E.T.S. de Ingeniería Industrial,
Informática y de Telecomunicación

Detección de dominios *typosquatting*

Grado en Ingeniería
en Tecnologías de Telecomunicación

Trabajo Fin de Grado

Autor: Javier Artiga Garijo

Directora: Silvia Díaz Lucas

Pamplona, 21 de noviembre de 2018

upna Universidad Pública de Navarra
Nafarroako Unibertsitate Publikoa

Palabras Clave

Typosquatting, Cybersquatting, Seguridad informática, Ciberseguridad, Python, Bash, Elasticsearch, Kibana, Telefónica, ElevenPaths

Resumen

Diseño y desarrollo (usando Python y Bash) de un sistema de monitorización de dominios *typosquatting*, basado en recoger información con peticiones DNS, Whois y HTTP y almacenarla en un servidor de búsqueda Elasticsearch, con el objetivo de analizar la viabilidad técnica de esta solución para ElevenPaths.

Key Words

Typosquatting, Cybersquatting, Computer security, Cybersecurity, Python, Bash, Elasticsearch, Kibana, Telefonica, ElevenPaths

Abstract

Design and development (using Python and Bash) of a *typosquatting* domains monitoring system, based on collecting information with DNS, Whois and HTTP requests and storing it into an Elasticsearch search server, with the aim of analyzing the technical feasibility of this solution for ElevenPaths.

Nomenclatura

ACPA	Anticybersquatting Consumer Protection Act
API	Application Programming Interface
CSV	Comma Separated Values
DNS	Domain Name System
GPL	The GNU General Public License
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
ICANN	Internet Corporation for Assigned Names and Numbers
IDN	Internationalized Domain Name
IP	Internet Protocol
JSON	JavaScript Object Notation
NRT	Near Real Time
PoC	Proof of Concept
REST	REpresentational State Transfer
SMTP	Simple Mail Transfer Protocol
TLD	Top-Level Domain
UDRP	Uniform Domain-Name Dispute-Resolution Policy
URL	Uniform Resource Locator
VPN	Virtual Private Network

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Contexto

El origen de este trabajo se enmarca dentro del "**Reto Ciberseguridad y Big Data**"[1] que propuso ElevenPaths, la unidad global de ciberseguridad del Grupo Telefónica, para el año 2018. Esta iniciativa facilita la incorporación de estudiantes en proyectos reales de la compañía, usando *datasets* de Telefónica y con la mentorización de sus profesionales. Entre los 21 proyectos ofertados, se incluía el que aquí se desarrolla. Lo elegí porque conocía previamente el tema del *Typosquatting* y me atrajo la idea de profundizar tanto en las técnicas que se emplean para llevarlo a cabo como en la metodología para detectarlo. Además, la escalabilidad y la mejora de la calidad de los resultados suponían un desafío interesante. Por otro lado, mi perfil cumplía con los requisitos demandados: desarrollo en Python y uso de APIs de social media. También se indicaban como deseables ciertos conocimientos de *clustering*, aspecto que dominaba menos pero en el que quería ampliar mis competencias.

La descripción exacta del proyecto era la siguiente:

"El proyecto ya se encuentra **en desarrollo** por parte de Telefónica como herramienta interna para la **detección automática de dominios sospechosos** relativos a un determinado cliente. La detección procede por diferentes vías, que incluyen la generación automática de *typos* fonéticos, semánticos y "de teclado", la búsqueda de dominios oportunista a partir de

las redes sociales y la detección de patrones entre los dominios detectados como sospechosos.

Lo que se espera es mejorar la detección automática de dominios oportunistas y encontrar patrones entre los dominios detectados. Se espera además priorizar adecuadamente estos dominios detectados según su peligrosidad. Por último, se espera mejorar la calidad y cantidad de información proporcionada al analista final y pulir la interfaz ofrecida al analista para que este proporcione feedback sobre los dominios detectados.

Hitos:

- Mejora de la parte de detección de dominios sospechosos a partir de redes sociales.
- Identificación automática de patrones entre dominios detectados.
- Definición de nuevos criterios de priorización entre dominios sospechosos."

1.2. Propuesta

Tras completar la solicitud a través de la web del programa "Open Future_" de Telefónica y ser entrevistado a finales de abril de 2018 por Mariano González Espín (Product Manager de Ciberseguridad en Telefónica y tutor de este proyecto), se me ofreció trabajar en este sistema de detección de dominios *typosquatting*. Acordamos que me incorporaría al proyecto el 14 de junio, colaborando de manera remota (lo que permitiría que los miembros del equipo se encontrasen en distintas ciudades) y guiado por Julio Gómez Ortega (Product Manager y Consultor de Seguridad de Telefónica) y Aruna Prem Bianzino (Investigador en el departamento de Innovación de Elevenpaths).

Ese día, en una reunión por vía telemática se explicó lo que hasta entonces Aruna había desarrollado. El sistema tomaba como entrada las marcas del cliente y a través de peticiones Whois y DNS descubría sus dominios potenciales. Si se observaban cambios en los datos recogidos sobre el dominio, se aumentaba su prioridad y se notificaba por email.

Con este sistema como base, se pretendía mejorarlo y escalarlo a un mayor número de clientes. Este proceso se articuló en 3 fases:

- Verificar la existencia del dominio de una marca con diferentes TLDs.
- Generar variaciones en los nombres de dominio y verificar la existencia de los resultados.
- De ser posible, mejorar la monitorización de las marcas en redes sociales (especialmente, en los Trending Topics de Twitter).

El **escalado a múltiples clientes** (por un factor del orden de varias decenas) planteaba diversos **retos técnicos**, como evitar posibles bloqueos de APIs u operar con un número de combinaciones mucho mayor que el actual pero todavía desconocido. Por ello, un aspecto importante era medir la *performance* en cada fase (cuántas peticiones se hacían, con qué frecuencia, cuánto tiempo total costaba cada paso en función del volumen de datos tratado, etc.).

1.3. Motivación

Mediante esta colaboración, la empresa buscaba reforzar el personal del Departamento de Innovación de Elevenpaths, encargado de prototipar en primera instancia el sistema de detección de dominios *typosquatting* que deseaban incorporar a su servicio de ciberseguridad para organizaciones.

A nivel personal, mi intención era tomar contacto con el mundo de la gran empresa y su modo de funcionar, aplicando por primera vez los conocimientos adquiridos durante la carrera a un entorno real. De esta experiencia también esperaba aprender nuevas aptitudes útiles en el plano profesional y constatar la importancia de tener asentados ciertos conocimientos técnicos y teóricos para desarrollar tareas de cierto nivel con soltura.

Este proyecto en concreto posibilitaba trabajar, por un lado, en coordinación con un equipo (manteniendo una comunicación permanente para conocer los progresos de cada uno, cumpliendo unos plazos para garantizar el progreso

constante del proyecto), y por otro, sobre una base dada (algo frecuente en las empresas, y más en el desarrollo de software) que habría que estudiar, comprender y aprender a utilizar para aprovechar los avances ya logrados.

1.4. Objetivo

Para concretar de manera clara, breve y directa en qué consistiría la labor que desarrollaría en el proyecto, se definió el siguiente objetivo:

Trabajaría en el diseño y desarrollo de un sistema de monitorización de dominios sospechosos basado principalmente en peticiones DNS. Mi función sería **analizar la viabilidad técnica de la solución**.

Dentro de ese análisis, se validaría:

- El número de peticiones que pueden realizarse sin baneo.
- El número de peticiones que cada máquina puede hacer al día.

1.5. Estado del arte

El *typosquatting* es una forma de *cybersquatting*[1], basada en el aprovechamiento de los **errores tipográficos** (erratas) que un usuario pueda cometer generalmente al introducir una dirección web en un navegador o al seguir un hipervínculo sin comprobar el destino al que apunta. Normalmente se enfocan a marcas conocidas, pero incluyendo una errata que pase lo más desapercibida posible.

Las razones que llevan a un *typosquatter* a comprar un dominio de este tipo suelen estar relacionadas con la **obtención de beneficios económicos** a través de varias maneras: venderlo al propietario de la marca, monetizarlo mediante publicidad, redirigir su tráfico o bien hacia la competencia o bien hacia la propia marca pero usando un enlace afiliado que genera una comisión,

[1] También conocido como *domain squatting*, se traduce como ocupación de dominios.

etc. Otros métodos para extraer un rendimiento económico de estos dominios son los relacionados con *phising*[2], frecuentemente haciéndose pasar por el sitio legítimo, o incluso instalando *malware*[3] o *adware*[4]. También puede aprovecharse el *typosquatting* con fines más políticos, como la expresión de una opinión diferente a la de la web original.

Un suceso que ilustra la relevancia de esta técnica es el del jucio de **Facebook** en 2013, cuando los juzgados de California del Norte fallaron a su favor en una acción legal emprendida contra 105 dominios *typosquatting*[2]. Estos dominios se devolvieron a la red social (existe para ello un procedimiento establecido por la ICANN llamado UDRP, además de una ley estadounidense de 1999 conocida como ACPA) junto con una indemnización de $2 800 000.

La práctica del *typosquatting* con estos objetivos fraudulentos se ha observado **durante toda la popularización de Internet**. En 2018, un interesante informe de la empresa de seguridad informática Sophos[3] ha revelado que, de media, un 77 % de las direcciones *typosquatting* analizadas para los sitios web de Apple, Google, Facebook, Twitter y Microsoft eran dominios activos. Pero ya en 2009, se estimaba que al menos 938 000 dominios *typosquatting* atacaban a los 3264 sitios .com más populares[4].

Otro caso más reciente y cercano puede ser el descrito por Chema Alonso en un artículo del blog elladodelmal.com[5] fechado en 2015, un ejemplo real en el que se suplanta a la empresa **AirBnB** con el dominio airbnb.directory y después con un *e-mail spoofing*[5] para que la víctima ingrese cierta cantidad de dinero en la cuenta bancaria del estafador. Son fraudes frecuentes y repetidos de manera periódica, como demuestra otro artículo del mismo blog publicado un año después[6].

[2]Se traduce como suplantación de identidad. Este modelo de abuso informático se caracteriza por intentar adquirir información confidencial (como contraseñas o información bancaria) de forma fraudulenta.

[3]En español "programa malicioso", es un tipo de software diseñado intencionadamente para infiltrarse o dañar un dispositivo informático o red.

[4]O "software publicitario", se refiere a cualquier programa que muestra automáticamente publicidad no deseada para lucro de su desarrollador

[5]Creación de un mensaje de correo electrónico con una dirección de remitente falsa

A continuación, se citan varias marcas con alcance mundial que han sufrido *cybersquatting* a lo largo de los años[7].

Nombre de dominio	Entidad afectada o reclamadora	Fecha de resolución
atleticodemadrid.com	Club Atlético de Madrid, S.A.D.	Diciembre 2015
facebookghana.com	Facebook, Inc.	Agosto 2015
hillaryclinton.net	Hillary Diane Rodham Clinton	Mayo 2015
googleglass.es	Google, Inc.	Abril 2014
twitter.co.kr	Twitter, Inc.	Febrero 2014
i-tune.com itunes.net	Apple, Inc.	Junio 2012
allianz-es.com	Allianz, S.A.	Septiembre 2009
googblog.com	Google, Inc.	Marzo 2009
bbvaneet.com	BBVA, S.A.	Diciembre 2005
mikerowesoft.com	Microsoft Corporation	Enero 2004
wwwdinersclub.com	Diners Club International, Ltd.	Julio 2003
pepsix.com pepsixxx.com	PepsiCo, Inc.	2003
chatyahoo.com yahoonews.com yahow.com	Yahoo!, Inc.	Octubre 2000
pfiezer.com pfzerforliving.com	Pfizer, Inc.	Agosto 2000

Cuadro 1.1: Algunos casos de *cybersquatting* entre 2000 y 2015

Muchas empresas del sector de la seguridad informática ofrecen a sus clientes servicios de protección frente a suplantaciones por *typosquatting*, cubriendo un cierto rango de peligros en función de las posibilidades de la organización y la inversión realizada. Por parte de Telefónica, su unidad ElevenPaths cuenta con **CyberThreats**, un producto que afronta la gestión de estos y otros riesgos de forma holística. Entre los ataques que detecta se halla el de *typosquatting*, que supone una amenaza atribuible a diversos escenarios: uso no autorizado de marca, suplantación/falsificación, robo de credenciales, fraude online y sus derivados, fugas de información, etc.

Estas soluciones normalmente se basan en:

- Generar variaciones de las URLs que nos interesa proteger, mediante todas las técnicas relacionadas con errores de escritura que se quieran considerar: adición/repetición/supresión/transposición de letras, inserción de teclas cercanas en el teclado, sustitución por caracteres homóglifos, separación por puntos o guiones, etc. La longitud de esta lista depende de los recursos de los que se disponga.
  El control del mayor número de variaciones posible permite una actitud proactiva ante el registro posterior de alguna de estas variaciones, disponiendo así de la capacidad de detección temprana que se desea.
- Comprobar la existencia de estos dominios con peticiones DNS y Whois.
- Verificar que en ese dominio se sirve un sitio web a través de peticiones HTTP y HTTPS.
- Es posible profundizar más en el análisis del sitio escaneando su contenido con *crawlers* o incluso capturas de pantalla.
- Toda esta información puede enriquecerse buscando relaciones en el conjunto de los datos recogidos (patrones).
- Adicionalmente, se puede efectuar un rastreo en redes sociales y webs de referencia para localizar posibles campañas fraudulentas.

