Utilización de Excepciones para Implementar Predicados Opacos en Técnicas de Ofuscación de Código Intermedio

Daniel Dolz
daniel_jose_dolz@yahoo.com.ar

Gerardo Parra
gparra@uncoma.edu.ar

Departamento de Ciencias de la Computación
Facultad de Economía y Administración
Universidad Nacional del Comahue
Buenos Aires 1400 - 8300 Neuquén - Argentina
Tel/Fax (54) (299) 4490312/313

Abstract

Microsoft's .NET Framework, and JAVA platforms, are based in a just-in-time compilation philosophy. Software developed using these technologies are executed in a hardware independent framework, which provides a full object-oriented environment, and in some cases allows the interaction of several components written in different programming languages.
This flexibility is achieved by compiling into an intermediate code which is platform independent. Java is compiled into ByteCode, and Microsoft .NET programs are compiled into MSIL (Microsoft Intermediate Code). However, this flexibility comes with a price. It is really easy, with tools available for free in the web, to decompile this intermediate code and obtain a working, readable version of the original source code.
Of all techniques developers can use to protect their intellectual property, obfuscation is the most accepted and commercially available one.
In the present work, we propose the use of try-catch mechanisms available in .NET as a way to improve the quality of one of the building blocks of obfuscation: opaques predicates.

Keywords: Obfuscation. Obfuscation Transformation. Opaque Predicates.

Resumen

La plataforma .NET de Microsoft se basa en una filosofía de *just-in-time compilation* (compilación bajo demanda al momento de la ejecución). Los programas desarrollados de esta manera se ejecutan en un entorno o framework independiente de la plataforma, basado en objetos y, en algunos casos, permitiendo que interactúen componentes desarrollados en distintos lenguajes de programación.
La clave de esta flexibilidad se da en que, tanto en JAVA como en las plataformas .NET, la compilación resulta en un código intermedio, independiente de la plataforma (bytecode y MSIL respectivamente). Sin embargo, dicha flexibilidad tiene un costo. Hoy en día, y utilizando herramientas gratuitas que pueden descargarse desde Internet, es sumamente fácil

aplicar tecnologías de ingeniería inversa a las dos plataformas de desarrollo más populares: JAVA y .NET.
De todas las técnicas que los desarrolladores pueden utilizar para proteger su propiedad intelectual, la ofuscación es la técnica más aceptada y de hecho, es la única utilizada comercialmente.
En este trabajo, proponemos el uso de los mecanismos de excepción (bloques try-catch) que brinda la plataforma .NET como una manera de mejorar la calidad de uno de los bloques básicos de la ofuscación, los predicados opacos.

PALABRAS CLAVES: Ofuscadores. Código Intermedio. Transformaciones de Ofuscación. Predicados Opacos.

1 Introducción

Tanto las aplicaciones JAVA, como aquellas desarrolladas para ejecutarse sobre cualquiera de las versiones del Framework de .NET, poseen la vulnerabilidad de que, con herramientas gratuitas y con solo conocimientos básicos de informática, es posible para cualquier persona que posea los distribuíbles de la aplicación obtener, de manera completa y con solo ligeras variaciones, el código fuente original completo de la aplicación.
La apropiación del código fuente de un software desarrollado por una organización en manos de personal no autorizado y con intenciones evidentemente ilegales, podría tener las siguientes consecuencias:

- Pérdida a manos de la competencia del dinero invertido en I+D[1]. La competencia puede, de forma desleal, lanzar al mercado el mismo producto, con un "lavado de cara" y aprovechando la inversión de la organización original.
- Una organización competidora podría descubrir fallas en el producto y utilizarlas en su beneficio.
- En el caso puntual de los algoritmos de encriptación modernos, cuya seguridad está basada en la existencia de una clave desconocida y no de un algoritmo en particular, el acceso al algoritmo por parte de manos malintencionadas podría servir para, previa modificación de los mismos, intentar ataques de fuerza bruta contra los datos cifrados.
- El acceso al código fuente de una aplicación facilita el "crackeo" de sistemas anti piratería, como ser la registración de software mediante *keys, expiration dates, hardlocks*, etc.
- Un empleado con conocimientos de informática podría, descompilar la aplicación de gestión administrativa de la empresa, modificar los strings de selección a las bases de datos de manera de eliminar restricciones y filtros, recompilarla, ejecutarla, y obtener acceso irrestricto a la base de datos de clientes. Esto podría resultar en la pérdida de valiosos secretos comerciales.

