ARTÍCULO ORIGINAL
INFORMÁTICA EMPRESARIAL

Validación de un modelo de protección basado en la ofuscación del código

Validation of a protection model based on code obfuscation

Miguel Rodríguez Véliz[1, *] https://orcid.org/0000-0003-4474-3853
Anaisa Hernández González[2] https://orcid.org/0000-0003-1169-301X
Roberto Sepúlveda Lima[3] https://orcid.org/0000-0002-9451-6395
Yulier Núñez Musa[4] https://orcid.org/0000-0001-6588-4896

[1]Facultad de Ciencias Informáticas, Universidad Técnica de Manabí, Portoviejo, Ecuador
[2]Facultad de Ingeniería Informática, Universidad Tecnológica de la Habana "José Antonio Echeverría" La Habana, Cuba
[3]Facultad de Ingeniería Informática, Universidad Tecnológica de la Habana "José Antonio Echeverría" La Habana, Cuba
[4]Investigador independiente, España

*Autor para la correspondencia: mjrodriguez@utm.edu.cu

RESUMEN
La ofuscación del código es una de las técnicas que se utiliza para la privacidad ante ataques de ingeniería inversa al código fuente. Esta investigación presenta un modelo para ofuscar el grafo de llamada como técnica de protección del código fuente y su validación a través de la realización de pruebas de caja blanca, caja negra y la evaluación el desempeño de la oscuridad y el costo del modelo en ataques estáticos de ingeniería inversa. A partir de las medidas que usualmente se emplean para cuantificar la calidad de las técnicas de ofuscación y los atributos de calidad de la ofuscación más comunes, se fundamenta el empleo del grado de diversificación y ofuscación alcanzado por el grafo de llamadas del programa ofuscado $\vartheta(P)$, respecto al programa original P en el modelo que propone este trabajo.
Palabas claves: modelo de protección de software; calidad de las técnicas de ofuscación; potencia de diversificación; potencia de ofuscación

ABSTRACT
Code obfuscation is one of the techniques used for privacy against reverse engineering attacks on the source code. This research presents a model to obfuscate the call graph as a source code protection technique and its validation through the performance of white-box and black-box tests and the evaluation of the obscurity performance and the cost of the model in static reverse engineering attacks. Based on the measures that are usually used to quantify the quality of obfuscation techniques and the most common obfuscation quality attributes, the use of the degree of diversification and obfuscation achieved by the call graph of the obfuscated program ϑ(P), with respect to the original program P in the model proposed by this work is based.
Keywords: software protection model; quality of obfuscation techniques; diversification power; obfuscation power.

Recibido:21/09/24
Aceptado:28/09/24

Introducción

La integridad, la confidencialidad, la autenticación, el no repudio y la auditabilidad y la disponibilidad, son dimensiones de la seguridad que pueden verse comprometidas por un atacante haciéndose valer de distintas técnicas y herramientas [1].

El software está sujeto a ataques constantes de usuarios maliciosos, por lo que se debe tener una constante actualización en seguridad informática, protegiéndolo con diferentes técnicas [2]. Es decir, las empresas deben constantemente adaptar sus procesos de desarrollo de software para incorporar prácticas de seguridad que dificulten la violación de sus aplicaciones.

El código fuente de una aplicación informática usualmente es vulnerable ante ataques cibernéticos que exponen sus funcionalidades, datos y comunicaciones. La seguridad informática consiste en la implementación de medidas y procedimientos diseñados para proteger la confidencialidad, integridad y disponibilidad de datos [3]. Para estos autores, los mecanismos de seguridad de una organización pueden estar vinculados al software (control de acceso, firewalls, antivirus, protección de software), al hardware (copias de software, construcción de entorno físico, planes preventivos) y a las personas (prácticas de seguridad y capacitaciones periódicas).

Garantizar la seguridad de un programa al 100% no es posible, pero sí se puede dificultar la tarea de quien pretenda robar o manipular el código fuente [4]. La ofuscación de código, la marca de agua, la marca de nacimiento y la protección de hardware son algunas de las estrategias que pueden emplearse con esta finalidad.

La ofuscación es una técnica que transforma el código en otro menos entendible, dificultando el entendimiento del mismo, pero manteniendo la

funcionalidad, aunque el código original haya sido modificado [5]. Debido a ello, es considerada una técnica de protección de software que intenta confundir al atacante de forma que no pueda identificar la parte del sistema que desea vulnerar [4].

La ofuscación se ha desarrollado durante las últimas tres décadas; sin embargo, hay preguntas no resueltas: ¿cuánto pueden confiar los desarrolladores en la técnica? y ¿cómo diseñar soluciones de ofuscación confiables [6]. Medir la calidad de las técnicas de ofuscación no solo es útil sino también necesario porque el ofuscador no puede decir si la ofuscación es efectiva si no hay una medida de su eficacia [7].

El grado de confusión de un modelo de protección basado en la ofuscación de código del grafo de llamada, se relaciona con la capacidad del modelo para dificultar la comprensión del funcionamiento del software original. Este converge, directamente con la complejidad de estas transformaciones, y se mide en función de la dificultad que tiene el atacante para construir el grafo de llamada original a partir del ofuscado, sin necesidad de llegar a obtener el código fuente.

En este trabajo se presenta un modelo para ofuscar el grafo de llamada como técnica de protección del código fuente y su validación a través de la realización de ataques de caja blanca, caja negra, la evaluación el desempeño de la oscuridad y el costo del modelo en ataques estáticos de ingeniería inversa.

Métodos

La ofuscación de programas puede abordarse desde dos puntos de vista [8]. El primero tiene un enfoque práctico en relación con la programación de sistemas, mientras que el segundo tiene un enfoque teórico desde el punto de vista de la criptografía.

