Universidade Federal do Rio de Janeiro

DISSERTAÇÃO DE MESTRADO

TIAGO MONTEIRO DO NASCIMENTO

RASTREABILIDADE ENTRE CÓDIGOS FONTES E CÓDIGOS BINÁRIOS

RIO DE JANEIRO
2011

Instituto de Matemática

Instituto Tércio Pacitti de Aplicações
e Pesquisas Computacionais

TIAGO MONTEIRO DO NASCIMENTO

RASTREABILIDADE ENTRE CÓDIGOS FONTES E CÓDIGOS BINÁRIOS

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Informática, do Instituto de Matemática e do Instituto Tércio Pacitti da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Informática.

Orientador: Luiz Fernando Rust da Costa Carmo

Rio de Janeiro

2011

N244 Nascimento, Tiago Monteiro do.
Rastreabilidade entre códigos fontes e códigos binários. / Tiago Monteiro do Nascimento – Rio de Janeiro: UFRJ, 2011.
116 f.: il.

Orientador: Luiz Fernando Rust da Costa Carmo.
Dissertação (Mestrado em Informática) – Universidade Federal do Rio de Janeiro, Instituto de Matemática, Instituto Tércio Pacitti, 2011.

1. Rastreabilidade de código. 2. Redes Neurais Artificiais. 3. Verificação de Código - Teses. I. Carmo, Luiz Fernando Rust da Costa (Orient.). II. Universidade Federal do Rio de Janeiro. Instituto de Matemática, Instituto Tércio Pacitti. III. Título.

CDD

TIAGO MONTEIRO DO NASCIMENTO

RASTREABILIDADE ENTRE CÓDIGOS FONTES E CÓDIGOS BINÁRIOS

Dissertação submetida ao corpo docente do Programa de Pós-graduação em Informática do Instituto de Matemática e do Instituto Tércio Pacitti, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do grau de Mestre em Informática.

Aprovada em: Rio de Janeiro, 30 de agosto de 2011.

Prof. Dsc. Dr. Luiz Fernando Rust da Costa Carmo - PPGI/UFRJ (Orientador)

Profª. Dsc. Drª. Luci Pirmez - PPGI/UFRJ

Prof. Ph.D.Dr. Adriano Joaquim de Oliveira Cruz - PPGI/UFRJ

Prof. Dsc. Dr. Roberto Willrich – INE/CTC/UFSC

Dr. Davidson Rodrigo Boccardo - Inmetro

Dedico à família e aos amigos

AGRADECIMENTOS

Agradeço a Deus, acima de tudo, pela vida, por iluminar meu caminho, pela minha disposição para lutar por meus objetivos e por ter me dado esta oportunidade.

Agradeço a meus pais, Luis e Marisa, pelos ensinamentos, pela educação e pelos exemplos.

Agradeço a minha família pelo apoio incondicional e pela compreensão durante os vários momentos de ausência em prol da dedicação acadêmica.

Agradeço a meu orientador, Professor Luiz Fernando Rust da Costa Carmo, pela paciência, orientação, oportunidade, pelas discussões e por sua amizade.

Agradeço a meu amigo e colega de trabalho Raphael Carlos dos Santos Machado pelo apoio e motivação em tempos árduos encontrados na realização desta dissertação.

Agradeço a meu colega de trabalho e amigo Davidson Rodrigo Boccardo pelo incentivo, paciência, motivação, auxílio e revisão textual na realização desta dissertação.

Agradeço aos professores do PPGI pelo esforço hercúleo de fornecer conhecimentos e ensinamentos.

Agradeço a todos os meus amigos e companheiros de curso do PPGI que me suportaram em momentos difíceis.

Agradeço à Universidade Federal do Rio de Janeiro pelo curso que estou concluindo, e a todos seus funcionários do Departamento de Ciência da Computação pela dedicação e carinho.

Agradeço ao Instituto Nacional de Metrologia, Qualidade e Tecnologia – Inmetro – pela oportunidade de aperfeiçoamento profissional e pessoal.

Agradeço à Fundação Carlos Chagas Filho de Amparo à Pesquisa do Estado do Rio de Janeiro – FAPERJ – pela bolsa de apoio entre os anos de 2009 e 2010.

Agradeço ao Conselho Nacional de Desenvolvimento Científico e Tecnológico –CNPq – pela bolsa de apoio no ano de 2011.

Agradeço a todos os meus companheiros e amigos da Divisão de Telecomunicação do Inmetro que me suportaram, me apoiaram e ajudaram tanto moralmente quanto tecnicamente. Obrigado pelo tempo que passamos juntos e pelo que me ensinaram nesses últimos anos.

RESUMO

NASCIMENTO, Tiago Monteiro do. **Rastreabilidade entre códigos fontes e códigos binários**. 2011. 116 f. Dissertação (Mestrado em Informática) – Instituto de Matemática, Instituto Técio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2011.

A concepção e a produção de componentes, equipamentos e sistemas para metrologia legal necessitam de bases normativas, metodológicas e de ferramental laboratorial compatíveis com a evolução da complexidade dos medidores. No campo da Metrologia Legal, é fundamental garantir que o software embutido em um medidor corresponde a uma versão que foi previamente aprovada pela Autoridade de Metrologia Legal. Para não se ter desconfiança de fraude ou má intenção de fabricante de dispositivos é preciso que haja uma rastreabilidade do código do software aprovado. Rastreabilidade de códigos refere-se ao mapeamento entre código binário e seu código fonte gerador, inclusive linguagens de programação de alto nível e de baixo nível. Neste trabalho, uma nova abordagem é proposta para correlacionar os códigos fontes e códigos binários usando redes neurais artificiais. Nossa abordagem correlaciona o código fonte com o código binário alimentando a rede neural com propriedades extraídas dos respectivos códigos. Qualquer incidência de falsos positivos é uma questão crítica para fins de avaliação do software, dado que não seria visto como um programa problemático. Nossa avaliação, usando exemplos de código real, mostra uma correspondência de 90% a 100% na rastreabilidade de códigos não-correspondentes e uma taxa em torno de 63% de códigos correspondentes.

Palavras-chave: Rastreabilidade de código. Redes Neurais Artificiais. Verificação de Código. Código Fonte. Código Binário. Metrologia Legal.

ABSTRACT

NASCIMENTO, Tiago Monteiro do. **Rastreabilidade entre códigos fontes e códigos binários**. 2011. 116 f. Dissertação (Mestrado em Informática) – Instituto de Matemática, Instituto Técio Pacitti, Universidade Federal do Rio de Janeiro, Rio de Janeiro, 2011.

The capacity to design and produce components, equipments and systems for legal metrology requires regulations, methods and compatibles laboratory tools according to the evolution of meters complexity. In the field of Legal Metrology, it is critical to guarantee that the software embedded in a meter corresponds to a version that was previously approved by the Legal Metrology Authority. Traceability of codes refers to the mapping between equivalent codes written in different languages – including high-level and low-level programming languages. In this dissertation, 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 a critical issue for software evaluation, since it would not be seen as a problematic program. Our evaluation using examples of real code shows a correlation of 90% to 100% of false identification for the traceability of binary code with a rate around 63% of true positives.

Keywords: Code traceability. Artificial Neural Network. Code verification. Source Code. Binary Code. Legal Metrology.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

FN - Falsos Negativos

FP - Falsos Positivos

Inmetro - Instituto Nacional de Metrologia, Qualidade e Tecnologia

kWh - quilowatt-hora

RNA - Rede Neural Artificial

SVM - *Support Vector Machine*

VN - Verdadeiros Negativos

VP - Verdadeiros Positivos

SUMÁRIO

1 INTRODUÇÃO

Este capítulo apresenta uma introdução do trabalho de pesquisa realizado nesta dissertação, abordando a caracterização do problema, sua contextualização, os trabalhos relacionados, os objetivos e a estruturação deste documento.

1.1 CARACTERIZAÇÃO DO PROBLEMA

Nos dias atuais, há um aumento no número de dispositivos que utilizam softwares embarcados. Estes softwares agregam um conjunto de desafios completamente novos, tais como: abordagens de validação, novas fontes de erros e imprecisões e falhas de segurança. No contexto de dispositivos de medição, um risco crítico associado à presença de software nestes dispositivos 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 ou aumentá-la para reduzir o imposto em relações comerciais. Outro caso seria o do proprietário de uma balança, o que pode modificar o software para que as novas medições (erradas) possam beneficiá-lo.

Atualmente, a maioria dos processos de medição envolve uma ou mais etapas de processamento dos dados originados pelo sensor que está, efetivamente, em contato com o evento físico mensurado. Naturalmente, a única forma de garantir o funcionamento correto de um dispositivo de medição é a realização de alguma forma de verificação externa: uma entidade não-comprometida com vendedor ou comprador que avalia as funcionalidades do dispositivo e ofereça algum tipo de certificação de seu bom funcionamento.

Em diversas áreas, os dispositivos envolvidos em relações comerciais devem ser submetidos a algum tipo de controle externo. Uma das áreas em questão – que motivou o presente trabalho – é a área controlada pela Metrologia Legal. Tipicamente, as entidades que realizam o controle e avaliação destes dispositivos, são do governo ou organizações sem fins lucrativos. O exemplo da balança, acima, é um exemplo típico de relação comercial inserida no domínio da Metrologia Legal. A Metrologia Legal existe para agregar confiança às transações entre o vendedor e o comprador, assegurando que a balança exiba corretamente o peso do produto a ser comercializado.

A evolução tecnológica em metrologia legal tem sido fortemente baseada em medidores eletrônicos, compostos de computadores dedicados (usualmente referenciados

como micro-controladores) com programas específicos desenvolvidos para as aplicações fins. Essa atualização na tecnologia da medição traz consigo uma maior exposição a fraudes e comportamentos errôneos não intencionais nos dispositivos de medição, aumentando os desafios da metrologia legal.

Tradicionalmente, os processos para aprovação de modelo de um sistema de medição envolvem várias etapas de testes metrológicos do dispositivo sob avaliação. Tais testes verificam se o dispositivo funciona corretamente em várias condições diferentes – por exemplo, em condições distintas de umidade e temperatura. Com o advento dos medidores com software embarcado, o processo de avaliação deve envolver, adicionalmente, a validação destes softwares. A entidade avaliadora deve garantir, por exemplo, que o software embarcado na balança não contenha algum *backdoor* que ative – sob o comando do proprietário/usuário do instrumento – e que ative uma função espúria que diminua ou aumente o peso exibido.

1.2 CONTEXTUALIZAÇÃO

Um exemplo de sistema de medição controlado por software é o sistema distribuído de medição de energia elétrica atualmente em uso no Brasil. Visando uma eliminação ou redução das fraudes na medição, é feito uma mudança da localização física do medidor, de dentro da casa para o topo do poste; e um rearranjo da ordem das linhas de média e baixa tensão nos postes: média tensão para baixo, e baixa tensão para cima. Com os medidores acomodados em um gabinete acima das linhas de média tensão, dificulta-se muito qualquer tentativa de captura da energia antes dos medidores. Este rearranjo traz consigo a necessidade da adoção de um mostrador remoto na casa do consumidor para indicação do consumo. Este novo sistema de medição é visto na figura 1. Basicamente, as medidas são realizadas e transmitidas para o consumidor a cada incremento de kWh, em seu respectivo mostrador.

Esse sistema de medição, com software embarcado, apresenta novas fontes de vulnerabilidades, que podem se transformar em ameaças, tanto para o consumidor, como para as concessionárias.

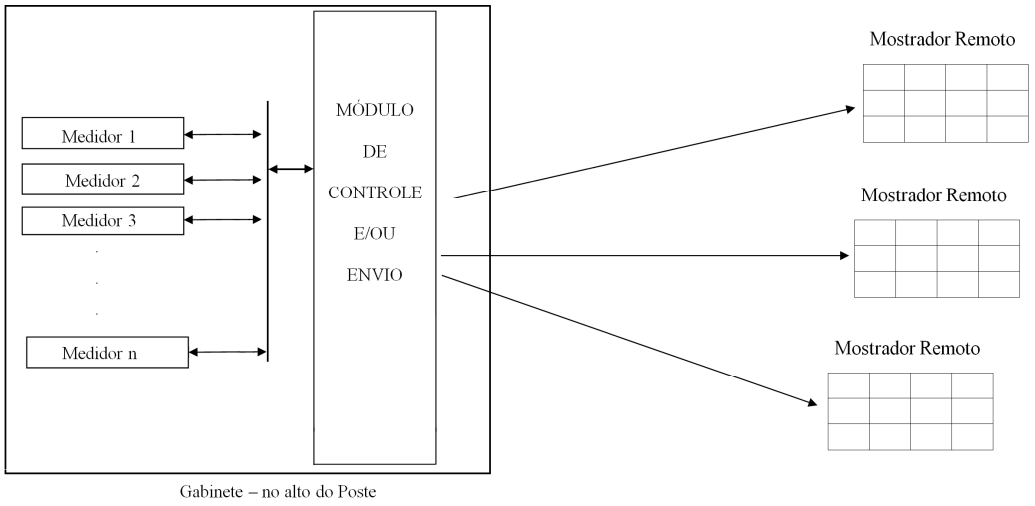


Figura 1: Exemplo de arquitetura de sistema de medição.
Fonte: o Autor (2011).

O Inmetro, como entidade de avaliação, através da portaria específica para este sistema de medição (INMETRO, 2009), detalhou os requisitos de software para este sistema. Estes requisitos estão listados a seguir:

- identificação de software
- influência da interface de usuário no sistema distribuído de medição;
- influência da interface de comunicação no sistema distribuído de medição;
- proteção contra mudanças acidentais/não-intencionais;
- proteção contra mudanças intencionais;
- proteção de parâmetros.

De acordo com Thompson (THOMPSON, 1984), a única forma de avaliação destes requisitos de um software é através da inspeção de seu binário. Em sistemas grandes e complexos, entretanto, a verificação binária pode exigir um esforço elevado para rastrear completamente as manipulações de variáveis e realizar análise de vulnerabilidades. Logo, uma abordagem usual é realizar essa verificação sobre o código fonte. Desta forma, o Inmetro estabeleceu como exigência no processo de aprovação de modelo a disponibilização do código fonte do software legalmente relevante, procedimentos inerentes a cadeia de medição e exibição do resultado (INMETRO, 2009). No entanto, a verificação do código fonte não é suficiente para dar qualquer garantia sobre o comportamento do software embarcado no

dispositivo relacionado. Podem ocorrer inserções no código binário ou modificações do mesmo no processo de compilação, porque o compilador pode estar modificado ou ser espúrio (MCDONALD, 2009). Para certificar que o código binário, que está em execução no dispositivo, correspondente ao código fonte previamente avaliado, é preciso garantir que esse código binário seja, de fato, gerado a partir do código fonte aprovado. As duas abordagens de validação são mostradas na figura 2.

Figura 2: Aborgadens de validação.
Fonte: Nascimento, T. et al. (2010)

Rastreabilidade entre códigos é o processo que estabelece a correspondência entre o código fonte avaliado e o código binário embarcado no dispositivo. Isto é, além da aprovação do código fonte da versão do software do sistema de medição em processo de avaliação de modelo, é necessário verificar se o código binário – que estará, de fato, em execução no dispositivo – corresponde a este código fonte.

No trabalho aqui apresentado, é proposta uma estratégia para verificar se um código fonte e um código binário apresentam o mesmo comportamento algorítmico. Este trabalho apresenta uma nova abordagem para a realização de rastreabilidade usando arbitradores lineares e não-lineares (hiperplanos e redes neurais artificiais). Mais especificamente, são extraídas propriedades dos códigos fontes e binários, tais como o número de caminhos de fluxo de execução, número de blocos básicos dos grafos de fluxo de controle e número de funções do grafo de chamadas.

1.3 TRABALHOS RELACIONADOS

Não há muitas contribuições relacionadas com a rastreabilidade de códigos na literatura. No entanto, existem alguns trabalhos de verificação e análise de códigos fonte e

binário, extrações de propriedades de códigos, bem como trabalhos relativos à semelhança entre binários que contribuíram na realização da abordagem desta dissertação.

Um trabalho que verifica propriedades de códigos é o trabalho de Brumley e Newsome (2006). É um trabalho que já expõe as fraquezas em realizar apenas análise em códigos fontes. Neste trabalho são feitas *disassembly* de códigos binários e utilizando análises dos códigos *assembly*, verificam-se registros de dados também como os pseudônimos, como nomes de variáveis e funções, do código binário, para a proteção de memória. O trabalho nos auxiliou na realização de identificação e de tratamento das propriedades integrantes em códigos binários, mas ainda era necessário utilizar propriedades referentes a códigos fontes.

No trabalho de Subramanian e Cook (1996), já é discutida uma preocupação em compiladores, se estes são confiáveis, e nas verificações de códigos fontes e códigos binários, e assim, a proposta deste artigo é realizar uma verificação automática entre um código objeto do microcontrolador MC68020 e códigos fontes em linguagem C, porém a mesma é específica para este micro controlador. Nesta dissertação tentamos desenvolver uma proposta genérica, independente de compiladores ou arquiteturas.

No trabalho de Quinlan e Panas (2009) é proposto um arcabouço para verificação de defeitos de software (*bugs*), tanto no código binário quanto no código fonte. Eles realizam verificação em códigos binários, identificando acessos indevidos em posições de memória ou transbordamento de dados (*buffers overflows*) e rastreiam as funções nos códigos fontes que causariam tais problemas. Entretanto, eles não comparam fonte e binário diretamente, como propõe esta dissertação.

Hassan, Jiang e Holt (2005) fazem comparações entre os dois métodos, análise em códigos binários e análise em códigos fontes, e após consideram a necessidade de se utilizar os dois métodos em conjunto. Eles observaram que a arquitetura de alguns programas está intrinsecamente relacionada com seu código fonte e código binário. Eles usaram dois tipos de extratores de propriedades de códigos (um para binário e outro para fonte) para recuperar arquitetura de grandes sistemas em código aberto, como Openssh, Postgres ou o próprio kernel do Linux-2.6.1. Para os códigos binários, eles utilizaram o LDX, um extrator de dependências de módulo e de funções (grafo de dependências) baseado no ligador *ld*,que é utilizado pelo compilador GNU GCC (GNU, 2011), e para códigos fontes foi utilizado um extrator que recupera informações, tais como *#if #else* e *#ifdef*, de códigos fontes em linguagem C chamado, CTAGX. Depois das extrações, é realizado um comparativo dos resultados obtidos de forma a inferir a arquitetura de software quanto à relação de

correspondência dos códigos. O trabalho foi importante para esta dissertação, pois apresenta as similaridades que podem existir entre binários, com seus grafos de dependência, e códigos fontes, mas o trabalho foca em dar relevância para utilização das ferramentas de extração e não dá importância em se ter uma rastreabilidade de códigos.

