Rastreabilidade de Códigos Executáveis usando Redes Neurais

Tiago M. Nascimento[1,2], Luiz F. R. C. Carmo[1], Davidson R. Boccardo,[1] Raphael C. Machado[1], Charles B. Prado[1]

[1]Instituto Nacional de Metrologia, Normalização e Qualidade Industrial (INMETRO) Rio de Janeiro - RJ

[2]Universidade Federal do Rio de Janeiro - Rio de Janeiro - RJ

{tmnascimento, lfrust, drboccardo, rcmachado, cbprado}@inmetro.gov.br

Abstract. *Traceability of codes refers to the mapping between equivalent codes written in different languages – including high-level and low-level programming languages. In the field of Legal Metrology, it is critical to guarantee that the software binary code embedded in a meter corresponds to a program source code that was previously approved by the Legal Metrology Authority. In this paper, we propose a novel approach for correlating source and binary codes using artificial neural networks. Our approach correlates the source code with the binary code by feeding the neural network with logical flow characteristics of such codes. Any incidence of false positives is obviously a critical issue for software evaluation purposes. Our evaluation using real code examples shows a typical correspondence rate between 62% and 90% for the traceability of the binary codes with the very low rate of 4% false positives.*

Resumo. *Rastreabilidade de códigos refere-se ao mapeamento entre códigos equivalentes escritos em linguagens diferentes — inclusive linguagens de programação de alto nível e de baixo nível. No campo da Metrologia Legal, é fundamental garantir que o código binário de um software embarcado em um medidor corresponde a um código fonte do programa que foi previamente aprovado pela Autoridade Metrológica. Neste trabalho, é proposta uma nova abordagem para correlacionar códigos fonte e binário usando redes neurais artificiais. Nossa abordagem correlaciona o código fonte com o código binário utilizando-se de uma rede neural artificial alimentada pelas características do fluxo lógico do programa. Qualquer incidência de falsos positivos é um fator crítico para fins de avaliação de software. Nossa avaliação, usando exemplos de código real, mostra uma correspondência entre 62% e 90% para a rastreabilidade dos códigos binários com uma taxa de 4% de falsos positivos.*

1. Introdução

O número de dispositivos baseados em software embarcado vem crescendo enormemente. Estes softwares embarcados trazem consigo um conjunto de desafios completamente novos, tais como novas formas de validação, exigências, possibilidades de erros, imprecisões e falhas de segurança. Um risco crítico associado à presença de

software em dispositivos de medição é o interesse na adulteração do software para se obter vantagens. Por exemplo, na área automotiva, uma pessoa mal-intencionada pode querer reduzir a quilometragem do veículo para obter um maior valor de revenda. Outro caso seria o do proprietário de uma balança, o qual pode modificar o software de tal forma para que as novas medições (erradas) possam beneficiá-lo.

Em diversas áreas, os dispositivos envolvidos nas relações humanas devem ser submetidos a algum tipo de controle externo. Uma das áreas em questão — que motivou o presente trabalho — é a área da Metrologia Legal. O exemplo da balança, acima, é um exemplo típico da relação humana no domínio da Metrologia Legal. Para se ter uma lealdade das transações entre o vendedor e o comprador, é importante assegurar que a balança exiba corretamente o peso do produto a ser comercializado. Naturalmente, a única forma de garantir o funcionamento correto dessa balança é a realização de alguma forma de verificação externa: uma entidade não-comprometida com vendedor ou comprador que avalia o peso da balança e oferece algum tipo de certificação de seu funcionamento correto. Tipicamente, essas entidades de certificação são governamentais ou organizações sem fins lucrativos.

Tradicionalmente, os processos para certificação da qualidade envolvem várias etapas de testes físicos do dispositivo sob avaliação. Tais testes devem verificar se o dispositivo apresenta um funcionamento correto independente de fatores externos como, por exemplo, umidade e temperatura. Com o advento dos dispositivos com software embarcado, o processo de certificação deve envolver, adicionalmente, a validação dos mesmos. A entidade de certificação deve garantir, por exemplo, que o software embarcado na balança não contenha algum *backdoor* que ative — sob o comando do proprietário da máquina — uma função espúria que diminua ou aumente o peso exibido.

A verificação do código binário é a única forma confiável de detectar as modificações intencionais escondidas, como demonstrado por Thompson em seu Prêmio Turing [Thompson 1984]. Apesar do trabalho de Thompson, ser considerado teórico, suas idéias foram colocadas em prática pelo malware W32.Induc.A [McDonald 2010]. Em sistemas grandes e complexos, entretanto, a verificação binária pode exigir um esforço elevado para rastrear completamente manipulações de variáveis e realizar análises de vulnerabilidade. Logo a abordagem usual é realizar essa verificação sobre o código fonte. No entanto, a verificação do código fonte não é suficiente para dar qualquer garantia sobre o comportamento do software embarcado no dispositivo relacionado. Para certificar que o código binário, que está em execução em algum dispositivo, funcione corretamente, é preciso garantir que esse código binário foi, de fato, gerado a partir do código fonte aprovado através de um processo de compilação honesto. As duas abordagens de validação são mostradas na Figura 1.