1.6. Antecedentes

Dado que el proyecto partía de un estado en desarrollo, antes de emprender cualquier acción nueva era absolutamente recomendable estudiar el código existente. Se me proporcionaron varios ficheros de documentación, entre los que destacaban los mencionados a continuación.

1.6.1. Typosquatting Report

Existía un informe interno, con fecha de abril de 2017, en el que se explicaba en profundidad la solución que Elevenpaths ofrecía en aquel momento. Al acabar el presente trabajo se elaboró un nuevo informe[8] que pudiera ser citado públicamente.

Ese primer informe advertía de la cantidad de intervención manual que requería aquel proceso, además de producir una amplia mayoría de falsos positivos. También se señalaba la dependencia de servicios externos para obtener información sobre el estado de registro de un dominio.

La arquitectura propuesta para mejorar aquella solución consistía en los siguientes bloques:

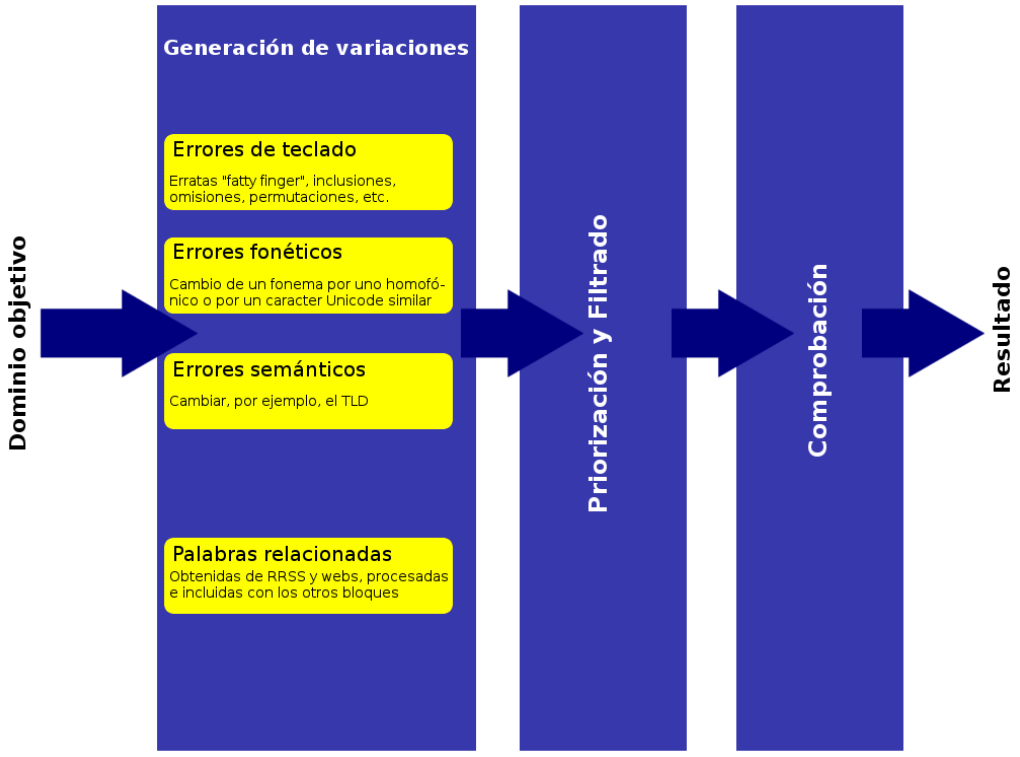


Figura 1.1: Arquitectura de la solución propuesta en *Typosquatting Report*

Cada bloque se implementaba de la siguiente manera:

- Para los errores de teclado, fonéticos y semánticos se empleaba dnstwist, herramienta que se ampliará en la siguiente sección.
- Las palabras relacionadas se obtenían a partir de twofi[6] y CeWL[7], filtrando sendas salidas con otro script en Python que las comparaba antes de incluirlas junto con las demás variaciones.
- En el filtrado se eliminaban los dominios propios del cliente (*whitelist*) y los resultados en los que el cliente no estaba interesado (*blacklist*).
- La priorización establecía la frecuencia con la que se comprobaría cada dominio.
- El bloque de comprobación se determinaba si el dominio estaba registrado y si resolvía a una página web.

En las pruebas de *stress* efectuadas sobre este último bloque se completaba una lista de 55641 dominios en 17 horas (lo que implica aproximadamente 1,1 segundos por dominio).

Se pusieron a prueba algunos servicios externos para valorar la idoneidad de su incorporación a la solución, pero se llegó a la conclusión de que, aunque el diccionario de variaciones que generaban era mayor, normalmente el diccionario que devolvían contenía dominios menos relacionados con el cliente, su resolución suponía un mayor retardo y además no incluía la clasificación por prioridad ni el *input* de redes sociales.

[6] Twitter Words of Interest: https://digi.ninja/projects/twofi.php
[7] Custom Word List generator: https://digi.ninja/projects/cewl.php

1.6.2. PoC Enel

Desarrollado por Daniel Belmar García en calidad de becario para ElevenPaths, entregado en julio de 2017. En la documentación, el autor recogió así el funcionamiento de su código:

"Hay tres scripts escritos en Python 2 que se encargan de:

1. **tsinsert.py**: Coge datos desde un .txt donde están ciertos dominios y subdominios considerados importantes, los cataloga, filtra y crea un .json con los datos obtenidos a partir de ciertas librerías como whois, dns y socket. Después los inserta en Elasticsearch. Este script se debe ejecutar una sóla vez.

2. **ts-updater.py**: Es un script muy similar al anterior. Coge los dominios desde Elasticsearch y comprueba si ha llegado el día de buscar los datos de cada uno nuevamente en la librería whois. Si es así, se ejecuta. Si no, sigue recorriendo los dominios. Está pensado para que no haya que ejecutarlo una y otra vez (por eso está dentro de un while True).

3. **tsnews_domains_uploader.py**: Se ejecuta en crontab, con lo cual no hay que preocuparse de su ejecución. Es un script pensado para enviar un correo electrónico si se ha añadido algún dominio al .txt donde se encuentran originalmente los dominios y subdominios."

Esta prueba de concepto sirvió de análisis de viabilidad para prestar un servicio de protección frente a amenazas *typosquatting* a la multinacional productora y distribuidora de energía Enel.

Capítulo 2

Análisis y diseño

Como primer acercamiento al proyecto, se estudió el código existente, así como las herramientas externas utilizadas en él, y se definieron una serie de directrices a seguir durante el trabajo.

Para favorecer la sencillez de uso y desarrollo, se optó por seguir la primera norma de la **filosofía UNIX**, que dice "*Make each program do one thing well*"[9], así que el diseño del sistema completo se articuló en pequeños scripts. Cada uno debía encargarse de una única tarea, perfeccionando su funcionamiento de manera iterativa a lo largo de las sucesivas versiones. De esta manera, la evolución de cada parte del sistema sería independiente, manteniendo siempre la coherencia en el formato de entrada y salida de cada una de estas partes ("*Expect the output of every program to become the input to another*", dice la filosofía UNIX al respecto) para conservar la interoperabilidad entre ellas, y se favorecería la mejora progresiva del conjunto al permitir trabajar concentrándose en un punto cada vez.

Se decidió aprovechar las ideas del código desarrollado antes, pero portándolo de Python 2 a Python 3, beneficiándonos así de las mejoras de la nueva versión y facilitando el futuro mantenimiento y evolución del código. En definitiva, se haría un ***rework*** a partir del material dado, manteniendo las funcionalidades ya conseguidas y añadiendo las nuevas que se consideraran oportunas.

Se eligió **git**[1] para llevar un control de versiones del código escrito, con el objetivo de disponer de copias de seguridad de cada avance, poder revertir cambios si fuera necesario, consultar de forma rápida el estado del desarrollo y trabajar en funcionalidades provisionales mediante ramas que se fusionarían con la rama principal[2] si finalmente se considerara oportuno.

Aparte, se adoptó la convención de incluir en forma de comentario una serie de datos básicos al principio de cada script. Primero, la directiva *hash-bang[10]* "`#!`" (interpretada por el *kernel* cuando un fichero empieza con esta secuencia, correspondiente en código hexadecimal a `0x23`, `0x21`) apunta dónde se encuentra la ruta del ejecutable que debe procesar el código (en este caso, `#!/usr/bin/env python3`). Después, se indica la codificación con `-*- coding: utf-8 -*-` y algunos "metadatos" del script: autor, fecha y versión. Por último, se incluye una breve descripción de la misión del script y de la sintaxis que sigue el mismo para recibir argumentos.

A continuación, se analizarán las herramientas ajenas en las que se apoya el sistema que se diseñó para este trabajo.

2.1. Elasticsearch

Elasticsearch es un motor de búsqueda basado en Lucene, con interfaz HTTP (es además una aplicación RESTful) y orientado a documentos JSON. Es una plataforma bastante popular a la hora de ofrecer **búsquedas distribuidas a tiempo real (NRT)**, con una latencia del orden de un segundo desde la inserción de un documento hasta que puede buscarse. La API en la que está basada, Lucene, es una librería de código abierto e implementada en Java para recuperación de información, muy usada en aplicaciones que demandan indexado y búsqueda de texto completo.

Durante la primera fase del proyecto, se empleó la **versión 5.2.2**, lanzada en febrero de 2017 y heredada del desarrollo que se hizo ese año sobre este

[1] https://git-scm.com
[2] https://git-scm.com/about/branching-and-merging

proyecto. Para la siguiente fase, se actualizó a la **v6.4.0** (actualmente sigue siendo la última versión), liberada en agosto de este 2018. Entre una y otra versión se incorporaron muchas mejoras, como soporte para Java 10, más opciones en su interfaz gráfica **Kibana**, mayor rapidez y varias correcciones de seguridad.

Conceptos básicos de Elasticsearch:

- **Cluster**: agrupación de uno o varios nodos, coordinados para ofrecer el servicio de motor de búsqueda deseado sobre un conjunto de datos.
- **Nodo**: servidor que participa en un *cluster* de Elasticsearch. Almacena los datos y posibilita las funciones de indexado y búsqueda.
- **Índice**: colección de documentos con características similares. Por ejemplo, en un *cluster* se puede tener un índice con los datos de los clientes y otro con los datos de los pedidos.
- **Documento**: unidad básica de información que puede indexarse. Por ejemplo, un documento puede contener la información de un único cliente y otro, la de un único pedido. Se expresa en formato JSON.

Existen otros conceptos importantes en Elasticsearch, como los *shards* (subdivisiones de índices) y las réplicas de *shards*, pero en este ámbito con los ya mencionados es suficiente para entender el proyecto.

2.2. dnstwist

dnstwist es una herramienta de código abierto (licencia Apache 2.0) creada por el polaco Marcin Ulikowski. La función principal de dnstwist es **tomar un dominio y generar una lista de potenciales dominios de *phising***, para después comprobar si están registrados. Incluye distribución multihilo del trabajo, formato de salida en CSV, JSON o directamente por pantalla, soporte de dominios con caracteres Unicode (IDN), además de un amplio rango de algoritmos de *fuzzing*[3] para generar los nombres de dominio. Existen más funcionalidades, pero estas son las que nos interesa utilizar aquí.

Se ha trabajado sobre su **versión 0.4b** (de noviembre de 2016), adaptándola a nuestras necesidades. En el momento de mi incorporación al proyecto, disponía de una **versión propia** con algunos ajustes que había llevado a cabo Aruna Prem Bianzino para ElevenPaths. Esa versión se hizo dentro de una PoC para Iberdrola. A fecha de hoy, el repositorio oficial de dnstwist[11] ofrece la versión **v20180623**, incluyendo algunas combinaciones de *typosquatting* más y también ligeras modificaciones en los argumentos del programa.

Analizando el código, se observan tres clases importantes:

- `UrlParser` comprueba la validez del dominio mediante una expresión regular construida en base al RFC 3986[12] y lo divide en partes (esquema, dominio, ruta, query) de acuerdo al estándar.
- `DomainFuzz` genera variaciones del dominio aplicando técnicas que después se explicarán en profundidad: *bitsquatting*, *homoglyph*, *hyphenation*, *insertion*, *omission*, *repetition*, *replacement*, *subdomains*, *transposition*, *vowel swap* y *addition*.
- `DomainThread` se encarga de repartir el diccionario en hilos de ejecución, con el fin de acelerar el proceso mediante paralelización, y recoger la información pedida.

[3] El *fuzzing* consiste en generar datos de forma masiva para la entrada de un programa.

En caso de que emplease la opción `--dictionary`, la clase `DomainDict` generaría otro tipo de variaciones adicional: añadiría al dominio las palabras del diccionario que se proporcionase.

La información que se obtiene para cada dominio se recibe mediante diversas peticiones. Si el módulo `dnspython` no está instalado, se establece un socket con el dominio en el puerto 80 para conseguir la dirección IP del dominio. Si está instalado, se hace una petición DNS y se clasifican los registros recibidos como A, AAAA, MX o NS[4]. Adicionalmente, se puede enviar un email de prueba con la opción `--mxcheck` para averiguar si hay un servidor de correo configurado para interceptar emails corporativos dirigidos a una dirección equivocada.

Otras opciones extra son las de recuperar banners de servicios HTTP y SMTP, geolocalizar la IP, hacer consultas Whois (en cuyo caso se desactiva la distribución multihilo) y muchas más, aunque no han sido necesarias para este trabajo.

[4]Registros de DNS: lista de recursos que proporciona el archivo de zona de un dominio.

