Análise de Malware .NET para Identificação de Comportamentos Similares

Cicero Silva Luiz Junior[1], **Paulo Lício de Geus**[1], **André Grégio**[2]

[1] Instituto de Computação (IC)
Universidade Estadual de Campinas (Unicamp)
Campinas – SP – Brasil

[2]Departamento de Informática (DInf)
Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brasil

Abstract. *Currently, the .NET Framework is one of the most popular platforms for software development. It was introduced around the year 2000, with the idea behind it that it should be safe and increase developers' productivity. However, it also attracted some malware developers, since the .NET framework could not only be used to improve their productivity but also to write multi-platform malware. More importantly, .NET is adopted widely across the industry, from large scale servers to embedded devices and micro-controllers. At the same time, however, the .NET framework enforces a strong richness in its metadata, which helps to statically analyze a compiled binary in a way that makes it possible to infer most of its behaviors.*

Resumo. *Hoje em dia, o .NET é um dos ambientes mais populares para desenvolvimento de Software. Ele foi introduzido por volta dos anos 2000, com o intuito de ser uma plataforma segura e que aumentasse a produtividade dos desenvolvedores. Contudo, esse aumento de produtividade também chamou a atenção de desenvolvedores de malware, que viram nele a possibilidade de se escrever programas maliciosos de forma rápida e que pudessem alcançar uma vasta gama de dispositivos, já que o .NET é executado em ambientes desde servidores de larga escala até microcontroladores ARM ou MIPS. Contudo, a estrutura do framework do .NET é muito rica em meta-informação, o que facilita a análise de binários para inferir seus possíveis comportamentos.*

1. Introdução

Estima-se atualmente que o sistema operacional Windows (família NT) é responsável pelo funcionamento de mais de 90% de todos os computadores pessoais existentes [for Internet Technologies 2016] . Nos anos 2000, com o intuito de melhorar a segurança, performance e confiabilidade do ecossistema MS-Windows, houve um esforço da parte da Microsoft em construir um ambiente de execução gerenciado e seguro. Esse ambiente deveria se tornar a principal plataforma de desenvolvimento de software para este ecossistema e é conhecido como .NET Framework, cuja primeira versão foi disponibilizada em 2002 e, desde então, sua importância dentro do ambiente MS-Windows só aumentou.

Embora o .NET tenha sido construído com o intuito de facilitar a construção de aplicativos legítimos, a sua capacidade de interagir de forma muito integrada com o sistema operacional também o tornou atraente para a construção de programas maliciosos

(*malware*). Além disso, o fato de ele estar disponível em praticamente qualquer instalação MS-Windows, desde as Desktop até as versões integradas e IoT o torna um bom meio de difusão de *malware*, pois um código escrito para uma plataforma pode ser executado em qualquer outra que também implemente o .NET [LLC 2015] .

Sabe-se, contudo, que programas .NET são compilados para uma linguagem intermediária (*CIL - Common Intermediate Language*) e que o *framework* exige que uma série de regras rígidas sejam cumpridas para que os binários executem. Dessa forma, é possível explorar um binário potencialmente malicioso compilado em *CIL* e obter estaticamente uma representação de alto-nível do seu código fonte. Tal representação pode ser usada para comparação entre diferentes programas de forma a se buscar similaridades, isto é, funções reutilizadas ou pequenas mudanças que indiquem que um *malware* é uma variante de outro, pertencendo assim à mesma família.

A contribuição deste artigo é introduzir um analisador estático automatizado de *malware* com enfoque em amostras compiladas para CIL, utilizar o algoritmo MOSS para detectar a similaridade entre trechos de um dado programa e comparar esses trechos a uma base de dados gerada pelos autores, a qual possui trechos de códigos maliciosos que representam funções que implementam comportamentos suspeitos conhecidos. Desta forma é possível detectar, em um compilado CIL arbitrário, se ele possui trechos com comportamento potencialmente malicioso. O texto está dividido da seguinte maneira: introduz-se na Seção 2 detalhes sobre o .NET Framework, técnicas de análise de *malware* e o algoritmo MOSS; na Seção 3, discute-se o sistema proposto (arquitetura e implementação); na Seção 4 mostram-se testes realizados e resultados obtidos e, por fim, na Seção 5 apresenta-se a conclusão do artigo.

