Revista Cubana de Ciencias Informáticas
Vol. 9, No. Especial CCIA 17, 2015
RNPS: 2301 | ISSN: 2227-1899
http://rcci.uci.cu
Pág. 19-34

Tipo de artículo: Artículo original
Temática: Seguridad informática
Recibido: 05/01/2015 | Aceptado: 15/02/2015

Arquitectura extensible para la protección automatizada de software: Un caso de estudio

Extensible architecture for the automated software protection: A study case

Yulier Nuñez Musa[1*], Miguel Bolívar Rodríguez[1], Humberto Díaz Pando[1], Roberto Sepúlveda Lima[1].

[1*] Instituto Superior Politécnico José Antonio Echeverría. Avenida 114 entre Ciclovía y Rotonda, Marianao, Habana, Cuba. {jnunezm, mbolivar, hdiazp, sepul}@ceis.cujae.edu.cu

*** Autor para la correspondencia:** jnunezm@ceis.cujae.edu.cu

---

Resumen

En la actualidad el software comercial es susceptible a la modificación y la observación de su código interno mediante ataques de ingeniería inversa. Estos ataques permiten la piratería del software, siendo billonarias las pérdidas ocasionadas a la industria de software por este concepto. Este trabajo se centra en el desarrollo de una arquitectura extensible para insertar de forma automatizada durante el proceso de compilación, mecanismos de protección en un software dado. La protección se realiza de forma transparente al desarrollador y además, permite diseñar e implementar de forma flexible y modular las diferentes técnicas de protección identificadas. Se implementó la técnica de ofuscación como caso de estudio para validar la arquitectura propuesta. Esta técnica fue probada sobre distintos algoritmos de pruebas, mostrándose los resultados obtenidos.
Palabras clave: integridad de software, Microsoft Phoenix, proceso de compilación, protección de software, ofuscación de código

Abstract

Nowadays the commercial software is susceptible to the modification and observation of its machine code, by means of reverse engineering attacks. These attacks allow the software piracy, being billionaire the losses caused to the software industry due to this concept. This work focuses on the development of an extensible architecture, that it

allows inserting in automated way in the course of compilation, protective mechanisms in given software. Protection carries out of obvious way the developer itself and besides, allows designing and implementing of flexible and modular way the different techniques of protection identified. In order to validate the proposed architecture, the technique of obfuscation as a case study was implemented. This technique was tested on different algorithms and its show the obtained results.
Keywords: *software integrity, Microsoft Phoenix, compilation process, software protection, code obfuscation.*

Introducción

En la industria del software moderna, los proveedores y desarrolladores de software sufren grandes pérdidas debido a la distribución ilegal de software, una práctica conocida comúnmente como piratería de software. Parte del problema de la piratería de software se debe al hecho de que los programas son distribuidos como archivos electrónicos a través de Internet, siendo vulnerables a la modificación y observación por los usuarios. Por lo tanto, incluso los programas de software que hacen cumplir las inscripciones en línea antes de su uso legal, como medio de prevenir el uso no autorizado, pueden ser modificados de forma local por un usuario malicioso para evitar el proceso de registro en línea.

Estos peligros aumentan, debido a que el software puede ser duplicado y distribuido masivamente, en particular en los países donde los proveedores de software del programa tienen menos control sobre sus productos. Como resultado, los propietarios del software tienen pérdidas de ingresos significativos. La tasa de piratería aumentó a 51 billones de dólares en el 2009 (BSA, 2010).

En los últimos años ha crecido con mayor importancia el tema de la seguridad de las aplicaciones, específicamente cómo proteger las aplicaciones frente a la observación o modificación de su código. Son diversas las técnicas propuestas por la comunidad científica y la cantidad de variantes que se obtienen al unir varias de ellas se incrementan cada año. Debido a que la mayoría presenta un alto grado de complejidad, no es común que los programadores tengan conocimientos de estas y cuando lo tienen, aplicarlos de forma manual es inviable, ya que consumiría una gran cantidad de tiempo de desarrollo. No existe un mecanismo de protección infalible, por lo que estas técnicas tratan de dificultar el proceso de observación y modificación de los ejecutables.

Las aplicaciones pueden ser atacadas atentando contra su privacidad o contra su integridad. Atentar contra la privacidad es cuando se quiere observar sus datos y funcionamiento interno. Cuando se atenta contra la integridad de una aplicación se modifica el contenido de esta. Primero se debe conocer el funcionamiento interno de la aplicación

para conocer cómo debe ser modificada esta. Mediante estos ataques se puede invalidar el mecanismo de protección que tenga la aplicación y de esta forma ser usada ilegalmente. Por ejemplo, si se modifica el algoritmo de comprobación de la licencia, se puede lograr que no se detecte el uso ilegal de la aplicación.