A rastreabilidade de códigos executáveis é o processo que estabelece essa correspondência do código fonte avaliado com o código binário embarcado no dispositivo. Isto é, uma vez que o código fonte de uma versão de software dado é analisado, avaliado e aprovado, é necessário verificar se um determinado código binário — que estará, de fato, em execução no dispositivo — corresponde a este código fonte. Uma maneira simples e direta de se realizar esta "rastreabilidade" envolve reproduzir exatamente o ambiente de desenvolvimento utilizado pelo desenvolvedor e

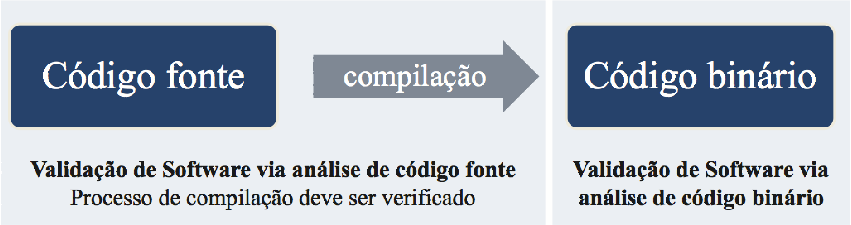


Figura 1. Abordagens para validação: análise do código fonte com verificação da compilação × análise do código binário.

compilar o código fonte aprovado neste ambiente, verificando se o código binário gerado é conforme o esperado. Contudo, essa abordagem apresenta duas desvantagens: a primeira desvantagem está relacionada com o custo e a complexidade necessários para manter ambientes de desenvolvimento de vários softwares, e a segunda com o Prêmio Turing de Thompson [Thompson 1984]: a menos que a "transformação de linguagem" realizada pelo compilador possa ser completamente caracterizada — e isso exigiria uma análise do código binário do compilador — não é possível garantir que o processo de compilação, por si só, não introduz algum tipo de falha ou comportamento malicioso no software. Outra forma de executar o software de rastreabilidade é a auditoria do ambiente de desenvolvimento de software do fabricante. Tal abordagem apresenta desvantagens semelhantes às descritas anteriormente.

No trabalho aqui apresentado é proposta uma estratégia diferente para verificar se dois programas, escritos em linguagens diferentes (tipicamente código fonte e código binário de máquina), descrevem o mesmo comportamento do software. Apresentamos uma nova abordagem para a realização de rastreabilidade de códigos executáveis usando redes neurais artificiais (RNA). Mais especificamente, foram coletadas propriedades do fluxo lógico do programa herdadas dos grafos de controle de fluxo das funções e do grafo de chamadas, e uma RNA foi utilizada como arbitrador para descobrir o grau de semelhança entre os códigos fonte e binário.

O restante do artigo está estruturado da seguinte forma. Seção 2 discute os trabalhos relacionados à rastreabilidade. Seção 3 descreve o método proposto. O método caracteriza-se pela extração das propriedades de códigos fonte e objeto e uso destas como dados de entrada para uma RNA. Nesta Seção também é mostrado a aplicação de uma RNA para descobrir o grau de semelhança entre um código fonte e um código de objeto. Seção 4 apresenta a avaliação empírica do método apresentado, e por fim, Seção 5 contém as conclusões observadas.

2. Trabalhos Relacionados

Não há muitas contribuições relacionadas com a rastreabilidade de códigos executáveis na literatura, ou seja, que efetivamente realizam o mapeamento entre fonte e binário. No entanto, existem trabalhos de verificação e análise de códigos fonte e binário que podem auxiliar na realização da abordagem proposta.

No trabalho de Quinlan *et al.* [Quinlan and Panas 2009] é proposto um arcabouço para verificação da existência de defeitos de software (bugs), tanto no código executável quanto no código fonte. Entretanto, não é realizado o mapeamento entre o fonte e o binário. Hassan *et al.* [Hassan et al. 1995] observa que a arquitetura de

alguns programas está intrinsecamente relacionada com seu fonte e objeto. Nesse método são utilizados dois extratores: um extrator de desvios de um código objeto (LDX), e um extrator de rótulos de um código fonte (CTAGX). Depois do processo de extração, é realizado um comparativo dos resultados obtidos de forma a inferir a arquitetura de software quanto à relação de correspondência dos códigos. Hatton [Hatton 2005] investigou a densidade de defeitos como uma relação entre um código objeto e código fonte. Para isto é usado o tamanho (número de linhas) do código fonte. Utilizando a densidade de defeito de um programa (defeitos por 1000 linhas) ele fez a busca do relacionamento com o código objeto. Nenhum desses artigos foca em rastreabilidade de um código executável nem sabemos da existência de tais trabalhos. Buttle [Buttle 2001] utiliza a estrutura lógica do código executável (grafo de controle de fluxo) para coincidir com a estrutura lógica do código fonte. Em nossa proposta também utilizamos as características do fluxo do programa, no entanto, apresentamos também outras relações que podem coexistir entre fonte e binário, com a finalidade de obter uma melhor correspondência.

Em uma direção tangencial, existem trabalhos que realizam o mapeamento entre códigos binários com o intuito de verificar versões sequenciais de um mesmo trecho de código do software, diferenças de um mesmo binário, assim como atualizações de software [Wang et al. 2002, Flake 2004]. Para isso, estes trabalhos utilizam da estrutura lógica do programa — grafos de controle de fluxo. Outros trabalhos sobre diferença de binários podem ser encontradas em [Oh 2009], que também apresentam técnicas para subverter algoritmos que utilizam grafos para comparação de binários. Estes trabalhos, apesar de estarem associadas somente com binários, podem ser trazidos para o problema de rastreabilidade.

Existem também trabalhos relacionados no âmbito de rede neural artificial e algumas contribuições em segurança, para abordagens de criptografia. Um novo algoritmo de criptografia digital de imagens utilizando redes neurais é apresentado em [Zhenga 2009]. Esse algoritmo utiliza uma rede neural celular hipercaótica usando características de sistemas dinâmicos caóticos. Métodos baseado em inteligência artificial para validação de software podem ser encontradas na literatura, para verificação e para confiabilidade de software. As abordagens em [Zeng and Rine 2004] e [Zhenga 2007] propõem a utilização de redes neurais na predição para confiabilidade de software, e em [Zeng and Rine 2004] é também proposto um método para correção de defeitos de software baseado na previsão da rede neural. Em [Reddy et al. 2007], uma rede neural é usada para prever falhas de módulos em uma aplicação web, e em [Lenic et al. 2004], um sistema de auto-organização é construído para se obter a confiabilidade desses módulos.

