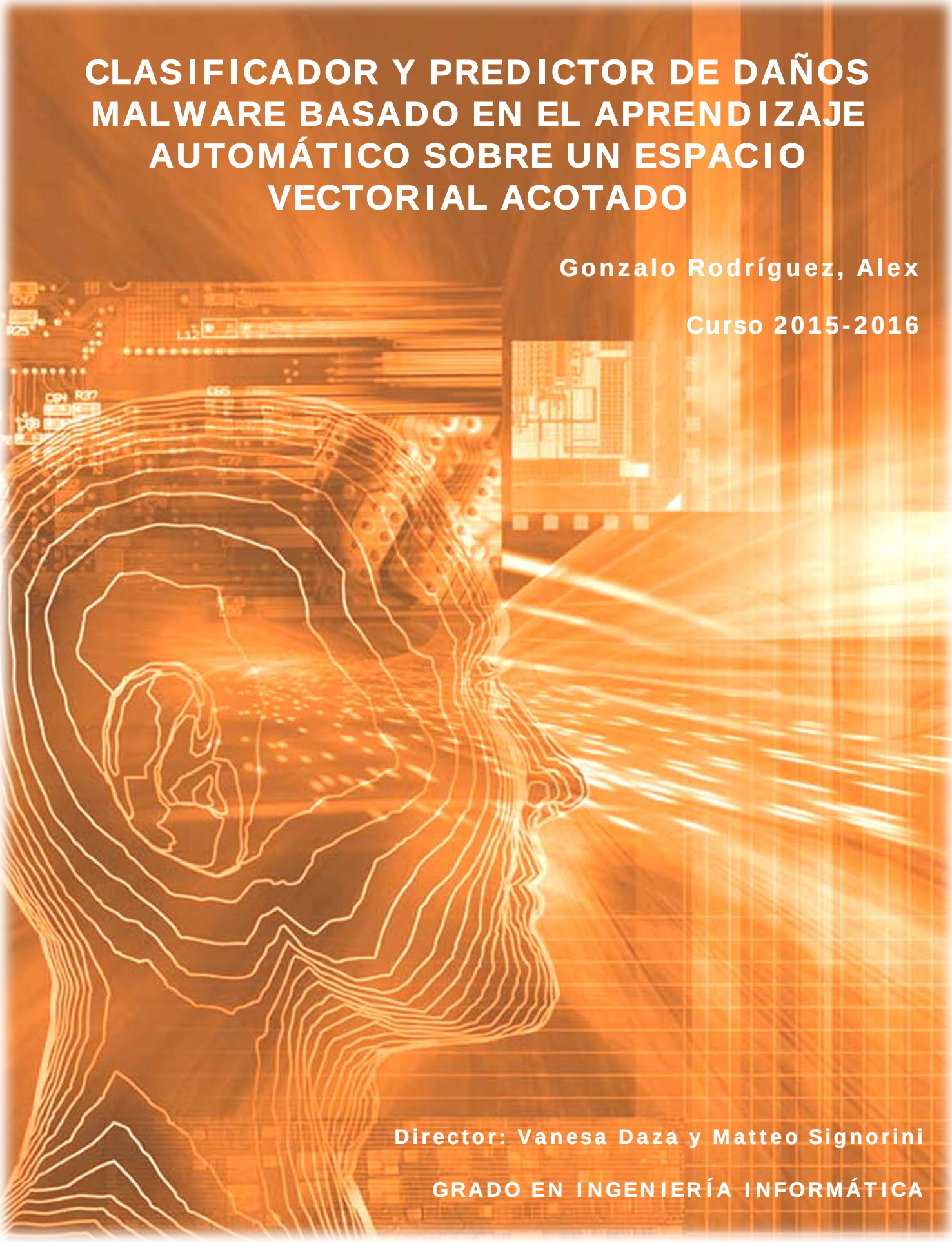
CLASIFICADOR Y PREDICTOR DE DAÑOS MALWARE BASADO EN EL APRENDIZAJE AUTOMÁTICO SOBRE UN ESPACIO VECTORIAL ACOTADO

Gonzalo Rodríguez, Alex

Curso 2015-2016



Director: Vanesa Daza y Matteo Signorini

GRADO EN INGENIERÍA INFORMÁTICA

upf. Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Trabajo de Fin de Grado

AGRADECIMIENTOS

En primer lugar, darles las gracias en especial a nuestra tutora Vanesa Daza, tanto por su gran involucración constante en el proyecto en cuanto a consejo y orientación, pero también en lo que a conocimiento y material se refiere.

En segundo lugar, darle también las gracias a Matteo Signorini, co-director del proyecto, que, pese a que su estancia no ha sido en Barcelona durante prácticamente todo el desarrollo, siempre ha estado disponible para cualquier duda, consulta o sugerencia cuando se le ha necesitado.

Por último, gracias a la Universidad Pompeu Fabra (UPF) por esta oportunidad, y a nuestros familiares por motivos obvios, estando siempre en todo momento para apoyarnos.

RESUMEN

Español

Clasificador y predictor de daños malware basado en el aprendizaje automático sobre un espacio vectorial acotado.

Con el creciente volumen de malware hoy en día y la evolución de éste, la amenaza que conllevan cada vez es mayor. El malware actual presenta ofuscación, es polimórfico, y junto con otros detalles, hacen que una mayoría de los Antivirus convencionales sean ineficaces. Es por ello que es necesario cada vez más, el uso de técnicas que permitan dar una solución dinámica y adaptable a este problema, como por ejemplo aquellas basadas en el auto aprendizaje.

Nosotros proponemos realizar un software basado en esta metodología, capaz de clasificar muestras desconocidas de malware y de atribuirles un nivel de peligrosidad que permita saber cuán perjudicial es la muestra en cuestión. Nuestra principal mejora radica en la extracción de comportamientos sobre XML's, y la asociación de éstos a estructuras de tamaño constante, generando así un espacio vectorial que garantiza la eficiencia en tiempo y espacio de nuestra propuesta.

English

Malware classifier and damage predictor based on machine learning over a bounded vectorial space.

Nowadays, with the growing volume of malware and its evolution, the challenge they lead is getting bigger as the time runs. The malware presents obfuscation and polymorphism, among other techniques, rendering most of conventional antiviruses ineffective.

That's why it is necessary the use of techniques that allow us to give a dynamic and adaptable solution to this challenge, such as the ones based on the machine-learning.

We propose a software based on this methodology, which is capable of classifying unknown malware and can assign them a threat level to know how of a harmful a sample is. Our main improvement is the behaviour extraction from XML files and its mapping to constant size structures, building a vectorial space which give us a better efficiency in time and space.

PRÓLOGO

Vivimos en una era en la que la informática, junto con Internet, forma parte de nuestro día a día, y en que cada vez más, es necesaria para cualquier tarea que se plantee: impulsar nuevos modelos de negocio al mercado o mejorar los ya existentes, hacer uso de las redes sociales, hacer uso de servicios web, buscar información, etc. Por lo que, en definitiva, ha habido un avance muy grande en el uso de la web en los últimos años por parte de los usuarios de todo el mundo.

Como resultado de esto, el volumen de datos que circulan por ésta ha ido creciendo exponencialmente dando lugar a nuevos sectores de la informática, como puede ser el fenómeno de la "Big data", el cual parte del análisis de grandes bloques de datos en beneficio de las empresas.

Por todo ello, no es de extrañar, que la cantidad de malware presente en la web se haya disparado y que éste, debido a la creciente variedad de contenidos en línea que hay, sea cada vez más complejo y difícil de eliminar.

El trabajo que hemos realizado a continuación, propone dar una solución flexible y eficiente a este problema. Gracias al uso de un algoritmo que permite capturar el comportamiento del malware en estructuras de tamaño constante, al aprendizaje automático, el análisis incremental, y la evaluación de peligrosidad, se consigue el objetivo marcado eficientemente.

ÍNDICE

LISTA DE FIGURAS

Pág.

LISTA DE TABLAS

LISTA DE ECUACIONES

Pág.

1 INTRODUCCIÓN

Problema

En la actualidad, debido a la creciente amenaza malware y nuevas técnicas evasivas, es cada vez más difícil detectar y clasificar las nuevas variantes que surgen debido a las propiedades que éstas presentan. El ejemplo más conocido y utilizado cómo técnica de evasión es la ofuscación.

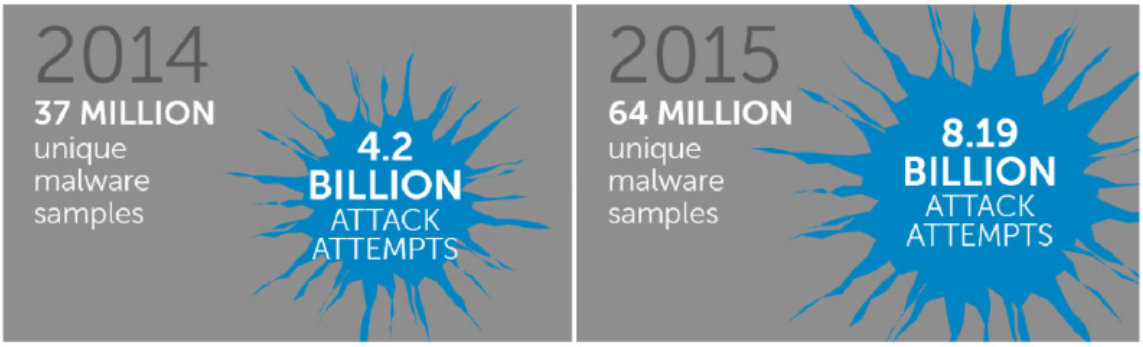


Fig. 1. Datos sobre la gran crecida de malware en 2014 y 2015 extraídos por Dell.

Esta técnica es implantada por el programador y su único objetivo es ocultar el flujo de control de un determinado software, así como también estructuras de información sensible además de tratar de mitigar el efecto de la ingeniería inversa e incluso dificultar la detección de este tipo de software por software especializado.

En el año 1997 Collberg et al. [15] definió una taxonomía para medir el efecto de una ofuscación definiendo un total de tres parámetros: potencia, resistencia y coste.

Por un lado, la potencia viene a expresar la dificultad que un ser humano tendría para entender un programa determinado ya ofuscado.

Por otro lado, la resistencia a diferencia de la potencia, define como de sólido y resistente es a la acción de un desofuscador automático un determinado código, que previamente también ha sido ofuscado.

Como último parámetro tenemos el coste, este parámetro hace referencia a la cantidad de esfuerzo y recursos empleados para llevar a cabo dicha ofuscación.

Por ello, junto con otros motivos, cada vez es más necesario emplear nuevas técnicas y algoritmos basados en aprendizaje automático, capaces de dar una solución dinámica y flexible, y a su vez, efectiva; capaz de resolver este problema creciente y polimórfico.

1.1 Motivación

La comunidad existente a día de hoy que trata de combatir tanto el malware, como aquellas personas que lo crean y fomentan es cada vez mayor, y parece que en los próximos lo será aún más. Ante este fenómeno, creemos que es necesario aportar todo el conocimiento posible para ayudar a los investigadores e intentar buscar una solución común para resolver este problema creciente.

Tras presentarse esta oportunidad y analizar el problema con determinación, dictaminamos que podría ser de ayuda el proyecto que aquí se presenta, el cual surge de la combinación de las aportaciones de dos artículos del mismo ámbito. Todo ello con el objetivo de dar una solución firme al problema planteado y poder aportar nuestro granito de arena a la comunidad de académicos, expertos, etc. contra el malware.

1.2 Objetivos

Los objetivos del proyecto se dividen en dos tipos: *principales* y *secundarios*.

Del primero no hay más que uno, que abarca el proyecto a nivel global y que radica en la realización de un programa basado en el paradigma del aprendizaje automático, capaz de clasificar malware desconocido a partir de sus propiedades (acciones), y de determinar su grado de peligrosidad en relación al sistema operativo que afecta.

Como secundarios, definimos un total de tres, que se citan a continuación:

1. **Eficiente en tiempo, con ejecución lo más cercana al tiempo real posible**: Desde un inicio nos marcamos como meta la idea de obtener un sistema eficiente y rápido. Creemos que hoy en día, este es uno de los factores que más se premia posibilitando su uso cuando se desee y no expresamente en un ámbito técnico sino también doméstico.

2. **Minimizar márgenes de error:**
   Este es un objetivo que puede deducirse por el carácter del proyecto, pero que no por ello se debe omitir y dejar de darle importancia. Los resultados obtenidos deben ser lo más positivos y certeros posibles.

3. **Garantizar la comprensión del código y reflejar en él nuestra idea de proyecto:**
   El último de ellos, y más dirigido a nosotros como autores, consiste en garantizar tanto la legibilidad como la comprensión del código. Igualmente, también se buscó la definición de una estructura lógica para el proyecto, así como la aplicación del paradigma “Clean Code” para un futuro mantenimiento del mismo.

1.3 Ventajas

Las ventajas, al igual que los objetivos, podemos dividirlas en dos tipos: *de funcionalidad* y *de propiedad*.

Por un lado, en cuanto a las de *funcionalidad*, tenemos tres puntos importantes a mencionar:

El primero de ellos es el hecho de que nuestra propuesta, además de aportar resultados de clasificación de malware, lo hace también de peligrosidad, refiriéndose a peligrosidad como cuan dañino sería un malware determinado si se ejecutase de forma libre en un sistema operativo.

El segundo es la combinación del algoritmo *Bounded Space Feature behaviour Modeling (BOFM)* [5], utilizado para el mapeo de muestras a un espacio vectorial, y el aprendizaje automático, en nuestro caso *Clustering*.

El tercero y último consiste en el uso de informes XML como formato de las muestras de análisis en vez del binario crudo del malware. Ello permite poder realizar un análisis mucho más intensivo del comportamiento de cada muestra individual que contrariamente no podría conseguirse. La conversión de binario a XML se realiza gracias a la herramienta web de análisis malware conocida como Anubis.

Por otro lado, tenemos las de *propiedad*:

La más destacable en este ámbito está directamente relacionada con uno de los objetivos, y ésta es su eficiencia en tiempo. Bien es cierto que las pruebas ejecutadas durante el proyecto no superan las 111 muestras analizadas, y que sería necesaria una evaluación de escalabilidad, pero basándonos en nuestras pruebas empíricas podemos garantizar que el programa es eficiente en tiempo.

Otra de las ventajas es de nuevo la eficiencia, pero esta vez en lo que al espacio respecta. Mientras que otros algoritmos escalan exponencialmente a la hora de capturar el comportamiento de los binarios malware en el espacio vectorial, la solución que proponemos logra hacerlo con vectores de tamaño fijo una vez el rango de acciones de malware está definido. Todo ello gracias al uso del BOFM, que nos ofrece un ahorro considerable de memoria en tiempo de ejecución.

1.4 Asignación y planificación del trabajo

Este proyecto ha sido realizado en conjunto con mi compañero de grado, Andrés Martín Autón. Desde el primer momento, ambos nos hemos organizado formando un equipo para garantizar la correcta elaboración del proyecto manteniendo mantener la filosofía de un equipo de ingenieros.

En lo que a la carga de trabajo respecta, ambos realizamos todo tipo de tareas, desde búsqueda de información en artículos elaborados por expertos pasando por testeo de muestras, desarrollo de código, etc. No obstante, yo personalmente me centré más en la parte de implementación referente a la parte de aprendizaje automático, abarcando extracción de prototipos, Clustering y clasificación. Además, sugerí y planteé la idea del mecanismo utilizado en la parte del proyecto referente a la predicción de daños malware.

Sin embargo, esta parte de daños fue implementada por Andrés, junto con la estructuración del sistema de informes, la clasificación previa de las muestras tanto de entrenamiento como de test, así como la impresión de información y extracción de datos en la fase de test. Todo ello con el objetivo de validar los resultados obtenidos del programa durante esta última fase.

Por último, tareas previas como la búsqueda de binarios o la extracción de informes a través de Anubis [7], fueron totalmente compartidas dando lugar a un resultado equilibrado donde ambos integrantes integramos todas las partes en las que el proyecto se divide.

1.5 Estructura del trabajo

Respecto a la estructura del proyecto, este consta de cuatro partes que engloban las distintas secciones del mismo.

La primera de ellas consiste en los antecedentes y la historia del arte, donde se presenta como está hoy en día el ámbito del análisis de malware.

Seguidamente, se da paso a la parte de explicación de la propuesta, la cual consta de un total de ocho secciones donde se detallan los diferentes algoritmos empleados, así como las decisiones personales tomadas en el desarrollo de la propuesta.

Tras ello, se realiza la evaluación de la propuesta y pasamos a presentar los resultados por parte de las dos vertientes del sistema, tanto de la clasificación como de la peligrosidad.

Finalmente, se argumentan posibles futuras mejoras a implementar a nuestro trabajo, las dificultades que han ido surgiendo a medida que avanzábamos en el proyecto, y, por último, las conclusiones, el glosario y la bibliografía.

2 ANTECEDENTES

2.1 Algoritmos basados en aprendizaje automático

El pasado del análisis de malware está poblado de nombres de Antivirus, algunos de los más sonados: Avast, Kaspersky, McAfee, Panda, etc. La base de actuación de una gran mayoría de éstos consiste principalmente en el análisis a nivel de fichero, es decir, en analizar el contenido/estructura del malware en cuestión, para determinar su tipología.

No obstante, hoy en día esto resulta insuficiente en un gran volumen de casos debido a que los hackers cuentan con herramientas y técnicas para eludir este tipo de análisis, dejando una gran parte de los antivirus obsoletos debido a este motivo. La técnica más utilizada por estos individuos es la de ofuscación del malware.

Por ello, expertos en las áreas de protección anti-malware, trabajan cada día en búsqueda de alternativas al procedimiento tradicional ya mencionado.

Una de las alternativas más recientes y que ha ido tomando relevancia con el paso de los años, consiste en el uso de aprendizaje automático ("machine learning"), que va más allá del tratamiento individual de muestras malware. El procedimiento en sí consta de una serie de procesos básicos que mencionamos a continuación:

1. En primera instancia se extrae información relativa a las muestras en cuestión. Si se hace mediante el uso de Antivirus, basta con extraer el output del mismo. Sin embargo, si tomamos la otra alternativa, que consiste en la ejecución de dichas muestras en un entorno virtual controlado, debemos realizar una preparación previa. Para ello lo que hacemos es, durante la ejecución de una muestra malware, interceptar las llamadas al sistema que ésta genera y anotarlas todas en un mismo registro, el cual, una vez finalizada la ejecución de la muestra, contendrá toda la información concerniente a ésta en forma de llamadas al sistema. De esta forma podemos encapsular el comportamiento de las muestras en registros individuales una por una.

2. Se mapean los comportamientos guardados en estos registros a un espacio N-dimensional, donde cada registro representa una dimensión del mismo. Este es uno de los pasos clave para el éxito en cuanto a eficiencia se refiere, más adelante haremos referencia a este apunte.

3. En este punto, ya entran en juego las técnicas de aprendizaje automático propiamente dichas. Algunas de las más utilizadas son las de *Clustering o PageRank.* La primera técnica consiste en la agrupación de elementos similares en estructuras llamadas clústeres donde todos los elementos que conforman un clúster comparten características o son muy similares entre ellos. Sin embargo, la segunda se fundamenta en la idea de interacción entre elementos, es decir, para un elemento no se tiene en cuenta únicamente su interacción con otro de forma singular, sino que también se tiene en cuenta como éste es afectado por el resto.

4. Este resultado del *Clustering* finalmente es comparado con unos datos iniciales existentes conocidos como conjunto de entrenamiento. A esta técnica se le conoce como aprendizaje automático *supervisado,* puesto que los datos de entrenamientos

han sido previamente analizados y categorizados por humanos, atribuyendo a estos datos pues, total certeza de su correctitud o validez. Gracias a esta técnica, es posible asignar una determinada categoría de entrenamiento a los clústeres generados anteriormente.