A partir del conocimiento de cómo debe ser modificada la aplicación se pueden crear otras aplicaciones que realicen este proceso de forma automática. Estas aplicaciones, conocidas como "*cracks*" o "*keygen*", pueden ser distribuidas por Internet y permiten que usuarios inexpertos logren violar la protección de las aplicaciones. Para contrarrestar estos mecanismos existen variadas técnicas de protección que garantizan en cierta medida la protección de su privacidad e integridad. La privacidad es garantizada mediante técnicas como la ofuscación (Lin y Debray, 2003; Collberg y Thomborson, 2002) y el cifrado (Wang 2005; Chow et. al., 2003; Chow et. al., 2002), y la integridad se garantiza mediante la auto-verificación (Horne et. al., 2002; Chang y Atallah, 2003; Jakubowski et. al., 2007; Aucsmith, 1996) y las marcas de agua (Myles et. al., 2005; Nagra et. al., 2002). Para lograr un nivel de seguridad adecuado en la aplicación, es necesario garantizar tanto la privacidad como la integridad, por lo que es necesario complementar técnicas que garanticen ambos atributos. Implementar estas técnicas dentro de una aplicación tiene ciertos inconvenientes:

- Existen una gran variedad de técnicas, y estas pueden llegar a tener un alto grado de complejidad.
- No es común que los programadores tengan conocimientos de estas. En los proyectos de desarrollo se necesita el rol de analista de seguridad que es el encargado de esta tarea.
- Implementar la mayoría de las técnicas de forma manual puede consumir gran parte del desarrollo de la aplicación e introducir muchos errores de implementación. Además, la mayoría de estas técnicas deben ser implementadas de forma automatizada.

Con el objetivo de solucionar los inconvenientes anteriores, existe software para aplicar técnicas de protección de forma automatizada. Estas aplicaciones solucionan parcialmente los problemas expuestos anteriormente pero introducen nuevos problemas:

- Estas herramientas están diseñadas para aplicar un grupo de protecciones específicas.
- No son extensibles, es decir, no permiten personalizar las protecciones que insertan ni permiten insertarle mecanismos de protección realizados por terceros.
- Al ser propietarias no permiten ver su funcionamiento interno, para comprobar que la protección es aplicada correctamente.

Para dotar a una aplicación de estos mecanismos hay que insertárselos en la misma, y esto se puede realizar en tres momentos distintos: pre-compilación, compilación y post compilación.

A partir de un estudio realizado se reconoció que la mejor fase donde insertar las protecciones es en el proceso de compilación, debido a que la mayoría de las protecciones pueden ser insertadas completamente en esta fase. Las protecciones que no se pueden insertar completamente en esta fase, necesitan pequeñas modificaciones en post-compilación.

El presente trabajo aborda el desarrollo de una arquitectura de protección basada en el compilador Phoenix, que permita automatizar la aplicación de mecanismos de protección en el proceso de compilación. Adicionalmente se realizan pruebas sobre la arquitectura desarrollada mediante la implementación de una técnica de ofuscación de código.

Materiales y métodos

En la presente sección se brindan los detalles del desarrollo de la investigación. Inicialmente se brindan los fundamentos teóricos actualizados en los cuales se basó la investigación, tales como la seguridad y amenazas existentes sobre el software a comercializar, las técnicas de protección a emplear para evitar los ataques de ingeniería inversa, y las posibles vías de automatización para aplicar las distintas técnicas de protección. Por otra parte, se brinda una descripción de la arquitectura propuesta para la protección automatizada de software.

Seguridad y amenazas del software

Al escoger la manera en que se va a proteger una aplicación, se tiene que crear un modelo de amenaza. Esto no es más que un estudio del tipo de ataque que se le pueden realizar y los recursos que valen la pena proteger. Este modelo no es una ciencia exacta, se basa en suposiciones y su calidad se mide en cuan realmente refleja la realidad del ambiente de ejecución. A partir de este modelo es que se analiza, qué tipo de protección aplicarle al software en cuestión para evitar los ataques, protegiendo los recursos necesarios. Dentro del modelo de amenaza se analizan que zonas del software es necesario proteger pues solamente se necesita proteger zonas críticas de su código, como por ejemplo, dónde se encuentre el algoritmo encargado de la comprobación de la licencia (Oorschot y Main, 2003).

Los modelos de amenazas existentes son: red (*network*), local (*insider*) y el anfitrión no confiable (*untrusted-host*) (Wang 2005; Oorschot y Main, 2003). El modelo red es donde el atacante realiza el ataque a partir de tratar de

infiltrarse a través de la red o enviando mensajes por la red para provocar comportamientos no esperados de la aplicación. El modelo local es donde el atacante se encuentra en la red dentro de la misma compañía y tiene más privilegios que un atacante exterior. El modelo anfitrión no confiable es donde el atacante tiene control completo de la computadora y sus recursos, donde se encuentra la aplicación y puede analizarla y modificarla. Para el modelo red y local existen soluciones para disminuir o anular la efectividad de los ataques. El modelo anfitrión no confiable no cuenta actualmente con una solución que garantice la inefectividad de los ataques.