3. Abordagem proposta

A abordagem proposta para a rastreabilidade de software envolve duas etapas. Na primeira etapa, extraímos características de ambos os códigos: fonte e binário. Estas características podem ser simples, como por exemplo, o tamanho do código, ou mais complexas, como as derivadas a partir dos grafos de controle de fluxo ou de chamada. Na segunda etapa, um arbitrador é utilizado para determinar a equivalência entre o código fonte e o código binário baseado nestas características. Esta seção exibirá como a abordagem foi colocada em prática com o uso de quatro características

(tamanho, número de procedimentos, o número de vértices do grafo de controle de fluxo e número de arestas do grafo de controle de fluxo) e uma rede neural artificial como arbitrador.

3.1. Processo de extração

Em um processo de compilação, uma grande quantidade de informações relevantes para o processo de rastreabilidade pode ser perdida. O tamanho desta perda é dependente do processo de compilação. A seguir, destacamos algumas propriedades, independente do código (fonte ou objeto), que podem ser explicitamente ou implicitamente disponíveis, com o intuito de adquirir insumos sobre a complexidade na concepção do método de rastreabilidade.

Considerando o fato de que a maioria dos softwares embutidos é escrita em linguagem imperativa, propriedades como nomes de variáveis, tipos de variáveis e nomes de procedimento podem ser perdidos durante o processo de compilação visto que o objetivo do compilador é maximizar desempenho. Esta maximização degrada a legibilidade do código, que por conseguinte torna o processo de extração de características fonte e executável uma tarefa complexa.

Uma outra possibilidade seria o uso do conteúdo dos dados dos códigos fontes e executáveis, que embora não explícito, pode ser calculado pela análise do fluxo de dados. O escopo da análise do fluxo de dados é perceptivelmente diferente para fonte e executável. Enquanto para o código fonte o escopo é no nível de variável; no executável é no nível de memória e registradores. Esta diferença de nível de linguagem é um dos fatores que aumentam o número de instruções contidos no executável em comparação com o código fonte, assim, beneficiando processos de otimização, porém exigindo uma análise de fluxo de dados mais dispendiosa. Assim, apesar desta propriedade ser uma candidata para rastrear o conteúdo dos dados do executável, a mesma é dispendiosa.

A lógica do programa, representada pelo fluxo de controle do programa, é certamente mantida no executável, ainda que sua árvore de execução não seja claramente estruturada como no código fonte. Alguns trabalhos para construção do grafo de controle de fluxo são encontrados em [Moretti et al. 2001]. O fluxo de controle de um programa pode ser caracterizado pelo seu grafo de chamadas e pelo grafo de fluxo para cada uma de suas funções. O grafo de chamadas exibe o relacionamento entre funções que chamam e funções que são chamadas. O grafo de fluxo representa os blocos básicos de cada função e o fluxo de informação baseado nos desvios condicionais e incondicionais.

As características utilizadas em nossa abordagem para rastreabilidade são baseadas na lógica do programa. Assim, foram extraídos o número de nós, arestas, funções baseado na lógica dos programas, assim como o tamanho (em bytes). Apesar dos tamanhos dos códigos serem obtidos diretamente, o número de nós e arestas do grafo de controle de fluxo para o código fonte, gerados a partir do depurador gcc com o parâmetro "–fdump-tree-cfg", foram obtidos através de um *shell script* que realiza a contagem destas propriedades no arquivo gerado pelo depurador. Para a extração do número de nós e arestas do grafo de controle de fluxo para o código executável foi utilizado um *script* em python sob o desmontador IDA [IdaPro 2010].

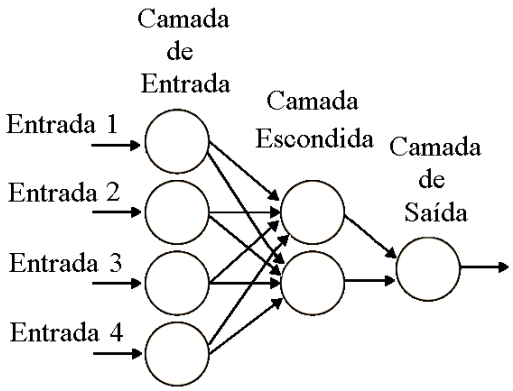


Figura 2. Topologia de rede neural.

O número de funções para o código fonte e executável foram obtidos através de um *shell script* e de um *script* em python, respectivamente, executando sobre um grafo de chamadas construído através da ferramenta GNU cflow [Poznyakoff 2010] para o código fonte e do depurador IDA para o código executável.

3.2. Rede neural artificial

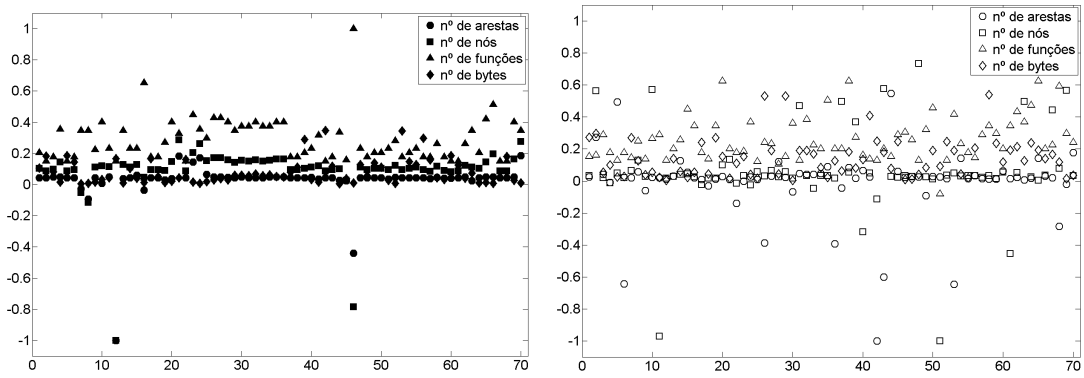
As redes neurais, com suas capacidades de extração de significado em dados complexos ou imprecisos, podem ser usadas para extrair padrões e detectar tendências, que são complexas para serem notadas manualmente ou por técnicas computacionais. Assim, uma rede neural treinada pode ser caracterizada como "*expert*" na categoria de informação a ela designada. A aplicabilidade das redes neurais é extensa, contudo pode sem geralmente dividida em três classes: classificação, reconhecimento e previsão.