En este trabajo, se aborda un enfoque práctico de la ofuscación, dada la necesidad de la protección de los sistemas informáticos en el contexto de la piratería de software. Por esta razón, se toma como referencia la definición de ofuscación de programas inicialmente introducida por [9] citada a continuación:

Definición de transformación de ofuscación: Dado un conjunto de transformaciones de ofuscación $T = \{T_1, ..., T_n\}$, y un programa P compuesto por objetos de código fuente (clases, métodos, declaraciones, etc.) $\{S_1, ..., S_k\}$, se busca un nuevo programa $\vartheta(P) = \{..., S'_j = T_i (S_j), ...\}$ tal que:

- $\vartheta(P)$ tiene el mismo comportamiento observable que P, es decir, las transformaciones
  conservan la semántica del programa original.
- $\vartheta(P)$ es un programa ofuscado respecto a P, es decir, entender y realizar ingeniería inversa en
  $\vartheta(P)$ consumirá más tiempo que entender y realizar ingeniería inversa en P.

- Se maximiza la resistencia de cada transformación $T_i\ (S_j)$, es decir, será difícil construir una herramienta automática para deshacer las transformaciones o ejecutar dicha herramienta implicará un consumo extremo de tiempo.
- Se maximiza el sigilo de cada transformación $T_i\ (S_j)$, es decir, las propiedades estadísticas de S'_j son similares a las de S_j.
- Se minimiza el costo (la penalización de tiempo/espacio de ejecución incurrida por las transformaciones) de $\vartheta(P)$.

Por otra parte, [5] mencionan que la ofuscación de código se considera una técnica consistente en alterar el código de un programa o software, de forma tal que sea difícil de entender o modificar por una persona que lo examine sin autorización. Dicha técnica se utiliza comúnmente en la industria del software para proteger la propiedad intelectual y evitar la ingeniería inversa.

Para identificar por adelantado las posibles vulnerabilidades y encontrar los fallos de seguridad y mitigarlos, hay que contar con un modelo de amenazas. En el caso que nos ocupa, es necesario definir las suposiciones sobre la situación del atacante al analizar una componente ejecutable presuntamente ofuscada:

(i) El atacante tiene control total del ordenador donde se ejecuta el software ofuscado. Es decir, se satisface el atributo del modelo de amenaza de host no confiable [10].
(ii) El atacante dispone de herramientas que permiten desensamblar el código de la aplicación, recuperar el grafo de llamadas, analizarlo y hacer comparaciones entre distintas instancias de grafos de llamadas ofuscados.
(iii) El atacante aspira a tomar el grafo ofuscado con el objetivo de reconstruir el grafo original a partir de la reconstrucción de los nodos (funciones), los arcos, o ambos.

Conocer la naturaleza de los ataques que un atacante puede llevar a cabo permite identificar a priori los elementos que lo dificultan, a la hora de diseñar el mecanismo de protección. El tipo de ataque está condicionado por el modelo de amenaza y el atributo de seguridad que hay que garantizar (la privacidad en este caso).

Un mecanismo de ofuscación que garantice la privacidad del programa puede ser efectivo contra un ataque estático mientras dicho programa está almacenado en disco, no necesariamente ocurre así cuando se trata de un ataque dinámico con el programa ejecutándose en memoria. Ambos ataques pueden complementarse en un modelo de ataque híbrido, en general, más efectivo y complejo [11].

El ataque estático consiste en inspeccionar el programa y los datos manejados por dicho programa mientras se encuentra en el medio de almacenamiento. Bajo este tipo de ataque, solo se obtiene información estática del programa, inspeccionando el código del programa después de un proceso de desensamblado o invocando a un descompilador.

El ataque dinámico se realiza sobre el código y los datos durante la ejecución en memoria. De esta manera, se obtiene información (no persistente), como: flujo de control y datos, valores de registro, pila, zonas de memoria, entre otros. Algunos de los procesos que permiten obtener esta información son: depuración, emulación, perfilado y rastreo de programas.

Ambos ataques pueden llevarse a cabo mediante una caja negra (prueba funcional) o un análisis de caja blanca (prueba estructural), o una combinación de ambos.

En el caso de un análisis de caja negra, el atacante se limita a un análisis inter-procedural del grafo de llamadas, es decir, a analizar las funciones (nodos) y las llamadas (arcos). Por otro lado, en un análisis de caja blanca, después de realizar primero un análisis de caja negra, el atacante realiza un análisis inter-procedural de cada función (nodo) de interés. La combinación de ambos ataques es más efectiva y costosa en recursos. Esta es la estrategia de ataque que se siguió en la evaluación de la propuesta.

En [7] se presentan los resultados de una revisión sistemática a la literatura que revisó las investigaciones publicadas entre 2010 y 2020 sobre las técnicas de ofuscación de software y los atributos de calidad. Los principales hallazgos encontrados son:

- La cantidad de software malicioso ha aumentado producto a la adopción generalizada de la informática y la evolución de la tecnología hacia la internet de las cosas.
- Los métodos y técnicas de ofuscación van desde técnicas sintácticas (por ejemplo, la inserción de predicados opacos) hasta semánticas (por ejemplo, los flujos de control).
- Las medidas empleadas para cuantificar la calidad de las técnicas de ofuscación son: complejidad (para medir la potencia), el esfuerzo humano (para medir la resiliencia), la eficiencia (para estimar el costo), las métricas de desempeño multiclase y las medidas de distancia y métodos de emparejamiento (para medir potencia).
- Los atributos de calidad de la ofuscación más comunes son: costo, resiliencia, la potencia, el sigilo y la similitud (Figura 1).

Fig. 1- Atributos de calidad de la ofuscación.
Fuente: Elaboración propia a partir de las definiciones de [7].