En el trabajo actual se considerará a las aplicaciones como confiables y el ambiente de ejecución no confiable. Esto corresponde con el modelo anfitrión no confiable, donde el objetivo del atacante es violentar la privacidad y la integridad de la aplicación, mediante el empleo de herramientas externas. El objetivo es proteger la integridad y privacidad de una aplicación confiable en un ambiente no confiable.

Las aplicaciones tienen dos atributos de seguridad: integridad y privacidad:

- Cuando se atenta contra su privacidad es cuando se intenta ver el contenido de la aplicación sin permiso del propietario. El objetivo de un ataque contra la privacidad de la aplicación es comprender el funcionamiento interno de esta.
- Atentar contra la integridad es cuando se intenta modificar el contenido de esta, igualmente sin permiso del propietario. El objetivo que persigue un ataque contra la integridad es variar el funcionamiento de las aplicaciones de forma que estas tengan un comportamiento distinto al original.

El proceso de ingeniería inversa es un ataque que atenta contra estas dos características de la aplicación. Este proceso consiste en la observación del funcionamiento de la aplicación y después la modificación, para poder usarla de forma ilegal. Estos ataques pueden ser realizados de forma estática o dinámica (Oorschot y Main, 2003):

- El ataque estático es cuando la aplicación no se encuentra ejecutándose, y se observa o modifica como si fuera un fichero binario, mediante herramientas llamadas desensambladores. Mediante el ataque estático, al no estar ejecutándose, no hay modificación de los registros y banderas del microprocesador.

El ataque dinámico se realiza con herramientas depuradoras (*debuggers*) y la aplicación se ejecuta mientras se puede observar el contenido de esta, detenerlo en algún punto específico o modificarlo si se desea. Como la aplicación se encuentra ejecutándose se puede observar como modifica los registros del microprocesador.

Técnicas de protección

Existen técnicas para disminuir la efectividad de los ataques aunque ninguna garantiza una seguridad absoluta. Para proteger la privacidad de la aplicación se logra con mecanismos que impiden al atacante observar el contenido de esta o dificultar su entendimiento. Cuando se intenta proteger la integridad de la aplicación, se le insertan mecanismos donde la aplicación puede verificarse a sí misma y comprobar que su contenido es correcto. Ninguno de estos mecanismos puede garantizar una protección absoluta.

Existen varias vías para proteger la privacidad e integridad de las aplicaciones. La privacidad puede ser protegida con técnicas como el cifrado (Wang 2005) y la ofuscación (Collberg et. al., 1997), y la integridad puede ser protegida con técnicas como la marca de agua en software (Nagra et. al., 2002) y la auto-verificación (Chang y Atallah , 2003; Aucsmith, 1996). Además, existe la técnica de diversidad de código (Wong y Stamp, 2006) que no ofrece resistencia contra estos ataques, pero impide que una vulnerabilidad en una aplicación se disperse por internet. Ninguna de estas técnicas garantiza que un software sea totalmente seguro. Todas tienen ventajas y desventajas y deben aplicarse según el tipo de ataque del que se quiera proteger.

Estas técnicas se usan en conjunto debido a que por sí solas es fácil encontrarles puntos débiles y explotarlos. Distintos autores han propuesto mezclas de estas técnicas para lograr un grado superior de protección: Aucsmith (1996) propone una arquitectura para la protección del software contra la modificación mezclando técnicas de ofuscación, auto-verificación de integridad y cifrado. Matias (2007) propone una protección basada en capas donde la ofuscación impediría entender el funcionamiento del software, la protección preventiva impediría que se pudiera inspeccionar el software y la técnica de auto-verificación detectaría modificaciones en la aplicación si ocurrieran.

En la Tabla 1 se expone un resumen sobre estas técnicas y el nivel de protección que ofrecen según el tipo de ataque. Se puede apreciar que a pesar de que el cifrado ofrece protección completa, si se logra obtener la llave de cifrado pierde toda la protección (Billet et. al., 2004; Goubin et. al., 2007; Wyseur et. al., 2007). La diversidad de código, parecería una mala técnica a emplear, por el hecho de que no ofrece ninguna protección contra análisis o modificación, pero sin embargo, es una técnica muy útil cuando se quiere evitar que un atacante pueda encontrar dentro del software la licencia o la llave, a partir de la comparación de versiones. Además, dentro de las vías para lograr diversidad de código se encuentra la ofuscación y el cifrado, por lo que si se utiliza la diversidad de código a partir de estas técnicas tendría sus ventajas.

Tabla 1. Resumen de técnicas de protección.

