuvo una precisión de 0.476, con la matriz de confusión de la Tabla 6.6

	buena	mala
buena	35.640 %	64.360 %
mala	40 %	60 %

Tabla 6.6: Matriz de confusión - Detector de URL Maliciosas

Y valor de AUC de 0.53.

Este es el mejor valor que se pudo obtener utilizando los datos de entrenamiento que utilizó el autor sin modificaciones. Y los resultados siguen siendo decepcionantes, ya que no solo se obtuvo un bajo nivel de detección para las url "malas", si no que además detectó más del 60 % de urls "buenas" como maliciosas.

Buscando resolver el primer problema solo obtuvimos una muy leve mejora, por lo que a continuación nos enfocamos en el segundo posible problema. El siguiente paso entonces fue complementar los datos. Se comenzó agregando al conjunto de URLs "buenas" las primeras 100000 URLs de Alexa y se realizó nuevamente los 280 experimentos con este nuevo conjunto de datos de entrenamiento. Se obtuvo una mejora considerable. El mejor resultado fue nuevamente usando la función `getToken2` pero con los parámetros de "C" y "tol", 2000 y 0.00001 respectivamente. En esta ocasión se obtuvo una precisión de 0.635, con la matriz de confusión de la Tabla 6.7.

	buena	mala
buena	57.093 %	42.906 %
mala	30 %	70 %

Tabla 6.7: Matriz de confusión - Detector de URL Maliciosas

Y la curva ROC de la Figura 6.6.

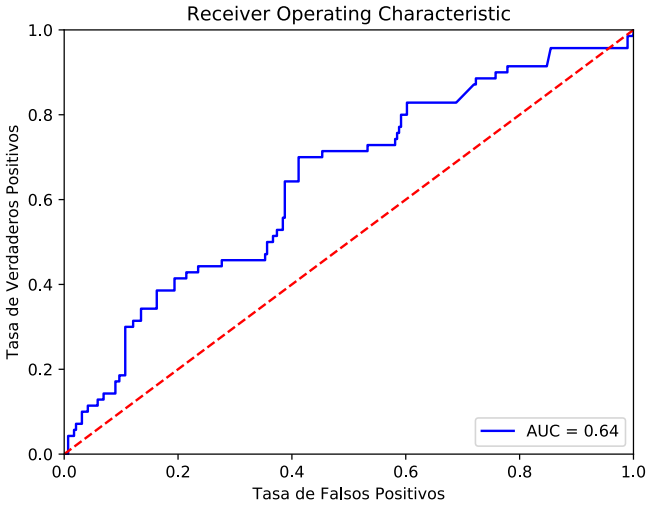
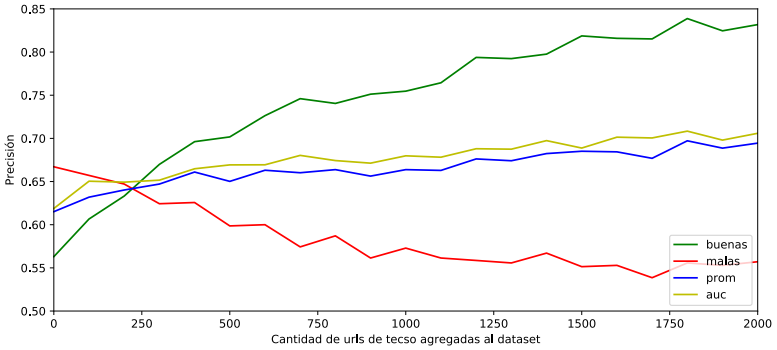
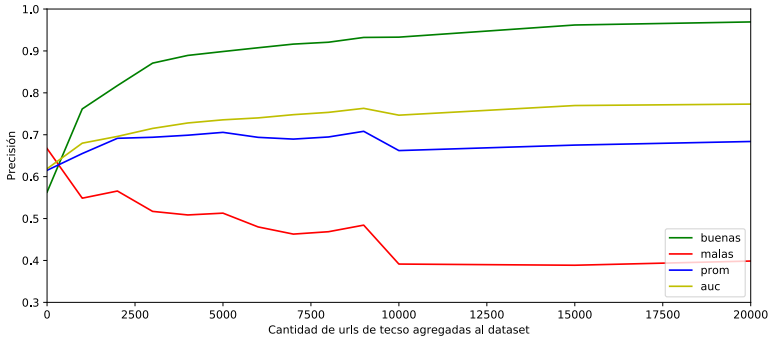


Figura 6.6: Curva ROC - Sobreajuste - Agregando URLs de Alexa - Detector de URL Maliciosas

En ambas iteraciones de experimentos, el mejor resultado surgió de utilizar la función *getToken2*, y también podemos observar una significativa mejora en los resultados al agregar las URLs de alexa como "buenas". Siguiendo la misma idea, realizamos dos series de experimentos más. En la primer serie, en cada experimento se incrementó el conjunto de datos de URLs "buenas" con 100 URLs capturadas de la red de *Tecso*, finalizando la serie con 2000 URLs agregadas. En la segunda serie, el incremento en cada experimento fue de 1000 Urls, concluyendo la serie con 20000.



(a) Evolución con +100, +200, ... , +2000



(b) Evolución con +1000, +2000, ..., +10000, +15000, +20000

Figura 6.7: Evolución de la precisión en el Detector de URL Maliciosas

Viendo las gráficas de las Figuras 6.7a y 6.7b podemos observar que a medida que agregamos URLs locales la precisión del detector para URLs buenas va mejorando pero a su vez va empeorando la precisión para detectar URLs malas. Si observamos el promedio o el valor de AUC podemos ver que van mejorando muy levemente, y manteniéndose bastante constantes luego de una cantidad de URLs agregadas. Prestando atención especialmente a la Figura 6.7b, se detectan dos picos en el promedio con precisión cercana a 0.7, el primero entre 2000 y 2500 URLs agregadas y el segundo entre 8000 y 9000. Si bien la precisión promedio en ambos puntos es parecida, los detectores tienen características algo distintas. El primero tiene una precisión para URLs buenas alrededor de 0.8 y para URLs malas alrededor 0.55, mientras el segundo superior a 0.9 para las buenas y cercano a los 0.5 para las malas. Entonces podemos concluir que con el primer modelo obtendremos mayor cantidad de falsos positivos, y con el segundo una mayor cantidad de falsos negativos.

Finalmente, al no encontrar resultados suficientemente buenos para incluir en nuestra plataforma de detección, intentamos una última variante en los datos. Esta vez se descartó el dominio de las URLs. Al descartar el dominio se pierde información, pero vale la pena ver si enfocando el detector en la ruta y los parámetros obtenemos mejor resultado. Al eliminar los dominios ya no podemos utilizar los datos de Alexa, ya que estos son todos dominios, por lo que primero se utilizó solo los datos originales, y nuevamente se fue agregando URLs locales y viendo su evolución.