2. Aspectos Técnicos e Trabalhos Relacionados

Nesta seção são apresentados conceitos básicos de como funciona o .NET Framework, alguns detalhes importantes para a execução da proposta deste trabalho, tais como a descrição resumida do que é uma análise estática, uma breve introdução ao algoritmo MOSS, bem como alguns trabalhos relacionados.

2.1. .NET Framework

O Microsoft .NET é um *framework* arquitetado e desenvolvido pela Microsoft[Microsoft 2016]. Seus principais componentes são a *FCL (Framework Class Library)* e o *CLR (Common Language Runtime)*. De maneira sucinta, pode-se dizer que a *FCL* é um conjunto de bibliotecas com funções prontas para uso por parte dos programas enquanto o *CLR* é uma combinação de compilador e sistema de suporte à execução[Wikipedia 2016b].

CLR - Common Language Runtime. O *CLR* é o alicerce de todo o ambiente .NET. Trata-se de um agente que gerencia código em tempo de execução e provê serviços essenciais, tais como gerenciamento de memória e *threads*. Adicionalmente, o *CLR* também é responsável por garantir a segurança de tipos de dados e várias outras tarefas ligadas à salvaguarda e robustez dos programas gerenciados por ele[Microsoft 2010]. Especialmente com foco em robustez, por exemplo, o *CLR* possui um módulo dedicado chamado *CTS (Common Type System)*, cujo objetivo é obrigar a todo e qualquer código compatível com o *CLR* ser auto-descritivo e se basearem em um conjunto pré-definido de tipos de dados

básicos. Devido a isso, códigos gerenciados podem consumir outros códigos gerenciados de forma completamente confiável, independente da linguagem de programação original em que eles foram escritos[Microsoft 2010, Microsoft 2015a, Microsoft 2015b]. Outras informações também contidas nessa "auto-descrição" incluem detalhes de todas as versões de bibliotecas que são usadas pelo programa em questão.

CIL / MSIL (Common Intermediate Language / Microsoft Intermediate Language). *MSIL* (ou *CIL*) é a linguagem intermediária produzida por qualquer compilador cujo alvo seja produzir programas que devem executar no *CLR*. Trata-se de uma linguagem orientada a objetos e que se assemelha com uma linguagem *bytecode*[Wikipedia 2016a].

2.2. Técnicas de Análise de Malware

Há vários métodos para se tentar determinar se a ação de um componente de *software* é benigna ou maligna. Os principais métodos, entretanto, podem ser divididos em dois grandes grupos:

- **Análise Estática:** a ideia é estudar o comportamento de um programa sem de fato precisar executá-lo. Para tanto, normalmente se utiliza ferramentas tais como descompiladores, analisadores de códigos-fonte, pesquisadores de *strings* e afins[Mehta 2015]. As principais vantagens desse método de análise são que ele é bastante rápido e pode possivelmente levar a descobertas sobre ramos de execução que nem sempre são exercitados em um ataque real. Por outro lado, sabe-se que na prática é impossível prever todos os caminhos de execução de um programa apenas com base na análise estática clássica, exceto para exemplares suficientemente pequenos e simples[Moser et al. 2007].
- **Análise Dinâmica:** trata de monitorar um certo componente de *software* enquanto ele está sendo executado. Para que a análise seja efetiva, o *software* (ou o ambiente de análise) precisa ser "instrumentalizado", ou seja, ligeiramente modificado para permitir ao analista obter informações bastante detalhadas sobre o programa analisado. Alguns dados que costumam ser de interesse são as saídas produzidas pelo programa, eventuais modificações que ele cause tanto no sistema de arquivos como no repositório central de configurações, tentativas de comunicação com servidores externos de forma periódica, alterações que ele provoca em outros programas executando ao mesmo tempo que ele, parâmetros enviados para *syscalls*, etc[Dan Farmer 2005, EGELE et al. 2012].