- A: Registro de dirección [RFC 1035]
- AAAA: Registro de dirección IPv6 [RFC 3596]
- MX: Registro de intercambio del correo [RFC 1035]
- NS: Registro de servidor de nombres [RFC 1035]

Capítulo 3

Desarrollo

A continuación, se detalla cómo se implementó el **sistema de monitorización de dominios *typosquatting*** en Python y Bash para en última instancia determinar la viabilidad técnica de esta solución.

La elaboración del sistema se estructuró en dos fases, ambas funcionales con independencia la una de la otra: una primera fase considerando solo los dominios originales de los clientes (combinándolos con los TLDs encontrados), y después una segunda fase de dimensiones notablemente mayores, al contar con todas las variaciones *typosquatting* posibles sobre esos dominios.

En esta sección solo se describe el funcionamiento de los scripts. En la siguiente sección se explicarán e interpretarán los resultados logrados.

3.1. Infraestructura

Como se operaba en remoto, era necesario conectarse a la red local de Elevenpaths a través de una VPN, donde se encontraban los servidores en los que se desplegaban las pruebas de cada paso. Para empezar, se me facilitaron las credenciales del servidor usado hasta entonces para este proyecto.

El primer servidor era una máquina virtual que contaba con un procesador Intel Xeon E5 (64 bits), 4 GB de RAM (más 2 GB de memoria swap) y un disco duro de 64 GB. Se había instalado Ubuntu 14.

En julio se habilitó un segundo servidor para la siguiente fase. Se había montado sobre otra máquina virtual de iguales características (salvo la versión de Ubuntu, que era la 16). Replicar el sistema desarrollado hasta entonces en una nueva máquina sirvió para confirmar la validez de la documentación que se había elaborado (dependencias, configuraciones, etc.).

3.2. Fase 1

Objetivo:

"Verificar la existencia de los dominios de los clientes con diferentes TLDs".

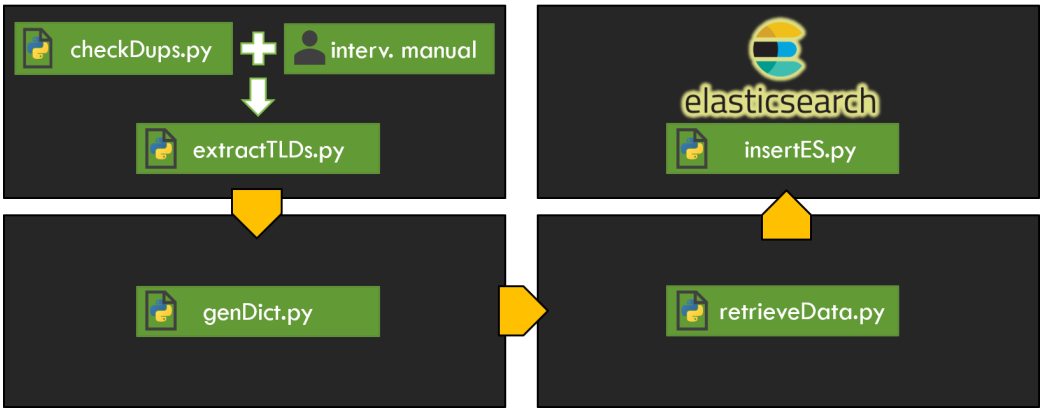


Figura 3.1: Flujo de trabajo general de la fase 1

La intención de la primera fase era desarrollar el sistema completo con todos los pasos necesarios, pero evitando por el momento la dificultad que supondría considerar las variaciones *typosquatting*.

Esta fase se llevó a cabo a lo largo de **5 semanas**.

La fase 1 también incluía como último paso actualizar la base de datos en la que se almacenaba la información recogida de todos los dominios, pero en realidad esta funcionalidad se terminó de implementar durante el desarrollo de la segunda fase.

3.2.1. Paso 1

Paso 1A

Eliminar dominios inválidos y duplicados de los ficheros `.xls`

Al comienzo del proyecto, recibimos una colección de 50 ficheros `.xls`. En ellos, cada cliente había incluido una lista con los nombres de dominio que quería que se vigilasen. En muchos casos, estas listas se habían generado de forma automatizada, por lo que aparecían varios dominios duplicados o inválidos (direcciones IP privadas, o nombres como "redext", "Nuevo Término" o "/"). Estos errores de exportación se filtraron con ayuda de unas pocas líneas de código y eliminando manualmente cada término incorrecto cuando este causaba la interrupción del script siguiente.

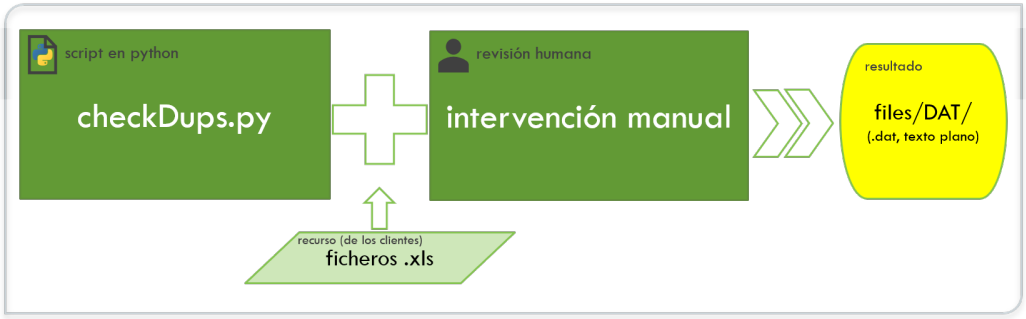


Figura 3.2: `checkDups.py` + intervención manual (fase 1)

Paso 1B

Extraer los ccTLDs que aparecen en los dominios oficiales.

A partir de estas listas depuradas de nombres de dominio (guardadas en unos nuevos ficheros `.dat`), se generaba otra lista de TLDs. Esto es, se recorrían todos los nombres de dominio buscando TLDs (las cadenas a partir del último punto, como ".com" o ".org") que no se hubiesen encontrado antes.

Como hoy en día se pueden registrar dominios con TLDs genéricos (por ejemplo, ".jobs" o ".travel") y hay más de 1200 opciones[13], se decidió limitar el número de posibilidades comprobando si se trataba de un *country-code* TLD (existen 240 ccTLDs, que son códigos geográficos como ".uk" o ".es") antes de incluirlo en el fichero `.txt` del resultado final.

En este paso también se incluía cierto filtrado de los errores de exportación previamente comentados: si el dominio era en realidad una ruta (es decir, aparecía el caracter "/"), se advertía por pantalla y se descartaba todo lo que viniese detrás de "/".

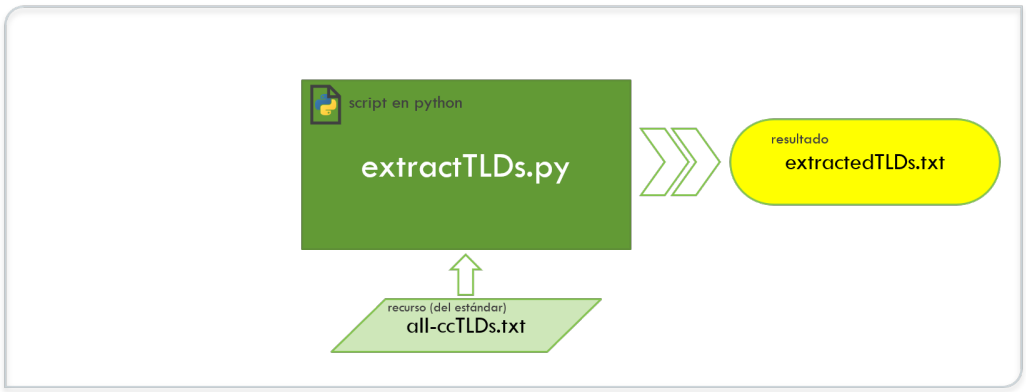


Figura 3.3: `extractTLDs.py` (fase 1)

3.2.2. Paso 2

Generar diccionario (JSON) con todas las combinaciones de dominios oficiales + TLDs.

Una vez se tenían, por un lado, los ficheros con los dominios oficiales y, por otro, la lista de los TLDs presentes en estos dominios, se combinaba cada dominio con todos los TLDs.

Empezamos operando con los 7 TLDs originales (“.com”, “.net”, “.org”, “.edu”, “.gov”, “.mil”, “.int”), para tener una primera aproximación en cuanto a volumen de combinaciones y tiempo que se tardaría luego en recoger su informacion. Después añadimos los 37 TLDs extraídos de los dominios oficiales.

```
1 - 11P       10 doms   (70 combs)
                [...]
50 - TEF_ES    9 doms   (63 combs)
```

```
TOTAL domains:                              2733
TOTAL combinations (with duplicates):  19131
                    removing duplicated domains...
TOTAL COMBINATIONS:                         12894 (6237 duplicates removed)
```

Cuadro 3.1: Salida por pantalla de `genDict.py` con 7 TLDs

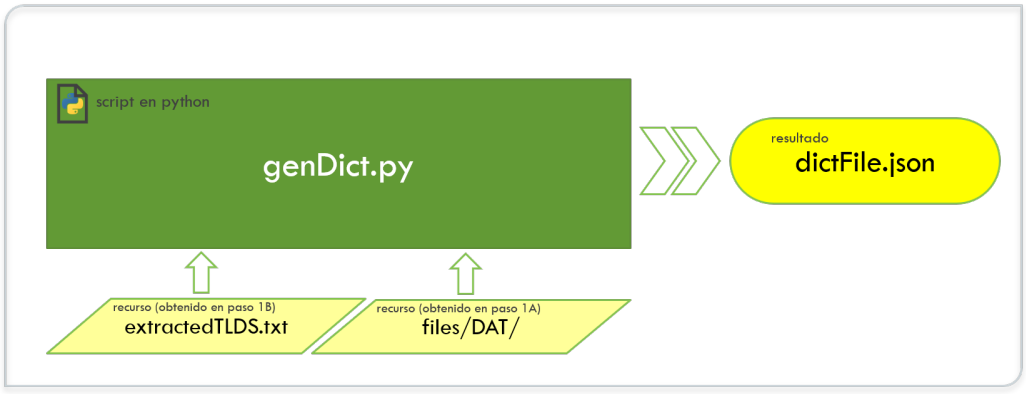


Figura 3.4: `genDict.py` (fase 1)

3.2.3. Paso 3

Obtener información de cada dominio: Whois, IP, peticiones Web.
En consecuencia, asignar campos:
prioridad, frecuencia de comprobación y estado.

Tomando el JSON de antes como entrada para este script, se procesaban los dominios contenidos en el diccionario. En esta fase, mediante 3 funciones se hacían peticiones y se interpretaba su respuesta para asignar un nivel de prioridad a cada dominio:

- `check_whois` consultaba la información relativa al registro del dominio: fecha de creación del dominio y fecha del último cambio (modificación de datos o renovación). Si la petición Whois se resolvía, la fecha de registro del dominio en la base de datos correspondía con la fecha del último cambio indicada en la respuesta. En caso contrario, se fijaba la fecha actual como fecha de registro del dominio en la base de datos.
- `get_ip` establecía un socket con la máquina tras el nombre de dominio y guardaba su dirección IP.
- `check_web` solicitaba por HTTP y por HTTPS la página web alojada en el dominio, asignando un valor booleano en función de la respuesta.

Cada nivel de prioridad implicaba una frecuencia de comprobación determinada: con prioridad alta, se revisaba el dominio cada día; con prioridad baja, cada 14 días. El campo de estado se reservaba para ofrecer información adicional al analista. La siguiente tabla recoge el criterio seguido para decidir el nivel de prioridad de cada dominio:

Prioridad *Estado*		Petición Whois	
		Resuelve	No resuelve
Fecha	Registrado ($<$ 1 semana)	**Alta** - *Por verificar*	**Alta** - *Muy sospechoso*
	Registrado ($>$ 1 semana)	**Baja** - *Aparcado*	**Alta** - *Sospechoso*
	No registrado	–	**Baja** - *Poco sospechoso*

Cuadro 3.2: Niveles de prioridad según el estado del dominio (fase 1)

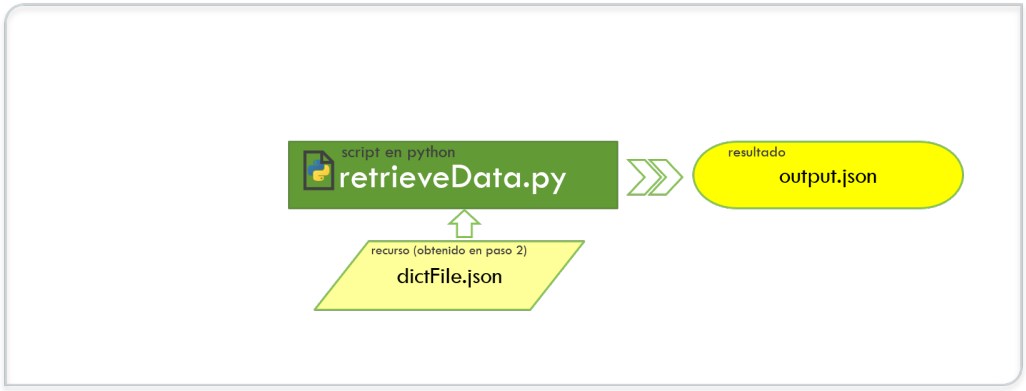


Figura 3.5: `retrieveData.py` (fase 1)

3.2.4. Paso 4

Insertar en Elasticsearch.