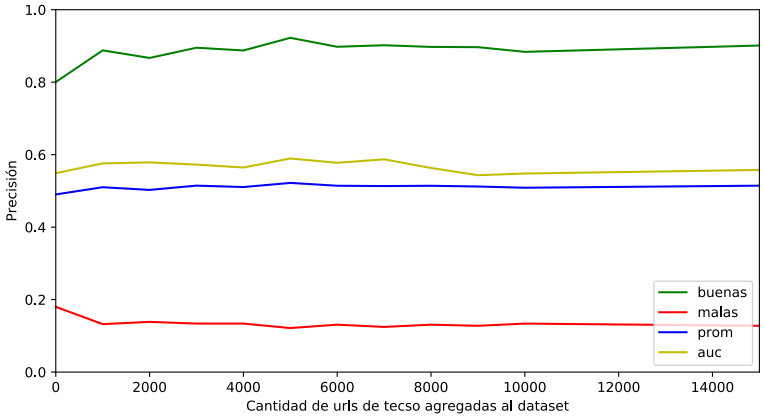


Figura 6.8: Evolución de la precisión en el Detector de URL Maliciosas - (sin dominios)

Como podemos ver en la figura 6.8, en este caso, el agregar URLs locales prácticamente no genera ninguna mejora independientemente de la cantidad agregada. Esto tiene bastante sentido ya que usualmente la mayor diferencia en las URLs de diferentes países se encuentra en el dominio. De todas formas, la precisión resulta un poco más baja aún que entrenando con los dominios, por lo que no parece ser el camino para mejorar el detector.

6.2.5. Resultados Finales

Llegado a este punto, podemos concluir que el detector no es lo suficientemente bueno para detectar URLs maliciosas con alta precisión. Además, también podemos concluir que los experimentos no reflejan la eficacia publicada por el autor. Sin embargo, no descartamos completamente su utilidad, ya que si de alguna forma se logra mejorar aún un poco más la precisión en las URLs buenas, reduciendo a un mínimo los falsos positivos, el detector podría servir. Imaginemos un detector que solo detecta un 40 % de URLs maliciosas, pero que prácticamente nunca reporta URLs buenas como maliciosas. Tal detector lo consideraría apto para incluir en la plataforma de detección.

6.3. Detector de TLS Malicioso

Tal como mencionamos en el capítulo 4.4, los trabajos estudiados no presentan los datos de entrenamiento, por lo que el primer trabajo fue intentar recolectar datos para entrenar un modelo. Como el objetivo es poder identificar tráfico malicioso sin descifrar los datos, la idea fue crear un detector que tomando como entrada una lista de certificados pueda distinguir aquellos que son maliciosos o al menos sospechosos. Es por esto que nos propusimos utilizar los datos de certificados de TLS como fuente principal de características para nuestro modelo. De hecho *Suricata* [39] recolecta toda la información relevante sobre los certificados aceptados, por lo que el detector sería fácilmente adaptable a la plataforma DAPA. Lamentablemente a causa de falta de datos no pudimos entrenar un modelo, pero se analizaron los pocos datos obtenidos como referencia a un futuro trabajo.

6.3.1. Datos

Dado que ninguno de los trabajos estudiados en la Sección 4.4 ofrecen los datos utilizados nos propusimos recolectar datos por nuestra cuenta.

Conseguir certificados legítimos fue sencillo, se utilizó aquellos proporcionados por diferentes navegadores (Firefox, Chrome, etc), se recolectaron certificados en las comunicaciones de la red privada de *Tecso* utilizando Suricata [39] y para completar se bajaron alrededor de 10 gigas de certificados desde [45] donde están todos los certificados públicos.

La principal dificultad fue obtener certificados maliciosos. Después de mucha búsqueda solo se pudo obtener unos pocos de un sitio [51] donde se expone una lista negra de SSL (*Secure Sockets Layer*, antecesor de TLS). La lista contiene alrededor de 60 sitios reportados, pero lamentablemente para este trabajo, al intentar capturar los certificados accediendo a cada sitio, solo unos 10 respondieron. El resto de los sitios ya habían sido dados de baja.

6.3.2. Implementación

Debido a la muy poca cantidad de datos de TLS maliciosos fue imposible entrenar un modelo supervisado. Por lo tanto, se propuso realizar una exploración de los datos, buscando semejanzas y diferencias entre los TLS maliciosos encontrados y un conjunto de certificados legítimos. Para ello se procesaron los datos en un algoritmo de agrupamiento (*clustering*), utilizando como conjunto de datos de entrada 300 certificados legítimos y los 10 maliciosos obtenidos del sitio [51].

De cada certificado se utilizaron los siguientes datos:

- C : *Country Code* (Código de País)
- ST: *State* (Estado)
- L: *Locality* (Localidad)

- O: *Organization* (Puede ser el nombre de la empresa validada o el nombre del dominio del sitio registrado dependiendo del tipo de certificado, *Organization Validated* o *Domain-Validated*)
- OU: *Organization Unit* (Unidad dentro de la organización o una URL, según si es *Organization Validated* o *Domain-Validated*)
- CN: Common Name (El nombre del dominio del sitio registrado)
- *EmailAddress* (Dirección de mail)

Cada campo fue vectorizado con Tfidf y para todos los ellos excepto el *Country Code* (solo contiene dos caracteres) se los procesó como n-gramas de entre 3 y 5 caracteres.

Para cada registro de certificado legítimo se armó una etiqueta concatenando "L_" con *Country Code*. Por otro lado, para los maliciosos se concatenó "MALICIOSO_" con un número de 1 a 10 y el *Country Code*. Como algoritmo de agrupamiento se utilizó Kmeans [47] y para poder graficar el resultado se aplicó PCA [48], reduciendo las dimensiones a dos.