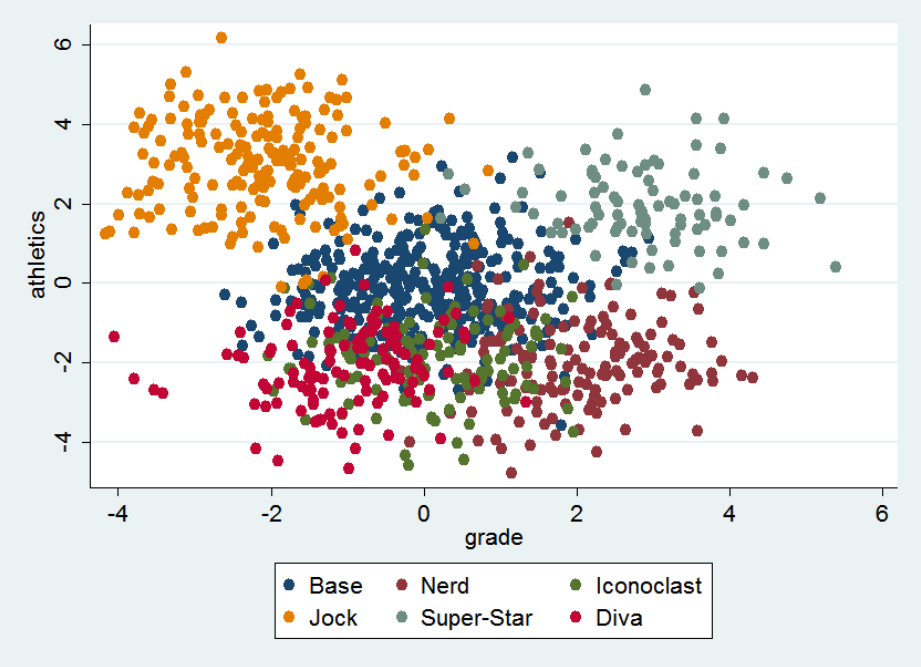


Fig. 2. Distribución de muestras en clases (cada color representa una clase distinta).

En la continuación de esta sección, presentamos diferentes trabajos realizados entorno a éste área, donde cada uno de ellos presenta una instancia diferente de cada uno de los pasos citados anteriormente.

El primero de ellos es el realizado por el equipo de Konrad Rieck [1], en cuyo artículo proponen una serie de soluciones a la realización de los procesos mencionados anteriormente.

Para el primer punto, éstos utilizan un servicio web online llamado Anubis [7], el cual puede encontrarse en el siguiente enlace [2], y cuya funcionalidad es la de convertir binarios de malware en informes XML. Los informes generados por Anubis son extremadamente detallados e incluyen información de todo tipo, tanto en lo que se refiere a las llamadas al sistema que ha llevado a cabo el malware, como al tipo de recurso del sistema operativo que éstas afectan: registros, sistema de ficheros, procesos, etc.

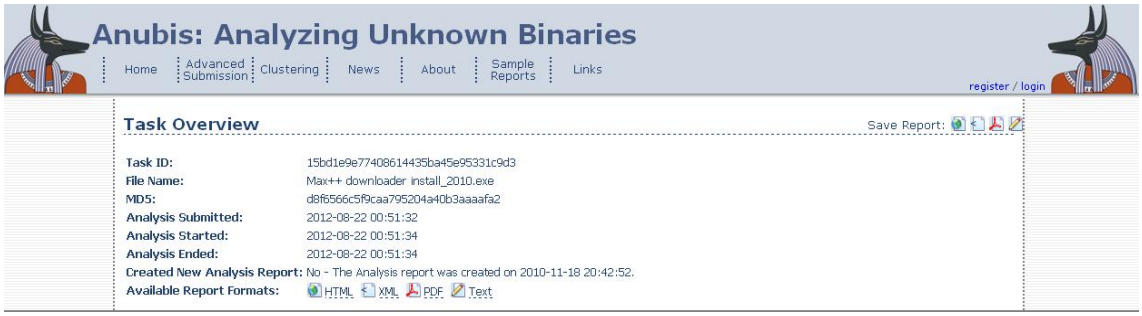


Fig. 3. Salida de informe de Anubis a una muestra malware.

En lo que respecta al segundo, al mapeo de muestras a un espacio vectorial, ellos definen un mecanismo llamado *q-grams,* que parte inicialmente de la creación de un tipo de formato llamado *Malware Instruction Set (MIST).* Este formato se elabora a partir de los informes extraídos mediante Anubis, donde cada instancia del mismo se corresponde con una acción del conjunto de acciones del malware. El formato que siguen estas instancias intenta mapear la misma información que cada entrada del informe generado por Anubis, pero de forma numérica, es decir, usando códigos numéricos para cada parte que compone la acción.

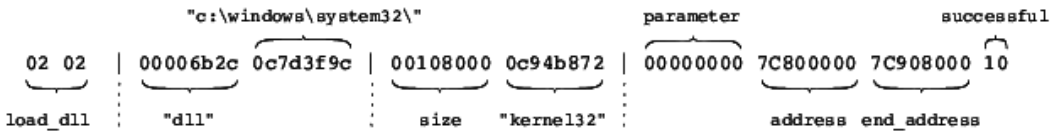


Fig. 4. Estructura de una instrucción MIST.

Además, éstas siguen una estructura similar a la que emplean las instrucciones de un procesador moderno. Una vez obtenido el formato, se procede de la siguiente manera:

1. Se obtienen todos los posibles q-grams, conjuntos formados por todas las posibles instrucciones que una muestra puede llevar a cabo, de longitud *q,* bajo formato MIST.

2. Una vez obtenido los conjuntos, se crea un vector para cada una de las muestras, en el cual se plasma el comportamiento de éstas. Para ello, basta con comprobar cuáles de los q-grams calculados previamente están contenidos en cada una de ellas. En caso positivo esa dimensión del vector toma valor uno, y en caso negativo, cero.

3. Una vez se tienen los q-grams calculados, se transfieren al espacio vectorial para poder ser manipulados luego mediante el Clustering.

Por último, comentamos el algoritmo de prototipado que emplean los autores del artículo, no sin antes mencionar que la búsqueda de prototipos está demostrada ser "NP-hard" [6], por lo que resulta inviable de implementar para un proyecto de este tipo en el que se busca una respuesta a tiempo real. Para solventar este problema, los autores proponen una adaptación del algoritmo de Garey y Johnson [6], capaz de dar una solución cuyo coste temporal es dos veces el tiempo de la solución óptima. El algoritmo consiste en encontrar determinadas muestras que representen un conjunto de éstas, actuando así, como puntos de referencia del resto de muestras del mismo conjunto, abarcando el mayor número de muestras posible. Una analogía de este algoritmo, podría ser el caso en el cual un país busca a un mandatario (prototipo) capaz de representar sus ideales contentando a la mayoría de la población (muestras). Con esto, lo que se quiere conseguir es una mayor eficiencia utilizando estos prototipos como base del Clustering para después heredar este resultado al resto de muestras a la hora de la clasificación. Obviamente, estos prototipos dan pie a los clústeres iniciales, por lo que juegan un papel crucial en la inicialización del Clustering.

Otro de los trabajos de la misma área es el [3]. Este difiere totalmente del anterior en cuanto a mecanismos, pero es totalmente firme con los procesos explicados, al igual que el anterior.

En este caso, para el primer punto, los autores toman la vía del uso de Antivirus. Para ello hacen uso de un recurso web llamado VirusTotal, que proporciona "feedback" de este tipo de software. Ellos lo obtienen de un total de 43 muestras.

Para afrontar el proceso de mapeo, hacen uso del *Function Call graph (FCG)*. Este mecanismo trabaja directamente con el desensamblado del binario, creando un grafo dirigido donde cada nodo del mismo representa una función del programa y cada una de las ramas una llamada a dicha función:

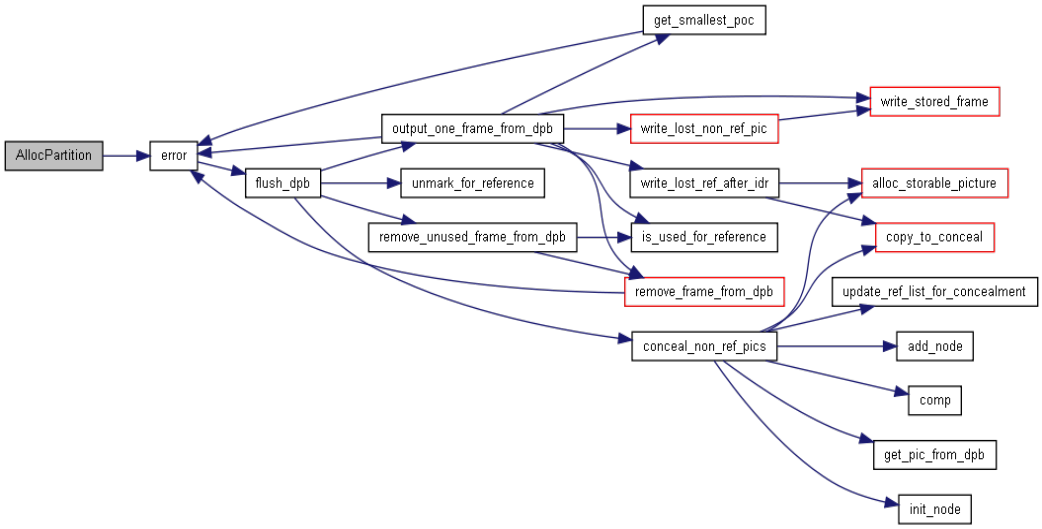


Fig. 5. Ejemplo de Function Call Graph (FCG).

Antes de iniciar este proceso, es necesaria la extracción de un seguido de información de las muestras. En el mismo artículo, proponen un total de seis atributos, que son los siguientes: *opcode* (frecuencia de aparición de cada código de operación), *API* (número de veces que se llama cada librería), *memory* (número operaciones de lectura/escritura en memoria en esa función), *IO* (el número de operaciones de entrada/salida en cada registro en esa función) y *Flag* (el número de cambios en cada flag).

Referente al segundo punto, más simple que en el caso del primer trabajo, basta con extraer un vector para cada uno de estos atributos. El conjunto de estos vectores modificará el comportamiento de cada muestra.

Para el tercer punto, se decantan por el Page-Rank. Estos aplican dicho algoritmo a una matriz a la cual llaman "pairwise matrix", la cual tiene dimensiones *nxm* donde *n* es el número de nodos del *FCG* de una muestra y *m* el número de nodos de otra. La idea del Page-Rank reside en que la distancia entre dos nodos no solo se refleja por la distancia entre ellos mismos sino también por la distancia entre la propia muestra y los nodos a los cuales llama.

Una vez obtenida esta matriz se procede con ella para calcular la distancia entre muestras. Para ello, inicialmente se considera que la distancia entre cada muestra es cero y se va actualizando haciendo uso de la matriz previamente calculada. Se selecciona en primera instancia la entrada de la matriz que refleja los dos nodos con menor distancia y se suma este valor a la distancia inicialmente nula. Una vez hecho esto, a dicha entrada se le asigna valor

infinito (para evitar repetir entrada). Este proceso se repite hasta que no quedan nodos con valor diferente a infinito.

Finalmente, se calcula la distancia final entre muestras. Para ello se crea una última matriz haciendo uso de un kernel Gaussiano.

Para dar solución a este problema, en [4] se propone el sistema *chatter*. Al igual que en [3], artículo anterior, consta de los mismos procesos del aprendizaje automático, pero ofrece diferentes mecanismos para afrontarlos.

Comienza dando solución al primer punto a través del análisis individual de muestras de malware en un entorno virtual protegido, donde se interceptan las llamadas de alto nivel a las que ellos reconocen como eventos, y las mapean como letras del alfabeto. Una vez tienen los eventos mapeados como letras, las concatenan formando palabras que representan trazas de ejecución de la muestra malware en cuestión, y de esta forma obtienen al final una serie de palabras que encapsulan el comportamiento del malware.

Entonces, aplicando técnicas de clasificación de documentos mediante *n-grams* (similar al q-grams del primer artículo) y aprovechando un corpus de malware ya etiquetado de forma analítica, generan las familias a las que pertenecen las muestras malware.

Por lo que, en definitiva, emplean un corpus de entrenamiento a modo de referencia (aprovechando segmentos de éste; buscando similitudes entre las cadenas de palabras obtenidas y las ya existentes en el corpus) para saber a qué familia de malware pertenece cada muestra, una evidencia del uso del aprendizaje automático aplicado en este contexto, aunque no sea el típico Clustering.

Finalmente, en [5], a diferencia de los anteriores, se centra únicamente en dar solución al segundo punto del aprendizaje automático, de una forma interesante y eficiente. El Bounded feature space behavior modeling (BOFM), como llaman los autores a su propuesta, consiste en un algoritmo que plantea la encapsulación del comportamiento del malware desde un enfoque centrado en cómo puede afectar el malware a los recursos que presenta el sistema operativo en cuestión. Por lo tanto, no realiza la extracción de los comportamientos basándose en las propiedades que presentan los propios binarios malware, sino que la hace en base al conjunto maximal de acciones posibles que éstos pueden llevar a cabo dentro de los recursos del sistema operativo.

Para ello, el algoritmo obtiene el sumatorio de la combinatoria que se calcula para cada conjunto de acciones posibles que puede llevar a cabo el malware en cuestión, sobre cada recurso disponible en el sistema operativo. Como resultado de esto, se obtiene un valor numérico que representa el tamaño del vector de comportamiento para todos los binarios malware, en ese sistema operativo (si el número de recursos en el SO cambia, el tamaño del vector también).

Finalmente, los valores para las posiciones (0 o 1) del vector de cada malware, dependerán de la combinación de los conjuntos de acciones que lleve a cabo el malware en los recursos

del sistema operativo, sobre el número máximo total de conjuntos de acciones posibles que éstos pueden realizar.

Por lo que, en definitiva, lo que se obtiene es un espacio vectorial caracterizado por tener un tamaño fijo y sin una dependencia directa al número de muestras; una alternativa al q-grams y n-grams, acotada por una constante en tamaño que se traduce en una mayor eficiencia en tiempo y espacio.

En la siguiente sección se explica nuestra propuesta en la se combina el algoritmo de BOFM como mapeo de muestras al espacio vectorial junto con el algoritmo de aprendizaje automático mencionado en [1]. Además, añade una nueva característica al algoritmo que consiste en la cuantificación del impacto de una muestra en un determinado sistema operativo, obteniendo así un orden de mayor a menor impacto de todo el conjunto de malware.

2.2 Técnicas de ofuscación de malware

Como hacíamos mención brevemente al inició de la anterior subsección, hoy en día las muestras de malware cuentan con técnicas que los capacitan para intentar eludir la detección se sistemas especializados.

El usó de estas está a la orden del día ya que los individuos encargados de utilizar malware con objetivos maliciosos hacen uso de ellas a diario para conseguir su cometido.

A continuación, haremos mención a algunas de las técnicas más novedosas utilizadas hoy en día en orden cronológico de aparición.

2.2.1 Común

La técnica común es la técnica de ofuscación más simple que consiste únicamente en la representación del código de una manera anormal. La manera de hacerlo es más o menos simple y consiste en realizar un cambio de nombre a variables utilizando caracteres sin sentido y representando expresiones, que a priori son simples, de manera poco frecuente.

```
<script language=javascript>
    function factorial( val, is_transient){
        var ret = val ==  1 ? 1 : factorial( val-1, 1) * val;
        if (!is_transient) {
           document.write("factorial of" + val + "is: " + ret + "<br>");
        };
    };
    factorial(6, 0) ;
    factorial(3, 0) ;
    factorial(9, 0) ;
</script>
```

Fig. 6. Fragmento de código sin ofuscar.

```
<script language=javascript>
function z60b72bb3fe (z387ef0e78e, zd06054e7d1)
{
   var z205fd1aa25 = z387ef0e78e == (0x397+8978-0x26a8) ?
    (0xd81+6110-0x255e) : z60b72bb3fe( z387ef0e78e - (0x1083+838-0x13c8),
    (0x463+3498-0x120c)) * z387ef0e78e;
   if (!zd06054e7d1) {
     document.write("\x66\x61\x63\x74\x6f\x72\x69\x61\x6c\x20\x6f\x66\x20"
       + z387ef0e78e + "\x20\x69\x73\x3a\x20" + z205fd1aa25 +
       "\x3c\x62\x72\x3e");
    };
}
    z60b72bb3fe((0x11e8+2586-0x1bfc), (0xa63+7224-0x269b));
    z60b72bb3fe((0xfc5+2132-0x1816), (0x1119+3554-0x1efb));
    z60b72bb3fe((0x10b3+1338-0x15e4), (0x846+7200-0x2466));
</script>
```

Fig. 7. Fragmento de código ofuscado.

2.2.2 Cifrado

Esta técnica se apoya en el concepto de encriptación donde cada muestra de malware está compuesta por dos módulos principales, el modulo descifrador y el módulo principal o cuerpo del malware.

En este caso, el módulo principal está totalmente encriptado de manera que la única posibilidad de determinar en plano cual es la forma original de dicha muestra es haciendo uso del otro módulo, el descifrador.

Este caso ya dificulta en gran medida el trabajo de los antivirus, ya que aquellos que trabajan haciendo uso de la firma digital, no les será posible hacerlo mediante la firma del módulo principal, sino que deberán hacerlo mediante la del módulo descifrador.

El modus operandi de infección de este tipo de malware es siendo descifrado en primera instancia por el módulo de descifrado que acto seguido da paso a la ejecución del cuerpo principal.

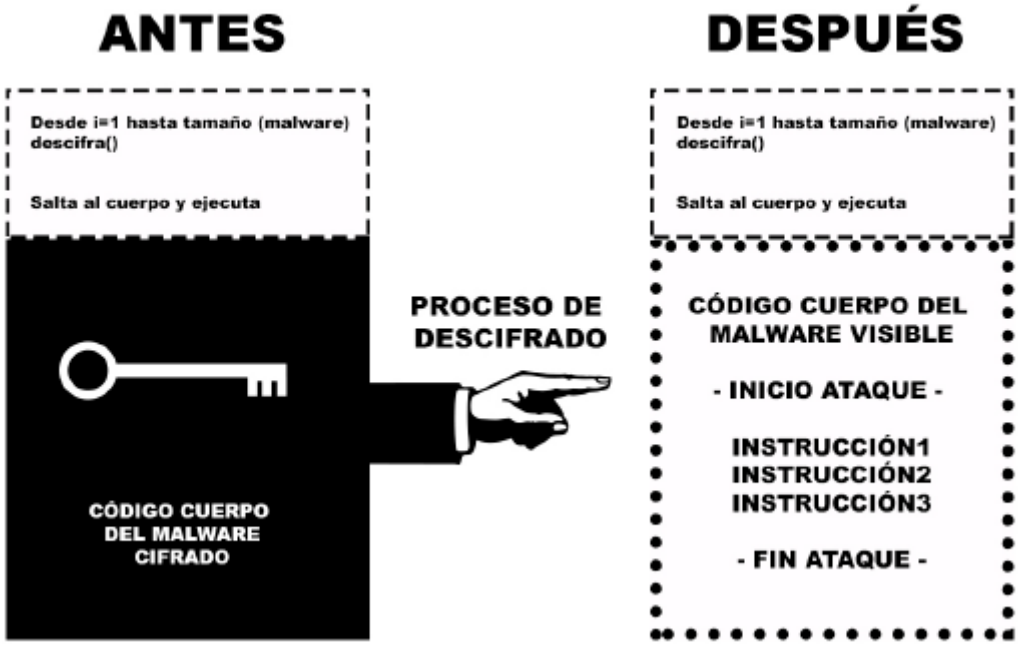


Fig. 8. Proceso de infección en malware cifrado.

2.2.3 Oligomorfismo

Está, al igual que la de cifrado, también hace uso de la encriptación. La estructura de las instancias de malware es exactamente la misma dando lugar a un módulo principal y a un módulo descifrador.

En este caso, a diferencia del método de cifrado, se dispone de una colección de descifradores estática, donde cada uno de ellos es capaz de descifrar el módulo principal. De esta manera, lo que se consigue es que el código descifrado obtenido no es idéntico en todos los casos, sino que varía dependiendo del módulo descifrador seleccionado ya que el descifrador es seleccionado de forma aleatoria cada vez.

Ese último punto, dificulta algo más la tarea de los antivirus, que no tienen bastante con la comprobación de un único descifrador, sino que han de hacerlo con varios.

En cuento al proceso de infección, este es idéntico al caso de la técnica de cifrado.

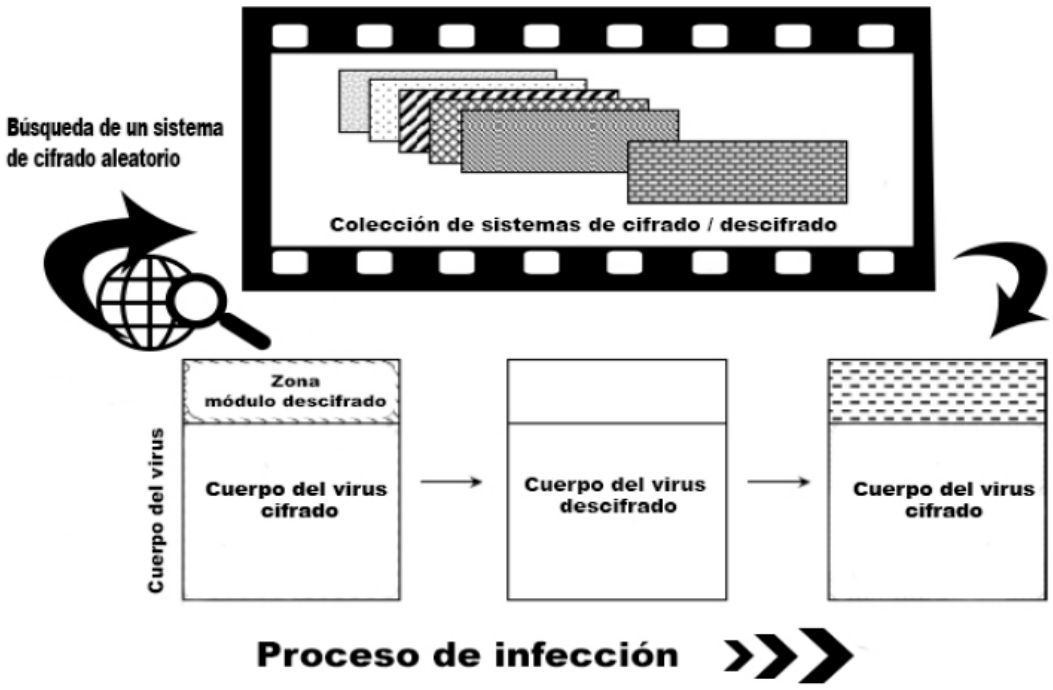


Fig. 9. Proceso de infección en malware oligomórfico.