Hatton (2005) investigou a relação entre um código binário e código fonte, através de densidade de defeitos, métrica que verifica número de defeitos por linhas de código. Tais defeitos são detectados no software ou componente de software durante um período definido de desenvolvimento ou operação. Então, o trabalho usa os defeitos que ocorrem em códigos binários em execução, para estimar a quantidade de número de linhas que teria o código fonte, respectivo àquele código binário. Portanto, o trabalho só apresenta uma rastreabilidade para estimativas do número de linhas do código fonte.

Nenhum desses artigos citados acima foca explicitamente em rastreabilidade de um código binário. Entretanto a tese de doutorado de Wahab (1998) apresenta uma rastreabilidade entre códigos que se aprofunda em aspectos teóricos. A tese apresenta uma definição de uma linguagem formal para representação das propriedades tanto de um código fonte como do respectivo código binário, habilitando uma posterior comparação entre as respectivas especificações. Este trabalho é importante, pois apresenta a existência de propriedades encontradas em códigos binários que são herdadas de seus códigos fontes respectivos. Em outra tese de doutorado, a de Buttle (2001), há também uma preocupação em rastreabilidade. Nesta tese, o autor utiliza a estrutura lógica de programa (grafo de fluxo de controle) do código binário para coincidir com a estrutura lógica de programa do código fonte e assim realizar a rastreabilidade.

Com isso, pudemos realizar pesquisas de extração de propriedades em códigos binários e em códigos fontes. Em nossa proposta também utilizamos as propriedades do fluxo do programa, como em Buttle (2001), no entanto, apresentamos também outras relações que podem coexistir entre fonte e códigos binário, com a finalidade de obter uma melhor correspondência.

Outros trabalhos pesquisados como os de Wang, Pierce e McFarling (2002), de Flake (2004) e Oh (2009), versam sobre diferença de binários. Estes trabalhos têm o intuito de verificar versões de códigos maliciosos, para analisar variantes de *malwares* em linguagens de alto nível (WANG; PIERCE; MCFARLING, 2002) ou para realizar comparações em atualizações de segurança (FLAKE, 2004). A maioria destas abordagens utiliza comparação de grafo de correspondência para verificar os binários. Um bom resumo desses trabalhos pode

ser encontrado em Oh (2009), que usa algoritmos baseados em equivalência de grafos para comparar versões de códigos maliciosos. Técnicas utilizadas nas pesquisas de diferença de binários, como comparação de grafos de fluxo de controle, podem ser aplicadas de modo análogo à rastreabilidade de códigos.

A maior contribuição dos trabalhos relacionados anteriormente é relativa à definição de conjunto de propriedades de um código, e as respectivas técnicas de extração destas propriedades passíveis de uso no problema de rastreabilidade.

1.4 OBJETIVOS

Esta dissertação tem como objetivo geral apresentar soluções que permitam estabelecer maiores subsídios quanto à origem de um determinado código binário e com isso gerar maior confiança quanto à relação de criação de um código binário vindo de seu código fonte respectivo.

O principal objetivo específico deste trabalho é gerar um método de avaliação da rastreabilidade de um código binário a partir de seu código fonte. Inicialmente é necessário selecionar um conjunto de propriedades do código fonte e do código binário que permita inferir uma relação de unicidade entre os dois códigos. Estas propriedades devem ser tais que permitam uma fácil extração nos dois códigos, gerando resultados singulares, ou seja, qualquer que seja a ferramenta utilizada para extração o valor da propriedade deverá ser único.

Além da evidente necessidade da propriedade estar presente tanto no código fonte quanto no binário, é necessário avaliar se esta traz algum benefício no processo de comparação (rastreabilidade) entre os dois códigos.

Em seguida devem-se avaliar as melhores abordagens para composição do processo de arbitração para rastreabilidade. Nesta etapa o principal requisito é a operacionalidade da solução. Busca-se uma abordagem que seja de fácil implementação, com tempo de execução compatível às exigências do processo de avaliação do sistema de medição, e que gere resultados conclusivos a partir do conjunto de propriedades disponível.

A abordagem selecionada deve ser validada experimentalmente através de exemplos concretos e criteriosamente avaliada quanto as suas limitações e potencialidades. Para isso é necessário o desenvolvimento de um processo de caracterização de refinamento da abordagem, ou seja, quão capaz é a identificação de sutis alterações.

1.5 ESTRUTURA DA DISSERTAÇÃO

Esta dissertação está organizada em cinco capítulos, sendo a introdução o primeiro capítulo.

O capítulo 2 faz uma apresentação dos conceitos teóricos necessários para compreensão deste trabalho, expondo as principais referências na literatura sobre o tema que serviram de base para o seu desenvolvimento. Em resumo são descritos algumas propriedades presentes nos códigos fontes e binários que foram avaliados neste trabalho bem como as definições básicas dos possíveis arbitradores.

O capítulo 3 apresenta uma proposta para rastreabilidade de códigos, utilizando-se de redes neurais artificiais para inferir sobre a correspondência entre um código fonte e binário a partir de propriedades obtidas através da análise de programas. As propriedades são: tamanho em bytes; número de caminhos no fluxo de execução e blocos básicos extraídos dos grafos de fluxo de controle individuais (um para cada função); e número de funções obtidas do grafo de chamadas. Após a extração das propriedades, procurou-se identificar a relevância destas para com o método de rastreabilidade através da análise do ganho de aderência fonte/binário para cada propriedade.

O capítulo 4 descreve diferentes cenários de avaliação da proposta através da criação de códigos binários adulterados a partir de códigos maliciosos. Para cada cenário, realizou-se, uma análise da eficiência do método proposto com um arbitrador linear. Também, neste capítulo, é apresentado um refinamento da proposta a partir da avaliação da sensibilidade do método através da contenção seletiva das alterações nos códigos binários.

O capítulo 5 descreve as considerações, contribuições e sugestões para continuidade desta pesquisa.

2 CONCEITOS BÁSICOS

2.1 INTRODUÇÃO

Este capítulo apresenta os conceitos básicos de propriedades encontradas nos códigos fontes e nos códigos binários a fim de melhorar a compreensão deste trabalho. Uma revisão das principais referências da literatura sobre análise de programas também é apresentada. Esta serve de base para escolher propriedades significativas entre um código fonte e binário.

Em seguida há uma exposição dos conceitos básicos sobre métodos lineares e redes neurais, que foram os arbitradores testados para identificar o grau de semelhança entre os códigos fontes e códigos binários.

2.2 ANÁLISE DE PROGRAMAS

Para se estudar e obter as propriedades de códigos foi preciso realizar análise de programas. Uma forma de analisar programas envolve a utilização de técnicas estáticas para calcular informações aproximadas confiáveis sobre o comportamento dinâmico (execução) dos programas. Aplicações incluem: compiladores (para otimização de código), validação de software para detecção de erros ou brechas de segurança, e transformações entre representação dos dados (NIELSON, F; NIELSON, H; HANKIN, 2005).

A maioria das técnicas usadas para análise de programas é dependente da construção dos grafos de fluxo de controle individualizado para cada função e do grafo de chamadas. A lógica do programa, seja ele na linguagem fonte ou binária, pode ser caracterizada por estes grafos. Um grafo de fluxo de controle é uma representação que usa notação de grafo direcionado para descrever todos os caminhos que podem ser percorridos por um programa durante sua execução. Um grafo de chamadas é um multi-grafo direcionado que representa as relações entre chamadas de funções ou sub-rotinas em um programa de computador.

Em um grafo de fluxo de controle, cada nó representa um bloco básico, isto é, uma região de código sequencial sem qualquer salto de execução. E, cada aresta representa o fluxo de informação entre os blocos básicos baseados em saltos condicionais e incondicionais.

Sucintamente, o algoritmo para criação de um grafo de fluxo de controle envolve as seguintes etapas:

- seleção dos blocos líderes: bloco que contem a primeira instrução do programa, ou que seja destino de um salto ou que seja subsequente a um salto condicional.
- delimitação dos blocos básicos, de modo que, cada bloco básico consista de instruções que se originam de um líder até o próximo líder, porém não o incluindo no mesmo bloco.
- união dos blocos básicos através da inserção de arestas, em que um bloco A é ligado com um bloco B (A → B), se houver um salto de A para B, ou B ser executado após o fluxo de execução passar por A.

As figuras 3 e 4 apresentam um código fonte e seu binário correspondente, respectivamente. As figuras 5(a) e 5(b) apresentam os grafos de fluxo de controle destes códigos.

```
int modexp (int y, int x[], int w, int n){
  int R, L;
  int k=0;
  int s=1;
  while (k<w) {
    if (x[k] == 1)
      R = (s*y) % n;
    else
      R=s;
    s = R*R % n;
    L = R;
    k++;
  }
  return L;
}
```

Figura 3: Exemplo de código fonte.
Fonte: o Autor (2011).

```
modexp:
push    ebp
mov     ebp, esp
sub     esp, 14h
mov     [ebp+var_C], 0
mov     [ebp+var_10], 1
loc_4012A4:
mov     eax, [ebp+var_C]
cmp     eax, [ebp+arg_8]
jge     short loc_401308
mov     eax, [ebp+var_C]
lea     edx, ds:0[eax*4]
mov     eax, [ebp+arg_4]
cmp     dword ptr [edx+eax], 1
jnz     short loc_4012DB
mov     eax, [ebp+var_10]
mov     edx, eax
imul    edx, [ebp+arg_0]
lea     eax, [ebp+arg_C]
mov     [ebp+var_14], eax
mov     eax, edx
mov     ecx, [ebp+var_14]
cdq
idiv    dword ptr [ecx]
mov     [ebp+var_4], edx
jmp     short loc_4012E1
loc_4012DB:
mov     eax, [ebp+var_10]
mov     [ebp+var_4], eax
loc_4012E1:
mov     eax, [ebp+var_4]
mov     edx, eax
imul    edx, [ebp+var_4]
lea     eax, [ebp+arg_C]
mov     [ebp+var_14], eax
mov     eax, edx
mov     ecx, [ebp+var_14]
cdq
idiv    dword ptr [ecx]
mov     [ebp+var_10], edx
mov     eax, [ebp+var_4]
mov     [ebp+var_8], eax
lea     eax, [ebp+var_C]
inc     dword ptr [eax]
jmp     short loc_4012A4
loc_401308:
mov     eax, [ebp+var_8]
leave
retn
endp
```

Figura 4: Código binário referente ao código fonte da figura 3.
Fonte: o Autor (2011).

Pode ser observado a partir da análise destes grafos que o fluxo de execução encontrado no código fonte, isto é, a sequencialidade dos comandos representada pelo fluxo de controle, é certamente mantido no binário. Algoritmos preliminares para estruturar e construir o grafo de fluxo de controle são necessários e, para isso, existem diferentes algoritmos na literatura (MORETTI; CHANTEPERDRIX; OSORIO, 2001) sobre o tema em questão.

Já em um grafo de chamadas cada nó representa uma função e cada aresta que existir entre uma função ‘f’ e uma função ‘g’ indica que ‘f’ chama ‘g’. Um ciclo neste grafo indica uma chamada recursiva.

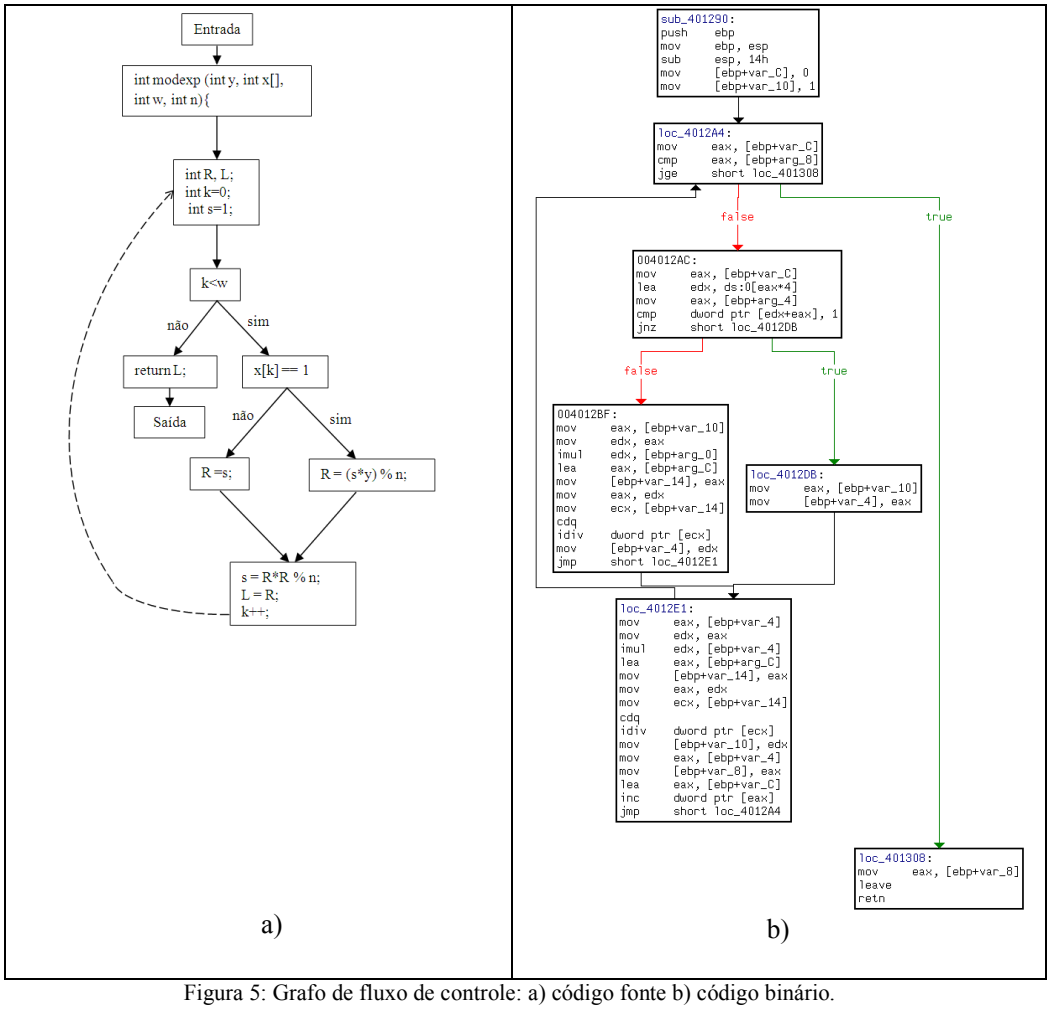
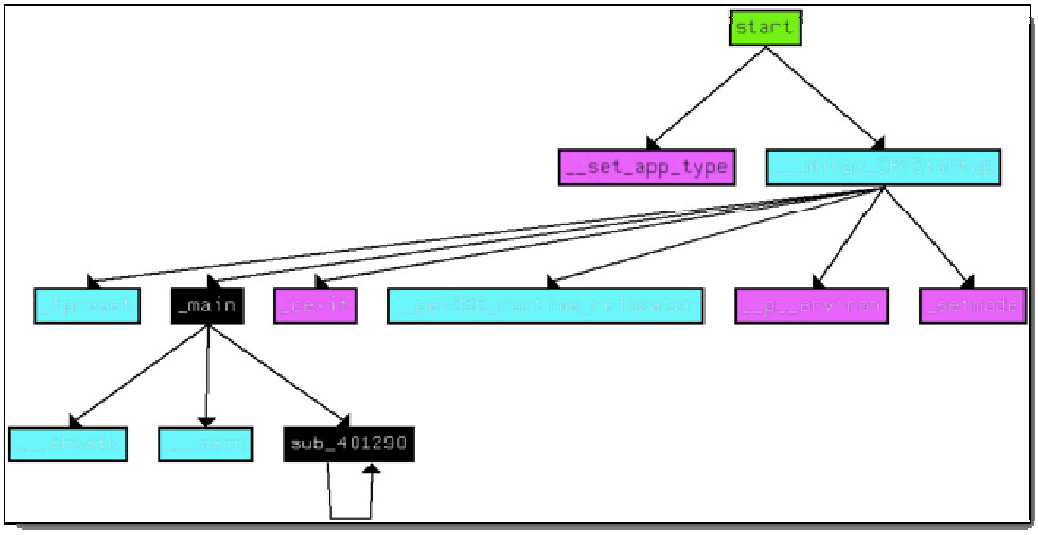


Figura 5: Grafo de fluxo de controle: a) código fonte b) código binário.
Fonte: o Autor (2011).

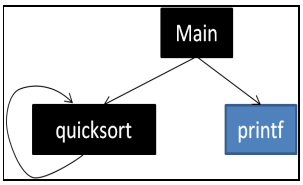
Uma simples aplicação do grafo de chamadas visa encontrar funções que nunca são chamadas. Tal grafo também pode ser usado como base para análises posteriores, tais como a análise que acompanha o fluxo de valores entre funções. Métodos para construção destes grafos podem ser encontrados no trabalho de Murphy (et al., 1998).

Na figura 6, são apresentados dois grafos de chamadas. O grafo de chamadas do programa binário na figura 6(a) e grafo de chamadas do código fonte na figura 6(b).

Conforme será visto no capítulo 3, as propriedades utilizadas na abordagem proposta nesse trabalho são derivadas a partir da lógica do fluxo de execução do programa, ou seja, obtidas através dos grafos supramencionados. Basicamente as propriedades são: número de blocos básicos, número de caminhos de fluxo de execução e número de funções.



a)



b)

Figura 6: Exemplo de grafo de chamadas de função: a) código binário b) código fonte.
Fonte: o Autor (2011).