Análises feitas sobre diversos tipos de programas maliciosos mostraram que vários deles, na verdade, compartilham código. Com o intuito de facilitar o trabalho dentro das comunidades de *hackers*, alguns destes construíram "ferramentas de administração", que nada mais são do que programas maiores que contêm uma grande quantidade de ataques disponíveis e uma interface de uso intuitivo, permitindo a criminosos não tão experientes criarem seu próprio *malware* personalizado[Pontiroli 2014, McDaniel and Schultz 2014].

2.3. MOSS - *Measure of Software Similarity*

O *MOSS* é um sistema cujo principal objetivo é detectar plágio de software [Aiken 1994]. Dados dois ou mais programas como entrada ele é capaz de retornar a porcentagem que esses dois ou mais programas compartilham de códigos entre eles. Adicionalmente, o algoritmo por trás da implementação do *MOSS* não pode ser trivialmente enganado, de

tal forma que renomear (ou até mesmo suprimir) nomes de variáveis, alterar a ordem de um código (mantendo sua semântica) ou outras transformações de natureza semelhante não produzem alterações significativas da taxa de detecção [Saul Schleimer 2003] . Desta forma, códigos que estiverem ofuscados, mas que conservem as suas principais estruturas lógicas continuam sendo detectáveis como semelhantes pelo algoritmo.

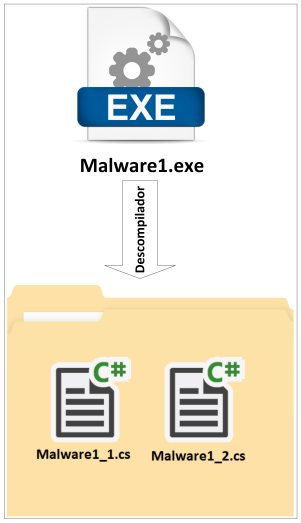
2.4. Trabalhos Relacionados

[Filho et al. 2010] propõem um sistema de análise dinâmica de *malware* baseado em *SSDT Hooking*. Dentro do contexto de sistemas operacionais Windows, *SSDT* significa *System Service Dispatch Table*, ou seja, trata-se da tabela de *Syscalls* do sistema operacional. Através da técnica de *Hooking* de *Syscalls*, o sistema BehEMOT permite obter, em tempo de execução, várias informações sobre um certo processo ou conjunto de processos em execução em uma máquina instrumentalizada. Dessa forma é possível gerar um conjunto detalhado de *logs* que permite a um analista determinar se um dado programa é maligno ou benigno. Contudo, a partir da versão 6.0 do Windows NT em 64 bits, não é mais possível fazer alterações em estruturas críticas do *Kernel*. Portanto, a abordagem tomada pelo BehEMOT só pode ser utilizada no NT 5.2 ou inferior ou, para versões acima da 6.0, apenas para as versões de 32 bits.

[Botacin et al. 2014] propõem uma alternativa que elimina a limitação do artigo anterior, permitindo a avaliação de forma dinâmica de um exemplar de *malware* de 64 bits executando no Windows 8 (NT 6.2). Para que esse objetivo fosse alcançado, construiu-se um *filesystem filter* e um monitor de *callback*, ambos em modo *kernel*, que permitem a uma ferramenta externa capturar atividades executadas no sistema em questão, como por exemplo escritas ou leituras de chaves de registro, alterações no sistema de arquivos, tráfego de rede etc. Notar que a abordagem do trabalho é ligeiramente diferente das abordagens tradicionais que eram aplicáveis a versões anteriores ao NT6, já que o NT6 de 64 bits introduziu novas funcionalidades que impedem o funcionamento de tais abordagens.

Por outro lado, [Christodorescu and Jha 2003] focam na construção de um sistema genérico capaz de detectar padrões de ações maliciosas em arquivos executáveis. De forma resumida, para que isso seja alcançável, primeiro constrói-se uma representação do que define um "comportamento malicioso". Em seguida, constrói-se uma representação do executável que será analisado. De posse de ambas representações, o sistema gera um autômato que contém uma representação genérica das dependências entre as variáveis do programa analisado. A partir disso, para cada procedimento do executável que será analisado é gerado um *CFG - Control-Flow-Graph*. Dando prosseguimento, o sistema de detecção recebe como entrada a representação do comportamento malicioso e um *CFG* e responde se o comportamento malicioso pôde ser detectado no *CFG*.

