RNPS 2487 • ISSN 2708-3411
Vol.6 • 2025 • e267

ARTÍCULO ORIGINAL

Protección del código fuente ofuscando el grafo de llamadas

Protection of source code by obfuscating call graph

Miguel Rodríguez Véliz
miguel.rodriguez@utm.ed.cu • ***https://orcid.org/0000-0003-4474-3853***

FACULTAD DE CIENCIAS INFORMÁTICAS, UNIVERSIDAD TÉCNICA DE MANABÍ, ECUADOR

Anaisa Hernández González
anaisa@ceis.cujae.edu.cu • ***https://orcid.org/0000-0003-1169-301X***

Roberto Sepúlveda Lima
rsepulvedalima@gmail.com • ***https://orcid.org/0000-0002-9451-6395***

FACULTAD DE INGENIERÍA INFORMÁTICA, UNIVERSIDAD TECNOLÓGICA DE LA HABANA "JOSÉ ANTONIO ECHEVERRÍA"

Yulier Núñez Musa
yuliernm@gmail.com • ***https://orcid.org/0000-0001-6588-4896***

INVESTIGADOR INDEPENDIENTE, ESPAÑA

Recibido: 2024-08-06 • Aceptado: 2025-07-02

RESUMEN

Nuevos mecanismos de seguridad se han desarrollado para obstaculizar el accionar de atacantes contra aplicaciones informáticas. Los ataques de ingeniería inversa exponen la vulnerabilidad del código fuente. Aunque no es posible garantizar un 100% de seguridad, la ofuscación del código constituye una opción que hace más difícil el acceso y comprensión de este. La ofuscación del grafo de llamadas es la alternativa que presenta este trabajo y ha demostrado ser altamente eficiente al lograr niveles significativos de diversificación y ocultación. Este trabajo presenta los resultados del análisis de diferentes fuentes donde se exponen algunas técnicas empleadas en la ofuscación del código y se describe una nueva propuesta que emplea a la ofuscación del grafo de llamada.

Palabras clave: Protección del código, Ofuscación del grafo de llamadas.

ABSTRACT

New security mechanisms have been developed to hinder attackers from attacking computer applications. Reverse engineering attacks expose the vulnerability of the source code. Although it is not possible to guarantee 100% security, obfuscation of the code is an option that makes it more difficult to access and understand it. Call graph obfuscation is the alternative presented

in this work and has proven to be highly efficient in achieving significant levels of diversification and concealment. This work presents the results of the analysis of different sources where some techniques used in code obfuscation are exposed and a new proposal that uses call graph obfuscation is described.

Keywords: Code protection, Call graph obfuscation.

INTRODUCCIÓN

Vivimos en un mundo digital: nos comunicamos con teléfonos móviles, redes sociales o correo electrónico, hacemos compras digitales de productos vitales para vivir, de ocio u otros, estudiamos por internet consumiendo diferentes recursos digitales, a través de un aula virtual, realizando búsquedas y de forma sincrónica y asincrónica, trabajamos empleando el trabajo a distancia y el teletrabajo y muchos otros etcéteras.

Anteriormente bastaba con la seguridad impuesta por el acceso físico y algunas simples barreras informáticas (Quiroz Zambrano & Macías Valencia, 2017) para garantizar la seguridad informática, pero la necesidad de interconexión para brindar cualquier servicio hace insuficientes los mecanismos de seguridad tradicionales. El software seguro es un software al que no se puede acceder, actualizar ni atacar un usuario no autorizado (Khan, Khan, Khan, & Ilyas, 2022).

Aunque nuestras conversaciones, mensajes, compras, intercambios u otras interacciones digitales son personales y solo deberían tener acceso aquellos que nos interesa; esto no es realmente cierto. La seguridad informática intenta proteger el almacenamiento, procesamiento y transmisión de información digital (Roa Buendía, 2013), protegiendo los equipos, las aplicaciones, los datos y las comunicaciones. Para Stallings Brown (2022), este término hace alusión a las medidas y controles que aseguran la confidencialidad, integridad y disponibilidad de activos de información de una organización. Whitman Mattord (2022) la protección contra el acceso, uso divulgación, interrupción, modificación o destrucción no autorizados con el fin de proporcionar confidencialidad, integridad y disponibilidad abarca a la información y los elementos críticos, incluidos los sistemas y el hardware que la utilizan, almacenan y trasmiten.

El objetivo primario de la seguridad informática es mantener al mínimo los riesgos sobre los recursos informáticos (entiéndase el equipo de cómputo y telecomunicaciones) y garantizar así la continuidad de las operaciones de la organización al mismo tiempo que se administran los riesgos informáticos a un costo aceptable (Quiroz Zambrano & Macías Valencia, 2017). La seguida del software es un factor esencial de calidad del software para protegerlo contra ataques maliciosos y otras amenazas de piratas informáticos, de modo que el software siga funcionando correctamente bajo tales riesgos potenciales (Morrison, Moye, Pandita, & Laurie 2018).

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 (Gatica, y otros, 2023). Las empresas deben constantemente adaptar su proceso de desarrollo de software para incorporar prácticas de seguridad que dificulten la violación de sus aplicaciones.