2.2.4 Polimorfismo

De nuevo, en este caso, aparece en juego la encriptación, pero en este caso de una manera dinámica. A diferencia del caso de Oligomorfismo, la colección de descifradores no es estática, sino que el propio malware tiene la capacidad de generar nuevos descifradores todos ellos diferentes.

Esto, en cuanto a detección, sí que dificulta en gran medida la tarea de antivirus ya que no solo han de tratar con diferentes descifradores, sino que además estos se van generando de forma dinámica.

El proceso de infección, también es idéntico al de los anteriores casos.

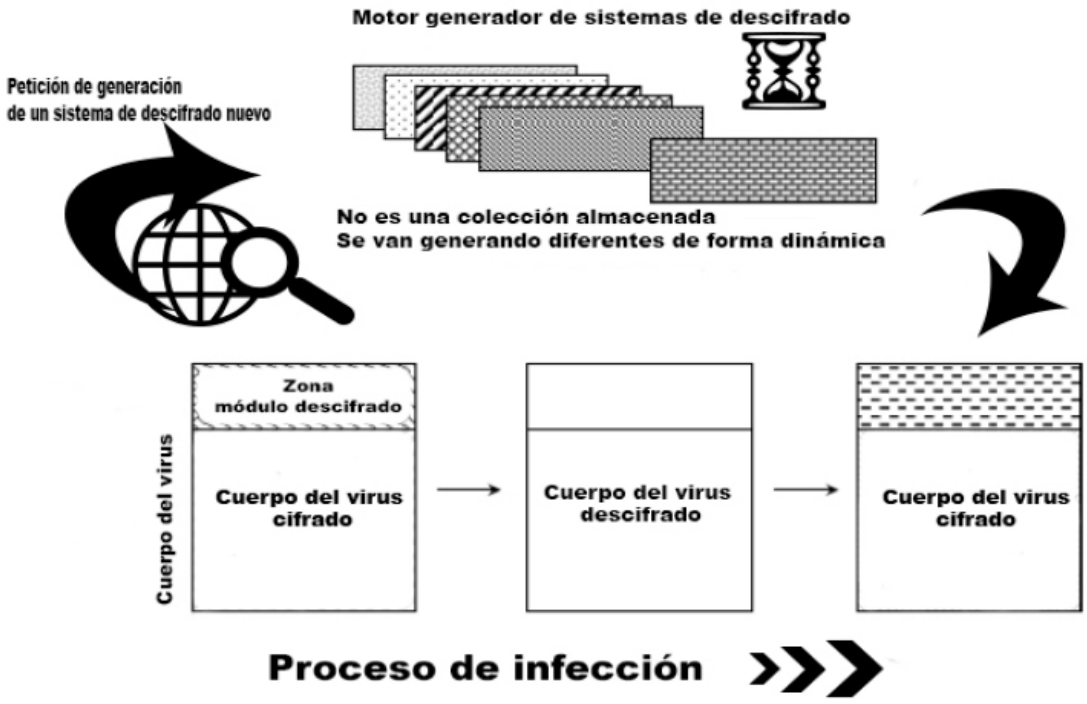


Fig. 10. Proceso de infección en malware polimórfico.

2.2.5 Metamorfismo

Este tipo de malware que hace uso de la técnica de metamorfismo dispone internamente de un motor metamórfico. En este caso, no se usa la encriptación, sino que lo que hace este tipo de malware es mutar todo su cuerpo y no solo el módulo descifrador, como en casos anteriores.

El motor metamórfico que permite a este tipo de muestras mutar debe contener las siguientes partes: desensamblador, analizador, transformador y ensamblador.

Una vez el propio malware localiza la ubicación de su código, la primera parte del motor de mutación, es decir, el desensamblador; lo que realiza es un proceso desensamblado para obtener el código fuente.

Tras obtener el código fuente, se le da paso al analizador cuyo objetivo es recabar información sobre la estructura y flujo de programa, subrutinas, variables y registros, entre otros.

Seguidamente, esta información es utilizada por el transformador, parte encargada de ofuscar el código además de cambiar la secuencia binaria de dicho malware.

Por último, es el ensamblador el que acaba transformando de nuevo el código mutado a código máquina para que pueda ser interpretado por la misma.

Esta es la técnica más sofisticada de todas, ya que es capaz de generar un número ilimitado de variantes similares en comportamiento y además sin vulnerabilidad a contener un patrón repetitivo que pudiese ser detectado.

En este caso, los sistemas especializados han de emplear técnicas basadas en el análisis de comportamiento, siguiendo un proceso similar al de la solución que se propone en este mismo documento.

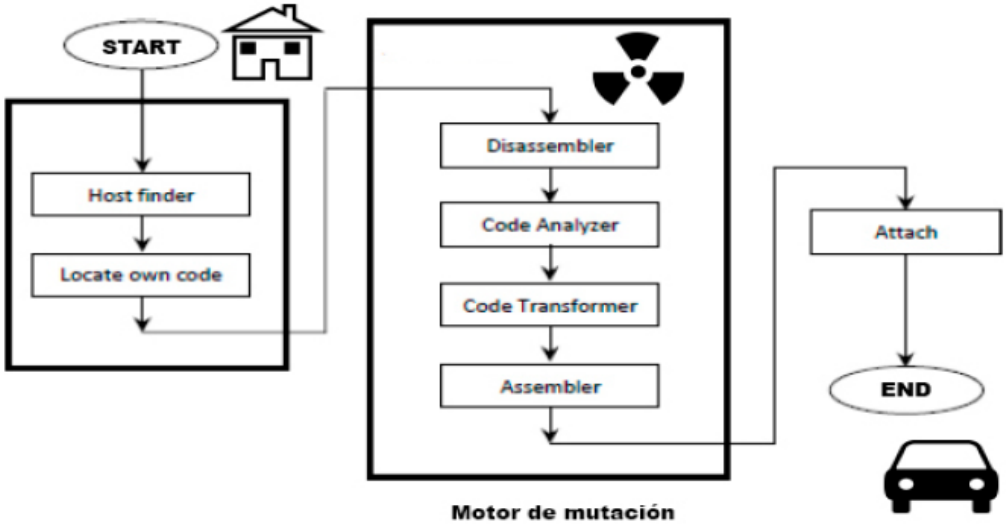


Fig. 11. Proceso de infección en malware metamórfico.

3 CLASIFICADOR DE MALWARE

La propuesta que se presenta a continuación pretende ante todo buscar la eficiencia para poder ser capaz de dar una respuesta en un tiempo lo más cercano al real posible, y además, ser lo más precisa posible para poder ser empleada en la determinación de incluso el malware más desconocido.

Dados los propósitos mencionados, y con el objetivo de cumplirlos, el diseño de la aplicación se basa en el uso de los diferentes algoritmos que contienen los procesos mencionados en el capítulo 2. *Antecedentes*.

Para el primero de ellos, al igual que en [1], se utiliza la plataforma online Anubis, la cual es capaz de ejecutar una muestra determinada de malware en un entorno virtual, y generar a partir de ésta un registro bajo XML, con todo detalle de las acciones realizadas por dicho malware en el sistema operativo; en nuestro caso Windows.

En lo que se refiere a la solución de cara al segundo punto, decidimos hacer uso del BOFM debido a sus bondades. Los autores de este modelo [2], ofrecen una solución óptima en cuanto a eficiencia y escalabilidad para el aprendizaje automático, que creemos que es uno de los elementos más innovadores a tener en cuenta en nuestro proyecto.

En cuanto al tercer punto, para la técnica de aprendizaje automático propiamente dicha, hacemos uso del Clustering, al igual que en el primer caso [1]. Finalmente, es el resultado del Clustering el que nos dará la base necesaria para aplicar el proceso de clasificación utilizando el concepto de prototipado.

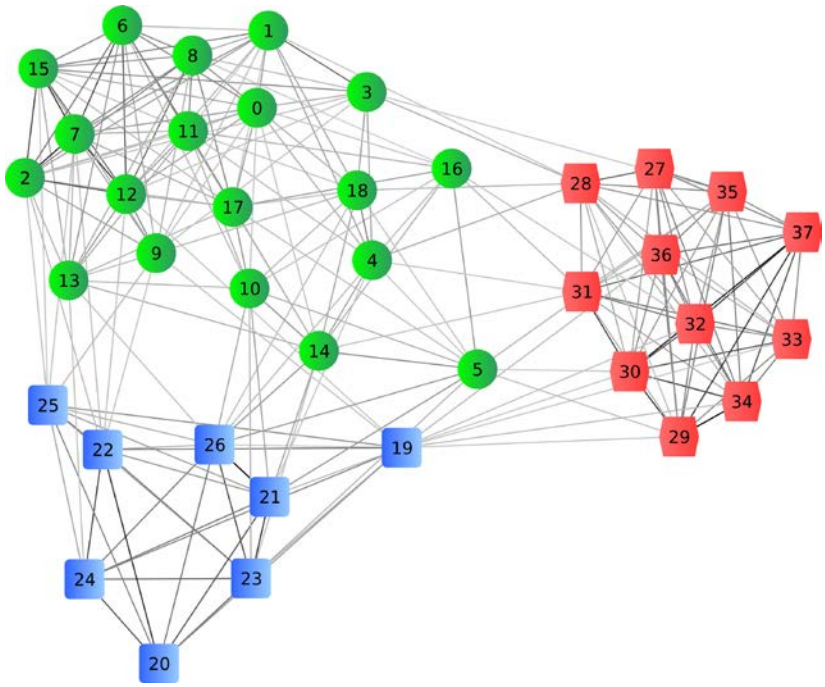


Fig. 12. Resultado final del Clustering (cada color representa un clúster distinto).

3.1 Anubis: Plataforma de análisis dinámico de malware.

En este primer apartado, pese a no ser un módulo propiamente de nuestro sistema, queremos hacer referencia a la herramienta de análisis de malware que se ha utilizado para llevar a cabo el proyecto: Anubis.

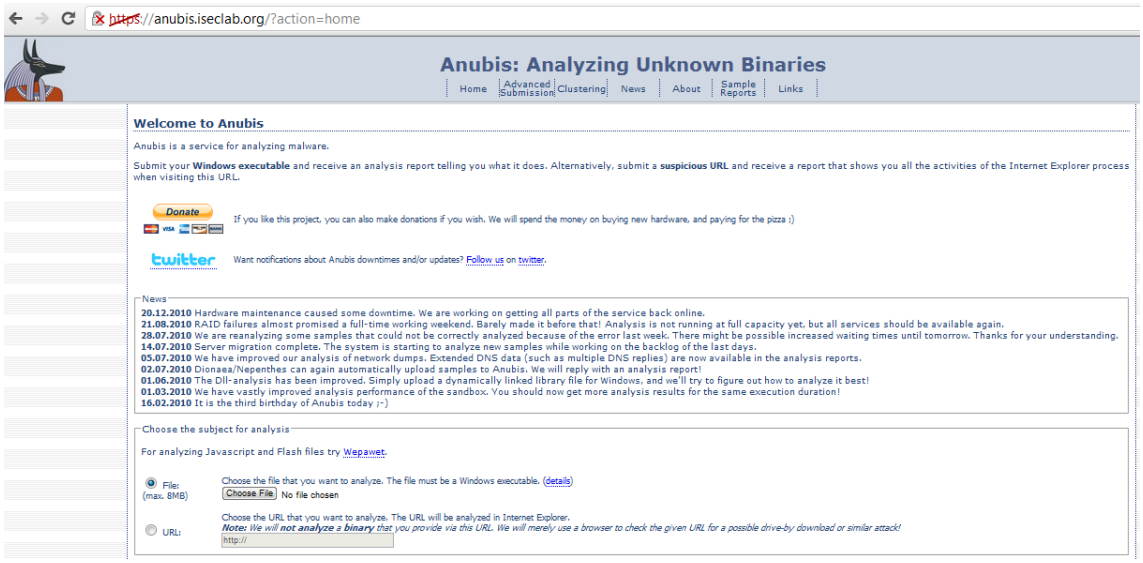


Fig.13. Interfaz web del menú principal de Anubis.

Tal y como lo definen sus autores, [2] consiste en una plataforma web de análisis de comportamiento malware, capaz de ejecutar muestras de malware en un entorno controlado y de extraer información concluyente a raíz de éstas. Para ello, lo que hace es emular la ejecución de estas muestras en un sistema operativo y durante la ejecución, interceptar tanto las llamadas al sistema, como a la API de Windows que se realizan; además de monitorizar el tráfico.

Como resultado, lo que ofrece Anubis al usuario es un registro detallado en formato XML o texto plano, con las diferentes acciones que la muestra en cuestión ha llevado a cabo. Además, éstas son clasificadas en función del recurso del sistema al cual han afectado, facilitando más todavía la comprensión de los datos extraídos.

Fig. 14. Fragmento XML del informe generado por Anubis para una muestra.

Los recursos del sistema que tiene en cuenta Anubis a la hora de realizar sus análisis son el sistema de ficheros, el registro del sistema de Windows, actividades de red, y la Interfaz Gráfica de Usuario (IGU) de Windows.

Como se menciona en [7], si no todos, la gran mayoría de malware lleva a cabo su objetivo entorno a los recursos mencionados anteriormente. Algunos de los comportamientos detectados para cada uno de ellos son los siguientes:

- Sistema de ficheros: Muchas de las instancias crean ficheros que consisten en una misma copia de su código binario pero polimórfico, con el objetivo de propagarse y cambiando el comportamiento en cada una de las nuevas instancias.

- Registro: Una gran cantidad de ellos siempre intentan que su eliminación sea lo más difícil posible. Para ello, lo que hacen es modificar ciertas llaves del registro de

Windows donde se especifican los programas que son automáticamente iniciados con el propio sistema operativo, en este caso Windows. La clave más usada es: *HKLM\System\Currentcontrolset\Services\%\Imagepat* seguida de: *HKLM\Software\Microsoft\Windows\Currentversion\Run%* .

- Actividad de red: Otra de las tareas comunes en malware es el escaneo de puertos, ya sea para detectar cuales están abiertos en el sistema o simplemente para abrir nuevos. Esto puede derivar en una causa grave, dando acceso a una entidad externa al sistema debido a la apertura indebida de los mismos.

Por último, mencionar algunas de las debilidades de Anubis y que sus mismos autores reconocen. Éstas están relacionadas con la habilidad del malware de ser capaz de detectar si está siendo monitorizado por un entorno como Anubis. Estas técnicas se basan principalmente en dos clases: *detección a nivel de API,* que radica en determinar la diferencia entre instrucciones de una CPU real y una emulada, y *detección a nivel de instrucción,* que analiza el entorno de ejecución haciendo uso de las propias funciones de la API del sistema operativo.

3.2 Modelado de características de comportamiento en un espacio acotado.

Como mencionamos anteriormente, hacemos uso de este algoritmo para implementar el segundo punto del aprendizaje automático y representa uno de los procesos más importantes en nuestro proyecto, porque de él deriva gran parte de la eficiencia de nuestro programa. Como se ha mencionado en el capítulo *2*, [5] encapsula el comportamiento de cada binario malware en un vector de 0's y 1's, cuyo tamaño resulta del sumatorio de las combinatorias que se hacen sobre el máximo número posible de acciones que puede llevar a cabo un conjunto de datos malware, sobre los recursos del sistema operativo.

Nuestra solución propone emplear el BOFM adaptándolo a los informes que extraemos de Anubis, no directamente sobre los binarios del malware. Con esto no sólo conseguimos obtener los conjuntos de acciones más fácilmente y rápidamente, sino que, además, al ser ficheros XML, ofrece la posibilidad de depurar cualquier problema encontrado mirando estos ficheros. Esto ayuda en gran parte a la hora de detectar acciones nuevas y añade transparencia a la resolución de los problemas.

Entrando más en detalle en el algoritmo, lo que hacemos en primer lugar es extraer el número de recursos del que dispone el sistema operativo en cuestión. En nuestro caso, que disponemos de Windows 7/10, tenemos los siguientes: registros de Windows, sistema de ficheros, procesos, y por último, elementos de sincronización, tales como semáforos de control ("mutexs"), gestores de espera entre eventos asíncronos, etc. Aunque podrían ser más recursos, los principales son estos cuatro, por el tamaño de cada uno y porque Anubis obtiene únicamente información de las acciones que se realizan sobre estos cuatro tipos de recursos principales. Todo ello, sumado a que la mayoría de las acciones extraídas de los informes malware afectan en la gran mayoría de casos únicamente a estos cuatro recursos mencionados, hace que no sea necesario extraer más recursos del sistema.

Una vez obtenidos los recursos del sistema operativo, obtenemos el máximo número de acciones posibles que puede llevar el conjunto de malware sobre ese recurso. Para ese fin, lo que hacemos es extraer "features": elementos de los recursos que se ven afectados por el malware y que tienen asociado un identificador característico y único según el tipo de recurso. En los registros por ejemplo es el nombre de la llave, en los ficheros el nombre de fichero, en procesos la id del proceso, etc. Por lo tanto, cada "feature" estará asociada a un tipo de recurso y a un conjunto de acciones que el malware en cuestión puede realizar sobre ésta.

Una vez generadas todas las "features", cada informe malware del conjunto, se mapeará a un vector como 0's y 1's según si presentan más o menos "features". Un binario malware que presentase todas las generadas se vería representado por un vector de 1's, mientras que no uno que no tuviese ninguna, por uno de 0's. De esta forma, con [5] acabamos obteniendo un vector para cada binario que representa su comportamiento, cuyos valores surgen de las ocurrencias de "features" que éste incorpora, en relación al conjunto maximal de ocurrencias sobre el total de recursos del sistema.

3.3 Arquitectura software del proyecto.

En esta sección se pasa a introducir la arquitectura del sistema, cuáles son sus módulos y cómo estos se comunican y complementan para llevar a cabo su cometido.

1. **Consola y control de entrada de parámetros:** Este módulo es aquel que permite la interacción del usuario u operador con el sistema, proveyéndole al mismo la información necesaria, ya sean directorios o parámetros de operación, para la correcta ejecución del programa. Además, también se encarga del control de los posibles errores que puedan derivar de la interacción humana, evitando así posibles roturas del flujo de programa que podrían aparecer contrariamente.

2. **Módulo de lectura de los XML's:** En este segundo módulo, el proceso que se realiza es el de extraer la información necesaria de los informes XML obtenidos de la plataforma online Anubis. Es gracias a este módulo que el sistema puede iniciar su cometido ya que deja preparada toda la información necesaria que será utilizada por los módulos posteriores.

3. **Conversión mediante BOFM:** Este es el primero de los módulos que componen el núcleo del programa como tal. Su función es tratar la información del módulo anterior y convertirla mediante el algoritmo del BOFM mencionado en el anterior capítulo, en vectores de tamaño fijo, para que éstos puedan ser representados dentro del espacio vectorial acotado, el cual se usará en los siguientes módulos.

   Cumple una de las funciones más importantes de todo el programa porque garantiza un mapeo sencillo de las muestras y una gran eficiencia en cuanto a volumen del espacio vectorial y a la rapidez de tratamiento de las muestras. Todo ello se traduce en un gran aumento de rendimiento en la ejecución.

4. **Extracción de los prototipos:** Este módulo es el encargado de la extracción de prototipos, alimentando a los dos módulos posteriores, el de Clustering y clasificación. Cabe decir que trata tanto con muestras de test como de entrenamiento.

5. **Clustering:** Como módulo de aprendizaje automático tenemos el módulo de Clustering, que es el encargado de agrupar elementos a priori desconocidos en grupos donde sus integrantes son en mayor o menor medida similares. Es vital para la tarea de clasificación ya que sin él no se podría extender el concepto de prototipado y clasificación al resto de muestras.