En [12] propusieron un modelo para proteger la propiedad intelectual de las expresiones aritméticas en el código fuente mediante técnicas de ofuscación de datos, donde presentan varias técnicas de ofuscación de datos para diferentes operadores aritméticos, como la generación adaptativa de números, el relleno de barras y la aritmética booleana mixta. Llevaron a cabo experimentos para validar la funcionalidad del modelo propuesto y medir las métricas de rendimiento, como la complejidad ciclomática, la complejidad temporal, la complejidad espacial y el análisis de potencia. Los resultados mostraron un ligero aumento en el número de líneas de código y el tiempo de ejecución del código ofuscado en comparación con el código original.
En otros estudios, como el de [13] emplearon técnicas de ofuscación de código, como la codificación de cadenas, el cifrado y el empaquetado, para crear aplicaciones confusas con fines de prueba y evaluación. Además, utilizaron la métrica del ratio de ganancia de información para cuantificar el poder de diferenciación de las funciones gráficas a la hora de clasificar el malware y las aplicaciones benignas.
Por otro lado, en [14] expresaron que el método tradicional de detección de malware basado en firmas dificulta reconocer los últimos programas maliciosos ofuscados y proponen un método para identificar variantes de malware basándose en gráficos de llamadas a funciones. Los gráficos de llamadas a funciones se crean a partir de códigos de programas desensamblados y se analizan las relaciones entre las funciones que llaman y las que llaman, junto con la información del código de operación. Utilizando técnicas de coloración de gráficos para medir la métrica de similitud entre dos gráficos de llamadas a funciones. Los resultados experimentales mostraron que el método propuesto es eficaz para identificar software malintencionado ofuscado.
En [15] se evalúa la diversificación binaria automática para ayudar a eludir los detectores de criptojacking de WebAssembly. Los resultados de los experimentos proporcionaron pruebas de la eficacia de la técnica de evasión. También respalda la afirmación de que la diversificación binaria es una técnica de evasión eficaz para evitar la detección.
En este trabajo se evaluará la ofuscación en base a potencia de diversificación y potencia de ofuscación, para asegurar que las transformaciones de ofuscación preserven la semántica de la ejecución del programa, se evalúa que: para el mismo conjunto de datos de prueba de entrada, tanto para el programa *P* como para el correspondiente programa ofuscado $\vartheta(P)$, la salida sea la misma.
Se propone evaluar el desempeño de la oscuridad y el costo del modelo propuesto en ataques estáticos de ingeniería inversa [9].

En el caso de la oscuridad, se evalúa el grado de diversificación y ofuscación alcanzado por el grafo de llamadas del programa ofuscado $\vartheta(P)$, respecto al programa original *P*.

Potencia de diversificación D_p: Se calcula el grado de diferencia $D_p\ (\vartheta, P)$ de cada instancia del grafo de llamadas de $\vartheta(P)$ respecto a *P*, expresado en porcentaje. Se define como [16]:

$$\mathrm{D_p}\ (\vartheta, \mathrm{P}) = 100\ \left(\frac{\mathrm{F+S}}{\mathrm{T}}\right) \qquad (1)$$

donde:

- F = |A(ϑ (P)) − A(P)| es el total de arcos que están en ϑ(P) y no están en P.
- S = |A(P) – A(ϑ(P))| es el total de arcos que están en P y no están en ϑ(P).
- T = F + S + |A(P) ∩ A(ϑ(P))|, siendo |A(P) ∩ A (ϑ(P))|[3] es el total de arcos que están en ϑ(P)
  y en P.

Para el cálculo del grado de diversificación se empleó la herramienta BinDiff [17], la cual es ampliamente utilizada en procesos de ingeniería inversa.

Potencia de ofuscación O_p: Se aplica la métrica de complejidad ciclomática sugerida por (18) para evaluar la complejidad de cada instancia del grafo de llamadas de $\vartheta\ (P)$, respecto al grafo de *P*. Se define como [16]:

$$\mathrm{O_p}\ (\vartheta, \mathrm{P}) = 100\ \left(\frac{\mathrm{C}(\vartheta(\mathrm{P})}{\mathrm{C(P)}} - 1\right) \qquad (2)$$

donde:

- C*(M)* es la complejidad ciclomática del grafo *M*. La complejidad ciclomática se calcula mediante la ecuación *C(M) = E - N + 2P*, donde *E* y *N* representan el número de arcos y nodos del grafo respectivamente, y *P* representa el número de componentes conectados, es decir, el número de nodos que son puntos de salida. En el caso del grafo de llamadas de un programa, *P* = 1 ya que un programa convencional tiene un solo punto de entrada, que a su vez es el único punto de salida (por ejemplo, la función principal).

Por otra parte, el costo está asociado a la sobrecarga del tiempo de ejecución y del tamaño (espacio en disco) del programa ofuscado respecto al programa original. La sobrecarga de tiempo T_{ovh} y de espacio S_{ovh} se definen como [16]:

$$\mathrm{T_{ovh}}(\theta, \mathrm{P}) = 100\ \left(\frac{\mathrm{T}(\theta(\mathrm{P}))}{\mathrm{T(P)}} - 1\right) \qquad (3)$$

$$S_{ovh}(\theta, P) = 100 \left(\frac{S(\theta(P))}{S(P)} - 1\right) \qquad (4)$$

donde:

T(M) y *S(M)* son el tiempo de ejecución y tamaño de *M* respectivamente.

Cada instancia del programa ofuscado ϑ(P)i se ejecuta tomando los datos de prueba almacenados en la carpeta de entrada correspondiente. Después de su ejecución, la instancia del programa ofuscado genera un resultado que se compara con los datos almacenados en la carpeta de salida del banco de prueba. Si los resultados coinciden, entonces la instancia del programa ofuscado ϑ(P)i generada conserva la semántica del programa original P (ver Definición 1) dado que, para los mismos datos de entrada, se genera los datos de salida esperados. Si los resultados no coinciden, se descarta la instancia del programa ofuscado que se generó.