La piratería de software genera en la actualidad pérdidas monetarias contabilizadas en cifras millonarias. Una vía para mitigar el problema, desde el punto de vista tecnológico, es el empleo de técnicas de protección de software.

Y es que, los ataques cibernéticos representan una gran amenaza para todos los sistemas o activos que componen una organización, ya que cada día las mismas invierten en mecanismos que sean capaces de mantener la seguridad de la información; ya sea para garantizar la disponibilidad del servicio ofrecido a los clientes o la integridad y confidencialidad de la información que en ella se maneja (Céspedes Maestre, 2021).

Para (Khan, Khan, Khan, & Ilyas, 2022) la mayoría de los ataques de seguridad son posibles debido a fallas de implementación, como validación de entrada incorrecta, mecanismos de autenticación y autorización incorrectos, administración de sesiones incorrecta y otras vulnerabilidades (identificación de sesión vulnerable o robo, cierre de sesión implementado incorrectamente, intentos fallidos de bloqueo por sesión de navegador, restricción de sesión entre pares y función de reproducción de registros) que comprometen la funcionalidad prevista de la aplicación. Aun cuando en un código “no se encuentren” fallas de implementación, cuando se tiene acceso al código fuente las organizaciones quedan expuestas.

Para proteger de ataques a las aplicaciones, usualmente se siguen tres enfoques: (1) ético (amnistía, apelación y shareware) que busca hacer énfasis en lo incorrecto de esta acción, (2) legal (derecho de autor, patente y licencia de software) que se centran en las posibles sanciones y (3) la aplicación de mecanismos técnicos (por ejemplo, marcas de agua, ofuscación, cifrado y autoverificación de integridad) que dificultan el acceso al código fuente.

En la actualidad, las aplicaciones se escriben en lenguajes de alto nivel justificado por las posibilidades y ventajas que estos ofrecen. Pero, también acarrean ciertas desventajas que hacen que la compilación de código incluya implícitamente numerosas técnicas de optimización que van desde eliminar código muerto, la asignación de registros óptima y asignaciones eficientes al objetivo conjunto de instrucciones de la arquitectura particular.

Acceder al código fuente de un software o aplicación incrementa las posibilidades de los llamados “cracks” (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2023), por lo que, existe un problema de vulnerabilidad del código ejecutable ante ataques de ingeniería inversa (ataques que se basan en desmontar el código para entender cómo funciona un software), exponiendo la lógica interna y comprometiendo con ello la propiedad intelectual y la seguridad del software.

Garantizar la seguridad de un programa al 100% no es posible, pero sí es posible dificultar la tarea para quien pretende robar o manipular el código fuente (Montejano Masa, Berón, Montejano, & Riesco, 2023).

Este trabajo presenta los resultados del análisis de diferentes fuentes donde se exponen algunas técnicas empleadas en la ofuscación del código y se describe una nueva propuesta que emplea a la ofuscación del grafo de llamada.

METODOLOGÍA

Esta investigación tiene como antecedentes una revisión sistemática descrita en (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2023) que consistió en un análisis cualitativo de referencias bibliográficas resultantes de un proceso de búsqueda en bases de datos de revistas certificadas y digitalizadas como la plataforma Elsevier Scopus y otras bases de datos académicas, en el periodo de 2013-2023, de los cuales se seleccionaron 30 estudios. Ese trabajo profundizó en las técnicas de ofuscación del código existente, por lo que en esta investigación se revisan en detalle aquellos trabajos identificados 2019 y 2023 que sugieren emplear una o varias técnicas de ofuscación.

Para cumplir con este objetivo, se revisaron publicaciones científicas que abordan la seguridad informática en sentido general y la técnica de ofuscación en específico, tanto desde el punto de vista conceptual como las experiencias prácticas de implementación. Esta revisión permitió identificar 18 trabajos que fueron estudiados tomando en cuenta el tipo de ofuscación que emplean y cómo lo hacen y que se resumieron en un cuadro comparativo. En su desarrollo se aplicó el método de análisis y síntesis para analizar la problemática y arribar a conclusiones.

A partir del estudio, se elaboró una solución que emplea la ofuscación basada en grafo de llamada. En su construcción se tomó en consideración que el grado de confusión debe ser directamente proporcional a la complejidad que enfrente el atacante al intentar descifrar el código bajo análisis (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2020). Para la concepción de la propuesta que se describe en este trabajo, se siguieron los pasos siguientes:

1. Determinar el grado de confusión a alcanzar.
2. Elegir el método de ofuscación.
3. Implementar la ofuscación.
4. Probar la implementación. Si no se aprueba la evaluación, pasar al paso 5. En caso contrario, termina el proceso con un algoritmo satisfactorio.
5. Corregir errores.
6. Regresar al paso 3.

La propuesta que se presenta emplea el grafo de llamadas con la intención de complicar la comprensión del programa al modificar la secuencia de llamadas entre funciones y las relaciones en el grafo.