Por fim, [Grégio et al. 2012] apresentam uma abordagem baseada na detecção de instruções de escrita em memória para identificar possíveis trechos de reuso de código. Para tanto, instrumentaliza-se um sistema com um *debugger* que o monitora por instruções baixo-nível específicas que alteram os valores de registradores ou de posições de memória. Em seguida, agrupa-se cada uma dessas instruções duas-a-duas em uma janela deslizante produzindo uma sequência de bigramas. Por fim, utiliza-se um algoritmo de clusterização sobre os bigramas produzidos, com o intuito de agrupar exemplares de *malware* que tenham gerado sequências parecidas de operações de escrita. Com base



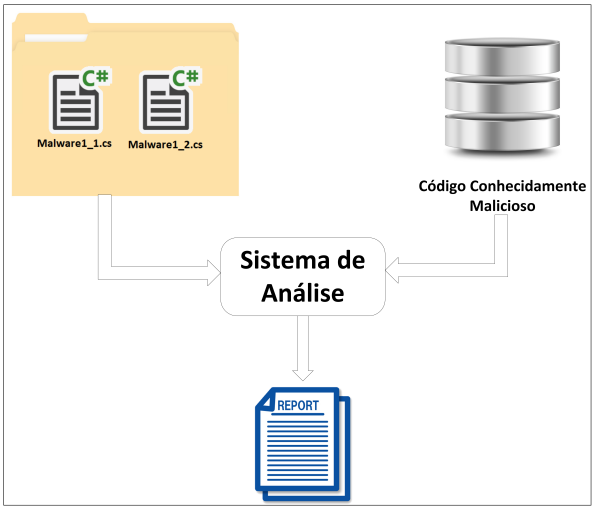


Figura 1. (à esq.) Visão geral da etapa de descompilação automática de uma amostra de *malware*

Figura 2. (à dir.) Esquema simplificado do sistema de análise heurística

nisso, o trabalho é capaz de afirmar quão similar um exemplar de *malware* é quando comparado a outro.

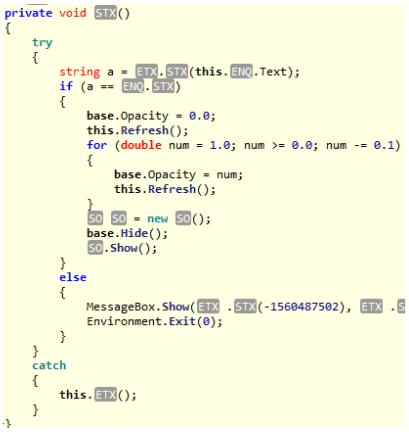
3. Arquitetura e Implementação

O sistema proposto é composto de duas partes: a primeira é responsável pela engenharia reversa, na qual códigos de alto nível são obtidos a partir dos binários .NET. A segunda faz uma análise sobre os códigos-fonte obtidos e tenta localizar trechos que contêm ações potencialmente maliciosas. As Figuras 1 e 2 ilustram, respectivamente, as entradas e as saídas esperadas para cada um dos módulos do sistema.

Módulo 1: Módulo de Engenharia Reversa. O Módulo de Engenharia Reversa é implementado por um programa autônomo, com o auxílio de bibliotecas capazes de analisar códigos *CIL* e transformá-los em linguagem de alto-nível. Neste caso, apenas por familiaridade, escolheu-se transformar código CIL em linguagem CSharp. [icsharpcode 2016, Jetbrains 2016, Gate 2016]. Exemplos dos tipos de código de alto-nível que podem ser extraídos utilizando-se bibliotecas como as mencionadas anteriormente podem ser observados nas Figuras 3 e 4. Na primeira delas trata-se de um código fracamente ofuscado, enquanto na outra trata-se de um código fortemente ofuscado.