6. **Clasificación:** En este caso, tenemos el módulo de clasificación, que como su nombre indica se encarga de atribuir una determinada clase de malware a cada una de las muestras de test inicialmente desconocidas. Este módulo se alimenta del conjunto de muestras de entrenamiento previamente clasificadas por la intervención humana para llevar a cabo su cometido.

7. **Análisis incremental:** Este séptimo módulo es el que engloba a todos los explicados en los puntos anteriores. Su objetivo es engranar los diferentes algoritmos del clasificador de malware para obtener un rendimiento tanto a nivel funcional y de eficiencia, como de resultados a cada iteración.

8. **Evaluación y guardado de las acciones de las muestras:** Este módulo aprovecha la lectura y extracción de información de los informes XML, explicada en el punto 2, para leer el conjunto de acciones de cada muestra y guardarlos en diferentes sets dentro de la misma instancia para su posterior uso.

9. **Cómputo de las puntuaciones para cada muestra**: Gracias a los conjuntos del punto anterior, se calcula la puntuación global para cada muestra, determinando su puntuación final en el ránking de peligrosidad. Ello conlleva el orden dentro de este ránking y la etiqueta "threat-level" que le será asignada, la cual puede ser: Low, Medium, High o Very High.

10. **Generación y muestra de resultados para módulo de clasificación y de peligrosidad:** Último módulo del programa. Se encarga de listar y escribir en ficheros de texto (.txt) tanto los resultados del módulo de clasificación, como los del de peligrosidad.

A continuación, se muestra una imagen en donde se capturan todos y cada uno de estos módulos, así como las relaciones existentes entre ellos:

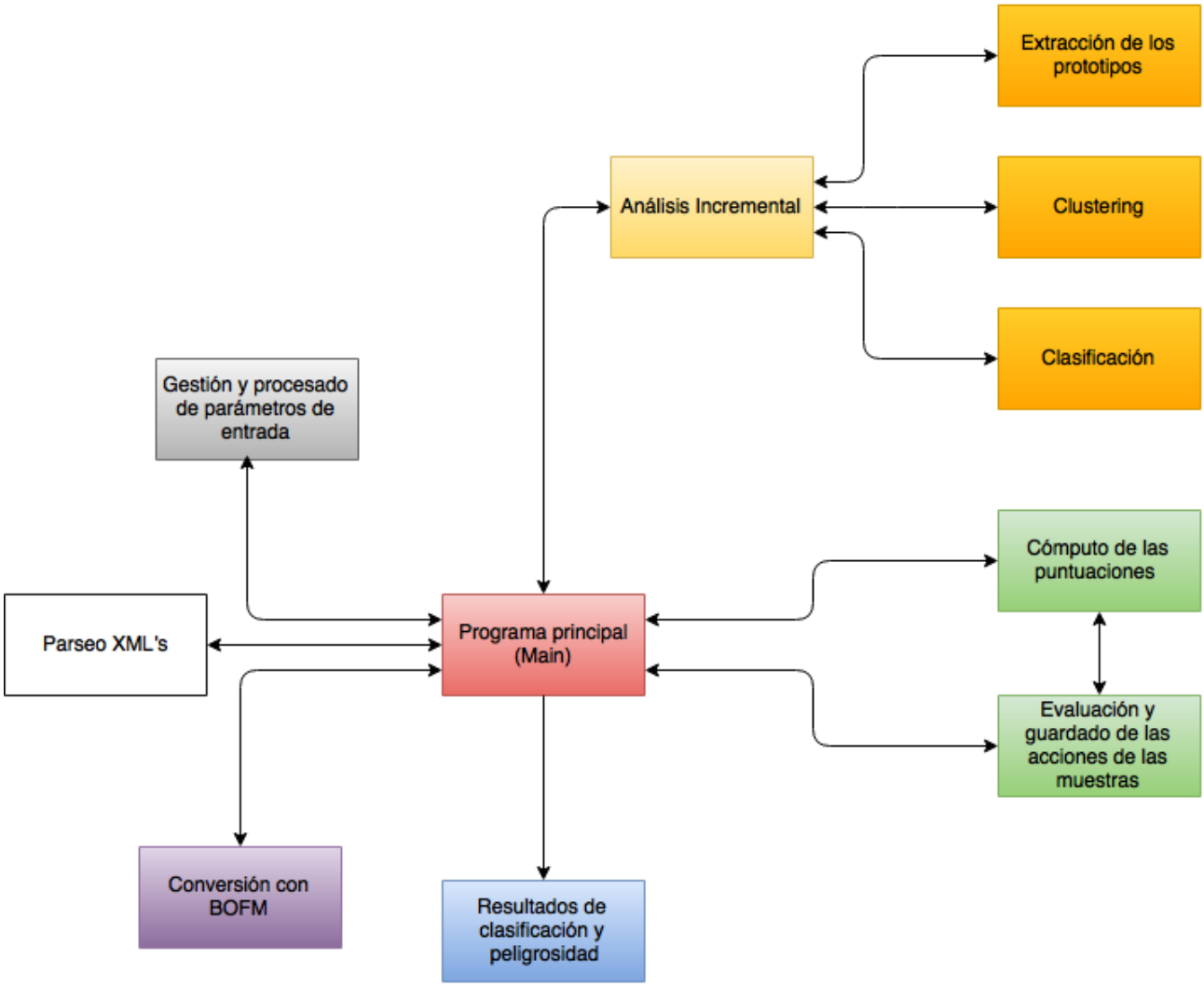


Fig. 15. Diagrama de módulos lógicos.

Como se puede apreciar en la imagen, existe un módulo principal conocido como el "main" de todo el programa, que actúa como origen del mismo. A partir de éste, surge la comunicación con el conjunto de módulos restantes, los cuales llevan a cabo el cometido del programa en su conjunción. Nótese que esta comunicación siempre se hace de manera bidireccional, representando que ésta vuelve al "main" cada vez que pasa por otro módulo, devolviendo los resultados obtenidos de éste, y pasando al próximo.

Por último, el único de ellos que es unidireccional es el de *resultados de clasificación y peligrosidad*, que, aunque si bien es cierto que acaba retornando al módulo principal para la finalización del programa, su principal cometido no es tratar datos que posteriormente serán utilizados por otros módulos, sino simplemente generar la salida del programa.

3.4 Extracción de prototipos

Otro punto en el que se basa nuestra propuesta, al igual que en el caso del artículo [1], consiste en la idea de la extracción prototipos, cuya finalidad radica en la búsqueda de la eficiencia en tiempo.

Tal y como se explica en el capítulo *2.*, un prototipo es una muestra como otra cualquiera, pero con la exclusiva característica de ser capaz de representar de la forma más homogénea posible a un conjunto de muestras. Obviamente, para hacer una representación fiel a la realidad, ésta requiere de más de un único prototipo, por lo que se necesitan varios para abarcar con certeza y rigor a todo el conjunto.

Fig. 16. Analogía gráfica de la idea de prototipo.

Nuevamente, hacemos énfasis en la dificultad de computación que conlleva el uso del algoritmo de prototipado, el cual está demostrado ser "NP-hard". Dado que uno de los objetivos que nos planteamos inicialmente en la elaboración del proyecto es buscar la eficiencia de nuestro sistema, así como la viabilidad de éste, está claro que no podemos asumir semejante coste de computación. Por ello, al igual que en [1] hacemos uso del algoritmo resultante de la adaptación del algoritmo de (Garey y Johnson), que es capaz de dar una solución cuyo coste temporal es dos veces el tiempo de la solución óptima y que, por tanto, es accesible para nosotros.

Entrando en lo que es el funcionamiento del algoritmo, este inicia considerando la distancia euclidiana entre todos y cada una de las muestras como infinito. A partir de ahí, empieza seleccionando la muestra asociada a la mayor de las distancias, la cual se convertirá en prototipo, y calculando la distancia euclidiana de ésta con el resto. El objetivo de esto, no es otro que actualizar los valores de las distancias de las distintas muestras, y para ello se comprueba si la distancia calculada en la iteración actual es menor a la distancia de la iteración previa (excepto la primera que es infinito). En caso afirmativo, se le asigna a la muestra objetivo la distancia en cuestión. Finalmente, la muestra seleccionada como prototipo es descartada para evitar ser seleccionada de nuevo en la próxima iteración de este proceso.

En definitiva, el objetivo del algoritmo es ir seleccionando muestras de forma que el valor de la distancia de todas las muestras sea la menor posible, consiguiendo así que toda muestra tenga un prototipo mínimamente cercano. Este proceso es iterativo mientras el valor máximo de las distancias supere el valor de una constante *Dp*.

```
1: prototypes ← empty
2: distance[x] → infinity for all x reports

3: while max(distance) > Dp
4:   choose z such that distance[x] == max(distance)
5:    for x € reports and x != z do
6:          if distance[x] > euclidean_distance(x,z) then
7:              distance[x] = euclidean_distance(z,x)
8:    add z to prototypes
```

Fig. 17. Seudocódigo del algoritmo de extracción de prototipos.

3.5 Clustering usando prototipos.

Tal y como se ha mencionado ya, como técnica de aprendizaje automático decidimos emplear el C*lustering*. Esta técnica busca la agrupación de diversos elementos, a priori desconocidos, a través de la relación entre las propiedades intrínsecas de éstos, las cuales se codifican como dimensiones en un espacio vectorial. Por lo que básicamente, los elementos son agrupados según la cercanía que presentan en este espacio, basándose en las distancias euclidianas calculadas entre éstos.

El algoritmo parte inicialmente de un conjunto de muestras individuales dentro del espacio vectorial de dimensión *N*, en nuestro caso fijada por el BOFM. Estas muestras son extraídas del conjunto de test haciendo uso del algoritmo de prototipado, las cuales, además, son consideradas clústeres en un inicio. Posteriormente el algoritmo trata de unir cada uno de ellos con otro candidato, intentando así formar clústeres de más de un elemento.

Para ello, en primera instancia únicamente se calcula la distancia euclidiana entre cada par de clústeres, tomando como candidatos a unirse aquellos clústeres con menor distancia, y por tanto más cercanos.

Tras determinar los candidatos, se hace uso de la técnica de *enlace completo,* que básicamente consiste en que cuando dos clústeres son unidos, la distancia de este nuevo clúster con el resto, es la máxima distancia de los clústeres individuales que conformaban al recién fusionado, con el resto de clústeres. Este proceso de unificación se realiza hasta que la distancia mínima entre cada par de clústeres es mayor que una constante dada: *Dc***.**

Finalmente, tras formar clústeres únicamente con muestras prototipadas, el algoritmo se extiende al resto. Para ello, asigna cada una de las muestras, al clúster al cual pertenezca el prototipo más cercano a ella, es decir, el prototipo que mejor la represente.

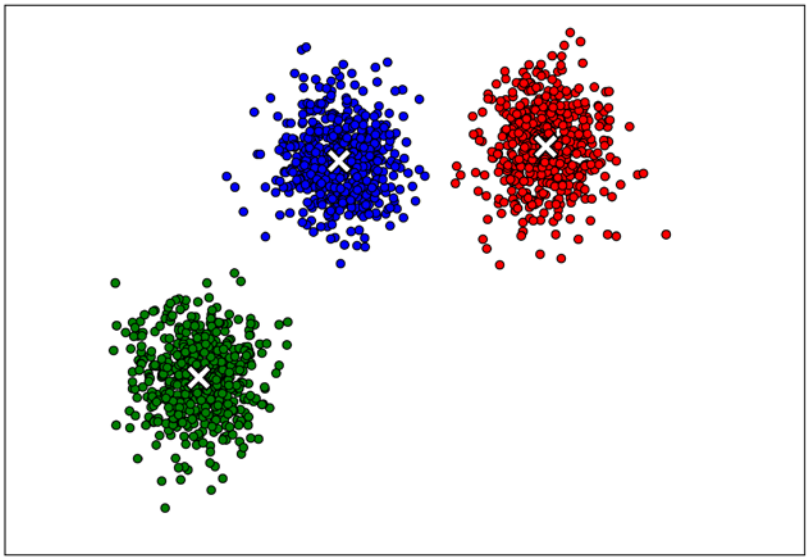


Fig. 18. Ejemplo de clústeres con su prototipo identificado.

Por último, aquellos clústeres con menos de X elementos, donde X es una constante impuesta de forma tácita, son rechazados ya que se considera que el número de muestras que lo

conforman no es suficientemente determinante y podríamos estar tratando con un tipo de malware infrecuente.

```
1: clusters_vector ← empty
2: distances_vector ← empty

3: for p ∈ test_prototypes do
4:     z ← new cluster(p)
5:     add z to clusters_vector

6:     for each p' ∈ test_prototypes do
7:         euclidean_distance(z, p') ← ||A(z) - B(z')||
8:         add euclidean_distance(z, p') to distances_vector

9: d ← min(distances_vector)
10: while d < Dc
11:     z, z' ∈ clusters_vector where d = euclidean distance(z,z')
12:     merge clusters z,z'
13:     update distances_vector using complete linkage

14: for x ∈ test_reports do
15:     p ← nearest prototype to x
16:     z ← cluster p belongs to
17:     assign x to cluster z

reject clusters with less members than m
```

Fig. 19. Seudocódigo del algoritmo de Clustering.

3.6 Clasificación usando prototipos.

El algoritmo de clasificación, al igual que el de Clustering se fundamenta en el concepto de prototipos, para posteriormente, extender el resultado al resto de muestras.

El proceso de clasificación podríamos dividirlo en dos partes, prácticamente idénticas pero que conviene separarlas para tener un mejor trazo del mismo.

La primera de ellas se encarga de clasificar los prototipos extraídos, del conjunto de test, durante la fase de extracción de prototipos. Para ello lo que se hace es buscar cuál de los prototipos del conjunto de entrenamiento es más cercano a cada uno de éstos. Llegado este punto, se compara si el resultado de la distancia euclidiana entre ambos es menor a una constante Dr y en caso positivo, simplemente se asigna dicho prototipo al clúster al cual pertenece el prototipo de entrenamiento.

La segunda de ellas, una vez se tienen los prototipos clasificados, se encarga de determinar cuál de los prototipos de test es el más cercano a cada una de las muestras de test, y que no son prototipos. De nuevo, al igual que en la primera parte asignamos dicha muestra al clúster al cual ha sido previamente asignado su prototipo.

Por último, cabe mencionar una modificación realizada al algoritmo de clasificación presentado en el apartado [1] para evitar una situación en particular que puede darse: la de que un prototipo de test no haya podido ser suficiente similar a un prototipo de entrenamiento, y por ende no haya sido asignado a ningún clúster al final de la ejecución; y tampoco sus simpatizantes sean clasificados.

La solución un poco más relajada que se propone es, además de realizar el comportamiento de [1], intentar clasificar aquellas muestras de test cuyo prototipo no ha podido ser clasificado contra todos los prototipos de entrenamiento y no solo contra los de test. De esta forma, al final del programa y como última instancia, aquellos informes no clasificados se intentarán asociar a los de entrenamiento, con la pretensión de que el número final de informes sin clasificar sea menor a cada iteración.

En otras palabras, lo que se hace es dar la posibilidad de que, si una muestra no es suficientemente similar a un prototipo, que recordemos, intentan representar de la manera más homogénea a todo el conjunto; ésta pueda ser asignada a otra muestra del conjunto de entrenamiento.

```
1: test_prototypes[x]
2: training_prototypes[y]
3: test_reports ← reports - test_prototypes
4: classified_test_prototypes ← empty

5: for x ∈ test_prototypes do
6:     choose y from training_prototypes such that min(euclidean_distance(x,y))
7:     if euclidean_distance(x,y) < Dr then
8:         assign x the cluster y belongs to
9:         add x to classified_test_prototypes .

10: for x ∈ test_reports do
11:     if x has not been rejected then
12:         find nearest prototype z to x in classified_test_prototypes
13:         if z exists then
14:             assign x the cluster z belongs to
15:         else
16:             jump to classify_against_training_prototypes:
17:     else
18:         classify_against_training_prototypes:
19:         for x ∈ training prototypes do
20:             find nearest prototype y to x from training_prototypes
21:                 if euclidean_distance(x,y) < Dr then
22:                     assign x to to the cluster y belong
23:                     add x to classified_test_prototypes .
```

Fig. 20. Seudocódigo del algoritmo de clasificación.

3.7 Análisis incremental.

De la idea de combinar todos los algoritmos descritos en las anteriores secciones aparece el concepto de análisis incremental. La base de este concepto es ofrecer al algoritmo cuanta más capacidad de clasificación mejor.

Fig. 21. Representación del concepto de análisis incremental.

Para ello, dividimos el análisis incremental en cinco tareas, todas ellas relativas a los algoritmos anteriores.

En la primera de ellas se realiza la extracción de los prototipos a partir del conjunto de entrenamiento, los cuales nos darán entrada para la segunda tarea.

En esta segunda, hacemos uso de los prototipos extraídos y aplicamos el algoritmo de clasificación contra todas las muestras de test, haciendo uso de estos prototipos. Lo que consigue aquí el algoritmo es realizar una primera clasificación de las muestras de malware del conjunto de test, que pertenecen a clases malware conocidas.

Ahora entramos en la tercera tarea, donde es necesaria otra extracción de prototipos, pero esta vez haciendo uso del conjunto de entrenamiento, que incluirá todos aquellos elementos que no han sido clasificados en la tarea anterior.

En la cuarta, aplicamos directamente Clustering sobre los prototipos extraídos en la tercera tarea, extendiéndolo al resto de muestras, tal y como se explica en la sección *3.3 Clustering usando prototipos.* Cabe añadir que la fase de descarte durante el Clustering permite al análisis incremental el poder aglomerar muestras pertenecientes a clases infrecuentes o desconocidas, hasta tener suficientes como para poder realizar un correcto Clustering posterior sobre éstas.
La idea detrás de esto reside en que la clasificación de muestras infrecuentes como elementos individuales es una tarea difícil, sin embargo, de forma grupal, si la clasificación del prototipo de las mismas es posible y la disparidad entre ellos es asumible, la clasificación sí será factible.

Finalmente es interesante hablar sobre el criterio de parada del algoritmo. En caso de que éste no sea capaz de clasificar todas las muestras por equis motivos (ej. insuficiencia de muestras de entrenamiento, distribución de muestras muy dispar debido al elevado número de muestras infrecuentes, etc.) ha de ser posible terminar su ejecución.
Lo que proponemos nosotros es limitar la ejecución del algoritmo a un número determinado de iteraciones y forzar el cese de ésta una vez superado el número. A cada iteración éste comprueba si ha sido capaz de clasificar alguna muestra de test y en caso erróneo toma constancia de ello y lo vuelve a intentar en siguientes iteraciones hasta alcanzar el límite establecido.

```
1: dead_iterations ← 0
2: max_die_times ← 5
3: test_prototypes ← empty
4: test_reports ← reports from test
5: training_reports ← reports from training
6: reports_not_classified ← size of test_reports
7: dead_iterations ← 0
8: training_prototypes ← extract prototypes from training_reports

9: while dead_iterations != max_die_times and reports_not_classified  != 0
10:     classify test_reports to known clusters using training_prototypes
       (* when a test_report instance is classified it is treated as a training report)
11:     update training_prototypes extracting prototypes from training_reports
12:     test_prototypes ← extract prototypes from test_reports
13:    cluster test_reports using test_prototypes
14:    reports_not_classified_after_iteration ← reports from test not classified yet

15:     if reports_not_classified == reports_not_classified_after_iteration then
16:       increment by one dead_iterations
17:   else
18:      reports_not_classified = reports_not_classified_after_iteration
```

Fig. 22. Seudocódigo del algoritmo de análisis incremental.

3.8 Predicción de daños malware.

El segundo de los dos módulos que componen nuestro sistema, es el referente a la predicción de daños, cuya función reside en la capacidad de cuantificar cuán dañino puede ser una determinada muestra si finalmente se llega a ejecutar en un sistema operativo.

Esta idea fue concebida con la intención de que la propuesta fuese capaz de generar un ranking de peligrosidad para ayudar a futuros operadores a crear y efectuar planes de actuación. Un ejemplo de escenario podría ser el de decidir a qué malware atacar en primera instancia con el objetivo de minimizar los daños en el sistema a corto o largo plazo.

En cuanto a su funcionamiento, este consta de un total de tres pasos sucesivos, que son los siguientes:

El primero de ellos consiste en la asignación de puntuaciones a las diferentes acciones que el conjunto de malware en su totalidad puede llegar a efectuar en el sistema operativo que nos atañe, en este caso Windows. Estas puntuaciones son asignadas en función de los efectos que pueden causar a nivel individual, cuanto más dañina, mayor es la puntuación y viceversa.

El segundo de ellos hace uso del anterior y tiene como objetivo el cálculo y asignación de una calificación global a cada una de las muestras. Esta calificación depende única y exclusivamente del tipo y número de acciones que determinan el comportamiento de cada una.
El proceso a seguir no es más que realizar el sumatorio de todas las puntuaciones correspondientes a las acciones que las definen.