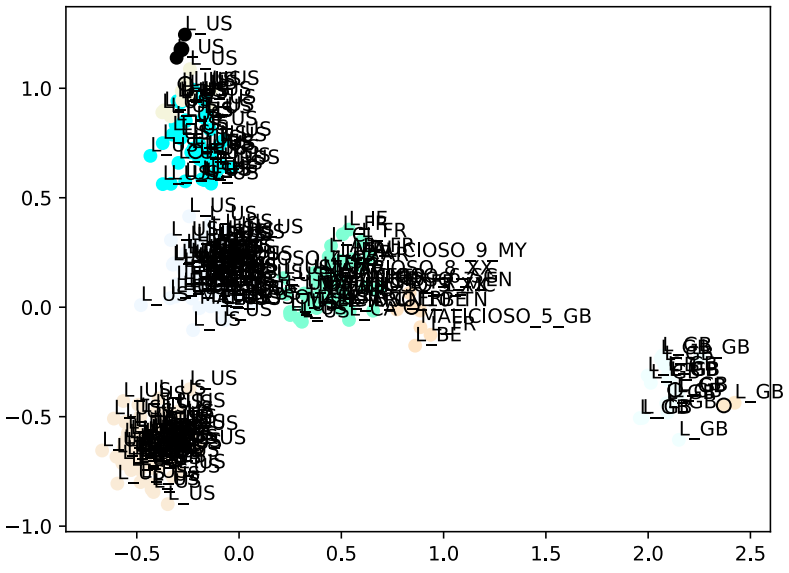


Figura 6.9: TLS - Agrupamiento

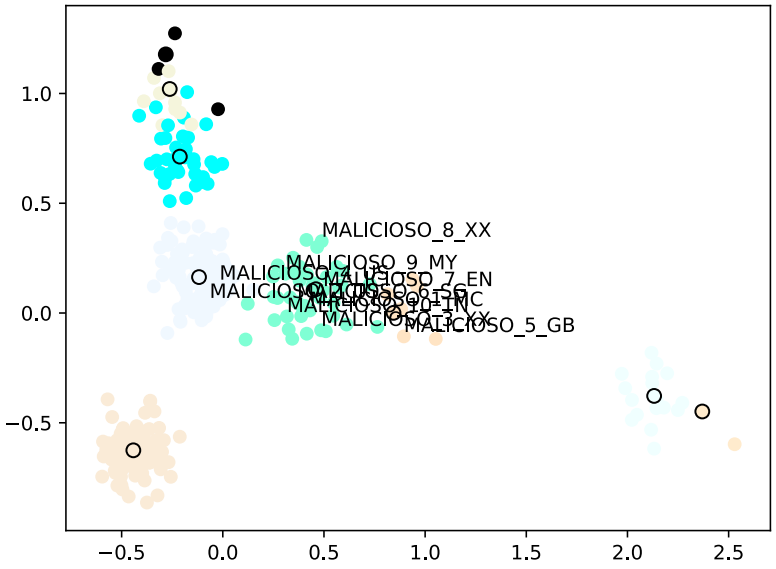
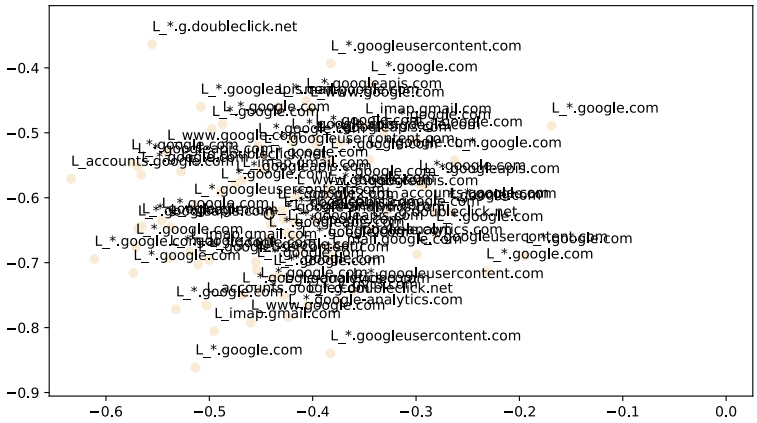


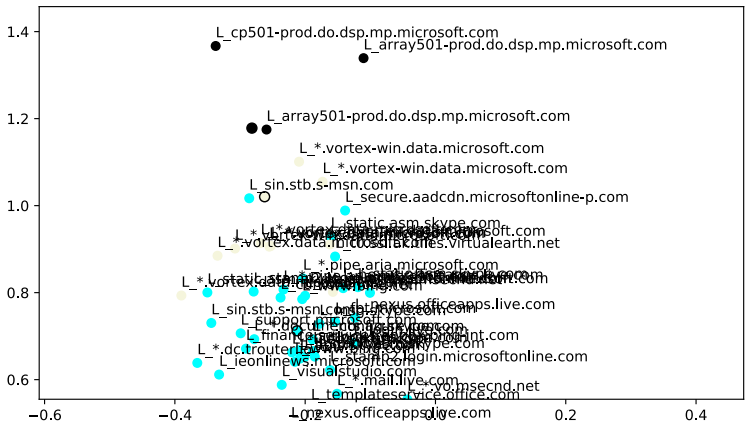
Figura 6.10: TLS - Agrupamiento - Sin etiqueta para los legítimos

En la figura 6.9 podemos ver que el algoritmo creo 5 grupos, la cuales están fuertemente definidas por el código del país. Los tres grupos (*clusters*) de la izquierda agrupan certificados con código del país 'US', el pequeño grupo de la derecha asocia aquellos de Gran Bretaña 'GB' mientras que el grupo del centro asocia una mezcla con distintos códigos de países. Los certificados maliciosos se agrupan en los dos grupos del medio, 8 de éstos en el grupo del centro y 2 que tienen código de país 'US' en el grupo de la izquierda.

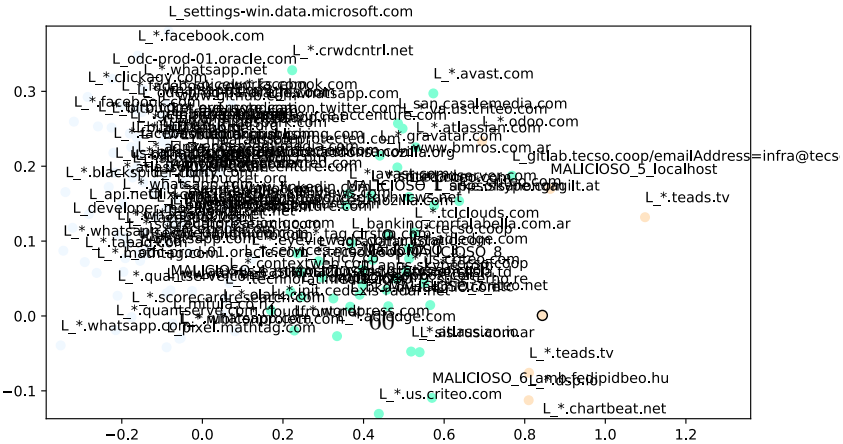
Ahora, de los 3 grupos de 'US', la división parece estar definida principalmente por el *CN* y el *O* ya que el grupo inferior junta certificados de Google, el superior izquierdo agrupa certificados de Microsoft y el resto en el grupo central (Figuras 6.11a, 6.11b y 6.11c). Esta forma de agrupación se produce debido a la gran cantidad de certificados de Google y Microsoft entre los 300 analizados, encontrado 81 del primero y 24 del segundo.