Neste trabalho, o foco principal será em redes neurais aplicado ao problema de classificação. Para isso, é proposta a utilização de uma rede neural Cascade-Forward Backpropagation, uma vez que é amplamente utilizada neste contexto de classificação [Ciocoiu 2002, Asadi et al. 2009, Haykin 1998]. Na próxima Seção, apresentamos os detalhes da implementação da rede neural.

4. Implementação da Rede Neural

Para desenvolver a rede neural artificial, as características do grafo de controle de fluxo (arestas e nós) foram introduzidas nos nós 1 e 2 de entrada da rede Cascade-Forward com o processo de treinamento Backpropagation [Hertz et al. 1991]. O número de funções (extraído do grafo de chamada) foi introduzido no nó 3 de entrada, e o tamanho dos códigos (em bytes) no nó 4 de entrada. A partir da análise empírica, a melhor configuração neural foi construída com dois neurônios na camada escondida, e um neurônio na camada de saída (veja a figura 2).

Para simular a rede neural utilizamos a ferramenta "Redes Neurais" do Matlab [Moler 1980]. A função de ativação foi a tangente hiperbólica e para a fase de treinamento foi definido um valor de +1 quando os parâmetros de entrada representam o código objeto correspondente ao seu código fonte (chamada classe 0), caso contrário, o valor alvo foi definido como -1 (chamada classe 1).



(a) Conjunto verdadeiro. (b) Conjunto negativo.

Figura 3. Treinamento dos conjuntos verdadeiros e falsos.

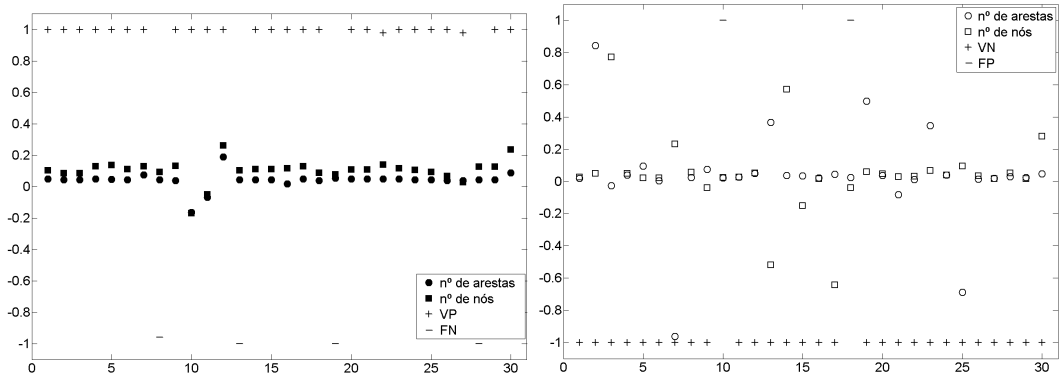
Para o primeiro processo de treinamento, 100 amostras para ambas associações: verdadeira (classe 0) e falsa (classe 1) entre o código executável e seu código fonte foram utilizadas. Para ambas as classes, o percentual usado para fase de treinamento foi de 70% e as amostras restantes foram utilizadas para a fase de testes. Figura 3 ilustra um exemplo de treinamento, em que a Figura 3(a) representa o treinamento para o conjunto positivo, *i.e.*, características equivalentes para o par código fonte e objeto, enquanto Figura 3(b) representa o treinamento para o conjunto negativo, criado simplesmente por valores aleatórios. As características das amostras da classe 0 são representadas por símbolos (círculo, quadrado, triângulo e losango) em preto, e as características das amostras da classe 1 são representadas pelos mesmos símbolos, porém, em branco. Todas as características são correlacionadas pela subtração da característica do código fonte pela correspondente característica do código executável, com exceção do tamanho, no qual é aplicada uma divisão. O eixo 'X' da Figura representa o enésimo programa e o eixo 'Y' as características normalizadas uma vez que a RNA utiliza somente valores contidos no intervalo [-1,1]. Esta etapa de normalização foi calculada dividindo todos dados de cada parâmetro pelo maior de seus valores.

4.1. Avaliação Empírica

Nesta seção serão apresentados os resultados de uma avaliação empírica da proposta de rastreabilidade apresentada neste artigo. Esta avaliação pode ser dividida em duas partes: 1) seleção das características relevantes e 2) avaliação dos programas modificados por códigos maliciosos. Para esta avaliação, compilou-se um conjunto de 100 códigos-fonte da linguagem C [Burkard 2010, Oliveira Cruz 2010] utilizando o compilador gcc do ambiente Linux. Após a fase de compilação, foi-se extraído as características dos códigos como previamente descrito na Seção 3.

4.1.1. Seleção das características relevantes

Para verificar a importância do conjunto de parâmetros de entrada a ser utilizado pelo método de rastreabilidade, foi realizada uma estratégia de treinamento individual, verificando a contribuição de cada parâmetro com a rastreabilidade. Os



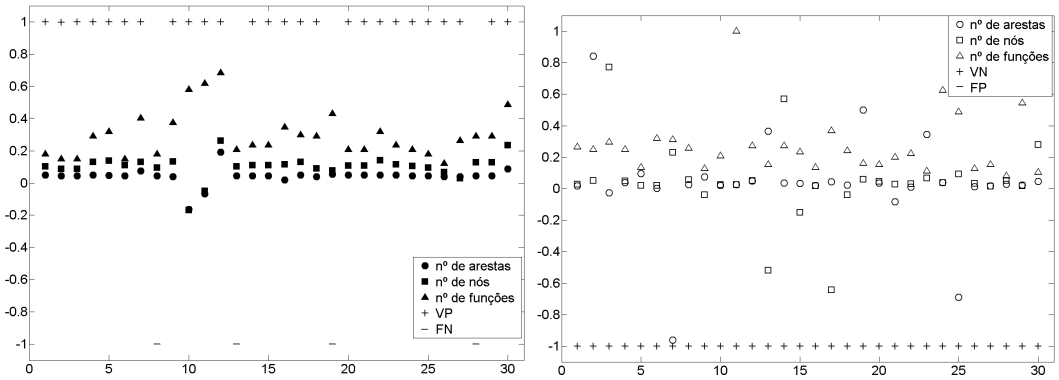
(a) Conjunto verdadeiro. (b) Conjunto falso.