RESULTADOS Y DISCUSIÓN

La ofuscación del código, las marcas de agua, la marca de nacimiento y la protección de software por hardware son algunas de las estrategias que pueden emplearse con la finalidad de dificultar el acceso al código fuente. Este trabajo se centra en la ofuscación como mecanismo para confundir al atacante.

La ofuscación del código es una técnica que oscure la estructura y/o el comportamiento del código del software sin afectar su funcionalidad esperada, de manera que el código resulte difícil de entender, analizar o aplicar ingeniería inversa. Transforma los programas informáticos en nuevas versiones que son semánticamente equivalentes con las originales, pero más difíciles de entender (Collberg, Thomborson, & Low, 1997; Bin Shamlan, Alaidaroos, Bin Merdhah, Bamatraf, & AA, 2020) ya que modifica la forma y la estructura del código dificultando su análisis y legibilidad (Moreno, 2023).

La ofuscación de código fuente representa la práctica de transformar el código fuente de un software en una forma en la que el código sea difícil de entender o leer para los humanos, pero que sea legible por máquinas. Ante esto, se considera que el objetivo primordial de la ofuscación es proteger tanto la propiedad intelectual como los secretos comerciales de las empresas, buscando evitar la ingeniería inversa (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2023).

Su aplicación no hace completamente inmune al software de ese ataque, pero sí aumenta el esfuerzo requerido para comprender el código ofuscad. Medir la calidad de las técnicas de ofuscación es útil y necesario para saber su efectividad (Ebad, Daren, & Abawagy, 2021). Aunque la ofuscación se ha desarrollado por más de 30 años, las preguntas aún no resueltas están asociadas con cuánto pueden confiar los desarrolladores en la técnica y cómo diseñar soluciones de ofuscación confiables (Xu, Zhou, Ming, & Lyu, 2020).

Esta técnica ha sido investigada por diversos autores, quienes proponen el uso de un tipo particular de ofuscación (del flujo de control, del grafo de llamadas, de instrucciones, de cadenas, de código falso, de renombrado de variables y de cifrado de cadenas) o la combinación de ellas (Tabla 1).

Los ofuscadores de control de flujo mezclan el control de flujo del software, cambiando el orden de las instrucciones o incorporando otras nuevas que ni siquiera son necesarias. Los ofuscadores literales cifran valores literales como cadenas, números y otras expresiones. En los ofuscadores de estructura se busca dividir el software ofuscado en partes más pequeñas para enmascarar las relaciones ocultas entre ellas. Los transformadores de código involucran la aplicación de algoritmos de transformación de código, que incluyen ofuscación de control de flujo, ofuscación de nombres, información de depuración, y fusión de bloques de código (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2023).

Tabla 1 Trabajos donde se emplea la ofuscación. Fuente: Elaboración propia

Trabajo	Tipo de ofuscación	Descripción
Control flow obfuscation via CPS transformation (Ming Lu, 2019)	Ofuscación del flujo de control	Emplea la transformación CPS (Continuation-Passing Style), reescribiendo el programa en un estilo de paso de continuación, fragmentando el gráfico de flujo de control original.
The Impact of Control Flow Obfuscation Technique on Software Protection Against Human Attacks (Bin Shamlan, Bamatraf, & Zain, 2019)	Ofuscación del flujo de control	Utiliza predicados opacos para introducir incertidumbre en la estructura del flujo de control del programa.
Code Obfuscation Based on Inline Split of Control Flow Graph (Li, Xiong, & Zhao, 2021)	Ofuscación del flujo de control	Consiste en convertir saltos entre bloques básicos dentro de una función (intra-procedural) en llamadas a funciones interprocedurales, crear funciones falsas (bogus functions) y añadir predicados opacos para confundir aún más el flujo de control del programa.
Android Control Flow Obfuscation Based on Dynamic Entry Points Modification (Yang, Zhang, Ma, Liu, & Peng, 2019)	Ofuscación del flujo de control	Específicamente diseñada para aplicaciones Android y se basa en la modificación dinámica de los puntos de entrada de los métodos en tiempo de ejecución.
Semantic redirection obfuscation: A control flow obfuscation based on Android Runtime (Wang,	Ofuscación del flujo de control	Técnica basada en la redirección semántica dentro del entorno de ejecución de Android (Android Runtime). Se centra en redirigir dinámicamente las invocaciones de métodos para ocultar el verdadero flujo de control del programa.

