Universidade Federal de Pernambuco
Centro de Informática

Pós-graduação em Ciência da Computação

ASAT: UMA FERRAMENTA PARA DETECÇÃO DE NOVOS VÍRUS

Eduardo Mazza Batista

DISSERTAÇÃO DE MESTRADO

Recife
6 de junho de 2008

Universidade Federal de Pernambuco
Centro de Informática

Eduardo Mazza Batista

ASAT: UMA FERRAMENTA PARA DETECÇÃO DE NOVOS VÍRUS

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Orientador: *Ruy J. Guerra B. de Queiroz*
Co-orientadora: *Anjolina Grisi de Oliveira*

Recife
6 de junho de 2008

.

Batista, Eduardo Mazza
ASAT: Uma ferramenta para detecção de novos vírus / Eduardo Mazza Batista - Recife: O Autor, 2008.
xi, 50 folhas : il., fig., tab.

Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2008.

Inclui bibliografia.

1. Vírus de computador. I. Título.

005.84 CDD (22.ed.) MEI2008-074

Dedico este trabalho a minha amada esposa que me apoiou para tornar este sonho possível.

AGRADECIMENTOS

Gostaria de agradecer a minha esposa que me apoiou desde a concepção até a finalização do trabalho.

Gostaria de agradecer a minha família que teve todo carinho e consideração comigo, compreendendo meus momentos de ausência.

Gostaria de agradecer também ao meu orientador Ruy Guerra e co-orientadora Anjolina Oliveira pelo suporte com o tema escolhido apesar dos desafios que ele apresentou.

Gostaria de agradecer aos meus amigos, eternos companheiros que sempre me mostraram o caminho da solução entre as diversas conversas (mesmo que eles não tenham percebido).

Agradeço também ao Conselho Nacional de Desenvolvimento Científico e Tecnológico pela ajuda prestada através do processo de bolsa de auxílio financeiro.

Por fim, gostaria de agradecer ao Grande Arquiteto do Universo.

Tenha em mente que tudo que você aprende na escola é trabalho de muitas gerações. Receba essa herança, honre-a, acrescente a ela e, um dia, fielmente, deposite-a nas mãos de seus filhos.

—ALBERT EINSTEIN

RESUMO

O termo vírus de computador pode ser utilizado para definir de maneira geral qualquer programa que possua intenções maliciosas. É de interesse que, desde grandes empresas até usuários domésticos, exista proteção contra tais ameaças virtuais. A informação atualmente é vista como um bem muito valioso e de total importância. Existem vários casos de prejuízos causados devido a danos em informação confidenciais de empresas. Os ataques de vírus visam prejudicar de alguma forma esse patrimônio, seja danificando-o, por remoção ou alteração, ou até mesmo roubando-o e em alguns casos seqüestrando-o. Diversas medidas de segurança podem ser adotadas com o objetivo de proteger a informação, como é o caso de realização de cópias de segurança. Umas das medidas mais utilizadas para prevenir os ataques de vírus ainda consiste nos programas de detecção de vírus que funcionam através da análise do código de máquina dos executáveis. Com o intuito de enganar os programas de antivírus os criadores de vírus estão sempre criando novas versões de seus vírus com modificações em seus códigos, fazendo com que a taxa de detecção dos antivírus seja reduzida para vírus desconhecidos. Visando resolver este problema nosso trabalho propõe ASAT, uma ferramenta para detecção de novos vírus que funciona baseada em estatísticas calculadas utilizando o código de máquina de arquivos executáveis. O trabalho inclui também uma comparação dos resultados obtidos por ASAT com o desempenho de ferramentas de antivírus comerciais.

Palavras-chave: vírus de computador, antivírus, kits de construção de vírus

ABSTRACT

The term malware can be used to define in general any computer program that has malevolent intentions. It is desirable for large companies as much as for domestic users there exists protection against such virtual threats. Information nowadays is seen as a very precious asset. There have been many cases in which damage was caused by information loss in companies. The malware attacks aim at harming somehow this asset, deleting or altering or even stealing data and in some cases even kidnapping data. Many security measures can be adopted with to protect the information, as is the case of security backup copies. One of the security measures most used to prevent virus attacks consists of malware detection software that analyses the malware machine code. With the intention of fooling the antivirus software, virus creators are always updating theirs malware versions by modifying the code therefore reducing the antivirus detection rate of unknown malwares. Aiming at solving this problem we propose ASAT, a tool to detection of new malware that works based on statistical calculations made on the executable assembly code. The work also includes a comparative study between ASAT and commercial antivirus software.

Keywords: computer virus, antivirus, virus construction kits

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

CAPÍTULO 1

INTRODUÇÃO

O termo vírus de computador pode ser utilizado para definir de maneira geral qualquer programa que possua intenções maliciosas. É de grande interesse de grandes empresas, e até mesmo de usuários domésticos, que exista proteção contra tais ameaças virtuais. A informação atualmente é vista como um bem muito valioso e de total importância. Os ataques de vírus visam prejudicar de alguma forma esse patrimônio, seja danificando-a, por remoção ou alteração, ou até mesmo roubando-a e em alguns casos seqüestrando-a, como uma situação onde a intenção do vírus é criptografar informações relevantes e exigir um preço pelo seu resgate. Diversas medidas de segurança podem ser adotadas com o objetivo de proteger a informação, como é o caso de realização de cópias de segurança. Uma das medidas mais utilizadas para prevenir os ataques de vírus ainda são programas de detecção de vírus através da análise do seu código.

1.1 MOTIVAÇÃO

Sendo clara a necessidade de proteção contra tais ameaças virtuais, existe atualmente uma diversidade de produtos comerciais que apresentam soluções para o problema de detecção de vírus. Embora algumas destas propõem identificar vírus através de outros métodos, a maioria delas utiliza o método de detecção por assinatura para descobrir um vírus, que consiste basicamente em identificar pedaços de código, dentro do vírus, longos o suficiente para identificá-los de maneira única. A desvantagem deste método é de haver sempre a necessidade de uma instância do vírus para gerar a sua assinatura. Desta maneira, os antivírus comerciais ficam relativamente vulneráveis a novos tipos de vírus. Outra desvantagem do método de assinatura é a facilidade, conhecendo a assinatura, para o criador do vírus de modificar o seu código de forma que o seu funcionamento continue o mesmo; assim, a assinatura não seria mais capaz de identificá-lo.

Uma ferramenta que pudesse identificar eficientemente vírus sem conhecimento prévio de suas instâncias e que ao mesmo tempo criasse uma dificuldade ao criador do vírus de modificar o código do vírus, com o objetivo de passar despercebido pelo antivírus, seria de grande utilidade. É importante deixar claro que o método de identificação de vírus por assinatura também possui vantagens, como a baixa taxa de falso positivo e o seu alto desempenho. A proposta deste trabalho é o desenvolvimento de um método que possa, aliado ao método clássico de assinatura, melhorar os resultados da detecção de vírus para os quais não se tem conhecimento prévio.

1.2 METODOLOGIA

A metodologia aplicada para realização do trabalho consistiu em:

- Fazer um estudo geral sobre vírus. O intuito deste estudo foi obter um conhecimento aprofundado sobre o funcionamento dos vírus. Outro objetivo foi o de conhecer características possíveis de vírus que pudessem ser utilizadas como parâmetros para distingui-los de programas regulares.

- Estudar as soluções existentes que tratassem o problema de detecção de vírus. Era necessário saber quais eram as atuais características e técnicas de classificação utilizadas para tentar distinguir os arquivos de vírus dos programas. Das soluções pesquisadas foi dada ênfase àquelas que tinham suas características obtidas através de análise do código de máquina dos executáveis, fosse ela trabalhando diretamente com o código binário ou através da ajuda de um programa de engenharia reversa.

- Para que fossem observadas as características dos executáveis era necessário obter uma base de dados. Através de endereços eletrônicos especializados foi obtida uma base de vírus. Uma outra fonte para obtenção de vírus foi a sua criação, realizada através do uso de ferramentas para construção de vírus. Além de vírus, programas tiveram de ser coletados para que as suas características fossem comparadas com as dos vírus. Desta forma, a coleta de programas foi realizada através de escolha de executáveis aleatórios do sistema operacional e também obtidos em endereços eletrônicos.

- Os arquivos coletados tiveram que passar por um processo de coleta das características. Foram coletadas as mesmas características para todos os executáveis, sendo eles vírus ou programas. Os dados coletados também foram separados em dois grupos: um primeiro grupo com informações que foram utilizadas no treinamento dos classificadores; e um segundo grupo que serviu para comparar os desempenhos dos classificadores. Neste ponto foi gerada a hipótese de que as características coletadas eram capazes de detectar de forma eficiente os vírus.

- Com as características coletadas foi possível realizar o treinamento dos classificadores. Nem todas as características foram utilizadas por todos classificadores, visto que em cada classificador havia uma maneira de determinar as informações relevantes para obtenção de seu resultado final.

- A partir dos classificadores foram montados esquemas de decisão que envolviam os resultados finais de cada classificador. Cada esquema montado teve seus resultados registrados. A solução proposta envolve o esquema de extração das características e processamento das informações dos classificadores previamente treinados.

- Após a obtenção dos resultados foi feita uma análise do desempenho geral da solução proposta. Também foi realizado um estudo comparativo no qual o desempenho da solução é confrontado com o desempenho de antivírus comerciais. No fim foram avaliadas as conclusões e que outros caminho poderiam ser seguindos em trabalhos futuros.

  O esquema contendo a metodologia aplicada neste trabalho pode ser visto na Figura 1.1.

Figura 1.1 Metologia aplicada no trabalho

1.3 CONTRIBUIÇÕES

As principais contribuições deste trabalho foram:

1. Propor uma nova metodologia para detecção de novos vírus. Esta nova metodologia toma como hipótese que novos vírus em sua maioria irão conter pedaços de códigos que se assemelham ao de antigos vírus. Estes pedaços de códigos são comparados por estatísticas calculadas em cima do próprio código do vírus;

2. Propor uma maneira de identificar vírus de forma que dificulte, para o criador do vírus, gerar uma nova versão do mesmo vírus, capaz de passar sem ser percebida pela ferramenta de detecção.

1.4 ORGANIZAÇÃO DO TRABALHO

Além da presente introdução, este trabalho está organizado da seguinte forma:

- O Capítulo 2 trata de forma sucinta da evolução dos vírus de computador segundo a forma que seu código se comportava. Além disso, também são explicados os principais termos da terminologia de vírus;

- O Capítulo 3 discute os principais meios de resolução do problema. Depois apresenta um estudo feito sobre as principais soluções na área de detecção de vírus através da análise do seu código;

- O Capítulo 4 apresenta o desenvolvimento detalhado da solução proposta. É descrito neste capítulo todo o procedimento para a obtenção dos dados referentes à pesquisa. Também é explicada em maiores detalhes a metodologia aplicada para a obtenção dos resultados apresentados;

- No Capítulo 5 mostraram-se os resultados obtidos na solução com a utilização dos dados coletados. Adicionalmente, também realizou-se um estudo comparativo dos resultados obtidos pela solução e resultados de outras soluções comerciais para os mesmo dados;

- O Capítulo 6 conclui o trabalho apresentado fazendo uma revisão do desempenho e resultados obtidos e comenta que outros trabalhos pretendem ser realizados futuramente.

CAPÍTULO 2

VÍRUS DE COMPUTADOR

Uma das primeiras definições de vírus data de 1984, pelo matemático Frederick B. Cohen [Coh94], que foi o primeiro a introduzir tal termo por recomendação de seu orientador, Leonard Adleman [Adl90], que já havia realizado estudos na área e sugeriu o termo "vírus de computador" baseado em estórias de ficção científica. Para Cohen um vírus podia ser definido como um programa que é capaz de infectar outros programas modificando-os para incluir uma cópia evoluída de si mesmo. O fato de tal definição não abranger todos os vírus de computador encontrados hoje leva vários autores a discutirem sobre seu conceito. Em seus trabalhos Cohen [Coh86] também provou que seria impossível existir um antivírus capaz de detectar todos os futuros vírus de computador em um tempo finito. Cohen reduz o problema de detecção de vírus a outro problema indecidível, mostrando assim que não é possível resolver de forma definitiva o problema de detecção de qualquer vírus. Em 2000, Chess e White [CW00] estendem o trabalho de Cohen e mostram, de forma teórica, que existem vírus de computador que nenhum algoritmo é capaz de detectar.

Apesar dos trabalhos que mostram ser impossível construir um programa para detectar todos os vírus de computador, os estudos sobre técnicas de antivírus são vastos na literatura. Os danos anuais causados em empresas por causa de vírus podem chagar a milhões, sendo sempre crescentes os estudos de detecção de vírus. No início, a remoção de vírus era gerenciada de forma simples, pois existiam poucos vírus (menos de 100 vírus eram conhecidos no começo do ano de 1990) [Szo05]. Ainda outro fator que contribuía para a detecção de vírus era a baixa velocidade com que os vírus se espalhavam comparada com a velocidade com os vírus atuais. Este fato ocorria devido à baixa capacidade de processamento dos computadores e redes na época. Seria facilmente previsto que com a evolução dos computadores e programas os novos vírus estariam cada vez mais complexos e poderosos.

Neste capítulo, serão explicados rapidamente alguns termos da terminologia usada no ambiente de programas maliciosos. Depois, será descrito como os códigos dos vírus evoluíram. Também será descrito como as técnicas de detecção precisaram evoluir para acompanhar o desenvolvimento de novos tipos de vírus.

2.1 TERMINOLOGIA DE VÍRUS

A nomenclatura usada para definir os diversos tipos de programas maliciosos é descrita a seguir de acordo com [Szo05] que por sua vez baseia seu trabalho nos estudos de [Jac90].

Vírus de computador é um código que recursivamente replica uma possível cópia evoluída de si mesmo. Um vírus pode infectar um arquivo ou área do sistema,

como por exemplo, a memória RAM. Neste trabalho o termo "vírus de computador" será referenciado pelo simples termo "vírus".

Criador de vírus será o termo utilizado aqui para descrever uma pessoa que constrói vírus com o propósito de prejudicar outros ou de obter vantagens de modos ilegais.

Worms são considerados vírus que se espalham em redes de computadores. Comumente *worms* são executados automaticamente sem que dependa de nenhuma ação do usuário.

Cavalos de Tróia (*Trojan horses*) são programas que se fazem parecer inofensivos para que o usuário não perceba suas ações maliciosas. Um exemplo de cavalo de Tróia seria se um criador de vírus desenvolvesse um jogo de computador que funcionasse normalmente, porém ao jogar o usuário teria todos seus arquivos pessoais removidos do sistema. Outro exemplo de cavalo de Tróia pode ser verificado quando são construídas versões alteradas de programas existentes, como por exemplo, o caso de um criador de vírus desenvolver uma versão alterada do programa "ps" no UNIX. Este programa serve para listar os processos que estão atualmente rodando no processador. O criador do vírus poderia alterar este programa de forma que escondesse, da lista de processos, outro programa que pudesse causar algum dano à máquina do usuário. Este último tipo de cavalo de Tróia é mais freqüente em sistemas abertos onde os códigos maliciosos podem ser inseridos mais facilmente no código fonte dos programas. Um exemplo é o cavalo de Tróia de nome `Linux.Trojan-Spy.Alk` que ataca o sistema operacional Linux.

Ferramentas (kits) de criação de vírus são programas capazes de gerar, através de um processo facilitado, como interfaces gráficas e linhas de comando, diversos vírus diferentes. Com tais ferramentas até mesmo usuários sem conhecimento sobre linguagens de programação de baixo nível podem desenvolver vírus poderosos. Existem exemplos de vírus que causaram grandes danos e foram criados por adolescentes através da utilização de tais ferramentas. Este é o caso do vírus "Anna Kournikova" [Szo05], tecnicamente chamado de `VBS.VBSWG.J`.

2.2 EVOLUÇÃO DOS CÓDIGOS

Criadores de vírus estão sempre testando novas maneiras de "enganar" os programas de antivírus. Para tentar passar despercebidos diversos novos tipos de vírus foram criados. Da mesma forma, os desenvolvedores de antivírus precisam criar técnicas cada vez melhores para detectar e remover os novos vírus. A primeira técnica usada por desenvolvedores de antivírus (que ainda é a mais utilizada) para detecção de vírus foi tentar achar um pedaço de código do vírus capaz de identificá-lo unicamente. Este pedaço de código deveria ser grande o suficiente para não correr o risco de ser encontrado em outros arquivos que não possuíssem o vírus. A esta seqüência de código dá-se o nome de "assinatura do vírus". Posteriormente as assinaturas de vírus começaram a abranger mais informações como o tipo de vírus e tipos de arquivos que cada um dos vírus atacava.