Figura 4. Avaliação da RNA usando dois parâmetros: número de nós e arestas dos grafos de controle de fluxo.

parâmetros avaliados referem-se a propriedades extraídas do grafo de chamadas e do grafo de controle de fluxo de cada função, assim como os tamanhos dos códigos. Nesta abordagem, a rede neural artificial foi treinada e simulada para todos subconjuntos possíveis destes parâmetros, verificando-se a importância de cada parâmetro. Nas Figuras 4, 5 e 6 são exibidos os resultados da melhor simulação utilizando-se dois, três e quatro parâmetros, respectivamente. Vale ressaltar que a melhor simulação visa a minimização dos falsos positivos visto que qualquer incidência de falsos positivos é obviamente um aspecto crítico para avaliação de software. Contudo, falsos negativos também foram considerados, pois uma alta taxa de falsos negativos exigiria uma análise secundária. A seguir, detalhamos como foi comparado o desempenho desta seleção.

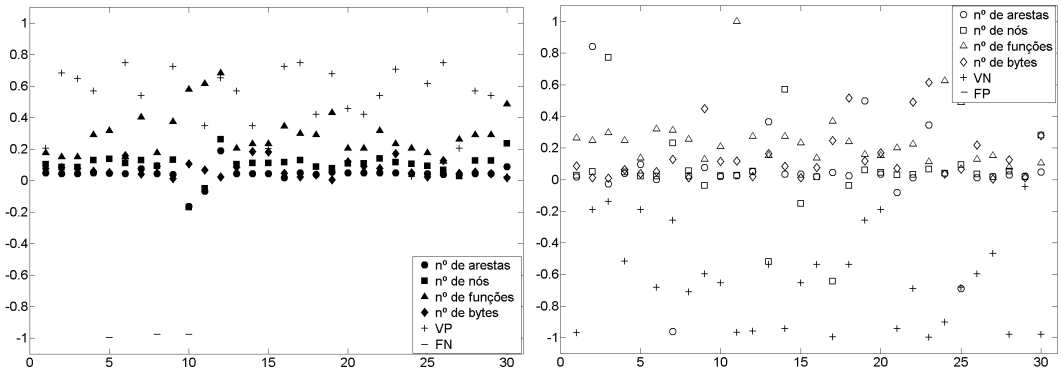
Primeiramente, para um conjunto de dois parâmetros, os mais relevantes foram os extraídos do grafo de controle de fluxo, ou seja, número de nós e de arestas de cada função do programa. Para três parâmetros, a melhor alternativa foi a adição do número de funções extraído dos grafos de chamadas, e por fim, os tamanhos (em bytes) dos códigos do programa. A Figura 4(a) apresenta os resultados de nosso método de treinamento e simulação dos parâmetros referentes aos grafos de controle de fluxo para um conjunto de pares verdadeiros (códigos fonte e objeto), e a Figura 4(b) apresenta os resultados para os pares falsos, ou seja, um objeto que não corresponde a um código fonte. A Figura 4(a) apresenta 4/30 ($\approx$13%) falsos negativos (FN) e 26/30 ($\approx$87%) verdadeiros positivos (VP), e a Figura 4(b) exibe 2 de 30 ($\approx$6%) falsos positivos (FP) e 28/30 ($\approx$94%) de verdadeiros negativos (VN).

As Figuras 5(a) e 5(b) apresentam os resultados para três parâmetros (adicionando o número de funções extraídos do grafo de chamadas). Nesta avaliação, os resultados obtidos pela rede neural artificial foram aperfeiçoados, o número de falsos positivos fornecido foi zero e a taxa de ($\approx$13%) de falsos negativos se manteve. Já para quatro parâmetros (adição dos tamanhos dos códigos), a avaliação apresentou menos falsos negativos (10%) em comparação com as avalições anteriores (veja Figuras 6(a) e 6(b)). A fase de treinamento para esta última avaliação é o da Figura 3. A seguir, avaliaremos nossa proposta diante de programas infectados por códigos maliciosos.



(a) Conjunto verdadeiro. (b) Conjunto falso.

Figura 5. Avaliação da RNA usando três parâmetros: número de nós e arestas dos grafos de controle de fluxo, e número de funções do grafo de chamadas.



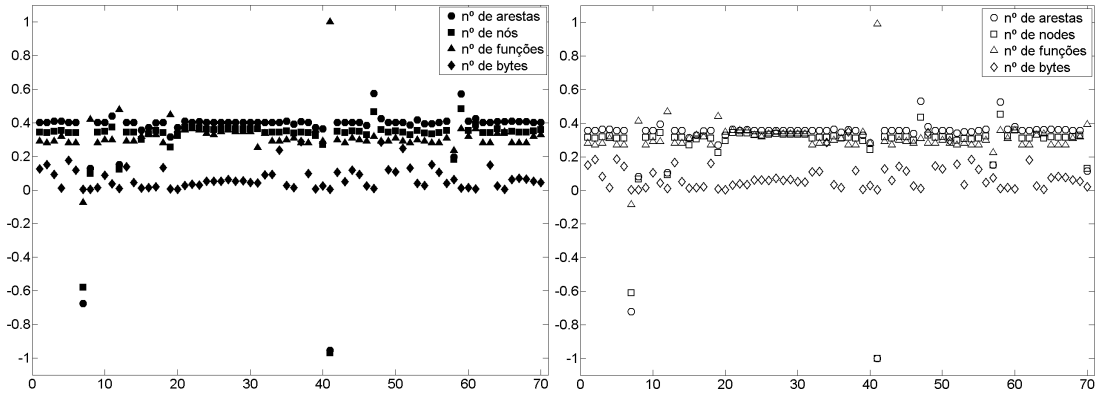
(a) Conjunto verdadeiro. (b) Conjunto falso.