Lo anterior indica problemas económicos, pero algunos gobiernos como el de los EEUU identifican al problema como de seguridad nacional [4]. En el ámbito privado, se sabe que el 75% de las empresas Fortune 500[11] utilizan de una manera u otra el paquete de desarrollo Microsoft Visual Studio 2005[5]. Viendo que las principales amenazas a la seguridad provienen, hoy por hoy, no de agentes externos a las organizaciones sino de elementos internos de la misma con acceso a los recursos de la empresa desde adentro (empleados, personal contratado, consultores, etc.), el problema no es menor [6].

[1] I+D: Investigación y Desarrollo

Nuestra línea de trabajo, iniciada en [12], muestra a las técnicas de ofuscación como la rama de la seguridad informática que puede brindar un nivel de protección superior al de las alternativas existentes. En este artículo mostramos cómo, mediante la utilización de excepciones, es posible incrementar de manera notoria la calidad de los predicados opacos, mejorando en consecuencia la calidad de la ofuscación y por ende la protección de la propiedad intelectual.

La estructura del trabajo es la siguiente. A continuación, se brinda una descripción introductoria de la ofuscación y de sus conceptos claves. Luego, se profundiza en las construcciones denominadas predicados opacos. En la sección 4, presentamos el aporte de este trabajo. Se describen las técnicas de generación de predicados opacos más avanzadas y se presenta la innovación de reemplazar el uso de sentencias de salto condicional por bloques del tipo *try-catch*. Finalmente, en la sección 5, se reportan las conclusiones y se presentan algunas líneas de trabajo futuro.

2 Ofuscación, conceptos claves

En términos generales, se entiende por ofuscar un código fuente o un código intermedio, un proceso mediante el cual se transforma utilizando diversos algoritmos de reescritura, un código perfectamente legible y entendible por una persona en otro de funcionalidad equivalente en un ciento por ciento, pero, en términos ideales, totalmente ilegible e incomprensible para un lector humano.
En la figura 1 se muestra, de manera esquemática, el proceso y concepto de aplicación de ofuscación de código intermedio.

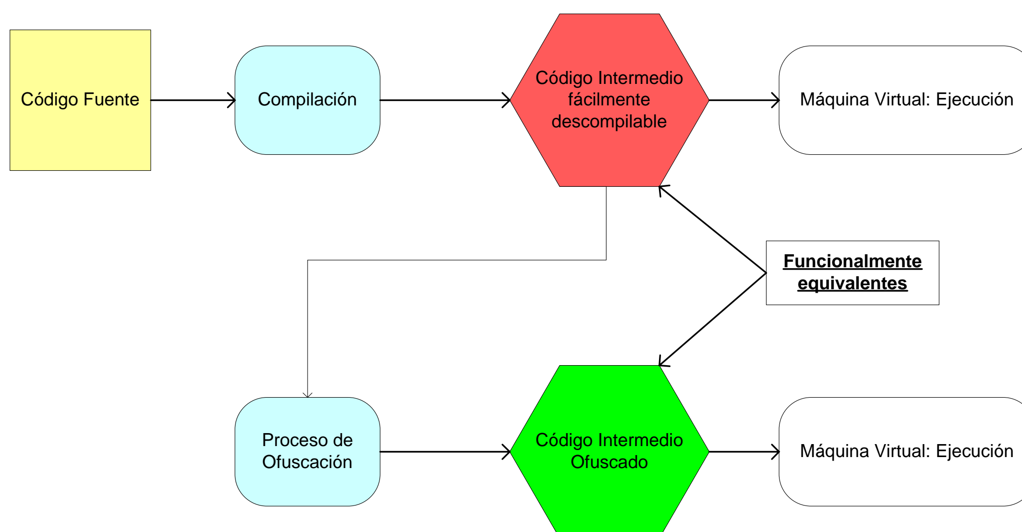

Figura 1: Ofuscación de Código Intermedio

En pocas palabras, algunas de las técnicas de ofuscación más comunes consisten en la inclusión de bucles irrelevantes, cálculos innecesarios, comprobaciones fuera de contexto, nombres de funciones y de variables que no tienen nada que ver con su cometido, funciones que no sirven para nada, interacciones inverosímiles entre variables y funciones, etc. Otras técnicas, sin embargo, son mucho más potentes, en el sentido de que requieren un conocimiento superior de las características del lenguaje, e incluso pueden estar diseñadas para burlar a herramientas de ingeniería inversa específicas.
A continuación, discutimos conceptos claves presentados por Collberg[1].