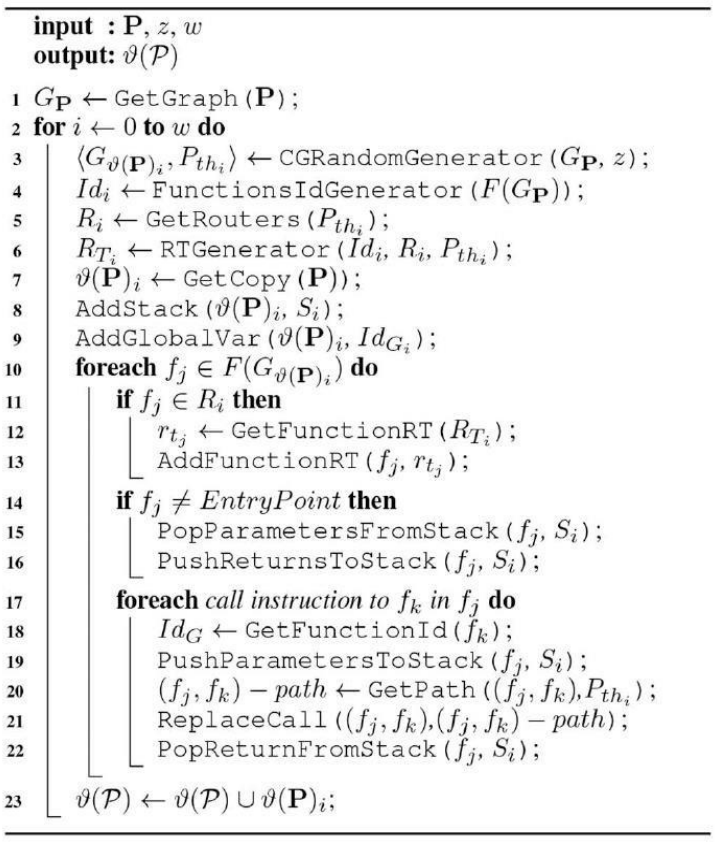
La plataforma de evaluación constó de un procesador Intel Core i3-2120 de doble núcleo (hyperthreaded) a 3.30 GHz, caché L1 de instrucción y datos de 32 KB por núcleo, caché L2 unificada de 256 KB, y caché L3 unificada de 3 MB, 2 GB de RAM, y el sistema operativo utilizado es una distribución Ubuntu 14.4 de 32 bits.

Resultados

La ejecución determinista de algoritmos de ofuscación (con la misma entrada ofrece siempre la misma salida y el mismo curso de ejecución), permite a los atacantes obtener más información lo que facilita que una secuencia organizada de ataques permita alcanzar conocimiento de forma asintónica y conducir al éxito. Una solución no determinista con la misma entrada ofrece múltiples salidas o resultados, sin conocerse de antemano las posibles relaciones entre ellas siguiendo una distribución de probabilidad determinada.

El modelo para la privacidad de software basada en la diversificación y ofuscación del grafo de llamadas sigue el enfoque no determinista. Se clasifica como una ofuscación inter-procedural ya que involucra cambios y transformaciones que abarcan más allá de una solo función o método.

Los componentes que lo integran son: generación aleatoria del grafo de llamadas, generación y asignación de identificadores, obtención del conjunto de enrutadores, generación de tablas de enrutamiento y paso de parámetros y valores de retorno. En la Figura 2 se presenta el algoritmo que implementa.



input : $\mathbf{P}, z, w$
output: $\vartheta(\mathcal{P})$

$G_{\mathbf{P}} \leftarrow$ GetGraph $(\mathbf{P})$;
for $i \leftarrow 0$ **to** w **do**
  $\langle G_{\vartheta(\mathbf{P})_i}, P_{th_i} \rangle \leftarrow$ CGRandomGenerator $(G_{\mathbf{P}}, z)$;
  $Id_i \leftarrow$ FunctionsIdGenerator $(F(G_{\mathbf{P}}))$;
  $R_i \leftarrow$ GetRouters (P_{th_i});
  $R_{T_i} \leftarrow$ RTGenerator (Id_i, R_i, P_{th_i});
  $\vartheta(\mathbf{P})_i \leftarrow$ GetCopy $(\mathbf{P})$);
  AddStack $(\vartheta(\mathbf{P})_i, S_i)$;
  AddGlobalVar $(\vartheta(\mathbf{P})_i, Id_{G_i})$;
  foreach $f_j \in F(G_{\vartheta(\mathbf{P})_i})$ **do**
    if $f_j \in R_i$ **then**
      $r_{t_j} \leftarrow$ GetFunctionRT (R_{T_i});
      AddFunctionRT (f_j, r_{t_j});
    if $f_j \neq EntryPoint$ **then**
      PopParametersFromStack (f_j, S_i);
      PushReturnsToStack (f_j, S_i);
    foreach *call instruction to* f_k *in* f_j **do**
      $Id_G \leftarrow$ GetFunctionId (f_k);
      PushParametersToStack (f_j, S_i);
      $(f_j, f_k) - path \leftarrow$ GetPath $((f_j, f_k), P_{th_i})$;
      ReplaceCall $((f_j, f_k), (f_j, f_k) - path)$;
      PopReturnFromStack (f_j, S_i);
  $\vartheta(\mathcal{P}) \leftarrow \vartheta(\mathcal{P}) \cup \vartheta(\mathbf{P})_i$;

Fig. 2 - Algoritmo de ofuscación y diversificación aleatoria de grafo de llamadas.
Fuente: Rodríguez Véliz (2020) [16].

La prueba de caja blanca se realizó empleando el análisis de código estático ayudando a identificar problemas de seguridad en el código fuente. La prueba de caja negra empleó el análisis de código dinámico para escanear vulnerabilidades que permitan a los equipos de software escanear aplicaciones en ejecución.

Para los seis programas seleccionados del banco de pruebas (164.gzip (7.6 KLOC), 256.bzip2 (3.9 KLOC) y 181.mcf (1.9 KLOC) (pertenecientes al banco de prueba SPECint2000), y 188.ammp (12.9 KLOC), 179.art (1.2 KLOC) y 183.equake (1.2 KLOC. Para cada programa, se generaron de forma aleatoria w = 64 instancias válidas de programas ofuscados, en las cuales el grafo de llamadas varía de una instancia a otra. En promedio, se añadieron z = 1 enrutadores a cada ruta de cada instancia de programa ofuscado generado. En el caso del grado de ofuscación y diversificación, se identificó el valor máximo obtenido de las 64 instancias generadas con: max (Dp, Op).

Sobrecarga de tiempo de ejecución y tamaño

En la Figura 3 se puede observar la sobrecarga mínima impuesta por el modelo de protección en el tiempo de ejecución. También se puede ver que

en los casos de las aplicaciones 181.mcf y 183.equake del banco de pruebas no hubo un impacto notable en el tiempo de ejecución.