Trabajo	Tipo de ofuscación	Descripción
Shan, Yang, Wang, & Song, 2020)		
Call graph obfuscation and diversification: an approach (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2020)	Ofuscación del grafo de llamadas	Basada en la modificación estática del grafo de llamadas mediante el enrutamiento de llamadas a funciones.
Improvements on Hiding x86-64 Instructions by Interleaving (Mahoney, McDonald, Grispos, & Mandal, 2023)	Ofuscación de instrucciones	Centrada en ocultar instrucciones de código máquina en la arquitectura x86-64 mediante el intercalado de instrucciones.
TAD: time side-channel attack defense of obfuscated source code (Fell, Pham, & Lam, 2019)	Ofuscación de instrucciones	Técnica diseñada para defender el código fuente ofuscado contra ataques de canal lateral basados en tiempo. Para ello aplica dos enfoques: 1. Eliminación de Diferencias de Tiempo en Ramas Condicionales. 2. Diversificación Dinámica de Instrucciones.
C Source code Obfuscation using Hash Function and Encryption Algorithm (Tambunan & Rokhman, 2023)	Cifrado de cadenas, ofuscación de instrucciones e inserción de código falso	Técnica que combina el uso de funciones hash y el algoritmo de cifrado DES (Data Encryption Standard).
Hybrid Obfuscation Technique to Protect Source Code From Prohibited Software Reverse Engineering (Al-Hakimi, Sultan, Ghani, Ali, & Admodisastro, 2020)	Cifrado de cadenas, Renombrado de variables e Inserción de código falso	Técnica híbrida de ofuscación para proteger el código fuente contra la ingeniería inversa que combina varias estrategias.
Design and Implementation of Obfuscating Tool for Software Code Protection (Kumar & Sharma, 2019)	Renombrado de variables y Cifrado de cadenas.	La herramienta aplica varias técnicas de ofuscación, incluyendo el renombrado de identificadores y la encriptación de la lógica del programa.
Embedding Java Classes with code2vec: Improvements from Variable Obfuscation (Compton, Frank, Patros, & Koay, 2020)	Renombrado de variables	Describe una técnica de ofuscación que renombra variables en los métodos de Java para hacer que el modelo code2vec confíe menos en los nombres específicos de las variables y más en la estructura del código.
Mechanisms for Source Code Obfuscation in C: Novel Techniques and	Renombrado de variables, Inserción de código falso,	Técnica de ofuscación de código fuente en C que combina varias técnicas.

Trabajo	Tipo de ofuscación	Descripción
Implementation (Ahire & Abraham, 2020)	Ofuscación de instrucciones y Cifrado de cadenas	
Research Based on LLVM Code Obfuscation Technology (Lv, Zhao, & Chen, 2022)	Ofuscación del flujo de control y Ofuscación de instrucciones	Ofuscación de código utilizando la infraestructura LLVM (Low-Level Virtual Machine).
DynOpVm: VM-Based Software Obfuscation with Dynamic Opcode Mapping (Cheng, Lin, Gao, & Jia, 2019)	Ofuscación del flujo de control y Ofuscación de instrucciones	Técnica de ofuscación de software basada en máquinas virtuales (VM) denominada "DynOpVm", utilizando un mapeo dinámico de opcodes (códigos de operación) para proteger el código del software.
CSE: A Novel Dynamic Obfuscation Based on Control Flow, Signals and Encryption (Hashemzade & Abdolrazzagh-Nezhad, 2019)	Ofuscación de flujo de control y Ofuscación de Instrucciones	La técnica trabaja en tiempo de ejecución para modificar dinámicamente el flujo de control y las señales utilizadas en el programa. Esto se logra mediante la encriptación de las tablas de gestión de flujo y la inserción de señales falsas, lo que dificulta el análisis estático y dinámico del programa.
An Efficient Control-flow based Obfuscator for Micropython Bytecode (Wang, Li, Zhang, Han, & Chen, 2021)	Ofuscación de flujo de control y Cifrado de cadenas	Ofuscador basado en el flujo de control específicamente diseñado para bytecode de MicroPython, mediante la inserción de trampas (traps) y la alteración de rutas de ejecución.
A Security Model and Implementation of Embedded Software Based on Code Obfuscation (Yi, Chen, Zhang, Li, & Zhao, 2020)	Ofuscación del flujo de control	Modelo de seguridad basado en la ofuscación del código para proteger software embebido contra ataques de ingeniería inversa. La técnica de ofuscación principal utilizada en este trabajo es el aplanamiento del flujo de control (control flow flattening).

De los 18 trabajos revisados, la mayoría emplea la ofuscación del grafo de control (50%), seguidas en orden por ofuscación de instrucciones (33%), cifrado de cadenas (28%), renombrado de variables (22%) e inserción de código falso (17). El empleo de la ofuscación del grafo de llamada aún es irrelevante en las investigaciones que se documentan en la literatura.

Los métodos no son mejores unos que otros, modifican el código siguiendo la base conceptual que les da nombre, por lo que resultan adecuados para algoritmos que contienen determinados elementos. Por ejemplo, el renombrado de variables cambia los nombres de las variables, funciones y métodos del código fuente por nombres aleatorios o sin sentido; haciendo difícil de entender un código que emplea varias variables, funciones y métodos. La ofuscación del grafo de control modifica el flujo de control del programa. En algoritmos que se caracteriza por invocar a otras funciones, modificar el grafo de llamadas lo protege mejor contra ataques de ingeniería inversa. No es práctica internacional comparar estos algoritmos y como se aprecia el grafo de llamadas aún no es una alternativa muy

empleada, pero que puede resultar válida para las particularidades antes descritas. La variante de emplear varias técnicas brinda una mayor protección.