2.1 Transformación de ofuscación

Sea $P \xrightarrow{\tau} P'$ la transformación de un código fuente o intermedio P en un código fuente o intermedio P’

$P \xrightarrow{\tau} P'$ es una transformación de ofuscación si P y P’ tiene el mismo comportamiento observable, entendido desde el punto de vista de lo que percibe el usuario.

Más precisamente, para que τ sea una transformación válida se debe verificar que:

- Si P falla al terminar dada una entrada, P’ podría o no terminar.
- De otra forma, dada una entrada, P’ debe terminar y producir el mismo resultado que P.

Idealmente, P’ debería tener características que dificultan la compresión del código fuente. Se define, de manera informal, a comportamiento observable como la percepción del usuario acerca de la entrada y salida del usuario. P’ podría tener comportamientos diferentes, cuyo objetivo es confundir a un posible desofuscador y/o a un hacker, que serían válidos mientras no sean parte de la experiencia del usuario de P’.

Las transformaciones de ofuscación pueden clasificarse en cuatro categorías o tipos [1]:

- Léxicas o de Estructura: renombramiento de identificadores, cambio de formatos.
- Ofuscaciones de Datos: encriptación de recursos embebidos, encriptación de metadatos, encriptación de cadenas almacenadas, modificación de jerarquías, unificación de variables.
- Ofuscaciones de Control: reconversión de flujos de control, reordenamientos de sentencias, bucles y expresiones, extensión de condiciones de loop.
- Ofuscaciones Preventivas: destinadas exclusivamente a provocar malfuncionamiento a herramientas de descompilación.

2.2 Calidad de una Transformación de Ofuscación

La calidad de una transformación de ofuscación se evalúa según los siguientes criterios: cuanto más difícil se vuelve de entender el código fuente por un lector humano (potencia), qué tan difícil resulta para una herramienta automática revertir la transformación (resistencia), qué tan bien las modificaciones introducidas se disimulan o mezclan con el resto del programa (stealth) y cuanto costo extra computacional se agrega a raíz de la aplicación de la transformación (costo).

2.2.1 Potencia

La potencia de una transformación indica, de una manera orientativa, en qué medida el código ofuscado es más difícil de comprender por un lector humano. Si bien el concepto “más difícil de comprender” no puede ser cuantificado objetivamente, se utilizan métricas de la ingeniería de software que miden la claridad conceptual y la mantenibilidad de un código como referencia.
A efectos prácticos se clasifica la potencia de cualquier transformación en baja, media y alta.

2.2.2 Resistencia

Un atacante merecedor de consideración seguramente dispondrá de herramientas configurables de ingeniería inversa, llegando al extremo de desarrollar las mismas y poder adaptarlas a las técnicas de ofuscación que vaya detectando. Debido a esto, la *resilience* o resistencia puede expresarse como la combinación de dos medidas:

Esfuerzo de Programación: la cantidad de tiempo que llevaría construir una herramienta de ingeniería inversa que efectivamente pueda reducir la potencia de una transformación τ.

Esfuerzo de Desofuscación: los recursos (tiempo / espacio) necesarios para que dicha herramienta efectivamente reduzca la potencia de una transformación τ.

Es importante entender la distinción entre potencia y resistencia. Una transformación es potente si logra confundir a un lector humano, mientras que es resistente si torna difícil la construcción de un desofuscador o hace que la ejecución del mismo se torne impráctica en entornos reales.

Las transformaciones más resistentes son aquellas que son irreversibles. Consisten generalmente en la eliminación de información presente en el programa pero que no es necesaria para la ejecución del mismo, como los nombres significativos de identificadores, entre otros.
Otras transformaciones, como el agregado de código basura que no afecta el comportamiento observable del programa, podrían ser revertidas con distintos niveles de dificultad.

2.2.3 Stealth

Es posible crear técnicas de ofuscación que modifiquen un programa de manera de hacerlo muy difícil de comprender (alta potencia) y que a su vez no sea fácil extraerlas para obtener el código original (alta resistencia).
Un ejemplo podría ser modificar la codificación de valores de variables. En vez que asignar a una variable el valor escrito en el código fuente, una técnica de ofuscación podría asignar valores enormes (del rango de los millones) y aplicar las mismas fórmulas a los valores con los que interactúa esta variable. De esta manera, comparaciones simples del tipo `while (I <= 10)` podrían transformarse en `while ((I * f(I) – 234)^12 <= 5748951478)` siendo la ejecución equivalente.
Este último código ofuscado, sin embargo, salta a la vista como sintético y forzado, y un atacante experimentado lo identificará enseguida como resultado de la aplicación de una técnica de ofuscación. Para mejorar el stealth, el código incorporado por un ofuscador debería parecerse lo más posible al código original, lo cual es un desafío dado que un código que podría ser *stealthy* en un programa podría no serlo en otro de estilo y dominio diferente.