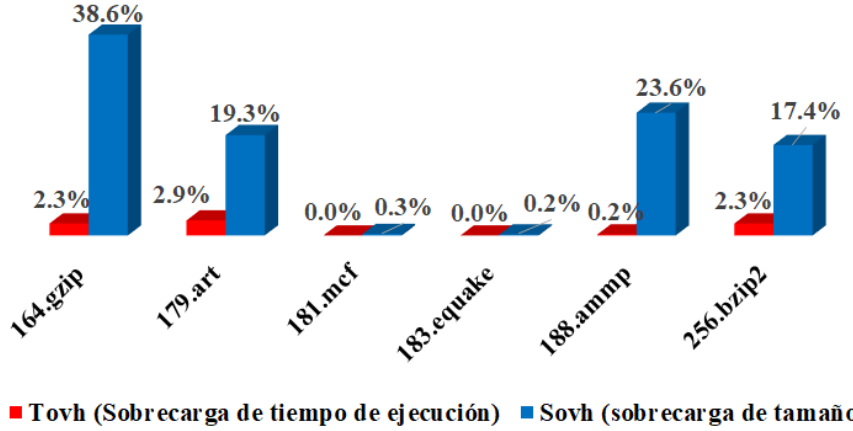


Fig. 3 - Resultados de la aplicación del algoritmo en los programas seleccionados.
Fuente: Elaboración propia.

En el caso de 188.ammp, el impacto es casi insignificante. Para el resto de las aplicaciones, el impacto no superó el 3%, lo cual es permisible para el usuario que utiliza el software, ya que el incremento en el tiempo de ejecución es muy bajo, si es comparado con otras propuestas.
Para la sobrecarga de tamaño mínima causada por el modelo de ofuscación, nótese de nuevo, que en el caso de las aplicaciones 181.mcf y 183.equake el impacto en el tamaño es mucho menor que en el resto de las aplicaciones. Para estas dos aplicaciones, el impacto es casi nulo, ya que no alcanza el 0.5% del tamaño original. Para el resto de las aplicaciones, el impacto es mucho mayor, aunque no supera el 40%.

Grado de diversificación y ofuscación

En la Figura 4, después de aplicar el modelo de ofuscación, se puede ver el valor máximo de diversificación alcanzado por cada aplicación del banco de pruebas y el grado de ofuscación *(16)*n.

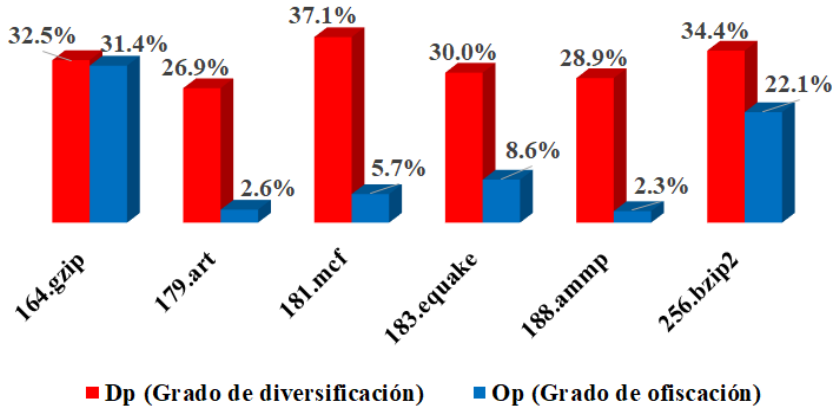


Fig. 4 - Grado de diversificación y ofuscación.

Fuente: Elaboración propia.

Es necesario destacar la eficiencia del modelo propuesto. Para todos los casos, se alcanzó un valor de diversificación superior al 25% respecto a cada programa original, con solo una sobrecarga del 3% en el tiempo de ejecución. En el caso de las aplicaciones 181.mcf y 183.equake, se alcanzó una diversificación de alrededor del 30% sin ningún impacto en el tiempo. Esto presupone la posibilidad de aumentar el nivel de diversificación sin tener un impacto considerable en el tiempo.

El grado de ofuscación alcanzado tiene una mayor variabilidad que en el caso de la diversificación. Además, se obtiene un valor menor en la calidad de la ofuscación respecto a la calidad de la diversificación.

Sin embargo, se alcanzó al menos un aumento del 2% en el grado de complejidad ciclomática para todos los casos, obteniendo los mejores resultados en las aplicaciones 164.gzip y 256.bzip.

Es importante destacar que, aunque el nivel de ofuscación logrado es menor que el de la diversificación, se obtienen muy buenos resultados en cuanto al impacto en el tiempo de ejecución. En el peor de los casos (188.ammp), se logró un aumento del 2.27% en la ofuscación, con solo un impacto del 0.18% en el rendimiento.

Evaluación de seguridad

Tomando como referencia las suposiciones antes mencionadas con respecto al modelo de amenazas, el objetivo del atacante es reconstruir el grafo de llamadas original a partir del grafo de llamadas ofuscado. Para ello, el atacante puede realizar un ataque estático, dinámico o una combinación de ambos mediante un análisis de caja negra o caja blanca como parte de la estrategia de ataque.

La Tabla 1 muestra un resumen de la resistencia a estos ataques del modelo propuesto.

Tabla 1 - Nivel de resistencia del modelo propuesto.

	Análisis de caja negra	**Análisis de caja blanca**
Ataque estático	**Alto** No es posible reconstruir el grafo original, ya que no se puede determinar si una llamada de función a otra ejecuta la lógica de negocio o enruta la llamada a la siguiente función.	**Medio** Se puede reconstruir satisfactoriamente el grafo de llamadas original con esfuerzo, mediante el uso de algún desensamblador. Sin embargo, se puede lograr un alto nivel de protección si se combina con un mecanismo de ofuscación intra-procedural,
Ataque dinámico	**Alto** No es posible reconstruir el grafo original, no se puede determinar qué nodos de esa ruta actúan como enrutadores y cuáles no.	**Bajo** Se puede reconstruir satisfactoriamente el grafo de llamadas original con esfuerzo, aún si se combina con un mecanismo de ofuscación intra-procedural,

Fuente: Elaboración propia.