A classificação de vírus, de acordo com [Szo05], segundo a evolução do seu código, bem como a evolução das técnicas de detecção de vírus, pode ser descrita como segue.

2.2.1 Vírus Encriptados

Uma das primeiras maneiras, usadas pelos criadores de vírus, para esconder os códigos dos vírus foi o uso de criptografia. Os vírus eram armazenados de forma criptografada e acompanhavam códigos para que, ao serem carregados na memória, fossem descriptografados de tal forma que o código do vírus nunca ficasse exposto em nenhum arquivo do sistema. Restava aos antivírus tentar reconhecer os vírus através da identificação do código usado na criptografia. Porém, logo surgiram os vírus que faziam uso de outras técnicas para descriptografar seu código, como por exemplo, tentando quebrar a criptografia por força bruta, ou ainda dividindo o código de criptografia em vários pedaços espalhados pelo código do arquivo. Outra técnica utilizada pelos programas de antivírus era a procura pelo código do vírus depois de terim sido carregados na memória.

2.2.2 Vírus Polimórficos

Ao perceber que os programas de antivírus poderiam detectar seus vírus através do código de criptografia, os criadores de vírus passaram a escrever códigos que pudessem mudar o estilo de criptografia a cada nova cópia do vírus, desta forma gerando milhares de versões diferentes do mesmo vírus. Cada nova versão de um vírus era chamada de "variante" e ao conjunto de variantes era dado o nome de "família". O primeiro vírus polimórfico se chamava "`1260`". Foi escrito em 1990 e utilizava diversas técnicas já previstas nos trabalhos de Fred Cohen [Coh86], dentre elas a habilidade de inserir código inútil em seu código de criptografia a cada nova instância gerada. Este novo estilo de vírus desafiou os desenvolvedores de antivírus, pois não era mais possível identificar uma seqüência de código comum a todas variantes de um vírus polimórfico.

Para que os programas de antivírus pudessem detectar tais versões de vírus, era necessário estudar os diversos tipos de códigos de criptografia e tentar identificar um pedaço de código comum a todos os tipos de vírus. O resultado foi o desenvolvimento de assinaturas de vírus mais complexas. As assinaturas agora teriam que conter condições e abranger diversas regras para identificação de um vírus. Neste ponto, a maioria dos programadores de antivírus foi obrigada a desenvolver máquinas virtuais para simular o comportamento do vírus em um ambiente seguro. Esta técnica consistia na execução do vírus em um ambiente simulado, onde cada executável tinha suas ações observadas e processadas para verificar se ele havia tido o comportamento de um vírus. Embora esta técnica exigisse mais do processamento do computador era necessária para compreender melhor o comportamento dos vírus polimórficos.

2.2.3 Virus Metamórficos

Vírus metamórficos são vírus que possuem a capacidade de mudar todo ou parte do seu corpo a cada nova infecção. São vírus capazes de criar novas instâncias que apesar de possuírem o mesmo funcionamento possuem o seu código escrito de formas diferentes. Um

dos primeiros vírus metamórficos apareceu quando um famoso criador de vírus, conhecido pelo codinome de "Vecna", criou o vírus chamado "`W95.Regswap`". Este vírus era capaz de mudar os registradores utilizados em todo seu código para que cada nova instância parecesse diferente da instância que a gerou. Tal técnica não envolvia alto nível de complexidade e os vírus poderiam ser identificados facilmente com o uso de caracteres coringas em suas assinaturas. Outros vírus tentaram utilizar técnicas como a permutação de partes do seu código de forma que não fosse afetado o funcionamento do vírus. Para que isso fosse possível bastava apenas que instruções de salto (*jump*) fossem distribuídas nas diversas partes do código de forma que garantisse a seqüência lógica do fluxo de execução do código original.

Versões mais complexas de vírus metamórficos começaram a ser criadas em meados de 2000. Estas versões possuíam uma porção de código metamórfica (*metamorphic engine*) que era capaz de gerar milhares de versões diferentes de um mesmo código. Um exemplo de variação de um mesmo código realizada pelo vírus `W32.Evol` pode ser vista na Figura 2.1. O código de um vírus metamórfico não fica constante nem mesmo na memória do computador, pois até mesmo ao ser carregado para execução o vírus pode aparecer de diferentes formas. A mudança de código também pode ser realizada pela substituição de suas instruções por outras de mesmo valor. Um exemplo seria a substituição da instrução `XOR EAX, EAX`, que tem como objetivo zerar o valor do registrador `EAX`, pela instrução `SUB EAX, EAX`, que significa subtrair do valor do registrador `EAX` o valor dele mesmo, resultando assim a substituição do valor de `EAX` por zero.

2.2.4 Ferramentas (kits) de criação de vírus

O fato da maioria dos vírus serem escritos na linguagem Assembly inspirou muitos criadores de vírus a desenvolver ferramentas que pudessem facilitar o desenvolvimento dos mesmos. Desta forma até mesmo com poucos conhecimentos de programação é possível criar um vírus poderoso que possua técnicas de criptografia, tipos específicos de arquivos para infectar e até mesmo técnicas para evitar a simulação em máquinas virtuais. Atualmente são conhecidas dezenas de ferramentas para criação de novos vírus, que em sua maioria foram feitas para gerar vírus para o sistema Microsoft Windows porém também é possível encontrar tais ferramentas para outros sistemas como é o caso da ferramenta PolyEngine.Linux.LIME.poly que é capaz de gerar vírus para o sistema Linux. Disponibilizadas em endereços eletrônicos, como VXHeavens [VXH], é possível obter diversas destas ferramentas, além de tutoriais para diversas técnicas de criação de vírus. Nos trabalhos de [WS06] foram comparados diversos kits de criação de vírus utilizando uma medida de semelhança entre programas para comparar os vírus gerados. A ferramenta que mostrou os melhores resultados foi a NGVCK (Next Generation Virus Construction Kit). Esta ferramenta possui uma interface gráfica que permite ao criador do vírus escolher entre diversas técnicas de criptografia e estratégias de espalhamento pela Internet (no caso de um *worm*), como pode ser visto na Figura 2.2.

A. Primeira instância:

```
movx dword ptr [esi],5500000Fh
mov dword ptr [esi+0004],5151EC8Bh
```

B. Segunda instância:

```
mov edi,5500000Fh
mov [esi],edi
pop edi
push edx
mov dh,40
mov edx,5151EC8Bh
push ebx
mov ebx,edx
mov [esi+0004],ebx
```

C. Terceira instância:

```
mov ebx,5500000Fh
mov [esi],ebx
pop ebx
push ecx
mov ecx,5FC000CBh
add ecx,F191EBC0h ;ecx = 5151EC8Bh
mov [esi+0004],ecx
```

Figura 2.1 Diferentes instâncias do vírus `W32/Evol`

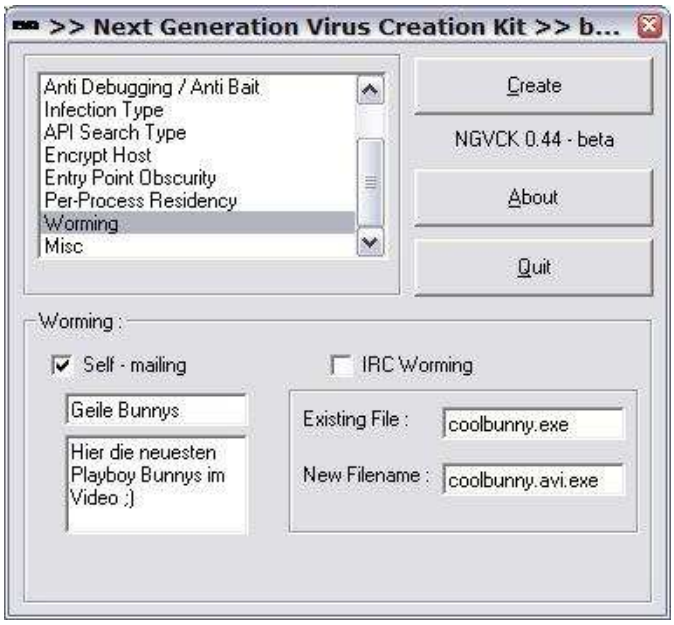

Figura 2.2 Interface grágica da ferramenta NGVCK

CAPÍTULO 3

ESTADO DA ARTE

Neste capítulo será apresentado como se encontra o estado da arte em detecção de vírus. Existem diversos trabalhos na área, porém dedicação foi dada aos trabalhos que fazem análise do código de máquina gerado a partir de ferramentas de engenharia reversa (*disassembler*), visto que o presente trabalho é baseado em dados obtidos do processo de engenharia reversa em vírus. Neste caso, apenas técnicas estáticas foram analisadas.

3.1 MINERAÇÃO DE DADOS

Dentre os trabalhos que usam mineração de dados para detecção de vírus destaca-se o trabalho de [SEZS01] como um dos primeiros trabalhos a utilizar inteligência artificial na detecção de vírus. Neste trabalho, os autores criam um *framework* para detecção de novos vírus que gera como resultado um conjunto de heurísticas para classificação dos arquivos. As técnicas de classificação usadas no *framework* foram um algoritmo gerador de regras chamado RIPPER e o classificador ingênuo de Bayes. As características usadas para classificar os vírus variavam para cada técnica aplicada.

No trabalho de [SEZS01] inicialmente foi construído um repositório com diversos executáveis, dentre eles vírus e programas regulares. Para obter características dos arquivos coletados os autores utilizaram ferramentas para extrair informações que se encontravam no cabeçalho dos executáveis, informações estas como, por exemplo, presença ou não de funções e bibliotecas com chamadas de execução em cada executável. Outra abordagem usada para a extração de características foi a utilização de ferramentas que pudessem extrair as seqüências de bytes mais comuns entre os executáveis. Esta última técnica, embora não citada no artigo, é também conhecida como N-gram.

Como resultado do treinamento pelo algoritmo RIPPER foram geradas regras que classificam um executável como vírus se a resposta de qualquer uma das seguintes perguntas for verdadeira:

- Não chama a função `EndDialog` da biblioteca `user32.dll` mas chama `EnumCalendarInfoA` da biblioteca `kernel32.dll`?
- Não chama as funções: `LoadIconA` da biblioteca `user32.dll`, `GetTempPathA` da biblioteca `kernel32.dll` ou qualquer função da biblioteca `advapi32.dll`?
- Chama a função `ExtractAssociatedIconA` da biblioteca `shell32.dll`?
- Chama qualquer função na biblioteca `msvbbm.dll`?

Outros tipos de entradas também foram utilizados no algoritmo RIPPER, como o número de funções chamadas de cada biblioteca, porém a execução do algoritmo concluiu que as

variáveis não eram relevantes a ponto de aparecerem nas regras. Como resultados, o algoritmo RIPPER apresentou uma taxa de detecção próxima de 90%, porém as taxas de falso positivo atingiram mais de 9%. O fato de ter uma grande taxa de falso positivo faz com que o algoritmo RIPPER não seja um algoritmo ideal para ferramentas de usuários domésticos.

As regras geradas pelo classificador ingênuo de Bayes neste caso específico obedeceram à forma $P(F|C)$ onde F indica o evento de encontrar uma determinada seqüência de caracteres no código e C indica a classe a ser classificada, que neste caso pode ser vírus ou não-vírus, e $P(F|C)$ indica a probabilidade condicional do evento F acontecer dado que o arquivo é da classe C. Um exemplo de regra gerada pelo classificador é:

- P("windows" — não-vírus) = 45/47
- P("windows" — vírus) = 2/47

Neste caso nota-se que a seqüência "windows" é muito mais encontrada em arquivos que não são vírus do que arquivos que são. Através da combinação do resultado de diversos classificadores de Bayes é possível classificar o arquivo usando a seguinte função:

$$Classe = \max_{C} \Big(P(C) \prod_{i=1}^{n} P(F_i|C) \Big),$$

onde $P(C)$ é a probabilidade do arquivo, a priori, pertencer à classe C e $P(F_i|C)$ é o resultado de cada uma das regras do classificador para cada uma das n palavras mais comuns. A função $\max_C$ é a função que escolhe a classe que corresponde ao maior do produtos calculados para cada classe.

Os resultados obtidos pelo classificador ingênuo de Bayes apresentam taxas de detecção ainda maiores que as do algoritmo RIPPER, com índice de falso positivo abaixo de 6%. A outra vantagem do resultado apresentado pelo classificador de Bayes sobre o algoritmo de classificação RIPPER é a facilidade de um criador de vírus quebrar a segurança do algoritmo RIPPER através da manipulação das funções utilizadas no código para não utilizarem os recursos que o identificariam como um vírus, enquanto que para alterar as seqüências mais comuns encontradas, como são utilizadas pelo classificador de Bayes, seria necessária uma mudança significante no código do arquivo.

3.2 ANÁLISE DA SEQUÊNCIA DE INSTRUÇÕES

Dentre os artigos que utilizam técnicas estáticas para detectar vírus destacam-se aqueles onde a análise é feita considerando a seqüência de instruções no código de máquina. Isto é, estudou-se a ordem como determinadas instruções aparecem no código fonte. O estudo é feito para poder considerar a hipótese que vírus, diferentemente de programas regulares, possuem determinadas seqüências de instruções características do seu comportamento. O objetivo final era identificar tais seqüências de maneira estática, ou seja, apenas com a análise do código fonte.

3.2.1 Static Analyzer of Vicious Executables (SAVE)

O trabalho de [SXCM04] apresenta uma técnica de detecção de vírus baseada em assinatura focada na detecção de variantes de um mesmo vírus. Para desenvolver esta técnica, os autores usam a hipótese de que todas as versões de um mesmo vírus possuem a mesma assinatura de código, que por sua vez é gerada através de diversas características no seu código. O método proposto funciona primeiramente analisando uma das variantes de uma família de vírus e posteriormente gerando uma assinatura baseada na seqüência de operações feitas ao sistema operacional contidas nos códigos de tais vírus.

No começo no trabalho os autores explicam o conceito de ofuscamento de código, ou seja, a mudança no código fonte de um programa, de forma que a funcionalidade não seja alterada. Um exemplo de ofuscamento de código é a inserção de operações do tipo NOP (*no-operation*) ou operações matemáticas que não alteram nenhum valor como uma adição ao valor zero ou multiplicação pelo valor um. Também é possível realizar ofuscamento de códigos mais complexos, como é o caso da mudança no fluxo de execução de um programa através do uso de instruções de mudança de fluxo para novos rótulos (*labels*). Na linguagem Assembly os rótulos servem para determinar pontos de referências específicos no código, sendo possível depois mudar o fluxo de execução atual para qualquer um dos rótulos através de instruções de salto.

Para demonstrar que a segurança de antivírus comerciais pode ser quebrada com simples operações de ofuscamento quatros vírus foram submetidos a operações de ofuscamento cada vez mais complexas até que os antivírus comerciais não fossem capazes de detectá-los. No final foram armazenadas todas as variantes dos vírus que não foram capazes de serem detectadas por qualquer um dos antivírus comerciais. Este experimento mostrou que não é necessária uma grande mudança no código dos vírus para que eles passem despercebidos pelos programas de antivírus. A maioria dos antivírus foi incapaz de detectar variantes de vírus após uma simples mudança no seu ponto de entrada (*entry point*), que pode ser realizada apenas com a adição de duas simples instruções de salto em seu código.

Após construir um repositório de variantes dos quatro tipos de vírus os autores propuseram uma maneira de medir a similaridade entre dois arquivos, usando como entrada a seqüência de chamadas em bibliotecas do sistema operacional presente no código. A medida de similaridade entre duas seqüências utilizada em [SXCM04] consistiu primeiro no cálculo do melhor alinhamento entre as duas seqüências. Depois que as seqüências são alinhadas, as medidas de similaridade de Cosine, Jaccard e correlação de Pearson são computadas e uma média aritmética é gerada de seus valores. O motivo da utilização de três medidas se similaridade foi o fato de nenhuma delas isoladamente conseguir determinar eficientemente todas comparações. Para determinar se um executável em questão era ou não um vírus calculou-se a média das medidas de similaridade entre o executável e todas as outras seqüências de um dicionário de seqüências encontradas em antivírus. Se o valor da média atingia um certo limite (determinado através de experimentos) classificava-se o executável como um vírus.