2.2.4 Costo

La aplicación de muchas técnicas de ofuscación, como la del ejemplo de la definición de stealth, implican de manera clara un overhead en el tiempo de la ejecución del programa debido a mayores operaciones (overhead termporal). Otro tipo de ofuscaciones podrían aumentar los requerimientos de recursos espaciales (típicamente memoria) de un programa ofuscado con respecto a su versión original, mientras que algunas transformaciones no incluyen overhead, como el renombrado de identificadores. A medida que aumenta el costo de una transformación disminuye la calidad de la misma.

3 Predicados Opacos

El presente trabajo se enfoca en los predicados opacos, que son el bloque básico de las transformaciones de ofuscación que oscurecen el programa modificando el flujo de control del mismo[2].
Las transformaciones de flujo de control generalmente realizan alguna de estas tres acciones:

- Ocultar el verdadero flujo de control de un programa entre sentencias irrelevantes que no contribuyen a la ejecución del programa.
- Introducir, en el código intermedio, secuencias de control sin correspondencia en el lenguaje de alto nivel en el que originalmente fue escrito el programa.
- Remover construcciones reales del flujo de control y/o introducir construcciones falsas.

Los predicados opacos son expresiones que no pertenecen al programa a ofuscar sino que son introducidos por el ofuscador. La realidad indica que, es el predicado opaco el que realmente hace que se ejecute el código que el programador pretendía ejecutar, y no el código basura o irrelevante insertado por la herramienta de ofuscación.
Informalmente, una variable *V* es opaca si tiene alguna propiedad que es conocida a priori por el ofuscador, pero es difícil de deducir para un ingeniero inverso.
Lo mismo puede decirse para un predicado *P* cuyo valor booleano es conocido por el ofuscador, pero no por el ingeniero inverso.
Por ejemplo, una variable opaca *V* introducida por el ofuscador de valor "10" puede usarse para generar expresiones verdaderas o falsas preguntando, por ejemplo si `V == 10`, si `V < 6`, etc. El ofuscador conoce el valor de *V* en cada momento dado que es el encargado de asignarla. Sin embargo, cuanto más difícil sea para el ingeniero inverso deducir que en tal punto del programa *V* tiene valor "10", mejor funcionará el predicado opaco.
La creación de predicados opacos que sean difíciles de deducir por el ingeniero inverso es uno de los desafíos más importantes para el creador de herramientas de ofuscación. De hecho, los predicados opacos son la clave para la resistencia de las transformaciones de control.
Utilizaremos las mismas medidas que han sido definidas para transformaciones (potencia, resistencia, stealth y costo) para los predicados opacos.

3.1 Uso de Predicados Opacos

Los predicados opacos son claves en las transformaciones de ofuscación que consisten en:

- Inserción de código muerto o irrelevante. El código muerto nunca debe ejecutarse. Su existencia solo tiene por objeto confundir a un posible atacante. La no ejecución de código muerto se deja en manos de un predicado opaco. Por ejemplo, el código muerto podría situarse en el bloque *`else`* de un *`if`* cuya expresión sea un predicado opaco que evalúa siempre a verdadero.
- Extensión de Condiciones de loop. Es posible oscurecer un bucle en el programa haciendo más compleja su condición de terminación. La idea consiste en utilizar predicados opacos de valor conocido para extender la expresión que determina el bucle.
- Conversión de flujo de control de Reducible a No Reducible: los lenguajes en los cuales se basa la necesidad de la existencia de la ofuscación, como son el Java y aquellos de la plataforma .NET, son compilados o traducidos a un lenguaje intermedio, denominados bytecode y MSIL respectivamente. La característica fundamental de estos lenguajes intermedios es que son más poderosos que los lenguajes originales. Esto debe ser así, debido a que no es posible que existan construcciones en los lenguajes de alto nivel que no puedan ser transformadas a código intermedio. La idea es, construir en lenguaje intermedio un flujo de control que no tenga equivalente de alto nivel, pero preservando la ejecución correcta mediante el uso de predicados opacos. Un ejemplo sería una bifurcación condicional a una sentencia que esté en el medio de un bloque `while`, protegida mediante un predicado opaco que evalúa siempre a falso de manera que la verdadera ejecución no es alterada.