Figura 6. Avaliação da RNA usando quatro parâmetros: número de nós e arestas dos grafos de controle de fluxo, número de funções do grafo de chamadas e tamanho dos códigos.

4.1.2. Avaliação dos programas modificados por códigos maliciosos

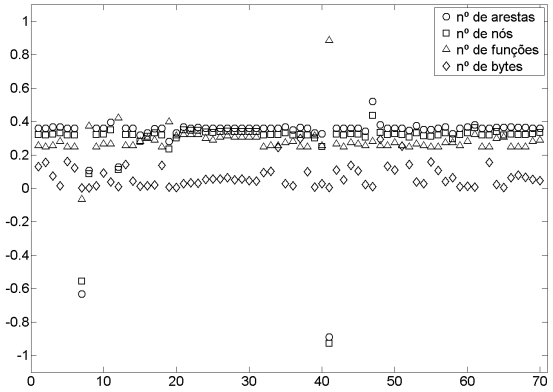
Visto que a maioria dos códigos maliciosos são predominantemente do ambiente Windows, recompilamos os 100 códigos previamente avaliados, entretanto, somente 94 destes foram compilados, devido a restrições de bibliotecas vinculadas ao ambiente Linux. Antes de infectar os códigos por códigos maliciosos, foi necessário estabelecer alguns critérios de segurança, para evitar a disseminação de um código malicioso para todo ambiente de trabalho. Para isso, foi configurada uma estação de trabalho isolada para o processo infecção dos códigos e extração das características. Os parâmetros extraídos dos códigos infectados foram utilizados para a elaboração do conjunto de pares falsos.

A Figura 8 apresenta a avaliação da RNA para programas infectados pelo código malicioso Virus.Win32.Cabanas.a. Os conjuntos de treinamento desta avaliação são exibidos nas Figuras 7(a) e 7(b), que representam os pares correspondentes e os não correspondentes (infectados pelo código malicioso Cabanas), respectivamente. Para o conjunto verdadeiro (veja Figura 8(a)), a RNA forneceu uma taxa de acerto de 19 de 24 (≈79%) e para o conjunto falso (veja Figure 8(b)), os resultado foi de 1

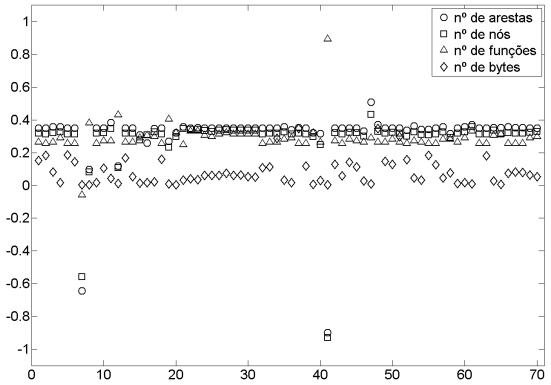


(a) Conjunto verdadeiro.

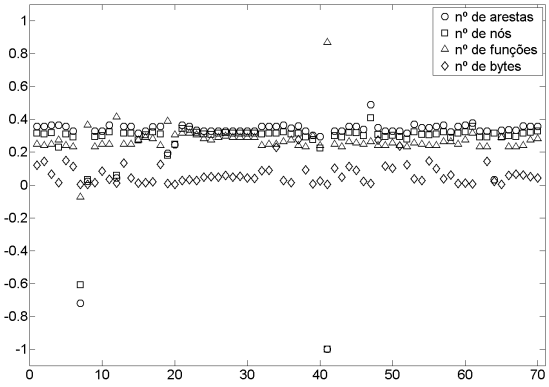
(b) Conjunto falso (Cabanas.A).



(c) Conjunto falso (NGVCK.1003).



(d) Conjunto falso (Qudos.4250).

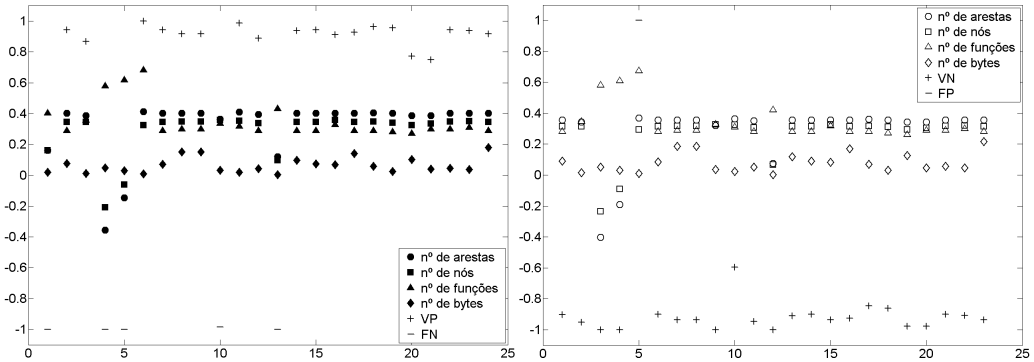


(e) Conjunto falso (Artelad.2173).

Figura 7. Conjunto de treinamento utilizando pares correspondentes e pares falsos criados a partir de programas modificados pelos códigos maliciosos.

de 23 (≈4%) falso positivo.

Para uma avaliação mais aprimorada sobre o impacto de códigos maliciosos no método proposto de rastreabilidade, outros três códigos maliciosos (Virus.Win32.NGVCK.1003, Virus.Win32.Qudos.4250, Virus.Win32.Artelad.2173) foram utilizados. A avaliação da RNA diante de programas infectados pelo código malicioso Virus.Win32.NGVCK.1003 resultou em uma taxa de acerto de 16/24 e de falsos positivos 2/24. Já para os códigos maliciosos Virus.Win32.Qudos.4250 e Virus.Win32.Artelad.2173 a RNA forneceu zero falsos positivos com 8/24 e 9/24 falsos



(a) Conjunto verdadeiro. (b) Conjunto falso.