Módulo 2: Módulo de Análise Heurística baseada no MOSS. O Módulo de Análise é composto por uma base de dados com trechos de código conhecidamente maliciosos e um sistema capaz de se utilizar destas informações para gerar comparações com os códigos descompilados obtidos na fase 1. Todos os códigos submetidos ao sistema são considerados, procurando-se por algum conjunto que se assemelhe ao que está armazenado no banco de dados. No final é gerado um relatório que indica, nos códigos descompilados, trechos de programa que têm alta probabilidade de serem maliciosos. No caso concreto é feita uma varredura por toda a base de dados e, utilizando o algoritmo MOSS, descobre-se quais trechos de código conhecidamente maliciosos aparecem, total ou parcialmente, dentro de cada um dos exemplares em análise.

```
private void InitializeComponent()
{
    this.components = new Container();
    this.axLUhrLLnI = new mWYMsmhuVo();
    this.bindingSource1 = new BindingSource(this.components);
    ((ISupportInitialize)this.bindingSource1).BeginInit();
    base.SuspendLayout();
    base.Controls.Add(this.axLUhrLLnI);
    base.AutoScaleDimensions = new SizeF(6f, 13f);
    base.AutoScaleMode = AutoScaleMode.Font;
    base.ClientSize = new Size(363, 248);
    base.Name = "Form1";
    this.Text = "Form1";
    base.Load += new EventHandler(this.Form1_Load);
    ((ISupportInitialize)this.bindingSource1).EndInit();
    base.ResumeLayout(false);
}
```



```
private void STX()
{
    try
    {
        string a = ETX.STX(this.ENQ.Text);
        if (a == ENQ.STX)
        {
            base.Opacity = 0.0;
            this.Refresh();
            for (double num = 1.0; num >= 0.0; num -= 0.1)
            {
                base.Opacity = num;
                this.Refresh();
            }
            SO SO = new SO();
            base.Hide();
            SO.Show();
        }
        else
        {
            MessageBox.Show(ETX .STX(-1560487502), ETX .S
            Environment.Exit(0);
        }
    }
    catch
    {
        this.ETX();
    }
}
```

Figura 3. (à esq.) Código-fonte levemente ofuscado extraído a partir de um compilado em *MSIL/CIL*: a maior parte das variáveis ainda possui seu nome original no código *CIL* e pode, portanto, ser reconvertida em código-fonte[MalwareTips 2014].

Figura 4. (à dir.) Código-fonte fortemente ofuscado extraído a partir de um compilado em *MSIL/CIL*: embora os nomes das variáveis e classes não estejam disponíveis, mesmo assim é possível obter código suficientemente de alto-nível de tal forma que seja possível inferir sua funcionalidade[Altman 2012].

3.1. Implementação

Todo o software foi implementado em linguagem C# no Microsoft Visual Studio. Por conveniência, decidiu-se utilizar o Microsoft SQL Azure como banco de dados para o armazenamento das informações, uma vez que isso contribui para a velocidade da análise. Adicionalmente, todos os processos de análise e teste são executados a partir de uma máquina virtual dentro do Microsoft Azure. Para a estrutura do sistema decidiu-se pela utilização do paradigma DDD (Desenvolvimento Dirigido a Domínio). Inicialmente optou-se por construir a Interface de Usuário utilizando-se o WPF (Windows Presentation Foundation), com o intuito de se economizar tempo de implementação.

3.1.1. Interface de Usuário

Tela Inicial. A tela inicial permite ao usuário escolher entre as principais funções do Programa. Em *Manage Code Snippets* ele pode gerenciar os comportamentos conhecidos pelo programa; em *Analyze New Malware* é possível analisar um único Malware e obter um relatório de comparação com a base de conhecimento; A função *Browse Analyzed* permite visitar análises antigas que foram guardadas na base de conhecimento e, por fim, a opção *Directory / Batch Analysis* permite selecionar um diretório inteiro contendo Malware e processar a análise de todos eles, gerando um relatório individual automatizado para cada um.