Com estas propriedades apresentadas são discutidos na próxima seção alguns arbitradores e suas peculiaridades para relacionar a correspondência de um código binário com seu respectivo código fonte.

2.3 ARBITRADORES

Classificação de dados é uma tarefa comum na aprendizagem de máquina. Dado pontos pertencentes a uma ou duas classes, o objetivo é decidir a qual classe um ponto de dados pertence. A seguir discutiremos duas classes de arbitradores: linear e não-linear.

2.3.1 Arbitrador Linear

Uma das técnicas linear e determinística para classificação de dados é a *Support Vector Machine*, máquina de suporte vetorial ou simplesmente SVM. SVM corresponde a um conjunto de algoritmos de aprendizagem supervisionado, desenvolvidos por Vladimir Vapnik e sua equipe nos laboratórios da AT&T (CORTES; VAPNIK, 1995).

Dado um conjunto de pontos em que cada um desses pontos possui sua classe definida, uma SVM constrói, a partir das características desses pontos, um modelo capaz de dizer a qual classe pertence um novo ponto dependendo das propriedades desse novo ponto.

Essa técnica de construção de vetor é relacionada a problemas de classificação e regressão e é baseada na separação de um conjunto de amostras em classes, por exemplo, classe verdadeira e classe falsa. Mais formalmente, uma SVM constrói um hiperplano ou um conjunto de hiperplanos em um espaço multidimensional, que separa de forma ótima as amostras, ou pontos, de uma classe de amostras e que estes hiperplanos construídos pela SVM sirvam como um modelo de definição de classe de novas amostras. A característica fundamental da SVM reside nesse conceito de separação ótima. Os algoritmos usados pela SVM buscam o hiperplano que tenha a maior distância, ou margem, dos pontos que estejam mais próximos desse hiperplano. Por isso as SVM são conhecidas também como classificadores de margem máxima, e o vetor formado pelos pontos mais próximos do hiperplano é chamado de vetor suporte. Na figura 7 são visualizados hiperplanos diferentes criados por SVM, para a classificação dos conjuntos brancos e pretos.

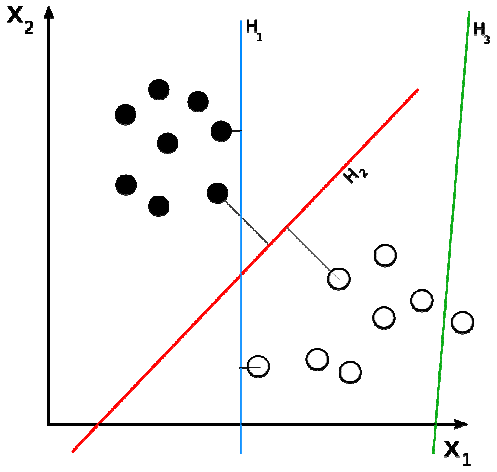


Figura 7: Exemplos de hiperplanos gerados por SVM.
Fonte: o Autor (2011).

Pode-se observar que o hiperplano H3 não separa as duas classes devidamente. O hiperplano H1 consegue fazer a separação das duas classes, porém com uma pequena margem. E por fim o hiperplano H2 faz a separação com margem máxima. Dentre estes hiperplanos desenhados, o H2 é o único que representa o classificador de margem máxima.

Após verificação e avaliação da SVM, idealizamos outro método linear e determinístico para classificar os dados, consistindo em traçar diversos planos para dividir duas classes de conjuntos. Porém, diferente do SVM, cada plano é constituído por pontos que estão no espaço, e não por vetores suporte. Este algoritmo foi denominado de Hiperplano de Separação e é apresentado na figura 8.

Hiperplano de Separação $(p_1, ..., p_n, k_1, ..., k_n, r)$

ENTRADA: conjunto P de vetores em 4 dimensões $p_1, ..., p_n \in \mathbb{R}^4$,
"valores correspondentes" $k_1, ..., k_n \in \{\text{VERDADEIRO}, \text{FALSO}\}$,
taxa máxima de falso positivo $r \in [0, 1/2)$.

ALGORITMO:

1. $FP := r$; $VP := 0$;
2. **para cada** $(p_1, p_2, p_3, p_4) \in P^4$
2.1. Seja H o hiperplano que passa por p_1, p_2, p_3 e p_4;
2.2. Seja u um vetor ortogonal a H;
2.3. $V^+(H) := |\{p \in P | \langle p; u\rangle \geq 0 \text{ e } k_p = \text{VERDADEIRO}\}|$;
2.4. $V^-(H) := |\{p \in P | \langle p; u\rangle \leq 0 \text{ e } k_p = \text{VERDADEIRO}\}|$;
2.5. $F^+(H) := |\{p \in P | \langle p; u\rangle \geq 0 \text{ e } k_p = \text{FALSO}\}|$;
2.6. $F^-(H) := |\{p \in P | \langle p; u\rangle \leq 0 \text{ e } k_p = \text{FALSO}\}|$;
2.7. **se**$(F^+(H) > F^-(H))$ **então**
2.7.1 $sinal_F :=$ '+'; $sinal_V :=$ '−';
2.8. **senão**
2.8.1 $sinal_F :=$ '−'; $sinal_V :=$ '+';
2.9. $VP(H) := V^{sinal_V}(H)/|\{p \in P | k_p = \text{VERDADEIRO}\}|$;
2.10. $FP(H) := F^{sinal_V}(H)/|\{p \in P | k_p = \text{FALSO}\}|$;
2.11. **se**$(FP(H) \leq r \text{ e } VP(H) > VP)$ **então**
2.11.1 $FP := FP(H)$; $VP := VP(H)$;

Figura 8: Algoritmo do hiperplano de separação.
Fonte: o Autor (2011).

Este algoritmo tem como entrada um conjunto '*P*' de 4 dimensões, representados por 'p_1, .., p_n'. Neste espaço de 4 dimensões é preciso 4 pontos para formar um hiperplano (H). É observado que o algoritmo acima funciona sob o pressuposto de que três pontos do

conjunto P são não-colineares e tal suposição, em geometria computacional, é chamada de "posição geral" (GOODMAN; O'ROURKE, 2004). Como parâmetros de entrada possuímos todos os elementos '$p_1, .., p_n$', seus valores 'k' correspondentes, VERDADEIRO e FALSO, e a taxa máxima de falso positivo 'r', que são valores entre 0 e ½ inclusive, ou seja, no exemplo apresentado seria aceitável um erro de até 50% dos elementos não-correspondentes.

O algoritmo, ao desenhar cada plano na linha 2.1 do algoritmo acima, utiliza produto vetorial de cada ponto com a normal do plano para separar em dois conjuntos os programas a ele apresentados. A normal é dada por '*u'*, contido na linha 2.2, e a multiplicação e separação, que é o cerne do algoritmo, se encontram nas linhas 2.3 até 2.6 do algoritmo. Por serem conhecidos previamente os valores 'k_p' de cada ponto, presentes nas linhas 2.7 e 2.8, é decidido que o lado onde esteja a maior quantidade de elementos falsos é o lado correto dos não-correspondentes.

Para conclusão do algoritmo, é realizado o teste de verificação da taxa de falsos positivos (FP), ou seja, os não-correspondentes que foram identificados como correspondentes, bem como dos verdadeiros positivos (VP). Na linha 2.11 verifica-se se a taxa de FP é menor ou igual que a taxa que procuramos '*r'* e se a taxa de VP seria melhor do que a última encontrada. O algoritmo com '*n'* pontos e '*d'* dimensões, possui complexidade $\Theta(n^d)$ e demanda custo computacional e tempo elevados devido à construção de todos os hiperplanos, à verificação de cada ponto em relação ao hiperplano e à escolha dos melhores hiperplanos.

2.3.2 Arbitrador Não-Linear

Esta seção apresenta uma técnica não-determinística e não-linear para classificação de dados, redes neurais artificiais, ou apenas redes neurais, ou ainda RNA. Estas podem ser utilizadas para computar com precisão as similaridades de informações presentes e identificáveis nos elementos a serem averiguados.

Redes neurais, com sua habilidade notável de extrair significado de dados complexos ou imprecisos, podem ser usadas para detectar tendências e extrair padrões que são demasiadamente complexos para serem notados pelo ser humano ou outras técnicas computacionais. Uma rede neural treinada pode ser pensada como um especialista ou perito na categoria de informação que foi fornecida para análise. Uma rede neural é representada por um conjunto de interconexão de neurônios artificiais, inspirados no sistema nervoso

biológico. Estes são dinamicamente acoplados e manifestam através de seu comportamento coletivo, alguns recursos computacionais úteis. Existem muitos tipos de redes neurais cada qual aplicada a um conjunto grande de problemas diferentes, tais como: classificação, reconhecimento, previsão entre outros. A tabela 1 apresenta modelos de redes neurais encontrados na literatura mais apropriados para os problemas mencionados acima.

Entrada	Relação	Saída	Caso	Rede
Vetor binário	N:1	Vetor binário	Reconhecimento de Símbolo (Sequência de padrão)	Elman
Matriz (grafo)	1:N	Matriz (Var de função)	Mapeamento de enlaces em rede (otimização)	OPTI-net
Matriz (grafo)	N:N	Matriz (grafo)	Grafo direcionado	Diversas criadas
Par de Matrizes	N:N	Par de Matrizes	Achar todos os autovalores.	Não especifica
Par	1:1	Par	Validar a rede	FAMR (mapa)

Tabela 1: Modelos de Redes Neurais
Fonte: o Autor (2011).

O trabalho de AboElFotoh e Al-Sumait (2001), faz uso de uma rede OPTI-net, baseada em RNA de multicamadas de perceptron, para aperfeiçoar o posicionamento de pontos em uma rede de computadores. Para isso, a RNA utiliza um grafo de posicionamento como entrada, resultando em diversas opções ótimas de fluxo para o grafo de entrada. Com isso é possível mapear enlaces em uma rede de computadores de um modo mais efetivo.

Outras redes neurais mostradas na Tabela 1 apresentam competência em encontrar similaridades e respostas entre dados que aparentemente não são de fácil entendimento (ANDONIE; SASU; BEIU, 2003; MA; TSAI, 1992; ZHANG; FENG; LIU, 2006).

Uma rede neural é basicamente consistida em camadas. A camada de entrada, onde a rede neural recebe os dados fornecidos, uma camada de saída, que a rede neural utiliza para apresentar a informação de resposta de acordo com o dado fornecido na entrada, e uma ou mais camadas ocultas, onde acontece o aprendizado e/ou a avaliação da rede neural. Cada camada, dependendo da complexidade da rede neural, contém um determinado número de neurônios e cada neurônio está ligado a outro neurônio da camada seguinte, exceto os neurônios da camada de saída.

Para que uma rede neural artificial possa fornecer resultados convenientes, é necessário que passe por uma fase de treinamento, onde seus pesos são ajustados de forma que ela se adapte aos diferentes estímulos de entrada. Pode-se treinar uma rede neural para

realizar uma função específica, ajustando os valores dos pesos e das conexões sinápticas, entre os neurônios. A RNA é treinada com um número predefinido de épocas, que é um período de tempo em que a rede neural obtém um aprendizado. Durante a fase de treinamento é onde ocorre o aprendizado geral da rede neural. A fase de treinamento envolve a normalização dos valores de entrada no intervalo escalar de [-1,1]. O treinamento de uma RNA implica na adaptação automática dos pesos de modo que a aplicação de um conjunto de valores de entrada forneça um conjunto de valores de saídas desejadas.

Após o treinamento, é testado um conjunto de valores de validação, ou seja, valores de entradas são apresentados à RNA e o conjunto de valores de saída obtidos é comparado com o conjunto de valores de saída desejados. Esta comparação faz com que a aprendizagem da RNA seja avaliada. Normalmente diversas entradas desses resultados são necessárias para formar uma rede neural então o processo se repete até que o desempenho da RNA com os dados de validação se estabilize em um valor considerado aceitável para o problema em análise.

Para que a RNA passe por processos de treinamento e aprendizagem, ela utiliza-se de algoritmos. Um algoritmo de aprendizado popular é o algoritmo *Backpropagation*, que é baseado na técnica de gradiente descendente para a redução de erros. O erro calculado é usado para ajustar os pesos em cada camada, até a função de erro retornar um erro menor. Neste algoritmo, antes de iniciar o processo de formação, os pesos são iniciados aleatoriamente e os conjuntos de valores de entradas e de saídas desejados devem ser normalizados.

Outra característica da RNA é a generalização na classificação das amostras. Uma técnica estatística clássica que é útil em determinar a capacidade de generalização de uma rede neural é a validação cruzada (STONE, 1978). A validação cruzada é utilizada durante o treinamento e ela tende a validar as respostas da RNA com um conjunto de dados diferentes daqueles que são usados para adaptar os pesos sinápticos. Esta técnica evita também que ocorra o fenômeno denominado "*overtraining*", treinamento excessivo, da rede neural. Uma RNA treinada em excesso aprende fielmente os dados de treinamento e apresenta péssima capacidade de generalizar este conhecimento. O uso da validação cruzada é altamente recomendável quando se tem um problema com volume de dados de treinamento grande e se deseja obter boa capacidade de generalização da rede neural, por isto a validação cruzada foi empregada em todos os procedimentos de treinamento das redes neurais desta dissertação.

Um dos focos principais para uso de redes neurais é aplicação a problemas de classificação. Um tipo de rede neural bastante utilizado neste tipo de problema é RNA do tipo

Cascade-Forward Backpropagation. Esse tipo de RNA possui seus neurônios fortemente conectados (HERTZ et al., 1991) e, além disso, possuem conexões de seus nós de entrada com cada camada posterior e cada camada subsequente tem os pesos e conexões vindos da entrada e de todas as camadas anteriores. A saída da rede se encontra na última camada. Um exemplo de *Cascade-Forward Backpropagation* é visto na figura 9.

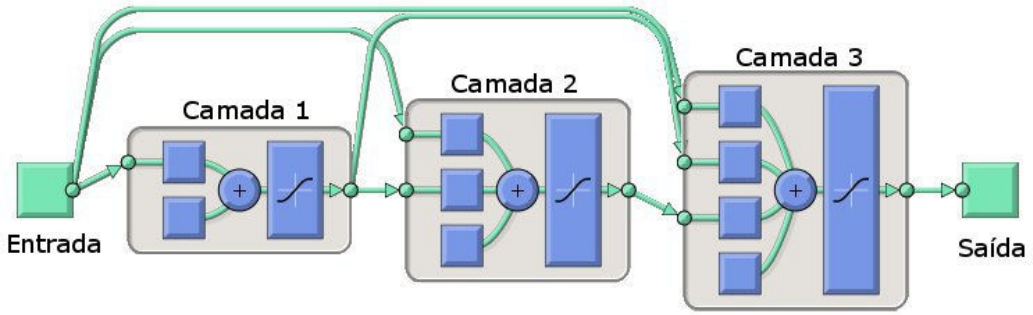


Figura 9: Exemplo de RNA Cascade-Forward.
Fonte: o Autor (2011).

2.3.3 Contextualização do Arbitrador Não-Linear

O estado da arte sobre redes neurais revela algumas contribuições definitivas para utilização da rede neural, no âmbito de classificação (ASADI et al., 2009; CIOCOIU, 2002; HAYKIN, 1998). Nestes trabalhos foram encontrados subsídios que auxiliam o projeto da identificação de similaridades entre código fonte e código binário.

Algumas contribuições em segurança da informação que utilizam redes neurais, para abordagens mais voltadas a conceitos de criptografia também foram estudadas. Um novo algoritmo de criptografia digital de imagens utilizando redes neurais é apresentado em Zhenga (2009). Esse algoritmo utiliza uma rede neural celular hiper-caótica usando características de sistemas dinâmicos caóticos.

Outras contribuições, que podem ser encontradas na literatura, apresentam métodos baseado em inteligência artificial para validação de software, usando previsão de erros para verificação e para confiabilidade de software. O trabalho de Zeng e Rine (2004) usa a reposta de previsão que a rede neural fornece para correção de defeitos em software, e na abordagem de Zhenga (2007) o sistema da previsão de erros é baseado em conjuntos de redes neurais. Já no trabalho de Reddy (et al., 2007), uma rede neural é usada para prever um módulo propenso

a falhas em uma aplicação web, e no trabalho de Lenic (et al., 2004), um sistema de auto-organização é construído para se obter a confiabilidade dos módulos.

2.4 CONCLUSÃO

Este capítulo apresentou os conceitos básicos dos assuntos abordados nesta dissertação. Foram apresentados os conceitos sobre a representação, geração e estrutura de um código fonte em binário, bem como os conceitos sobre SVM, hiperplanos, redes neurais artificiais e técnicas utilizadas.

Foram apresentados também os principais trabalhos encontrados na literatura científica das diversas áreas relacionadas, e que serviram de base e inspiração para o desenvolvimento da metodologia que será proposta no capítulo 3.

3 RASTREABILIDADE DE CÓDIGOS

3.1 INTRODUÇÃO

Este capítulo descreve a metodologia utilizada para a validação da rastreabilidade de códigos binários. Esta metodologia compreende a obtenção de propriedades intrínsecas aos códigos fonte/binário através da análise de código e a avaliação da significância dessas propriedades através de uma RNA levando-se em conta a correspondência entre um código fonte e um código binário. Tais propriedades versam sobre a lógica do fluxo de execução de um programa, como número de arestas e número de nós extraídos dos grafos de fluxo de controle individuais (um para cada função); e número de funções obtidas do grafo de chamadas mais o tamanho em bytes.

3.2 ABORDAGEM PROPOSTA

A abordagem proposta para a rastreabilidade de código binário envolve duas etapas. Na primeira etapa, utilizam-se ferramentas de software para extrair propriedades de ambos os códigos, fonte e binário. Tais propriedades podem ser simples, como tamanho, ou podem ser mais sofisticadas, como aquelas derivadas a partir dos grafos de fluxo de controle ou grafo de chamadas. Na segunda etapa, usa-se um arbitrador para avaliar a relevância das propriedades escolhidas a fim de determinar a correspondência entre um código fonte e um código binário. O método de rastreabilidade foi posto em prática com o uso de quatro propriedades: tamanho, número de procedimentos retirado do grafo de chamadas, o número de vértices e número de arestas retirados dos grafos de fluxo de controle.