No final do trabalho os autores apresentaram uma comparação realizada entre a técnica proposta e os antivírus comerciais. Mostrou-se que a nova técnica foi capaz

de detectar todas as variantes do repositório. Apesar dos resultados positivos, não foram realizados quaisquer experimentos para medir os casos em que os programas regulares eram classificados como vírus. Uma análise com programas regulares e uma base de vírus maior que quatro tipos seria necessária para mostrar com maior segurança a eficiência da técnica proposta em [SXCM04].

3.2.2 Sistema Livre de Assinatura

Para estudar arquivos infectados com vírus os autores de [YUN06] fizeram uso da técnica de mapas auto organizáveis [Koh95] (*self organized maps* - SOM). Tais mapas consistem em redes neurais que utilizam uma estratégia de algoritmo de aprendizagem não supervisionada, ou seja, uma rede neural onde o usuário não especifica as saídas desejadas, que é o caso de uma aprendizagem supervisionada. Porém, antes de servir como entrada para os SOMs os arquivos infectados tiveram que passar por um processo de padronização dos dados. Neste processo, os arquivos binários eram transformados em uma tabela de valores numéricos, que serviria de entrada para o SOM.

O processo de padronização começou com a divisão do arquivo a ser analisado em diversas linhas e cada linha dividida em oito células, de quatro bytes cada. Desta forma cada célula da tabela representou um pedaço de informação binária do arquivo, que depois foram convertidos em um número inteiro. É importante salientar que esta forma de divisão não é comum em SOM onde, de maneira usual, cada linha da tabela de entrada deve representar uma instância da amostra e as colunas representam as variáveis, fazendo desta forma com que as linhas contenham os valores para as variáveis de uma instância.

O processo de treinamento e visualização do SOM, descrito em [YUN06] utilizou as informações dos arquivos em tabelas que, depois de padronizados, foram ainda divididos em um número definido de grupos (quatro grupos neste caso) onde cara grupo continha um número aproximadamente igual de linhas. O processo regular de treinamento da rede é realizado e como resultado obteve-se uma matriz onde a informação dos grupos ficava distribuída de forma organizada, chamada de Umatrix (*unified distance matrix*). Na Umatrix cada célula recebia um valor numérico que correspondia à sua densidade local, ou seja, à distância média entre a célula e seus vizinhos. É fácil depois transformar os valores da Umatrix em uma representação visual com um esquema de cores, onde valores baixos são representados como cores claras (quentes) e cores escuras (frias) para representar valores altos. Um exemplo de representação pode ser visto na Figura 3.1. Embora visualizações tivessem sido geradas, a técnica proposta trabalhou diretamente com os valores da Umatrix.

O passo seguinte foi a interpretação dos valores da Umatrix para poder identificar o código do vírus dentro do arquivo infectado. Pela observação de imagens da Umatrix de um arquivo, antes e depois de ser infectado, era possível identificar marcas no meio do código indicadas por cores escuras, que representam justamente o código do vírus. Um exemplo da visualização de dois arquivos antes e depois de serem infectados pelo mesmo vírus (`Win95.CIH`) pode ser visto nas Figura 3.2 e Figura 3.3, respectivamente. O processo de detecção de vírus começou com a transformação da Umatrix e uma representação de matriz em apenas duas cores, preto para altos valores de densidade, que é o caso do

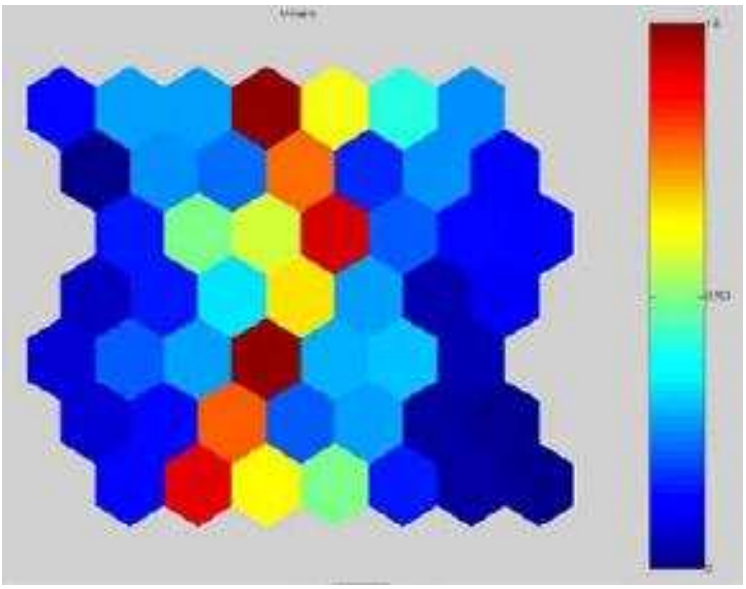

Figura 3.1 Exemplo de representação gráfica da Umatrix

código do vírus, e branco para baixos valores. O valor que determina se a densidade será considerada, alta ou baixa foi obtido através de experimentos. Após a transformação da Umatrix em uma tabela de valores preto ou branco o número de células próximas uma das outras com valor preto iria determinar se o arquivo estava ou não infectado por vírus.

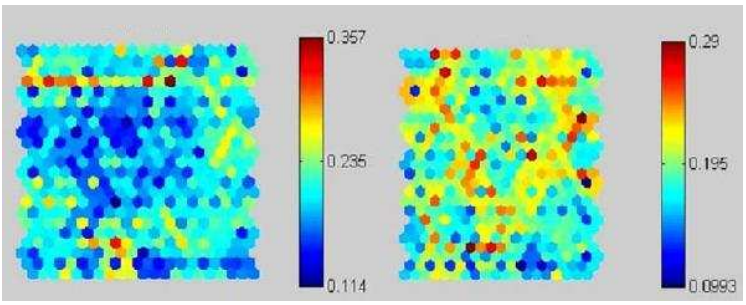

Figura 3.2 Exemplo de representação gráfica da Umatrix de dois arquivos antes de serem infectados

É fácil perceber que quanto menor fosse o limite para considerar um valor da Umatrix alto, maior seria a chance de aparecer valores pretos na matriz usada para classificar, ou seja, maior seria a chance de encontrar no arquivo um vírus. De certa forma isto ajudaria a diminuir a taxa de falso negativo, porém ao mesmo tempo passaria a considerar arquivos não infectados como se possuísse o código do vírus, aumentando assim a taxa de falso positivo. Para ilustrar o fato os autores construíram um gráfico que pode ser visto na Figura 3.4. Neste gráfico pode-se observar o valor da taxa de falso negativo e falso positivo de acordo com a variação do limite que determina se um valor da Umatrix será classificado como preto ou branco.

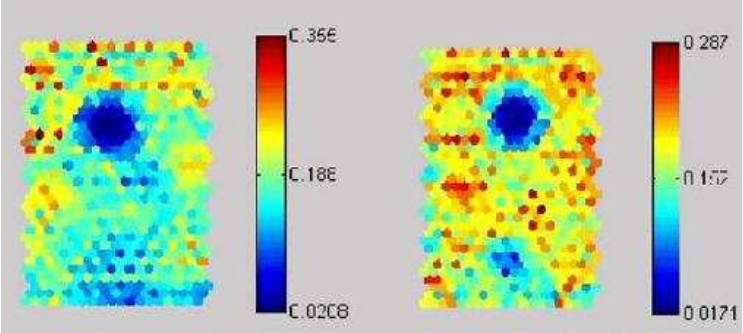

Figura 3.3 Exemplo de representação gráfica da Umatrix dos arquivo da Figura 3.2 infectados pelo vírus `Win95.CIH`

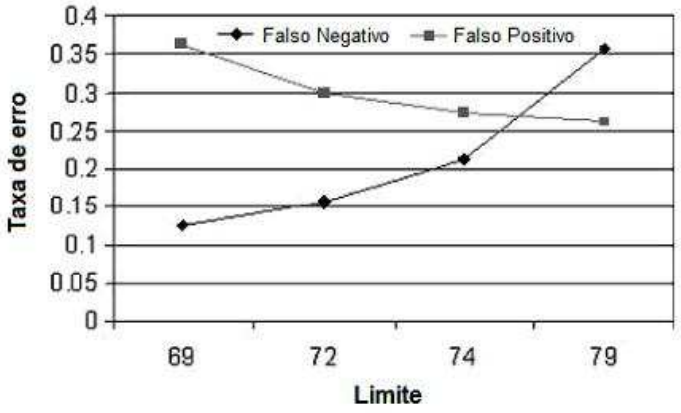

Figura 3.4 Gráfico da taxa de falso positivo e falso negativo segundo o limite utilizado para considerar se o arquivo está ou não infectado

3.2.3 Cadeias de Markov Escondidas

No trabalho [WS06] foi apresentada uma proposta para detecção de novos vírus baseado na seqüência de operações de código de máquina. Neste trabalho os autores utilizam cadeias de Markov escondidas (*hidden Markov model*) para determinar se uma seqüência de códigos de operações de máquina aproxima-se ou não do padrão encontrado em um vírus. Para a realização do experimento os autores primeiramente criaram diversos vírus utilizando cinco ferramentas diferentes de criação de vírus. Além dos vírus também foram utilizados alguns programas regulares escolhidos aleatoriamente no sistema operacional.

Cadeias de Markov escondidas são modelos matemáticos nos quais o sistema modelado é suposto seguir um processo de Markov com parâmetros desconhecidos. Existem algoritmos eficientes que são capazes de, partindo da realização de experimentos, estimar os parâmetros desconhecidos. Utilizam-se Cadeias de Markov escondidas em diversas aplicações de reconhecimento de padrões, como, por exemplo, reconhecimento de voz e reconhecimento de escrita.

Também, em [WS06], mostrou-se como resolver o problema de determinar se um dados texto está escrito ou não em inglês com a ajuda de cadeias de Markov escondidas. O

algoritmo proposto consegue ainda identificar textos em inglês mesmo que eles apresentem alguma criptografia por substituição. A idéia do autor foi utilizar uma adaptação deste algoritmo para que através de seqüências de operações de código de máquina de vírus fossem treinadas cadeias de Markov capazes de classificar corretamente se um dado executável é ou não um vírus.

Para o treinamento da cadeia de Markov foi realizada a criação de 200 exemplos de vírus por diversas ferramentas de construção de vírus. Quando treinadas com múltiplas seqüências, as cadeias de Markov possuem a propriedade de como resultado apresentar o comportamento da média de todas seqüências. Baseado nesta propriedade os códigos dos vírus foram concatenados e serviram como dados de treinamento para diversas cadeias de Markov com o número de estados escondidos variando de dois a seis e o número de operações consideradas variando de 70 a 80. Após o treinamento foi gerado para teste um repositório contendo vírus gerados pela mesma ferramenta de construção de vírus do treinamento, vírus gerados por outras ferramentas e arquivos que não eram vírus.

Nos testes dos resultados com as diversas cadeias geradas foi possível perceber que existe uma diferença nos resultados obtidos calculando a probabilidade de uma determinada seqüência de operações pertencer ou não à família de vírus para o qual a cadeia foi treinada. Embora tenham havido casos de falsos positivos, estes ocorreram apenas em vírus que não foram gerados pela ferramenta utilizada para o treinamento e, desta forma, os autores não consideraram tais casos como falhas.

Apesar do trabalho [WS06] ter apresentado resultados negativos para detecção de vírus gerados pela mesma ferramenta de criação, o sistema proposto ainda tem necessidade de que, para cada nova ferramenta de criação de vírus, seja construída uma nova cadeia de Markov para classificar seus vírus gerados. Mesmo não sendo estudado no trabalho, o desempenho da técnica proposta depende da comparação de uma instância com diversas cadeias para verificar se ela pertence a cada uma das famílias de vírus, porém sabe-se que o algoritmo proposto para cálculo da probabilidade é eficiente.

3.3 ANÁLISE DO GRUPO DE INSTRUÇÕES

Outra maneira de analisar o código fonte é através do estudo do grupo de instruções que juntas, ou associadas a outro grupo de instruções, são capazes de identificar comportamentos de vírus em arquivos executáveis. Existem trabalhos que fazem análise do código de máquina para poder extrair instruções que aparecem em grande parte de variantes dos vírus mesmo depois da aplicação de transformações metamórficas.

3.3.1 MECiC

No trabalho de [SRMS05b] também foi desenvolvida uma ferramenta chamada MEDiC (*Malware Examiner using Disassembled Code*) para detecção de vírus baseada no código de máquina do executável. A ferramenta foi desenvolvida baseado na hipótese de que códigos de vírus possuem em comum certos grupos de instruções em seu código de máquina. O objetivo da ferramenta é detectar importantes grupos de códigos de máquina para poder gerar uma assinatura que possa detectar qualquer variante de um determinado

vírus. Usando ferramentas de engenharia reversa um repositório de vírus foi analisado para detectar os conjuntos de instruções mais relevantes na classificação entre vírus e programas.

Para determinar as partes mais relevantes nos vírus primeiramente foi realizada uma análise de cada código fonte dos arquivos do repositório de vírus e cada par composto por rótulo/conjunto de instruções foi identificado. Para tais pares os autores deram o nome de "ilhas de código" (*code islands*). Se o número de instruções no conjunto fosse maior ou igual ao número a um limite determinado, o par era considerado uma ilha de código importante e sendo assim era armazenada em um dicionário onde eram salvas todas as ilhas de código que deveriam ser observadas na detecção de uma variante do vírus.

Após análise do repositório de vírus o algoritmo de detecção da ferramenta consiste simplesmente em observar quantas vezes as ilhas de código do arquivo analisado eram encontradas no dicionário construído. Se o número de vezes superasse um limite estabelecido o arquivo analisado era considerado um vírus. Se mesmo depois da primeira análise o arquivo não fosse considerado um vírus uma segunda análise era feita, só que desta vez ilhas de código com nomes de rótulos diferentes e conjunto de instruções iguais eram computadas também, e novamente o resultados comparados com o limite estabelecido. Se mesmo assim o arquivo não fosse considerado vírus, um terceiro processo era executado, porém desta vez além de não considerar rótulos com nomes diferentes, como no segundo passo, eram consideradas semelhantes também as ilhas de código onde as ordens das instruções diferem.

Na conclusão de [SRMS05b] foi feita uma comparação entre a ferramenta construída e diversos antivírus comerciais. Os resultado apresentado mostrou que a ferramenta MEDiC foi capaz de detectar variantes de vírus que outros antivírus comerciais classificaram como programa. As variantes foram obtidas através da manipulação de vírus conhecidos. Nos códigos dos vírus foram inseridas operações irrelevantes (como NOP ou uma adição ao valor zero) além de transposições de códigos. No trabalho é citado também que não houve casos de falso positivos. Uma das vantagens da abordagem utilizada é que por utilizar código de máquina a ferramenta poderia ser facilmente adaptada para detectar vírus em qualquer sistema operacional.

3.3.2 Assinatura de Motores Metamórficos

Nos trabalhos de [CL06] os autores propuseram um modelo de avaliação de vírus gerados pelo mesmo kit de criação de vírus ou pelo mesmo motor metamórfico. O método de detecção proposto baseou-se na observação de que a maioria das instruções da variante, derivada do vírus original, foi transformada pelo motor metamórfico. A idéia é determinar o quanto um motor metamórfico está presente nas transformações de um dado vírus. Ou seja, o objetivo é determinar a densidade das transformações presentes no código do vírus depois que foi modificado pelo motor metamórfico.

Seja E um motor metamórfico capaz de transformar instruções de código por outras sem alterar a funcionalidade do programa. O trabalho começa definindo o conceito de "pista" (*clue*) referindo-se como qualquer fração, tipicamente uma seqüência de instruções, de qualquer informação extraída de um segmento de código, sugerindo que este

segmento possa ter sido gerado pelo motor E. Também se denotou por T como sendo o conjunto de regras associadas a E. Cada regra contida em T mapeia uma instrução (lado esquerdo da regra) a uma seqüência de uma ou mais instruções (lado direito da regra). Assim, os lados direitos das regras são exemplos de pistas. Tais pistas são selecionadas e atribuídas a pesos iguais ao seu número de instruções que contêm em seu código.