	Protección contra				
	Ingeniería Inversa		**Modificación**		
Técnica	**Estática**	**Dinámica**	**Estática**	**Dinámica**	**Garantizan**
Diversidad de código	Ninguna	Ninguna	Ninguna	Ninguna	Ninguna
Marcas de Agua	Ninguna	Ninguna	Parcial	Parcial	Integridad
Cifrado	Completa	Parcial	Completa	Parcial	Privacidad
Auto-verificación	Ninguna	Ninguna	Parcial	Parcial	Integridad
Ofuscación	Parcial	Parcial	Parcial	Parcial	Privacidad

A pesar de que esta tabla no refleja exactamente la importancia de cada técnica, da una buena idea de para qué tipo de ataque ofrecen mejor protección. Dentro de las técnicas que garantizan integridad la auto-verificación es la más fuerte, debido a que existe una diversidad de ataques satisfactorios a varios algoritmos de marcas de agua. Las marcas de agua son usadas más con propósitos jurídicos para probar autenticidad (Nagra et. al., 2002). Por otra parte de las técnicas que garantizan privacidad, la mejor es la ofuscación por el hecho de que en el cifrado es muy difícil ocultar la llave con la que se descifra en un modelo de amenaza de anfitrión no confiable.

En el trabajo actual se implementará una técnica de ofuscación. Para comprender mejor el funcionamiento de esta, a continuación se da una explicación más completa.

Ofuscación

La ofuscación (Lin y Debray, 2003; Collberg y Thomborson, 2002) es una técnica que propone modificar una aplicación estructuralmente, manteniendo su funcionalidad, con el objetivo de que sea más complejo analizarla. La ofuscación no esconde el contenido de la aplicación, sino que este es incomprensible para un humano o autómata y por lo tanto, es difícil determinar cómo violentar la protección. El atacante puede acceder fácilmente al código de la aplicación, pero necesita comprenderlo para que el ataque sea satisfactorio. Es en este punto donde la ofuscación interviene, modificando el código de forma que este sea incomprensible en un tiempo razonable. Se dice que "en un tiempo razonable" porque "con suficiente tiempo, esfuerzo y determinación, un programador competente siempre va a ser capaz de hacerle ingeniería inversa a una aplicación" (Collberg et. al., 1997).

Revista Cubana de Ciencias Informáticas
Vol. 9, No. Especial CCIA 17, 2015
RNPS: 2301 | ISSN: 2227-1899
http://rcci.uci.cu
Pág. 19-34

Es por la característica de la ofuscación de que no basa su fuerza en esconder el código, que se utiliza mucho con los lenguajes de Java y .NET por el hecho de que en estos lenguajes, el código no se compila en código máquina, sino que se lleva a una interpretación intermedia, y se compila solamente en el momento de ejecución, permitiéndoles un alto grado de portabilidad. Esta ventaja hacen que sean más vulnerables a la ingeniería inversa y por lo tanto, con descompiladores se puede obtener el código fuente casi igual a como lo escribió el programador.

Clasificaciones de ofuscación

Las ofuscaciones tienen varias clasificaciones según el objeto que ofuscan. Pueden ser clasificadas en control, datos o preventivas. Las ofuscaciones de control tienen como objetivo modificar el control de flujo de una aplicación. Este grupo de transformaciones son útiles en todos los lenguajes porque a partir del análisis del control de flujo de una aplicación, se obtiene información importante de cuál es el funcionamiento interno.

Las ofuscaciones de datos modifican la forma en que los datos son almacenados en la aplicación. Son útiles debido a que al oscurecer la forma en que los datos son almacenados es más difícil por un análisis de ingeniería inversa identificar qué tipo de dato está siendo procesado.

Las ofuscaciones preventivas no realizan ofuscación en el código. No están destinadas a proteger la aplicación contra el ataque de un humano, sino están destinadas a explotar debilidades de de-ofuscadores[1] automáticos y desensambladores con el objetivo de impedir que estos realicen su ataque (Collberg et. al., 1997).

Evaluación de la ofuscación

Las ofuscaciones tienen tres formas de ser evaluadas: potencia, flexibilidad y costo. La potencia es una medida del nivel de complejidad añadido al código. Para evaluarla se mide el nivel de complejidad en la aplicación, antes y después de ofuscar, a partir de una métrica de complejidad. Estas métricas miden características del código, como el nivel de anidamiento, cantidad de bloques básicos, etc. La potencia de una ofuscación depende del software donde se aplique pues cada software tiene características distintas. La flexibilidad de una ofuscación es la capacidad de soportar ataques automatizados por otro software, llamado de-ofuscador. Esta es una medida subjetiva pues no existe forma de medirla con valores exactos. El costo de una ofuscación es la carga agregada a la aplicación en tamaño y tiempo de ejecución. Esta medida se debe tratar que sea menor, pues sino la aplicación sería muy lenta

[1] Software creado con el objetivo de revertir las transformaciones de ofuscación.