3.2.1 Processo de extração das propriedades

Em um processo de compilação, uma grande quantidade de informações que devem ser levadas em conta na concepção de rastreabilidade dos códigos binários pode ser perdida. A quantidade de informações disponíveis para o código compilado é totalmente dependente do processo de compilação. A seguir, destacam-se algumas propriedades, independentemente

de códigos fonte e código binário, que podem ser explicitamente ou implicitamente disponíveis, a fim de entender a complexidade na concepção de um método de rastreabilidade.

Considerando o fato de que a maioria dos softwares embarcados é escrita em linguagem imperativa, algumas propriedades como nomes de variáveis, tipos de variáveis e nomes de procedimento podem ser perdidos durante o processo de compilação, porque o objetivo do compilador é maximizar desempenho. Este processo de maximização normalmente diminui a legibilidade do código binário, dificultando-se assim a extração dessas propriedades e, portanto, inviabilizando o uso das mesmas no método de rastreabilidade.

Outra possibilidade para rastrear um código binário seria o uso dos possíveis valores que as variáveis poderiam assumir em pontos distintos do programa. Esses valores podem ser computados através de técnicas de análise de fluxo de dados. No entanto, a análise de fluxo de dados para os códigos fonte e binário é perceptivelmente diferente. Enquanto que no código fonte o espaço é no nível de variáveis, no código binário é no nível de memória (pilha, heap e registradores). Essa diferença de níveis das linguagens aumenta o número de instruções contidas no binário, em comparação com o respectivo número de instruções do código fonte. Esta característica, apesar de ser positiva na otimização de códigos por compiladores, (uma vez que o rastreamento é feito em uma granulosidade fina) a mesma demanda maior complexidade de análise.

3.2.2 Relevância das propriedades

A metodologia empregada para verificar a importância do conjunto de propriedades a ser utilizado pelo método de rastreabilidade foi baseada em uma estratégia individual, verificando a contribuição de cada propriedade com a rastreabilidade utilizando como arbitrador uma rede neural artificial. As propriedades avaliadas foram as extraídas do grafo de chamadas e do grafo de fluxo de controle de cada função, assim como o tamanho dos códigos. O tamanho dos códigos foi utilizado dado que em um processo de compilação a relação de estruturas em um código fonte é normalmente refletida para um determinado número de instruções no código binário, assim podendo contribuir para com o método de rastreabilidade.

Para o método de rastreabilidade, aqui apresentado, é utilizada a sequencialidade de execução das instruções, seja ela de um código fonte ou de um código binário. O conjunto de todos possíveis caminhos de execução (i.e. sua sequencialidade) que ambos os códigos (fonte e binário) podem trilhar pode ser adquirido através da análise de fluxo de controle. Esta

análise demanda a construção de grafos de fluxo de controle individualizados para cada função e do grafo de chamadas. Para obtenção destes grafos foram usadas as seguintes ferramentas:

- GNU GCC (GNU, 2011) para códigos fontes, com a opção de compilação utilizada para *debugging* "–fdump–tree–cfg" que gera um arquivo *.cfg* que é a representação de grafo de fluxo de controle de um código compilado, que posteriormente será utilizado para extração das propriedades;
- IDA Pro (IDAPRO, 2011) para códigos binários, cuja interface fornece recursos para visualização dos grafos de fluxo de controle individuais e do grafo de chamadas.

Diante da obtenção destes grafos, foi possível extrair algumas propriedades encontradas em ambos os grafos de códigos binários e fontes. As propriedades foram: número de blocos básicos (nós do grafo) e de número de caminhos do fluxo de execução (arestas do grafo), extraídos dos grafos de fluxo de controle individuais para cada função e número de funções extraído do grafo de chamadas. Depois essas propriedades foram verificadas quais seriam relevantes para representar a origem de um código binário através de um código fonte.

Para o processo de extração dessas propriedades criaram-se *scripts* diferenciados tanto para os códigos fontes quanto para os códigos binários. Para os códigos fontes, foi criado um s*hell script* para compilar os códigos fontes com a opção de compilação "–fdump–tree–cfg", gerando assim todos os grafos de fluxo de controle (arquivos *.cfg).* Em seguida, outro *script* foi criado para extrair destes arquivos *.cfg* as propriedades de cada código fonte. De modo infortúnio, não foi possível fazer um *script* que coletava todas as propriedades de todos os códigos binários de uma só vez. Assim, a extração dessas foi feita individualmente para cada binário através de *scripts* escritos na linguagem Python. Um *script* contabiliza o número de blocos básicos e o outro o número de caminhos do fluxo de execução, ambos extraídos dos grafos de fluxo de controle. O número de funções foi adquirido de forma explícita pela própria interface da ferramenta IDA Pro.

Para verificar a relevância das propriedades, a rede neural artificial foi treinada e simulada para subconjuntos possíveis destas propriedades, verificando-se a importância de cada uma para a rastreabilidade. Para utilizar um arbitrador, baseado em RNA, que verifique a relevância das propriedades, o desenvolvimento da estrutura adequada incluiu: a escolha da função de transferência, seleção de um número de entradas, seleção do número de camadas e

escolha do número de neurônios em cada camada. Além disso, para desenvolver uma rede neural artificial é necessário que haja normalização dos valores utilizados como entrada dado que uma rede neural somente aceita entradas de -1 a +1 (função de ativação tangente hiperbólica sigmóide). Esta normalização foi realizada dividindo-se todos os valores de cada propriedade pelo maior de seus valores em módulo.

Contudo, antes da etapa de normalização foi feita uma etapa para associar um código fonte com um código binário. Esta correlação foi realizada para todas as propriedades extraídas (número de blocos básicos, número de caminhos do fluxo de execução, número de funções e tamanho em bytes de cada código). Para cada par correspondente de código, ou seja, fonte e seu respectivo binário, foi feita a subtração da propriedade do código fonte pela propriedade correspondente do código binário, com exceção do tamanho, no qual foi aplicada uma divisão do valor do código binário pelo valor do código fonte. A correspondência dos valores de número de blocos básicos foi chamada de número de nós, pois serem os nós dos grafos de fluxo de controle, dos valores de número de caminhos do fluxo de execução foi chamada de número de arestas, pois serem formados pelas as arestas dos grafos de fluxo de controle.

A rede neural utilizada para verificar a relevância das propriedades foi a *Cascade-Forward Backpropagation* com a função de ativação tangente hiperbólica, uma vez que seu uso é amplamente utilizado no contexto de classificação (ASADI et al., 2009; CIOCOIU, 2002;. HAYKIN, 1998). A escolha de números de neurônios na camada escondida envolveu treinamentos e testes de diferentes arranjos da RNA. Em todas essas redes foi usado o método de validação cruzada para critério de decisão e a partir da análise empírica, a melhor configuração de rede neural foi construída com dois neurônios na camada escondida e um neurônio para a camada de saída. Os valores de saída da RNA são fornecidos entre valores de [-1,1] assim como os seus valores de entrada. Para simular a rede neural foi utilizada a ferramenta de redes neurais, *Neural Network Toolbox*, do Matlab® (MOLER, 1980).

Para a fase de treinamento foram definidas duas classes, uma de valor +1 para representar valores de entrada de códigos correspondentes, ou seja, fonte e binário correspondente, e outra classe de valor -1 para representar os códigos não-correspondentes, porém o binário não sendo originado deste fonte. A associação dos códigos da classe não-correspondente foi feita de modo que cada código fonte fosse relacionado com um código binário diferente de seu par correspondente, e esta associação foi feita aleatoriamente.

Neste processo de validação das propriedades, 100 amostras formadas por 100 códigos fontes escritos em linguagem C (BURKARD, 2011; CRUZ, 2011), foram utilizadas para ambas as classes. Estes foram compilados utilizando o compilador GNU GCC para ambos ambientes Linux e Windows.

Para ambas as classes, o percentual usado para fase de treinamento foi de 70% e as amostras restantes foram utilizadas para a fase de avaliação. As figuras 10 e 11 ilustram exemplos de treinamento, na qual a figura 10 representa o treinamento para a classe de conjunto de códigos correspondentes, enquanto a figura 11 representa o treinamento para a classe de conjunto de códigos não-correspondentes. As propriedades da classe correspondente são representadas pela cor preta, e as propriedades da classe não-correspondente são representadas pela cor branca. As propriedades são representadas pelos seguintes símbolos: círculo, para o número de arestas; quadrado, para o número de nós; triângulo, para o número de funções e; losango, para o tamanho do código em bytes.

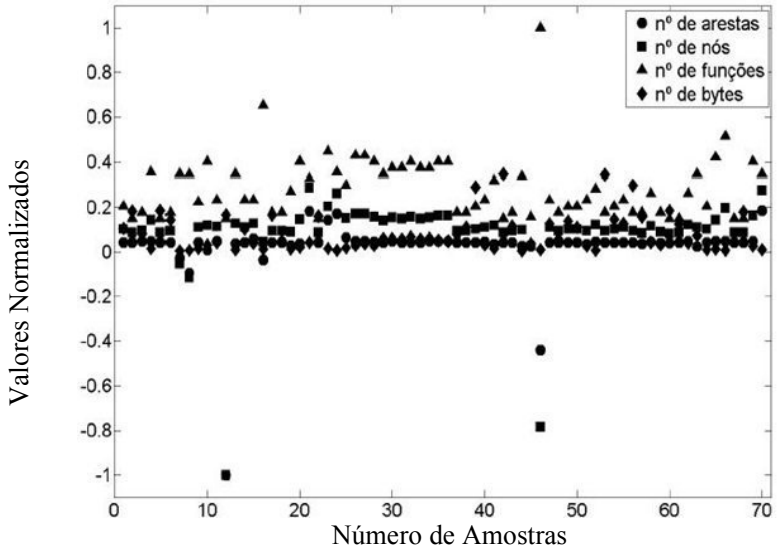


Figura 10: Conjunto de treinamento para a classe correspondente.
Fonte: o Autor (2011).

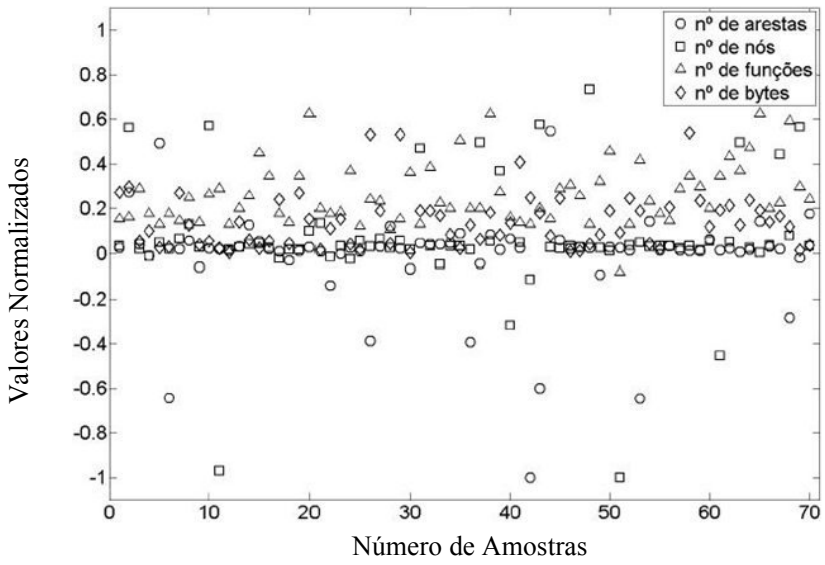


Figura 11: Conjunto de treinamento para a classe não-correspondente.
Fonte: o Autor (2011).

Os valores no eixo das abscissas das figuras 10 e 11 representam a enésima amostra e o eixo das ordenadas representa os valores das propriedades normalizados. Antes de prosseguir com a apresentação dos resultados da validação das propriedades propostas para o método de rastreabilidade, é importante definir as métricas de análise estatística utilizadas na avaliação: sensibilidade e especificidade. Estas métricas são usadas para validar as propriedades e para avaliar a classificação de códigos nas classes correspondentes e não-correspondentes, e são baseadas nos seguintes parâmetros:

- Verdadeiro Positivo (VP), que indica a quantidade de códigos correspondentes identificados como sendo códigos correspondentes;
- Falso Negativo (FN), que indica a quantidade de códigos correspondentes identificados como sendo não-correspondentes;
- Verdadeiro Negativo (VN), que representa a quantidade de códigos não-correspondentes classificados como sendo códigos não-correspondentes;
- Falso Positivo (FP), que representa a quantidade de códigos não-correspondentes classificados como sendo códigos correspondentes.

A especificidade indica quão boa é uma avaliação de identificação de códigos não-correspondentes. Ela é definida como a fração de códigos não-correspondentes que foi informada corretamente nos resultados como sendo não-correspondentes. Seu complemento é a taxa de falso positivo definida como a quantidade de códigos considerados correspondentes quando na verdade não são. A especificidade formalmente é dada pela seguinte equação:

$$\text{Especificidade} = \frac{\text{N.º de VN}}{\text{N.º de VN + N.º de FP}}$$

Já a sensibilidade indica quanto um teste de correlação foi bom, e é uma característica importante, dado que evita um retrabalho de verificação de um código correspondente. Sensibilidade no método de rastreabilidade é a relação entre códigos correspondentes e o número de correspondentes que foi informado no resultado. Esta tem como seu complemento a taxa de falso negativo, que são os códigos correspondentes informados como códigos não-correspondentes. A sensibilidade pode então ser descrita conforme a equação:

$$\text{Sensibilidade} = \frac{\text{N.º de VP}}{\text{N.º de VP + N.º de FN}}$$

Vale ressaltar que o objetivo é minimizar a taxa de falsos positivos, ou seja, aumentar a especificidade, visto que qualquer incidência de falsos positivos é obviamente um aspecto crítico para avaliação de software. Falsos positivos, no caso dos testes, seria a quantidade de elementos identificados incorretamente como correspondentes, quando na verdade estes elementos pertencem ao conjunto dos não-correspondentes. Por exemplo, em algum cenário que exista um medidor inteligente de energia elétrica que contenha um software não-correspondente embarcado, este pode ter uma função espúria que possibilite um atacante provocar um apagão na rede elétrica.

Contudo, os falsos negativos, que são a taxa de erros da identificação dos elementos correspondentes, também foram considerados, pois uma alta taxa de falsos negativos exigiria outra análise de cada programa a fim de se determinar se o programa é de fato correspondente. A seguir, será detalhado como foi realizada a seleção das propriedades utilizando-se uma rede neural artificial.

Primeiramente, para um conjunto de duas propriedades, os resultados obtidos foram das propriedades extraídas do grafo de controle de fluxo, ou seja, número de nós e de arestas de cada função do programa, como visto na figura 12 para os códigos correspondentes e na figura 13 para os códigos não-correspondentes.

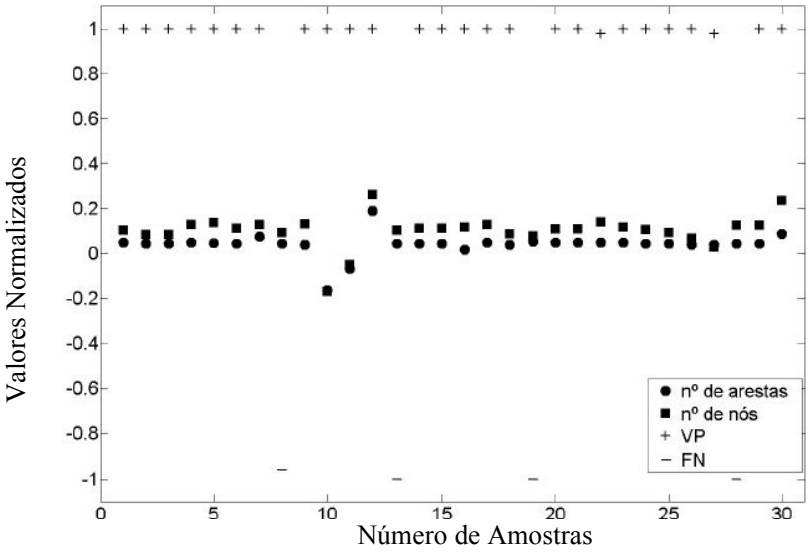


Figura 12: Avaliação da RNA com duas propriedades para a classe correspondente.
Fonte: o Autor (2011).

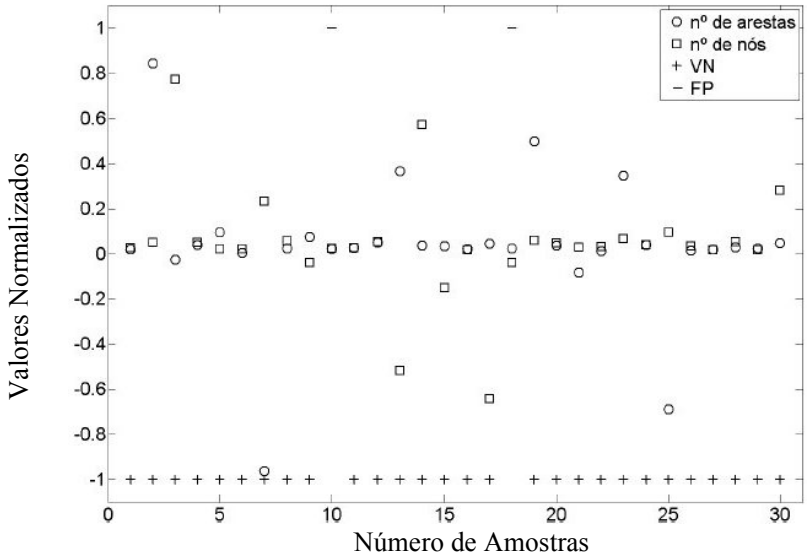


Figura 13: Avaliação da RNA com duas propriedades para a classe não-correspondente.
Fonte: o Autor (2011).

A figura 12 apresenta os resultados do método de validação das propriedades referentes aos grafos de controle de fluxo para um conjunto de pares de código fonte e código binário correspondentes, e a figura 13 apresenta os resultados para os pares de código fonte e binário não-correspondentes. Observa-se que a figura 10 apresenta 4/30 (≈13%) falsos negativos (FN) e 26/30 (≈87%) verdadeiros positivos (VP), resultando em uma sensibilidade de 0,87, e a figura 11 exibe 2 de 30 (≈6%) falsos positivos (FP) e 28/30 (≈93%) de verdadeiros negativos (VN), resultando em uma especificidade de 0,93.