Dada as definições de pista e regra, definiu-se que um código é E-amigável (E-*friendly*) se foi escrito tendo o motor E como referência, isto é, seu código tem ocorrências no lado esquerdo do conjunto de regras T. Um índice maior de E-amizade (E-*friendliness*) deve ser obtido através de variantes de um vírus geradas pelo motor E. Este índice é obtido através de ocorrências dos lados esquerdos das regras associados ao lado direito das mesmas. Cada uma das regras de T acompanha uma probabilidade de aplicação, que representa a probabilidade de que ela tenha sido aplicada dado que seu lado esquerdo foi encontrado. O conjunto de regras T assume-se ser encontrado através de uma análise manual ou através de diversos experimentos com o motor metamórfico em um ambiente controlado. Exemplos de regras que envolvem o motor metamórfico do vírus `W32.Evol` podem ser vistos na Figura 3.5

```
mov [esi+4], 9      →    mov [esi+4], 6
                         add [esi+4], 3
mov [ebp+8], ecx    →    push eax
                         mov eax, ecx
                         mov [ebp+8], eax
                         pop eax
push 4              →    mov eax, 4
                         push eax
push eax            →    push eax
                         mov eax, 2Bh
```

Figura 3.5 Exemplo de regras encontradas no vírus `W32.Evol`

A função que determina o índice de E-amizade para um dado motor E recebe como entrada um segmento de código e retorna um número que indica quão possível foi achar transformações de E dentro deste código. Para um dado segmento de código V, o índice de E-amizade de V é denotado por $S_E(V)$. A medida pode ser interpretada como quanto V é parte de um código gerado por E. A medida $S_E(V)$ é diretamente proporcional ao número de regras de E encontradas em V e inversamente proporcional ao número de instruções de V. Desta maneira, $S_E(V)$ também poderia ser vista como a densidade em V de pistas de transformações geradas por E. Devido aos pesos atribuídos a cada uma das regras a expressão $S_E(V)$ leva em conta o fato de que algumas pistas são mais relevantes que outras. A expressão de definição formal de $S_E(V)$ é dada por:

$$S_E(V) = \frac{\sum_C \sum_X w_C e_{CX}}{|V|},$$

onde $|V|$ é equivalente ao total de instruções de V, w_C é o peso da pista C e e_{CX} será igual a um se a pista C estiver na instrução X ou igual a zero caso contrário. O algoritmo para cálculo de $S_E(V)$ pode ser interpretado como uma análise de cada instrução X de V. Se X contiver o começo de uma ou mais pistas de E o peso da pista é armazenado em uma variável. No final esta variável será dividida pelo número total de instruções de V e este valor será o resultado final.

Para avaliar os resultados do trabalho os autores fizeram um protótipo, que passou por duas avaliações distintas. A primeira avaliação teve como objetivo analisar quão bem e para quais escolhas de parâmetros a função que avalia o índice $S_E(V)$ pode ajudar a classificar corretamente entre códigos gerados pelo motor E e códigos arbitrários. Esta avaliação consistiu em rodar vários experimentos. Para cada experimento foram criadas versões de códigos, consideradas como a primeira geração, que possuíam um mesmo índice de E-amizade. Em seguida utilizando o motor de transformações do vírus `W32.Evol` foram geradas mil versões para cada nova geração da segunda até a sétima geração descendente do segmento de código inicial. Os resultados estão exibidos na Figura 3.6.

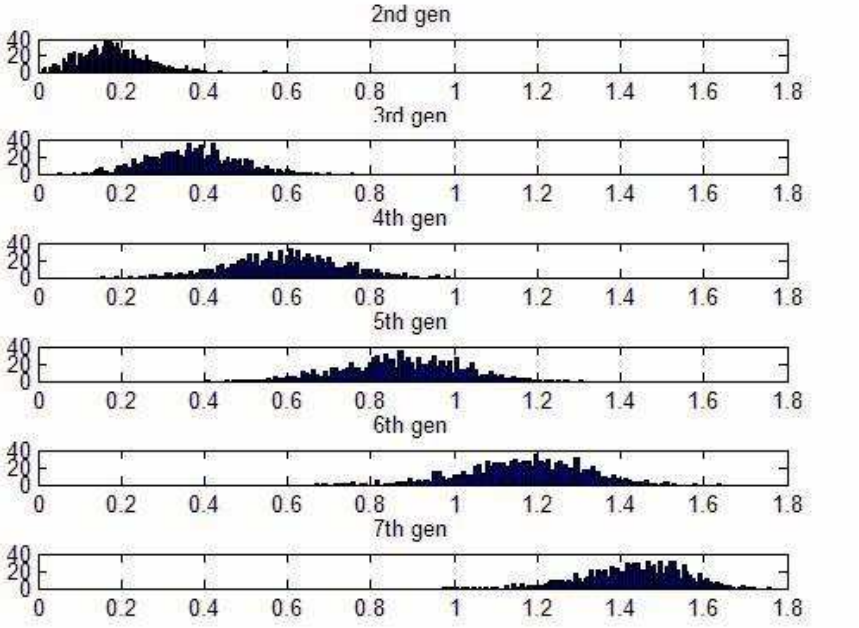

Figura 3.6 Distribuição de freqüência dos índices da segunda até a sétima geração

O segundo experimento teve como objetivo analisar quão bem, e para que escolha de parâmetros, a função do índice $S_E(V)$ pode ajudar a distinguir entre variantes de um vírus conhecido e códigos arbitrários. Para realizar este experimento os autores extraíram as operações de código de uma instância do vírus `W32.Evol` e, utilizando as regras de transformações que o motor deste vírus usa em seu código, foram geradas mil variantes da segunda até a quarta geração. Foram analisados os índices de E-amizade e medida a distribuição de freqüência. Os resultados podem ser vistos na Figura 3.7

As avaliações mostraram que a função para calcular o índice de E-amizade mostrou

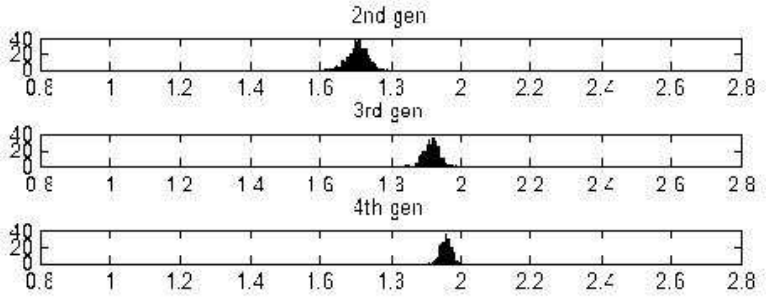

Figura 3.7 Distribuição de freqüência dos índices da segunda até a quarta geração para variantes do vírus `W32.Evol`

resultados animadores, podendo assim ser utilizada para detecção de variantes de vírus. Também é importante observar que o índice de E-amizade aumentava à medida em que cresciam as gerações, indicando assim que a função não é capaz de detectar com grande probabilidade pequenas alterações. Um ataque feito por um construtor de vírus a tal metodologia poderia envolver, desta maneira, a utilização um motor de transformação metamórfica que realizasse apenas pequenas alterações entre cada variante do vírus. Porém, ao mesmo tempo, o pequeno número de transformações não apresentaria grandes problemas para o processo regular de identificação do vírus, através dos métodos de assinaturas dinâmicas, utilizadas por sistemas de antivírus.

3.3.3 Padrões em funções do sistema

Uma das maneiras utilizadas para identificar similaridades entre dois programas é através do conjunto de chamadas realizadas ao sistema operacional ou bibliotecas de funções importadas pelo programa. Desta maneira [ZR07] identificou padrões em executáveis, baseado em chamadas a funções, para poder medir o grau de semelhança entre eles com relação à sua funcionalidade. Os autores criaram uma função capaz de medir este grau de semelhança ou similaridade entre dois executáveis. Como parâmetros a função aceita os dois programas que devem ser comparados. Como saída, a função calcula um número, entre zero e um, de maneira que quanto mais próximo do máximo fosse o resultado, maior seria a semelhança entre o funcionamento de ambos os programas.

Diferente dos estudos de [CJS+05], onde de forma manual são gerados padrões para comportamentos maliciosos (como laços de criptografia), o trabalho proposto em [ZR07] gera de forma automática padrões que caracterizam a semântica de um programa, utilizando estes padrões para comparar a funcionalidade entre programas. O método proposto também serve como uma forma de comparar diversas versões de um mesmo programa e verificar se sua funcionalidade apresentou altos ou baixos graus de mudança. Porém, a principal funcionalidade da metodologia seria a de comparar e reconhecer diversas instâncias de um mesmo vírus.

Para determinar a semântica de um executável os autores propõem um mecanismo

de extração do que eles denominaram "padrões" (*pattern*). Estes padrões usam como características as chamadas de funções do sistema operacional ou de funções em bibliotecas. A hipótese de tais características serem de grande relevância para comparar funcionalidades em programas foi defendida pelos autores pelo argumento que seria muito difícil para programas causarem um prejuízo a um sistema sem a utilização de funções do sistema operacional. Como exemplo pode-se tomar o caso do vírus `Sapphire`, que executa o seguinte conjunto de chamadas ao sistema: `LoadLibrary`, `GetProcAddress`, `GetTickCount`, `socket`, `sendto`. Seria muito difícil fazer com que um vírus possuísse a mesma funcionalidade do `Sapphire` sem que este conjunto de instruções fosse utilizado, ou sem mesmo que tivesse um código consideravelmente maior.

O primeiro passo da extração dos padrões de executáveis é a obtenção do seu código de máquina através do uso de uma ferramenta de engenharia reversa. Uma vez obtido o código fonte obteve-se o controle de fluxo através de uma análise estática do código. O controle de fluxo é extraído para que posteriormente seja possível detectar as chamadas de funções no executável. O processo de extração tem como resultado um conjunto de blocos de instruções e a transferência do controle entre tais blocos. O próximo passo é identificar instruções que afetam os valores em memórias ou registradores utilizados por funções do sistema quando elas são chamadas pelo programa. O trabalho de [ZR07] limita-se apenas à utilização do parâmetro "endereço de destino" (*target address*) das instruções de mudança de fluxo tipo `CALL`. Para cada um dos blocos de instruções do programa são determinadas as funções ou chamadas ao sistema que são afetadas devido aos parâmetros deste bloco de instruções.

Os autores determinaram o "caminho maximizado" (*maximal instruction trace*) como sendo a seqüência de instruções que afeta os parâmetros e valores utilizados por funções do sistema. Cada uma destas instruções do caminho maximizado é substituída por uma representação intermediária de mesmo valor semântico. Desta forma instruções como `LOOP` ou `JCC` são associadas ao mesmo tipo de operação, pois ambas realizam a mesma operação de transferência de controle diferindo apenas na maneira como tratam o parâmetro que as acompanha. Da mesma forma a transformação para a forma intermediária faz com que instruções como: `SUB ECX, ECX` e `XOR ECX, ECX` sejam identificadas como equivalentes a `MOV ECX, 0`, que significa a transformação do valor armazenado no registrador `ECX` para zero.

O último passo para identificação dos padrões é a execução de cada instrução do caminho maximizado de cada executável de maneira limitada. Esta execução consiste apenas na propagação de valores constantes, ou seja, supondo que um valor C, quando atribuído a um registrador ou alocação de memória, afete as funções ou chamadas do bloco de instruções ao sistema operacional, então esta instrução é armazenada no que é nomeado pelos autores de "subpadrão". O subpadrão de um bloco é o conjunto de instruções que afeta os parâmetros utilizados em funções do sistema. O conjunto de subpadrões de um executável é denominado o padrão deste executável. O intuito desta análise é identificar os recursos do sistema operacional utilizados pelos vírus e mesmo que ele o acesse de maneira diferente (através de uma transformação de código) duas instâncias de um mesmo vírus irão compartilhar, em certo grau, os mesmos padrões. A Figura 3.8 mostra os subpadrões extraídos de duas versões do vírus Sapphire. Pode-se

perceber a existência de vários subpadrões comuns entre as duas versões, embora em ordens distintas.

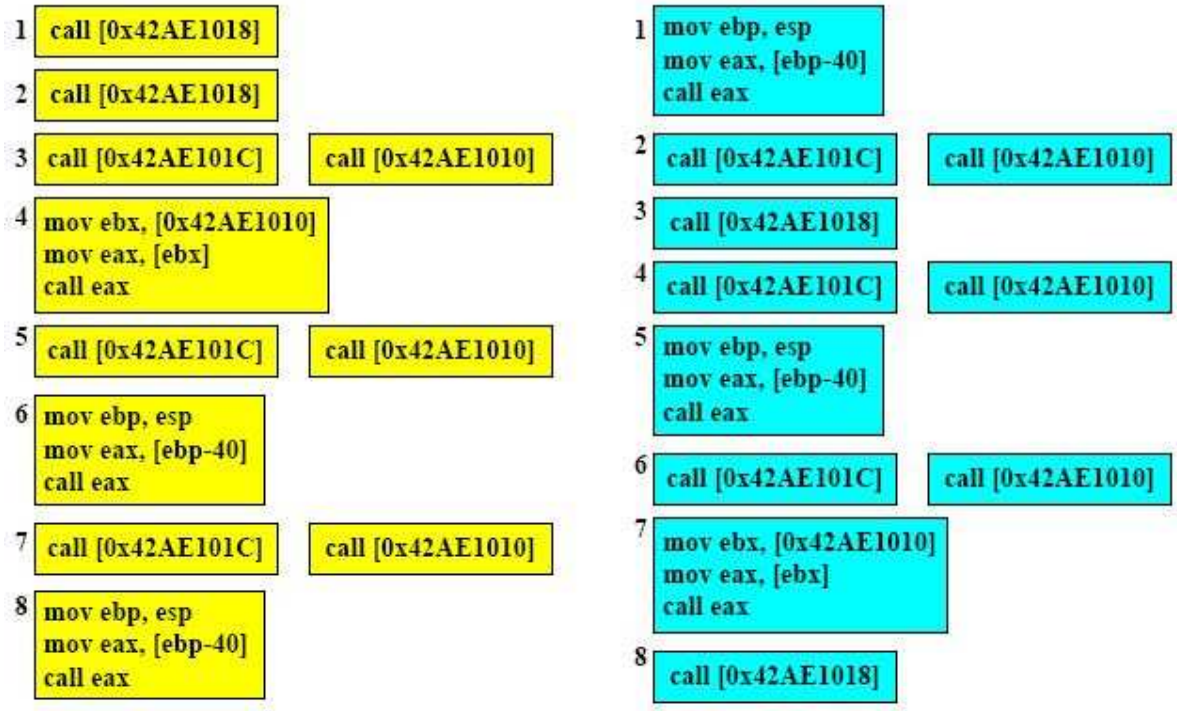

Figura 3.8 Padrões de duas versões do vírus Sapphire

Após a extração dos padrões de ambos os executáveis que se pretende comparar, os mesmos servem como parâmetros para uma função de avaliação. Esta função funciona comparando os padrões e associando entre os executáveis um número pertencente ao intervalo [0,1]. Este número representa quão próximas as funcionalidades dos executáveis se encontram. Desejou-se que executáveis com funcionalidades semelhantes obtivessem resultados mais próximos de 1 e caso contrário mais próximos de 0. A função proposta por [ZR07] consiste em comparar cada um dos subpadrões dos executáveis entre si e computar o seu grau de semelhança ou similaridade. Para realizar tal operação os autores propõem que operações similares obtenham maiores pesos que operações que não agem da mesma forma. Desta maneira, operações como adição e subtração possuiriam mais pontos que operações de salto. Para a resolução de problemas com pesos o algoritmo húngaro [Wes01] foi utilizado.

Para avaliar a metodologia de análise de executáveis apresentada foi realizada uma implementação do método proposto para trabalhar em arquivos do sistema operacional Linux ou Microsoft Windows, com foco em arquivos compilados para a arquitetura Intel x86. Para o teste experimental da solução foram criados três conjuntos. O primeiro foi um conjunto de doze programas regulares que passaram por uma ferramenta de randomização de código proposta por em [KJB+06]. O segundo conjunto consistia de variantes de vírus conhecidos para o sistema operacional Microsoft Windows, obtidas

no sítio VXHeavens [VXH]. O terceiro conjunto abrangia diversos programas regulares escolhidos aleatoriamente, sendo todos compilados para plataforma Linux.