Figura 8. Avaliação da RNA diante de códigos infectados pelo vírus Win32.Cabanas.a.

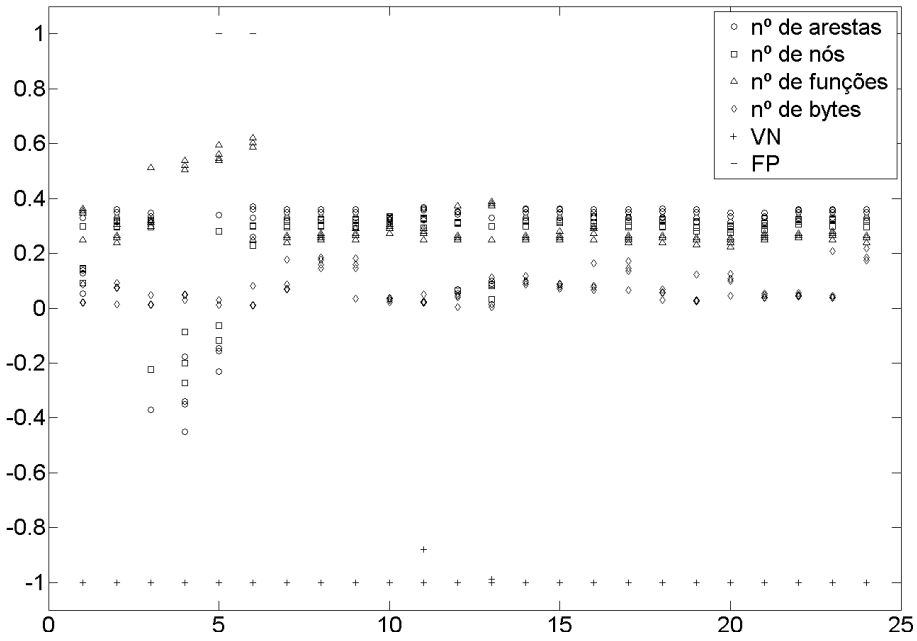


Figura 9. Avaliação da RNA diante de códigos infectados.

negativos, respectivamente. Também avaliamos a RNA agrupando o conjunto verdadeiro com todos conjuntos falsos presentes na Figura 7, na qual foram avaliadas 280 amostras infectadas (70 para cada código malicioso), assim como 70 características das amostras não infectadas. A Figura 9 exibe os resultados da avaliação da RNA, com 4/93 ($\approx 4\%$) de falsos positivos e 9/24 ($\approx 37\%$) de falsos negativos. Apesar dos resultados da última avaliação apresentarem uma taxa relativamente alta de falso negativos, nossa proposta de rastreabilidade é ainda relevante, pois apresenta uma baixa taxa de falsos positivos.

Nós comparamos os resultados de nosso método proposto utilizando-se de redes neurais com os resultados obtidos utilizando-se de uma máquina de vetores suporte, ou *Support Vector Machine* (SVM). A técnica SVM é tipicamente aplicada para construção de classificadores [Men et al. 2008, Angulo et al. 2006]. O método para encontrar o hiperplano de separação foi o de programação quadrática, em que a

máquina de vetores suporte é de margem suave e composta por dois vetores normais. Para o mapeamento dos dados de treinamento para o espaço de classificação foi utilizado uma função linear. Os dados fornecidos para treino da SVM foram os mesmos utilizados na RNA (Figuras 7(a) e 7(b)). A SVM teve que identificar uma divisão no espaço de 4 dimensões, ou seja um hiperplano em 3 dimensões para dividir o espaço de 4-D em dois semi-espaços. O conjunto da simulação exibido na Figura 8 foi classificado de acordo com o hiperplano traçado. A SVM conseguiu identificar 19 dentre os 24 pares de códigos correspondentes, contra 7 pares de 23 de falsos positivos. Observa-se que o método que utiliza SVM obteve uma eficiência considerável na identificação de códigos correspondentes (exatamente como a RNA — 19/24) , porém com um número maior de falsos positivos (7/23 contra 1/23 da taxa de acerto da RNA). Tentamos também reproduzir o método SVM para o conjunto da Figura 7, mas a SVM não convergiu para um hiperplano que separasse os grupos.

5. Conclusões

Este trabalho aborda a questão da rastreabilidade de códigos executáveis. O problema é fundamental para a validação de software, pois dado que a avaliação é normalmente realizada no código fonte, dada a complexidade da mesma ser realizada em código executável, torna-se importante garantir que o processo de compilação não introduza falhas, *backdoors* ou comportamentos indesejados.

No método proposto, são extraídas algumas características do código fonte e do código binário para alimentar um arbitrador não determinístico, que irá decidir se um código binário corresponde a um código fonte previamente avaliado. A inovação da abordagem, aqui apresentada, se estabelece no fato da extração de características revelantes do fluxo lógico do programa. A proposta também aborda o uso de uma RNA para decidir sobre a legitimidade de um código binário com base nestas características. O desempenho da abordagem proposta está bem caracterizado através de uma avaliação experimental que, além de confirmar uma taxa muito baixa de falsos positivos (considerada como um requisito básico), também oferece uma quantidade razoável de falsos negativos.