Llegados a este punto, en primera instancia se decidió realizar una media ponderada sobre el número de muestras de forma que aquellas muestras con menos acciones, pero no por ello menos dañinas, salieran perjudicadas de forma notable. Sin embargo, pronto nos dimos cuenta de que esta no era la mejor solución ya que, si bien es cierto que muestras pequeñas no salían perjudicadas, lo sufrían aquellas muestras grandes con acciones de puntuación baja/media. Estas últimas conseguían una puntuación global mayor al resto, pero a la hora aplicar la media salían desfavorecidas por la poca aportación de estas en el cómputo global.

Por ello, se decidió modificar el proceso explicado anteriormente rechazando esta primera vía y pensando en una alternativa.
La nueva propuesta lo que pretende es basarse en la función logaritmo $f(x) = log_{10}(x)$.

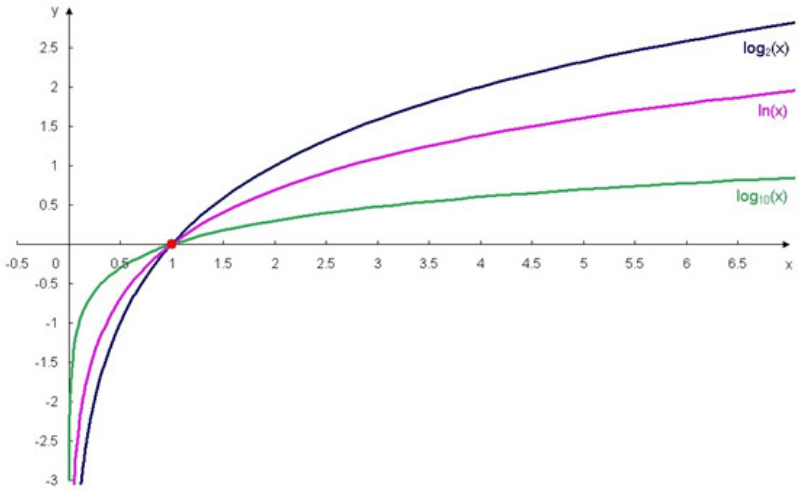


Fig. 23. Gráfica de funciones logaritmo de distinta base.

dejando a un lado la función lineal $f(x) = mx + b$.

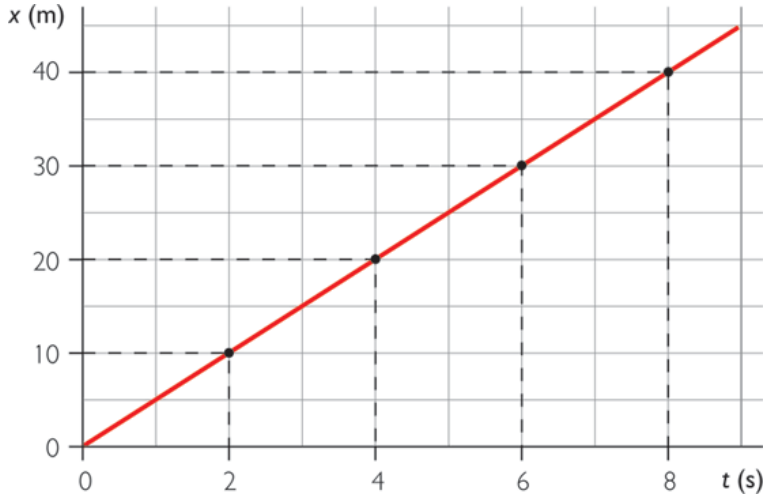


Fig. 24. Gráfica de la función lineal o recta.

La función logaritmo lo que nos permite es minimizar o suavizar el rango de valores que se pueden obtener durante el cálculo del cómputo global de cada muestra. Por definición, la función logaritmo tiene un crecimiento menor al de la función lineal, siendo además la operación inversa a la exponencialización.

Si consideramos que tenemos un rango [10, 1000] representado por una función lineal, el rango de valores que obtendremos aplicando la función logaritmo (en este caso en base 10) será de [1, 3].

Seguidamente, se realiza la diferencia de los valores máximo y mínimo logarítmico, y este resultado se utiliza para crear un sistema de rangos que posteriormente será utilizado para cuantificar la peligrosidad de las muestras.

Por último, cabe mencionar que no existe implementación de desempate, con lo cual, aquellas muestras que obtengan una misma puntuación serán tratadas de igual manera por el sistema sin ningún tipo de distinción.

3.9 Valoración de peligrosidad de las acciones

A continuación, detallamos qué valores han sido asignados al conjunto de acciones posibles, argumentando brevemente los motivos que han llevado a cada elección basándonos en la documentación ofrecida por el foro de Anubis [14]. Para este fin, dividimos las acciones en base al recurso del sistema operativo que afectan:

Registro:

Nombre de acción	**Puntuación**
REG_KEY_CREATED	2
REG_KEY_MODIFIED	3
REG_KEY_VALUE_READ	1
REG_KEY_MONITORED	5
REG_KEY_DELETED	5

Tab. 1. Pesos asignados a las acciones de registro.

- REG_KEY_CREATED: Es una acción que de por sí sola no supone mucho riesgo, pero cuando varias llaves son creadas estratégicamente pueden manipular el comportamiento de programas enteros según las características de éstos. Aún con todo, es más probable que esto último ocurra al modificar registros y no al crear de nuevos.

- REG_KEY_MODIFIED: Acción que consiste en modificar las propiedades de una llave de registro. Según el programa al que esté asociado la llave puede resultar muy dañino o inofensivo. Dañino si por ejemplo se modifican registros que controlan el lanzamiento de procesos, el inicio de servicios, el gestor de tareas de Windows, etc. Un ejemplo de ello sería una modificación a la siguiente llave de arranque “startup”: (HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders.

- REG_KEY_VALUE_READ: Leer un valor de llave de registro. De por sí es algo inofensivo; la acción como tal. No obstante, es un indicador de algo mayor, ya que la lectura de un registro es probable que implique una posterior modificación del mismo, o que se está comprobando cierto aspecto de un programa para saber por dónde atacarle.

- REG_KEY_MONITORED: Monitorización de una llave de registro por parte de un host remoto. Ya sea por un motivo u otro, indica en la mayoría de casos que un agente externo está intentando acceder al ordenador afectado o quiere emplearlo para motivos dañinos. Este tipo de acción está bastante presente en los troyanos, por lo que es de alto riesgo, al igual que la tipología de malware en cuestión que los incorpora.

- REG_KEY_DELETED: Eliminación de una llave de registro. Los efectos de semejante acción pueden desencadenar en una multitud de problemas diferentes, muchas veces difíciles de reconocer o concretar, por lo que supone una acción muy peligrosa en según qué casos, que puede afectar al sistema de forma irreversible.

Sistema de ficheros:

Nombre de acción	**Puntuación**
FILE_CREATED	3
FILE_MODIFIED	3
FILE_READ	1
FILE_DELETED	5
DIRECTORY_CREATED	4
SECTION_OBJECT_CREATED	6
FS_CONTROL_COMM	6
DEVICE_CONTROL_COMM	12

Tab. 2. Pesos asignados a las acciones del sistema de ficheros.

- FILE_CREATED: Crea un archivo en una ruta del sistema operativo. Según el tipo de archivo creado será mayor o menos el índice de peligrosidad, por lo tanto, la puntuación es un promedio entre los dos posibles casos.

- FILE_MODIFIED: Modifica las propiedades de un fichero genérico. Al igual que con la creación, dependiendo de las propiedades modificadas se puede hacer mucho daño o ninguno.

- FILE_READ: Lee de un fichero. El daño es similar al leer una llave de registro: la acción como tal no causa daños físicos, pero puede sustraer información para un posterior ataque.

- FILE_DELETED: Borra un archivo del sistema de ficheros. Es igual de peligroso que borrar una llave de registro, pero dado que un fichero puede encapsular mucha más información y parámetros, el riesgo y los daños provocados pueden resultar mucho mayores.

- DIRECTORY_CREATED: Crea un directorio nuevo. Según la cantidad de subdirectorios, carpetas, etc. que se incluyan en este, será más o menos peligroso. Consecuencias derivadas de esta acción serían: creación de librerías malware, reemplazamiento de dlls de Windows por otra modificada, etc.

- SECTION_OBJECT_CREATED: Creación de memoria compartida. Es muy explotable porque el acceso a ésta permite acceder a otros recursos del sistema operativo: procesos, servicios, archivos, etc. A cualquier otro recurso en general que se ubique dentro de esa sección de memoria.

- FS_CONTROL_COMM: Control de sistemas de ficheros. De alta peligrosidad porque permite el borrado de los ficheros de ese sistema en cuestión, así como el duplicado de éste, lo cual puede llevar al flooding del disco duro.

- DEVICE_CONTROL_COMM: Permite el control de dispositivos periféricos conectados al sistema, así como de los componentes internos del ordenador. Posiblemente la acción más peligrosa que puede llevar a cabo el malware, sobre todo si afecta al sistema de arranque del ordenar (boot), ya que puede dañarlo de forma permanente.

Procesos y derivados

Nombre de acción	**Puntuación**
REMOTE_THREAD_CREATED	7
FOREIGN_MEM_AREA_READ	2
FOREIGN_MEM_AREA_WRITE	7
PROCESS_CREATED	10
PROCESS_KILLED	9

Tab. 3. Pesos asignados a las acciones de procesos y derivados.

- REMOTE_THREAD_CREATED: Crea un hilo de ejecución de forma remota en el ordenador. Muy peligroso según el comportamiento del hilo, ya que puede llevar a cabo cualquier tipo de tarea según la cadena de instrucciones que lo conformen.

- FOREIGN_MEM_AREA_READ: Lectura de la memoria. Al igual que el resto de lecturas, el hecho en sí no es dañino, pero las consecuencias posteriores sí lo son. Además, al ser una lectura de la memoria, es peor todavía y conlleva mayor riesgo si cabe.

- FOREIGN_MEM_AREA_WRITE: Escribe en memoria. Muy peligroso porque puede provocar fallos gordos en el sistema, pérdida de información de los programas en ejecución, incoherencias en la memoria, en los procesos, servicios etc.

- PROCESS_CREATED: Crea un proceso malicioso que puede incorporar a su vez varios hilos de ejecución, propiedad que hace que esta acción sea potencialmente peligrosa, ya que puede provocar daños en múltiples áreas del sistema operativo simultáneamente.

- PROCESS_KILLED: Mata a un proceso activo, lo que puede provocar la interrupción del funcionamiento de diversos aspectos del sistema operativo: conexión web (svchost), inicio de los servicios, indexación del sistema de ficheros, etc. Según el proceso matado puede resultar desastroso.

Sincronización

Nombre de acción	Puntuación
MUTEX_CREATED	6
EXCEPTION_OCURRED	4
KEY_WAS_CHECKED	1

Tab. 4. Pesos asignados a las acciones de sincronización.

- MUTEX_CREATED: Crea un semáforo que controla la sincronización de procesos en ejecución dentro del sistema operativo. La creación de semáforos adicionales puede provocar problemas en la sincronización que pueden resultar en efectos desastrosos: interrupciones de múltiples procesos, acceso múltiple simultáneo a un recurso compartido generando así, memoria incoherente, etc.

- EXCEPTION_OCURRED: Lanza o genera una excepción. No hay detalles claros sobre esta acción al respecto, pero puede generar problemas de todo tipo de forma aleatoria.

- KEY_WAS_CHECKED: Comprueba el valor de una llave del registro. Su finalidad es similar a la lectura, por lo que conllevan un mismo o similar efecto.

3.10 Control de entrada y gestión de errores

La utilización de nuestro software conlleva obligatoriamente la interacción del usuario, por ello es necesario que este sea capaz de absorber o corregir errores que pueda cometer el mismo durante la utilización del sistema.

Una vez se inicia la ejecución del programa gracias a los parámetros de entrada que el operario o usuario ha introducido, los primeros pasos que se realizan son a través del módulo de control y gestión de errores. Éste lo que pretende es que cualquier dato externo recibido como entrada no pueda afectar negativamente la ejecución o evitar que directamente que el sistema operativo la aborté por algún posible error posterior.

Este control comprueba diferentes características de los parámetros de entrada como son:

- Petición de ayuda por parte del usuario.
- Falta de todos o algún parámetro de entrada.
- Detección de posibles errores de sintaxis.
- Detección de algún parámetro de control no soportado.

Fig. 25. Vínculo entre el control de errores y la calidad en el software.

Si alguno de estos casos se produce y se considera que la ejecución puede seguir su transcurso, el programa realizará su cometido produciendo la salida correspondiente. De lo contrario, se abortará la ejecución del mismo informando al usuario y mostrándole un mensaje con las indicaciones para hacer un uso correcto del programa.

Dejando de lado la parte de control, el sistema también está dotado de una opción de ayuda, exactamente *-help*, que permite a un usuario desconocedor del programa solicitar indicaciones de uso al propio software.

A continuación, se indica cual es el orden cronológico que se utiliza para hacer cada una de las comprobaciones anteriormente mencionadas:

1. Petición de ayuda por parte del usuario.
2. Falta de todos o algún parámetro de entrada.
3. Verificación de si los parámetros de control introducidos son correctos.
4. Verificación de si los directorios de entrada son correctos.

Por último, además de lo ya mencionado, decir que el programa también realiza comprobaciones intermedias durante su ejecución con el único objetivo de detectar posibles focos de errores posteriores que aborten la ejecución del programa.

4 EVALUACIÓN DE LA PROPUESTA

4.1 Puesta a punto del entorno

En esta sección se procederá a la explicación de la puesta a punto del entorno, o lo que es lo mismo, cuales son los pasos a seguir para poder beneficiarse de la utilidad de nuestra propuesta.

Primero de todo, es necesario saber para qué plataformas ha sido implementada la propuesta. Nuestro software a día de hoy está programado para Windows, por lo que aquellos usuarios de otros sistemas no podrán hacer un uso correcto del mismo.

Seguidamente se procederá a mostrar la estructura de carpetas de dicha propuesta. Esta es muy simple, consta únicamente de dos directorios. El primero de ellos es el directorio raíz en el cual está situado el ejecutable que es capaz de ofrecer nuestra funcionalidad. El segundo es aquel donde residen todas las muestras de malware tanto de test como de entrenamiento para el correcto funcionamiento del software.

A continuación, se muestra una imagen de lo anteriormente mencionado:

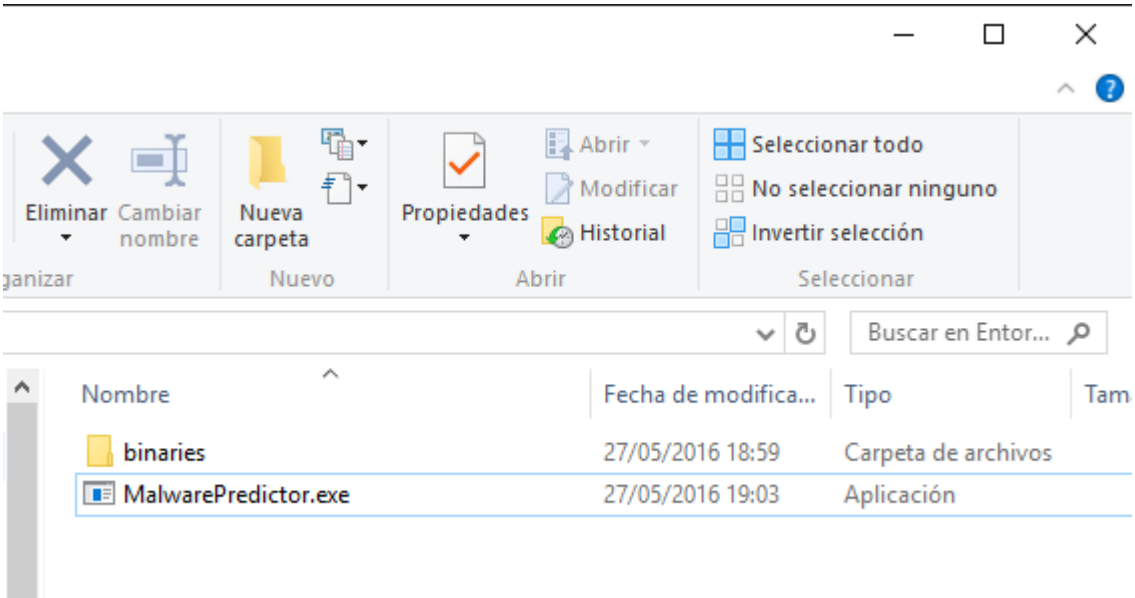


Fig. 26. Directorios del sistema.

Por último, algo que no es necesario para su ejecución, pero de lo que creemos que es importante hablar es del entorno de programación utilizado. Para el proyecto hemos utilizado el IDE de Microsoft Visual Studio 2013, un entorno de desarrollo que cuenta con un sinfín de funciones, herramientas y complementos, totalmente apto para la programación en C++.

Seguidamente, mostramos una imagen donde se puede ver la interfaz principal de este con parte del código “main” de nuestro sistema.

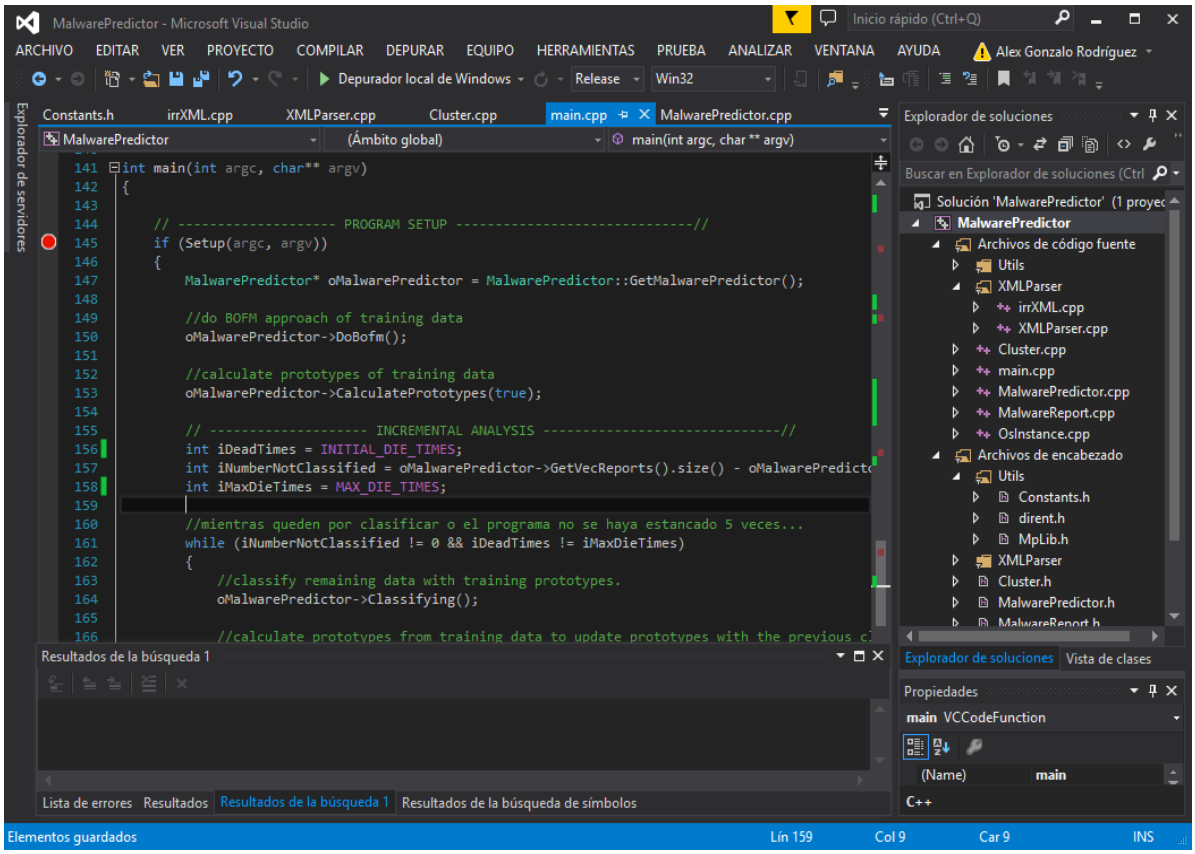


Fig. 27. Interfaz gráfica del IDE Visual Studio 2013 de Microsoft.

4.2 Uso y salidas del programa

El uso del programa es bastante sencillo. En esta misma sección se realizará una explicación detallada de las diferentes funcionalidades que ofrece el sistema, así como de sus diferentes salidas.