En el trabajo actual se implementará una técnica de ofuscación del grupo de control. Esta técnica se llama "Aplanamiento del grafo de control de flujo" (Matias, 2007) y el objetivo que persigue es eliminar la información que se obtiene al realizarse ingeniería inversa a una aplicación y obtener el grafo de control de flujo. Con esta técnica todos los nodos del grafo de control de flujo tienen el mismo sucesor y predecesor, dificultando de esta forma el análisis de la continuidad del código.

Vías de automatización de la protección

Las aplicaciones desde que son creadas pasan por tres estados básicos: pre-compilación (Liem et. al., 2008; Wang et. al., 2000), compilación (Jakubowski et. al., 2009) y post-compilación (Ya-qi y Li, 2007). En cada uno de estos estados se le puede hacer modificaciones para insertarle la protección deseada, aunque no todas las protecciones pueden ser insertadas en todos los estados.

En el código fuente (pre-compilación)

En el estado de pre-compilación, la aplicación se encuentra en el código fuente y la protección a insertar se realiza mediante la modificación de su código fuente. Hay transformaciones aplicadas al código fuente que no afectan el funcionamiento del programa pues están orientadas a modificar los identificadores y a explotar características del pre-procesador. Este grupo de transformaciones se utiliza cuando es obligado o necesario distribuir el código fuente de la aplicación. Hay otras transformaciones aplicadas al código fuente que sobreviven al proceso de compilación, o sea, que modifican el funcionamiento de la aplicación. Este grupo de transformaciones modifican a la aplicación estructuralmente y por lo tanto son más difíciles de aplicar que las anteriores.

Aplicar protecciones de forma automática en el código fuente es complejo debido a que primero se necesita interpretarlo y crear estructuras que lo representen para después modificar esta estructura y al final volver a escribir esta estructura como código fuente. La protección que más se aplica al código fuente es la ofuscación, y específicamente las transformaciones de capa. En el código fuente se pueden aplicar otras transformaciones de forma manual, pero consume gran cantidad de tiempo.

En el proceso de compilación

La aplicación en el proceso de compilación, pasa por una fase donde se encuentra en una representación intermedia. Es en este estado donde se aplican las transformaciones a la aplicación para que sean realizadas durante el proceso de

Revista Cubana de Ciencias Informáticas
Vol. 9, No. Especial CCIA 17, 2015
RNPS: 2301 | ISSN: 2227-1899
http://rcci.uci.cu
Pág. 19-34

compilación. En este estado se encuentran gran cantidad de la información que se encuentra en el código fuente por lo que la mayoría de las transformaciones que se pueden realizar en el código fuente se pueden realizar en esta fase.

La protección de código auto-modificable y la auto-verificación no es posible aplicarla completamente, porque en este estado no tiene las direcciones de memoria de los saltos. Sin embargo, a pesar de que no se puede implementar completamente, se puede implementar la mayor parte de la lógica que necesitan estas transformaciones. La ventaja de este estado es que se abstrae del proceso de interpretar el código fuente, dejándole este proceso al compilador, y es independiente de la arquitectura, al dejar que el compilador genere el código máquina para la arquitectura específica.

En el software final (post-compilación)

El momento de post-compilación es cuando el compilador procesa esta representación intermedia y se obtiene el código compilado. Este código puede tener varias formas según como sea compilada la aplicación y bajo que plataforma. Cuando la aplicación compilada está escrita en código máquina, es muy difícil aplicarle transformaciones, por el hecho de que al insertar código se necesitan actualizar los valores de los saltos y estos no expresan una dirección absoluta, sino, son relativos a su posición dentro del código.

Realizar una transformación en este estado es igual a realizar un ataque estático mediante modificación, que tiene una gran complejidad, por el hecho de que se debe interpretar el código máquina y existen muchos tipos de arquitecturas de hardware con estructuras particulares para cada uno.

Por lo anterior expuesto se decidió insertar las protecciones en el proceso de compilación. Esto se debe a que durante este proceso se pueden insertar la mayor cantidad de protecciones. Hay protecciones que no se pueden insertar completamente durante la compilación, pero se puede insertar la mayor parte de su lógica durante este proceso, necesitando pequeños cambios después de ser compiladas, como es el caso de la auto-verificación.

Herramientas

Se identificaron una serie de herramientas que permiten aplicar los mecanismos de protección en los tres momentos identificados.

En el proceso pre-compilación pueden emplearse la herramienta NetBeans[2]. Por otra parte durante el proceso de compilación, pueden estarse empleando compiladores que permitan la extensión del proceso de compilación, entre los

[2] http://netbeans.org/

que se encuentran el GCC (GCC, 2010) y el Microsoft Phoenix (Microsoft, 2010). Por último, en post-compilación se puede emplear la herramienta Yoda‘s Protector (Yoda, 2010) y Themida (Oreans, 2010).