Finalmente, la información que se había recogido de cada dominio y se había almacenado en un fichero `.json` se volcaba en la base de datos. Para ello, se creaba un índice de Elasticsearch y se pasaba la información en el cuerpo de una petición HTTP al puerto 9200 de la máquina local. En todo este proceso se hacía uso de la biblioteca `elasticsearch` para Python.

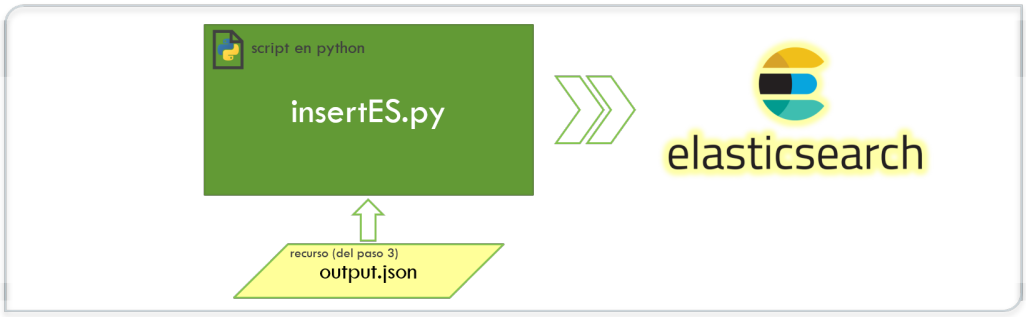


Figura 3.6: `insertES.py` (fase 1)

3.3. Fase 2

Objetivo:

"Incluir las variaciones del nombre de cada dominio".

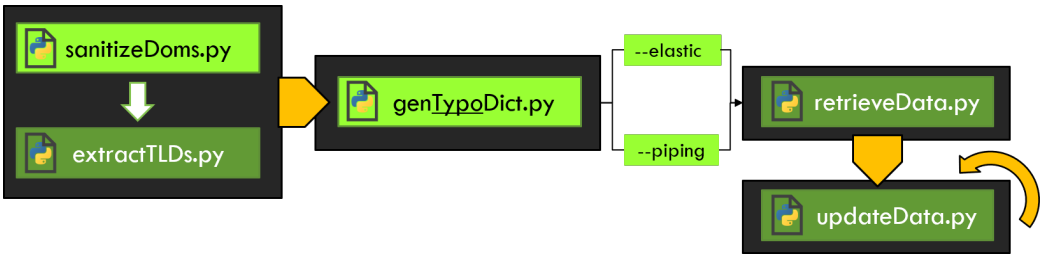


Figura 3.7: Flujo de trabajo general de la fase 2

En la segunda fase se consideraron también las variaciones *typosquatting* obtenidas mediante las técnicas que ofrece dnstwist. El aumento de escala asociado a esta decisión implicó la necesidad de resolver nuevos problemas derivados. Por ejemplo, se presentaron problemas de memoria por el tamaño de los ficheros que `genTypoDict.py` generaba, ante lo cual se adoptaron varios métodos hasta encontrar la solución óptima.

La ejecución de esta fase (junto con las mejoras aplicadas a la fase anterior) duró **10 semanas**.

En todo momento se procuró mantener la retrocompatibilidad de las nuevas funcionalidades con las usadas previamente. Para ello, se añadieron argumentos opcionales a las herramientas de la primera fase o se sustituyeron estas por nuevos programas que conservaban la función del script original y añadían más posibilidades.

3.3.1. Paso 1

Paso 1A

Eliminar dominios inválidos y duplicados de los ficheros `.xls`

Haciendo uso de la clase `UrlParser` de dnstwist, se completó la tarea que antes cumplía `checkDups.py`, reorganizando el código, estructurándolo en funciones separadas y sobre todo **evitando la intervención manual**.

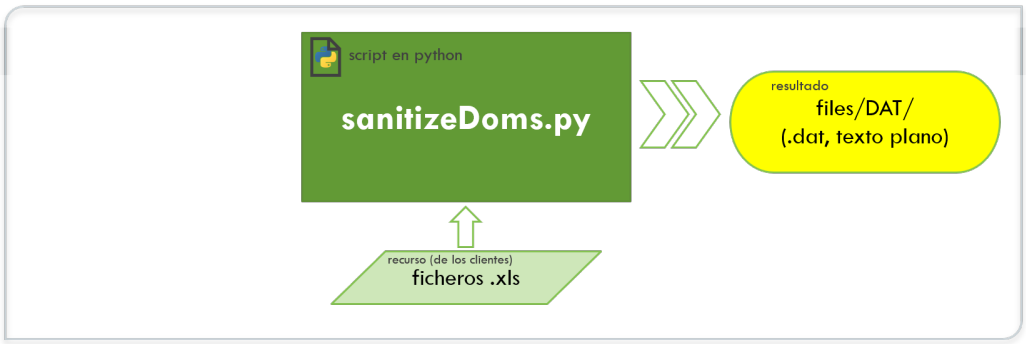


Figura 3.8: `sanitizeDoms.py` (fase 2)

Paso 1B

Extraer los ccTLDs que aparecen en los dominios oficiales.

Se utiliza el mismo script `extractTLDs.py` que en la fase anterior.

3.3.2. Paso 2

Generar diccionario (JSON) incluyendo las variaciones de las combinaciones entre cada dominio oficial y todos los TLDs. Las variaciones se obtienen mediante las técnicas *typosquatting* de dnstwist.

Para obtener las variaciones *typosquatting* de los dominios originales, se importó la clase `DomainFuzz` de dnstwist, que además añadía un campo al dominio en el que indicaba el tipo de técnica *typosquatting* que se había usado en su creación. Estas técnicas eran:

- *Bitsquatting*: resultado de un error de transmisión aleatorio.
- *Homoglyph*: cambiar caracteres de grafía similar.
- *Hyphenation*: separar por guiones.
- *Insertion*: añadir teclas cercanas.
- *Omission*: suprimir letras.
- *Repetition*: repetir letras.
- *Replacement*: sustituir una letra por otra.
- *Subdomains*: añadir puntos.
- *Transposition*: intercambiar dos letras.
- *Vowel swap*: intercambiar vocales.
- *Addition*: añadir letras al final.

En esta fase se pasó de 7 oTLDs + 37 ccTLDs (44 TLDs) a **41 TLDs**, ya que se descartaron los TLDs originales ".mil", ".int" y ".edu" porque eran de tipo restringido (el registro de un dominio con estos TLDs está muy controlado) y además no aparecían en los dominios oficiales (no así ".gov", que se mantuvo).

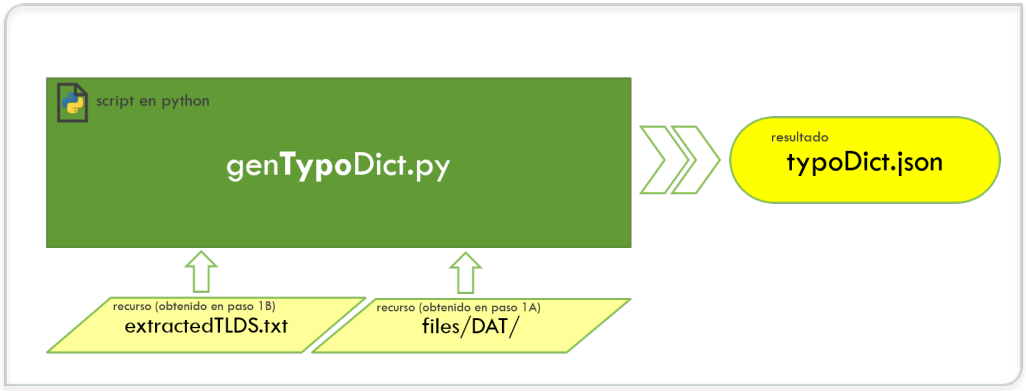


Figura 3.9: `genTypoDict.py` (fase 2)

```
1 - 11P          10 doms   (410 combs)   (132170 vars)
                          [...]
50 - TEF_CORP    8 doms    (328 combs)   (129488 vars)
```

```
TOTAL domains:                              2733
TOTAL combinations:                         75522
TOTAL variations (with duplicates):         33433684
```

Cuadro 3.3: Salida por pantalla de `genTypoDict.py` con 41 TLDs

3.3.3. Conexión entre Paso 2 y Paso 3

El script anterior devolvía un fichero con 36852288 dominios, que ocupaban 3,2 GB. Cargar este fichero provocaba **errores de memoria**, por lo que fue necesario buscar una solución para transferir la información del paso 2 (generación del diccionario) al paso 3 (recogida de información).

3.3.3.1. Solución 1

--piping: Cada variación *typosquatting* se pasa a `retrieveData.py` según se genera (a través de un pipeline de UNIX).

Se ejecutaba con la siguiente sintaxis:

```
$ python genTypoDict.py [args] --piping | python retrieveData.py [opt args]
```

En este caso también se podría introducir un solo dominio individual de la siguiente manera:

```
$ echo "{'fuzzer': 'Original*', 'domain-name': 'movistar.com'}"| python retrieveData.py
```

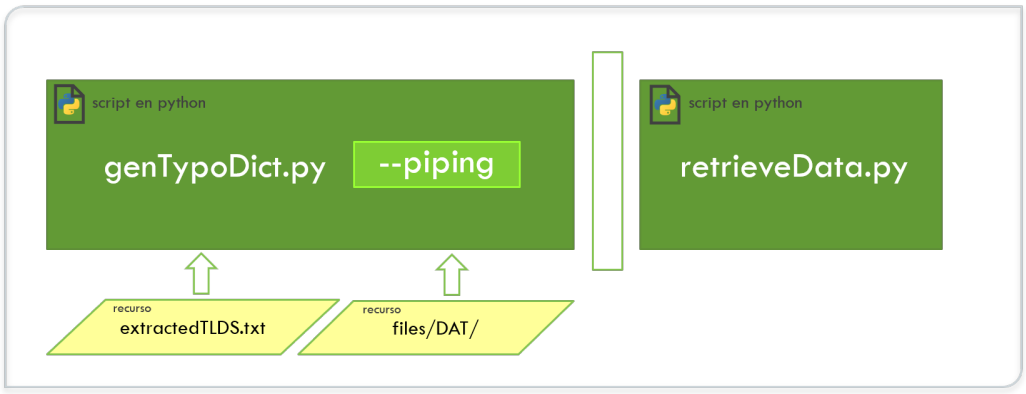


Figura 3.10: `genTypoDict.py --piping` (fase 2)

3.3.3.2. Solución 2

Dividir el diccionario typoDict.json completo (3,2GB) en 50 diccionarios (por clientes).

Para ello se escribió un nuevo script `splitTypoDict.sh` que leía el fichero compact-typoDict.json (en el que cada línea correspondía a un dominio junto con sus variaciones) y separaba los dominios por los 50 clientes, volcándolos en sendos ficheros.

Después, simplemente se ejecutaba `retrieveData.py` en 50 procesos, mediante el siguiente código:

```
1 dicts=typoDicts/*
2 mkdir output-typoDicts; mkdir time-typoDicts
3 for dict in $dicts ;
4 do echo "processing $dict..";
5 /usr/bin/time -o time-$dict.txt \
6     python3 retrieveData.py -d $dict -o output-$dict &
7 done
```

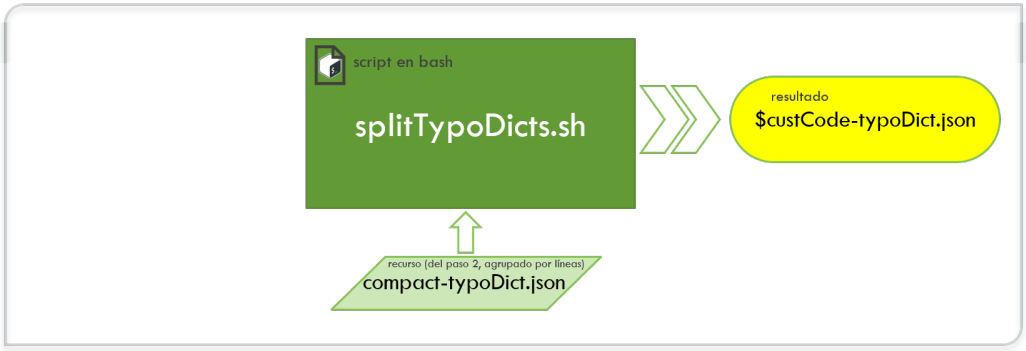


Figura 3.11: `splitTypoDict.sh` (fase 2)

3.3.3.3. Solución 3

--elastic: El resultado se almacena directamente en ElasticSearch, evitando problemas de tamaño de ficheros.

Para utilizar esta opción, `retrieveData.py` también tuvo que adaptarse a cargar los diccionarios desde ElasticSearch.