Primeramente, empezaremos por la ejecución. Tal y como se menciona en la sección anterior, en el directorio raíz se dispone del ejecutable del programa, así que para ejecutarlo debe hacerse lo siguiente:

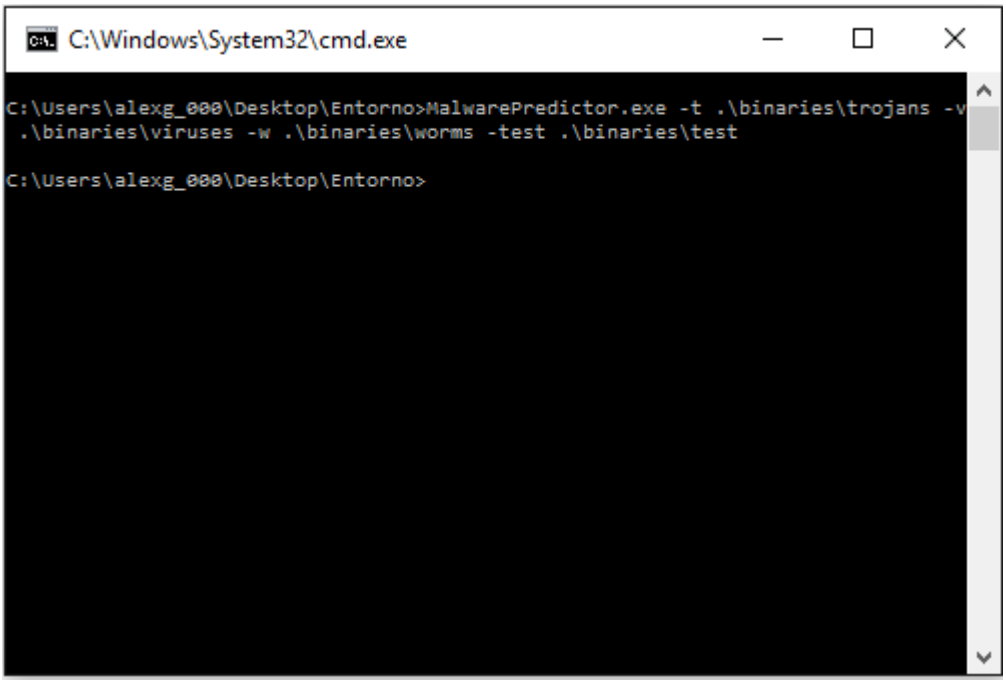


Fig. 28. Ejecución del sistema mediante terminal.

Como se puede apreciar en la imagen, basta con introducir la instrucción:

MalwarePredictor.exe -t .\binaries\trojans -v .\binaries\viruses -w .\binaries\worms -test .\binaries\test

Donde el primer argumento es el ejecutable, y el resto de parámetros tal y como siguen a continuación:

- -t → indica el tipo de muestras del directorio que le sigue, en este caso troyanos.
- .\binaries\trojans → directorio donde se ubican las muestras de troyanos.
- -v → indica tipo de muestras de virus.
- .\binaries\viruses → directorio donde se ubican las muestras de virus.
- -w → indica tipo de muestras de gusano.
- .\binaries\worms → directorio donde se ubican las muestras de gusanos.
- -test → indica las muestras de test.
- .\binaries\test → directorio donde se ubican las muestras de test.

En caso de ser la primera vez que ejecutamos el programa y por tanto no sabemos cómo hacerlo, basta con pedirle al sistema un poco de ayuda con la opción -help.

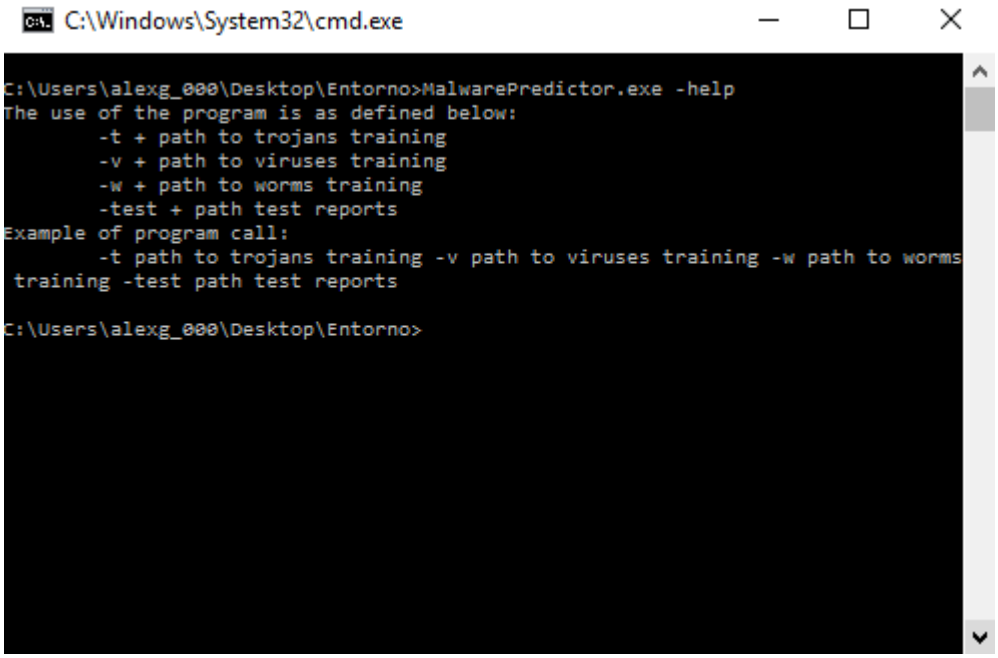


Fig. 29. Uso de la opción de ayuda -help.

Si por algún motivo, llegamos a incumplir alguna de las premisas de ejecución, el sistema te informará de sus condiciones de uso además de mostrarte un ejemplo de llamada. A continuación, se muestran los diferentes casos posibles:

- Sin parámetros de entrada:

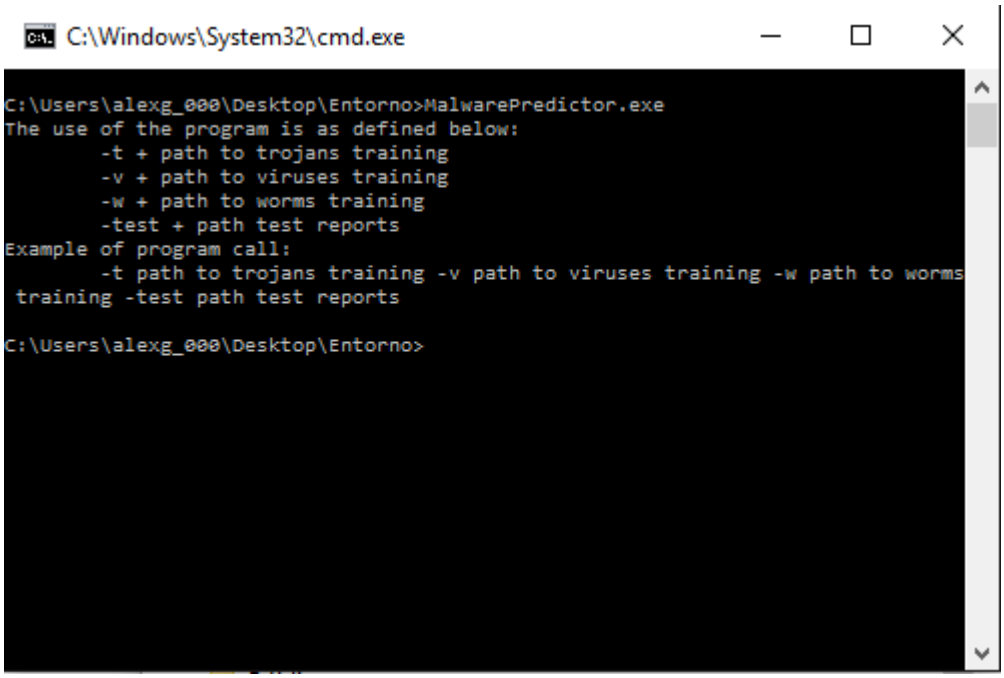


Fig. 30. Ejecución del sistema sin parámetros de entrada.

- Olvido de alguno de los parámetros (en este caso el directorio de “worms”)

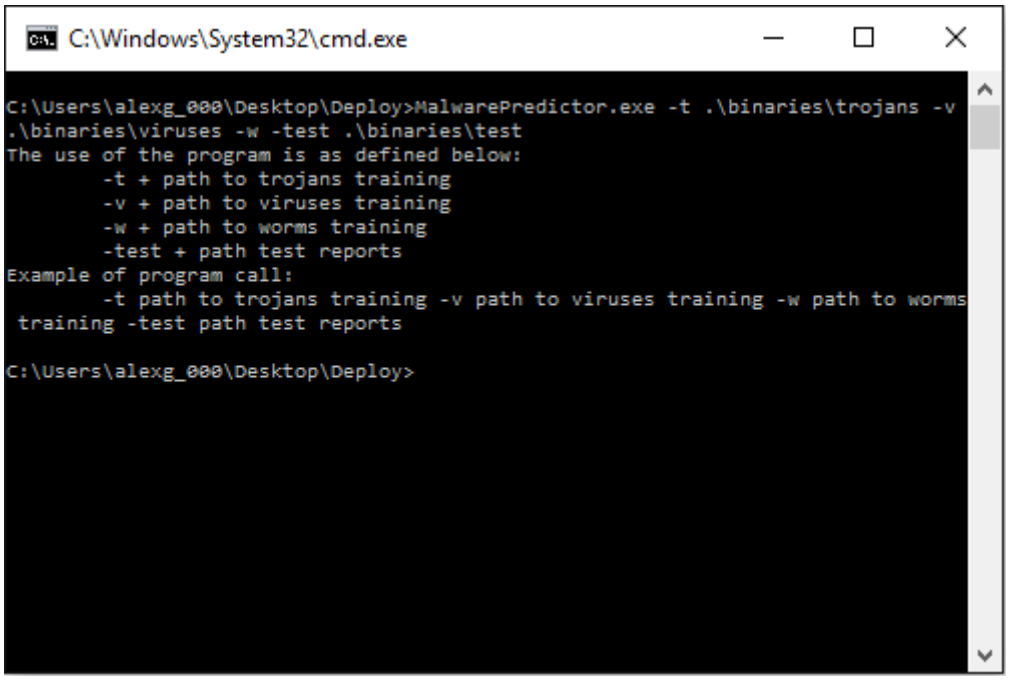


Fig. 31. Ejecución del sistema sin todos los parámetros necesarios.

- Indicación de alguna opción no soportada, por ejemplo -p:

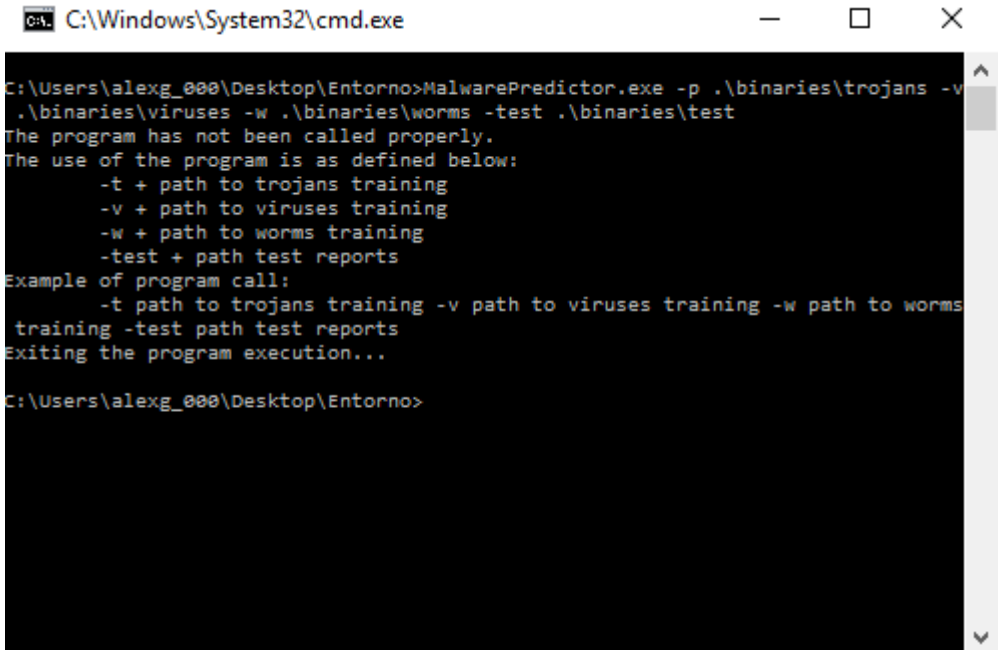


Fig. 32. Ejecución del sistema utilizando una opción no soportada (ej. -p).

Por otro lado, si lo que es incorrecto es el directorio a alguna de las carpetas donde están ubicadas las muestras, el sistema te informará de ello de la siguiente forma:

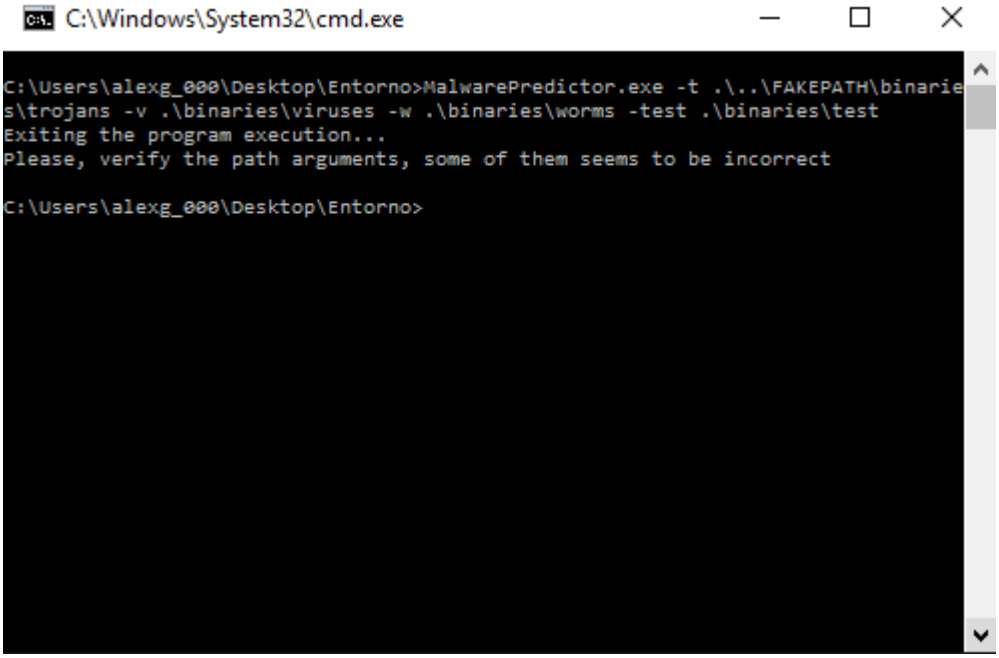


Fig. 33. Ejecución del sistema indicando un directorio erróneo.

Además, otro caso de uso que podría llegar a darse por error del usuario, es que algunas de las opciones (-t ,-v, -w, -test) las indicase en mayúsculas. Pues bien, el sistema también lo controla:

Como resultado de la ejecución, el software creará dos ficheros llamados *Classification_Output.txt* y *Damage_Report.txt* para los resultados de clasificación y peligrosidad respectivamente.

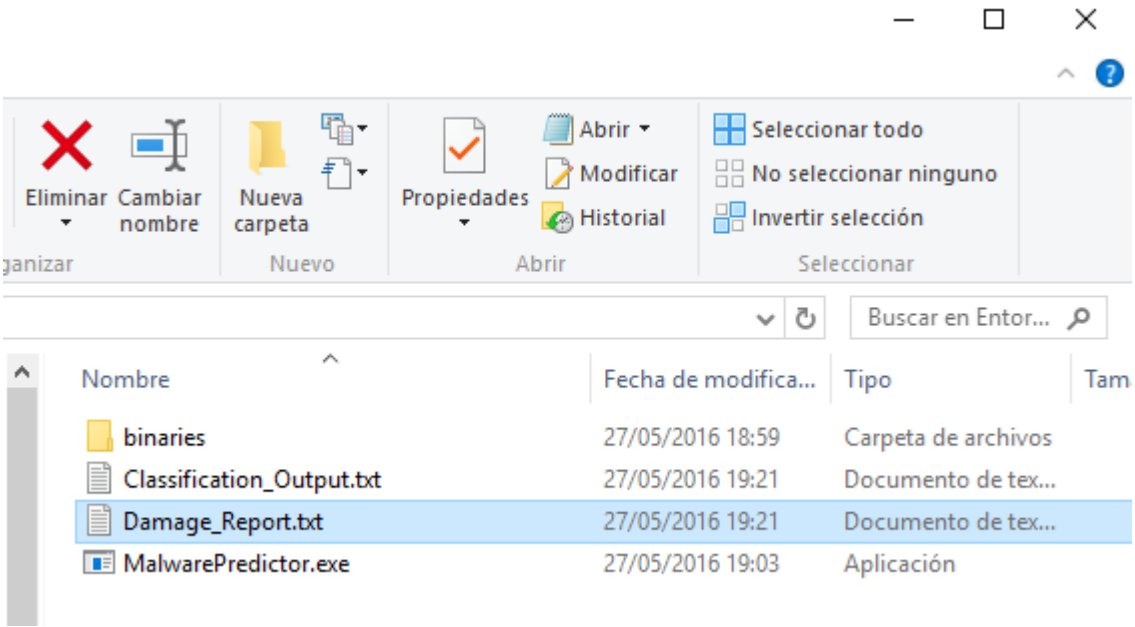


Fig. 34. Ficheros de texto generados como salida del sistema.

En el primero de ellos se nos mostrarán un total de dos columnas, en la primera de ellas se nos indicará los diferentes tipos de malware clasificados y en la segunda la clase asignada por el sistema a cada uno de ellos.

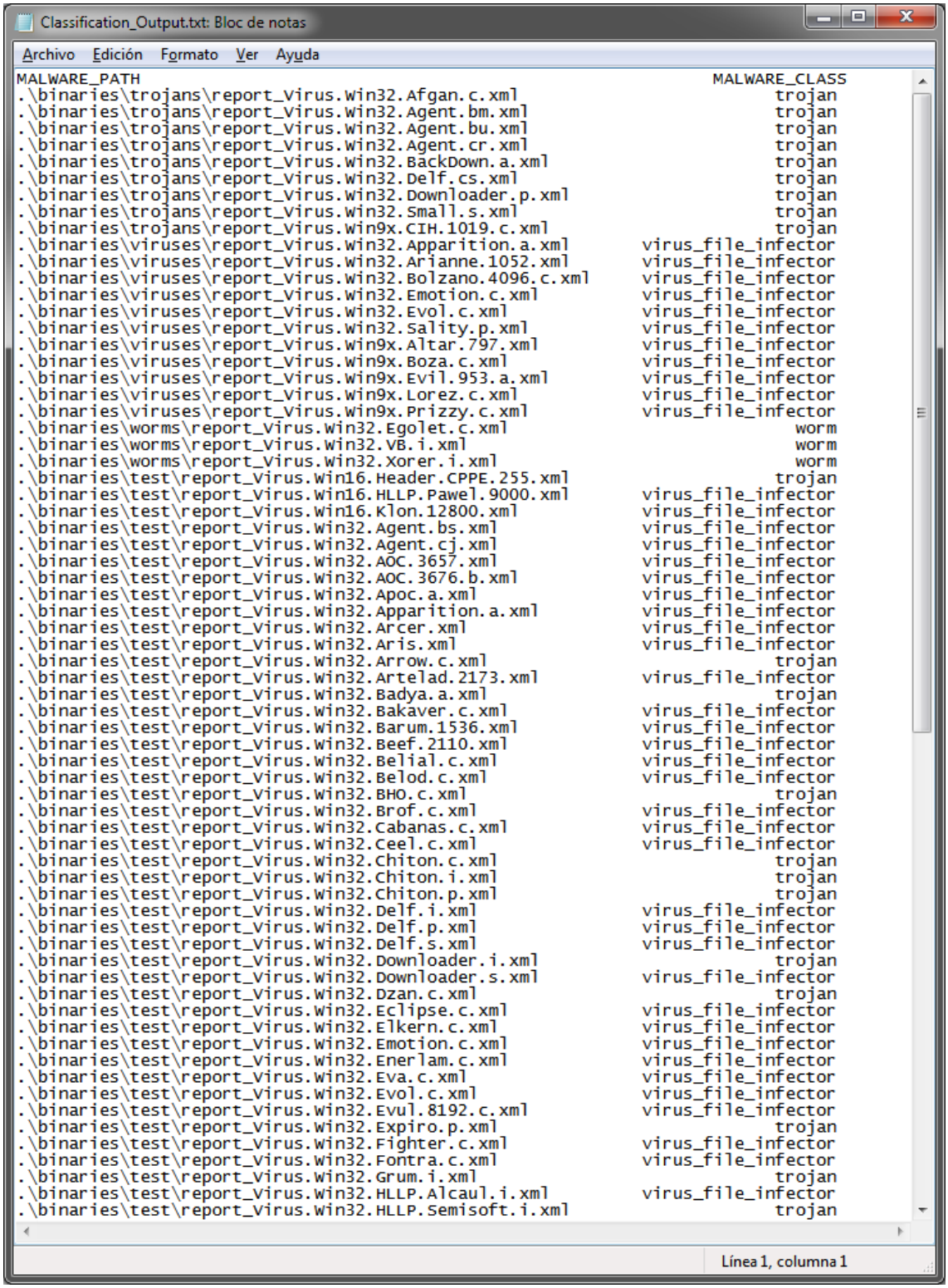


Fig. 35. Salida del módulo de clasificación.