3.2 Construcción de Predicados Opacos

Los predicados opacos son vitales hasta el punto que la resistencia de las transformaciones está relacionada directamente con la calidad de dichos predicados

Los predicados obvios que podrían pensarse, como `P == 0`, `Q != null`, tienen resistencia a lo sumo, débil, siendo la mayoría triviales. Esto significa que un desofuscador automático podría deducir su valor mediante un análisis estático, local o global, sin insumir mucho esfuerzo. Obviamente, es necesario un nivel de protección mayor, identificando expresiones opacas cuyo esfuerzo requerido, en el peor caso, sea exponencial con respecto al tamaño del programa, pero que solo requiera un tiempo lineal o polinomial para construirlos.

3.3 Técnicas avanzadas

Existen varias técnicas avanzadas de construcción de predicados opacos. Sin embargo, tienen dos problemas fundamentales. Uno de ellos es el costo y el otro es la visibilidad del salto del flujo de control.

A continuación, describimos las técnicas más avanzadas de construcción de predicados opacos.

3.3.1 Construcción de Predicados Opacos usando Objetos y Alias

Los análisis estáticos de cualquier tipo sobre un programa se tornan significativamente costosos cuando existe la posibilidad de que existan alias para los objetos. Se ha demostrado que el análisis estático de código fuente cuando existe aliasing de objetos es NP-Hard [9] o incluso indecidible [10].

La idea básica es construir una estructura dinámica compleja con punteros y alias y mantener un conjunto de punteros a esta estructura. De esta manera, el ofuscador sabe si p es igual a q (siendo p y q punteros de una estructura) pero un desofuscador no podría saberlo realizando un análisis estático. En la figura 2, mostramos un posible ejemplo de generación de predicados opacos.

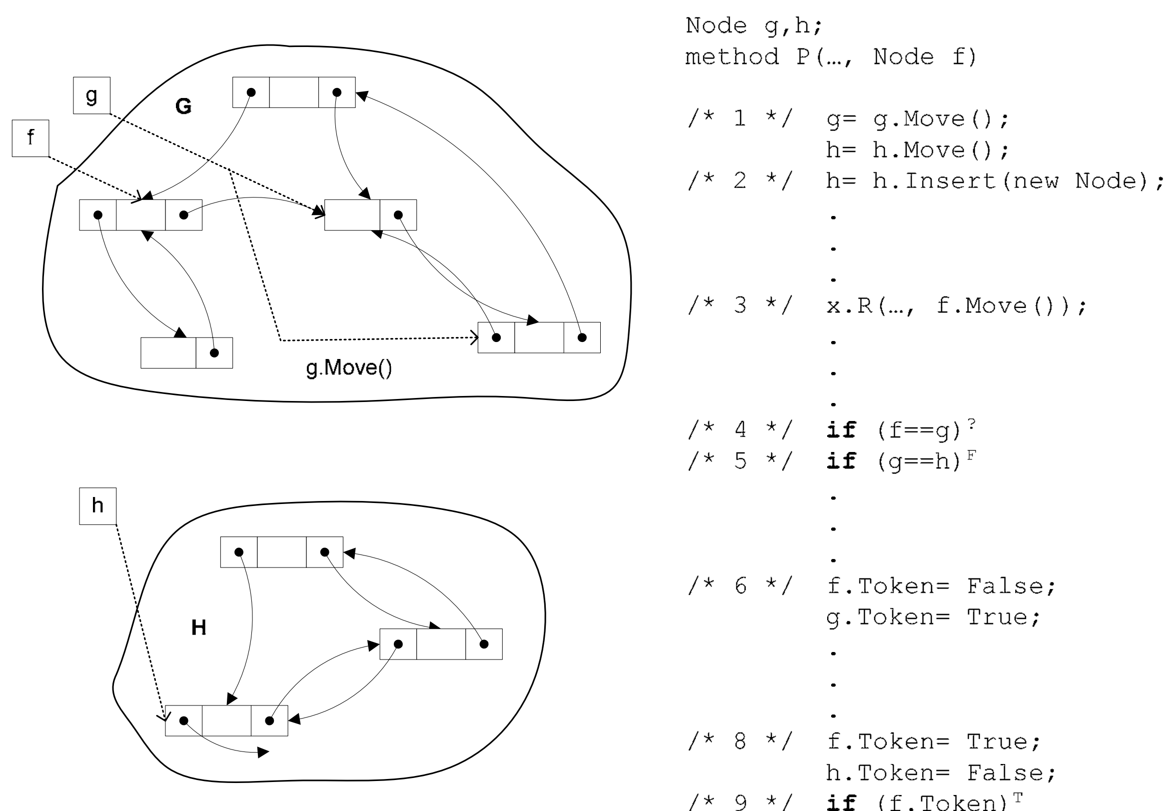

Figura 2: Predicados opacos construidos a partir de objetos y alias.