(a) Grupo inferior izquierdo



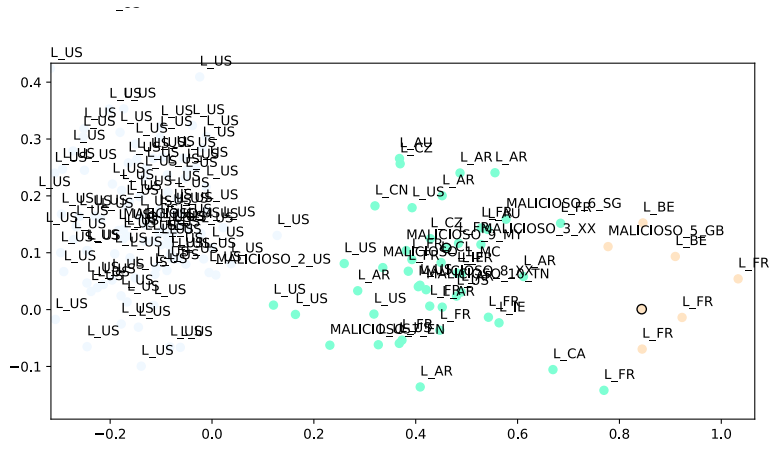
(b) Grupo superior izquierdo



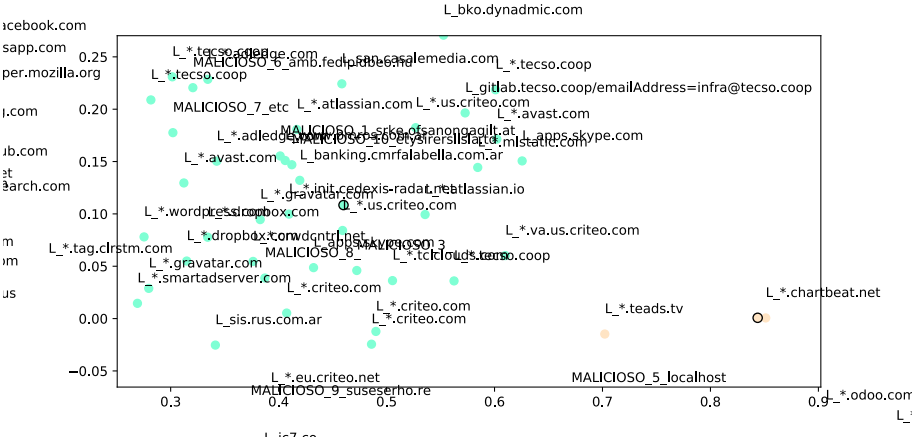
(c) Grupos centrales

Figura 6.11: TLS - Agrupamiento - Etiquetas con *Common Name*

Analizando entonces los grupos en donde están contenido los certificados maliciosos, los dos centrales, podemos ver que junto a los maliciosos se encuentran varios certificados legítimos. Pero estos certificados legítimos se caracterizan en ser certificados con pocas ocurrencias entre los 300 utilizados.



(a) Etiquetas con *Country*



(b) Etiquetas con *Common Name*

Figura 6.12: TLS - Agrupamiento - Grupos Centrales

6.3.3. Resultados

Es muy difícil llegar a una buena conclusión con tan pocas muestras de certificados maliciosos, pero hay dos características encontradas que pueden guiarnos a qué tipo de modelo sería conveniente utilizar en el caso conseguir una adecua-

da cantidad de datos de entrenamiento. La primer característica se desprende directo del resultado del algoritmo de agrupamiento y es que los certificados maliciosos generalmente estarán en el grupo de certificados que aparezcan escasamente. La segunda característica se puede encontrar analizando los datos y es que en general, entre los datos obtenidos de los certificados maliciosos podremos encontrar al menos un valor anómalo entre sus campos. Un valor que a simple vista pareciera sospechoso.

Entonces creemos que un buen camino para detectarlos sería con un modelo de detección de anomalías, entrenando el modelo específicamente para la red en donde se desea utilizar, logrando entonces que el detector alerte sobre aquellos certificados raros y nunca visto.

6.4. Anagram

Los anteriores detectores se implementaron para adaptarse directamente a la plataforma de detección DAPA, analizando los datos recolectados de toda la red protegida. *Anagram*, por otro lado, se implementó para aplicar en servidores específicos en donde se requiera una seguridad mayor y más estricta. *Anagram* toma como entrada la cadena de *bytes* (sin interpretar) perteneciente al *payload* de los paquetes y reporta aquellos que considera anómalos. Por lo que se trabajó en un modulo individual capaz de leer tráfico de red y de reportar aquellos paquetes considerados raros en forma de alertas almacenadas en *solr*. En este caso la mecánica utilizada en el detector es la publicada en el trabajo estudiado en 4.5 .

6.4.1. Datos

Para reproducir *Anagram* no es requerido recolectar previamente datos. Al ser un detector de anomalías, el modelo necesita "aprender" el tráfico de la red a la cual se pretende proteger, por lo que datos genéricos almacenados no proporcionan ninguna ventaja. Sin embargo, si tuviéramos almacenado tráfico de la red de interés, sin duda aceleraría mucho el tiempo de entrenamiento del modelo. Por otro lado, si tuviéramos almacenado el tráfico de diferentes *malwares* en un *Bad Bloom Filter* como se menciona en el trabajo 4.5, tanto el entrenamiento como el análisis de anomalías serían mucho más seguros. Lamentablemente, para esta tesina no se obtuvo suficiente información para crear un *Bad Bloom Filter* que produzca una real diferencia en los experimentos, por lo que no se utilizó para el análisis.

6.4.2. Implementación

En este caso, el trabajo no presenta una implementación del algoritmo, pero explica suficientemente bien la mecánica como para programar una versión. Se implementó una versión de Anagram en Python en su versión 3. En Anagram hay tres funcionalidades de configuración que influyen consideradamente en el funcionamiento del modelo. La primera es que valores de n-gramas se utilizan para realizar los cortes del *payload*; por ejemplo, si se configura con 3 y 5, y tenemos un *payload* de 10 *bytes* de tamaño, obtendremos cuatro 3-gramas y dos 5-gramas. La segunda es la función que determina en que momento se considera el modelo entrenado y la última es la función que determinará si un paquete es anómalo o no.

El funcionamiento básico detrás de un detector de anomalías es "aprender" todo el tráfico normal, para así distinguir el "raro". Ahora, si tomamos como ejemplo una red privada pero no restrictiva, donde el volumen del tráfico de la red es tan grande y variante, en donde los usuarios entran diariamente a páginas de diarios, redes sociales y muchas otras que suelen actualizar constantemente su contenido, el entrenamiento de un modelo para tal red podría no terminar nunca o tardar un tiempo realmente enorme. En su lugar, enfocamos el algoritmo para

detectar anomalías en donde el tráfico de red es acotado y poco variante a lo largo del tiempo, como por ejemplo un servidor de base de datos.

En Anagram la dificultad del entrenamiento del modelo varía mucho según se realice *online* (en tiempo real), u *offline*, con tráfico almacenado previamente. En el caso de entrenamiento *offline* se requiere tener almacenado en pcaps, o algún otro medio de almacenamiento, suficiente tráfico que abarque toda o casi toda la variación usual del servidor. El principal problema de este modo es que se requiere una enorme cantidad de espacio para almacenar incluso el tráfico de una red muy pequeña, siendo prácticamente impracticable en redes muy grandes. Sin embargo, este método tiene una ventaja considerable en cuanto a limpiar la impureza (tráfico malicioso) que puede contener el tráfico, ya que antes de procesar se puede analizar el tráfico almacenado y eliminar cualquier amenaza que se detecte. Entrenando el modelo en tiempo real claramente no se tiene el problema de almacenamiento, pero resulta extremadamente difícil asegurar que en el tráfico capturado no haya *malwares*, y surgen otras cuestiones tales como qué método utilizar para determinar si el modelo ya está suficientemente entrenado.