Dicho modelo propuesto ofrece una alta resistencia a un ataque estático o dinámico de caja negra, ya que el atacante se ve limitado a realizar un análisis funcional (solo nodos y arcos) del grafo de llamadas.

En el caso de un ataque estático de caja negra, después de observar únicamente el grafo de llamadas generado por algún desensamblador (por ejemplo, el Desensamblador IDA), el atacante no puede determinar si la llamada de una función a otra se realiza para ejecutar la lógica de negocio de la función llamada, o si dicha función enruta la llamada a la siguiente función. En otras palabras, para una ruta dada del grafo, el atacante no puede determinar si los nodos intermedios son llamados para ejecutar su lógica de negocio o para enrutar la llamada. Por lo tanto, el atacante no puede diferenciar un tipo de llamada de otro, por lo que no es posible reconstruir el grafo original.

En el caso de un ataque dinámico de caja negra, la diferencia con respecto al ataque estático de caja negra es que el atacante puede observar las rutas que se ejecutan en el grafo de llamadas (por ejemplo, al realizar un perfilado y observar la pila de llamadas o la traza de la pila resultante del perfilado) para cada entrada de datos específica. Del mismo modo, el atacante no puede determinar qué nodos de esa ruta son llamados como enrutadores y cuáles no.

Dada la dificultad de realizar un ataque de caja negra, ya sea estático o dinámico, el atacante se ve obligado a realizar un ataque de caja blanca (prueba estructural), que le permita analizar la estructura interna de cada función.

En el caso de un ataque estático de caja blanca, el modelo propuesto ofrece una resistencia media y el atacante puede reconstruir satisfactoriamente el grafo de llamadas original con mayor o menor esfuerzo. El atacante sólo tiene que desensamblar o descompilar cada método con algún desensamblador (por ejemplo, el Desensamblador IDA) y analizar el valor de la variable global $\boldsymbol{Id_G}$ para determinar qué método debe ser llamado. Para hacer esto, el atacante analiza la tabla de enrutamiento de cada método, hasta que identifica el método en el que se encuentra la lógica de negocio asociada al valor inicial asignado a la variable global $\boldsymbol{Id_G}$.

Es necesario destacar que el modelo propuesto puede lograr un alto nivel de protección contra un ataque estático de caja blanca si se combina con un mecanismo de ofuscación intra-procedural, es decir, del CFG de la función. La seguridad, en este caso, depende de la seguridad del mecanismo de ofuscación intra-procedural que se utilice. Por ejemplo, es posible aplicar predicados opacos para ofuscar el flujo de control de la tabla de enrutamiento y así aumentar la seguridad del modelo propuesto.

Como se mencionó anteriormente, el atacante podría realizar un ataque eficiente al modelo de protección sólo si se lleva a cabo un ataque dinámico de caja blanca. Bajo este ataque, el modelo de protección propuesto ofrece un

nivel bajo de resistencia, incluso cuando se contempla cualquier mecanismo de ofuscación intra-procedural.

Para llevar a cabo el ataque, el atacante realiza un proceso de depuración del código binario previamente desensamblado con una herramienta (por ejemplo, el Desensamblador IDA). A continuación, se muestra un conjunto de pasos que el atacante puede seguir para reconstruir de manera efectiva el grafo original utilizando un ataque dinámico de caja blanca (este no es el único método de ataque, puede haber otros a identificar) (16):

1. Colocar un punto de interrupción al inicio (antes de cualquier instrucción de llamada) de la lógica de negocio de cada función, incluyendo el punto de entrada del programa. A continuación, el atacante procede a iniciar la ejecución de la aplicación en modo de depuración.
2. Marcar la función que detiene la ejecución del programa en el primer punto de interrupción correspondiente como c*aller*. Al comenzar la ejecución del programa, el primer *caller* es siempre el punto de entrada del programa (por ejemplo, el método *main*). Luego, el atacante continúa con la ejecución del programa.
3. Marcar la función que detiene la ejecución del programa en el siguiente punto de interrupción correspondiente como *callee*. En este momento, el atacante ha identificado un arco válido ***(caller, callee)***. Si el atacante observa la pila de llamadas del programa, entonces se puede observar la ruta que reemplaza al arco que se ha identificado. Las funciones intermedias entre *caller* y *callee* son los enrutadores de dicha ruta. Luego, el atacante continúa con la ejecución del programa para identificar un nuevo arco.
4. Marcar el *callee* del arco anterior como *caller* de dicho nuevo arco, cuando la ruta correspondiente todavía se encuentra almacenada en la pila de llamadas en el momento en que la ejecución del programa se detiene en el siguiente punto de interrupción. De lo contrario, el punto de entrada del programa se marca como *caller* del nuevo arco y el *callee* del nuevo arco es la función que contiene el punto de interrupción actual.

Los pasos del 2 al 4 se repiten hasta que se finaliza la ejecución del programa, momento en el cual el atacante habrá identificado todos los arcos reales y podrá construir el grafo de llamadas original del programa.

La complejidad de este ataque radica en los siguientes dos aspectos [16]:

- El atacante debe ser capaz de identificar un área de la lógica de negocio de cada método para colocar un punto de interrupción antes de la primera instrucción para llamar a una función. En este caso, se asume que el atacante tiene las habilidades y las herramientas necesarias de ingeniería inversa para realizar esta tarea, por lo que no sería difícil de hacer.
- Dado que el atacante realiza un ataque de caja blanca, entonces debe ser capaz de identificar el conjunto de casos de prueba necesarios para lograr la máxima cobertura de funciones y, así, ser capaz de identificar todas las llamadas potenciales (arcos). En este sentido, la complejidad ciclomática

representa el número máximo de rutas linealmente independientes del grafo de llamadas, que a su vez representa el número mínimo de pruebas necesarias para cubrir todos los arcos. De esta manera, al aumentar la complejidad ciclomática del programa ofuscado, se obliga al atacante a generar un conjunto más grande de pruebas necesarias para cubrir todos los arcos, lo que aumenta el tiempo necesario para llevar a cabo un exitoso ataque dinámico de caja blanca. Por ejemplo, al observar el grado de ofuscación logrado para el programa 164.gzip (ver Figura 13), el atacante tiene que hacer un esfuerzo adicional de aproximadamente el 31% para generar el número mínimo de pruebas necesarias, al igual que implementado.