No primeiro grupo mediu-se o índice de similaridade após a utilização da ferramenta de randomização. Depois dos doze programas passarem pela ferramenta e terem seus índices mensurados encontrou-se um índice, em relação ao original, superior a 95% para dez deles, enquanto os outros dois obtiveram índices maiores que 75%. Estes resultados mostram que, para a maioria dos executáveis, a metodologia proposta foi capaz de reconhecer que se tratavam do mesmo programa, sendo um limite de 95% considerado.

Para o segundo conjunto avaliou-se quão bem a ferramenta proposta poderia identificar diversas variantes do mesmo vírus. Consideraram-se 200 pares de variantes, entre eles executáveis de vírus, *worms* e cavalos-de-Tróia. As diversas variantes podiam ter sido geradas ou por mudanças manuais ou automaticamente através de motores metamórficos dos próprios vírus. O resultado encontrado para o cálculo dos índices de similaridade, para cada par de variante do segundo conjunto, mostrou que mais de 90% dos pares forneceram um índice maior que 70%, mostrando assim que também variantes de um mesmo vírus podem ser identificadas com uma grande probabilidade. Ainda no segundo conjunto um outro experimento foi realizado com o objetivo de avaliar se vírus que não possuiam relação obtinham baixos índices de similaridade, desta maneira evitando erros de classificação. Os 200 pares foram comparados com outros que não sua variante correspondente. O resultado mostrou que 90% dos pares comparados tiveram índices menores que 1%, e aproximadamente 1% teve índices de 70% ou mais, que também podem ser explicados pelo fato que alguns vírus utilizam o mesmo código base em seu desenvolvimento.

Construiu-se um terceiro conjunto com a intenção de comparar quão bem o método proposto é capaz de reconhecer variações entre diferentes versões de um mesmo programa. Diferentemente dos dois primeiros conjuntos este grupo de programas não foi intencionalmente modificado para confundir o antivírus, ou seja, suas funcionalidades nas diferentes versões podiam apresentar diferenças. Versões entre 2.10 e 2.17 de ferramentas do projeto "GNU binutils" [GNU] foram utilizadas para realizar este experimento. Para a grande maioria dos casos o índice de similaridade calculado foi maior que 70%, mostrando assim que nos outros casos as mudanças de versões indicavam grandes mudanças em suas funcionalidades.

A nova metodologia proposta mostrou bons resultados para avaliação de variantes do mesmo vírus, além de ser útil também para comparação entre versões do mesmo programa com o intuito de analisar o grau de mudança em suas funcionalidades. Apesar disso existem limitações apontadas pelos autores no que diz respeito aos vírus que utilizam criptografia em seu código para confundir os antivírus.

3.4 NORMALIZAÇÃO DE CÓDIGO

Considera-se normalização do código o processo de eliminar ofuscamentos no código fonte de um programa, eliminando assim operações de transformações metamórficas, utilizadas pelos vírus com o intuito de enganar os programas de antivírus. Embora não envolvam diretamente detecção de vírus, os trabalhos de normalização de código analisam o código de máquina do vírus de forma estática e modificam quaisquer partes que classificarem

como uma tentativa de mudança de código do próprio vírus em seu código. De tal forma, os trabalhos que tratam de normalização sugerem que um executável antes de ser classificado pelo antivírus deve passar por um processo de "limpeza" do código. Esta medida fará com que a taxa de acerto dos antivírus aumente em relação às diversas variantes do mesmo vírus.

Um exemplo de processo de normalização pode ser visto na Figura 3.9, onde a primeira coluna representa o código de máquina de um vírus antes da transformação, a segunda coluna representa o mesmo código após ter passado por um ofuscamento de código e finalmente a versão normalizada do código pode ser vista na terceira coluna. É fácil perceber que o código escrito na terceira coluna se assemelha bem mais ao da primeira coluna que o da segunda coluna. Embora as versões normalizadas não consigam voltar todo o código transformado à sua forma original é possível criar versões próximas o bastante para que elas sejam detectadas pelos antivírus.

```
   lea  eax, [ebp+Data]
   push esi
   push eax
   call ds:GetWindowsDirectoryA
   lea  eax, [ebp+Data]
   push eax
   call _strlen
   cmp  [ebp+eax+var_129], 5Ch
   pop  ecx
   jz   short loc_40
   lea  eax, [ebp+Data]
   push offset asc_408D80
   push eax
   call _strcat
   pop  ecx
   pop  ecx
loc_40:
   lea  eax, [ebp+Data]
   push offset aServices_exe
   push eax
   call _strcat
```

```
     jmp  lab
lan: add  [esp], 1
     jmp  lay
lop: cld
     jmp  lah
lac: scasb
     jmp  lam
laz: mov  al, 99
     jmp  lop
lav: loop lop
     jmp  law
las: dec  edi
     jmp  lav
lab: mov  edi, offset der
     jmp  laz
lam: push edi
     jmp  lan
lay: pop  edi
     jmp  las
lah: xor  byte ptr [edi],1
     jmp  lac
law: jmp  short der
     ...
der: db 8c 84 d9 ff fe fe
     db ...
     db 01 98
```

```
   lea  eax, [ebp+data1]
   push esi
   push eax
   call ds:GetWindowsDirectoryA
   lea  eax, [ebp+data1]
   push eax
   call _strlen
   cmp  [ebp+eax+data2], 5Ch
   pop  ecx
   jz   short label1
   lea  eax, [ebp+data1]
   push offset data3
   push eax
   call _strcat
   pop  ecx
   pop  ecx
label1:
   lea  eax, [ebp+data1]
   push offset data4
   push eax
   call _strcat
```

Figura 3.9 Exemplo do processo de normalização de código

O trabalho de [CJK+07] tratou de normalização de código e construiu um protótipo que é capaz de avaliar e desfazer diversos tipos de ofuscamento de código. Em seus resultados experimentais [CJK+07] ainda mostrou a eficiência de sua ferramenta de acordo com diversos tipos diferentes de transformações metamórficas. Os autores começaram classificando as transformações metamórficas como:

Reordenamento do código – o reordenamento de código acontece quando as ordens de quaisquer duas instruções, ou conjuntos de instruções, no código são alteradas sem que isso prejudique seu funcionamento. O reordenamento de código pode ser obtido através da troca de instruções que não possuem relação semântica entre si, como o caso das instruções `MOV AL, 99` e `MOV EDI, 100`, responsáveis por colocar

o valor 99 e 100 nos registradores `AL` e `EDI`, respectivamente. Embora realizem ações diferentes, não importa a ordem em que forem executadas, desde que sejam executadas uma depois da outra, pois o resultado final será o mesmo. Outra forma de obter o reordenamento de código é através do uso de instruções de salto. As seqüências de códigos reordenadas, em maioria, possuem instruções de salto não condicionais desnecessárias e os autores de [CJK+07] se concentram apenas neste tipo de transformação metamórfica em seu trabalho;

Inserção de lixo – a inserção de lixo é feita através de instrução de valor semântico igual ao de operações do tipo `NOP` (*no-operation*) que não executam alteração de valores aos registradores ou endereços de memória. Operações deste tipo podem ser removidas do código sem que haja qualquer alteração em sua funcionalidade;

Compactação - a compactação é a operação de executar um código gerado na memória montado em tempo de execução. Estas porções de código ficam armazenadas de forma compactadas dentro do código de máquina do executável. Antes de serem executadas o programa descompacta o código e o transfere para a memória com a intenção de executá-lo. Desta maneira quando um programa de detecção de vírus tenta encontrar o código fonte do vírus não pode encontrá-lo devido à grande dificuldade de determinar de que forma ele está compactado.

Para que a ferramenta de normalização funcione cada um dos tipos de ofuscamento citados pelos autores é identificado e tratado de forma diferente. O código fonte de máquina do arquivo a ser analisado era percorrido, através de uma análise estática, e outro executável, na forma normalizada, era gerado para ser finalmente avaliado pelo programa de antivírus. Para resolver o problema de vírus que reordenam seu código a ferramenta proposta utiliza a estrutura do fluxo de execução para juntar os códigos que aparentemente sofreram um reordenamento de código através do uso de instruções de saltos não condicionais. Para detectar tais conjuntos de instruções a ferramenta verifica se uma porção de código é executada apenas uma vez durante o fluxo do programa. Se este for o caso, o fragmento de código em questão pode ser anexado no lugar da instrução de salto responsável pela sua execução.

Também através da análise estática do código a ferramenta eliminava algumas das instruções de lixo que encontrava. A intenção era verificar se, para cada uma das instruções do programa, seu resultado era responsável pela alteração de valor de qualquer registrador ou parte da memória. Desta forma, instruções inúteis, como adições de valores nulos, eram eliminadas do código.

Para eliminar as transformações de compactação a ferramenta, a partir do ponto de entrada do programa, começa executando o código em um ambiente controlado para assim identificar as instruções que transferem o controle para partes do código que tenham sido geradas na memória. Com esta informação uma nova versão do executável é gerada contendo o código que era executado na memória, ao invés da instrução de salto responsável por sua execução. Este processo pode ser considerado também como um processamento dinâmico, o que tornaria esta solução uma forma híbrida de resolver o problema de detecção de vírus, visto que ambas as abordagens de análise são empregadas.

Para avaliação dos resultados da metodologia de normalização de código proposta, foi gerado um grande número de variantes do vírus `Beagle.Y` e posteriormente foram medidas as taxas de detecção destas variantes, antes e depois do processo de normalização, para diversos antivírus. Cada uma das variantes foi gerada aplicando transformações de reordenamento de código em fragmentos em seu código de máquina escolhidos aleatoriamente. Os resultados mostraram que, para o programa McAfee VirusScan, os percentuais de detecção aumentaram de 36,83% para 100% quando uma taxa de ofuscamento de 10% foi aplicada. A taxa de ofuscamento diz em quanto do código foram aplicadas transformações de reordenação. Outras ferramentas como Sophos Anti-Virus passaram a detectar 100% das variantes depois de normalizadas, antes tendo índice de detecção de 5% para ofuscamentos em 50% do código.

Testes adicionais foram realizados utilizando outras técnicas de ofuscamento. Para testes da normalização contra transformação de adição de operações semanticamente nulas, ou seja, sem valor semântico foram obtidos resultados com índices de detecção atingindo 100% depois da normalização em todos os antivírus testados, enquanto que antes da normalização ferramentas como Sophos Anti-Virus e McAfee VirusScan tiveram índices inferiores a 4,5% e 17,2%, respectivamente.

Uma última avaliação envolveu testar o processo de normalização quando atacado por ferramentas de compactação. Para este último teste foram selecionadas sete ferramentas de compactação, tais como: UPX, Packman, UPack e PE Pack. Cada ferramenta foi aplicada a diversas variantes dos vírus `Netsky` e `Beagle`. Os programas de antivírus tinham, da mesma forma, suas taxas de detecção registradas antes e depois das variantes transformadas passarem pela etapa de normalização em seu código. Ferramentas como McAfee VirusScan tiveram suas taxas de detecção aumentadas de 57,8% para 94,4% e outros programas como ClamAV e Sophos Anti-Virus tiveram seus índices aumentados de 28,9% para 82,2% e de 58,9% para 83,3%, respectivamente.

Embora esta metodologia de normalização tenha apresentado bons resultados, ainda existem limitações em sua aplicação. Uma das limitações apontadas no trabalho de [CJK+07] é o fato da aplicação da normalização de ofuscamento por reordenamento de código poder ser enganada pela adição de códigos não atingíveis, ou seja, fragmentos de código que não possuem chamadas para sua execução. Outra limitação que pode ser observada é a falta de uma abordagem que trabalhe com códigos que utilizam criptografia para não serem detectados.

Dentre outros trabalhos que utilizam técnicas de normalização de código para aumentar os índices de detecção se destaca a metodologia proposta por [BMM07]. Neste trabalho os autores estudam técnicas de otimização de código utilizadas pelos compiladores com a intenção de gerar executáveis com códigos mais rápidos e menores. Quando tais técnicas foram aplicadas em variantes de vírus que sofreram transformações metamórficas, as taxas de detecção dos arquivos otimizados foram maiores do que anteriormente. Baseando-se nesta observação os autores propuseram a construção de uma ferramenta que pudesse gerar versões otimizadas (e normalizadas) de executáveis antes que eles fossem avaliados pela ferramenta de detecção de vírus. Após a normalização também foi proposta uma forma de comparação entre executáveis que permite que variantes de um mesmo vírus sejam detectadas.

Dentre as técnicas de otimização utilizadas em [BMM07] está utilizada a do uso de instruções de meta-representação. Todas as instruções do código de máquina eram classificadas como:

Saltos : que são instruções capazes de desviar o fluxo de controle para outras partes do código;

Chamadas de funções e retornos : são responsáveis por criar sub-rotinas que podem ser executadas com apenas uma chamada;

Atribuições : são quaisquer outros tipos de operações que alteram os valores dos registradores ou memória.

A utilização de instruções de meta-representação envolve substituir as operações regulares do código por uma representação de alto nível de instruções de máquina expressando o mesmo valor semântico. Um exemplo de aplicação desta técnica pode ser visto na Figura 3.10, onde em uma coluna é mostrada a representação de máquina e em outra, a meta-representação.

Instrução de máquina	Meta-representação
pop eax	r10 = [r11]
	r11 = r11 + 4
lea edi,[ebp]	r06 = r12
dec ebx	tmp = r08
	r08 = r08 - 1
	NF = r08@[31:31]
	ZF = [r08 = 0?1:0]
	CF = (~(tmp@[31:31]) ...
	...

Figura 3.10 Exemplo de meta-representação

A propagação é outra técnica utilizada na normalização de código. Propagação é utilizada para propagar valores atribuídos ou calculados por instruções intermediárias. Quando uma instrução define uma variável (seja ela um registrador ou endereço de memória) e esta variável é utilizada por instruções subseqüentes sem que seu valor tenha sido alterado, então todas suas ocorrências podem ser substituídas pelo valor computado na instrução anterior. A maior vantagem da propagação é permitir a geração de instruções de alto nível com mais de dois operadores, reduzindo assim o seu código. Além disso, a propagação também ajuda na eliminação de variáveis temporárias. Um exemplo da utilização da técnica de propagação é apresentado na Figura 3.11.

A eliminação de código morto é citada pelos autores como outra técnica para otimização de código. Instruções de código morto são aquelas que nunca são utilizadas. Por exemplo, se uma variável está no lado esquerdo de duas atribuições mas não possui seu valor utilizado, ou seja, nunca é utilizada no lado direito, entre as duas atribuições então a primeira atribuição pode ser considerada um código morto, visto que sua eliminação

Antes da propagação	Depois da propagação
`r10 = [r11]`	`r10 = [r11]`
`r10 = r10 | r12`	`r10 = [r11] | r12`
`[r11] = [r11] & r12`	
`[r11] = ~[r11]`	
`[r11] = [r11] & r10`	`[r11] = (~([r11] & r12)) & ([r11] | r12)`

Figura 3.11 Exemplo de propagação

não irá afetar o funcionamento e a semântica de todo o código. Esta técnica pode ser comparada com a eliminação de lixo citada pelos autores de [CJK+07].

Ainda sobre as técnicas de otimização de código, que visam à normalização do mesmo, é importante citar a simplificação algébrica. Como a maioria das instruções encontradas no código de máquina possui operadores aritméticos ou lógicos eles podem ser simplificados de acordo com as regras das operações algébricas. Através da utilização de tais técnicas é possível transformar instruções como $(t_1 + t_2)\, t_3$ em $t_1 t_3 + t_2 t_3$, ou de $t_1 + 0$ em apenas t_1. Algumas das operações de simplificação algébrica podem ser vistas na Figura 3.12, onde a letra c representa uma constante e os registradores estão representados pela letra t.

Expressão original	Espressão simplificada
$c_1 + c_2$	a soma das constantes
$t_1 - c_1$	$-c_1 + t_1$
$t_1 + c_1$	$c_1 + t_1$
$0 + t_1$	t_1
$t_1 + (t_2 + t_3)$	$(t_1 + t_2) + t_3$
$(t_1 + t_2) * c_1$	$(c_1 * t_1) + (c_1 * t_2)$

Figura 3.12 Exemplo de simplificações algébricas

A última técnica de otimização citada pelo autor é a otimização do grafo do fluxo de execução. O grafo do fluxo de execução consiste em um grafo onde os vértices representam fragmentos do código a serem executados sem mudança do fluxo de execução. As arestas são representações do fluxo de controle com a mudança de controle entre tais fragmentos de código. O grafo do fluxo de execução de um vírus pode conter falsas instruções de saltos. Através da propagação é possível determinar quais instruções de saltos condicionais possuem sempre o mesmo valor, para assim determinar se o salto será sempre realizado. Desta maneira é possível também determinar se qualquer dos vértices poderia ser anexado a outro sem que a semântica do programa fosse alterada, desta maneira otimizando o grafo do fluxo de execução.