Tela de Gerenciamento de *Snippets*. Nesta tela é possível gerenciar os comportamentos maliciosos conhecidos pelo programa.

Tela de Análise de novo *Malware*. Nesta tela é possível submeter um único exemplar de Malware para Análise. Assim que a análise é concluída é exibido um relatório, conforme

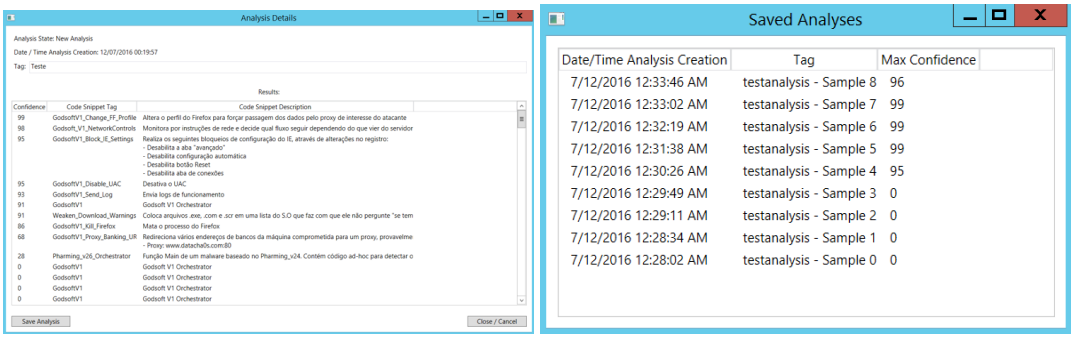


Figura 5. (à esq.) Exemplo de tela de relatório obtida após uma análise. *Confidence* representa a probabilidade de um certo comportamento estar presente no exemplar fornecido como entrada para o sistema.

Figura 6. (à dir.) Relatório gerado a partir de uma análise em batch, demonstrando os *hits* que aconteceram para cada um dos exemplares. *Max Confidence* representa o maior valor de uma detecção dentro do exemplar sendo analisado. Caso a maior detecção seja baixa, quer dizer que o exemplar em análise não é semelhante a nada presente na base de dados. Caso seja alta, algum dos *snippets* da base de conhecimento se assemelha a alguma porção de código do programa em análise.

apresentando na Figura 5.

Análise de Malware em *Batch*. Neste modo é possível fornecer um diretório com vários executáveis. O programa faz a análise e guarda internamente o relatório para cada um dos exemplares. Para facilitar a busca, todos os exemplares analisados a partir de uma requisição via *batch* compartilham a mesma *tag*, e a isso é adicionado um contador, para indicar qual dos arquivos é de que se trata.

3.1.2. Engine

Para implementar a abordagem DDD, o *Engine* do sistema é dividido em quatro grandes módulos: Domínio (responsável por definir a lógica principal e os contratos que a implementação deve seguir), Repositório (responsável por interagir com o armazenamento de dados), Serviço (responsável por coordenar a interação entre o Repositório, os Modelos do Domínio e outros serviços) e o Util, que possui adaptadores que permitem ao programa acessar ferramentas externas (tais como o descompilador ILSpy [icsharpcode 2016] , que foi o utilizado para transformar *CIL* em C# para posterior análise e uma interface *REST* que comunica com o serviço online do MOSS disponibilizado pela Universidade de *Stanford*.)

3.1.3. Modelos do Domínio

Os Modelos do Domínio são responsáveis por representar os dados que são armazenados no sistema, além de conter métodos que operam sobre esses dados. Os modelos implementados para o programa em questão são:

BaseModel. Todos os modelos herdam deste modelo base. Seu único campo é um Id, composto de 128 bits que tem como objetivo identificar de forma única um objeto no sistema

Analysis. Representa uma análise completa. Suas informações são um *TimeStamp*, uma estrutura *FileDisassebly* (que representa a descompilação de um executável em *CIL* para C#), uma lista de estruturas do tipo *CodeCompareResult* (em que cada estrutura representa a comparação entre um arquivo de código fonte do Malware sendo analisado e um dos *snippets* de conhecimento do sistema), uma *Tag* (que é um campo texto, apenas para facilitar a busca dos dados na base) e um *MaxConfidence*, que não é armazenado na base, mas sim recalculado sempre se obtendo o maior valor de *Confidence* dentro da lista de *CodeCompareResult*.