Discusión

La fiabilidad del software es un problema causado por la ausencia de técnicas de protección de software, especialmente la ausencia de protección de código de software contra la ingeniería inversa. La piratería informática y el uso ilegal de software causan un gran daño a la economía. El desarrollo de nuevos enfoques y la modificación de las tecnologías de ofuscación existentes, es una tarea actual dirigida a aumentar la eficacia de la codificación y la protección contra la ingeniería inversa [19].

La evaluación de la sobrecarga impuesta por el modelo de protección en el tiempo de ejecución es altamente favorable. En la mayoría de los casos, el impacto en el tiempo de ejecución se mantiene por debajo del 3%, lo que se considera aceptable para la experiencia del usuario.

La sobrecarga de tamaño causada por el modelo de ofuscación varía según la aplicación, en algunos casos, siendo prácticamente insignificante (por debajo del 0.5% del tamaño original) y en los otros no sobrepasando el 40% del tamaño original. Este rango de impacto se considera manejable y aceptable en función de los requisitos y restricciones del sistema.

El modelo de ofuscación ha demostrado ser altamente eficiente al lograr niveles significativos de diversificación, con un incremento mínimo en el tiempo de ejecución. Los resultados muestran que se alcanzaron valores de diversificación superiores al 25%, incluso llegando al 30% en ciertos casos, sin impacto considerable en el rendimiento temporal. Aunque la calidad de la ofuscación presenta una variabilidad mayor y valores inferiores en comparación con la diversificación, la mejora en el grado de complejidad ciclomática, donde todos superaron el 2%, sugiere un fortalecimiento efectivo de la seguridad del código.

El prototipo implementado tiene una serie de limitaciones, que se describen de la siguiente manera [16]:

- Dado que las transformaciones de ofuscación se realizan a nivel de representación intermedia del código fuente, no es posible ofuscar las llamadas a bibliotecas externas cuyo código fuente no esté disponible.

- No es posible ofuscar las llamadas indirectas a funciones (punteros que hacen referencia a funciones) ya que, en tiempo de compilación estática, es computacionalmente difícil determinar la función a llamar a partir de su referencia.
- No es posible proteger los programas que hacen uso de múltiples hilos, ya que es necesario manejar el acceso concurrente a la variable global Id_G para realizar un correcto enrutamiento de cada hilo de ejecución.
- No es posible ofuscar llamadas que involucren funciones con parámetros variables (ellipsis), dado el uso de la propia convención de llamada desarrollada.

Conclusiones

La definición precisa de las suposiciones sobre las capacidades del atacante, tales como: el conocer si tiene control total del ordenador donde se ofusca el software o si dispone de herramientas que le permitan reconstruir el grafo original a partir del grafo ofuscado; es esencial para evaluar la robustez y la resistencia de los modelos de ofuscación. Comprender la amplitud de las amenazas potenciales permite desarrollar mecanismos más efectivos para proteger el software en entornos donde la seguridad es crítica.

En el diseño de mecanismos de protección efectivos es primordial la comprensión de los diferentes tipos de ataques: estáticos, dinámicos o híbridos; así como, de los análisis: caja negra o caja blanca, que pueda llevar a cabo el atacante. Con ello, se crea una base sólida de características que debe tener el mecanismo para elevar su nivel de resistencia ante posibles amenazas.

La oscuridad puede ser medida calculando la potencia de diversificación, evaluada con herramientas como BinDiff; y la potencia de ofuscación, evaluada mediante la aplicación de la métrica de complejidad ciclomática. Además, se debe abordar el costo asociado a la sobrecarga del tiempo de ejecución y del tamaño del programa ofuscado en comparación con el original.

Se debe abordar el costo asociado a la sobrecarga del tiempo de ejecución y del tamaño del programa ofuscado en comparación con el original.

La valoración de la sobrecarga introducida por el modelo de protección durante la ejecución refleja una perspectiva sumamente positiva. En la mayoría de los casos, la repercusión en el tiempo de ejecución se mantiene por debajo del 3%, que desde la perspectiva del usuario es un valor que se estima adecuado. La sobrecarga de tamaño originada por la aplicación del modelo de ofuscación experimenta variaciones dependiendo de la aplicación. El rango de repercusión se evalúa como manejable y aceptable según los requisitos y restricciones del sistema.

La eficiencia del modelo de ofuscación se manifiesta de manera destacada al lograr niveles sustanciales de diversificación, evidenciando un incremento mínimo en el tiempo de ejecución. A pesar de que la calidad de la ofuscación presenta cierta variabilidad y valores relativamente inferiores en comparación

con la diversificación, la mejora apreciable en el grado de complejidad ciclomática, donde todos superaron el 2%, señala un fortalecimiento efectivo de la seguridad del código.

El modelo exhibe una resistencia limitada frente a ataques dinámicos de caja blanca, situación en la cual el atacante logra reconstruir con eficiencia el grafo original. Este proceso se realiza mediante una minuciosa depuración del código binario desensamblado, permitiendo al agresor sortear las defensas establecidas.