Partiendo del alcance de las técnicas de ofuscación y a la forma en que afectan al código, la ofuscación de software se puede clasificar en inter-procedural e intra-procedural. La ofuscación inter-procedural se refiere a la aplicación de técnicas que perturban múltiples procedimientos o funciones en un programa. Por otro lado, la ofuscación intra-procedural operan dentro de un solo procedimiento o función en un programa. El cifrado de cadenas, el renombrado de variables, el grafo de control, la inserción de código falso y la ofuscación de instrucciones son consideradas intra-procedural y el grafo de llamadas inter-procedural (Rodríguez Veliz, 2025).

La ofuscación del grafo de llamada es una técnica específica de ofuscación inter-procedural que implica manipular la estructura de las llamadas entre funciones en un programa. El grafo de llamada representa las dependencias y las conexiones entre las funciones de un programa, mostrando cómo se comunican y se llaman entre sí.

Siguiendo los pasos para la obtención del algoritmo y una vez decididos a emplear el grafo de llamadas, se implementaron tres algoritmos:

- Algoritmo de ofuscación y diversificación aleatoria de grafos de llamada.
- Algoritmo de generación aleatoria del grafo de llamadas.
- Algoritmo de generación de tablas de enrutamiento.

Estos algoritmos se describen en detalle en (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2020). En la Figura 1 se muestra un fragmento de código de un programa que fue ofuscado empleando los algoritmos anteriores. En la Figura 2 se muestra el código resultante y el grafo de llamadas asociado. Es importante resaltar que es posible obtener más de una ofuscación (diversificación), mostrando en la Figura 2 c) ejemplos de otros grafos de llamadas de otros programas ofuscados derivados del mismo programa ejemplo.

```c
int main(int argc, char* argv[])
{
  printf("main");
  int iA1 = 5;
  int iA2 = 2;
  double a;
  a = A(iA1, iA2);

  double dB1 = 4.5;
  double dB2 = 3.2;
  double b;
  b = B(dB1, dB2);

  char str[12] = "Hello ";
  char* tmp;
  tmp = C(str);

  printf("A: %.2f\n", a);
  printf("B: %.2f\n", b);
  printf(tmp);

  return 0;
}

double A(int a, int b)
{
  printf("A");
  double c = a + b;
  c = D(c);
  c--;
  return c;
}
```

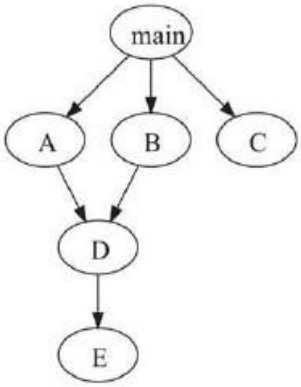
```c
double B(double a, double b)
{
  printf("B");
  double c = a - b;
  c = D(c);
  c++;
  return c;
}

char* C(char* a)
{
  printf("C\n");
  char* str = "World";
  strcat(a, str);
  return a;
}

double D(double a)
{
  printf("D");
  double b = a / 2;
  b = E(b);
  b += a;
  return b;
}

double E(double a)
{
  printf("E");
  double b = a + 1;
  return b;
}
```

(a)



(b)

Figura 1 Ejemplo de aplicación del grafo de llamadas. a) Programa ejemplo. b) Grafo de llamadas del programa ejemplo. Fuente: (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2020).

```c
unsigned char stack[16384];
unsigned short offset = 0;

#define PushStack(object, size){\
memcpy((unsigned char*)\
stack + offset, \
object, size);\
offset += size;}

#define PopStack(object, size){\
offset -= size;\
memcpy(object, (unsigned char*)\
stack + offset, \
size);}

int func_idx;

int main(int argc, char* argv[])
{
  printf("main");
  int iA1 = 5;
  int iA2 = 2;
  func_idx = 1;
  PushStack(&iA2, sizeof(iA2));
  PushStack(&iA1, sizeof(iA1));
  double a;
  D(7); //A();
  PopStack(&a, sizeof(a));

  double dB1 = 4.5;
  double dB2 = 3.2;
  func_idx = 2;
  PushStack(&dB2, sizeof(dB2));
  PushStack(&dB1, sizeof(dB1));
  double b;
  D(1); //B();
  PopStack(&b, sizeof(b));

  char str[12] = "Hello ";
  func_idx = 3;
  void* pstr = (void*)&str;
  PushStack(&pstr, sizeof(pstr));
  char* tmp;
  D(5); //C();
  PopStack(&tmp, sizeof(tmp));

  printf("A: %.2f\n", a);
  printf("B: %.2f\n", b);
  printf(tmp);

  return 0;
}

double A(int a, int b)
{
  PopStack(&a, sizeof(a));
  PopStack(&b, sizeof(b));
  printf("A");
  double c = a + b;
  func_idx = 4;
  PushStack(&c, sizeof(c));
  E(9.3); //D();
  PopStack(&c, sizeof(c));
  c--;
  PushStack(&c, sizeof(c));
  return c;
}

double B(double a, double b)
{
  PopStack(&a, sizeof(a));
  PopStack(&b, sizeof(b));
  printf("B");
  double c = a - b;
  func_idx = 4;
  PushStack(&c, sizeof(c));
  E(19.5); //D();
  PopStack(&c, sizeof(c));
  c++;
  PushStack(&c, sizeof(c));
  return c;
}

char* C(char* a)
{
  switch (func_idx){
  case 3: {
    PopStack(&a, sizeof(a));
    printf("C\n");
    char* str = "World";
    strcat(a, str);
    PushStack(&a, sizeof(a));
    return a;
  }
  case 2:
    B(31.2, 17.6);
    break;
  case 5:
    E(1.7);
  }
}

double D(double a)
{
  switch (func_idx) {
  case 4: {
    PopStack(&a, sizeof(a));
    printf("D");
    double b = a / 2;
    func_idx = 5;
    PushStack(&b, sizeof(b));
    C(NULL); //E();
    PopStack(&b, sizeof(b));
    b += a;
    PushStack(&b, sizeof(b));
    return b;
  }
  case 1:
    A(44, 0);
    break;
  case 2:
    C("test");
    break;
  case 3:
    C("func");
    break;
  }
}

double E(double a)
{
  switch (func_idx) {
  case 5: {
    PopStack(&a, sizeof(a));
    printf("E");
    double b = a + 1;
    PushStack(&b, sizeof(b));
    return b;
  }
  case 4:
    D(13.4);
  }
}
```