Se construye una estructura dinámica formada por Nodos. Cada nodo cuenta con un campo booleano Token y dos campos puntero que pueden apuntar a otros nodos. La estructura está diseñada por dos componentes conectados, G y H. Existen dos punteros globales, g y h, que apuntan a G y a H respectivamente.

3.3.2 Construcción de Predicados Opacos usando Threads

Los programas paralelos o multiproceso son mucho más difíciles de analizar estáticamente que sus contrapartidas secuenciales. La razón es el intercalamiento: si *n* segmentos de código pueden ejecutarse de manera paralela, la cantidad de formas en las que puede ejecutarse es *n!*. Si *n* es grande, rápidamente puede intuirse la dificultad de realizar un análisis estático.
La técnica es la misma que la del punto anterior, pero sumando la complejidad que brinda el multithreading.

3.4 Problemas detectados

Las técnicas analizadas arriba son las técnicas de construcción de predicados opacos más avanzadas del estado del arte. Sin embargo, en este trabajo sugerimos que no son prácticas ni viables porque implican un alto costo y no logran el stealth que sería necesario para justificarlo.

3.4.1 Costo

En ambas técnicas (Objetos-Alias y Multithreading) se observa la creación y el mantenimiento de una estructura de grafos que es totalmente ajena al programa original solo al efecto de preguntar, en algún momento, si `P == Q` o alguna pregunta similar. Puede verse que la técnica de objetos-alias

implica un costo temporal (el tiempo de procesamiento requerido para generar y mantener la estructura) y también un costo espacial (la cantidad de memoria).
Para el caso del multithreading, que consiste en generar lo mismo mediante threads, hay que considerar también el overhead de los cambios de contexto.
Además, en ambos casos, hay una consideración práctica: los programas suelen fallar. Cuando esto ocurre, los entornos informan al usuario con un *dump* del *stack trace*, lo cual suele ser útil al desarrollador para identificar las causas de la falla. En el caso de una falla generada por un error del programa es posible que el *stack trace* esté contaminado por las llamadas del ofuscador, lo que también es un problema.

3.4.2 Visibilidad

En este trabajo sugerimos que, en realidad, no es tan importante la complejidad del cálculo del predicado opaco sino su visibilidad.
Es posible mantener una estructura enorme, compleja y cara para que el ingeniero inverso no pueda nunca llegar a saber si `P == Q`, pero eventualmente este ingeniero inverso detectará que `P` y `Q` no tienen nada que ver con el programa que está analizando y llegará a la conclusión de que **`P == Q` es un predicado opaco,** por más que no pueda conocer su valor en cada momento que es invocado.
El hecho de identificar predicados opacos ya es una información invaluable a los efectos de desofuscar el programa, dado que el ingeniero sabe que algunos de los dos casos (verdadero o falso) esconde código resultado de transformaciones, y ese conocimiento podría ser suficiente para lograr los propósitos de la ingeniería inversa.
Una herramienta inicial para detectar predicados opacos es, sencillamente, identificar a todas las sentencias de salto condicional del código intermedio, sabiendo que algunas corresponden a la lógica del programa y que otras son introducidas por el ofuscador.
Esto es casi trivial. De hecho, la herramienta ildasm.exe provista por Microsoft que extrae el código intermedio a partir de ejecutables portables .NET ya marca con un espacio aquellas instrucciones que son un salto, condicional o no, haciendo muy fácil identificar los predicados opacos potenciales.

4 Predicados Opacos Superiores: utilización de bloques try-catch-finally

En esta sección, introducimos nuestra propuesta para generar predicados opacos de mayor calidad.

4.1 Bloques Try-Catch-Finally

Los lenguajes sobre los que trata este trabajo, implementan el manejo de excepciones basado en bloques *try-catch-finally*. No es el objetivo de este trabajo describir en detalle esta construcción. Basta comentar que *try, catch y finally* son palabras clave que delimitan bloques disjuntos de código de manera que, si ocurre una excepción, de cualquier tipo, dentro del bloque *try* (la excepción puede ocurrir incluso en otro método de otra clase, o incluso en otro módulo que puede estar en un archivo binario distinto) la ejecución se deriva, automáticamente, a la primera sentencia del bloque *catch*. Si no ocurre ninguna excepción, el bloque *catch* no se ejecuta.
El bloque *finally*, que es optativo, se ejecuta en todos los casos y suele utilizarse para realizar tareas de limpieza que deben realizarse tanto si ocurrió una excepción, como si no. El ejemplo clásico de un bloque *finally* consiste en el cierre de una conexión a una base de datos.