Já para o conjunto de três propriedades, o resultado foi obtido com a adição do número de funções, extraído dos grafos de chamadas, às propriedades apresentadas anteriormente. As figuras 14 e 15 apresentam os resultados para estas três propriedades. Observa-se que diante da adição dessa propriedade, a rede neural adaptou seu conhecimento a todos os elementos não-correspondentes, aperfeiçoando os resultados, com o número de falsos positivos sendo nulo, especificidade de valor 1, e a taxa de falsos negativos manteve-se em ≈13%, sensibilidade de 0,87.

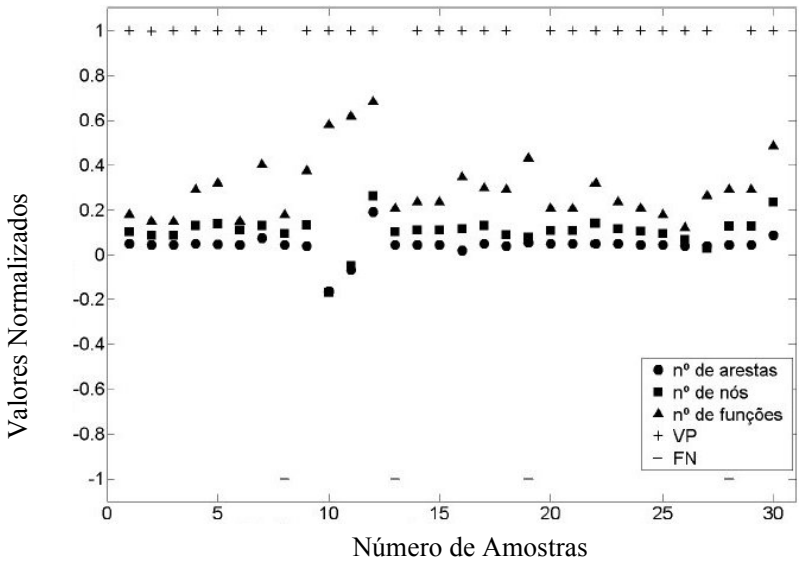


Figura 14: Avaliação da RNA com três propriedades para a classe correspondente.
Fonte: o Autor (2011).

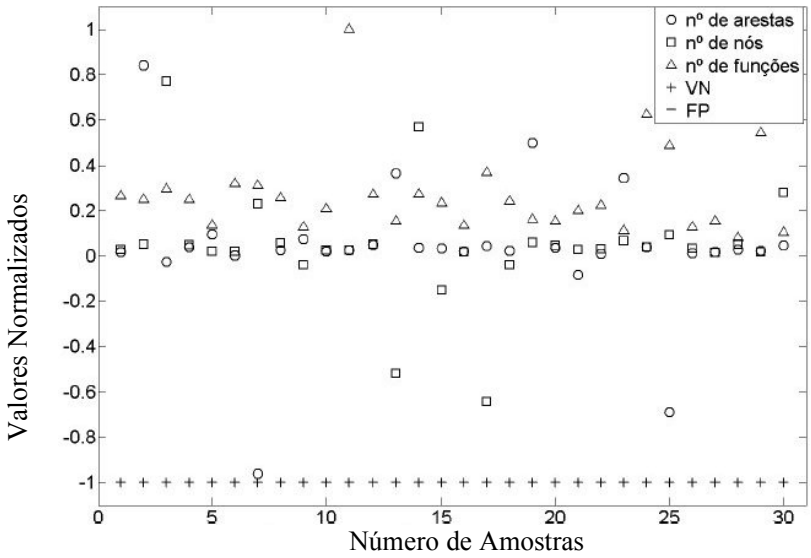


Figura 15: Avaliação da RNA com três propriedades para a classe não- correspondente.
Fonte: o Autor (2011).

Já para o conjunto de quatro propriedades, adicionando-se ao conjunto anterior o tamanho de código, a avaliação apresentou menos falsos negativos (10%), ou seja, houve acerto de 27 dos 30 códigos correspondentes usados, obtendo sensibilidade de 0,9. A avaliação para as quatro propriedades é visualizada nas figuras 16 e 17.

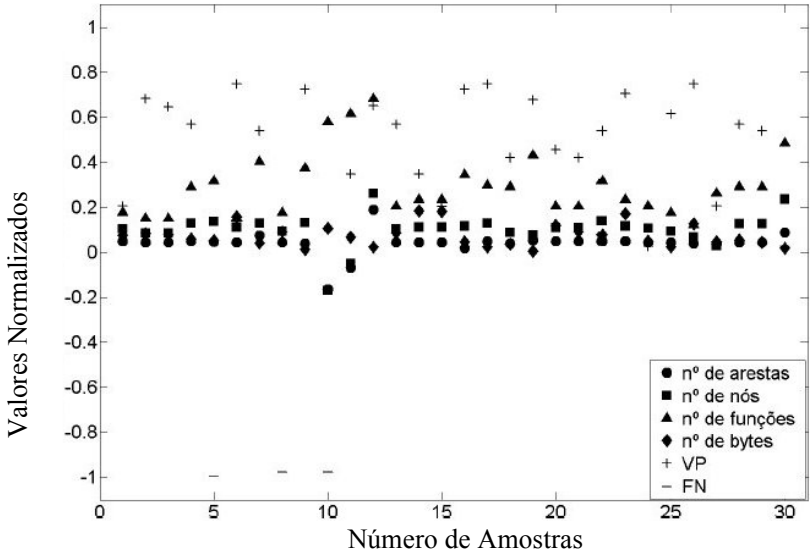


Figura 16: Avaliação da RNA com quatro propriedades para a classe correspondente.
Fonte: o Autor (2011).

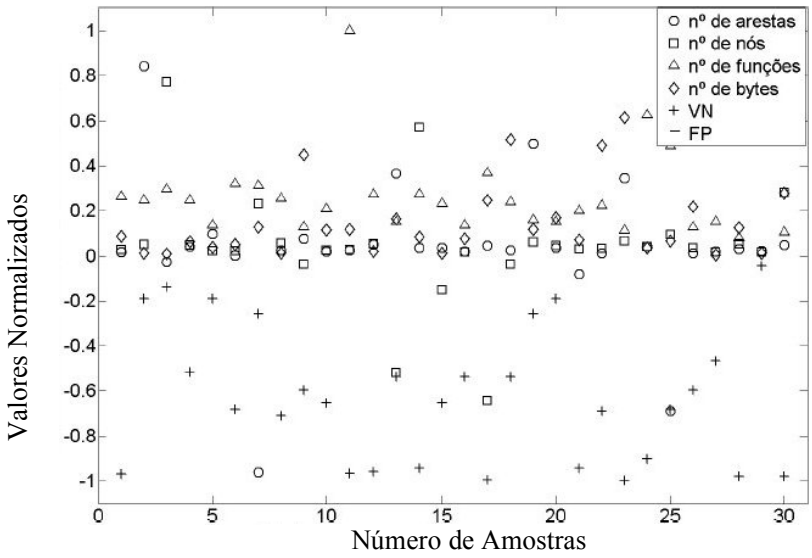


Figura 17: Avaliação da RNA com quatro propriedades para a classe não-correspondente.
Fonte: o Autor (2011).

Os resultados da rede neural para os não-correspondentes, figura 17, se apresentam entre os intervalos de valores [0, -1], ou seja, valores negativos. Com isso a taxa de acertos se manteve, ou seja, especificidade igual a 1, ao encontrar programas não-correspondentes. A fase de treinamento para esta última avaliação, apresentada nas figuras 16 e 17, é com os mesmos conjuntos apresentados nas figuras 10 e 11.

A tabela 2 apresenta o resultado da RNA na fase de avaliação para o conjunto de quatro propriedades. Foram apresentadas para a RNA pares de código correspondentes e não-correspondentes que nunca tinham sido a ela apresentados. As colunas representam as classes com seus respectivos número de acertos e de erros.

Classe	Acertos	Erros
Correspondente	27	3
Não-Correspondente	30	0

Tabela 2: Resultado da Rede Neural
Fonte: o Autor (2011).

As propriedades de cada par de códigos foram introduzidas nos nós de entrada da rede. As entradas número de arestas e número de nós, propriedades extraídas dos grafos de fluxo de controle, foram introduzidas no primeiro e no segundo nó de entrada,

respectivamente. O número de funções, extraído do grafo de chamada, foi introduzido no terceiro nó de entrada, e o tamanho de código no quarto nó de entrada. A figura 18 apresenta a topologia desta rede.

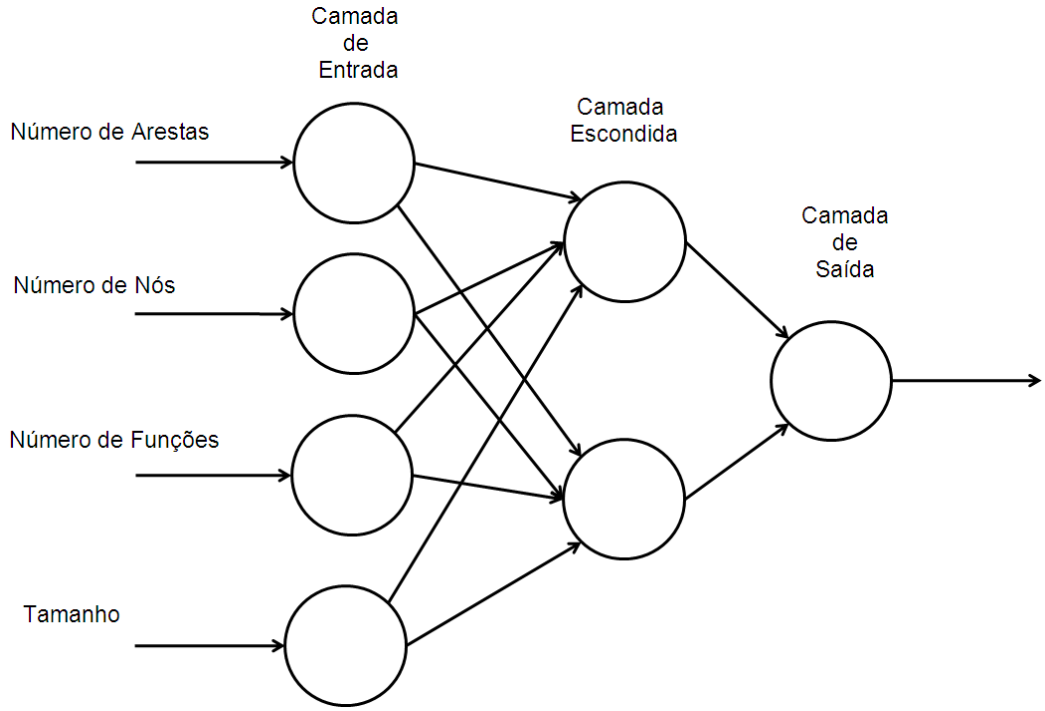


Figura 18: Topologia da rede neural.
Fonte: o Autor (2011).

A função de convergência escolhida para RNA foi a de erro médio quadrático, a condição de escolha foi a validação cruzada e durante o processo de treinamento, a época 17 com erro abaixo de 0,2, marcados na figura por um círculo, representa a melhor época obtida. A figura 19 exibe a convergência da melhor época atingida pela RNA, com menor erro.

Pela figura 19, percebe-se que a curva de validação apresenta quedas e ascensões no valor do erro, porém o treinamento continuou e foi possível encontrar um ponto com erro menor, por causa da validação cruzada. O treinamento só se encerrou quando foram atingidos 10 testes de validação, e logo em seguida se fez a escolha pela validação cruzada da melhor época em que a rede se desenvolveu melhor, ou seja, com menor erro.

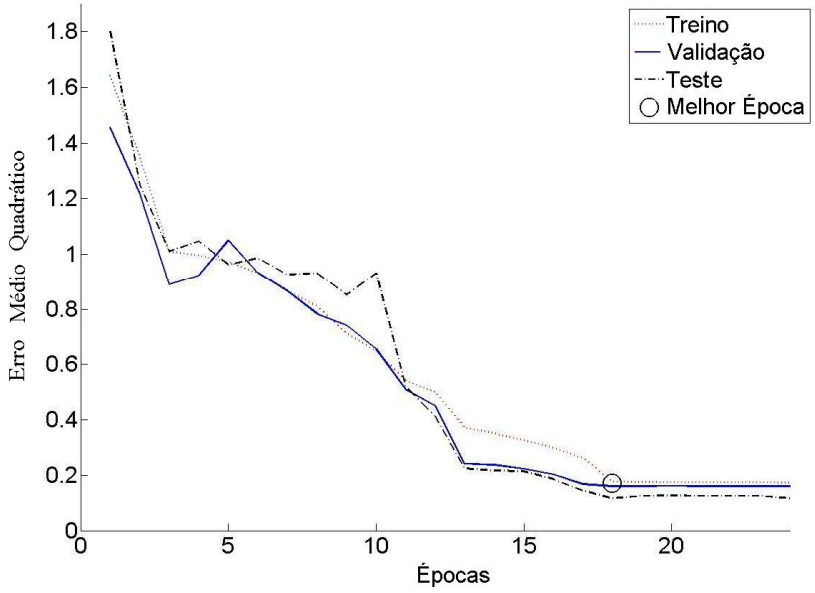


Figura 19: Convergência de treinamento usando validação cruzada.
Fonte: o Autor (2011).

3.3 CONCLUSÃO

Este capítulo apresentou uma proposta de validação para a metodologia de rastreabilidade. A metodologia proposta foi validada através da contribuição de cada propriedade em testes sucessivos utilizando-se uma RNA, na qual foi possível estabelecer qual conjunto de propriedades obteve melhor contribuição para rastrear códigos binários. As propriedades foram número de arestas e número de nós extraídos dos grafos de fluxo de controle, número de funções extraído do grafo de chamadas e o tamanho de código.

A seguir, avaliaremos nossa proposta diante de modificações dos códigos binários oriundas da infecção destes por códigos maliciosos reais, assim avaliando o método de rastreabilidade para um cenário mais realístico. Este cenário de infecção por códigos maliciosos é propício para avaliação dado à magnitude de ameaça que estes possam representar para aplicações sensíveis a segurança, por exemplo, sistemas de software militares.

4 AVALIAÇÃO E EVOLUÇÃO DA RASTREABILIDADE

4.1 INTRODUÇÃO

Neste capítulo é apresentada a avaliação experimental da metodologia proposta diante de um cenário onde os códigos não-correspondentes são códigos modificados por códigos maliciosos. É apresentada uma breve descrição da elaboração dos testes, bem como do processo de infecção dos códigos binários. A avaliação do método foi feito através do comparativo de diferentes arbitradores utilizando-se as propriedades vistas no capítulo 3. Os resultados foram avaliados através das métricas: sensibilidade e especificidade. Por fim, é apresentado um refinamento do método a partir da contenção seletiva das alterações nos códigos binários.

4.2 ELABORAÇÃO DOS CONJUNTOS DE TREINAMENTO ENVOLVENDO CÓDIGOS MALICIOSOS

Visto que a maioria dos códigos maliciosos tem como alvo predominantemente o ambiente Windows, focamos o método neste ambiente. Foram utilizados noventa e quatro códigos fontes, obtidos em Burkard (2011) e Cruz (2011), para o procedimento de avaliação. Antes de infectar os programas por códigos maliciosos, foi importante estabelecer alguns critérios de segurança para evitar a disseminação destes códigos maliciosos para todo ambiente de trabalho. Para isso, foi configurada uma estação de trabalho isolada para o processo infecção dos códigos e para posteriormente, extração das propriedades. As propriedades extraídas dos códigos infectados foram utilizadas para a elaboração da classe não-correspondente. Os códigos maliciosos utilizados foram: Vírus.Cabanas.a, Vírus.Win32.NGVCK.1003, Vírus.Win32.Qudos.4250 e Vírus.Win32. Artelad.2173 obtidos no portal VXHeavens (2011).

Foi avaliado o método de rastreabilidade utilizando-se individualmente cada um dos vírus supramencionados. Para o Vírus.Cabanas.a, noventa e três binários dos noventa e quatro compilados foram infectados. Destes noventa e três, setenta foram utilizados na fase de treinamento e, os vinte e três restantes, foram utilizados na avaliação da classe dos não-correspondentes. O conjunto de treinamento para esta avaliação é exibido na figura 20. Para

os demais códigos maliciosos Vírus.Win32.NGVCK.1003, Vírus.Win32.Qudos.4250 e Vírus.Win32.Artelad.2173, a totalidade dos códigos binários foram infectados (noventa e quatro), utilizando-se setenta binários infectados para o conjunto de treinamento e vinte e quatro para a fase de avaliação. Figuras 21, 22 e 23 exibem estes conjuntos para os vírus Vírus.Win32.NGVCK.1003, Vírus.Win32.Qudos.4250 e Vírus.Win32.Artelad.2173, respectivamente.

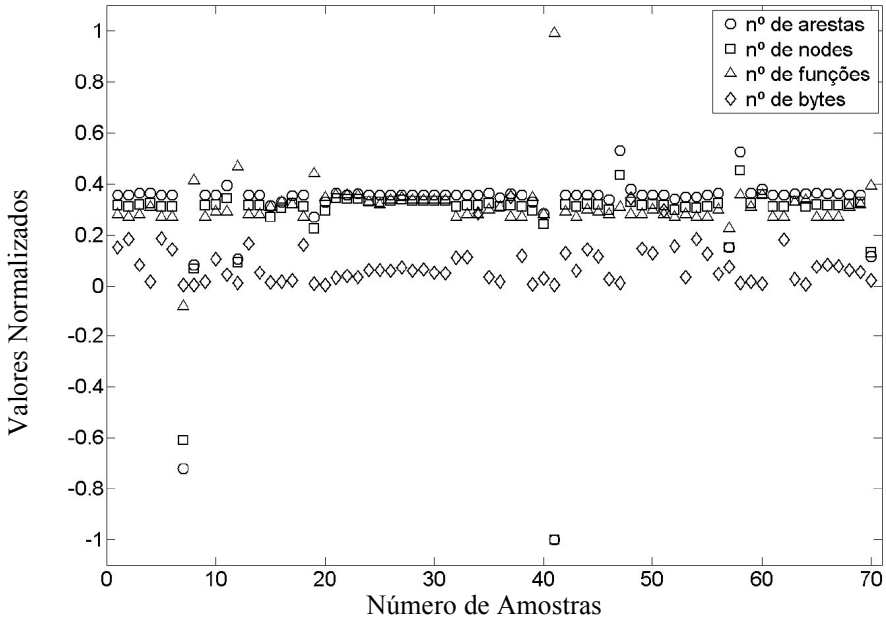


Figura 20: Conjunto de treinamento para a classe não-correspondente usando Cabanas.a
Fonte: o Autor (2011).