(a)

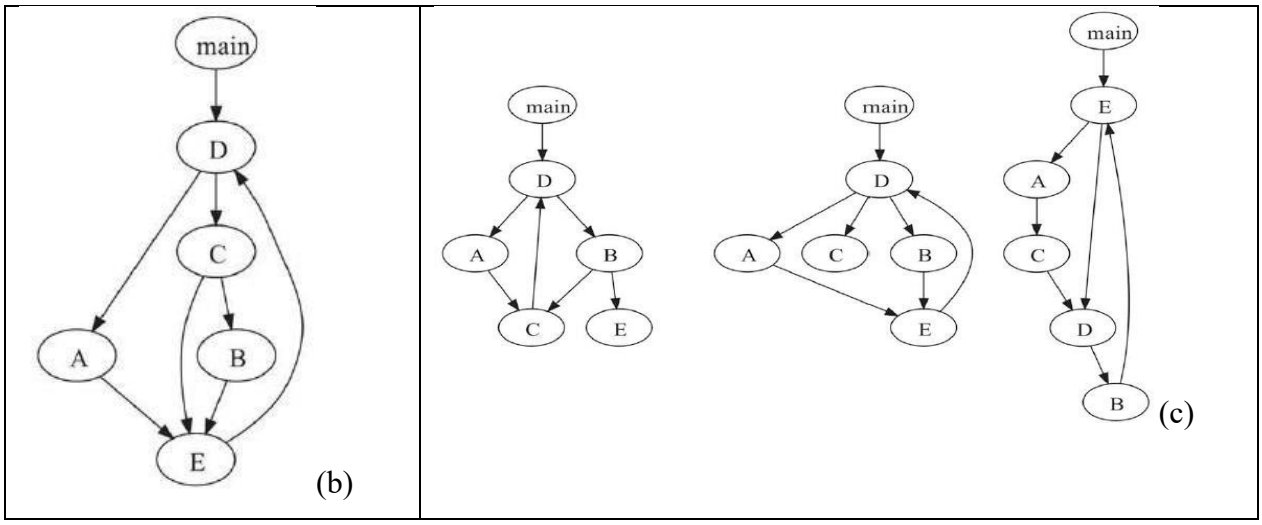


Figura 2 Resultado de la aplicación del algoritmo. a) Programa ejemplo ofuscado. b) Grafo de llamadas del programa ofuscado. c) Otros grafos de llamada. Fuente: (Rodríguez Véliz, Núñez Musa, & Sepúlveda Lima, 2020).

La implementación del modelo de protección propuesto se lleva a cabo de manera totalmente automatizada utilizando la infraestructura de compilación LLVM. El proceso, realizado en tiempo de compilación, implica la compilación de archivos fuente en lenguaje C o C++ (en la investigación se trabajó con este lenguaje y se investiga en códigos desarrollados en otros lenguajes) a representación intermedia en formato bitcode. La fusión de estos archivos en un único programa, la aplicación del modelo de ofuscación mediante el optimizador de LLVM, y finalmente la generación del código binario ejecutable.

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.

CONCLUSIONES

El tema de la ofuscación del código como mecanismo para protegerlo, ha sido objeto de ocupación y preocupación. Los diferentes enfoques para su implementación se encuentran presentes en las investigaciones revisadas, aunque es el grafo de control la técnica que prima entre las tomadas en consideración; no siendo así con el grafo de llamadas que se encuentra en el otro extremo. En todos los casos estas técnicas transforman el código en

uno equivalente funcionalmente, pero oscuro de entender por los elementos que se modifican en dependencia de la o las técnicas que se implementen. Este tipo de seguridad se considera activa pues protege de los ataques mediante la adopción de medidas que protegen los activos de la organización.