6.4.3. Experimentos

Como el objetivo es analizar su eficiencia para utilizarse en redes reales, y para evitar el problema de almacenar enormes cantidad de datos, primero enfocamos el entrenamiento utilizando el método *online*. Si bien este enfoque tiene mayores complicaciones, a primera vista su implementación parece más factible en tales redes. El método implementado para determinar cuándo dejar de entrenar fue sencillo. Primero se implementó un buffer de tamaño configurable en donde almacenar los sucesivos valores de *score* calculados. Luego, en cada cada momento en que se incorpora un nuevo *score*, se calcula promedio de ellos. Y si el promedio es menor que un valor mínimo configurado (por defecto 0.0001) se considera el modelo entrenado. El primer problema que surgió durante las primeras pruebas fue que el tráfico que se estaba analizando presentaba la característica de tener largos períodos de poco tráfico y muy poco variante. Entonces, al comenzar el entrenamiento durante uno de estos períodos el entrenamiento terminaba sin realmente llegar a ver gran parte del tráfico real. Para evitar estas situaciones se agregó un tiempo mínimo configurable donde el modelo debe seguir entrenando sin importar el valor de los *scores.* Con esta técnica nos aseguramos un entrenamiento mínimo razonable configurado según la comunicación que se pretenda aprender.

Se comenzó haciendo un experimento sencillo. Se puso en funcionamiento un servidor de base de datos, y un pequeño cliente que permite ingresar un campo de texto. El texto es enviado al servidor por http, sin utilizar encriptación. A continuación, se entrenó un modelo capturando los paquetes generados al ingresar una frase simple de 22 caracteres. Luego se comenzó a evaluar con el modelo y se ingresaron dos frases, la misma utilizada al entrenar y otra en la que se agregó una *select* simple de base de datos.

	n-gramas totales	n-gramas nuevos
paquete con frase original	707	10
paquete con frase agregado el *select*	733	35

Tabla 6.8: Resultado de evaluación en modelo simple

Con la frase original se detectaron 10 n-gramas nuevos, dando a entender que algunos pocos n-gramas varían siempre o en muchos casos, por lo que hay que tener en cuenta esta varianza mínima para que no se generen una enorme cantidad de falsos positivos. Más importante, podemos observar que para la frase en donde se agrego el *select* se generaron 30 n-gramas más de los cuales 25 son nuevos, demostrando que el modelo detecta con efectividad paquetes con datos desconocidos.

Realizando pruebas en un entorno simple experimental fue fácil mostrar la eficacia de Anagram para detectar anomalías. Al tener poca variedad de tráfico, el modelo es entrenado rápidamente, y cualquier tráfico desconocido es fácilmente detectado. Pero ¿qué ocurre en un entorno real con una gran cantidad de tráfico variante? ¿cuánto tiempo lleva entrenar el modelo para que esté listo para ser usado? ¿qué tamaño tendrá el modelo entrenado? Para responder estas preguntas, se experimentó con un sitio real de uso interno en la cooperativa de trabajo *Tecso*. Este sitio es utilizado por todos los asociados para realizar diferentes tareas, tales como consultar información, asignar proyectos, carga de horas de trabajo y varias otras actividades. Analizando el tipo de actividades que se realizan en el sitio, las cuales varían según la época del año, para lograr un buen entrenamiento del modelo estimamos que lo mejor sería entrenar por ejemplo 2 o 3 semanas en varios intervalos durante un año, pero en lo que se refiere a esta tesina no se dispone de tanto tiempo, por lo que se apuntó a realizar entrenamientos de alrededor de 2 semanas y analizar las pruebas considerando esta falta de entrenamiento en la cantidad de falsos positivos.

El primer experimento se realizó configurando Anagram generando n-gramas de longitud 3 y 5, y entrenando como mínimo una semana. El entrenamiento terminó automáticamente luego de 8 días, pero se volvió a reentrenar durante una semana más. Luego se testeó durante un día y se analizaron las detecciones de anomalías. La función de detección utilizada considera anómalo cualquier paquete con mas de 20 % de n-gramas nunca vistos. En solo un día se generaron alrededor de 1200 alertas. Sin duda tal número refleja una enorme cantidad de falsos positivos, pero analizando las alertas se puede observar una situación especial. De todas ellas, el 70 % son alertas generadas de paquetes con menos de 40 n-gramas. Ahora, si el total de n-gramas es de 40, el 20 % es solo 8 y por lo tanto el modelo genera alertas con solo 8 n-gramas nuevos. Esta medida de análisis es bastante estricta y genera una muy alta cantidad de falsos positivos. Pensando en una implementación final, lo mejor sería utilizar otro tipo de cálculo más preciso para determinar cuando se considera anómalo un paquete, pero para reducir los falsos positivos y facilitar la evaluación del modelo en esta tesina se dejó el límite de 20 % y se ignoró aquellos paquetes con muy pocos n-gramas (menores a 50). Por otro lado, para reducir la cantidad de n-gramas y así acelerar

el entrenamiento se cambiaron los cortes de 3 y 5, por cortes de 5 y 7 gramas.

El segundo experimento entonces se configuró generando n-gramas de longitud 5 y 7 y se entrenó durante 3 semanas. En esta oportunidad el testeo resultó bastante diferente. En varios días de análisis no se detectó ni un solo falso positivo. Fue importante analizar si el buen resultado se debió a un muy buen entrenamiento, o si por otro lado la función que determina paquetes anómalos resultó muy permisiva. Se pudo observar que si bien el entrenamiento fue mejor, en efecto la función quedó poco estricta. En 10 días de análisis, encontramos los siguientes resultados.

días	10
total paquetes	2429138
paquetes con nuevos n-gramas = 0	2198507
paquetes con nuevos n-gramas entre 0 y 50	222619
paquetes con nuevos n-gramas > 50	8012

Tabla 6.9: Resultados de análisis

La gran cantidad de paquetes en donde no se detectaron nuevos n-gramas es una prueba concreta de que el entrenamiento resultó bastante mejor, pero también podemos observar que hay 8.012 paquetes en donde se encontraron mas de 50 n-gramas nuevos, encontrando incluso paquetes con alrededor de 450 n-gramas nuevos. Estos paquetes no son reportados ya que son paquetes muy grandes en donde esos 450 n-gramas no llegan a superar el 20 % del total de n-gramas en el paquete. Esto prueba que utilizar la función que mide como límite el 20 % no es una buena medida, ya que creemos que estos paquetes deberían ser reportados como anómalos sin importar el tamaño total. Consideramos que una buena medida sería una simple función de límite para una cantidad fija, pero variable a decisión del administrador del sistema. Por ejemplo, configurando la variable en 50, se reportarían todos los paquetes con mas de 50 n-gramas nunca vistos. Aumentando o disminuyendo el valor de la variable hacemos menos o más estricto el modelo, reduciendo o aumentando la cantidad de falsos positivos.

Si consideramos el límite de 50 n-gramas nunca vistos, el modelo habría reportado 8012 paquetes como anómalos. Son más de 800 paquetes por día en promedio, lo que es una cantidad enorme de falsos positivos para un buen modelo de detección, pero es comprensible debido a el corto tiempo de entrenamiento. Se entrenó solo durante tres semanas seguidas, cuando nuestro análisis del sitio determinó que se requerirían varias intervalos de dos o tres semanas de entrenamiento durante un año para lograr un excelente modelo.