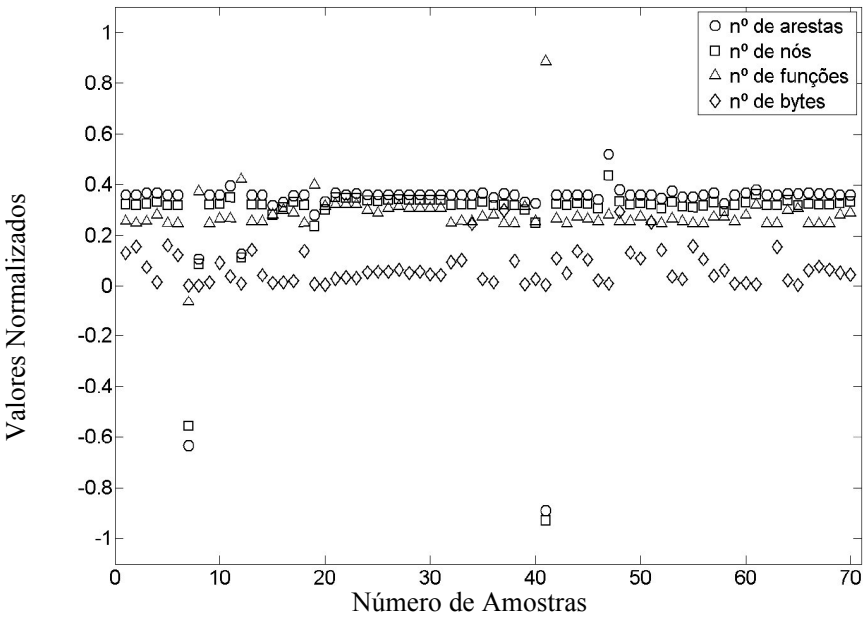


Figura 21: Conjunto de treinamento para a classe não-correspondente usando NGVCK.1003
Fonte: o Autor (2011).

Fonte: o Autor (2011).

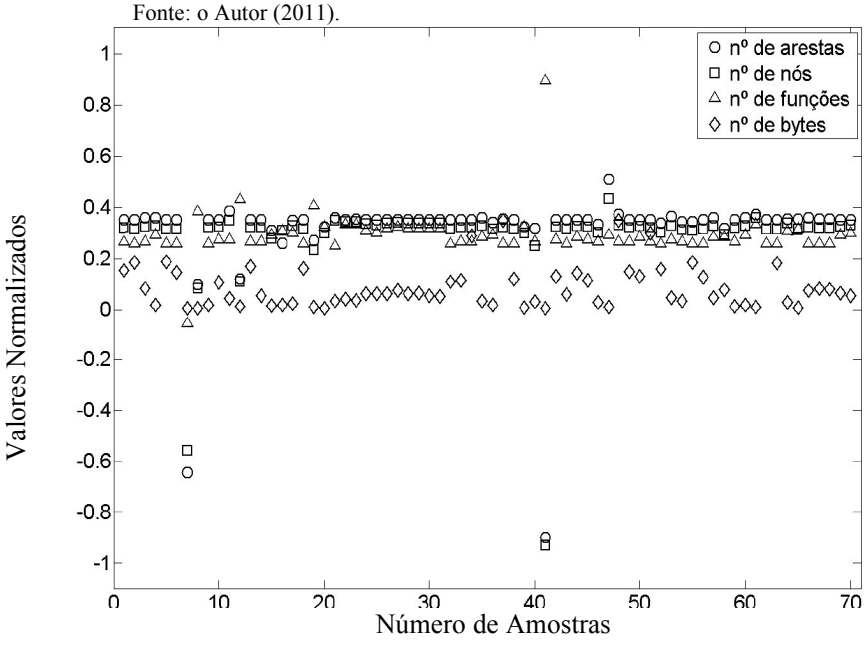


Figura 22: Conjunto de treinamento para a classe não-correspondente usando Qudos.4250
Fonte: o Autor (2011).

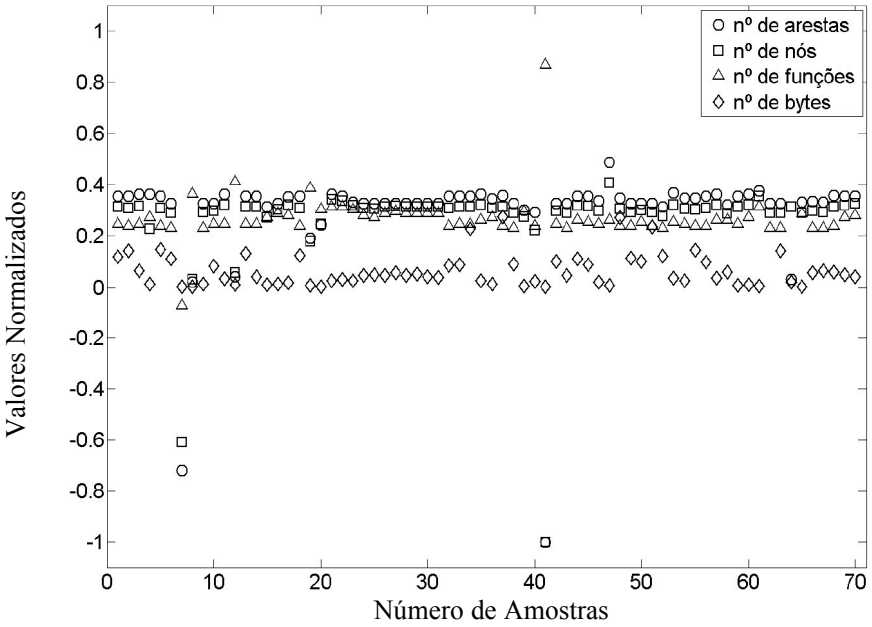


Figura 23: Conjunto de treinamento para a classe não-correspondente usando Artelad.2173
Fonte: o Autor (2011).

Enquanto que para a classe dos correspondentes utilizou-se os setenta originais para o conjunto de treinamento e vinte e quatro para o conjunto de avaliação. Este conjunto é exibido na figura 24.

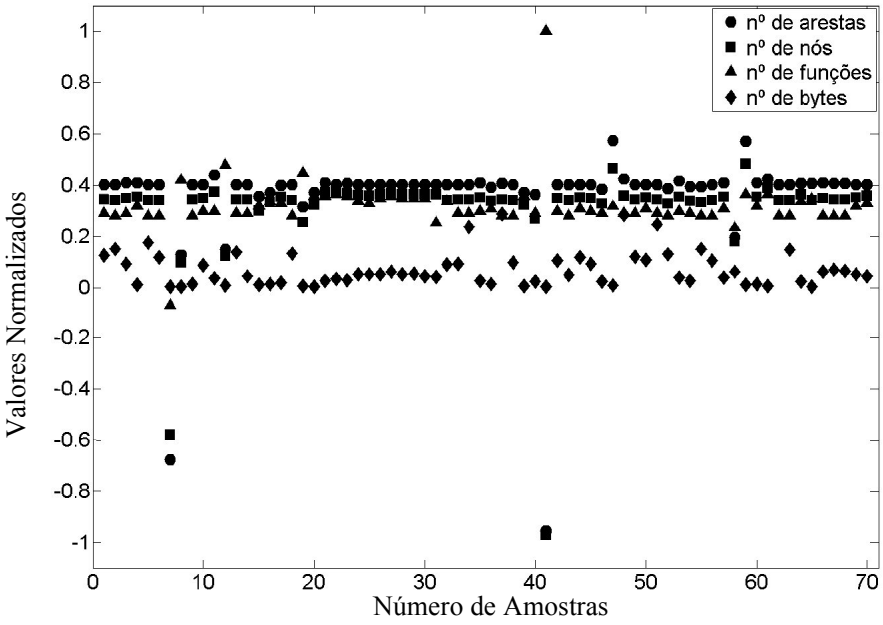


Figura 24: Conjunto de treinamento para a classe dos correspondentes
Fonte: o Autor (2011).

Após a avaliação para cada código malicioso, foi realizado um teste envolvendo o agrupamento de códigos correspondentes com os códigos não-correspondentes infectados pelos quatro códigos maliciosos. Para não existir uma desigualdade de conjuntos na realização desta avaliação, optou-se por um treinamento com 68 amostras de códigos infectados, dezessete para cada código malicioso, representando a classe de códigos não-correspondentes (vide figura 25). Para a classe dos códigos correspondentes utilizou-se o mesmo conjunto de treinamento (vide figura 24).

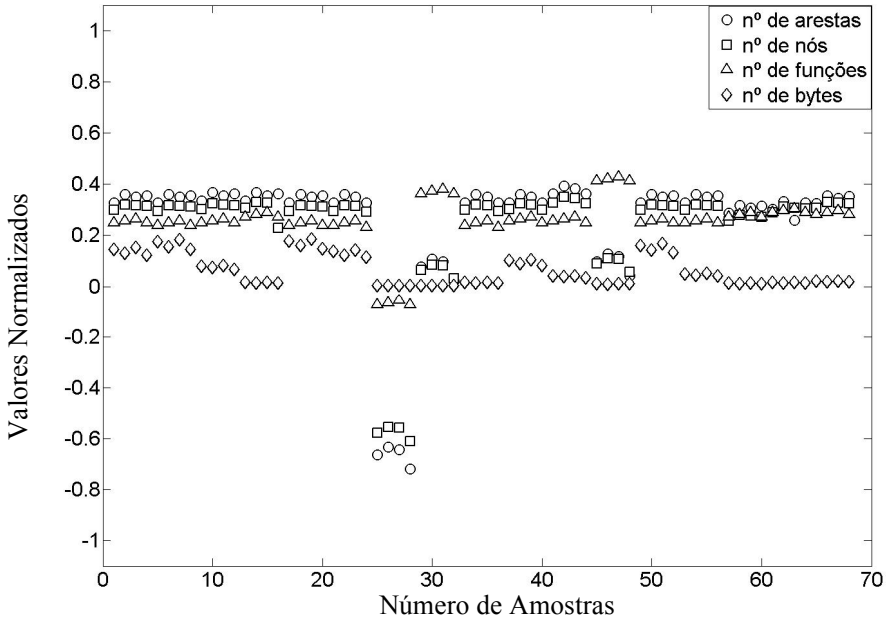


Figura 25: Conjunto de treinamento usando todos os códigos maliciosos.
Fonte: o Autor (2011).

Também foi realizada uma avaliação envolvendo todo o conjunto de códigos não-correspondentes infectados pelos códigos maliciosos. Para isso foi idealizado um treinamento maciço para a RNA. Com setenta exemplares de cada programa infectado por cada código malicioso foi obtido um conjunto de duzentos e oitenta elementos para o treinamento de não-correspondentes, enquanto para o conjunto dos códigos correspondentes foram utilizados os mesmos setenta programas originais.

Com os dados de treinamento devidamente idealizados e separados, a seguir são apresentados os resultados da avaliação.

4.3 AVALIAÇÃO DA RASTREABILIDADE USANDO RNA

Após a fase de elaboração dos conjuntos de treinamento, explicados na seção 4.2, os conjuntos de avaliação foram aplicados, que representam aproximadamente 30% do total de códigos. Na avaliação, para os elementos dos códigos correspondentes, a RNA devia fornecer uma resposta entre 0 e 1, ou seja valores positivos, enquanto que para os códigos não-correspondentes a RNA devia responder valores negativos, entre 0 e -1.

A primeira avaliação foi realizada com o código malicioso Vírus.Cabanas.a. A figuras 26 e 27 apresentam os resultados da avaliação da RNA para programas infectados pelo código malicioso Vírus.Cabanas.a. Para a avaliação da classe dos correspondentes, cujo conjunto de treinamento é visto na figura 24, a RNA forneceu uma taxa de acerto de 19 entre os 24 avaliados (≈79%), obtendo sensibilidade de 0,79, e para a classe não-correspondentes, cujo conjunto de treinamento é visto na figura 20, a RNA obteve como resultado 1/23 (≈4%) falso positivo, resultando em uma especificidade de 0,96.

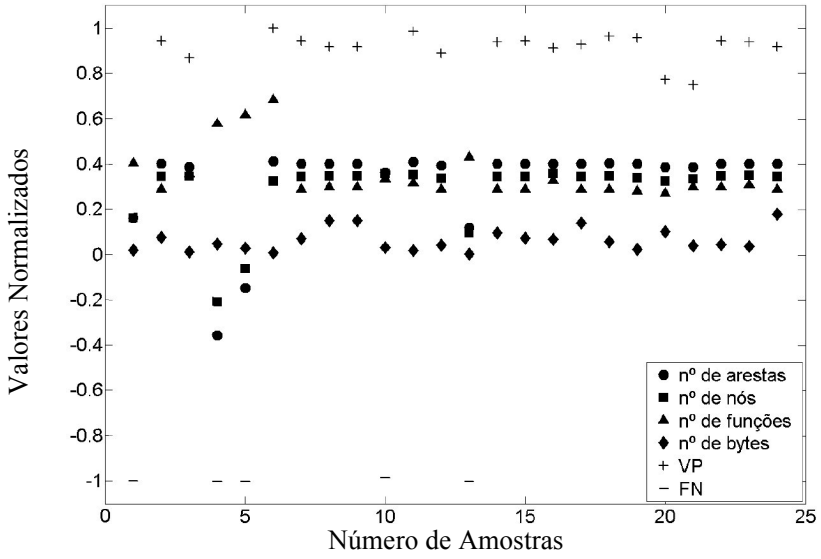


Figura 26: Avaliação da RNA para códigos da classe correspondente tendo como treinamento os conjuntos apresentados nas figuras 20 e 24.
Fonte: o Autor (2011).

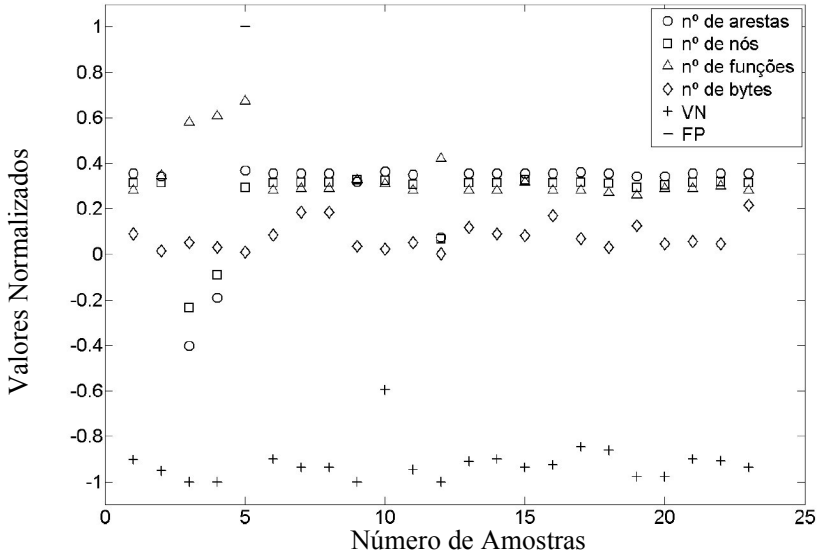


Figura 27: Avaliação da RNA para a classe não-correspondente, tendo como treinamento os conjuntos apresentados nas figuras 20 e 24.
Fonte: o Autor (2011).

Com isso a RNA, treinada com os conjuntos apresentados nas figuras 20 e 24, foi capaz de entender as nuances que o Vírus.Cabanas.a faz ao modificar um binário. Após, foi realizado a avaliação da RNA diante de programas infectados pelo código malicioso Virus.Win32.NGVCK.1003. Essa avaliação resultou em uma sensibilidade de 0,66, ou seja, taxa de acerto de códigos correspondentes de 16/24, e para os códigos não-correspondentes obteve-se uma especificidade de 0,91, ou seja, dois falsos positivos entre os 24 testados. As figuras 28 e 29 exibem estas avaliações da RNA com esse código malicioso.

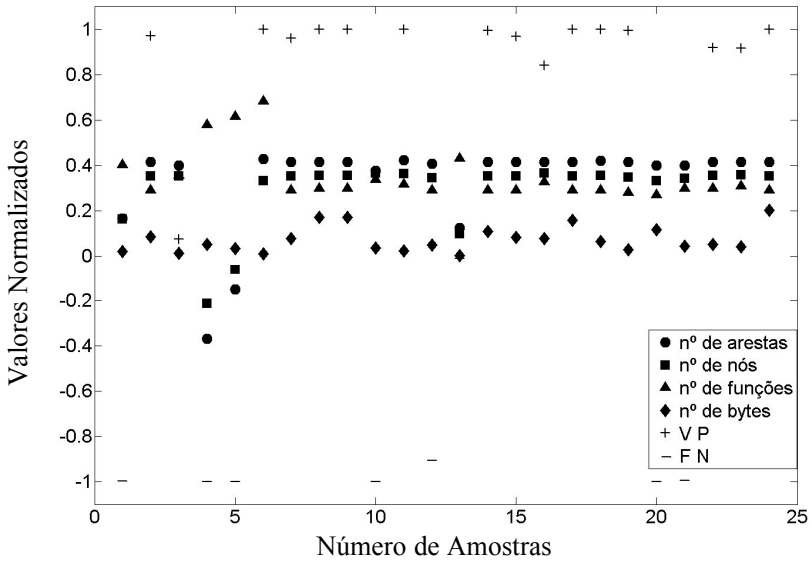


Figura 28: Avaliação da RNA para a classe correspondente tendo como treinamento os conjuntos apresentados nas figuras 21 e 24.
Fonte: o Autor (2011).