A pesar de que no hay garantía de que el código ofuscado sea inmune completamente a ataques de ingeniería inversa, la ofuscación aumenta el esfuerzo requerido para entender la funcionalidad asociada al código ofuscado. La medición de la calidad de la ofuscación es una necesidad del ofuscador para conocer cuán eficaz es su propuesta. El modelo que se describe en este trabajo ha demostrado indicadores que resultan favorables para este especialista y desalentadores para los atacantes.

REFERENCIAS

Ahire, P., & Abraham, J. (2020). Mechanisms for Source Code Obfuscation in C: Novel Techniques and Implementation. International Conference on Emerging Smart Computing and Informatics (ESCI), (págs. 52-59). Pune. doi:https://doi.org/10.1109/ESCI48226.2020.9167661

Al-Hakimi, A. M., Sultan, A. B., Ghani, A. A., Ali, N. M., & Admodisastro, N. I. (2020). Hybrid Obfuscation Technique to Protect Source Code From Prohibited Software Reverse Engineering. IEEE Access, 8, 187326-187342. doi:https://doi.org/20.1109/ACCESS.2020.3028428

Bin Shamlan, A., Alaidaroos, A. S., Bin Merdhah, M. H., Bamatraf, M. A., & AA, Z. (2020). Experimentalevolution of the obfuscation techniques against reverse engineering. Proceedings of ICACI2020. Advances on smart and soft computing, (págs. 382-390). Spring Singapore.

Bin Shamlan, M., Bamatraf, M., & Zain, A. (2019). The Impact of Control Flow Obfuscation Technique on Software Protection Against Human Attacks. 2019 First International Conference of Intelligent Computing and Engineering (ICOICE). Hadhramout. doi:https://doi.org/10-1109/ICOIICE48418.2019.9035187

Céspedes Maestre, M. (2021). Detección de URLs maliciosos por mendio de técnicas de aprendizaje automático. Tesis de maestría, Universidad Nacional de Colombia. Obtenido de https://repositorio.unal.edu.co/handle/unal/79722

Cheng, X., Lin, Y., Gao, D., & Jia, C. (2019). DynOpVm: VM-Based Software Obfuscation with Dynamic Opcode Mapping. Applied Cryptography and Network Security (ACNS 2019, 11464, págs. 155-174. Cham. doi:https://doi.org/10.1007/978-3-030-21568-2_8

Collberg, C., Thomborson, C., & Low, D. (1997). Taxonomy off obfuscation transformations. Reporte técnico #148, Auckland University. Obtenido de https://research.sapace.aunckland.ac.nz/handle/2292/3491

Compton, R., Frank, E., Patros, P., & Koay, A. (2020). Embedding Java Classes with code2vec: Improvements from Variable Obfuscation. Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20). Association for Computing Machinery, (págs. 243-253). New York. doi:https://doi.org/10.1145/3379597.3387445

Ebad, S., Daren, A., & Abawagy, J. (2021). Measuring software obfuscation quality -a systematic literature review. IEEE Access, 9, 99024-99038.

Fell, A., Pham, H. T., & Lam, S. (2019). TAD: Time Side-Channel Attack Defense of Obfuscated Source Code. 24th Asia and South Pacific Design Automation Conference (ASP-DAC), (págs. 1-6). Tokyo. doi:https://doi.org/10.1145/3287624.3287694

Gatica, J., Beron, M., Riesco, D., Pereira, M. J., Henriques, P., & Novais, P. (2023). Protección de activos de software. XXV Workkshop de Investigación en Ciencias de la Computación, (págs. 699-703). Junín. Obtenido de https://sedici.unlp.ed.ar/hadle/10915/164036

Hashemzade, B., & Abdolrazzagh-Nezhad, M. (2019). CSE: A Novel Dynamic Obfuscation Based on Control Flow, Signals and Encryptio . Journal of Computing and Security,, 6, 53-63. doi:https://doi.org/10.22108/jcs.2020.115402.1017

Khan, R. A., Khan, S. U., Khan, H. U., & Ilyas, M. (2022). Systematic literature review on security risks and its practices in secure software development. ieee Access, 10, 5456-5481.

Kumar, A., & Sharma, S. (2019). Design and Implementation of Obfuscating Tool for Software Code Protection. Advances in Interdisciplinary Engineering, (págs. 665–676). Singapore. doi:https://doi.org/10.1007/978-981-13-6577-5_64

Li, Y., Xiong, X., & Zhao, Y. (2021). Code Obfuscation Based on Inline Split of Control Flow Graph. 2021 IEEE International Conference on Artificial Intelligence and Computer Applications (ICAICA), (págs. 632-638). Dalian. doi:https://doi.org/10.1109/ICAICAS2286.2021.9498241

Lv, D., Zhao, L., & Chen, B. (2022). Research Based on LLVM Code Obfuscation Technology. International Conference on Industrial IoT, Big Data and Supply Chain (IIoTBDSC), (págs. 163-167). Beijing. doi:https://doi.org/10.1109/IIoTBDSC57192.2022.00039

Mahoney, W., McDonald, J., Grispos, G., & Mandal, S. (2023). Improvements on Hiding x86-64 Instructions by Interleaving. Proceedings of the 18th International Conference on Cyber Warfare and Security, (págs. 246-255). Maryland. doi:https://doi.org/10.34190/iccws.18.1.987