Del mismo modo, en el segundo tendremos dos columnas. En la primera cada uno de los malware analizados ordenados de mayor peligrosidad a menor en orden descendente junto con la etiqueta de peligrosidad, y en la segunda, el nivel de amenaza de cada uno. A continuación, se puede ver una imagen que lo ilustra:

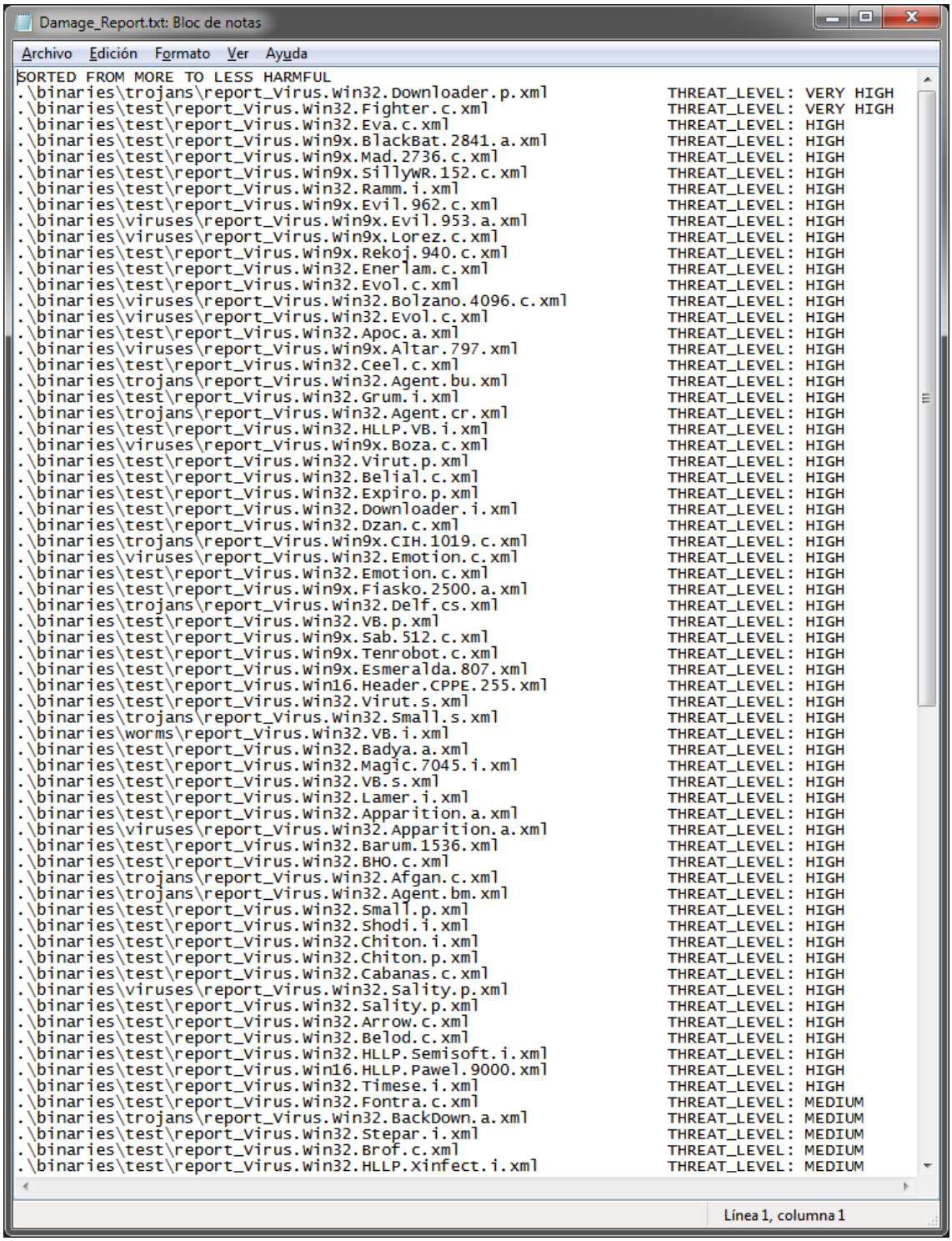


Fig.36. Salida del módulo de peligrosidad.

4.3 Métricas

Para evaluar y calibrar los componentes del primer módulo de nuestra propuesta: extracción de prototipos, Clustering y clasificación, haremos uso de tres métricas conocidas como son *precisión*, *recall* y *F-measure.*

Fig. 37 Vínculo del concepto de métricas con el resto de variables.

Por un lado, la *precisión* nos dará idea de cómo de efectivo es nuestro software. Aplicándolo a nuestro proyecto, cuántas muestras de todas las contenidas en el grupo de test inicial, ha sido capaz de clasificar nuestra propuesta de forma acertada.

Por otro, el *recall* nos será útil para saber cuánto es capaz de abarcar nuestro sistema, o lo que es lo mismo, a cuántas de las muestras de test ha sido capaz de asignarle una etiqueta de clasificación y, por ende, cuántas han sido clasificadas.

Definimos cada una de ellas de la siguiente forma:

$$P = \frac{Muestras\ correctamente\ clasificadas}{Muestras\ totales\ de\ test}$$

Ecuación 1. Métrica de precisión.

$$R = \frac{Muestras\ con\ etiqueta}{Muestras\ totales\ de\ test}$$

Ecuación 2. Métrica de recall.

Una vez tenemos *precisión* y *recall,* se procederá a realizar el cálculo de la conocida métrica *F-Measur*e, que nos servirá como puntuación de rendimiento final. Ésta viene definida por la siguiente ecuación:

$$F = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Ecuación 3. Métrica de F-Measure.

Cabe mencionar que estas métricas no se aplican al módulo de peligrosidad porque tal y como se ha planteado la realización de éste, no es necesario emplearlas.

4.4 Resultados del módulo de clasificación

Antes de comenzar la explicación, como recordatorio:

Por un lado, teníamos *Dr*, constante que indica cómo de tolerante es el sistema a la hora de asignar un prototipo a una determinada clase, es decir, cuál es la distancia mínima entre muestras para llevarse a cabo dicha clasificación.

Por otro lado, estaba *Dc*, cuyo funcionamiento es similar a Dr, pero realiza la función de limitador e impone la distancia mínima necesaria entre cada par de clústeres del conjunto, para que se pueda llevar a cabo la unión de éstos.

En tercer lugar, aparece *Dp*,que intenta representar el nivel de homogeneidad con la que los prototipos extraídos han de representar al conjunto de muestras de test. Como consecuencia se seguirán extrayendo prototipos hasta cumplir este requisito.

Por último, está la constante *m*, la cual se encarga de indicar la tolerancia con los clústeres extraídos. Cuanto más pequeño sea *m* menos tolerante se será con muestras infrecuentes y viceversa.
A continuación, se muestran las diferentes configuraciones que llevaremos a cabo, junto con la óptima al final, teniendo la constante *m* puesta a 2. La *m* no la cambiamos porque ya tiene asignado el valor más adecuado al número de muestras malware de las que disponemos.

Dr	***Dc***	***Dp***
1	**1**	**1**
1	**1**	**2**
1	**1**	**3**
1	**2**	**1**
1	**2**	**2**
1	**2**	**3**
1	**3**	**1**
1	**3**	**2**
1	**3**	**3**
2	**1**	**1**
2	**1**	**2**
2	**1**	**3**
2	**2**	**1**
2	**2**	**2**
2	**2**	**3**
2	**3**	**1**
2	**3**	**2**

2	**3**	**3**
3	**1**	**1**
3	**1**	**2**
3	**1**	**3**
3	**2**	**1**
3	**2**	**2**
3	**2**	**3**
3	**3**	**3**
4	**3**	**2**

Tab. 5. Combinaciones de constantes utilizadas para el módulo de clasificación.

De las configuraciones ya expuestas justo encima, la que nos proporciona un resultado más favorable teniendo en cuenta las métricas de precisión y recall es la última de ellas, es decir, aquella donde:

- Dr → 4
- Dc → 3
- Dp → 2

Obteniendo un recall del 100%, una precisión de ≈ 60% y 75% de F-Measure.

A continuación, se muestran dos gráficas con las distintas métricas utilizadas para cada una de las configuraciones utilizadas anteriormente.

La primera de ellas es la de precisión y recall. Los aspectos más destacables son el hecho de que la constante Dp, cuanto más alta más penaliza el resultado del programa si la combinación de las otras dos no compensa este factor. Esto es debido a que cuando mayor es Dp, el número de prototipos extraídos es menor. Además, si se da que Dc también es baja, será difícil que se unan clústeres dando lugar a un número mayor de estos con un número menor de elementos y con mayor probabilidad a ser descartados por la constante *m*.

Con todo, es necesario remarcar que, dado que el número de muestras empleadas para la evaluación del programa no ha sido él deseado desde un inicio, no se puede extraer un veredicto del todo concluyente, y por ello, la correlación entre las constantes y los datos es un tanto abstracto y no queda del todo definido aun teniendo en cuenta y sabiendo la finalidad de cada constante (ubicando su función en el programa; qué determina, cómo afecta a los resultados, etc.)

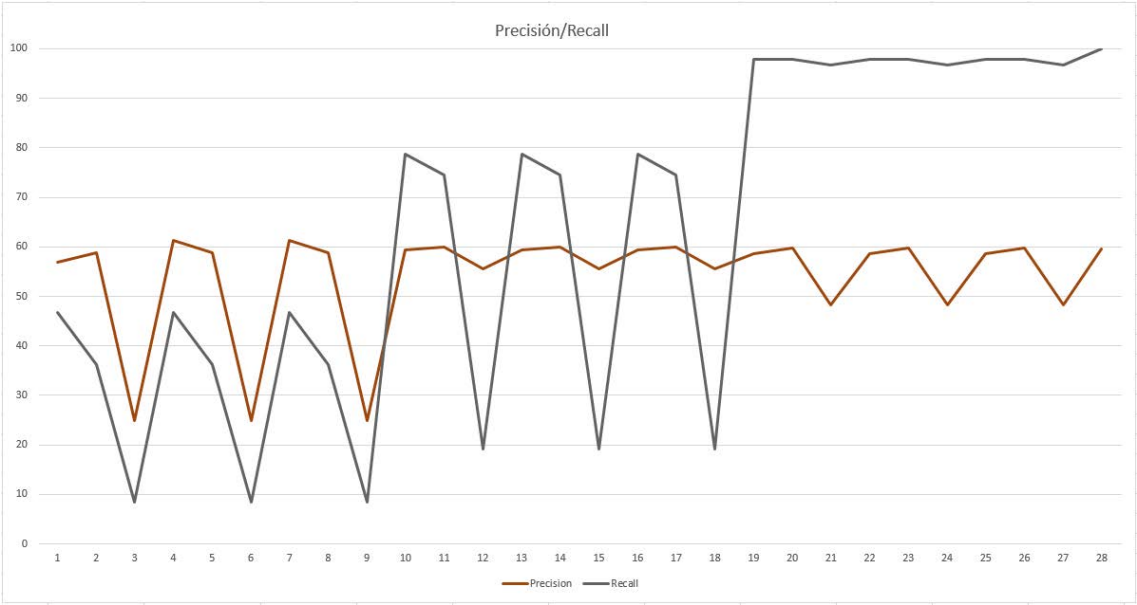


Fig. 38. Resultados de precisión y recall de las diferentes configuraciones.

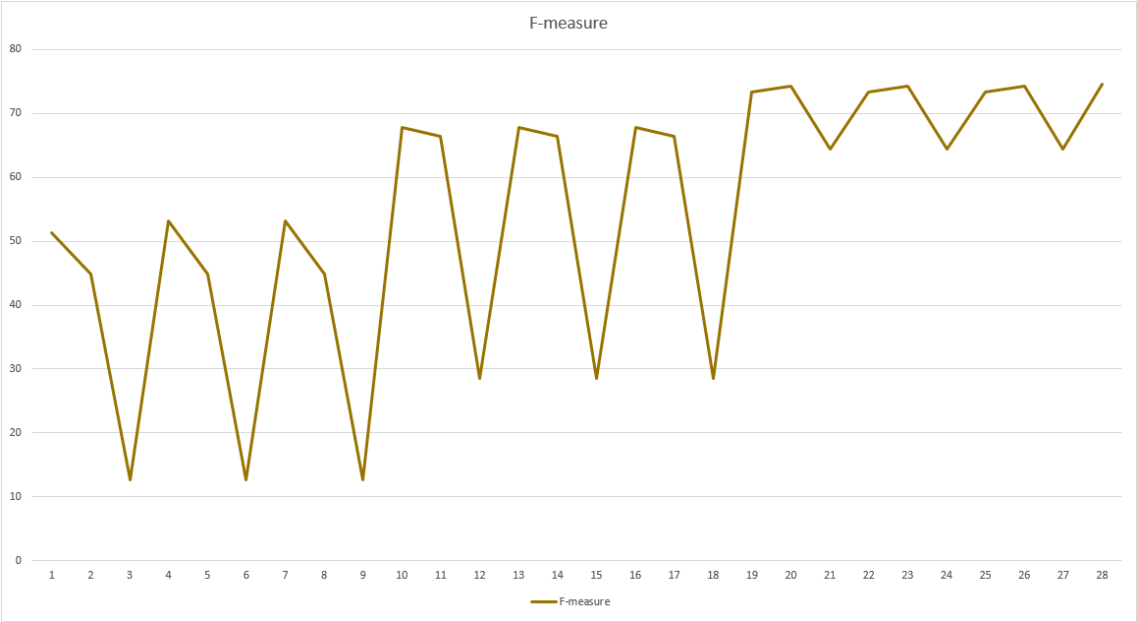


Fig.39. Resultados de F-Measure de las diferentes configuraciones.

4.5 Resultados del módulo de predicción de daños malware

Pasamos ahora a comentar la segunda parte de los resultados de nuestro trabajo, la cual difiere de la clasificación y se centra únicamente en determinar cómo de dañino puede ser un determinado malware.

Basándonos en la estimación de daños ya explicada, y en la asignación de valores a las diferentes acciones que puede llevar el malware sobre el sistema operativo, los resultados que obtenemos son bastante certeros, puesto que tanto el orden como la cuantificación del daño para cada muestra se acerca mucho a la realidad.

Gracias a la estructura de puntuación aditiva, y a la conversión de los resultados sobre base 10, la asignación de las etiquetas de peligrosidad se realiza con correctitud basándonos en las fuentes de confianza. [8] [9] [10].

Concretando más, y con el objetivo de afianzar y corroborar un poco más los resultados obtenidos, se procederá a comentar el top tres ordenados de mayor a menor peligrosidad respectivamente:

1. **Win32.Downloader.p.xml → Tamaño: 439 KB**
2. **Win32.Fighter.c.xml → Tamaño: 237 KB**
3. **Win32.Eva.c.xml → Tamaño: 150 KB**

El top 3 está conformado por dos troyanos y un virus altamente dañino, de tamaño bastante grande:

El primero de ellos es la variante .p de una serie de troyanos bastante conocidos, los Downloader, que actúa de "backdoor" para nuevos malware, descargándolos desde servidores remotos y ejecutándolos sin autorización del usuario infectado; de ahí el apodo. Tiene un tamaño considerablemente grande, junto con un conjunto de acciones que en su mayoría tienen asociadas una alta puntuación en lo que a riesgo refiere. Así mismo, es uno de los troyanos más molestos y difíciles de eliminar, por lo que tiene merecido estar entre los primeros. Se puede encontrar más detalles al respecto en [11]

El segundo, es una variante de troyano similar al primero, pero que en vez de descargar nuevos ejecutables, inyecta archivos .dll a diferentes procesos activos del sistema operativo, de entre los cuales encontramos explorer.exe y scvhost.exe de Windows. Provoca la apertura y ejecución de nuevos procesos indeseados que, entre otras cosas, saturan el sistema operativo y dan paso a más amenazas [12]. Además, al igual que el Downloader, es muy difícil de eliminar.

El tercero y último, consiste en una variante .c de la cepa del Win32.Eva, que al igual que el resto de variantes, es altamente dañina. Aunque no hemos logrado conseguir gran información al respecto, se cree que es un virus que parasita en la memoria, causando grandes estragos en ésta. Fuentes como Microsoft confirman su alta peligrosidad [13].

5 FUTURAS MEJORAS A IMPLEMENTAR

En esta sección, lo que pretendemos es indicar o sugerir algunas posibles mejoras que se podrían implementar en el sistema de cara al futuro.

La primera de ellas está relacionada con el algoritmo de extracción de prototipos. Éste está implementado de tal forma que, por la propia construcción del algoritmo, y con esto nos referimos al hecho de que al inicio todas las distancias entre prototipos se inicializan a infinito, el primer prototipo seleccionado es siempre el mismo.

Debido a ello, durante la fase de testeo de la propuesta nos percatamos de que si generamos una distribución de muestras en la cual pocas de ellas son de test, el algoritmo no obtiene un buen resultado en cuanto a clasificación. Este hecho es debido al punto anteriormente explicado. El algoritmo parte de una muestra inicial, que es la primera de todo el conjunto en ser prototipo, y a partir de ahí continúa seleccionando al resto de ellos. Este funcionamiento, en caso de tener la distribución de muestras anterior, calcularía prototipos en un inicio donde estos no cambiarán prácticamente en iteraciones posteriores.

No cambiarán porque la realimentación que sufrirá el algoritmo por parte de las muestras clasificadas será pequeña, debido a las pocas muestras de test de las que se dispone. Por tanto, cuando el algoritmo quiera volver a calcular prototipos, además de que éste iniciará el cálculo por la misma muestra que en casos anteriores, tampoco dispondrá de muchas nuevas muestras que puedan cambiar la distribución de prototipos seleccionados, imposibilitando al mismo la clasificación de posteriores muestras.

Además, en caso de que la extracción sea sobre muestras de entrenamiento, dado que están vinculadas desde un inicio a una clase determinada de malware, se está condicionando el algoritmo a determinadas clases.

Como contramedida a esta dificultad, lo que proponemos es mantener el mismo proceso de extracción de prototipos, pero con un ligero cambio: Realizar de forma aleatoria el proceso de selección de la primera muestra. De esta forma al final del proceso se conseguirá una distribución de prototipos totalmente distinta a la calculada en iteraciones anteriores, y probablemente facilitará al sistema la clasificación de muestras en posteriores iteraciones.
Como consecuencia de la aplicación de esta contramedida, también evitaremos que el sistema dependa del orden de entrada de las muestras a la hora del cómputo del resultado final.

Fig. 40. Representación del concepto de mejora continua.

La segunda propuesta es la de llevar nuestro software a un público no especializado. Para ello sería necesaria la inclusión de una interfaz gráfica de usuario que permitiese al mismo hacer uso del programa de una manera más cómoda y sobretodo más sencilla.

Otra de ellas sería la opción de calibrar el programa desde el exterior, o lo que es lo mismo, no tener la necesidad imperiosa de modificar el código de la propuesta para modificar las constantes que regulan los diferentes algoritmos. Con esto evitaríamos recompilaciones futuras a cada modificación, así como tiempo a la hora de la calibración.

Finalmente, como última mejora y alternativa a lo ya existente, se podría aplicar la normalización a los vectores de comportamiento, con el objetivo de que estos tuviesen módulo unitario. Ello facilitaría la calibración del sistema, ya que el rango de valores que se podría obtener durante el cálculo de distancias euclidianas sería de entre cero y uno, permitiendo al operador haciendo uso del mismo, hacer una mejor previsión de los efectos sobre la clasificación del delta aplicado a cada cambio de constantes.

6 DIFICULTADES DEL PROYECTO

Pasamos a explicar cuáles son las dificultades que han surgido durante el proyecto.

Todas ellas han ido surgiendo durante la ejecución del mismo y se podría decir que han sido más de gestión que de implementación. Cabe decir que, pese a ser dificultades, han sido encajadas como un punto positivo sobre el cual iniciar un proceso de mejora.

La primera de ellas se remonta a los inicios del proyecto, donde surgieron problemas con la búsqueda de la plataforma de análisis de malware que posteriormente utilizaríamos. Tras indagar por la web en busca de diferentes propuestas, finalmente nos decantamos por Anubis por dos factores: su riqueza en cuanto a información y detalles en los informes malware, y su versatilidad en lo que a formatos de documento se refiere.