Se tuvieron en cuenta una serie de elementos que ayudaron a decidirse por utilizar una herramienta que permite insertar los mecanismos de protección en el proceso de compilación y no en las otras etapas:

- Resulta un trabajo difícil y complejo aplicar las técnicas de protección en la etapa de post-compilación debido a que las técnicas deben ser desarrolladas en código maquina dependiente de la arquitectura.
- Lo fácil que resulta eliminar el protector del software al realizarle un ataque por modificación.
- Netbeans construyen clases que permiten obtener información sobre el código fuente, pero no permiten su modificación.
- El GCC es una herramienta con muchas prestaciones pero tiene un alto nivel de complejidad para utilizarlo en la modificación de código. Se espera en trabajos futuros utilizarlo.

Por lo anterior expuesto, se puede ver que el Phoenix es una buena herramienta. Esto se demuestra con el hecho de que tiene mejor soporte que las demás herramientas y uno de los objetivos de su creación fue permitir la ofuscación de las aplicaciones creadas. Provee de una extensa librerías como soporte para la realización de *plugin* y herramientas. También se justifica la elección de esta herramienta con el hecho de que se han realizado estudios anteriores sobre aplicaciones, en análisis y modificación (Jakubowski et. al., 2009).

Descripción de la arquitectura propuesta

La solución propuesta tiene el objetivo de ser flexible, permitiéndoles a los usuarios extender sus funcionalidades. Mediante esta cualidad el sistema puede ser fácilmente ajustado a distintos ambientes y necesidades. Permite la abstracción del proceso de protección del proceso de desarrollo y además, permite que le sean integradas protecciones y funcionalidades desarrolladas por terceros, dando solución al problema inicial. Como compilador se escogió el Phoenix por las facilidades que tiene. Este compilador da la posibilidad de mediante *plugins* manipular el listado de fases que se ejecuta.

El sistema que se propone crear introduce una capa de abstracción que separa el mecanismo de protección de la forma de inserción. Para lograr un buen entendimiento de la arquitectura del sistema se muestra un diagrama, en la figura 1, utilizando el patrón de arquitectura conocido como capas (*layer*), organizado por responsabilidades. Mediante el

patrón de capas por responsabilidades se separa en capas las partes del sistema, cada una tiene ciertas responsabilidades y las superiores usan las facilidades que dan las inferiores.

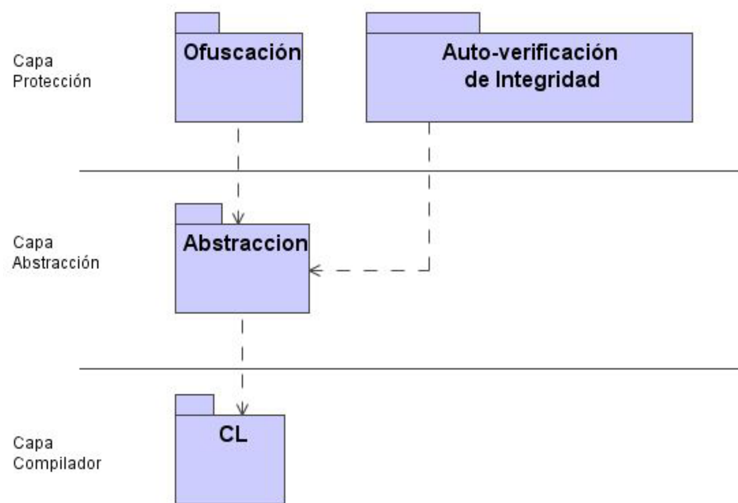

Figura 1. Diagrama de capas introducido por el sistema.

Capa Compilador: Esta capa es donde se encuentra el compilador del Phoenix (CL) y tiene la responsabilidad de compilar la aplicación deseada mediante la aplicación de un grupo de fases. Esta capa da la facilidad para integrar un *plugin* y manipular el listado de fases, a través de la eliminación o creación. Esta capa no es introducida por el sistema. En esta se encuentra el paquete “CL” que agrupa todas las funcionalidades del compilador.

Capa Abstracción: Esta capa es introducida por el sistema y es la encargada de mantener la lógica de negocio para la manipulación de las protecciones que se desean insertar y las funcionalidades que se desean utilizar. Ofrece la funcionalidad, a las capas superiores, de permitir la inserción de varios mecanismos de protección utilizando diversas funcionalidades que se pueden incluir si son deseadas.

Capa Protección: Esta capa es introducida por el sistema y es en esta donde se implementa la protección que se desea incluir en la aplicación que se va a compilar. Es también en esta donde se implementan las funcionalidades nuevas que se pudieran usar cuando sean deseadas. En esta capa se encuentran los paquetes “Ofuscación” y “Auto-verificación”. Ambos paquetes tienen incluido dentro técnicas de ofuscación y de verificación de integridad. En esta capa se pueden incluir más paquetes con técnicas nuevas desarrolladas por terceros.