CodeSnippet. Representa o conhecimento de um pedaço de código malicioso. Seus campos são um *CodeSnippetSource* (texto), que possui o código em C# que implementa o comportamento em questão, o campo *CodeSnippetDescription*, que é uma descrição em linguagem natural do que faz o comportamento em questão e uma *Tag*, utilizada para facilitar buscas e agrupamentos na base de dados.

DisassemblyResult. Representa a descompilação de um *malware* por um descompilador. Seus campos são um campo dizendo qual o descompilador que foi utilizado, uma lista de estruturas do tipo *SourceFile* (que representa um arquivo de código fonte) e um apontador para uma estrutura do tipo *FileDisassembly*, que é a entidade pai que guarda objetos do tipo *DisassemblyResult*

FileDisassembly. Representa a descompilação de um executável de *CIL* para C#. Como o programa permite que se pluguem vários descompiladores a ele, um *FileDisassembly* na verdade possui uma lista de *DisassemblyResult*, onde cada um deles está associado a um descompilador. Isso foi introduzido com o intuito de se comparar os resultados obtidos, caso mais de um descompilador seja utilizado. Além do campo mencionado também são guardados um *TimeStamp* da criação da descompilação, extensão do arquivo, comprimento do arquivo original em *bytes*, nome do arquivo original e um *snapshot* completo do arquivo, para futura reanálise.

SourceFile. Representa um arquivo de código fonte que foi descompilado a partir do binário em *CIL* fornecido. Seus campos são o *FileContent* (código fonte em C# propriamente dito), *FileName* (que na maioria das vezes também é recuperado) e o *FileLengthInBytes*, que representa o comprimento do arquivo extraído.

4. Testes e Resultados

Para validar o funcionamento do sistema, separou-se um conjunto de exemplares de *Malware* e seus códigos-fonte foram analisados utilizando a ferramenta ILSpy [icsharpcode 2016]. A partir destas análises foi possível determinar que vários destes programas maliciosos eram construídos a partir de *toolkits*, e isso ficava bastante evidente nos fontes descompilados. A partir disso, algumas das funcionalidades encontradas foram documentadas e introduzidas no programa como porções de código da base de conhecimento. Em seguida fez-se uma análise em *batch* sobre um segundo conjunto composto por 9 exemplares, dos quais 5 eram sabidamente construídos com a ferramenta GodSoft. Como o esperado, o sistema foi capaz de identificar os 5 exemplares que continham algum tipo de código parecido com o que havia sido introduzido em sua base de conhecimento e, ao mesmo tempo, não gerou falsos-positivos para exemplares que não devem ser detectados. Na figura 6 é possível verificar o relatório gerado para esta análise.

Adicionalmente também foi feita uma análise individual sobre um décimo exem-

plar que também havia sido construído com a ferramenta GodSoft. O relatório detalhado para esta detecção pode ser observado na Figura 5 e, como o esperado, todos os componentes do GodSoft foram identificados, ao mesmo tempo em que componentes alheios tiveram uma baixa taxa de detecção.

5. Conclusão

Como se pode observar, analisar exemplares de *malware* utilizando-se um descompilador e um analisador baseado no algoritmo do *MOSS* é viável. Adicionalmente, devido às propriedades deste algoritmo em questão, ofuscações mais simples (baseadas na alteração ou supressão do nome de variáveis) não alteram a detecção, já que o próprio MOSS é, por construção, imune a estes tipos de alterações.

Referências

Aiken, A. (1994). *A System for Detecting Software Plagiarism*. `http://theory.stanford.edu/~aiken/moss/`. Acessado em Setembro/2016.

Altman, T. (2012). Reverse-engineer an obfuscated .net application. `http://travisaltman.com/reverse-engineer-an-obfuscated-net-application/`. Acessado em Setembro/2016.