La segunda dificultad, relacionada con esta primera, surgió al hacer uso de la plataforma en cuestión. Tras subir los diferentes ejemplos de malware a ésta, Anubis no iniciaba el análisis de los mismos, indicando un tiempo de espera desorbitado y en según qué casos siendo incapaz de procesarlos. Buscando por la web, así como visitando diferentes foros de opinión, descubrimos finalmente que este problema de inestabilidad era habitual en Anubis.

La causa del mismo no la llegamos a saber a ciencia cierta, pero todo apuntaba a que era la sobrecarga de trabajo a la que Anubis se veía sometido por los usuarios. Por tanto, nos vimos obligados a realizar intentos sucesivos en diferentes horas del día con el objetivo de obtener análisis de las diferentes muestras.

Fig 41. Concepto de resolución de problemas en equipo.

Relacionada también con Anubis, otra de las dificultades más recientes que tuvimos, a principios de abril de 2016, fue su cierre. La plataforma online que hasta la fecha nos proporcionaba los informes malware, fue cerrada por los propios autores debido a una falta de financiación para sostenerla y por una fusión posterior de los mismos con otro equipo dedicado también al análisis del malware.

En la planificación del proyecto incluimos en su momento un período de tiempo tras el desarrollo del programa principal pensado para aumentar el número de muestras. El objetivo de este proceso era poder probar el clasificador con mayor volumen de malware, y de esta forma, poder extraer unos resultados más concluyentes y precisos. No obstante, debido al cierre de Anubis cesó el poder generar más informes de malware, y por ende, tuvimos que abortar esta fase de aumento de muestras.

Como consecuencia de lo anteriormente mencionado, el número máximo de muestras que pudimos llegar a obtener fue de 111. El valor obtenido, tal y como se puede ver, no es pequeño, pero tampoco es adecuado para testear la eficacia de un programa basado en el aprendizaje automático como el nuestro. Ello lleva a que el valor de las constantes definidas: Dr, Dc y Dp estén vinculadas fuertemente al número de muestras que tenemos, y que sea necesario calibrarlas cada vez que se cambian la distribución de estas muestras.
Las causas de esto fueron totalmente ajenas a nuestra persona, sin posibilidad de actuación, por la inexistencia de más analizadores con el formato de Anubis, y por el avanzado estado del proyecto en el que nos encontrábamos.

A todas estas dificultades, se suman otras ajenas a Anubis o a sus consecuencias, que así mismo, es importante tener en cuenta.

En primer lugar, tenemos el comportamiento del malware y la división de éste en sus diferentes variantes. Toda clase de fichero o archivo malicioso que pretenda provocar daños, puede representarse como malware. No obstante, la diversidad y complejidad del conjunto de malware hoy en día es tan grande, que una misma tipología puede dividirse en una infinitud de subconjuntos con comportamiento similar.

Debido a ello, en la actualidad es muy difícil encontrar información exacta y de confianza sobre la categoría a la que pertenece cada muestra del conjunto malware elegido para las pruebas. Si a ello le sumamos que ciertos malware pueden pertenecer a varias categorías o subcategorías por similitudes en el comportamiento, hace que realizar una clasificación a priori de según qué muestras particulares, ya sea para incluirlas como datos de entrenamiento, o para evaluar la fiabilidad y precisión del programa, sea una labor complicada.

Así pues, nos ubicamos en una escena en la cual los informes de malware que previamente hemos clasificado, quizás no pertenecen realmente a la clase que les hemos asignado, o pertenecen a múltiples. Aún con todo, gracias a las enciclopedias online, creemos que se ha asignado la clase correcta a prácticamente la totalidad de las muestras.

En particular nos fueron de gran ayuda las enciclopedias de Symantec [8], Trend Micro [9] y de Microsoft [10], siendo la primera de éstas la más explicativa, la segunda la más exhaustiva (se pueden encontrar muestras de todo tipo) y la tercera la más minimalista. Así pues, gracias a las referencias de estas páginas y realizando comparativas coherentes, fuimos capaces de clasificar en gran medida nuestro conjunto malware. Sin embargo, tal y como hemos comentado al principio, es difícil clasificar el malware en categorías aun cuando se cuenta con información de confianza debido a la gran diversidad de muestras, y por ello, nada garantiza la fiabilidad al 100% de la clasificación de éstas.

A raíz de todo lo ya mencionado, nuestra última dificultad radica en la complejidad patente que existe a la hora de evaluar nuestro sistema. Sabemos que el código desarrollado, por la

lógica y estructura de los algoritmos en los que se fundamenta, así como por toda la depuración y pruebas que se han ido haciendo durante el transcurso del proyecto, es estable. Sin embargo, pese a que cumple con su propósito como clasificador malware al 100%, si se le hubiese podido proveer de un mejor contexto de entrenamiento, probablemente los resultados obtenidos hubieran sido mejores.

Por último, pese a las dificultades que hemos tenido que sobrepasar durante el transcurso del proyecto, creemos que se han cumplido las expectativas iniciales del mismo.

7 DECISIONES TÉCNICAS DE IMPLEMENTACIÓN Y DISEÑO

Durante la elaboración del proyecto, se llevaron a cabo una serie de decisiones que en mayor o menor medida afectaron al mismo. A continuación, se mencionan cada una de ellas:

La primera, consiste en el uso del lenguaje de programación C++. Siempre una de las prioridades ha sido la búsqueda de eficiencia, y C++ era y es el lenguaje más adecuado para ello.

Fig. 42. Logotipo del lenguaje de programación C++.

Hoy en día es el lenguaje más utilizado en el desarrollo de videojuegos, principalmente por su rapidez de ejecución, debido a que este se ejecuta directamente sobre la CPU a diferencia de otros lenguajes como Java con su máquina virtual o Python con su intérprete.

Además de este principal motivo, también hay otros que motivaron su elección:

El primero y más importante fue la experiencia que se tenía haciendo uso de este lenguaje seguido de las facilidades que C++ te ofrece con librerías estándares como STL que son más que útiles para desarrollos de este tipo de software. Por último, y menos importante, un factor que condicionó su elección, fue la comodidad que sentía al hacer uso del IDE Visual Studio de Microsoft.

Desde un inicio se quería que el código fuese de calidad, mantenible y eficiente por lo que el diseño fue uno de los puntos principales en su desarrollo.

Por ello, se emplearon patrones de diseño, más concretamente del patrón Singleton. Este fue utilizado para aquellas clases que tenía sentido que únicamente hubiese una única instancia en todo el programa. Este concepto facilitaba que estas clases fuesen utilizadas desde cualquier punto donde su funcionalidad fuese requerida proporcionando un acceso global a las mismas.

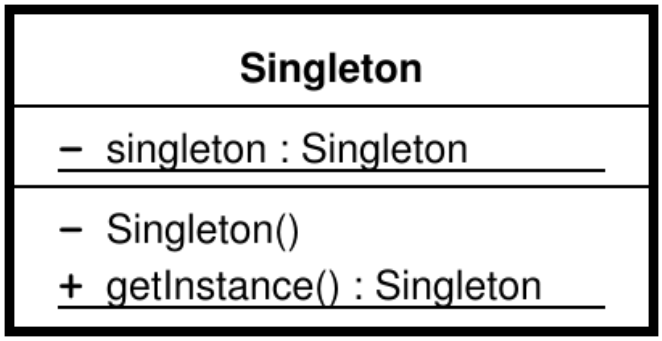


Fig. 43. Estructura de clase del patrón de programación Singleton.

Además, se puso mucho énfasis en el concepto de modularización y encapsulación o abstracción.

El primero de ellos lo que nos aporta es evitar la redundancia del código, es decir, aquellas tareas utilizadas repetidas veces en determinados puntos del programa podían ser invocadas tantas veces como fuese necesario.
Además, también aporta una ventaja clara y es la estructuración del proyecto, dejando constancia clara de los diferentes módulos que componen el sistema.

Por otro lado, el segundo término lo que nos aporta es un seguro de integridad sobre los datos manipulados durante la ejecución del programa, donde los datos de cada instancia únicamente pueden ser manipulados mediante métodos controlados para ello.

La última de las decisiones y que ya ha sido mencionada brevemente al inicio de esta sección, es el uso de la librería STL que provee C++. Esta provee al programador de una serie de estructuras genéricas y eficientes que se adaptan perfectamente a las condiciones de nuestro proyecto, en el cual la manipulación de vectores está más que garantizada.

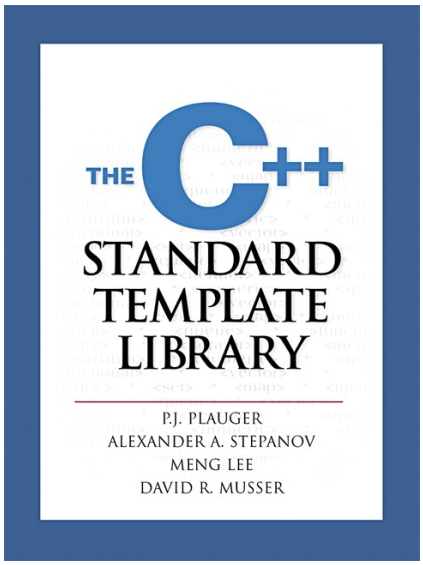


Fig. 44. Logotipo de la librería STL para C++.

8 COSTES DEL PROYECTO

A continuación, procederemos a detallar cuáles han sido los costes involucrados en el desarrollo de todo el proyecto de principio a fin.

Para realizarlo de una manera intuitiva y fácil de comprender haremos distinción entre dos tipos de costes, fijos y variables.

El primer término aplicado a este proyecto hace referencia a aquellos costes que su cantidad no varían durante el transcurso del mismo.

En nuestro caso, como concepto de costes fijos tenemos el equipo informático utilizado que asciende a un total de 2.100€. Además, no hay que olvidar la mano de obra directa, es decir, la retribución mensual de la persona o personas involucradas en el proyecto.

Para este último caso, es conveniente hacer una tabla donde veamos el desglose de horas empleadas:

Concepto	**Horas**
Documentación e investigación	50
Recogida de muestras y subida a Anubis	20
Búsqueda de framework externo de procesado de XML's	2
Creación y puesta a punto del proyecto	3
Desarrollo del módulo de clasificación	
● Procesado de XML's	15
● Implementación del algoritmo de BOFM	35
● Implementación del algoritmo de extracción de prototipos	40
● Implementación del algoritmo de Clustering	60
● Implementación del algoritmo de clasificación	50
● Inclusión de mejoras mencionadas.	45
Desarrollo del módulo de peligrosidad	30
Arreglo de bugs (incluye depuración del	60

programa)	
Gestión y control de errores	15
Elaboración y revisión de la documentación final	80

Tab. 6. Desglose de horas del proyecto.

Como resultado de cada una de las tareas anteriormente mencionadas, obtenemos un total de 505 horas. Ahora, solo nos queda saber el coste total de estas horas, para ello seguiremos el proceso abajo descrito:

- Salario bruto anual: 22.000€
- Salario neto anual:
  - 4,70% seguridad social por parte del trabajador.
  - 13% de IRPF.
  - $22.000 - (22.000 \cdot 0{,}177) = 18.106€$
- Horas anuales:
  - 240 días laborables.
  - 2 horas diarias aproximadamente.
- Coste por hora: $\frac{18.106}{480} = 37{,}72€$

Coste total por mano de obra: $37{,}72 \cdot 505 = 19.050€$

Por otro lado, tenemos los costes variables, que estos a diferencia de los costes fijos, sí que pueden variar durante el transcurso y el nivel de actividad empleado en el proyecto.

Concepto	**Precio/Unidad (€)**
Dietas	995
Transporte	50
Gastos de habitabilidad (luz, agua, gas, etc.)	100

Tab. 7. Desglose de costes variables del proyecto.

Por último, ya podemos calcular el coste total del proyecto sumando cada uno de los costes individuales:

Coste total: $2100 + 19.050 + 1.145 = 22.295€$

9 CONCLUSIONES

El software malicioso, conocido más comúnmente como malware, es una de las mayores amenazas de internet hoy en día. Prácticamente, la gran mayoría de ataques cibernéticos emplean este tipo de software para llevar a cabo sus objetivos y su proliferación va en aumento. Además, sus creadores, de entre otros, *hackers*, hacen uso de técnicas para vulnerar los análisis a los cuales son sometidos por software especializado, como por ejemplo Antivirus.

Todo este cúmulo de factores hacen necesaria la búsqueda de vías alternativas al análisis estático, como el análisis dinámico, caracterizado por hacer uso de técnicas de aprendizaje automático.

Por ello, nosotros proponemos un análisis dinámico confiando en el *Clustering* como técnica de aprendizaje automático, pero combinándolo con el *Bounded Space Feature behaviour Modeling*, con el objetivo de obtener un incremento de eficiencia en tiempo y espacio.

Así mismo, ofrecemos también un sistema de cuantificación de peligrosidad capaz de reconocer que muestras malware son más ofensivas. Este último punto es de utilidad tras la detección del malware, permitiendo dar una orientación del orden de prioridad a la hora de aplicar contramedidas.

En base a los resultados obtenidos, nuestra propuesta ofrece un entorno de análisis dinámico de malware para combatir la cantidad ingente de malware que circula por las redes.

10 GLOSARIO: CONCEPTOS GENÉRICOS

- **Análisis incremental**: Análisis continuo donde en cada iteración refina su capacidad de reconocimiento y extracción de información.

- **Antivirus**: Programa que analiza el sistema operativo de un usuario en busca de malware para tratarlos. El tratamiento puede ir desde meter en cuarentena al malware en cuestión, a eliminarlo por completo.

- **Anubis**: Programa web desarrollador por el grupo de iseclab.org que permite la conversión de binarios crudos de malware, en informes que encapsulan toda la información del comportamiento de los mismos. Proporciona información detallada y fácil de leer y procesar.

- **Aprendizaje automático**: Consiste en la autonomía de un sistema por aprender en base a pasados eventos. El proceso requiere de unos datos previos conocidos como datos de entrenamiento, los cuales usa como referencia inicial para poder desarrollar esta capacidad con nuevos conjuntos de datos.

- **BOFM:** Consiste en el algoritmo empleado en nuestro trabajo, el cual permite la encapsulación del comportamiento de las muestras malware en vectores de tamaño fijo.

- **Clasificación**: Algoritmo que se emplea para determinar la clase o tipología de una muestra malware o de un objeto en términos generales.

- **Clustering**: Técnica empleada comúnmente en el aprendizaje automático que radica en la asociación de elementos de comportamiento similar en conjuntos cerrados con el objetivo de unificar a todos como un único elemento.

- **C++**: Lenguaje de alto nivel basado en C, que se ha empleado para el desarrollo del proyecto. Es flexible, eficiente y a diferencia de su predecesor, cuenta con soporte para la creación de objetos.

- **Entorno virtual (sandbox)**: Entorno protegido empleado para realizar las simulaciones del programa. Cualquier ejecución que inmiscuya malware con gran capacidad dañina o que pueda tener graves consecuencias en el sistema operativo, no sale de este entorno. Por lo que es una garantía de seguridad frente a simulaciones que puedan ocasionar daños en el sistema.

- **Espacio vectorial acotado**: Espacio heterogéneo en el que son representados los conjuntos de vectores de las muestras. El que sea acotado significa que crece de forma lineal conforme al número de vectores y que no escala de forma drástica como podría ocurrir con escalas exponenciales o cuadráticas.

- **FCG**: Grafo que representa bajo la estructura de árbol, el conjunto de llamadas a función de un programa o malware en cuestión.

- **Gusano**: Clase de malware conocido por la capacidad de propagarse sin necesidad de hospedarse en un programa. Puede duplicarse en tiempos muy cortos y sobrecargar redes informáticas o PC's.

- **IDE**: Entorno de desarrollo integrado que consiste en un programa con interfaz propia, que facilita al desarrollador la programación proporcionándole herramientas como el "debugger", la creación dinámica de ficheros mediante interfaz, y otros conjuntos herramientas muy útiles.

- **Malware**: Software desarrollado con objetivos maliciosos o dañinos. Existe una gran variedad de tipologías de malware en la actualidad y el volumen sigue en aumento por cada año que pasa.

- **MIST**: Formato que representa una acción determinada de un malware empleando códigos numéricos para cada parte que compone la acción. La estructura es similar a la que se utilizan en instrucciones para procesadores modernos.

- **Muestra/Report/Informe**: Fichero .XML que encapsula la información de las acciones que realiza un malware en cuestión. Se consiguen gracias al programa Anubis, el cual lleva a cabo la conversión a partir del binario crudo del malware.

- **N-GRAM y Q-GRAM**: Técnicas que permiten codificar conjuntos de acciones de las muestras como estructuras formadas por "n" o "q" grams. Los grams surgen a raíz de las combinaciones estadísticas de aparición de una ocurrencia en dicha acción, y es el conjunto de estos grams en tamaño "n" o "q" lo que define el comportamiento final de la muestra.

- **PageRank**: Valor numérico que representa la importancia de una página web en Internet.

- **Prototipo**: Muestra malware cuyas propiedades representan a un conjunto de otras muestras similares por su cercanía a ellas.

- **Troyano**: Tipología de malware que es conocida por sustraer información privada de los objetivos infectados y por realizar una puerta de entrada a otros malware. ("backdoor").

- **Virus**: Malware por definición. También son conocidos como "File-infectors", y la propiedad principal de éstos consiste en que necesitan de un medio para propagarse; de un huésped, ya sea en la memoria, en ejecutables, procesos del sistema, etc. Es el malware más abundante.

- **Visual Studio**: IDE de Microsoft que se ha empleado para el desarrollo de nuestro trabajo. Es uno de los entornos más completos en desarrollo del mundo, y cuenta con un sinfín de herramientas y funcionalidades de gran utilidad para el usuario programador.

- **Windows**: Sistema operativo desarrollador por Microsoft sobre el cual basamos la ejecución del programa de nuestro trabajo.

11 BIBLIOGRAFÍA

[1] Konrad Rieck1, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic Analysis of Malware Behavior using Machine Learning. Published in the Journal of Computer Security, 2011. http://www.mlsec.org/malheur/docs/malheur-jcs.pdf

[2] Anubis Malware Analyzer (25/2/16).
https://anubis.iseclab.org/?action=home

[3] Deguang Kong and Guanhua Yan. Transductive Malware Label Propagation: Find Your Lineage From Your Neighbors.
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6848075

[4] Aziz Mohaisen, Andrew G. West, Allison Mankin and Omar Alrawi. Chatter: Classifying Malware Families Using System Event Ordering.
http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6997496&url=http%3A%2F%2Fieeexplore.ieee.org%2Fxpls%2Fabs_all.jsp%3Farnumber%3D6997496

[5] Mahinthan Chandramohan, Hee Beng Kuan Tan, Lionel C. Briand, Lwin Khin Shar, and Bindu Madhavi Padmanabhuni. A Scalable Approach for Malware Detection through Bounded Feature Space Behavior Modeling.
http://ieeexplore.ieee.org/xpl/login.jsp?tp=&arnumber=6693090&url=http%3A%2F%2Fieeexplore.ieee.org%2Fiel7%2F6684409%2F6693054%2F06693090.pdf%3Farnumber%3D6693090

[6] M. Garey and D. Johnson. Computers and Intractability: A Guide to the Theory of NPCompleteness. Freeman and Co., 1979.

[7] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda and Christopher Kruegel. A View on Current Malware Behaviors.
https://www.usenix.org/legacy/event/leet09/tech/full_papers/bayer/bayer.pdf

[8] Respuesta de seguridad. Symantec (08/6/2016)

https://www.symantec.com/security_response/

[9]Enciclopedia de Trend Micro (08/06/2016). http://www.trendmicro.com/vinfo/us/threat-encyclopedia/

[10] Centro de protección malware de Microsoft (08/6/2016)
https://www.microsoft.com/security/portal/mmpc/default.aspx

[11] Detalles técnicos sobre Trojan-Downloader de la empresa F-Secure. (08/06/2016).
https://www.f-secure.com/v-descs/trojan-downloader.shtml

[12] Detalles técnicos sobre Trojan-Fighter.c. (08/06/2016). http://remove-removevirus.com/post/Tips-to-Remove-Win32Fighter.C-From-Your-Computer-Removal-Support_15_169793.html

[13] Centro de protección malware de Microsoft. Resultados sobre la tipología de virus Eva.X (08/6/2016) https://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Virus%3AWin32%2FEva.F

[14] Hilo del foro de Anubis. Evaluación de los tags del XML (31/2/16) https://www.anubis.iseclabforum.org/tag-annotations/

[15] C. Collberg, C. Thomborson, D.Low: "A Taxonomy of Obfuscating Transformations", Dept of Computer Science, Universidad de Auckland, Julio 1997.