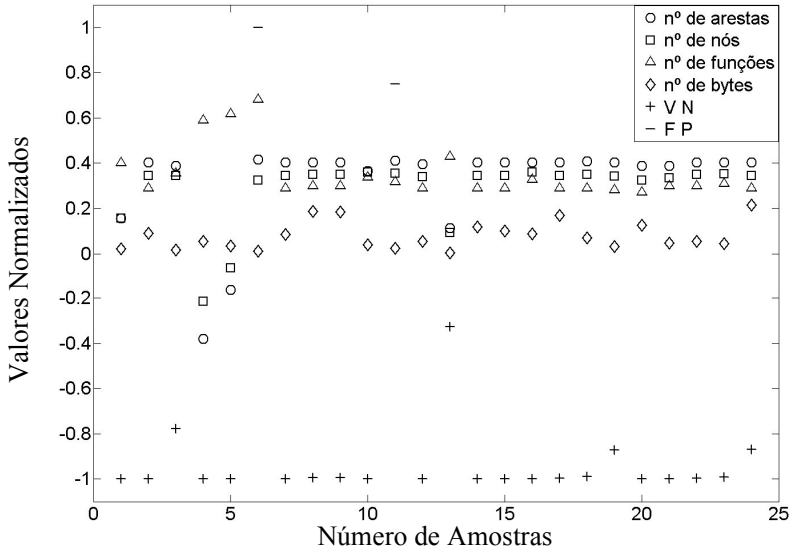


Figura 29: Avaliação da RNA para a classe não-correspondente, com infecção pelo Win32.NGVCK.1003 tendo como treinamento os conjuntos apresentados nas figuras 21 e 24.
Fonte: o Autor (2011).

Já para os demais códigos maliciosos a RNA se superou em identificar códigos não-correspondentes. Com a avaliação dos binários modificados pelo Virus.Win32.Qudos.4250 a RNA forneceu zero falso positivo com 16/24 de verdadeiros positivos, especificidade igual a 1 e sensibilidade igual a 0,66. Já nas avaliações com o código malicioso

Virus.Win32.Artelad.2173 foram obtidos 15/24 acertos de códigos correspondentes e zero falso positivo, sensibilidade igual a 0,63 e especificidade igual a 1. As tabelas 3, 4, 5 e 6 exibem os resultados da RNA.

	Saída	
Entrada	Correspond	Não-Corresp
Correspond	16	8
Não-Correspond	0	24

Tabela 3: Avaliação com Virus.Win32.Qudos.4250
Fonte: o Autor (2011).

	Saída	
Entrada	Correspond	Não-Corresp
Correspond	15	9
Não-Correspond	0	24

Tabela 4: Avaliação com Virus.Artelad.2173
Fonte: o Autor (2011).

	Saída	
Entrada	Correspond	Não-Corresp
Correspond	19	5
Não-Correspond	1	22

Tabela 5: Avaliação com Virus.Cabanas.a
Fonte: o Autor (2011).

	Saída	
Entrada	Correspond	Não-Corresp
Correspond	16	8
Não-Correspond	2	22

Tabela 6: Avaliação com Virus.NGVCK.1003
Fonte: o Autor (2011).

Após as avaliações individuais para cada vírus, foi feita uma avaliação da RNA com características de infecção de todos os quatro vírus (17 para cada vírus), visualizados na figura 25. Com isso foi possível identificar todos os códigos maliciosos, não havendo assim nenhum falso positivo (especificidade 1).

As figuras 30 e 31 exibem os resultados desta avaliação da RNA, com 100% na identificação de códigos não-correspondentes e 9/24 (≈37%) de falsos negativos (sensibilidade de 0,63).

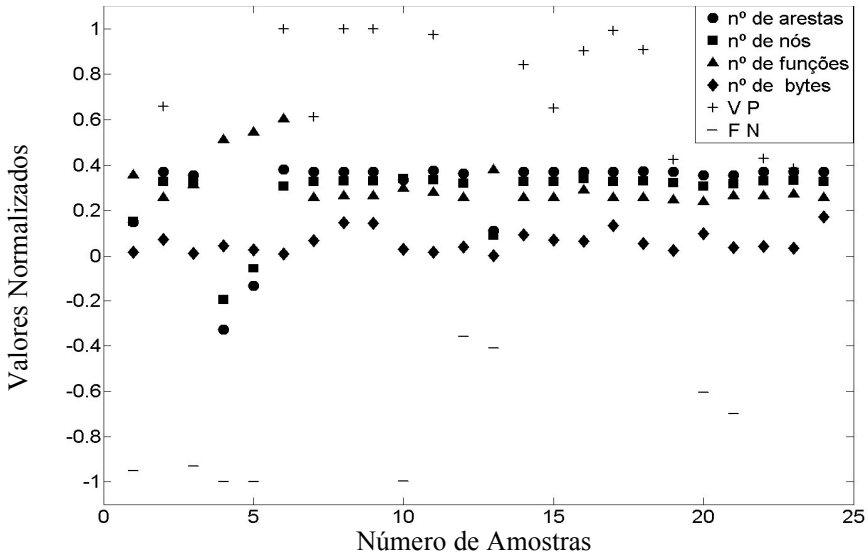


Figura 30: Avaliação da RNA para classe correspondente tendo como treinamento os conjuntos apresentados nas figuras 24 e 25.
Fonte: o Autor (2011).

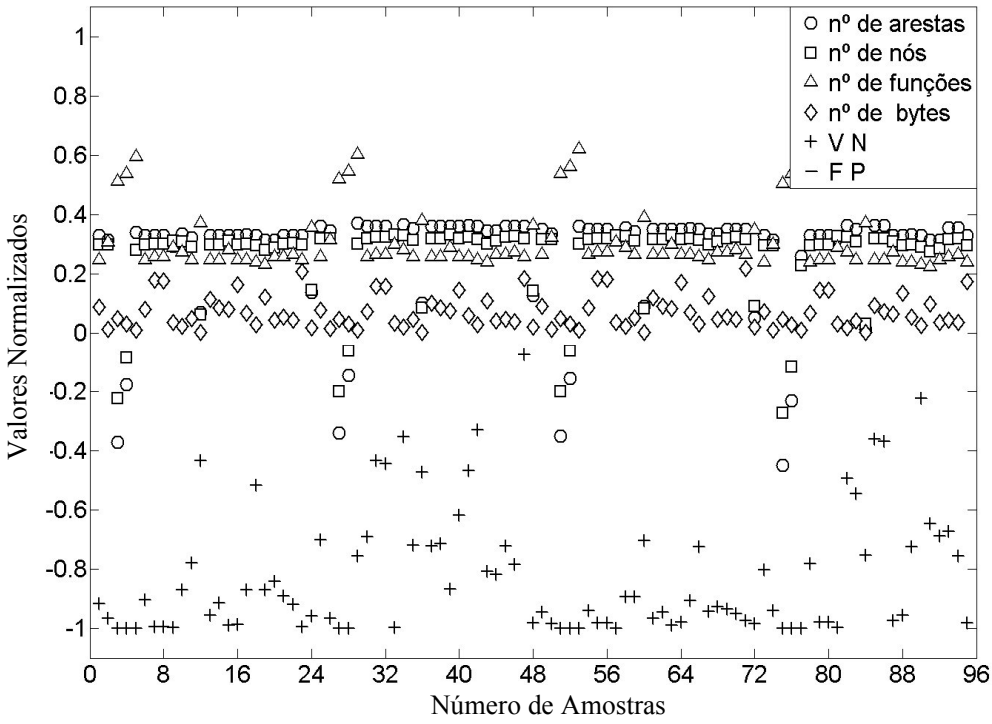


Figura 31: Avaliação da RNA para a classe não-correspondente tendo como treinamento os conjuntos apresentados nas figuras 24 e 25.
Fonte: o Autor (2011).

A RNA também foi avaliada agrupando todos os elementos de ambas as classes. Para isso a RNA usou para seu treinamento o conjunto da classe correspondente, visto na figura 24, com todos os conjuntos da classe não-correspondentes apresentados nas figuras 20 a 23. Logo foram usadas para treinamento da classe não-correspondente 280 amostras infectadas, 70 para cada código malicioso e apenas 70 elementos de códigos não infectados. E com esse treinamento maciço, a RNA conseguiu obter um resultado favorável com taxa de 15/24, sensibilidade em torno de 0,63, na identificação de programas correspondentes, e obtendo uma taxa de 4% de falsos positivos (especificidade de 0,96).

As figuras 32 e 33 mostram o resultado dessa avaliação. Neste caso também foram usados 24 elementos do conjunto dos correspondentes para avaliação e 95 dos não-correspondentes.

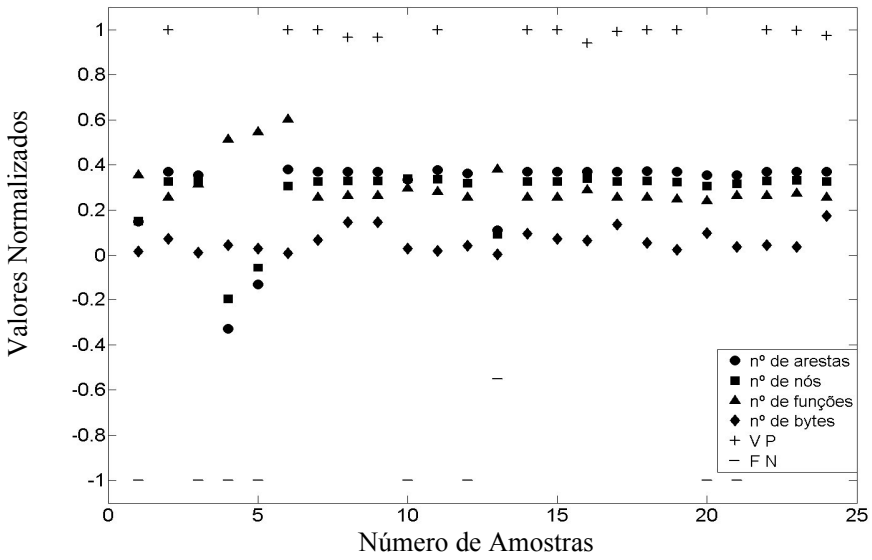


Figura 32: Avaliação para classe correspondente com treinamento maciço
Fonte: o Autor (2011).

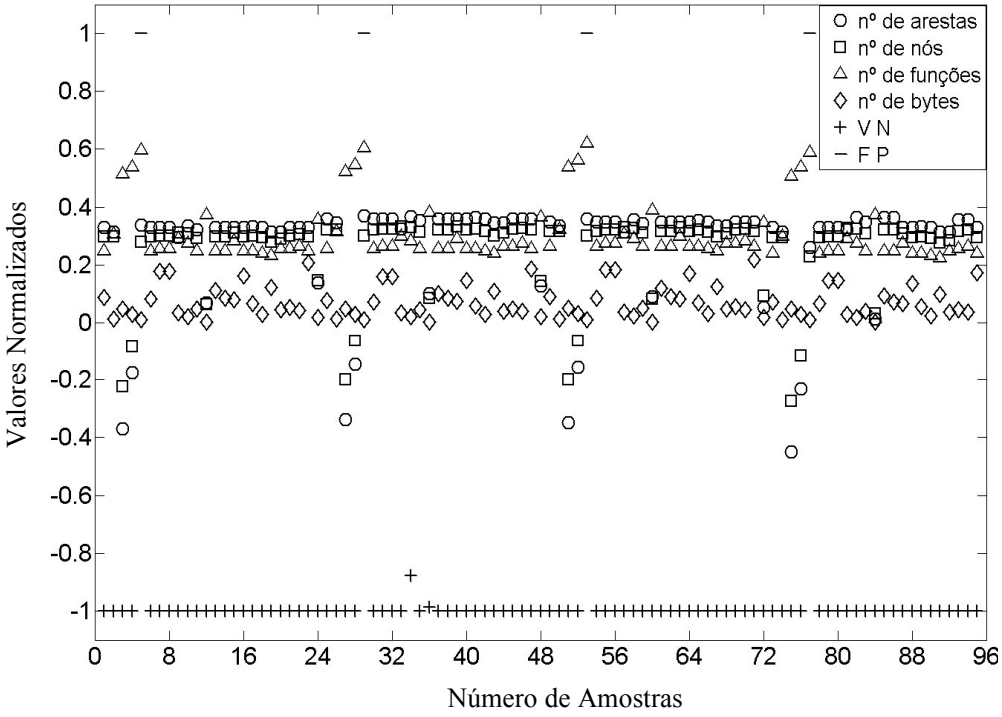


Figura 33: Avaliação para classe não-correspondente com treinamento maciço.
Fonte: o Autor (2011).

Apesar dos resultados das últimas avaliações apresentarem uma taxa de 9/24 de falsos negativos, o método de rastreabilidade é ainda relevante, pois apresenta uma baixa taxa de falsos positivos (4/95), com especificidade 0,96, que significa que o método é eficaz quanto à identificação de alterações.

A seguir é feito um comparativo da RNA com demais arbitradores lineares, levando em consideração a taxa de sensibilidade e especificidade, requisitos importantes para avaliação de software.

4.4 COMPARATIVO DA RNA COM ARBITRADORES LINEARES

Para a comparação da RNA com métodos lineares foram investigados dois arbitradores que utilizam técnicas de separação linear: SVM e o algoritmo de Hiperplano de Separação.

A técnica de SVM é tipicamente aplicada para construção de classificadores (ANGULO et al., 2006; MEN et al., 2008), por isso as avaliações da rastreabilidade utilizando

SVM também foram comparadas. O classificador escolhido de forma empírica foi uma SVM de margem suave e norma-2. Para o mapeamento dos dados de treinamento para o espaço de classificação foi utilizado uma função linear. Para encontrar o hiperplano de separação foi utilizada uma função de programação quadrática.

Os dados fornecidos para o treinamento da SVM foram os mesmos utilizados na RNA. Porém, para a SVM cada programa foi transformado em um ponto. Cada ponto foi formado pela combinação das propriedades dos códigos, número de arestas, número de nós, número de funções e tamanho de código. A SVM teve então que identificar uma divisão no espaço de quatro dimensões, ou seja, um hiperplano que dividia o espaço de 4-D em dois semi-espaços. Os conjuntos das avaliações que foram classificados de acordo com o hiperplano traçado foram os mesmos utilizados apara avaliar a rede neural. Para o teste com o código malicioso Vírus.Cabanas.a, 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 – sensibilidade de 0,79, 19/24, porém com um número maior de falsos positivos 7/23, com especificidade de 0,70, contra 1/23 da taxa da RNA.

Já na avaliação com os demais códigos maliciosos, a SVM apresentou resultados diferentes da identificação da RNA. Com o Vírus.Win32.NGVCK.1003 a SVM forneceu 14 códigos como sendo códigos correspondentes dentre os 24 do conjunto dos correspondentes, obtendo sensibilidade 0,58, porém dos 24 não-correspondentes, ela classificou 11 como sendo códigos correspondentes, especificidade de 0,54. No teste de avaliação com os programas modificados pelo Vírus.Win32.Qudos.4250 a SVM classificou corretamente 19 elementos dos 24 totais de correspondentes, sensibilidade 0,79, e apenas 17/24 dos não-correspondentes, especificidade 0,70, e para a última avaliação com modificação dos binários, pelo código malicioso Vírus.Win32.Artelad.2173, a SVM conseguiu classificar 16 dos 24 correspondentes, obtendo sensibilidade 0,66, e classificou erroneamente 10 dos 24 elementos não-correspondentes, resultando em especificidade de 0,58. As tabelas de 7 a 10 apresentam os resultados de avaliação da SVM.

	Saída	
Entrada	Correspond	Não-Corresp
Correspond	19	5
Não-Correspond	7	17

Tabela 7: Avaliação SVM Virus.Win32.Qudos.4250
Fonte: o Autor (2011).

	Saída	
Entrada	Correspond	Não-Corresp
Correspond	16	8
Não-Correspond	10	14

Tabela 8: Avaliação SVM Virus.Artelad.2173
Fonte: o Autor (2011).

	Saída	
Entrada	Correspond	Não-Corresp
Correspond	19	5
Não-Correspond	7	16

Tabela 9: Avaliação SVM Virus.Cabanas.a
Fonte: o Autor (2011).

	Saída	
Entrada	Correspond	Não-Corresp
Correspond	14	10
Não-Correspond	13	11

Tabela 10: Avaliação SVM Virus.NGVCK.1003
Fonte: o Autor (2011).

As avaliações da SVM com os códigos maliciosos separados forneceram melhores resultados do que a RNA, quanto a sensibilidade, exceto para o Virus.Win32.NGVCK.1003. Contudo, obteve baixa especificidade na classificação dos códigos não-correspondentes.

Quando foi realizada a avaliação da SVM utilizando conjunto de códigos correspondentes, visto figura 24, com a combinação dos códigos maliciosos, 17 para cada código malicioso como visto na figura 25, essa também resultou em uma alta taxa de identificação dos códigos correspondentes. A SVM classificou apenas 4 elementos, de um total de 24, do grupo como não-correspondentes, sensibilidade de 0,83. Em contra partida a RNA classificou de modo incorreto 9 destes elementos.

Porém, com a avaliação dos códigos não-correspondentes, a SVM não obteve resultados relevantes como a RNA. De um total de 95 elementos não-correspondentes avaliados, sendo 23 oriundos das modificações feitas pelo Vírus.Cabanas.a e 24 de cada um dos três códigos maliciosos restantes, a SVM classificou apenas 26 elementos como sendo realmente do conjunto de não-correspondentes, ou seja, especificidade baixa em torno de 0,27 uma taxa de erro de aproximadamente 72%. A tabela 11 apresenta os resultados da avaliação da combinação dos códigos maliciosos da SVM em comparação com os da RNA.

	Saída			
	RNA		SVM	
Entrada	Correspondente	Não-Correspondente	Correspondente	Não-Correspondente
Correspondente	15	9	20	4
Não-Correspondente	0	95	69	26

Tabela 11: Avaliação com a combinação dos códigos maliciosos com treinamento maciço.
Fonte: o Autor (2011).