Botacin, M., Afonso, V., de Geus, P. L., and Grégio, A. (2014). Monitoração de comportamento de malware em sistemas operacionais windows nt 6.x de 64 bits. *Anais do SBSEG 2014*.

Christodorescu, M. and Jha, S. (2003). Static analysis of executables to detect malicious patterns. *In Proceedings of the 12th USENIX Security Symposium*.

Dan Farmer, W. V. (2005). Malware analysis basics. `http://www.porcupine.org/forensics/forensic-discovery/chapter6.html`. Acessado em Setembro/2016.

EGELE, M., SCHOLTE, T., KIRDA, E., and KRUEGEL, C. (2012). A survey on automated dynamic malware analysis techniques and tools. *ACM Computing Surveys*.

Filho, D. S. F., Grégio, A. R. A., Afonso, V. M., Santos, R. D. C., Jino, M., and de Geus, P. L. (2010). Análise comportamental de código malicioso através da monitoração de chamadas de sistema e tráfego de rede. *Anais do SBSEG 2010*.

for Internet Technologies, M. S. S. (2016). *Operating System Desktop Market Share*. `http://marketshare.hitslink.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0`. Acessado em Setembro/2016.

Gate, R. (2016). .net reflector. `http://www.red-gate.com/products/dotnet-development/reflector/`. Acessado em Setembro/2016.

Grégio, A. R. A., de Geus, P. L., Kruegel, C., and Vigna, G. (2012). Tracking memory writes for malware classification and code reuse identification. *DIMVA'12 Proceedings of the 9th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*.

icsharpcode (2016). Ilspy - .net decompiler. http://ilspy.net/. Acessado em Setembro/2016.

Jetbrains (2016). Dotpeek. https://www.jetbrains.com/decompiler/. Acessado em Setembro/2016.

LLC, S. L. (2015). *Netduino*. http://www.netduino.com/. Acessado em Setembro/2016.

MalwareTips (2014). Malware analysis 3 - reverse engineering .net malware. http://malwaretips.com/threads/malware-analysis-3-reverse-engineering-net-malware.42338/. Acessado em Agosto/2015.

McDaniel, D. and Schultz, J. (2014). Reversing multilayer .net malware. http://blogs.cisco.com/security/talos/reversing-multilayer-net-malware. Acessado em Setembro/2016.

Mehta, L. (2015). Malware static analysis. http://resources.infosecinstitute.com/malware-analysis-basics-static-analysis/. Acessado em Setembro/2016.

Microsoft (2010). .net framework conceptual overview. https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.100).aspx. Acessado em Setembro/2016.

Microsoft (2015a). Common language runtime (clr). https://msdn.microsoft.com/en-us/library/8bs2ecf4(v=vs.110).aspx. Acessado em Setembro/2016.

Microsoft (2015b). Common type system. https://msdn.microsoft.com/en-us/library/zcx1eb1e(v=vs.110).aspx. Acessado em Setembro/2016.

Microsoft (2016). .net framework. https://www.microsoft.com/net/. Acessado em Setembro/2016.

Moser, A., Kruegel, C., and Kirda, E. (2007). Limits of static analysis for malware detection. *23rd Annual Computer Security Applications Conference*.

Pontiroli, S. (2014). Garfield garfield true, or the story behind syrian malware, .net trojans and social engineering. https://securelist.com/blog/virus-watch/59360/garfield-garfield-true-or-the-story-behind-syrian-malware-net-trojans Acessado em Setembro/2016.

Saul Schleimer, Daniel Wilkerson, A. A. (2003). *Winnowing: Local Algorithms for Document Fingerprinting*. http://theory.stanford.edu/~aiken/publications/papers/sigmod03.pdf. Acessado em Setembro/2016.

Wikipedia (2016a). Common intermediate language. https://en.wikipedia.org/wiki/Common_Intermediate_Language. Acessado em Setembro/2016.

Wikipedia (2016b). .net framework (wikipedia). https://en.wikipedia.org/wiki/.NET_Framework/. Acessado em Setembro/2016.