Este sistema facilita el proceso de protección de aplicaciones. Mediante este, los desarrolladores de aplicaciones no necesitan preocuparse por las protecciones debido a que estas son insertadas automáticamente por el sistema.

Resultados y discusión

Para demostrar la validez del sistema y el correcto funcionamiento de las protecciones creadas, se realizó una prueba para insertarlas en tres aplicaciones distintas. Estas aplicaciones son parte de la suite MiBench (Guthaus et. al., 2001), utilizadas para probar el rendimiento en procesadores. La aplicaciones donde se van a probar las protecciones son: *Susan*, *Dijkstra* y *Sha*. *Susan* es un algoritmo para detectar bordes y esquinas en imágenes de resonancia magnética del cerebro. *Dijkstra* es una prueba donde se construye un grafo en una matriz y después calcula el camino más corto entre cada par de nodos usando repetidas corridas del algoritmo de *Dijkstra*. El algoritmo de Dijkstra es una solución al problema del camino más corto y se completa en tiempo polinomial. *Sha* es el algoritmo que produce un mensaje de 160 bits para una entrada dada.

La técnica de ofuscación, llamada "Aplanamiento del Grafo de Control de Flujo", se utilizó sobre estos 3 algoritmos y se obtuvieron las medidas de potencia y costo vistos en la Tabla 2Tabla. La potencia del algoritmo fue medida con una métrica que mide la complejidad del grafo de control de flujo.

Tabla 2. Resultados de las pruebas de ofuscación.

Algoritmo	**Potencia**	**Costo (espacio)**	**Costo (ejecución)**
Susan	2,8053	1,1311	2,8160
Dijkstra	3,0909	1,0158	1,6959
Sha	1,3333	1,0169	1,4008

Esta tabla muestra como la potencia de esta técnica es adecuada, pero aumenta el costo en ejecución significativamente. Esto se debe a que la ofuscación fue aplicada a todos los métodos de los algoritmos, por lo que es necesario antes de proteger una aplicación saber cuáles son los métodos donde la protección es necesaria. Los resultados obtenidos para el algoritmo "*Dijkstra*" son muy favorables debido a que aumenta significativamente la potencia manteniendo bajo los costos de ejecución y espacio.

Conclusiones

El presente trabajo tiene un alto valor práctico ya que por una parte permite a los desarrolladores de software aplicar de forma transparente y fácil los mecanismos de protección, incluso sin tener mucho conocimiento de ellos. Este proceso sería casi transparente para los desarrolladores pues no necesitan saber en detalle cómo se aplica la protección, solamente como afecta ésta a la aplicación.

Por otra parte a los que desarrollan técnicas de protección esta arquitectura le permite insertar sus módulos sin mayores contratiempos, además de reutilizar las bibliotecas de otras técnicas ya desarrolladas.

Referencias

D. AUCSMITH. Tamper Resistant Software: An Implementation. En Proceedings of the First International Workshop on Information Hiding. Lecture Notes in Computer Science. London, UK: Springer-Verlag, 1996, p. 317-333.

O. BILLET, H. GILBERT, AND C. ECH-CHATBI. Cryptanalysis of a white-box AES implementation. En Proceedings of the 11th Annual Workshop on Selected Areas in Cryptography. Lecture Notes in Computer Science. London, UK: Springer-Verlag, 2005, p. 227-240.

BUSINESS SOFTWARE ALLIANCE. Global software piracy study. [En línea] BSA, 2010. [Consultado el: 3 de junio de 2010]. Disponible en: [http://www.bsa.or].

H. CHANG AND M. J. ATALLAH. Protecting Software Code by Guards. En Sander, Tomas (editores). Security and Privacy in Digital Rights Management. Purdue University, USA: Springer Berlin / Heidelberg, 2002. Vol. 2320, p. 125-141.

S. CHOW, P. EISEN, H. JOHNSON, AND P. V. OORSCHOT. A white-box DES implementation for DRM applications. Digital Rights Management. Ottawa, Canada: Springer Berlin / Heidelberg, 2003. Vol. 2696, p. 1-15.

S. CHOW, P. A. EISEN, H. JOHNSON, AND P. C. V. OORSCHOT. White-Box Cryptography and an AES Implementation. En Revised Papers from the 9th Annual International Workshop on Selected Areas in Cryptography. SAC 2002 St. John's, Newfoundland, Canada: Springer-Verlag, 2003, p. 250-270.

C. S. COLLBERG, C. THOMBORSON, AND D. LOW. A Taxonomy of Obfuscating Transformations. [En línea] Department of Computer Science, The University of Auckland. Technical Report #148, 1997, [Consultado el 17 de mayo de 2010]. Disponible en https://researchspace.auckland.ac.nz/handle/2292/3491.