4.2 Predicados Opacos utilizando Excepciones

La propuesta de este trabajo consiste en la utilización de predicados opacos simples y de relativamente poco costo (como `q == 0`, `p == null`) pero prescindiendo del uso de una sentencia de salto condicional para implementarlo y, en cambio, forzando una excepción.
De esta manera, si bien los valores son simples, a un ingeniero inverso le resultaría difícil identificar los posibles lugares donde efectivamente ocurre el salto condicional mediante un análisis estático.
A continuación mostramos ejemplos utilizando el lenguaje de código intermedio de .NET, llamado MSIL. Está fuera del alcance de este trabajo explicar el funcionamiento del MSIL, remitiendo a las especificaciones de la ECMA al respecto [7, 8].

Ejemplo: Predicado Opaco utilizando una sentencia IF. Corresponde a un if (Predicado Opaco False) then (código real) else (código bogus)

```
IL_0000:  ldarg.1
IL_0001:  brtrue.s   IL_003e          /*** salto condicional ***/
      ........... /* Código real de la aplicación */
IL_0033:  ldstr      "Codigo real"
IL_0038:  call       void [mscorlib]System.Console::Write(string)
IL_003d:  ret
      ........... /* Código falso */
IL_003e:  ldstr      "Codigo Bogus introducido por el ofuscador"
IL_0043:  call       void [mscorlib]System.Console::Write(string)
IL_0048:  ret
```

Nótese la línea IL_0001 (resaltada), con la sentencia de salto condicional *brtrue* sobre el argumento 1 del método (ldarg.1). Es importante destacar que, por muy complicada que sea la manera en la que el ofuscador oculta que ese parámetro es cero, es suficiente con saber que el predicado opaco está allí, quedando solamente analizar los bloques *then* y *else* y el comportamiento de los mismos.

Ejemplo: Predicado Opaco Utilizando Bloques *Try-Catch*

```
IL_0000:  ldc.i4.0
IL_0001:  stloc.0
.try
{
      ........... /* Código real */

  IL_0032:  ldstr      "Codigo real"
  IL_0037:  call       void [mscorlib]System.Console::Write(string)
  IL_003c:  ldloc.0
  IL_003d:  ldarg.1
  IL_003e:  div
      ........... /* Código falso */
  IL_003f:  call       string [mscorlib]System.Convert::ToString(int32)
  IL_0044:  call       void [mscorlib]System.Console::Write(string)
  IL_0049:  ldstr      "Codigo Bogus introducido por el ofuscador"
  IL_004e:  call       void [mscorlib]System.Console::Write(string)
  IL_0043:  leave.s    IL_0032
}  // end .try
catch [mscorlib]System.Object
{                    ........... /* Código real */

  IL_0055:  pop
  IL_0056:  ldstr      "Codigo real"
  IL_005b:  call       void [mscorlib]System.Console::Write(string)
  IL_0050:  leave.s    IL_0032
}  // end handler
IL_0052:  ret
```

Un ejercicio interesante podría ser intentar identificar en que instrucción se encuentra el predicado opaco sabiendo que el valor del parámetro es, al igual que en el caso anterior, un entero de valor cero.
La respuesta es, en la línea *IL_003e: div*. Lo que esta sentencia hace es provocar división por cero. El resultado neto es, entonces, la continuación de la ejecución en el bloque *catch* donde reside el resto del código real.
Lo importante de este esquema es que el error inducido puede estar en cualquier parte dentro de un bloque de código, sin tener como requerimiento el uso de una sentencia de salto condicional que son fácilmente identificables. Esto convierte al predicado opaco en una sentencia más, no identificable estáticamente.
Algunas de las excepciones que podrían inducirse son división por cero, uso inválido de *null*, error de conversión de tipos, valores fuera de rango, *cast* inválidos, entre otros.

4.3 Predicados Opacos utilizando Excepciones con Stealth mejorado

Intentemos llevar este concepto aún más allá. Dado que la bifurcación del flujo de control que puede manipular el ofuscador surge de la generación inducida y controlada de un error en *runtime*, un buen ofuscador podría utilizar construcciones comunes del programa siendo ofuscado e inducir errores gracias a los valores opacos conocidos, pero generando estructuras exactamente iguales a la que utilizó el programador de la aplicación.