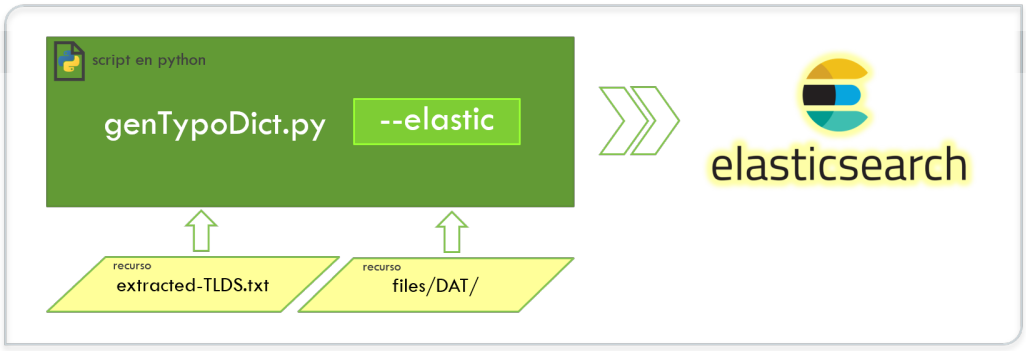


Figura 3.12: `genTypoDict.py --elastic` (fase 2)

3.3.4. Paso 3

Obtener información de cada dominio: registros DNS.
En consecuencia, asignar campos:
prioridad, frecuencia de comprobación y estado.

Al nivel de volumen de datos que se manejaba en esta fase, se hacía inoperable hacer las mismas peticiones que en la primera fase sobre todos los nuevos dominios (la constatación de este hecho se amplía en la siguiente sección de Resultados), así que se decidió recoger solo la información que da una petición DNS, mediante la función `get_dns`.

Debido a este cambio, fue necesario cambiar el criterio que asignaba niveles de prioridad. Se replicó la clasificación descrita en el cuadro 3.2, pero con peticiones DNS en vez de Whois.

3.3.4.1. Incorporación del Paso 4 anterior a los 3 y 4 actuales

La inserción de los datos finales en ElasticSearch durante la fase anterior suponía un paso extra. Era necesario ejecutar `insertES.py` una vez terminada la actuación de `retrieveData.py`.

Para esta segunda fase, se mantuvo dicho script, pero se integraron las funciones que contiene mediante su importación en los scripts `genTypoDict.py` y `retrieveData.py` (también en `updateData.py`). De esta manera, se simplificaba la interacción entre la base de datos y los scripts (de hecho, una vez generado el diccionario, este se almacenaba en ElasticSearch y así ya no eran necesarios los ficheros de texto), aprovechando los beneficios de ElasticSearch para tratar grandes cantidades de información.

3.3.5. Paso 4

Actualizar información.

Este paso se automatizó añadiendo las siguientes líneas con `$ crontab -e`:

```
30 2,10,18 * * * date >> ~/log-multiUpDat.txt
30 2,10,18 * * * bash ~/multiUpDat.sh results >> ~/log-multiUpDat.txt
```

El script `multiUpDat.sh` al que se llamaba desde crontab simplemente consistía en las siguientes líneas:

```
1  echo "$(date) - running script.."
2  arrFiles=(~/files/DAT/*)
3  for filename in ${arrFiles[@]} do
4          name=$(basename $filename)
5          cust=${name%_-_*} #cust=filename.split('_-_')[0]
6          /usr/bin/time -o ~/times/time-upDat-$cust.txt \
7                  python3 ~/updateData.py $cust '*' $1 -v \
8                  >> ~/logs/logs-upDat/log-$cust.log #&
9          echo "$(date) - $cust's data updated" done python3 ~/sendNotifs.py
10 echo "$(date) - script done."
```

Como se puede observar, este código en Bash llamaba a `updateData.py` cliente por cliente. Aunque se probó a enviar el proceso a segundo plano, finalmente se decidió prescindir de ello (comentando el & que aparece en la 8ª línea) para evitar excepciones por sobrecargar la máquina, ya que, a la hora de actualizar los datos, el ahorro de tiempo de procesado no era demasiado necesario.

Resumiendo la labor del script `updateData.py`, este se encargaba de comparar la fecha actual con la que cada dominio tenía fijada como "fecha de revisión". Si había llegado el momento de consultar de nuevo los registros DNS de un dominio, se volvía a usar la función `get_dns` para actualizar la información que el servidor DNS daba sobre el dominio en cuestión.

Se decidió agrupar la notificación de novedades en la base de datos en un solo email, enviando el mismo correo a una dirección de correo electrónico de prueba y a la dirección de Telefónica habilitada para esta tarea.

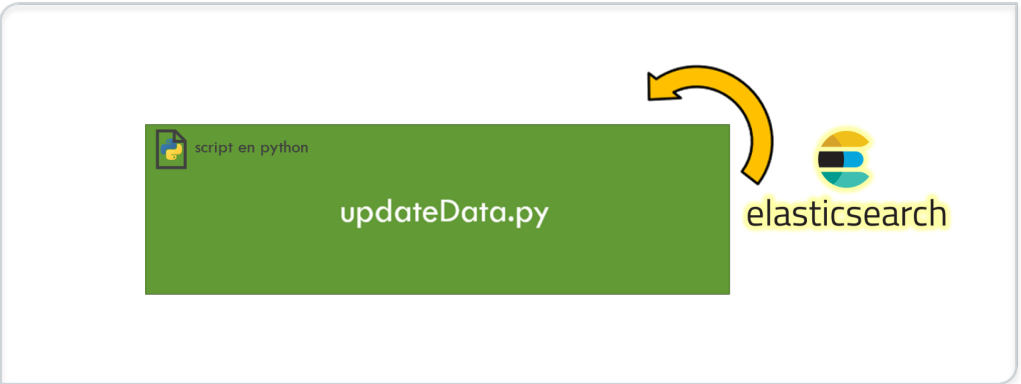


Figura 3.13: `updateData.py` (fase 2)

3.4. Fase extra

Sobre la **monitorización de las marcas** de los clientes **en redes sociales**, existía previamente un script `trendingtopic.py` que llevaba a cabo esta inspección de los Trending Topics (los términos más populares) de Twitter.

`trendingtopic.py` cumplía su función en tres pasos:

- Buscaba palabras clave en las webs y cuentas de Twitter oficiales del cliente (para ello hay herramientas de código abierto).
- Filtraba de allí las palabras demasiado comunes para quedarse solo con los términos relevantes.
- Finalmente comprobaba si se encontraban estos términos en los primeros diez Trending Topics de localizaciones geográficamente relevantes para el cliente.

Si algún Trending Topic coincidía con alguna palabra clave de nuestros clientes, ese Trending Topic se añadía al diccionario de dominios para ser monitorizado con los TLDs que se trabajaba.

Este script recurría a servicios externos como trends24.in. **Para evitar dicha dependencia**, sugerí la incorporación de un script que había desarrollado personalmente y que estaba bajo una licencia *copyleft* como es GPL (libre y gratuita).

El script en cuestión (llamado trenca.py) hacía uso de la API de Twitter para recoger los Trending Topics de varias localizaciones y permitía almacenarlos en un fichero JSON. Se comprobó la correcta integración de este programa con el resto del sistema de monitorización.

Listado de código 3.1: Código de trenca.py (disponible en Github)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# just a script to collect twitter's TTs
#
# usage:
# python3 trenca.py FILE [-c] [-n] N [-l] N [-f] [json/sqlite] [-v/s]
#
# example:
# python3 trenca.py output.json -n 10 -l 1 --stdout
# | awk '{print ""$1".com"}'
# | xargs -L 1 python2 dnstwist.py >> typoDomainsTTs.txt

__author__ = "@jartigag"
__version__ = '0.6'

import tweepy
import argparse
import json
from collections import OrderedDict
import sqlite3
from collections import OrderedDict
from datetime import datetime
from time import time,sleep
import signal
import sys
import os
from secrets import secrets

SLEEP_INTERVAL = 15*60 # secs between reqs
woeids = json.load(open('woeids.json'),object_pairs_hook=OrderedDict)
results = {}
n=0 # number of api reqs
FORMAT = "json"
FILE = ""

def main(verbose,stdout,format,file,nTop, nLocs):
    global secrets,results,n
    try:
```

```
39
40          init_time = time()
41          auth = tweepy.OAuthHandler(secrets[0]['consumer_key'],
42              secrets[0]['consumer_secret'])
43          auth.set_access_token(secrets[0]['access_token'],
44              secrets[0]['access_token_secret'])
45          dt = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
46          api = tweepy.API(auth, compression=True)
47          for place in list(woeids)[:nLocs]:
48              tt = api.trends_place(woeids[place])
49              n+=1
50              if dt in results:
51              #if this 'place' isn't the first consulted in this 'dt'
52                  results[dt].append({'as_of':tt[0]['as_of'],
53                  'locations':tt[0]['locations'],
54                  'trends':tt[0]['trends'][:nTop]})
55                  #append new results list.
56              else:
57                  results[dt] = [{'as_of':tt[0]['as_of'],
58                  'locations':tt[0]['locations'],
59                  'trends':tt[0]['trends'][:nTop]}]
60                  #create key 'dt' and add new results list.
61              if verbose:
62                  print('[*]',dt,place.upper())
63                  for t in tt[0]['trends'][:nTop]:
64                      print(' ','%02d'%(tt[0]['trends'].index(t)+1),'-',
65                          t['name'],'(%s tweets)'%(t['tweet_volume']) \
66                          if t['tweet_volume'] is not None else '')
67              elif stdout:
68                  for t in tt[0]['trends'][:nTop]:
69                      print(str.strip(t['name'],"#")) #remove possible "#"
70          if format=='json':
71              write_json(file)
72          elif format=='sqlite':
73              write_sqlite(file)
74
75      except tweepy.error.RateLimitError as e:
76          current_time = time()
77          running_time = int(current_time - init_time)
78          print("[\033[91m#\033[0m] api limit reached! \
79              \033[1m%i\033[0m api reqs were made (running time: %i secs)."
80              % (n,running_time))
81      except tweepy.error.TweepError as e:
82          print("[\033[91m!\033[0m] twitter error: %s" % e)
83      except Exception as e:
84          print("[\033[91m!\033[0m] error: %s" % e)
85
86  def write_json(outFile):
87      """
88      write results as a json (an array of jsons, actually) to dbFile
89
90      :param outFile: .json output file
91      """
92      with open(outFile,'w') as f:
93          print("{",file=f)
94          for dt in results:
95              print('"%s":'%(dt),file=f)
96              print("[",file=f)
97              for loc in results[dt]:
98                  if results[dt].index(loc)==len(results[dt])-1:#last element:
99                      print(json.dumps(loc,indent=2,sort_keys=True),file=f)
100                 else:
```

```
                    print(json.dumps(loc,indent=2,sort_keys=True),
                        end=",",file=f)
            print("],",file=f)
        print("}",file=f)

def write_sqlite(outFile):
    """
    write results as a sqlite database to dbFile

    :param outFile: .db sqlite database file
    """
    ## REMOVED, BUT AVAILABLE IN GITHUB)
    pass

def sigint_handler(signal, frame):
    print("\nexiting.. ",end="")
    if FORMAT=="json":
        with open("outputs/out11sep.json",'rb+') as f:
            #remove last "," in json ("}],}"):
            f.seek(-4,os.SEEK_END)
            f.truncate()
            f.write(str.encode("\n}\n"))
    print("bye!")
    sys.exit(0)

if __name__ == '__main__':

    signal.signal(signal.SIGINT, sigint_handler)

    parser = argparse.ArgumentParser(
        description="just a script to collect twitter's TTs, \
        v%s by @jartigag" % __version__,
        usage="%(prog)s FILE [-c] [-n] N [-l] N [-f] [json/sqlite] [-v/s]")
    onlyOneGroup = parser.add_mutually_exclusive_group()
    parser.add_argument('-c','--continuum',action='store_true',
        help='run continuously')
    parser.add_argument('-f','--format',choices=['json','sqlite'],
        default=FORMAT, help='format to store data')
    parser.add_argument('-l','--nFirstLocations',type=int,metavar='NUMBER',
        help='limit to NUMBER first locations (sorted as in woeids.json)')
    parser.add_argument('-n','--nTopTTs',type=int,metavar='NUMBER',
        help='limit to NUMBER top TTs on every location')
    parser.add_argument('file',
        help='output file to store data')
    onlyOneGroup.add_argument('-v','--verbose',action='store_true')
    onlyOneGroup.add_argument('-s','--stdout',action='store_true',
        help='print only TTs names')
    args = parser.parse_args()
    FORMAT = args.format
    FILE = args.file

    if args.continuum:
        while True:
            main(args.verbose,args.stdout,args.format,
                args.file,args.nTopTTs,args.nFirstLocations)
            sleep(SLEEP_INTERVAL)
    else:
        main(args.verbose,args.stdout,args.format,
            rgs.file,args.nTopTTs,args.nFirstLocations)
```

Capítulo 4

Resultados

En las siguientes páginas se recogen los resultados de las distintas pruebas hechas sobre cada parte del sistema, cómo estos fueron variando según se fue refinando su funcionamiento y de qué forma también influían en el rumbo que tomaba la evolución del proyecto.

Las modificaciones sobre los scripts se integraban constantemente en las pruebas, según se iban produciendo, de forma que la calidad de los resultados mejoraba continuamente.

Al igual que en la sección anterior, esta también se divide en dos fases. Se procura respetar el orden cronológico en el que se produjeron las acciones, para facilitar el entendimiento de cada avance de forma progresiva y consecuente con lo descubierto en los pasos previos.

4.1. Fase 1

En la primera semana de trabajo, utilizando los 7 TLDs originales con `genDict.py` en su primera versión se obtuvieron 2743 dominios oficiales (todavía faltaban algunos dominios inválidos por detectar) y 12782 combinaciones (cifra también inexacta, por errores en la supresión de combinaciones duplicadas). Aunque en este punto ya se había propuesto excluir los TLDs originales ".gov", ".mil", ".int" y ".edu" por ser de tipo

restringido, se acordó seguir incluyéndolos porque no suponían un gran esfuerzo y sobre todo porque lo que nos interesaba era comprobar hasta qué número de TLDs podíamos manejar en tiempos razonables.

Vista la *performance* de `retrieveData.py` con 7 TLDs (de media costaba unas 2 horas), en la segunda semana decidimos pasar a emplear todos los *country-code* TLDs encontrados (es decir, 37 TLDs). En la misma reunión semanal se me comunicó que un compañero, que había trabajado en este proyecto previamente y que corrigió algunos bugs, haría pruebas en paralelo en otra máquina, contando con mis scripts como punto de partida. Era por tanto necesaria una **documentación** completa que posibilitase desplegar una réplica del servidor de pruebas para su uso.

También detecté un problema con **Whois** en el servidor de pruebas que no se daba en local. Resultó estar causado por **diferencias entre las versiones** 5.2.17 y 5.1.1 (presente en el servidor) de la herramienta Whois del sistema (de la cual depende la librería Whois de Python). En consecuencia, en la documentación tuvo que incluirse este hecho. Además, se preparó un fichero setup.sh que recogía todas las instrucciones pertinentes para la configuración de este entorno, donde este problema se solventaba con las siguientes líneas:

```
## FIX WHOIS VERSION (5.2.17)
wget https://github.com/rfc1036/whois/archive/v5.2.17.tar.gz
tar -xzf v5.2.17.tar.gz
cd whois-5.2.17
sudo apt-get install gettext
sudo make install
whois --version
rm -r whois-5.2.17
rm v5.2.17.tar.gz
```

Tras revisar los ficheros de los clientes de forma manual, quedaron 2733 dominios oficiales válidos, con los que se crearon varios diccionarios compuestos cada uno del siguiente número de combinaciones:

Nº TLDs	Nº combinaciones
3	5526
7 (*original* TLDs)	12894
37 (*country-code* TLDs)	68154
44 (*o+cc*TLDs)	81048

Cuadro 4.1: Nº combinaciones obtenidas en función del número de TLDs

De tal forma que se pudo abordar progresivamente (probando primero con diccionarios más cortos) la recolección de información mediante `retrieveData.py`, correspondiente al siguiente paso.

Respecto a la recolección de información con `retrieveData.py`, pensamos que sería útil medir **qué dominios daban más problemas**, en concreto cuánto tardaba en responder a una petición Whois o hasta cuántas peticiones se permitían antes de ser bloqueados. Para ello, se analizaron los logs que producía `retrieveData.py`.

Las conclusiones de todas estas cuestiones se presentaron en la 4ª reunión, poco más de tres semanas después del inicio del proyecto. Se solicitaba responder las siguientes preguntas:

- ¿**Cuánto tiempo** cuesta recoger la información de **todos los dominios**?
  Tras varias pruebas, se vio que la ejecución completa de `retrieveData.py` suponía unas **20 horas**, pero era un tiempo bastante variable (mínimo 18 horas, máximo 22).

- ¿Durante **cuánto tiempo seguido** se han podido lanzar las pruebas? ¿Se producen bloqueos?
  Se pudo ejecutar el script de forma ininterrumpida durante varios días. Se esperó una semana más para confirmar que **no se producían bloqueos totales**.
  Sin embargo, analizando los logs de `retrieveData.py` más en detalle ("manualmente", con ayuda de scripts escritos en Bash específicamente para estos logs), se descubrió que **19 de las 75 peticiones Whois más lentas se hacían a dominios oficiales**. Por supuesto, estos

dominios no podían desestimarse, al ser dominios oficiales, pero sí se encontraron otros patrones más interesantes. Dejando de lado estos dominios oficiales, **las peticiones Whois que tardaban más en resolverse correspondían a TLDs ".pl" (de Polonia)** (60-70 segundos). Además, los dominios con TLDs **".lu" (Luxemburgo) y ".ma" (Marruecos)** tardaban 40-50 segundos.
Se dedujo que **era probable que los servidores** que ofrecían el servicio **Whois para estos TLDs estuvieran introduciendo un retardo constante como mecanismo de defensa** en respuesta al uso agresivo que estábamos haciendo. Se intuyó en base a los tiempos de resolución de estos dominios, que pasaban a costar 5,00 segundos (más algunas centésimas) después de varias peticiones.

- ¿Qué posibles mejoras de eficiencia encontramos?
  Se observó que el 90 % de las peticiones Whois se resolvían en menos de 1 segundo cada una, y el 98 % en menos de 10 segundos. En cambio, el 2 % restante (unas 1250 peticiones) costaban como 40 segundos de media cada una, es decir, entre 13 y 14 horas de las 20 horas totales. En definitiva: **el 2 % de las peticiones (las más lentas) tardan más que el otro 98 %**.

En el Apéndice A, enviado al equipo de desarrollo, se incluyen unas gráficas que explican la eficacia de las soluciones adoptadas para reducir las 20 horas de ejecución de `retrieveData.py`. Durante una semana, se probaron dos estrategias:

- Dividir el diccionario original en **dos diccionarios**: un "diccionario rápido" y otro "lento". Una vez se tenía la primera ejecución, en función de los tiempos de resolución de cada dominio que quedaban reflejados en los logs, con `gen2Dicts.py` se clasificaban los dominios en dos categorías: los que se resolvían en menos de 10 segundos y los que excedían este máximo. Se descartó esta posibilidad porque había una importante variabilidad en el tiempo que costaba procesar el "diccionario rápido" (en ocasiones incluso más que el "diccionario lento").

- **Multihilo**: hacer peticiones con 30 procesos simultáneos. Resultó ser la mejor aproximación para mejorar la eficiencia de `retrieveData.py`, ya que se pasó de 20 a **7 horas para resolver todo el diccionario**. Esto fue posible porque haciendo tantas peticiones a la vez tampoco encontramos ningún bloqueo.

Con la idea de dejar el sistema desarrollado hasta entonces en **ejecución continua**, se desplegaron los scripts y sus dependencias en un nuevo servidor. En el servidor anterior se dejaría corriendo `updateData.py` periódicamente, controlado mediante crontab (aunque en realidad se alcanzó una versión funcional de este script más tarde).

En la misma semana, se me hicieron algunas sugerencias para mejorar el código, que en verdad ya se habían implementado desde la versión en la que se proponían. Se sugería:

- Almacenar los datos de los clientes en formato JSON.
- Mantener un formato de fechas coherente a lo largo de todo el sistema.
- Controlar el orden en el que se manejaban las listas, para no provocar falsas alertas si se recibían los mismos elementos pero en distinto orden.

También empezaron a incluirse funciones auxiliares para soportar un mayor número de resultados en Elasticsearch.

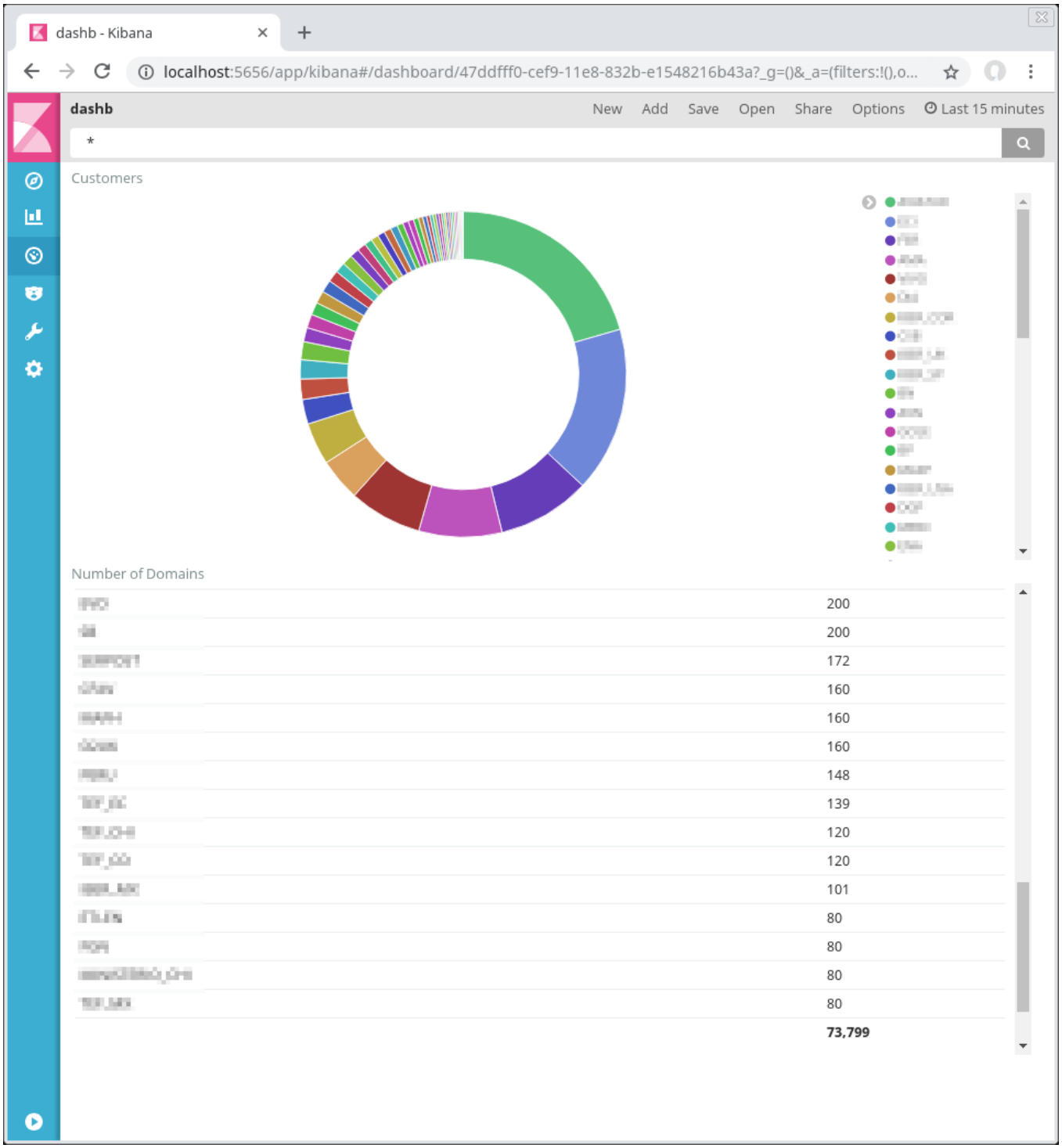


Figura 4.1: Resultados de la fase 1, visualizados en un panel de Kibana

4.2. Fase 2

En primer lugar, se hizo un nuevo análisis de la herramienta dnstwist, para discernir qué partes concretas del script nos podían ser útiles en nuestro sistema. Una vez quedó claro el proceso que sigue dnstwist en su funcionamiento, se pasó a incorporarlo a `genDict.py` (ahora renombrado como `genTypoDict.py`) y a `retrieveData.py`.

Pero antes, se lanzó una prueba con el nuevo diccionario generado a partir del anterior más las variaciones *typosquatting* de dnstwist, para tener una primera aproximación de cuánto costaría procesar el nuevo volumen de datos, y posiblemente distinguir qué técnicas *typosquatting* ofrecen más resultados. La prueba consistió en pasar por dnstwist cada una de las combinaciones dominio + TLD que teníamos antes (esto es, un total de **36 millones de variaciones**), aprovechando la opción multihilo de la herramienta:

```
cat merge-combinations-dom+TLD.txt | while read domain
do
   python2 dnstwist.py $domain -r -t 300 >> totalOutput.txt
done
```

Completar esta ejecución costó **76 horas**. Comparándolo con el rendimiento de `retrieveData.py` visto hasta ahora, que resolvía 81 mil dominios en 7 horas (con 30 procesos simultáneos), se estimó que la cantidad de dominios actual costaría 126 días, quizás 12 días si fueran 300 procesos simultáneos. En todo caso, era un tiempo excesivo a todas luces.

Se observó que esta diferencia de tiempos radicaba también en qué tipo de peticiones hacía dnstwist cuando se arrancaba en modo multihilo. Frente a este modo, la herramienta desactivaba las peticiones Whois y solo mandaba paquetes DNS, de los que obtenía información suficiente para conocer el estado de un dominio (obviando fechas de registro).

Esta prueba arrojó los siguiente resultados:

Técnica *typosquatting:*	Registrados (y % del total)	Registros **MX** (y % de registrados)	Registros **A** (y % de registrados)
Varias	4265 (0 %)	2121 (49 %)	3951 (92 %)
Original	6444 (1 %)	3049 (47 %)	4847 (75 %)
Repetición	11587 (2 %)	4343 (37 %)	11488 (99 %)
Guionización	11646 (2 %)	4325 (37 %)	11491 (98 %)
Transposición	18878 (3 %)	9151 (48 %)	18587 (98 %)
Intercambio de vocales	20104 (3 %)	10293 (51 %)	19680 (97 %)
Omisión	25255 (4 %)	13436 (53 %)	23753 (94 %)
Subdominio	36745 (6 %)	20777 (56 %)	36390 (99 %)
Adición	40478 (7 %)	26588 (65 %)	39204 (96 %)
Homóglifo	79188 (14 %)	31251 (39 %)	78716 (99 %)
Sustitución	83718 (14 %)	37822 (45 %)	81975 (97 %)
Bitsquatting	92593 (16 %)	42100 (45 %)	90671 (97 %)
Inserción	131058 (23 %)	51326 (39 %)	130536 (99 %)
TOTAL:	561959 (**0,707 % de las variaciones**)	256582 (**45 % de los registrados**)	551289 (**98 % de los registrados**)
Variaciones TOTALES:	**36285593**		

Cuadro 4.2: Resultados de la prueba con dnstwist

Conclusiones de esta prueba:

- Se generan **36 millones de variaciones** a partir de los 2733 dominios oficiales (con los TLDs, se comprueban las variaciones de un total de 81048 combinaciones), de las cuales **están registradas un 0.707 % (562 mil)**. De estos dominios registrados, el 98 % tienen asociada IP y el 45 %, servidor MX.

- Las **técnicas *typosquatting*** más relevantes en cuanto a resultados son: añadir caracteres cercanos en el teclado, bitsquatting, reemplazar caracteres y sustituir caracteres por homoglifos.
  El **coste de incluir cada técnica** en el proceso depende de la lista de dominios, pero aproximadamente, basándonos en qué operaciones básicas supone la implementación de cada técnica:

  - El método **bitsquatting** lo multiplica por un factor$\times$**63**.
  - **Homoglyph** incrementa en un$\times$**115** el número de variaciones.
  - **Replacement**, en un$\times$**250** (en promedio,$\times 27,6$ por cada letra de cada dominio, que puede tener unas 18 letras de media).
  - **Insertion**, en un$\times$**500** (funciona como replacement, pero en vez añadir una variación con el caracter sustituido, añade una con la tecla cercana delante y otra con esa misma letra detrás).

En vista de estos resultados, decidimos utilizar **registros DNS solamente**, que proporcionan información suficiente sobre el estado de todos los nombres de dominio, y dejar las peticiones Whois y HTTP para cuando el analista se interese por un dominio concreto.

En cuanto al número de variaciones, con una prueba usando solo 3 TLDs se vio que dnstwist devolvía un archivo de 450 MB y 4,4 millones de variaciones, así que iríamos **activando** las funciones que generan **cada técnica *typosquatting* de forma gradual** para poder escalar.

Como siguiente tarea, nos propusimos averiguar cuánto se tardaría en generar un diccionario de nombres de dominio que incluya los 36 millones de variaciones y cuánto costaría ejecutar `retrieveData.py` con este diccionario.

¿Cuánto tarda `genTypoDict.py`?
La primera incógnita se despejó rápidamente, simplemente incluyendo la clase `DomainFuzz` de dnstwist en `genDict.py` y haciendo las modificaciones pertinentes para dar lugar al nuevo script `genTypoDict.py`. El nuevo tiempo de generación del diccionario estaba **entre 40 y 50 minutos**.

¿Cuánto tarda `retrieveData.py`?
Se hacía evidente que era necesario agilizar el procesado de dominios en `retrieveData.py`, así que se experimentó con las 3 soluciones mencionadas en la subsección 3.3.3.

- La primera solución (haciendo uso de la opción `--piping` implementada para ello) intentaba establecer un flujo entre ambos scripts, pero se reveló como demasiado lenta.
- La segunda (en la que previamente se dividía el diccionario en 50 partes para ejecutar un proceso paralelo de `retrieveData.py` por cada una) ofreció los mejores tiempos: **en 48 horas** se habían procesado **la mayoría** de los diccionarios, pero a costa de trabajar con ficheros de texto y por tanto obligando a insertar posteriormente en Elasticsearch los ficheros JSON obtenidos, además de que los diccionarios de los clientes más grandes quedaban sin resolverse porque todavía se producían errores de memoria con ellos.
- Así que se adoptó la tercera solución (la correspondiente a la opción `--elastic`), que integraba la función que ya se hacía durante la fase anterior en este paso (recoger la información de cada dominio) con insertar los resultados en la base de datos en cuanto se tenían.

La depuración y perfeccionamiento de esta última solución se prolongó hasta el final del proyecto, por varias razones. En primer lugar, el volumen de datos en esta fase complicaba el testeo, haciendo también que los fallos fueran más probables. Además, las propuestas de pequeños cambios sobre la marcha en este paso implicaba que los datos guardados hasta entonces quedaban de alguna manera "invalidados" por diferir respecto del nuevo formato. Pero sobre todo, la razón principal por la que `retrieveData.py` no alcanzó su última versión hasta la última semana fue que a su desarrollo se sumaron otros encargos más urgentes sobre la primera fase, esencialmente referidos a `updateData.py` y las notificaciones que se esperaban de él.

En torno a la semana 10ª de proyecto, se nos pidió **poner una prueba estable de la fase 1** en una máquina **que enviara notificaciones** por email. Se retomó la última versión de `updateData.py` en ese momento (la de la 5ª semana) y se actualizó para que encajara con el estado en el que se encontraban el resto de scripts. Al poner el sistema en marcha (con esta versión actualizada de la fase 1), el envío del email dejaba de funcionar por intentar demasiadas conexiones, así que se cambió el planteamiento y formato de estas notificaciones para que se enviase un solo correo cada vez, agrupando todas las novedades.

También se corrigió la definición que determinaba qué cambios debían ser notificados, ya que **las peticiones Whois eran propensas a dar errores** (por ejemplo, si cambiaba el formato de la respuesta, la librería que las parseaba lanzaría una excepción, o si se daban posibles problemas de conexión con depende qué servidores). Como este dato solía provocar frecuentes modificaciones en los campos "prioridad" y "estado" (hubiera o no un cambio real), se decidió que **solo se notificarían los cambios en los registros DNS**, que de verdad son un indicio de actividad en el dominio.

Desde este cambio de criterio, en ninguno de los 3 correos diarios que recibía la cuenta de monitorización se notificó cambio alguno en ningún dominio. Por otra parte, esto era algo esperable, ya que las modificaciones en registros DNS son poco habituales.

En cuanto a la recogida de datos mediante `retrieveData.py`, se extrajo "técnica *typosquatting*" como argumento del script, para poder incorporar resultados progresivamente, **según la técnica *typosquatting* que tuviese mayor interés**. Se valoraron las técnicas (anteriormente descritas) en función del rendimiento que pudieran dar y el coste de cada una. Se pensó en estudiar unos historiales de peticiones a un servidor DNS de los que disponía la empresa, para empezar por las técnicas más relevantes, pero finalmente se desestimó la propuesta porque no se podrían aportar conclusiones a tiempo.

Intentando limitar el conjunto de posibles variaciones, propuse **reducir el número de TLDs** usados. Hasta ahora, si en un dominio oficial de un cliente aparecía, pongamos como ejemplo, ".au", ese TLD se usaría para generar combinaciones con todos los dominios de todos los clientes, con lo que el diccionario crecería linealmente en combinaciones (se sumarían 1842 combinaciones más) y exponencialmente en variaciones (entendiendo estas como las derivadas de aplicar técnicas *typosquatting*). Probablemente considerar un TLD que solo se ha encontrado en un dominio de un cliente no será relevante para el resto de clientes. Es por eso que analicé qué TLDs podrían retirarse.

Para empezar, los TLDs ".gov", **".edu", ".mil" y ".int"** podían considerarse fuera de peligro, ya que, como se ha comentado antes, su registro está muy controlado. Pero en los dominios oficiales de estos clientes sí que aparecían 2 ".gov" (".gov.ar" 11 veces y ".gov.co" 1 vez), así que se aprobó el descarte de los tres TLDs originales que no aparecían.

A continuación, seleccionando los 7 TLDs que aparecían menos de 10 veces, más otros 2 TLDs que aparecían menos de 30 veces, se reducían las combinaciones a 62628, lo que supone un 22,7 % menos. Sin embargo, al exponerse esta propuesta en reunión, se señaló que lo que se pretende es controlar todos los nombres de dominios posibles (cuantos más, mejor), así que descartar ciertos TLDs porque aparecen poco no era apropiado.

En conclusión, se admitió **acortar la lista a 41 TLDs**. Así, el número de combinaciones se reduciría a 75522, un 7 % menor que antes.

Una vez se tuvo el código de este proyecto en su **versión final** (la cual se cerró en la primera semana de octubre), se pasó a recoger de los datos del diccionario completo.

Aplicada la última medida explicada en los párrafos anteriores, el número total de entradas en la base de datos que almacenaba el diccionario quedó en **33.433.684**.

Se adaptó el script `multiRetrDat.sh` (que al principio simplemente lanzaba varios procesos simultáneos de `retrieveData.py` para acelerar el proceso) para que se recorriese el índice "diccionario" de Elasticsearch **seleccionando los dominios por cliente**. Este bucle se recogió en un script llamado `multiRetrDatCust.sh`.

Finalmente, el proceso **se segregó también por técnicas** (para ello, este último script en Bash llamaba a su vez al script `multiRetrDatTech.sh`), pudiendo elegir así qué técnicas *typosquatting* se querían analizar primero.

Si se recuerda el objetivo inicial (*analizar la viabilidad técnica de la solución*, en especial qué tipo de bloqueos se reciben por parte de servidores externos), se comprenderá por qué **no se priorizó la optimización** del sistema. Lo que esperaba el equipo era conocer hasta dónde podían llegar las ideas con las que se mejoraría el producto final en un futuro, y para materializarlas se proponía este proyecto en forma de **prueba de concepto**. Además, la integración de las mejoras que derivasen de este trabajo sería tarea del equipo de Desarrollo, que usaba otro lenguaje de programación para la implementación de cara al cliente. Por eso, desde el Departamento de Innovación de Elevenpaths se recomendaba no perder de vista este propósito principal, anteponiendo la consecución de un prototipo completo y funcional al perfeccionamiento de su rendimiento.

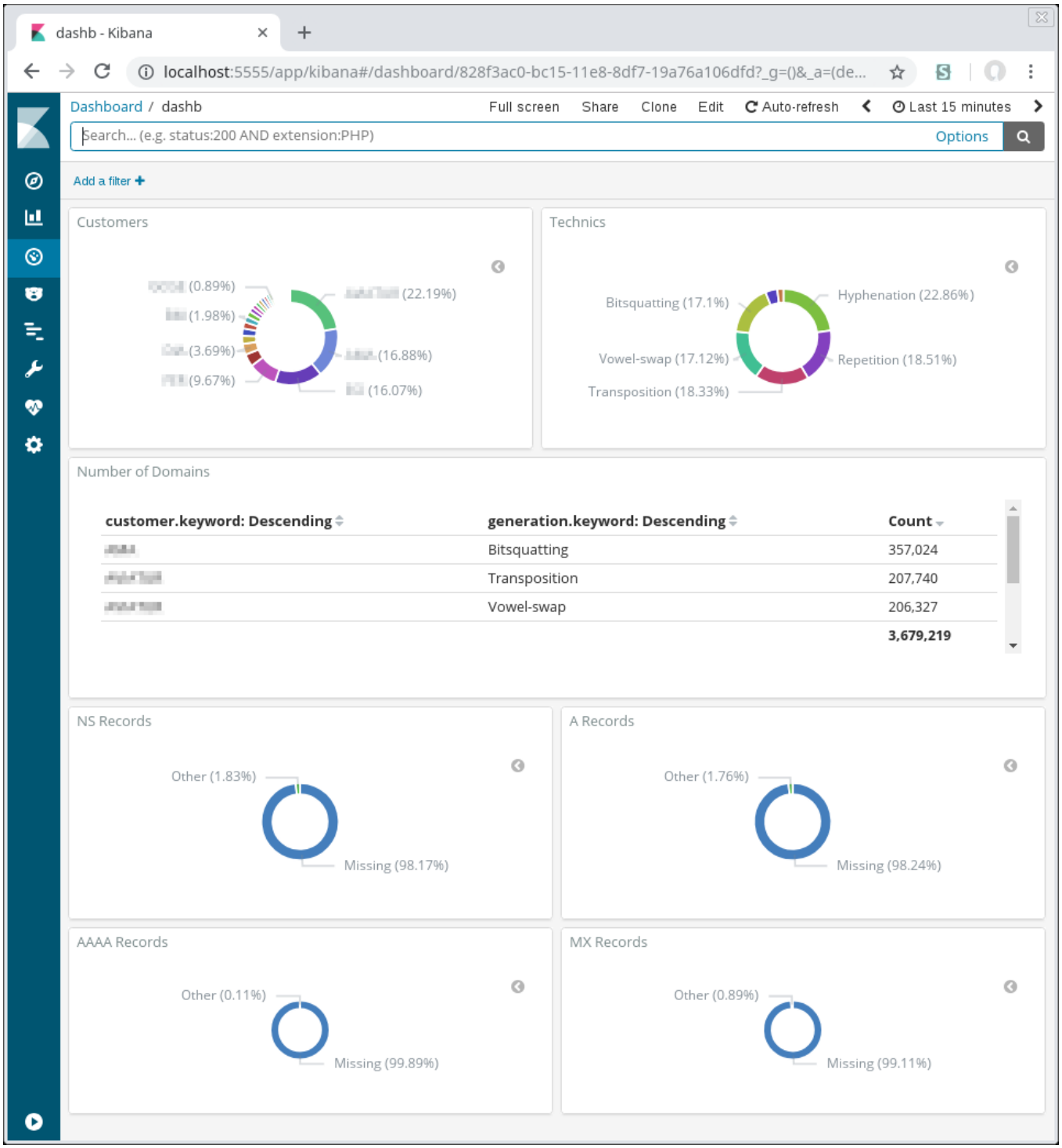


Figura 4.2: Resultados de la fase 2, visualizados en un panel de Kibana