Poder-se-ia argumentar que a abordagem proposta não funcionaria para detectar modificações no código binário que não alterassem os grafos de chamada e de controle de fluxo, como por exemplo, a mudança de uma única constante. Observa-se, contudo, que tal modificação seria imediatamente notada por testes convencionais realizados em verificação de instrumentos de medição. Voltando ao exemplo da balança, descrita na Seção 1, uma simples duplicação de uma constante no código binário poderia fazer com que a balança apresentasse o dobro do valor correspondente. Tal comportamento seria imediatamente notado por um teste simples. Bastaria utilizar um peso padrão de 1 Kg, por exemplo, na balança e observar o peso indicado. Os tipos de modificações que nossa abordagem propõe tratar são mais sutis. Por exemplo, um fabricante mal-intencionado poderia implementar um *backdoor*, que uma vez ativado, exibiria um comportamento ilegítimo. Esse tipo de modificação dificilmente seria notado por meio de testes funcionais, tais como o descrito anteriormente (teste com o "peso padrão"). No entanto, não seria possível para o fabricante incluir um *backdoor* no software embarcado sem modificar o fluxo lógico

do programa. Isto faz com que essas modificações sejam exatamente as propícias a serem detectadas pela nossa abordagem, que é baseada em características herdadas dos grafos de chamadas e de controle de fluxo.

É importante notar que nossa proposta é genérica, serve para qualquer ambiente de desenvolvimento, apenas exigindo um novo período de treinamento sobre este. Uma questão em aberto na abordagem proposta, e também objeto de pesquisa em curso, refere-se à necessidade de considerar também códigos ofuscados durante a fase de treinamento da rede neural, para melhor compreender as suas implicações. Por exemplo, a ofuscação do fluxo de controle, altera o fluxo de controle da aplicação por reordenação de declarações, procedimentos e laços de repetição. A ofuscação do fluxo de controle se obtém usando predicados opacos e substituindo instruções de transferência de fluxo [Boccardo et al. 2009]. Usando tal ofuscação, algumas propriedades utilizadas em nossa abordagem podem ser alteradas de modo a violar os resultados obtidos em nossa proposta. Finalmente, em linhas de pesquisa futuras, pretendemos combinar a técnica proposta com técnicas baseadas na análise do fluxo de dados de modo a identificar possíveis alterações no conteúdo das variáveis, assim como investigar outras propriedades relevantes que possam ser herdadas dos códigos fonte e binário, tais como invariantes de grafo — crossing numbers, cobertura por ciclos, números cromáticos etc — para aperfeiçoar a correspondência dos mesmos.

Referências

Angulo, C., Ruiz, F., González, L., and Ortega, J. A. (2006). Multi-classification by using tri-class svm. *Neural Processing Letters*, 23(1):89–101.

Asadi, R., Mustapha, N., and Sulaiman, N. (2009). New supervisioned multi layer feed forward neural network model to accelerate classification with high accuracy. *European Journal of Scientific Research.*, 33(1):163–178.

Boccardo, D. R., Lakhotia, A., Manacero Jr, A., and Venable, M. (2009). Adapting call-string approach for x86 obfuscated binaries. In *Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais*.

Burkard, J. (2010). C software. http://people.sc.fsu.edu/ burkardt/. (Último acesso Junho 2010).

Buttle, D. L. (2001). *Verification of Compiled Code*. PhD thesis, University of York, UK.

Ciocoiu, I. B. (2002). Hybrid feedforward neural networks for solving classification problems. *Neural Processing Letters.*, 16(1):81–91.

Flake, H. (2004). Structural comparison of executable objects. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*. IEEE Computer Society.

Hassan, A. E., Jiang, Z. M., and Holt, R. C. (1995). Source versus object code extraction for recovering software architecture. In *WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering*, pages 67–76, Washington, DC, USA. IEEE Computer Society.

Hatton, L. (2005). Estimating source lines of code from object code. In *Windows and Embedded Control Systems*.

Haykin, S. (1998). *Neural Networks: A Comprehensive Foundation*. Prentice Hall.

Hertz, J. A., Krogh, A. S., and Palmer, R. G. (1991). *Introduction to the Theory of Neural Computation*. Addison-Wesley, Redwood City, CA, USA.

IdaPro (2010). Ida pro - disassembler. http://www.hex-rays.com/idapro/. (Último acesso Junho 2010).

Lenic, M., Povalej, P., Kokol, P., and Cardoso, A. I. (2004). Using cellular automata to predict reliability of modules. In *Proceeding (436) Software Engineering and Applications*.

McDonald, J. (2010). Delphi falls prey. http://www.symantec.com/connect/blogs/delphi-falls-prey. (Último acesso Junho 2010).

Men, H., Wu, Y., Gao, Y., Kou, Z., Xu, Z., and Yang, S. (2008). Application of support vector machine to heterotrophic bacteria colony recognition. In *CSSE (1)*, pages 830–833.

Moler, C. B. (1980). MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science.

Moretti, E., Chanteperdrix, G., and Osorio, A. (2001). New algorithms for control-flow graph structuring. In *CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*, page 184, Washington, DC, USA. IEEE Computer Society.

Oh, J. (2009). Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *Blackhat technical Security Conference*.

Oliveira Cruz, A. J. (2010). C software. http://equipe.nce.ufrj.br/adriano /c/exemplos.htm. (Último acesso Junho 2010).

Poznyakoff, S. (2010). Gnu cflow. http://savannah.gnu.org/projects/cflow. (Último acesso Junho 2010).

Quinlan, D. and Panas, T. (2009). Source code and binary analysis of software defects. In *CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research*, pages 1–4, New York, NY, USA. ACM.

Reddy, C. S., Raju, K. V. S. V. N., Kumari, V. V., and Devi, G. L. (2007). Fault-prone module prediction of a web application using artificial neural networks. In *Proceeding (591) Software Engineering and Applications*.

Thompson, K. (1984). Reflections on trusting trust. *Commun. ACM*, 27(8):761–763.

Wang, Z., Pierce, K., and McFarling, S. (2002). Bmat - a binary matching tool for stale profile propagation. In *The Journal of Instruction-Level Parallelism*.

Zeng, H. and Rine, D. (2004). A neural network approach for software defects fix effort estimation. In *IASTED Conf. on Software Engineering and Applications*, pages 513–517.

Zhenga, J. (2007). Predicting software reliability with neural network ensembles. *Expert Systems with Applications*, (36):2116–2122.

Zhenga, J. (2009). A digital image encryption algorithm based on hyper-chaotic cellular neural network. *Journal Fundamenta Informaticae*.