Del resultado de los experimentos anteriores se puede proyectar que con el tiempo adecuado para entrenar el modelo, se puede lograr en efecto un muy buen modelo que produzca muy baja cantidad de falsos positivos (dependiendo de qué estricto lo hagamos) por lo que se procedió a testear su comportamiento en casos de "ataques". Por supuesto, al estar probando en un sitio real, en funcionamiento, no fue posible utilizar *malwares* reales, pero como el detector considera cualquier contenido nunca visto como posible ataque, para probar el detector basta con

generar tráfico raro y ver si es detectado.

En principio, al no estar completo el entrenamiento, simular un ataque generando tráfico raro no difiere de los 8.012 paquetes detectados como anómalos, ya que en efecto, para el modelo esos paquetes contienen n-gramas nunca vistos y por lo tanto los considera como amenaza. Aún así, se realizaron algunas pruebas enviando datos atípicos al sitio, en particular, ingresando consultas de base de datos en algunos de los pocos campos de texto del sitio en donde se permite ingresar datos. El resultado fue similar al encontrado en el primer experimento sencillo que realizamos, es decir, el modelo detectó alrededor de 20 y 30 n-gramas desconocidos más de lo usual. La consulta de bases de datos ingresada fue sencilla, simulando un simple SELECT con alguna condición en un WHERE, también similar a la utilizada en el primer experimento, por lo que el resultado tiene sentido. Ahora, si el modelo estuviera utilizando el límite de 50 n-gramas nunca vistos para reportar el paquete como anómalo, estos paquetes con solo 30 no serían reportados. Nuevamente nos encontramos dependiendo de cuando consideramos un paquete como anómalo. Si bajamos el límite a pocos n-gramas, detectaremos mayor cantidad de posibles ataques, pero aumentará la cantidad de falsos positivos. Por el contrario, si aumentamos el límite, reduciremos los falsos positivos, pero probablemente dejaremos pasar inadvertidos algunos posibles intentos de ataques.

Finalmente se probó entrenar un modelo en forma offline. Se capturó el tráfico de unos días en un archivo pcap y se utilizó para entrenar un modelo. Rápidamente se pueden ver las ventajas de éste método. En primer lugar, como se mencionó anteriormente, previamente a procesar el pcap para entrenar el modelo se puede pasar algún analizador para detectar si hay en los datos algún *malware*, y descartar aquellos paquetes infectados. Por otro lado se obtiene una ventaja en cuanto al tiempo en que se está ejecutando el programa de entrenamiento, ya que en poco más de una hora el algoritmo procesó el tráfico de varios días. Y el modo *offline* trae aún otra ventaja, que es que no se requiere instalar ni realizar cambios importantes en las redes que se desean proteger hasta que ya tengamos listo el modelo para desplegar. Por lo tanto, a partir de nuestros experimentos, recomendamos el modo *offline* de entrenamiento.

6.4.4. Resultados

Si bien debido a la gran cantidad de tiempo que se requiere no pudimos entrenar un modelo que de como resultados una adecuada baja cantidad de falsos positivos, creemos que los experimentos realizados bastan para mostrar que Anagram es una buena opción entre los detectores de anomalías, bastante potente para detectar incluso muy pequeños cambios en la comunicación analizada, siempre que estemos dispuesto a aceptar una cantidad razonable de falsos positivos. Creemos que esto dependerá de qué tan bien entrenamos nuestro modelo. Si dedicamos el tiempo adecuado para crear un modelo que aprenda casi toda la comunicación normal del sitio, podremos utilizar un límite bajo de n-gramas nuevos para la detección de paquetes anómalos. Este modelo tendrá entonces un gran poder de detección de *malwares* con baja cantidad de falsos positivos. Por

otro lado, si no logramos entrenar el modelo lo suficiente, se puede aumentar el límite para reducir los falsos positivos, resignados a detectar solo paquetes con una alta cantidad de n-gramas nunca vistos.

También podemos asegurar que la implementación no es complicada. El algoritmo es medianamente sencillo de reproducir en Python, obteniendo buenos resultados de desempeño. En cuanto a los recurso requeridos, dependerá de la cantidad de trafico que queramos aprender. El tamaño de los modelos depende principalmente del tamaño de los *bloom filters.* En nuestros implementación, el modelo requiere solo 50 MB para dos *bloom filter* capaces de almacenar alrededor de 10.000.000 de elementos cada uno (para un óptimo ratio de falsos positivos), pero si reservamos dos *bloom filter* capaces de almacenar 100.000.000 cada uno, el modelo ya requiere alrededor de 500 MB.

Tanto la enorme cantidad de tiempo que se requiere para entrenar, como el enorme espacio requerido para almacenar una gran cantidad de elementos, son limitaciones importantes. Estas limitaciones nos parecieron suficientes para recomendar Anagram solo para proteger sitios en donde el tráfico no sea abismal y que además sea bastante estable, o sea, no cambie demasiado con el tiempo.

Si bien las limitaciones parecen ser bastante importantes, creemos que Anagram es un buen agregado para la plataforma de detección, entrenando modelos para proteger solo aquellas máquinas que requieran una seguridad mayor, debido a su información sensible. Convenientemente, estas maquinas suelen tener un tráfico más acotado por motivos de seguridad, siendo ideales para una implementación de Anagram.

Tal como podemos ver en la figura 5.2, agregamos en nuestra plataforma de detección una versión de Anagram como un módulo individual (Dapagram). Nuestra versión ofrece tanto el modo de entrenamiento *offline* como el *online*, aunque de acuerdo a los experimentos se recomienda el modo *offline*. Una vez entrenado y configurado en el sitio que se desea proteger, el módulo generará alertas sobre los paquetes anómalos que detecte, y éstas serán almacenadas en nuestra base de datos indexada, pudiendo observarlas en tiempo real en el tablero de alertas.

Capítulo 7

Conclusiones

7.1. Conclusiones

La rápida y constante naturaleza evolutiva de los ataques maliciosos reduce el potencial de los algoritmos de aprendizaje automatizado, que son mayormente efectivos en problemas con distribución de datos mayormente estable en el tiempo. Surgieron entonces varias preguntas. ¿Es posible reproducirlos? ¿Qué tan efectivos son? ¿Qué alcance tienen? ¿Qué tanto afecta utilizar un modelo de detección de ataque maliciosos en redes de diferentes países? ¿qué se necesita para utilizarlas con éxito?

Comenzamos este trabajo intentando responder estas preguntas y a lo largo de nuestros experimentos logramos responderlas.