Infelizmente não se pode esperar que todas as variantes de um mesmo vírus tenham adquirido uma mesma estrutura, mesmo depois de passarem pela normalização. Existem

diferenças, como instruções de atribuição a variáveis temporárias, que não podem ser tratadas e usualmente permanecem sobre o código normalizado. Por esta razão não seria eficiente comparar bit a bit as formas normalizadas dos executáveis. Para lidar com tal problema os autores desenvolveram um método de medir similaridades entre diferentes fragmentos de código que permite capturar, de forma mais eficiente, possíveis variantes de um mesmo vírus.

A metodologia de detecção utilizada foi proposta no trabalho de [KDM+96] onde a estrutura de um fragmento de código é caracterizada por um vetor de medidas calculadas sobre o código fonte e uma medida de distância entre diferentes fragmentos é proposta. As medidas adotadas em [BMM07] para realização de tal experimento foram:

- m_1 – número de nós do grafo de fluxo de execução,
- m_2 – numero de arestas do grafo de fluxo de execução,
- m_3 – número de chamadas diretas,
- m_4 – número de chamadas indiretas,
- m_5 – número de saltos diretos,
- m_6 – número de saltos indiretos,
- m_7 – número de saltos condicionais.

Tais medidas foram escolhidas com o intuito de capturar a estrutura de um código executável. Desta forma, a "assinatura" de um fragmento de código era determinada pelo vetor $(m_1, m_2, m_3, m_4, m_5, m_6, m_7)$. Depois foi definido que dois fragmentos de código são equivalentes se a sua distância euclidiana é menor que um certo limite. A distância euclidiana entre dois fragmentos de código a e b é definida como

$$\sqrt{\sum_{i=1}^{7} (m_{i,a} - m_{i,b})^2},$$

onde $m_{i,a}$ e $m_{i,b}$ são a i-ésima medida calculada respectivamente nos fragmentos a e b. A distância ideal entre dois fragmentos equivalentes deveria ser nula, porém um limite é estabelecido em função de pequenas imperfeições que possam ocorrer no processo de normalização.

Foi desenvolvido, pelos autores, um protótipo com intenção de realizar um teste da eficiência do processo de normalização proposto. Este protótipo começa obtendo o código fonte dos executáveis através do uso de uma ferramenta de engenharia reversa. Depois um processo de análise do código fonte é feito e todas as técnicas de otimização de código são aplicadas com o intuito de gerar a forma normalizada do executável. Desta maneira, o resultado final deste processo é a geração de um grafo de fluxo da execução em sua forma otimizada. Um exemplo de fragmento de código antes e depois deste processo pode ser visto na Figuras 3.13 e 3.14, respectivamente.

```
8048354: mov    $0x1000400c,%eax
8048359: mov    %eax,0x10004014
804835e: add    %eax,%ebx
8048360: test   %ebx,%ebx
8048362: jne    804836a
8048364: push   %ebx
8048365: mov    $0x0,%ebx
8048365: inc    %eax
804836c: jmp    *%ebx
```

Figura 3.13 Exemplo de um fragmento de código antes do processo de normalização

Um experimento com o protótipo desenvolvido envolveu a utilização de um motor de transformações metamórficas. O motor é chamado de METAPHOR [Met]. O experimento consistiu primeiramente na coleção de 114 amostras de executáveis regulares infectados. Os programas infectados eram comparados com suas cópias não infectadas, sendo assim possível identificar o corpo do vírus. O vírus servia então como entrada para a ferramenta que iria realizar a normalização e a comparação. Primeiramente o vetor de medidas era calculado antes da normalização, depois era calculada a sua distância euclidiana em relação ao arquivo original. Após o processo era feito um novo cálculo do vetor e sua distância era computada novamente em relação ao executável original. O resultado do experimento, como pode ser visto no gráfico da Figura 3.15, mostrou que poucas variantes tiveram suas medidas iguais ou próximas de zero para o executável antes da normalização, enquanto que depois da normalização as distâncias calculadas sofreram uma significante redução, exceto os casos em que a medida já era muito próxima ou igual a zero, que possivelmente foi resultados de transformações metamórficas simples. O valor sugerido para o limite da distância que determina se um código é considerado equivalente a outro foi medido pelo experimento como 4,9.

Para comparar a distância entre dois executáveis antes e depois da normalização realizou-se ainda um segundo experimento utilizando 1000 executáveis escolhidos aleatoriamente sem a presença de vírus em seu corpo. A experiência feita consistia em calcular, para todos os programas, as distâncias entre a versão original e a versão normalizada para poder avaliar se a redução da distância é realmente significante na maioria dos casos. Os resultados, como podem ser vistos na Figura 3.16, mostram que para maioria dos casos (827 casos) as distâncias entre os executáveis são maiores que 4,9.

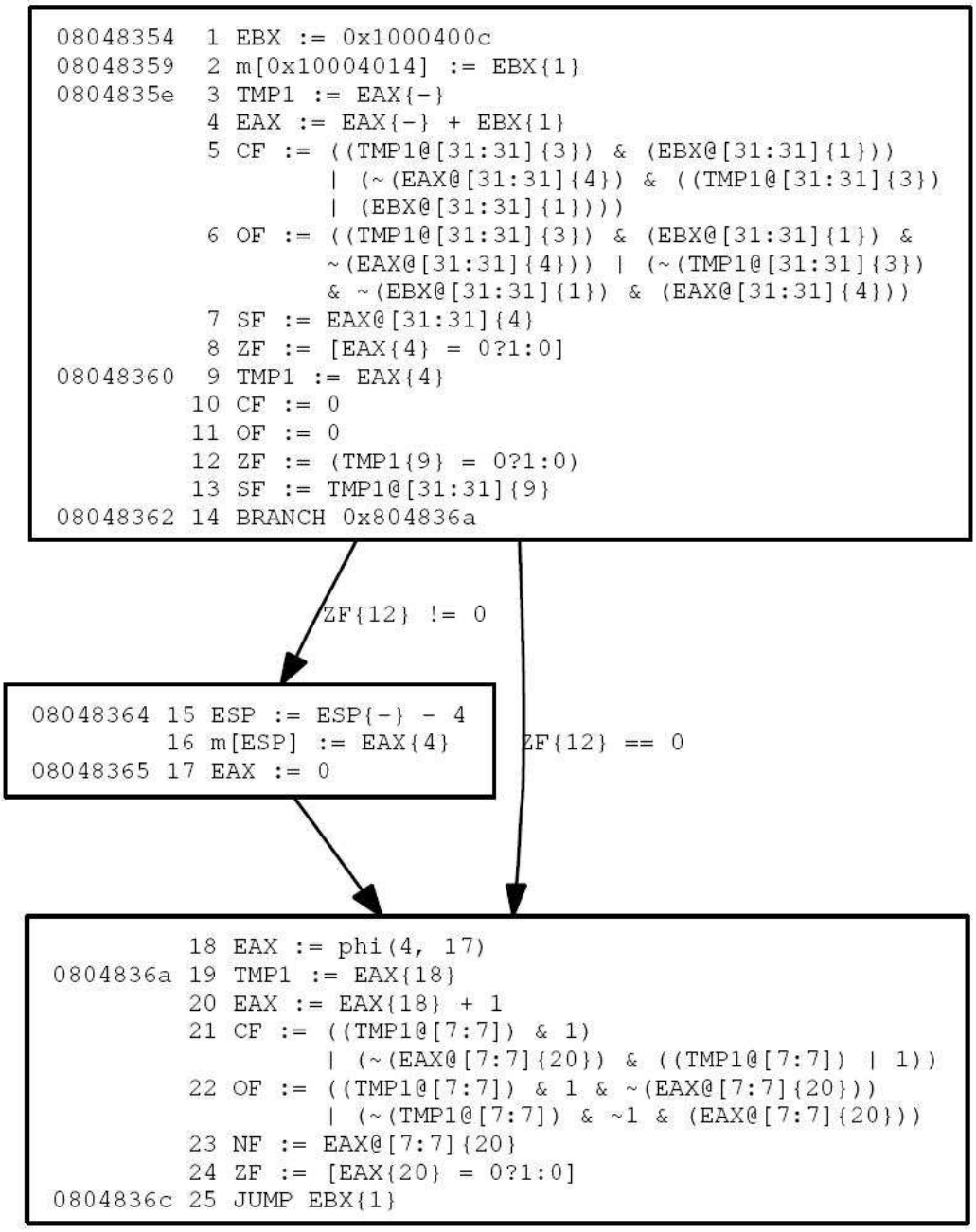

Figura 3.14 Exemplo do resultado do processo de normalização do fragmento de código da Figura 3.13

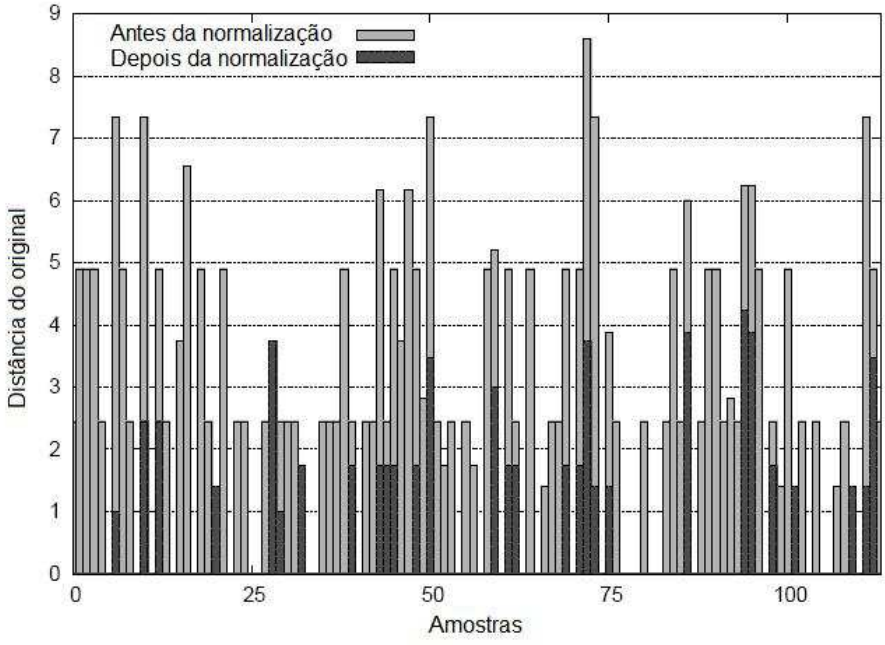

Figura 3.15 Comparação entre as distâncias ao original antes de depois do processo de normalização para arquivos com vírus

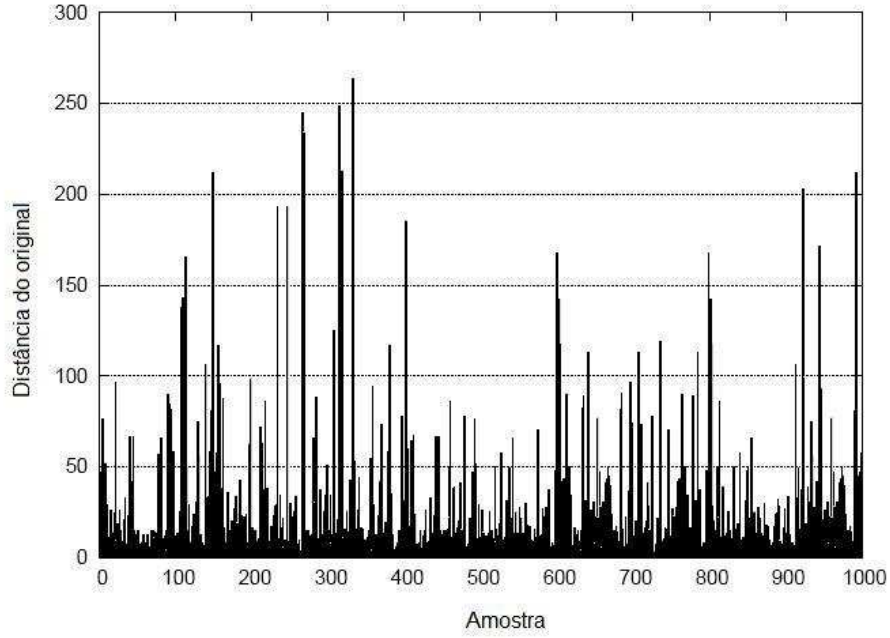

Figura 3.16 Comparação entre as distâncias ao original antes de depois do processo de normalização para executáveis sem infecção de vírus

CAPÍTULO 4

ASAT

Neste capítulo será apresentado como foi realizado o processo de construção da solução proposta e preparação dos dados para que fossem realizados os experimentos de treinamento e avaliação dos resultados.

4.1 EXTRAÇÃO DAS CARACTERÍSTICAS

A solução proposta baseia-se em características extraídas do código de máquina dos executáveis para classificá-lo como um vírus ou não. A idéia principal é obter as características de diversos vírus e depois ao tentar avaliar um executável verificando se existe semelhança entre o executável e os vírus. Em seguida será descrito o processo usado para extração das características.

4.1.1 Repositórios de vírus e programas

Antes da extração das características um repositório com diversos vírus e programas foi construído. Neste repositório foram coletados vírus obtidos através da Internet pelo sítio VXHeavens [VXH]. Na solução proposta foram utilizados apenas vírus para Microsoft Windows, embora o fato de trabalhar com o código na linguagem Assembly extraído dos executáveis facilite posteriormente portar seu código para que trabalhe com outros tipos de vírus e em outros sistemas operacionais. Também não houve utilização de arquivos infectados por vírus, já que a intenção do ASAT é identificar os vírus antes do processo de infecção. Outros vírus foram obtidos através do uso de kits para criação de vírus, como NGVCK e VCL32, obtidos de mesma forma no endereço eletrônico VXHeavens.

Os programas foram obtidos através da Internet, porém também foram considerados alguns executáveis encontrados no próprio sistema operacional (Microsoft Windows XP) e em diversos outros programas, como por exemplo, o Adobe Reader e o Microsoft Office. Também foram obtidos, através dos relatórios de [AVC], diversos programas que em algum momento foram classificados como vírus erroneamente por programas de antivírus comerciais. Desta maneira foi possível incluir programas que possuíam comportamentos de vírus, como, por exemplo, alteração de registro do sistema ou pacotes de atualização que em sua execução costumam alterar diversos executáveis.

Existem técnicas utilizadas por vírus (e em alguns casos por programas regulares) para dificultar o processo de engenharia reversa. Muitas destas técnicas foram explicitadas em [LD03]. Existem também trabalhos que se preocupam em combater tais técnicas, como é o caso de [KRVV04]. O presente trabalho não aborda qualquer das técnicas para dificultar ou ajudar o processo de engenharia reversa. Por tais motivos nenhum dos vírus ou programas utilizados nos experimentos que tivesse sinais que não podia passar pelo processo de engenharia reversa foi considerado. Ou seja, qualquer executável que

apresentasse erro ao tentar se obter seu código fonte ou que fosse necessário um tempo muito elevado para obter seu código era descartado.

Todos executáveis, sendo eles vírus ou não, foram distribuídos em dois repositórios. Um primeiro repositório contendo diversos programas e vírus obtidos na Internet. Neste repositório não foram incluídos vírus gerados por kits de construção de vírus. O objetivo deste primeiro repositório é fazer com que os classificadores sejam treinados com informações sobre vírus e programas, por isso foi denominado "repositório de treinamento". No final, no repositório de treinamento havia 1405 vírus e 525 programas regulares.

O segundo repositório foi denominado "repositório de teste" e seu objetivo é avaliar o desempenho da solução ASAT. Este segundo repositório também foi utilizado para comparar o desempenho da solução ASAT com outras soluções comerciais. Para construção do repositório de teste foram considerados vírus conhecidos, além de vírus gerados por kits de construção de vírus, como NGVCK e VCL32. Os programas regulares que entraram neste repositório foram também obtidos na Internet, porém dentre eles estavam os programas que em algum momento obtiveram uma classificação de vírus em algum antivírus comercial. Nenhum dos vírus ou programas utilizados no repositório de treinamento foi utilizado no repositório de teste. O segundo repositório conteve 336 vírus e 99 programas regulares.