Morrison, P., Moye, D., Pandita, R., & Laurie, W. (2018). Mapping the Field of Software Life Cycle Security Metrics. Information and Software Technology, 102, 146-159. Obtenido de https://www.researchgate.net/profile/Dr-Rafiq-Khan/publication/341129438_The_State_of_the_Art_on_Secure_Software_Engineering_A_Systematic_Mapping_Study/links/5eb1197292851cb267741f4c/The-State-of-the-Art-on-Secure-Software-Engineering-A-Systematic-Mapping-Study.pdf

Ming Lu, K. (2019). Control flow obfuscation via CPS transformation. Proceedings of the 2019 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM 2019). Association for Computing Machinery, (págs. 54–60). New York. doi:https://doi.org/10.1145/3294032.3294083

Montejano Masa, J. P., Berón, M., Montejano, G. A., & Riesco, D. E. (2023). Métodos, técnicas y herramientas para la protección de sistemas de software. XXV Workkshop de Investigación en Ciencias de la Computación, (págs. 719-723). Junín. Obtenido de https://sedici.unlp.ed.ar/hadle/10915/164036

Moreno, A. (2023). Técnicas de evasión de antivirus y EDR. Tesis de grado, Escuela Técnica Superior de Ingeniería de sistemas de la Universidad Politécnica de Madrid, Madrid. Obtenido de https://oa.upm.es/id/eprint/75850

Quiroz Zambrano, S., & Macías Valencia, D. (2017). Seguridad en informática. consideraciones. Dominio de las Ciencias, 23(5), 676-688. doi:http://dx.doi.org/10.23857/dom-cien.pocaip.2017.3.5.ago.676-688

Roa Buendía, J. (2013). Seguridad informática. Madrid, España: McGraw Hill Education.

Rodríguez Véliz, J., Núñez Musa, Y., & Sepúlveda Lima, R. (2020). Call graph obfuscation and diversification: An approach. IET Information Security, 14(2), 241-252. doi:jttps://doi.org/10.1049/iet-ifs.2019.0216

Rodríguez Véliz, J., Núñez Musa, Y., & Sepúlveda Lima, R. (2023). Study of Code Obfuscation Techniques for the Security of Software Components. Intelligent Systems and Applications in Engineering, 11(10), 913-922. Obtenido de https://ijisae.org/index.php/IJISAE/article/view/3385

Rodríguez Véliz, J. (2025). Modelo para la privacidad de software basada en la diversificación y ofuscación del grafo de llamadas. Tesis de doctorado, Facultad de Ingeniería Informática, Universidad Tecnológica de La Habana.

Stallings, William Brown, Lawrie (2022) Computer security: Principles and practice (5th edition). Pearson Education. Obtenido de https://studylib.net/doc/27914471/computer-security-principles-and-practice-5th-edition---w...

Tambunan, S., & Rokhman, N. (2023). C Source code Obfuscation using Hash Function and Encryption Algorithm. . Indonesian Journal of Computing and Cybernetics Systems (IJCCS), 17, págs. 227-236. doi:https://doi.org/10.22146/ijccs.86118

Wang, L., Li, Y., Zhang, H., Han, Q., & Chen, L. (2021). An Efficient Control-flow based Obfuscator for Micropython Bytecode. 2021 7th International Symposium on System and Software Reliability (ISSSR), (págs. 54-63). Chongqing. doi:https://doi.org/10.1109/ISSSR53171.2021.00028

Wang, Z., Shan, Y., Yang, Z., Wang, R., & Song, S. (2020). Semantic Redirection Obfuscation: A Control flow Obfuscation Based on Android Runtime. 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom),, (págs. 1756-1763). doi:https://doi.org/10.1109/TrustCom50675.2020.00241

Whitman, Michael. Mattord, Herbert (2022). Principles of Information Security (7th edition). Cengage Learning.

Xu, H., Zhou, Y., Ming, J., & Lyu, M. (2020). Layered obfuscation: a taxonomy of software obfuscation tecniques for layered security. Cibersegurity, 3, 1-18. Obtenido de https://link.springer.com/article/10.1186/542400-020-00049-3

Yang, X., Zhang, L., Ma, C., Liu, Z., & Peng, P. (2019). Android Control Flow Obfuscation Based on Dynamic Entry Points Modification. 22nd International Conference on Control Systems and Computer Science (CSCS), (págs. 296-303). Bucharest. doi:https://doi.org/10.1109/CSCS.2019.00054

Yi, J., Chen, L., Zhang, H., Li, Y., & Zhao, H. (2020). A Security Model and Implementation of Embedded Software Based on Code Obfuscation. En 2020 (Ed.), 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), (págs. 1606-1613). Guangzhou. doi:https://doi.org/10.1109/TrustCom50675.2020.00222

Copyright © 2025, Autores: Rodríguez Véliz, Miguel, Hernández González, Anaisa, Sepúlveda Lima, Roberto, Núñez Musa, Yulier

Esta obra está bajo una licencia de Creative Commons Atribución-No Comercial 4.0 Internacional