La primera pregunta que nos hicimos fue si es posible reproducir los algoritmos y los resultados que se prometen. Para responder esta inquietud creemos que es adecuado separarla en distintas cuestiones. Por un lado podemos hablar propiamente de la implementación de los detectores, que gracias a las librerías libres que se ofrecen desde la comunidad para el lenguaje Python, tanto para el manejo de información (Pandas [40]) como para las técnicas de aprendizaje automatizado (ScikitLearn [46]), es realmente simple crear muy buenos algoritmos y realmente está al alcance de cualquiera con una formación mínima adecuada en el área. Por otro lado, si hablamos de reproducir los resultados publicados la situación es un poco más complicada. Como pudimos ver en nuestros experimentos, en algunos casos obtener un conjunto de datos de entrenamiento de calidad es posible, logrando una reproducción exitosa y de muy buenos resultados (como en el caso del Detector de DGA 6.1), mientras que en otros casos puede ser extremadamente complicado o incluso imposible siquiera obtener una cantidad mínima aceptable de datos necesarios para entrenar nuestros modelos (tal fue el caso del Detector de TLS Malicioso 6.3). Lamentablemente, no son muy comunes los casos en donde se compartan los conjuntos de datos recolectados, haciendo muy difícil implementar y entrenar correctamente algunos modelos, limitando enormemente el potencial del área.

Otra inquietud que surgió fue saber si los modelos publicados serían igualmente de efectivos con datos de redes locales o si por el contrario solo funcionan correctamente con el tráfico perteneciente a los países en donde se realizaron sus estudios. Veamos nuestros experimentos. En el caso del detector de DGA obtuvimos muy buenos resultados sin tener que agregar datos locales, logrando detectar correctamente posibles dominios generados algorítmicamente con muy baja cantidad de falsos positivos. Pero que tan diferentes son los dominios locales a los utilizados en el entrenamiento?. Como mencionamos durante los experimentos, para entrenar el modelo se utilizó una lista de los dominios mas populares en el mundo, obtenidos de Alexa [1], una lista de palabras del diccionario inglés y por supuesto una colección de dominios generados algorítmicamentes. Si analizamos los sitios que suelen visitarse en nuestras redes locales veremos que no difieren demasiado de los utilizados en el entrenamiento. Por un lado, en nuestro país se consume mucha cultura de países de habla inglesa, visitando muchos de los mismos dominios listados en Alexa, y por otro lado, los sitios locales en español tienen cierta similitud en su formas debido que utilizan el mismo alfabeto y derivan de un mismo lenguaje. Sin duda, el conjunto de datos de entrenamiento sería prácticamente inútil si estuviéramos probando en redes de países orientales en donde utilizan otros alfabetos. En el caso del detector de URLs maliciosas 6.2 pudimos observar claramente la importancia de la procedencia de los datos de entrenamiento. Mostramos como mejoraba significativamente los resultados agregando URLs locales a nuestro conjunto de datos de entrenamiento. Aunque también dudamos seriamente de la eficacia del modelo publicado en redes del país en donde se realizó el estudio, ya que resulta raro pensar que en tal país no accedan a la URL `www.google.com`, la cual el modelo detectó como maliciosa. Finalmente, en el caso de Anagram 6.4, al ser un detector de anomalías, el modelo debe necesariamente entrenarse con los datos de la red en la cual se desea implementar, por lo que en este caso la pregunta no es válida realmente.

La pregunta más importante que nos planteamos es si los métodos propuestos son realmente efectivos en redes reales localizadas en empresas o instituciones del país. Nuestros experimentos muestran que sí, pero muchas veces con limitaciones o al menos consideraciones a tener en cuenta. El principal punto es que los datos utilizados en varios detectores no fueron muy buenos, causando bastante ruido en el análisis y produciendo no tan buenos resultados. Otro aspecto a tener en cuenta es que el problema a resolver es bastante difícil, debido a su naturaleza cambiante y evolutiva, provocando la necesidad de actualizar constantemente los detectores.

Finalmente, nuestros experimentos nos mostraron que si bien existe un gran potencial en el área de aprendizaje automatizado para la detección de ataques maliciosos, lamentablemente el área también proporciona grandes ventajas a los ciber-delincuentes, ya que los usuarios no suelen cambiar de acuerdo al comportamiento de los delincuentes, mientras que estos últimos si cambian respecto al conocimiento y seguridad que utilizan los usuarios.

7.2. Trabajos Futuros

Hay una inmensa cantidad de trabajos que pueden desprenderse sobre la detección de ataques maliciosos con aprendizaje automatizado. Hay muchos métodos distintos y es un área suficientemente grande como para encarar el problema de diferentes maneras. Pero centrándonos en nuestros resultados podemos listar algunas tareas en particular.

La primera es bastante lógica, pero difícil de lograr. Obtener datos de mejor calidad que permitan mejorar las evaluaciones. Y profundizar en la búsqueda de datos requeridos para detectar tráfico malicioso en redes cifradas (https).

También sería interesante probar restringiendo la variedad de datos a un área especifica, como por ejemplo Bancos, logrando quizás reducir el "ruido" en los datos.

Otro tarea interesante es ampliar el número de técnicas utilizadas, como por ejemplo, investigar y probar otros métodos de detección de anomalías.

En este trabajo se comenzó con métodos sencillos más fáciles de implementar, pero sin duda sería interesante explorar el problema utilizando métodos más avanzados, como por ejemplo, Redes Profundas (*Deep Learning*).

Bibliografía

[1] Alexa. The top 500 sites on the web. `http://www.alexa.com/topsites`, 2014.

[2] Ammar Almomani, BB Gupta, Samer Atawneh, A Meulenberg, and Eman Almomani. A survey of phishing email filtering techniques. *IEEE communications surveys & tutorials*, 15(4):2070–2090, 2013.

[3] Blake Anderson, Subharthi Paul, and David McGrew. Deciphering malware use of tls (without decryption). *arXiv preprint arXiv:1607.01639*, 2016.

[4] Apache Lucene. Solr. `http://lucene.apache.org/solr/`, 2017.

[5] James Blustein and Amal El-Maazawi. Bloom filters. a tutorial, analysis, and survey. *Halifax, NS: Dalhousie University*, pages 1–31, 2002.

[6] Andrew P Bradley. The use of the area under the roc curve in the evaluation of machine learning algorithms. *Pattern recognition*, 30(7):1145–1159, 1997.

[7] Peter A Burrough, Pauline FM van Gaans, and RA MacMillan. High-resolution landform classification using fuzzy k-means. *Fuzzy sets and systems*, 113(1):37–52, 2000.

[8] J Michael Butler. Finding hidden threats by decrypting ssl. 2013.

[9] Sung-Hyuk Cha and Sargur N Srihari. A priori algorithm for sub-category classification analysis of handwriting. In *Document Analysis and Recognition, 2001. Proceedings. Sixth International Conference on*, pages 1022–1025. IEEE, 2001.

[10] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.

[11] Cisco. Hiding in Plain Sight: Malwares Use of TLS and Encryption. `http://blogs.cisco.com/security/malwares-use-of-tls-and-encryption`, 2016.

[12] Ton J Cleophas, Aeilko H Zwinderman, and Henny I Cleophas-Allers. *Machine learning in medicine*. Springer, 2013.

[13] Michael K Daly. Advanced persistent threat. *Usenix, Nov*, 4, 2009.

[14] Ramón Díaz-Uriarte and Sara Alvarez De Andres. Gene selection and classification of microarray data using random forest. *BMC bioinformatics*, 7(1):3, 2006.

[15] Elastic. Kibana. `https://www.elastic.co/products/kibana`, 2017.

[16] What APT Means To Your Enterprise and Greg Hoglund. Advanced persistent threat.