O primeiro passo para realizar a extração de características foi aplicar o processo de engenharia reversa em cada um dos arquivos pertencentes aos repositórios de treinamento e teste. A ferramenta de engenharia reversa utilizada foi o IDA Pro [IDA] responsável por gerar todos códigos fontes dos executáveis. O segundo passo foi inserir cada instrução e parâmetro do código fonte obtido em um banco de dados relacional. As instruções e parâmetros eram inseridos de forma que possuíssem uma chave estrangeira para o executável de origem. Este processo gerou dois bancos de dados que foram denominados de "banco de treinamento" e "banco de teste" dependendo se a informação de tal banco de dados vinha do repositório de treinamento ou teste, respectivamente. Os bancos de treinamento e teste foram formados a partir de duas tabelas, uma onde se registrou o nome do executável e se ele era ou não um vírus e outra onde eram registrados as instruções e parâmetros.

4.1.2 Geração de relatório estatístico

Após a inserção de todos dados nos bancos de treinamento e teste era necessário gerar uma entrada de forma que os classificadores pudessem utilizar para seu treinamento. Desta maneira foram determinados instruções e parâmetros que ocorreram mais freqüentemente e encontrados na maior parte dos executáveis. A intenção é determinar se a proporção de tais instruções e parâmetros em relação ao número total de instruções em todo código do executável é suficiente para determinar se ele é ou não um vírus. Tais proporções foram escolhidas como características que serviriam de entrada para os classificadores. O motivo de escolher as instruções encontradas em mais executáveis foi o de não encontrar um número elevado de falta de informação, embora esta medida pudesse deixar de fora instruções importantes que classificassem de forma eficiente.

No final foram escolhidas 22 instruções e 20 parâmetros para o cálculo das carac-

terísticas. Uma característica é determinada pelo número de vezes que a instrução ou parâmetro aparecia no código dividido pelo número total de instruções do executável correspondente de tal forma que cada característica indica a proporção, em relação a todo o código, com que aquela instrução ou parâmetro aparece dentro do código fonte do vírus ou programa analisado. Nem todas instruções ou parâmetros são significantes, ou nem mesmo possuem um grau de significância igual, para classificar os executáveis. Cada classificador possui uma forma de tratar as características, dando a elas um peso maior ou menor, ou até mesmo ignorando-a, na classificação.

A tabela final que possuía as características de todos executáveis pertencentes ao banco de treinamento foi chamada de "tabela de treinamento". Da mesma forma para o banco de teste foi gerada uma tabela chamada de "tabela de teste". Embora, em busca de melhores resultados, outras características poderiam ser utilizadas em conjunto com as apresentadas a intenção do trabalho era mostrar que tais proporções formam um bom conjunto de características que podem vir a ajudar na classificação de vírus.

4.2 TREINAMENTO DOS CLASSIFICADORES

Os dados da tabela de treinamento foram utilizados em cada classificador individual para que seus resultados servissem de base para decidir se um determinado executável se comporta ou não como um vírus. As técnicas de classificação utilizadas foram: classificador ingênuo de Bayes, rede neural do tipo Perceptron e regressão logística. Todos os classificadores tiveram o mesmo treinamento, ou seja, foram alimentados com os mesmos dados de treinamento. Para implementação da solução foi utilizada a ferramenta Weka [WF05][Wek]. Cada um dos classificadores possui parâmetros de entrada para configurar seu comportamento, porém nenhuma mudança de parâmetro foi explorada, visto que o objetivo principal do trabalho foi explorar ao máximo as informações contidas nas estatísticas criadas a partir do código fonte.

Cada um dos classificadores possuía pré-requisitos, em relação às suas entradas, para que fossem utilizados de maneira correta. A seguir será feita uma breve descrição do funcionamento e características de cada um dos classificadores escolhidos.

4.2.1 Classificador Ingênuo de Bayes

O classificador ingênuo de Bayes tem como base o teorema de Bayes, que permite calcular a probabilidade condicional

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)},$$

onde $P(A)$ e $P(B)$ são as probabilidades a priori dos eventos A e B, respectivamente e $P(A|B)$ e $P(B|A)$ são as probabilidades de A acontecer dado que o evento B ocorreu, e de B acontecer dado o evento A ocorreu, respectivamente.

Para utilização do classificador ingênuo de Bayes assume-se que todos os eventos utilizados são independentes. Em nosso trabalho foi assumida independência entre as proporções de cada uma das instruções ou parâmetros, embora se saiba que determinados parâmetros podem ser utilizados somente com certas instruções, como é o caso do

comando de rótulo (*label*) que deve conter apenas referências para endereços dentro do código fonte. Apesar deste fator poder indicar um certo grau de dependência entre as variáveis, a utilização do classificador ingênuo de Bayes foi encorajada pelos trabalhos apresentados em [Kun06][YSR+06], que mostram ser possível atingir resultados eficientes e eficazes mesmo utilizando dados que abrangem variáveis dependentes.

Como o classificador de Bayes utiliza um método de aprendizagem supervisionado, foram utilizados os dados da tabela de treinamento para seu treinamento. Embora somente um classificador tenha sido utilizado é fácil pensar em uma arquitetura com mais de um classificador trabalhando em paralelo. Seria possível pensar em uma arquitetura com diversos classificadores de Bayes, onde cada um deles seria treinado com informações de diversos tipos diferentes de vírus. O treinamento de um classificador de Bayes funciona com a utilização de variáveis categóricas e, como as variáveis de treinamento utilizadas eram numéricas, foi necessário realizar uma transformação das variáveis de proporção em dados nominais. Esta transformação foi realizada pela própria ferramenta Weka e consiste em criar diversos intervalos de forma que possam abranger todos os valores numéricos possíveis para cada variável e depois classificar os valores de tais variáveis dentro destes intervalos.

Adaptando o teorema de Bayes ao nosso trabalho podemos escrevê-lo da seguinte forma:

$$P(C|F_1, \ldots, F_n) = \frac{P(C)\ P(F_1, \ldots, F_n|C)}{P(F_1, \ldots, F_n)},$$

onde $P(C|F_1, \ldots, F_n)$ representa a probabilidade do executável ser da classe C (vírus ou programa) dado que as características $F_1, \ldots, F_n$ foram apresentadas. Como o denominador da fração é comum para as duas classes em consideração, é necessário apenas calcular os numeradores das frações de cada classe para assim determinar a classe em que, com tais características apresentadas, obteve maior probabilidade. Como é assumida uma independência entre as variáveis o numerador da fração também pode ser escrito da seguinte forma:

$$\begin{aligned} P(C)\ P(F_1, \ldots, F_n|C) &= P(C) \times P(F_1|C) \times P(F_2|C) \times P(F_3|C) \times \cdots \times P(F_N|C) \\ &= P(C) \prod_{i=1}^{n} P(F_i|C) \end{aligned}$$

As probabilidades $P(C)$ e $P(F_i|C)$ são facilmente estimadas através do uso dos dados de treinamento. Desta maneira, este produto é calculado para as duas classes C do problema em questão e a classificação final será decidida pelo produto de maior valor.

4.2.2 Multi-layer Perceptron

Redes neurais artificiais são utilizadas em diversas áreas e podem ser aplicadas para resolver vários problemas de reconhecimento de padrões, como por exemplo, reconhecimento de fala ou recuperação de informações em imagens. Geralmente redes neurais trabalham com dados numéricos, facilitando assim a aplicação dos dados da tabela de treinamento

diretamente, sem que fosse necessário discretizá-los, como no classificado de Bayes. Uma das desvantagens do classificador que usa redes neurais artificiais é o tempo de treinamento, consideravelmente maior que os dos demais classificadores, porém como este treinamento é realizado apenas uma vez, no nosso caso, não foram apresentados grandes problemas em sua aplicação.

Uma rede Perceptron consiste em uma rede neural direta (*feedforward*), ou seja, cujo grafo não possua ciclos, de múltiplas camadas. A técnica de aprendizagem é o algoritmo de *back-propagation*. Todos os nós da rede utilizam uma função *sigmoid* como função de ativação. Em seu estado inicial cada nó da rede tem seu peso preenchido com um valor aleatório e a ativação do nó é realizada em função do produto entre os pesos e a entrada, onde se o produto calculado é maior que o limite de ativação do nó, então o nó é ativado. O algoritmo de *back-propagation*, após interação dos dados de treinamento, é responsável por corrigir o peso de cada nó, além de corrigir o número de nós em cada camada. O algoritmo de aprendizagem é executado diversas vezes até que todos os pesos estejam prontos para serem utilizados com os dados de teste.

4.2.3 Regressão Logística

O classificador de regressão logística constrói um modelo de regressão logística com estimadores rígidos [SLC92] para estimar a probabilidade de que, dado um conjunto de características, representado pelo conjunto de proporções de cada instrução ou parâmetro, a instância analisada pertença a uma determinada classe, vírus ou programa. O resultado final será decidido pela classe que apresentar maior probabilidade calculada. Uma das vantagens da classificação através de regressão logística é que, ao contrário da regressão linear, não é necessário assumir que os dados sigam distribuição normal. Desta forma não foi necessário nenhum teste de normalidade dos dados. Embora todos os dados utilizados sejam numéricos a regressão logística é também capaz de trabalhar com dados armazenados na forma de categorias.

A fórmula para o cálculo da probabilidade para uma classe j entre o total de k classes pode ser descrita da seguinte forma:

$$P_j(X_i) = \frac{e^{X_i B_j}}{(\sum_{j=1}^{k-1} e^{X_i B_j}) + 1},$$

com exceção da última classe, que tem sua probabilidade calculada pelo complemento da soma das probabilidades de todas as outras classes. No nosso trabalho foi suficiente calcular a probabilidade do executável pertencer à classe dos vírus, e se esta probabilidade fosse maior que 50% (cinqüenta por cento) então ele seria considerado um vírus, se não era considerado um programa regular. O valor de X_i representa o vetor de características de uma dada instância i, enquanto que B_i representa a matriz computada através do método de otimização de Quasi-Newton. A matriz B_i contém os parâmetros que serão multiplicados pelos valores das variáveis de entrada em X_i. No modelo de regressão logística nem todas as variáveis possuem o mesmo peso, portanto é possível determinar aquelas que não mostram um grau de significância relevante para o cálculo da probabilidade e decidir se ela deve ou não ser considerada, que no caso do Weka é feito

de forma automática baseado com o trabalho de [SLC92].

4.3 DADOS DE TESTE DO DESEMPENHO

Após o treinamento de todos classificadores, eles foram testados com um conjunto especial de teste preparado de forma que:

- Não houvesse vírus ou programas que estivessem contidos no repositório de treinamento;
- Não houvesse vírus ou programas repetidos;
- Houvesse vírus gerados por kits de criação de vírus;
- Houvesse vírus conhecidos pelos antivírus comerciais;
- Dentre os programas regulares analisados, foram incluídos alguns que em algum momento formam classificados como vírus por algum antivírus comercial. Estes dados foram obtidos através dos relatórios encontrados no endereço eletrônico AV-Comparatives [AVC].

O objetivo de não incluir vírus que estivessem contidos no repositório de treinamento foi o de avaliar como a solução se comportaria em um universo onde todos os vírus fossem desconhecidos. Os vírus gerados por kits de criação de vírus também eram desconhecidos pela ferramenta. Para garantir que nenhum dos vírus ou programa tivesse sido repetido nos dois repositórios foi utilizado um algoritmo *hash* (SHA1 - *Secure Hash Algorithm*), que gerou uma identificação para cada arquivo. Antes dos vírus serem adicionados ao repositório de teste, o resultado do seu *hash* era comparado com todos os resultados dos arquivos no repositório de treinamento e somente era utilizado de não pertencesse aos dados de treinamento.

Para montar o resultado final da solução ASAT decidiu-se considerar a resposta dos três classificadores e fazer um sistema onde a maioria dos resultados fosse considerada. Neste caso se dois ou mais dos classificadores tivessem como resultado da análise do executável a classificação de vírus o resultado final da solução seria acusar o arquivo de ser um vírus. No caso contrário, se dois ou mais classificadores escolheram o arquivo como um programa regular, então o resultado final da solução era de não considerar o arquivo um vírus. A arquitetura com o funcionamento do trabalho proposta para detecção de vírus pode ser vista na Figura 4.1.

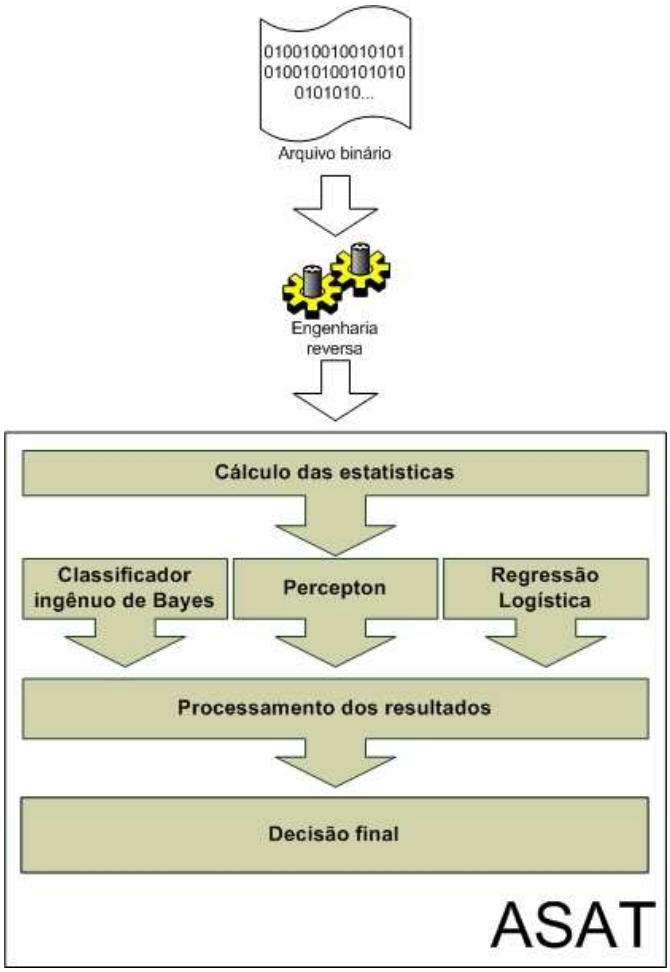

Figura 4.1 Arquitetura da solução ASAT

CAPÍTULO 5

RESULTADOS

Após testar a solução contra as informações contidas na tabela de teste foram registradas as taxas de falso negativo e falso positivo de cada classificador e do resultado final da solução ASAT. Estes resultados podem ser visto no gráfico da Figura 5.1.

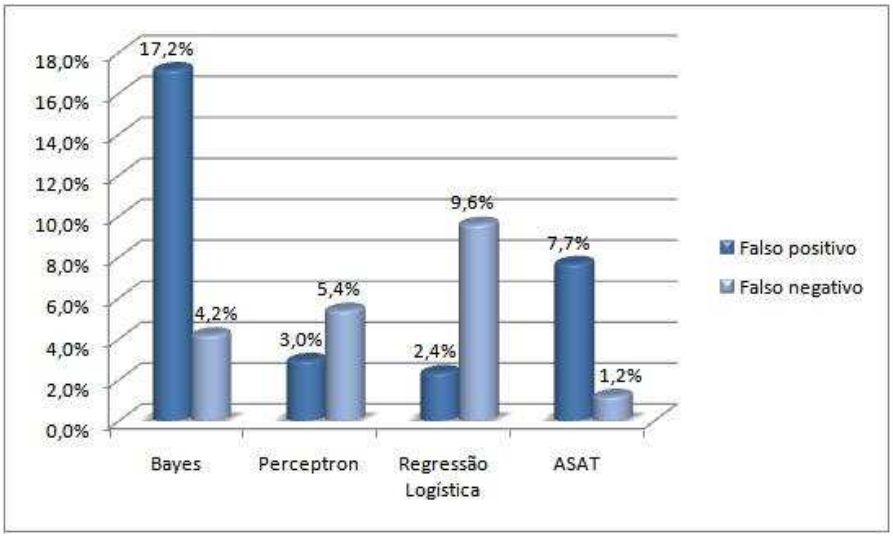

Figura 5.1 Desempenho dos classificadores e da solução ASAT

Resalta-se que o evento de interesse foi a presença de vírus. Desta forma, os casos de falso positivo aconteceram quando um programa regular era classificado como vírus, e os casos de falso negativo quando uma instância de vírus era classificada como programa. Pode-se observar pela tabela que a solução ASAT teve um desempenho melhor do que qualquer um dos classificadores isoladamente quando de considera a taxa de falso negativo.