4.3.1 Ejemplo: uso inválido de null

Supongamos que el ofuscador detecta que el programa a ofuscar hace uso intensivo de objetos que son instancias de una clase que llamaremos *CACIC* y un método llamado *Compartir()*. Esto quiere decir que, las sentencias en IL del tipo *A.Compartir*, siendo *A* una variable instancia de la clase *CACIC* son comunes y frecuentes.
El ofuscador podría entonces ingresar predicados opacos de la forma *A.Compartir()*, pero en un momento en el cual conoce que *A* tiene valor nulo.
La ofuscación del flujo de control se realiza perfectamente: para el ingeniero inverso es todo un bloque de código coherente con construcciones normales y muy utilizadas, como por ejemplo *A.Compartir()*.
Sin embargo, desde el momento que *A* es un valor nulo, la ejecución se bifurca hacia el bloque *catch*, lo cual es muy difícil de determinar automáticamente con un análisis estático, y es muy difícil de detectar con un análisis visual del código, dado que el salto se produce en una sentencia totalmente común al programa. En consecuencia, éste predicado goza de altos niveles de stealth.

4.3.2 Ejemplo Dos: colaboración con el programador.

Si la aplicación utilizara de manera intensiva acceso a datos, con una pequeña colaboración del desarrollador, la excepción podría ser generada mediante llamados SQL erróneos. Esto confundiría aún más a un ingeniero inverso desprevenido, ya que es, a priori, impensable que un mecanismo automático como un ofuscador introduzca sentencias de llamado a datos.

4.4 Algunas limitaciones

Este esquema presenta algunas limitaciones. Una de ellas consiste en que los bloques *try-catch-finally* pueden anidarse pero nunca solaparse, de manera que no es posible realizar cualquier construcción arbitraria. Otra limitación es, claramente, que es necesario respetar los bloques *try-catch-finally* que son verdaderamente parte del programa, respetando el funcionamiento del mismo.

5 Conclusiones

La ofuscación es la técnica estándar para la protección del código fuente en ambientes de desarrollo modernos. Dentro de las técnicas de ofuscación, los predicados opacos son el bloque de construcción básico y su fortaleza determina en gran medida la calidad de las transformaciones de ofuscación aplicadas.
Hemos analizado las técnicas de generación de predicados opacos más avanzadas, como la utilización de estructuras utilizando alias y multithreading y concluímos que por su elevado costo, tanto temporal como espacial, no son la solución óptima.
La propuesta de este trabajo ha sido la creación de predicados opacos utilizando el mecanismo de manejo de excepciones como una manera de generar predicados de mayor calidad mediante la reducción del costo y el aumento del stealth.
Como líneas de trabajo futuro podemos mencionar el estudio en mayor detalle del uso de características avanzadas de los ambientes modernos para la protección de la propiedad intelectual, como podrían ser el uso de atributos personalizados y el mencionado manejo de excepciones. También hay un gran campo por explorar en el estudio y análisis del concepto de "funcionalidad equivalente" entre un programa sin ofuscar y un programa ofuscado.

REFERENCIAS

[1] Christian S Collberg, Clark Thompson. Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection,

[2] Christian S Collberg, Clark Thompson, Douglas Low. Manufactoring Cheap, Resilient, and Stealthy Opaque Construct

[3] Willy Alexánder Marroquín. Ofuscadores (De la protección relativa del código intermedio), http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art146.asp

[4] Mr. Jeff Hughes, Dr. Martin R. Stytz, Ph.D. Advancing Software Security– The Software Protection Initiative, 2001

[5] Microsoft Software Developer Network, Microsoft Visual Studio 2005

Evaluation Guide. http://msdn.microsoft.com/vstudio/tour/evaluation/default.htm

[6] Revista Information Technology. Suplemento Especial Seguridad, Junio 2005.

[7] The Common Language Infrastructure (CLI) Partition II*: Metadata Definition and Semantics* http://msdn.microsoft.com/net/ecma/; http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[8] The Common Language Infrastructure (CLI) Partition III: *CIL Instruction Set* http://msdn.microsoft.com/net/ecma/; http://www.ecma-international.org/publications/standards/Ecma-335.htm.

[9] S. Horwitz. Precise flow insensitive May-Alias analysis is NP-Hard. TOPLAS, 19(1):1-6, 1997.

[10] G. Ramalingam. The undecidability of aliasing. TOPLAS, 16(5):1467-1471, Septiembre 1997.

[11] Fortune 500. http://en.wikipedia.org/wiki/Fortune_500

[12] D. Dolz, G. Parra. Ofuscadores de Código Intermedio. Reporte Preliminar. *WICC* 2006.