[17] FSecurify. Using Machine Learning to Detect Malicious URLs. `http://fsecurify.com/using-machine-learning-detect-malicious-urls/`, 2016.

[18] Terrence S Furey, Nello Cristianini, Nigel Duffy, David W Bednarski, Michel Schummer, and David Haussler. Support vector machine classification and validation of cancer tissue samples using microarray expression data. *Bioinformatics*, 16(10):906–914, 2000.

[19] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *computers & security*, 28(1):18–28, 2009.

[20] Google. Waymo. `https://waymo.com/`, 2016.

[21] Steve R Gunn et al. Support vector machines for classification and regression. *ISIS technical report*, 14:85–86, 1998.

[22] Shuang Hao, Matthew Thomas, Vern Paxson, Nick Feamster, Christian Kreibich, Chris Grier, and Scott Hollenbeck. Understanding the domain registration behavior of spammers. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 63–76. ACM, 2013.

[23] Jay Jacobs. Building a DGA Classifier. `http://datadrivensecurity.info/blog/posts/2014/Sep/dga-part1/`, 2014.

[24] James M Joyce. Kullback-leibler divergence. In *International Encyclopedia of Statistical Science*, pages 720–722. Springer, 2011.

[25] Oleg Kolesnikov and Wenke Lee. Advanced polymorphic worms: Evading ids by blending in with normal traffic. Technical report, Georgia Institute of Technology, 2005.

[26] Igor Kononenko. Machine learning for medical diagnosis: history, state of the art and perspective. *Artificial Intelligence in medicine*, 23(1):89–109, 2001.

[27] Terran Lane and Carla E Brodley. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conference*, volume 377, pages 366–380. Baltimore, USA, 1997.

[28] K Ming Leung. Naive bayesian classifier. *Polytechnic University Department of Computer Science/Finance and Risk Engineering*, 2007.

[29] Zhenkai Liang and R Sekar. Fast and automated generation of attack signatures: A basis for building self-protecting servers. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 213–222. ACM, 2005.

[30] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991.

[31] lucidworks. Banana. `https://github.com/lucidworks/banana`, 2017.

[32] Justin Ma, Lawrence K Saul, Stefan Savage, and Geoffrey M Voelker. Learning to detect malicious urls. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):30, 2011.

[33] Malware-Traffic-Analysis. A source for pcap files and malware samples. `http://malware-traffic-analysis.net/`, 2017.

[34] Darren Manners. The user agent field: Analyzing and detecting the abnormal or malicious in your organization. *SANS Institute reading room site*, 2012.

[35] Ryszard S Michalski, Jaime G Carbonell, and Tom M Mitchell. *Machine learning: An artificial intelligence approach*. Springer Science & Business Media, 2013.

[36] David Moore, Colleen Shannon, Geoffrey M Voelker, and Stefan Savage. Internet quarantine: Requirements for containing self-propagating code. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, volume 3, pages 1901–1910. IEEE, 2003.

[37] Imran Naseem, Roberto Togneri, and Mohammed Bennamoun. Linear regression for face recognition. *IEEE transactions on pattern analysis and machine intelligence*, 32(11):2106–2112, 2010.

[38] Netflix. Netflix Prize. `https://www.netflixprize.com/`, 2009.

[39] OISF. Suricata. `https://suricata-ids.org/`, 2017.

[40] Pandas. Pandas, Python Data Analysis Library. `https://pandas.pydata.org/`, 2017.

[41] Feng Qian, Abhinav Pathak, Yu Charlie Hu, Zhuoqing Morley Mao, and Yinglian Xie. A case for unsupervised-learning-based spam filtering. In *ACM SIGMETRICS Performance Evaluation Review*, volume 38, pages 367–368. ACM, 2010.

[42] Russell Reed. Pruning algorithms-a survey. *IEEE transactions on Neural Networks*, 4(5):740–747, 1993.

[43] Payam Refaeilzadeh, Lei Tang, and Huan Liu. Cross-validation. In *Encyclopedia of database systems*, pages 532–538. Springer, 2009.

[44] VF Rodriguez-Galiano, M Chica-Olmo, F Abarca-Hernandez, Peter M Atkinson, and C Jeganathan. Random forest classification of mediterranean land cover using multi-seasonal imagery and multi-seasonal texture. *Remote Sensing of Environment*, 121:93–107, 2012.

[45] scans.io. Internet-Wide Scan Data Repository. `https://scans.io/study/sonar.ssl`, 2017.

[46] Scikit-Learn. scikit-learn, Machine Learning in Python. `http://scikit-learn.org/`, 2017.

[47] sklearn. Kmeans. `http://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`, 2018.

[48] sklearn. PCA. `http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html`, 2018.

[49] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305–316. IEEE, 2010.

[50] Yingbo Song, Angelos D Keromytis, and Salvatore J Stolfo. Spectrogram: A mixture-of-markov-chains model for anomaly detection in web traffic. In *NDSS*, volume 9, pages 1–15. Citeseer, 2009.

[51] sslbl.abuse.ch. SSL Blacklist. `https://sslbl.abuse.ch/blacklist/`, 2017.

[52] Stuart Staniford, Vern Paxson, Nicholas Weaver, et al. How to own the internet in your spare time. In *USENIX Security Symposium*, volume 2, pages 14–15, 2002.

[53] Salvatore J Stolfo, Ke Wang, and Wei-Jen Li. Towards stealthy malware detection. In *Malware Detection*, pages 231–249. Springer, 2007.

[54] trendmicro. Can malware be spotted in TLS without having to decrypt the traffic. `http://blog.trendmicro.com/can-malware-be-spotted-in-tls-without-having-to-decrypt-the-traffic/`, 2016.

[55] Venturebeat.com. Pandora uses machine learning to make sense of 80 billion thumb votes. `https://venturebeat.com/2017/07/12/pandora-uses-machine-learning-to-make-sense-of-80-billion-thumb-votes/`, 2017.

[56] Kiel Wadner. seconds on the wire: A look at malicious traffic. *Recuperado de http://www. sans. org/reading-room/whitepapers/detection/60-seconds-wire-malicious-traffic-34307*, 2013.

[57] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 255–264. ACM, 2002.

[58] Ke Wang, Janak J Parekh, and Salvatore J Stolfo. Anagram: A content anomaly detector resistant to mimicry attack. In *International Workshop on Recent Advances in Intrusion Detection*, pages 226–248. Springer, 2006.

[59] Wireshark. Libpcap File Format. `https://wiki.wireshark.org/Development/LibpcapFileFormat`, 2015.

[60] Wireshark Foundation. Tshark. `https://www.wireshark.org/docs/man-pages/tshark.html`, 2017.

[61] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. A close look on n-grams in intrusion detection: anomaly detection vs. classification. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 67–76. ACM, 2013.

[62] Sandeep Yadav, Ashwath Kumar Krishna Reddy, AL Reddy, and Supranamaya Ranjan. Detecting algorithmically generated malicious domain names. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 48–61. ACM, 2010.

[63] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. On early stopping in gradient descent learning. *Constructive Approximation*, 26(2):289–315, 2007.