Pode-se notar que o classificador de Bayes obteve uma taxa elevada de falso positivo (17,1%). Este fato pode ter ocorrido baseado na presença de um grau de dependência entre as variáveis, porém o classificador de Bayes foi utilizado por apresentar a menor taxa de falso negativo entre os três classificadores, fazendo assim com que o resultado final da solução seja balanceado pelo resultado dos três classificadores. A menor taxa de falso positivo foi registrada através do método de regressão logística (2,3669%) e a maior foi através do classificador ingênuo de Bayes; em relação à taxa de falso negativo a ferramenta proposta apresentou o menor valor e a maior taxa ocorreu com o classificador de regressão logística.

Visando otimizar o resultado da solução para certas situações outras duas maneiras de decisão do resultado final foram computadas. A combinação dos resultados foi feita

de forma que se pelo menos um dos classificadores considerasse o arquivo um vírus, então este executável seria classificado como vírus. Esta forma de calcular o resultado final visa diminuir a taxa de falso negativo, visto que mais programas serão considerados vírus, porém aumenta-se a taxa de falso positivo. O desempenho obtido pela ferramenta ASAT, utilizando tal metodologia para decisão do resultado final, pode ser visto na Tabela 5.1. De forma semelhante, porém visando diminuir a taxa de falso positivo, também foram calculados os resultados considerando um executável um vírus somente quando ele tivesse sido classificado como um vírus por todos os classificadores, gerando os resultados também apresentados na Tabela 5.1.

Tabela 5.1 Desempenho da solução ASAT segundo a metodologia de decisão

Metodologia	**Taxa de falso positivo**	**Taxa de falso negativo**
Maioria	7.6923%	1.2012%
Baixa taxa de falso positivo	1.1834%	9.0090%
Baixa taxa de falso negativo	28.9941%	1.2012%

5.1 COMPARAÇÃO COM ANTIVÍRUS COMERCIAIS

Diversas ferramentas comerciais podem ser encontradas hoje no mercado para lidar com o problema de detecção de vírus. Alguns antivírus possuem versões gratuitas com algumas limitações que os diferenciam das versões comercializadas. Considerando este fato, as versões do antivírus utilizadas nos testes comparativos foram versões integrais dos produtos. Contendo apenas, em alguns casos, uma limitação do tempo de utilização do antivírus.

Estes antivírus comerciais utilizam em sua grande maioria a técnica de assinaturas de arquivos para detectar os vírus. Esta técnica [Szo05] consiste em conhecer uma ou mais instâncias do vírus e analisar que partes podem ser utilizadas para identificar o vírus de forma única, ou seja, procura-se identificar pedaços de código de máquina que estejam presentes em todas as instâncias do mesmo vírus. O fragmento de código utilizado nas assinaturas dos antivírus deve ser longo o suficiente de forma a diminuir a probabilidade de que o mesmo seja encontrado em outros executáveis que não o vírus, evitando assim um erro do tipo falso positivo.

As estruturas das assinaturas, da mesma forma que o vírus, também evoluíram e atualmente contêm mais informações que apenas fragmentos de código. É possível encontrar em assinaturas informações como: tipos de arquivos atacados pelo vírus, forma de proliferação do vírus, sistemas operacionais que o vírus ataca, ou até mesmo caracteres coringas que servem para identificar o vírus mesmo que tenham sido feitas pequenas mudanças em seu código. Também é comum que a assinatura do vírus seja composta pelo resultado de algum algoritmo hash, que utiliza como entrada o fragmento de código de máquina. Esta medida faz com que o tamanho da assinatura não varie independentemente do tamanho do código utilizado para identificar o vírus.

Existem ainda programas de antivírus que contêm em seu código máquinas virtuais, capazes de simular o funcionamento de alguns vírus. O objetivo desta máquina virtual é de fazer o vírus funcionar em um ambiente seguro e controlado, onde as ações dos executáveis serão analisadas de forma a decidir se o mesmo possui o comportamento de um vírus. Esta metodologia dinâmica, apesar de apresentar bons resultados principalmente em vírus que utilizam criptografia em seu código, exige um processamento superior ao da detecção de vírus através de assinaturas.

A detecção de vírus por assinaturas não apresenta bons resultados na classificação de vírus com apenas algumas mudanças em seu código [SXCM04]. Outra desvantagem do método de detecção por assinatura é a necessidade de possuir pelo menos uma instância do vírus para gerar sua assinatura. Isto faz com que os antivírus não consigam apresentar bom desempenho com vírus para os quais ainda não se possuem assinaturas. Visando melhorar este problema algumas empresas de segurança desenvolvem em seus antivírus detecção através de máquinas virtuais ou heurísticas, que irão classificar um executável baseado em comportamentos comuns ao de vírus.

Para avaliar o desempenho de detecção de vírus da solução ASAT em relação aos antivírus comerciais, foram comparados os resultados da solução com a análise de todos os vírus utilizados para construir o repositório de testes, pelos seguintes antivírus comerciais:

- McAfee VirusScan 8.5*i*
- AVG 7.1 [AVG]
- avast! v4.7 [Ava]
- AVIRA v7.00 [AVI]
- Norton Antivirus 2008

Todos os antivírus utilizados na pesquisa tiveram seus arquivos de assinaturas atualizados com suas respectivas versões mais recentes. O resultados comparativos das taxas de falso negativo são apresentadas no gráfico da Figura 5.2. Nenhum dos programas de antivírus apresentou casos de falso positivo. Este gráfico mostrou que a menor taxa de falso negativo correspondeu a ferramenta ASAT e a mesma variou de 2,5% a 34,5% para os antivírus comerciais.

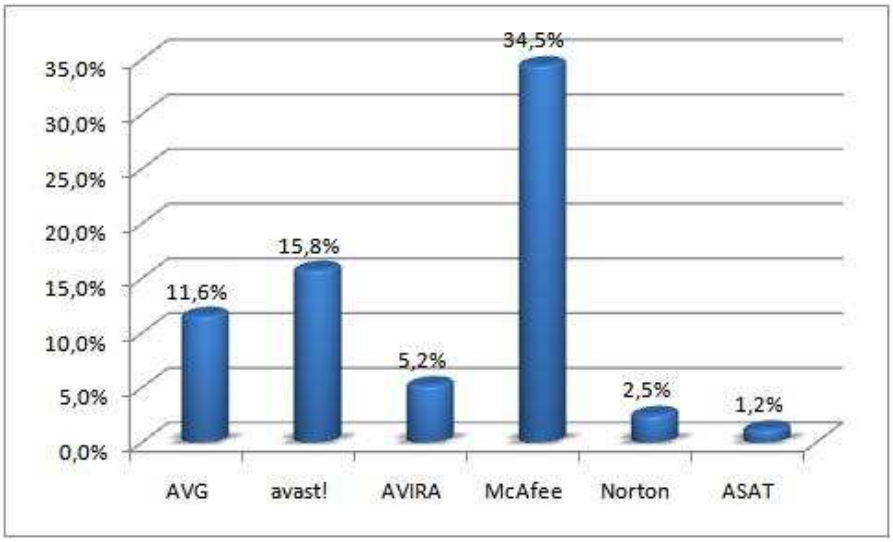

Figura 5.2 Desempenho da solução ASAT segundo a metodologia de decisão

CAPÍTULO 6

CONCLUSÃO E TRABALHOS FUTUROS

Como conclusão do nosso trabalho podemos dizer que foram obtidos resultados animadores que mostraram avanços no ramo de detecção de vírus. Foi apresentada uma metodologia nova para reconhecimento de padrões em arquivos maliciosos desconhecidos. Esta metodologia tem como base a criação de parâmetros, que por sua vez são computados utilizando como referência algumas estatísticas calculadas sobre o código de máquina de executáveis. Tais parâmetros servem como entrada para classificadores que têm suas respostas combinadas e transformadas na resposta final. Foram realizados testes sobre a metodologia proposta. Os testes consistiram em avaliar o resultado da solução em relação a novos vírus e também em comparar seu desempenho com os de outros programas de detecção de vírus.

Como trabalhos futuros, pretendemos utilizar como parâmetros outras estatísticas extraídas do código, como ,por exemplo, o número de chamadas para bibliotecas ou funções do sistema operacional. Outra abordagem a ser testada será a utilização de técnicas específicas para lidar com um sistema de múltiplos classificadores, como é sugerido em [SGD06], para poder melhor combinar os resultados dos classificadores. Ainda outro ponto a ser explorado será a verificação dos diferentes tipos de classificadores para os diferentes tipos de vírus, como por exemplo worms e cavalos-de-Tróia. Visando melhorar o desempenho de ASAT, a ferramenta poderia eliminar a fase de obtenção do código de máquina e extrair as informações necessárias diretamente do código binário. Neste caso, ASAT passaria a funcionar somente para sistemas operacionais no qual estivesse configurado com a arquitetura do processador.

REFERÊNCIAS BIBLIOGRÁFICAS

[AACKS04] Tony Abou-Assaleh, Nick Cercone, Vlado Keselj, and Ray Sweidan. N-gram-based detection of new malicious code. *COMPSAC*, 2:41–42, 2004.

[Adl90] Leonard M. Adleman. An abstract theory of computer virus. In *Advances in Cryptology - CRYPTO'88*, pages 354–374. Springer, 1990.

[Ava] avast! http://www.avast.com/.

[AVC] Av-comparatives. http://www.av-comparatives.org/.

[AVG] Avg. http://www.download.com/.

[AVI] Avira. http://www.avira.com.

[BMKK06] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[BMM07] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Code normalization for self-mutating malware. *Security and Privacy, IEEE*, 3(1):46–54, 2007.

[CJ03] Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, pages 169–186, Berkeley, CA, USA, 2003. USENIX Association.

[CJ04] Mihai Christodorescu and Somesh Jha. Testing malware detectors. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 34–44, Boston, MA, USA, 2004. ACM Press.

[CJK+07] Mihai Christodorescu, Somesh Jha, Johannes Kinder, Stefan Katzenbeisser, and Helmut Veith. Software transformations to improve malware detection. *Journal in Computer Virology*, 3(4):253–265, 2007.

[CJS+05] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Xiaodong Song, and Randal E. Bryant. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*, pages 32–46, 2005.

[CL06] Mohamed R. Chouchane and Arun Lakhotia. Using engine signature to detect metamorphic malware. In *WORM '06: Proceedings of the 4th ACM workshop on Recurring malcode*, pages 73–78, New York, NY, USA, 2006. ACM.

[Coh86] Frederick B. Cohen. *Computer viruses*. PhD thesis, University of Southern California, 1986.

[Coh94] Frederick B. Cohen. *A Short Course on Computer Viruses*. Wiley Professonal Computing, 1994.

[CW00] David M. Chess and Steve R. White. An undetectable computer virus. In *Virus Bulletin Conference*, 2000.

[Dow] Download.com. http://www.download.com/.

[Eur] Euro download. http://www.eurodownload.com/.

[GNU] Gnu binutils. http://www.gnu.org/software/binutils/.

[IDA] Ida pro. http://www.datarescue.com/.

[Jac90] Keith Jackson. Nomenclature for malicious programs. *Virus Bulletin*, 1990.

[JL95] George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, 1995.

[KDM+96] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Applied Categorical Structures*, 3(1):77–108, 1996.

[KJB+06] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *22nd Annual Computer Security Applications Conference*, pages 338–348, 2006.

[KKM+05] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *8th International Symposium on Recent Advances in Intrusion Detection RAID*, 2005.

[Koh95] Teuvo Kohonen. *Self-Organizing Maps*. Springer, 1995.

[KRVV04] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2004. USENIX Association.

[Kun06] Ludmila I. Kuncheva. On the optimality of naïve bayes with dependent binary features. *Pattern Recogn. Lett.*, 27(7):830–837, 2006.

[LD03] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299, New York, NY, USA, 2003. ACM.

[LS95] P. Langley and S. Sage. Induction of selective bayesian classifiers. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pages 399–406, 1995.

[LSC+06] Zhichun Li, Manan Sanghi, Yan Chen, Ming-Yang Kao, and Brian Chavez. Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 32–47, Washington, DC, USA, 2006. IEEE Computer Society.

[Met] Metaphor. http://securityresponse.symantec.com/avcenter/venc/data/w32.simile.html.

[NKS05] James Newsome, Brad Karp, and Dawn Song. Polygraph: Automatically generating signatures for polymorphic worms. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 226–241, Washington, DC, USA, 2005. IEEE Computer Society.

[SEZS01] Matthew G. Schultz, Eleazar Eskin, Erez Zadok, and Salvatore J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 38–49. IEEE Computer Society, 2001.

[SGD06] Zbigniew Suraj, Neamat El Gayar, and Pawel Delimata. A rough set approach to multiple classifier systems. *Fundam. Inf.*, 72(1-3):393–406, 2006.

[SL02] Prabhat K. Singh and Arun Lakhotia. Analysis and detection of computer viruses and worms: an annotated bibliography. *SIGPLAN Not.*, 37(2):29–35, 2002.

[SLC92] J. C. Van Houwelingen S. Le Cessie. Ridge estimators in logistic regression. *Applied Statistics*, 41:191–201, 1992.

[SRMS05a] A. Sulaiman, K. Ramamoorthy, Srinivas Mukkamala, and Andrew H. Sung. Disassembled code analyzer for malware *dcam*. In *Information Reuse and Integration, Conf, 2005*, pages 398–403. IEEE Computer Society, 2005.

[SRMS05b] A. Sulaiman, K. Ramamoorthy, Srinivas Mukkamala, and Andrew H. Sung. Malware examiner using disassembled code *medic*. In *Proceedings of the*

2005 IEEE Workshop on Information Assurance and Security, pages 428–429. IEEE Computer Society, 2005.

[SXCM04] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala. Static analyzer of vicious executables *save*. In *ACSAC '04: Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC'04)*, pages 326–334, Washington, DC, USA, 2004. IEEE Computer Society.

[Szo05] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison Wesley Professional, 2005.

[VXH] Vx heavens. http://vx.netlux.org/.

[VY06a] Amit Vasudevan and Ramesh Yerraballi. Cobra: Fine-grained malware analysis using stealth localized-executions. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy (S&P'06)*, pages 264–279, Washington, DC, USA, 2006. IEEE Computer Society.

[VY06b] Amit Vasudevan and Ramesh Yerraballi. Spike: engineering malware analysis tools using unobtrusive binary-instrumentation. In *ACSC '06: Proceedings of the 29th Australasian Computer Science Conference*, pages 311–320, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.

[Wek] Weka. http://www.cs.waikato.ac.nz/ml/weka/.

[Wes01] Douglas B. West. *Introduction to Graph Theory*. Prentice-Hall, 2001.

[WF05] Ian H. Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[WM06] Matt Webster and Grant Malcolm. Detection of metamorphic computer viruses using algebraic specification. *Journal in Computer Virology*, 2(3):149–161, 2006.

[WS06] Wing Wong and Mark Stamp. Hunting for metamorphic engines. *Journal in Computer Virology*, 2:211–229, 2006.

[YSR+06] Stewart Yang, Jianping Song, H. Rajamani, Taewon Cho, Yin Zhang, and R. Mooney. Fast and effective worm fingerprinting via machine learning. In *IEEE International Conference on Autonomic Computing*, pages 311–313. IEEE Computer Society, 2006.

[YUN06] In Seon Yoo and Ulrich Ultes-Nitsche. Non-signature based virus detection. *Journal in Computer Virology*, 2(3):163–186, 2006.

[Zha04] Harry Zhang. The optimality of naïve bayes. In *Proc. 17th Internat. FLAIRS Conf.*, 2004.

[ZR07] Qinghua Zhang and Douglas S. Reeves. Metaaware: Identifying metamorphic malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 411–420, 2007.

Dissertação de Mestrado apresentada por **Eduardo Mazza Batrista** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"ASAT: Uma Ferramenta para Detecção de Novos Vírus"**, orientada pelo **Prof. Ruy José Guerra Barretto de Queiroz** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Silvio de Barros Melo
Centro de Informática / UFPE

Prof. Francisco Cribari Neto
Departamento de Estatística / UFPE

Prof. Ruy José Guerra Barretto de Queiroz
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 6 de junho de 2008.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.