C. COLLBERG AND C. THOMBORSON. Watermarking, tamper-proofing, and obfuscation - tools for software protection. IEEE Transactions on Software Engineering, 2002, 28(3): p. 57-72.

PROYECTO GNU. GNU Compiler Collection. [En línea]. GCC, 2010. [Consultado el: 15 de mayo de 2010]. Disponible en http://gcc.gnu.org/.

L. GOUBIN, J. MASEREEL, AND M. QUISQUATER. Cryptanalysis of white box DES implementations. En Selected Areas in Cryptography. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer-Verlag, 2007, p. 278-295.

M. R. GUTHAUS, J. S. RINGENBERG, D. ERNST, T. M. AUSTIN, T. MUDGE, AND R. B. BROWN. Mibench: A free, commercially representative embedded benchmark. En Proceedings of the Workload Characterization. DC, USA: IEEE Computer Society, 2001, p. 3-14.

B. HORNE, L. R. MATHESON, C. SHEEHAN, AND R. E. TARJAN. Dynamic Self-Checking Techniques for Improved Tamper Resistance. En Sander, Tomas (editores). Security and Privacy in Digital Rights Management. Santa Clara CA, USA: Springer Berlin / Heidelberg, 2002. Vol. 2320, p. 77-124.

M. JAKUBOWSKI, P. NALDURG, V. PATANKAR, AND R. VENKATESAN. Software Integrity Checking Expressions (ICEs) for Robust Tamper Detection. En IH'07: Proceedings of the 9th international conference on Information hiding. Lecture Notes in Computer Science. Saint Malo, France: Springer-Verlag, 2007, p. 96-111.

M. H. JAKUBOWSKI, N. SAW, AND R. VENKATESAN. Iterated Transformations and Quantitative Metrics for Software Protection. En International Conference on Security and Cryptography (SECRYPT 2009): Milan, Italy, 2009, p. 359-368.

C. LIEM, Y. X. GU, AND H. JOHNSON. A compiler-based infrastructure for software-protection. En Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security. ACM SIGPLAN workshop on Programming languages and analysis for security. Tucson, AZ, USA: ACM, 2008, p. 33-44.

C. LINN AND S. DEBRAY. Obfuscation of executable code to improve resistance to static disassembly. En Proceedings of the 10th ACM conference on Computer and communications security. ACM conference on Computer and communications security. Washington D.C., USA: ACM, 2003, p. 290-299.

M. MATIAS. Application Security through Program Obfuscation. Phd thesis. Electronic and information system department. University of Gent, Gante, 2007.

MICROSOFT CONNECT. Microsoft connect. [En línea]. Microsoft Phoenix, 2010. [Consultado el: 27 de junio de 2010]. Disponible en http://connect.microsoft.com/phoenix.

G. MYLES, C. COLLBERG, Z. HEIDEPRIEM, AND A. NAVABI. The evaluation of two software watermarking algorithms: Research Articles. Software Practice & Experience, 2005, 35(2): p. 128-137.

J. NAGRA, C. THOMBORSON, AND C. COLLBERG. A functional taxonomy for software watermarking. Australian Computer Science Communications, 2002, vol. 24(3): p. 177-186.

P. C. OORSCHOT AND A. MAIN. Software protection and application security: Understanding the battleground. En International Course on State of the Art and Evolution of Computer Security and Industrial Cryptography. Heverlee, Belgium: Springer LNCS, 2003, (to appear).

OREANS TECHNOLOGIES. Oreans Technologies [En línea]. Themida, 2010. [Consultado el 17 de mayo de 2010]. Disponible en http://www.oreans.com/es/themida.php.

P. WANG. Tamper Resistance for Software Protection. Master Thesis in Science. Daejeon Information and Communications University, Korea, 2005.

C. WANG, J. HILL, J. KNIGHT, AND J. DAVIDSON, Software Tamper Resistance: Obstructing Static Analysis of Programs. [En línea]. University of Virginia, Charlottesville, VA, USA. [Consultado el 19 de mayo de 2010]. Disponible en http://www.cs.virginia.edu/~jck/publications/acmacac.2000.pdf

W. WONG AND M. STAMP. Hunting for metamorphic engines. Journal in Computer Virology, 2006, 2(6): p. 211-229.

B. WYSEUR, W. MICHIELS, P. GORISSEN, AND B. PRENEEL. Cryptanalysis of white-box DES implementations with arbitrary external encodings. En: Adams, Carlisle and Miri, Ali and Wiener, Michael (editores). Selected Areas in Cryptography. Berlin: Lecture Notes in Computer Science, 2007, p. 264-277.

S. YA-QI AND L. LI. Experiment with control code obfuscation. Communication and Computer, 2007, 4(3): pp.37-45.

YODA'S PROTECTOR. Yoda's Protector. [En línea] Yoda's Protector, 2010. [Consultado el: 15 de mayo de 2010]. Disponible en http://yodap.sourceforge.net/