Referencias

1. Díaz Pando H., Rodríguez Veliz M., Núñez Musa Y., Sepúlveda Lima, R. La seguridad de la información: un oscuro espacio multidimensional. Informática y Sistemas. Revista de Tecnologías de la Información y las Comunicaciones. 2018; 2(2): 75-84. Disponible en: https://revistas.utm.edu.ec/index.php/Informaticaysistemas/article/download/1427/1503. ISSN 2550-6730.
2. Gatica J., Beron M., Riesco D., Pereira M. J., Henriques P., Novais P. Protección de activos de software. 2023- En: XXV Workkshop de Investigación en Ciencias de la Computación, Junín, págs. 699-703. Disponible en: https://sedici.unlp.ed.ar/hadle/10915/164036. ISBN: 978-987-3724-66-4.
3. Marreros J., Acosta D., Mendoza, A. Mecanismos de seguridad de la información en una organización: una revisión sistemática. Revista Científica Ciencias Ingenieriles. 2024; 4(1): 79-90. Disponible en: https://revistas.unh.edu.pe/index.php/ricci/article/view/384. ISSN: 2961-2357 (Enlínea) ISSN: 2961-2446 (Impreso).
4. Montejano Masa J. P., Berón M., Montejano G. A., Riesco D. E. Métodos, técnicas y herramientas para la protección de sistemas de software. 2023. En: XXV Workkshop de Investigación en Ciencias de la Computación, Junín, págs. 719-723. Disponible en: https://sedici.unlp.ed.ar/hadle/10915/164036. ISBN: 978-987-3724-66-4.
5. Rodríguez Véliz J. M., Núñez Musa Y., Sepúlveda Lima, R. Study of Code Obfuscation Techniques for the Security of Software Components. International Journal of Intelligent Systems and Applications in Engineering. 2023; 11: 1-10. Disponible en: https://ijisae.org/index.php/IJISAE/article/view/3385. ISSN: 2147-6799.
6. Xu H., Zhou Y., Ming J., Lyu M. Layered obfuscation: a taxonomy of software obfuscation tecniques for layered security. Cibersegurity. 2020: 3: 1-18. Disponible en: https://link.springer.com/article/10.1186/542400-020-00049-3.
7. Ebad S., Daren A., Abawagy J. Measuring software obfuscation quality -a systematic literature review. IEEE Access. 2021; 9: 99024-99038. Disponible en:

https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9474335. Electronic ISSN: 2169-3536.

8. Hosseinzadeh S., Rauti S., Laurén S., Mäkelä J. M., Holvitie J. H., Leppänen V. Diversification and obfuscation techniques for software security: A systematic literature review. Information and Software Technology. 2018; 104: 72-93. Disponible en: https://doi.org/10.1016/j.infsof.2018.07.007. ISSN: 0950-5849.
9. Collberg C. S., Thomborson C. Watermarking, tamper-proofing, and obfuscation—Tools for software protection. IEEE Transactions on Software Engineering. 2002; 28(8): 735-746. Disponible en: https://doi.org/10.1109/TSE.2002.1027797. ISSN: 0098-5589 (print) 1939-3520 (web).
10. Main A., Oorschot P. V. Software Protection and Application Security: Understanding the Battleground ?. 2003. En: International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography, Heverlee, Belgium, 19. Disponible en: https://www.semanticscholar.org/paper/Software-Protection-and-Application-Security-%3A-the-Main-Oorschot/82217aeae8b2532f710a20dcad711a.
11. Bai J., Shi Q., Mu S. A Malware and Variant Detection Method Using Function Call Graph Isomorphism. Security and Communication Networks. 2010; e1043794. Disponible en: https://onlinelibrary.wiley.com/doi/epdf/10.1155/2019/1043794. Online *ISSN*:1939-0122.
12. Ahire P., Abraham J. Mechanisms for Source Code Obfuscation in C: Novel Techniques and Implementation. 2022. En: International Conference on Emerging Smart Computing and Informatics (ESCI), Pune, págs. 52-59. Disponible en: https://doi.org/10.1109/ESCI48226.2020.9167661. Electronic ISBN:978-1-7281-5263-9.
13. Ikram M., Beaume P., Kaafar M. A. DaDiDroid: An Obfuscation Resilient Tool for Detecting Android Malware via Weighted Directed Call Graph Modelling. 2019. Disponible en: https://doi.org/10.48550/arXiv.1905.09136.
14. Xu M., Wu L., Qi S., Xu J., Zhang H., Ren Y., Zheng N. A similarity metric method of obfuscated malware using function-call graph. Journal of Computer Virology and Hacking Techniques. 2013: 9. Disponible en: https://doi.org/10.1007/s11416-012-0175-y. ISSN: 2263-8733.
15. Cabrera Arteaga J., Monperrus M., Toady T., Baudry B. WebAssembly diversification for malware evasion. Computers & Security. 2023; 131. doi:https://doi.org/10.1016/j.cose.2023.103296. Disponible en: https://www.researchgate.net/profile/Javier-Cabrera-4/publication/370872428_WebAssembly_Diversification_for_Malware_Evasion/links/64672d7770202663165969 51/WebAssembly-Diversification-for-Malware-Evasion.pdf. ISSN: 0167-4048.

16. Rodríguez Véliz J., Núñez Musa Y., Sepúlveda Lima R. Call graph obfuscation and diversification: An approach. IET Information Security. 2020; 14(2): 241-252. Disponible en: jttps://doi.org/10.1049/iet-ifs.2019.0216. ISSN: 1751-8709.
17. Arutunian M., Hovhannisyan H., Vardanyan V., Sargsyan S., Kurmangaleev S., Aslanyan H. A method to evaluate binary code comparison tools. 2021. En: Ivannikov Memorial Workshop, Yerevan, Armenia, págs. 145-147. Disponible en: https://csit.am/2021/proceedings/MCA/MCA_1.pdf. Electronic ISBN:978-1-6654-2327-4.
18. Cappaert J., Preneel B. A general model for hiding control flow. 2010. En: Proceedings of the tenth annual ACM workshop on digital rights management, (págs. 35-42). Disponible en: https://doi.org/10.1145/1866870.1866877. ISBN: 978-1-4503-0091-9.
19. Gnaytuk S., Kinzeryavgy V., Stepanenko I. Code obsfusaction tecnique for enhacing software protection against reverse engineering. 2020. En: Z. Hu, S. Petovkhov, & M. He, Advances in Artificial Systems for Medicne and Education II. Advances in Intelligent Systems and Computing, págs. 571-580. Disponible en: https://doi.org/10.1007/978-3-030-12082-5_52. Online ISBN: 978-3-030-12082-5.

Conflicto de intereses

Los autores declaran que no hay conflicto de intereses.

Contribución de cada autor:

Miguel Rodríguez Véliz: Desarrollo de la investigación y revisión del artículo.

Anaisa Hernández González: Escritura del artículo y ajuste a las normas de la revista.

Roberto Sepúlveda Lima: Recolección de la información, elementos conceptuales y diseño de la investigación.

Yulier Núñez Musa: Gestor de la idea, líder de la investigación, diseño de su concepción.