Capítulo 5

Conclusiones

5.1. Conclusiones

Para acabar, en la última sección se exponen, de forma concisa, las observaciones más relevantes derivadas de este proyecto.

Escalabilidad Se ha llevado a cabo una reelaboración completa de la prueba de concepto para un único cliente de la cual disponía la empresa, consiguiendo escalarla para **múltiples clientes** y haciendo posible ampliar su cantidad sin dificultades añadidas.

En concreto, se soportaron 50 clientes que pedían proteger de todas las amenazas *typosquatting* posibles a 55 dominios cada uno de media.

Esto, por parte del sistema, supuso en última instancia **monitorizar 33,4 millones de dominios**.

Rendimiento En términos de métricas de rendimiento, se ha pasado de los 1,1 segundos/dominio (con 55 mil dominios) que se alcanzaban en la solución descrita en el informe del punto 1.6.1 a 0,3 segundos/dominio en la fase 1 (que también incluía peticiones Whois, con 81 mil dominios). En la segunda fase, haciendo solo peticiones DNS pero sin paralelizar más de dos procesos

simultáneos, **se bajó a los 0,24 segundos/dominio** (es decir, en torno a los 4 dominios por segundo).

Como se puede ver (y se profundizará en la sección 5.2), aunque se ha mejorado la rapidez del procesado, todavía queda margen de mejora.

Técnicas más frecuentes Según lo reflejado en las pruebas, se puede deducir que las variaciones *typosquatting* que corresponden con un porcentaje mayor del total de dominios registrados son: **inserción** de caracteres cercanos en el teclado (23 %), ***bitsquatting*** (16 %), **sustitución** de una letra por otra (14 %) y reemplazamiento de símbolos por otros de grafía similar u **homóglifos** (14 %).

Debe aclararse que aunque el *bitsquatting* (que confía en un error de transmisión fortuito para que por el cambio de un bit se establezca una conexión con la URL equivocada) aparezca en segundo lugar, en la práctica la frecuencia con la que este engaño tiene éxito es ínfima, por lo que tan solo valdría la pena preocuparse por esto en casos muy sensibles (como gobiernos, por ejemplo).

Número de peticiones Ya que dentro del objetivo principal del trabajo se especificó validar cuántas peticiones podían hacerse por máquina y sin ser bloqueados, de los *logs* se extraen las siguientes cifras:

- Número de peticiones sin bloqueo
  - Whois (fase 1): 11600 peticiones/hora
  - DNS (fase 2): 15000 peticiones/hora
- Número de peticiones por máquina:

  Como las pruebas se hacían desde una máquina y no se encontraron bloqueos, el número de peticiones coincide con el anterior. Sin embargo, en ambos casos todavía puede seguir creciendo el número de procesos paralelos, presumiblemente sin bloqueos.

Viabilidad En vista de los resultados del proyecto explicados en este documento, se considera que la presente solución es **completamente viable**. Incluso pueden alcanzarse mejores resultados con medidas como las que se sugieren a continuación.

5.2. Líneas futuras

De tener interés en seguir mejorando la calidad de este trabajo y sus resultados, podría continuarse su desarrollo incluyendo las siguientes propuestas:

- Para reducir el tiempo de procesado de los datos, sería beneficioso balancear su carga implementando un sistema de **Round Robin** con los servidores DNS usados en la fase 2 (también podría adaptarse a las peticiones Whois de la fase 1), lo que además aumentaría su tolerancia a fallos.

- Podría enriquecerse el análisis de la información obtenida **tratando las fechas** que se recogían en la fase 1, de manera que puedan aprovecharse las funcionalidades propias de Elasticsearch destinadas a convertir fechas en valores numéricos (*milliseconds-since-the-epoch*) para compararlas y ordenarlas cronológicamente.

- Tal y como se comprobó al final de la fase 1, se puede optimizar el tiempo total mediante una **pila común** de la que consuman los diferentes procesos (la arquitectura de Elasticsearch ofrece esta posibilidad). Además, ya se demostró que mantener 30 procesos paralelos no era problemático, así que podría estudiarse hasta dónde se puede elevar el **límite de procesos paralelos**.

- A la hora de elaborar el producto final, sería conveniente **optimizar el rendimiento del código fuente**, incluso reescribirlo bajo otros paradigmas de programación más adecuados u otros lenguajes que permitan optimizar a más bajo nivel.

- Este proyecto permite su **integración** como un bloque **dentro de la solución anterior** que se explicaba en el punto 1.6.1, la cual aportaba un diccionario de palabras relacionadas y tenía funcionalidades como la detección de patrones. Ambas partes funcionarían de manera coordinada y el sistema global se vería beneficiado.

- Podrían incorporarse otras muchas ideas nuevas, como por ejemplo:
  - Incluir el criterio del analista en el proceso de priorización.
  - Tener en cuenta el contenido de las páginas web que se sirven en los dominios maliciosos.
  - Extraer estadísticas de los cambios observados en estos dominios a lo largo del tiempo, que puedan usarse como realimentación para gestionar la comprobación de estos dominios fraudulentos.

Apéndice A

Informe para el equipo de desarrollo sobre la fase 1

Se adjunta en las siguientes páginas el informe redactado al término de la primera fase para el equipo de Desarrollo de Elevenpaths.

Este equipo se encargaría de reimplementar el sistema desde cero, adaptándolo a la infraestructura y tecnologías que la empresa usa para ofrecer este servicio en forma de producto final.

Report for Development Team: Typosquatting Project, Stage 1

Introduction

The development of this typosquatting detection tool (which is part of the Telefónica's CyberThreats service) is divided in 3 stages:

1. Verify the existence of a customer's domain with different TLDs
2. Include variations of the domain's name
3. Monitor customers' trademarks in social networks (trending-topics)

Most of the existing code (developed in spring/summer 2017 for this typosquatting project) has been rewritten and ported to Python 3. Progress made this month in stage 1 is detailed below, as well as possible issues and some conclusions.

General Workflow

1. **Remove invalid domains and duplicates** from the official domains `.xls` files. Dump results for each file in `files/DAT/CUSTOMER_CODE_-_Domains.dat` (plain text).

`>> python3 offDoms/check_dups.py` (and some manual intervention) was used for this step

2. Extract **country-code TLDs** that appear in the official domains.

`>> python3 offDoms/extract-tlds.py >> offDoms/extracted-TLDs.txt`,
along with `all-cc-TLDs.txt`, made this task

3. Generate a **dictionary (a JSON file)** from `extracted-TLDs.txt` and `.dat` files.

`>> python3 genDict.py offDoms/extracted-TLDs.txt files/DAT/ files/dictFile.json`

4. Retrieve data: **whois**, **dns and mx** servers, **ip**, **http** and **https** requests. Assign **priority** and **test_freq** (priority=high → test_freq=1day; priority=low → test_freq=14days), with extra info for the analyst in **status** field.

`>> bash continuum-retrieveData.sh files/dictFile.json`
was left running, in order to measure its performance continuously

5. When previous script finish an execution (so `log-files/$dictFile/output$i.json` is ready), **insert data into ElasticSearch** database.

`>> python3 insertES.py log-files/$dictFile/output$i.json elasticSearchIndex`

6. **Logs** of `retrieveData.py` **can be superficially analyzed**.

`>> bash stats-retrieveData-logs.sh log-files/$dictFile/log$i.log` gives some quick stats

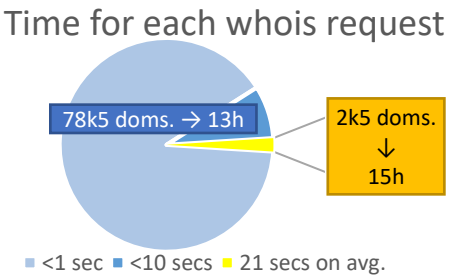
The real work is made with the bash script of the 4th step, which runs `retrieveData.py` endlessly and gives performance info about how long it took in total (`time$i.txt`) and for each domain (`log$i.log`). The other scripts only need to be executed once (scripts of the 5th and 6th steps may be executed whenever necessary).

Another script for update the database was written (`updateDataES.py`), but it's less refined. It will be improved on next stage.

Possible issues

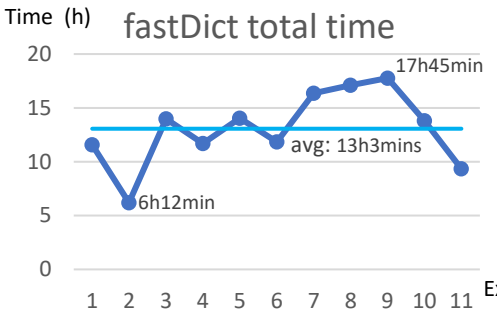
- **Domain delays**:

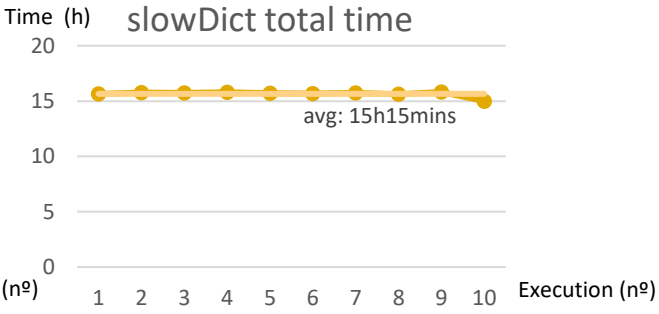
  90% of the whois requests take <1 second each one, and 98% take <10 secs. The remaining 2-3% (like 2500 domains) take on average 21 secs each one, that is, up to 15 hours.



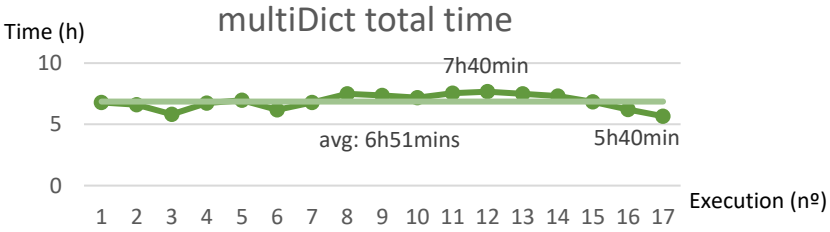
Solutions:

- **fast/slowDict**: `genDict.py` script was adapted to create 2 dictionaries from a `retrieveData.py` log: fastDict and slowDict. Resulting script is `gen2Dicts.py`.
  After testing it for a week, it was observed a significant variation in the time **fastDict** takes to process its 79652 domains (from 6h12min to 17h45min, 13h on average).
  On the other hand, **slowDict** kept rather constant around 15h15min.





- **Multithreading**: it seems to be the best solution. Testing an implementation of `retrieveData.py` with a 30 processes Pool (`multi-retrieveData.py`) gives complete results in 6h51min on average (from 5h40min to 7h40min).
  There wasn't any problem with blocking.



Some conclusions

	time/request
• Longest whois requests are **.pl** domains	60-70 secs
• Some TLDs (**.ma**, **.lu**) apparently follow a pattern, because many of our customers' domains are registered with these TLDs	40-50 secs

Development suggestions

Time can be optimized with a **shared stack** across different processes, as proved in `multi-retrieveData.py`. Further analysis might reveal even better optimization techniques, but less than 7 hours to retrieve the complete dataset is probably enough.

Referencias

[1] ElevenPaths. Reto Ciberseguridad y Big Data. [En línea]. Disponible en:. *https://www.elevenpaths.com/es/labsp/universidades*, 2018.

[2] U.S. District Court for Northern California. Sentence for Case No.: CV 11-03619-YGR (KAW). [En línea]. Disponible en:. *http://www.entlawdigest.com/2013/05/03/typosquat.pdf*, 2013.

[3] Paul Ducklin. Typosquatting – what happens when you mistype a website name?. [En línea]. Disponible en:. *https://nakedsecurity.sophos.com/typosquatting/*, 2018.

[4] Benjamin Edelman and Tyler Moore. Measuring the Perpetrators and Funders of Typosquatting. [En línea]. Disponible en:. *http://www.benedelman.org/typosquatting/typosquatting.pdf*, 2009.

[5] Chema Alonso. Cuidado con los ofertones de AirBnb que te ofrecen. [En línea]. Disponible en:. *http://www.elladodelmal.com/2015/09/cuidado-con-los-ofertones-de-airbnb-que.html*,
2015.

[6] Omar Ruiz Rodríguez. Cómo te estafan 1.282 euros por un falso piso en alquiler en #AirBnB o #Idealista. [En línea]. Disponible en:. *http://www.elladodelmal.com/2016/08/como-te-estafan-1282-por-un-falso-piso.html*,
2016.

[7] Wiktor Nykiel and Iván Portillo. CyberSquatting dot es. [En línea]. Disponible en:. *https://cybercamp.es/cybercamp2016/videos/cybersquatting-dot-es.html*, 2016. Tabla reproducida con el permiso de los autores.

[8] Aruna Prem Bianzino. Innovation activity to detect and prevent Typosquatting attacks. *ElevenPaths, Telefonica*, 2018.

[9] Doug McIlroy. Unix Time-Sharing System: Foreword. *The Bell System Technical Journal, Bell Laboratories*, pages 1902–1903, 1978.

[10] Dennis Ritchie. Hash-bang directive. *Unix Edition 7 and 8 (Research Unix)*, 1980.

[11] Marcin Ulikowski. dnstwist. [En línea]. Disponible en:. *https://github.com/elceef/dnstwist*, 2015.

[12] IETF. RFC 3986. [En línea]. Disponible en:. *https://tools.ietf.org/html/rfc3986*, 2005.

[13] ICANN. New Generic Top-Level Domains. [En línea]. Disponible en:. *https://newgtlds.icann.org/en/program-status/delegated-strings*, 2018.