Para as avaliações no caso de treinamento maciço, 70 amostras de cada código malicioso totalizando 280 amostras de códigos não-correspondentes, figuras 20 a 23, e setenta amostras de códigos correspondentes, figura 24, o arbitrador utilizando SVM apresentou resultados inferiores quanto ao arbitrador utilizando ANN. No desenvolvimento do arbitrador utilizando SVM foi percebido, que quando um conjunto de códigos não-correspondentes sobrepunha em mais que o dobro em quantidade o conjunto de códigos correspondentes, a SVM não conseguia dividi-los de modo eficaz. Logo, a SVM não convergiu para um hiperplano que separasse os conjuntos do treinamento maciço. Os resultados da SVM pioraram, pois classificavam todos os elementos do conjunto dos correspondentes como sendo modificados por códigos maliciosos, já que a maioria dos programas avaliados pertencia ao conjunto dos não-correspondentes.

Após verificação e avaliação da SVM, verificamos com nosso algoritmo, denominado Hiperplano de Separação. O algoritmo é consistindo em traçar diversos planos para dividir duas classes de conjuntos, em nosso caso, conjunto de códigos correspondentes e de códigos não-correspondentes. Porém, diferente do SVM, cada plano é constituído por pontos que representam cada programa, e não por vetores suporte, como descrito no capítulo 2. Cada ponto do nosso universo, p_1, .., p_n, é a representação de um código. Cada dimensão de um ponto significa uma propriedade extraída do código, portanto foram usadas quatro dimensões, uma para cada propriedades. A taxa '*r*', que é a taxa para limitar o número de falsos positivos, utilizada em nossos testes foi no intervalo de 0 e 1/20 (5%) inclusive.

Portanto ao se realizar a avaliação deste algoritmo com o conjunto de treinamento maciço, obteve-se um resultado melhor do que a SVM, chegando a ter como saída 15/24 de verdadeiros positivos, mas em contrapartida a RNA conseguiu obter o mesmo resultado, com tempo mais eficaz e com custo computacional menor. A tabela 12 apresenta os resultados da avaliação do algoritmo em comparação com os resultados da avaliação da RNA.

	Saída			
	RNA		Algoritmo	
Entrada	Correspondente	Não-Correspondente	Correspondente	Não-Correspondente
Correspondente	15	9	15	9
Não-Correspondente	4	91	0	95

Tabela 12: Avaliação de todos os códigos maliciosos.
Fonte: o Autor (2011).

4.5 REFINAMENTO DO MÉTODO

O refinamento do método de rastreabilidade utilizando RNA foi desenvolvido para obter um melhoramento no arbitrador em sua capacidade de identificar diferentes graus de infecção dos códigos. Sendo assim, o arbitrador escolhido para o refinamento foi a rede neural artificial uma vez que ela identificou todos os elementos não-correspondentes infectados pelos quatro códigos maliciosos, ou seja, de especificidade igual a 1.

Para realizar o refinamento, os valores obtidos das propriedades dos códigos binários modificados pelos códigos maliciosos foram alterados para que aproximassem gradativamente dos valores das propriedades dos códigos correspondentes. Os valores das propriedades de número de funções, arestas e nós assim como o tamanho dos códigos modificados foram alterados para que um código não-correspondente parecesse com um programa autêntico, ou seja, um código correspondente.

Estas alterações nos valores das propriedades foram realizadas gradativamente, simulando uma redução no grau de infecção. Primeiramente fizeram-se as subtrações dos valores das propriedades dos códigos correspondentes com os valores das propriedades dos códigos não-correspondentes. Depois, para simular a redução da infecção, utilizou-se o valor de 10% dessas subtrações para somá-lo aos valores das propriedades dos códigos não-correspondente, gerando assim, um conjunto de códigos não-correspondentes com 10% de redução em sua infecção. A cada iteração do processo de simulação de redução da infecção, somou-se mais 10% dessa subtração, gerando conjuntos de códigos não-correspondentes com valores de propriedades cada vez mais próximos dos valores das propriedades dos códigos correspondentes. Logo, ao final de 10 iterações, todos os códigos não-correspondentes obtiveram valores de propriedades iguais aos dos códigos correspondentes e existiam onze conjuntos de códigos não-correspondentes para realizar o refinamento, um conjunto

concebido pela infecção dos quatro códigos maliciosos e dez concebidos pela simulação de redução de infecção.

Para realizar a avaliação do arbitrador escolhido para o refinamento, esse foi avaliado onze vezes, uma para cada conjunto de códigos não-correspondentes, e a cada avaliação foram utilizados 95 códigos de cada conjunto. O resultado da RNA é exibido na figura 34 como sendo a curva preta. Esta mostra a taxa de falsos positivos para cada conjunto de não-correspondentes, junto com o nível de infecção variando de códigos infectados pelos quatro códigos maliciosos (mostrado como 1) até os códigos não-correspondentes com os valores de propriedades iguais aos códigos correspondentes (mostrado como 0). Por exemplo, com o conjunto de não-correspondente com grau de infecção 1,0, o arbitrador obteve zero falso positivo, com o conjunto com grau de infecção 0,9, o arbitrador obteve um falso positivo, e foram obtidos 60 falsos positivos para a avaliação com o conjunto com grau 0, onde os valores das propriedades dos códigos são idênticos aos valores das propriedades dos códigos correspondentes.

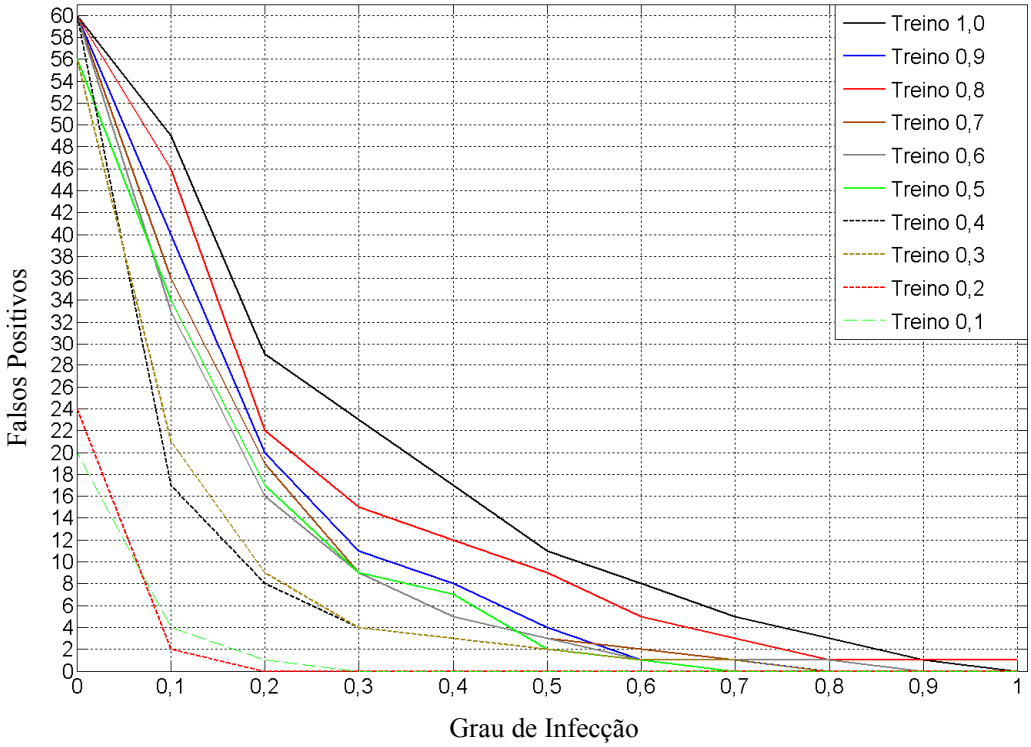


Figura 34: Taxa de falsos positivos para treinamentos distintos
Fonte: o Autor (2011).

As curvas na figura 34 são resultados de avaliações de diversos arbitradores baseados na mesma construção da RNA para o método de rastreabilidade porém com conjuntos de treinamento distintos. Como pode ser visto na figura, são avaliados 10 arbitradores, cada qual com um conjunto de treinamento para os códigos não-correspondentes alterados pelo processo de simulação de redução de infecção.

Pela figura 34, também se pode observar, que a taxa máxima de falsos positivos alcançados é de sessenta códigos. Quando todos os valores dos códigos não-correspondentes são idênticos aos valores dos códigos correspondentes, dos noventa e cinco códigos não-correspondentes avaliados, trinta e cinco destes são classificados como falsos negativos pelo arbitrador, pois esses estão relacionados com os nove falsos negativos identificados pelo arbitrador na avaliação com os códigos correspondentes, visto na figura 30. Nesse conjunto de trinta e cinco, existem nove códigos para cada código não-correspondente caracterizado pelos códigos maliciosos NGVCK.1003, Qudos.4250 e Artelad.2173 e 8 códigos para o Cabanas.a com valores de propriedades idênticos as valores de propriedades dos códigos correspondentes relacionados erroneamente como códigos não-correspondentes. Portanto a taxa máxima é de sessenta códigos falsos positivos como esperado. A tabela 13 expressa os resultados dos 10 arbitradores.

Arbitrador \ Grau de Infecção	**1**	**0,9**	**0,8**	**0,7**	**0,6**	**0,5**	**0,4**	**0,3**	**0,2**	**0,1**	**0**
Treino 1,0	0	1	5	8	10	12	20	24	36	55	60
Treino 0,9	0	0	0	1	1	4	8	11	20	40	60
Treino 0,8	1	1	1	3	5	9	12	15	22	46	60
Treino 0,7	0	0	1	1	2	3	5	9	19	36	60
Treino 0,6	0	0	1	1	1	3	5	9	16	33	60
Treino 0,5	0	0	0	0	1	2	7	9	17	34	56
Treino 0,4	0	0	0	1	1	2	3	4	8	17	60
Treino 0,3	0	0	0	1	1	2	3	4	9	21	56
Treino 0,2	0	0	0	0	0	0	0	0	0	2	24
Treino 0,1	0	0	0	0	0	0	0	0	1	4	20

Tabela 13: Taxa de falsos positivos por treinamentos distintos
Fonte: o Autor (2011).

Para concluir com o refinamento do método, é importante dizer que cada arbitrador citado nesta seção foi treinado com o mesmo conjunto de códigos correspondentes. Dos noventa e quatro códigos correspondentes compilados no ambiente Windows, setenta desses, que são apresentados na figura 24, foram usados para treinamento e vinte e quatro desses

utilizados para avaliação. A figura 35 mostra o número de verdadeiros positivos para cada avaliação de cada arbitrador.

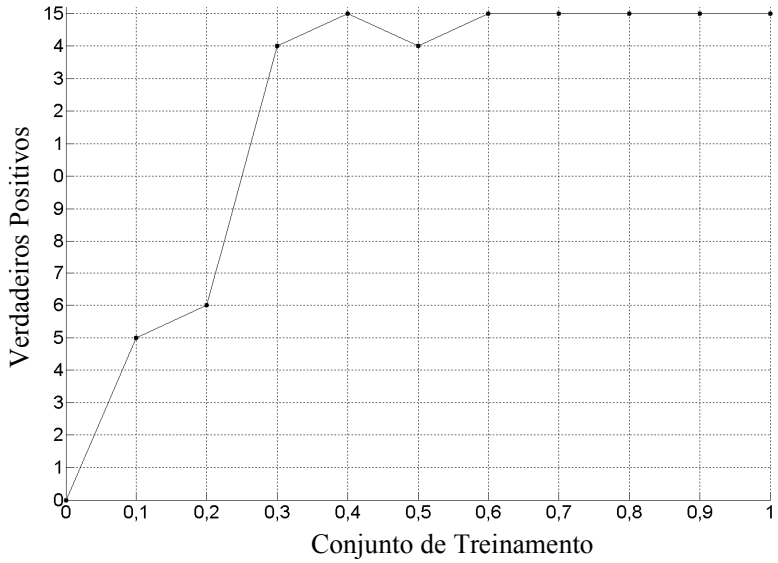


Figura 35: Taxa de verdadeiros positivos para treinamentos distintos
Fonte: o Autor (2011).

Nota-se que a maior taxa de verdadeiros positivos é de 15/24, para os arbitradores treinados com conjunto dos não-correspondentes modificados em 0,4, 0,6, 0,7, 0,8, 0,9 e 1,0 de grau de infecção.

Ao observar a figura 35 e a tabela 13, nota-se que o arbitrador com o conjunto de treinamento com grau de infecção 0,4 foi o arbitrador que obteve o melhor desempenho, ou seja, resultados com menor taxa de falso positivo (maior especificidade), com o melhor desempenho de verdadeiros positivos (maior sensibilidade). O conjunto de treinamento com grau de infecção 0,4 é aquele que possui 60% de redução nos valores das propriedades quando comparado aos códigos modificados por códigos maliciosos.

4.6 CONCLUSÃO

Com diferentes cenários de avaliação da proposta para a rastreabilidade, a escolha de um RNA como método de rastreabilidade mostrou ser bastante eficaz e eficiente, pois a RNA produziu resultados melhores quando comparados com outros arbitradores lineares.

Avaliações com o arbitrador linear SVM não retornaram resultados satisfatórios e o algoritmo de Hiperplano de Separação apesar de retornar resultado equivalente a RNA, este se mostrou inviável dado suas restrições de tempo e de escalabilidade, e por sua complexidade ser de $\Theta(n^d)$.

Dado que a melhor escolha para arbitrar quanto à correspondência entre um código fonte e seu respectivo binário foi da RNA, buscou-se o refinamento deste através da observação de seu grau de percepção quanto às alterações nas propriedades destes códigos. O conjunto para refinamento foi elaborado através da contenção seletiva das propriedades que definem um código correspondente e um código não-correspondente. Resultados comprovaram que com o conjunto de treinamento com 0,4 de grau de infecção a RNA obteve desempenho equivalente quando o conjunto avaliado foi dos códigos infectados e melhor desempenho a partir de conjuntos de avaliação com graus de infecções reduzidas, logo esta RNA serve como o melhor arbitrador para a rastreabilidade.

5 CONSIDERAÇÕES FINAIS

É apresentado, neste capítulo, o resumo das conclusões da pesquisa descrita nesta dissertação, expondo as principais contribuições e sugestões para trabalhos futuros.

5.1 RESUMO DO TRABALHO

Rastreabilidade de códigos refere-se à verificação de unicidade entre um código binário e seu código fonte respectivo. No campo da Metrologia Legal, é fundamental garantir que o software embarcado em um dispositivo corresponde a uma versão que foi previamente aprovada pela Autoridade Metrológica.

Dado que a avaliação de software em medidores é normalmente baseada na avaliação do código fonte torna-se importante garantir que o processo de compilação não introduza falhas de segurança, *backdoors* ou comportamentos indesejados. Para isso, buscou-se correlacionar códigos binários (embarcados no dispositivo) com códigos fontes previamente avaliados através de propriedades inerentes a suas lógicas de execução.

Esta correlação através da lógica tomou como base ferramentas de análise de programas, na qual é possível determinar o fluxo de execução, seja no código fonte como no código binário, bem como determinar o fluxo de dados de tais códigos. A estratégia, utilizada para correlação do presente dissertação, foi baseada no tamanho dos códigos e na suas lógicas de execução obtida através de dois grafos comumente utilizados em análise de programas: grafo de fluxo de controle e grafo de chamadas. O primeiro estabelece os possíveis caminhos de execução para cada função existente; já o segundo, estabelece o relacionamento de chamadas entre estas funções.

Após uma fase de obtenção das propriedades extraídas dos grafos supracitados, o objetivo foi identificar quais eram relevantes para a rastreabilidade. Para isso, foi implementada uma rede neural para inferir sobre a correspondência entre códigos fontes e códigos binários diante de um processo associativo das propriedades previamente adquiridas. Este processo basicamente consistiu da adição de cada propriedade, uma a uma, na entrada da rede neural buscando-se como resposta a menor taxa de falsos positivos. O conjunto de entradas resultante foi composto de tamanho dos códigos, número de caminhos de fluxo de execução e número de blocos básicos extraídos dos grafos de fluxo de controle e número de funções extraídos dos grafos de chamadas.

Logo depois de se concluir o processo de extração e de verificação da aderência das propriedades passou-se para a fase de avaliação experimental, na qual foi avaliada a eficácia do método de rastreabilidade diante de códigos binários infectados por códigos maliciosos. Nesta avaliação foram desenvolvidos e comparados métodos lineares e não lineares. Notou-se que os métodos lineares ou demandavam muito tempo e muito custo computacional ou não apresentavam resultados satisfatórios ou não eram escalonáveis. Em comparação com os métodos lineares, o método não linear de rede neural artificial obteve resultados satisfatórios com custo temporal e computacional reduzido.

Após a fase de comparação de diferentes métodos para rastrear códigos binários, foi realizada uma fase de refinamento cunhada de análise de sensibilidade. Nesta, foi possível aprimorar a sensibilidade da rede diante de um grau de infecção reduzido em 60%, obtendo-se os mesmos resultados para o conjunto de avaliação original tanto para os códigos correspondentes como para os não correspondentes. Contudo, com resultados melhores para os conjuntos de avaliação com códigos não correspondentes com graus de infecções diferenciadas.

5.2 PRINCIPAIS CONTRIBUIÇÕES

A principal contribuição desta dissertação foi a elaboração de uma metodologia para rastreabilidade de códigos binários dado um processo qualquer de compilação. Agregar confiança neste processo de compilação, em relações comerciais principalmente, é extremamente útil para Metrologia Legal. Esta age como uma entidade externa que realiza controle e avaliação de dispositivos com software embarcado, assegurando-os que diante de um código fonte avaliado o mesmo comportamento é mantido no seu respectivo binário compilado e gerado a partir de um processo de compilação não assegurado. Por exemplo, assegurar que um binário embarcado em um dispositivo de medição possui o mesmo comportamento de um código fonte previamente avaliado.

Através da correlação de duas áreas distintas, análise de programas e redes neurais artificiais, pôde-se desenvolver uma metodologia para rastrear um par de códigos (fonte e binário) baseada na extração de propriedades relevantes e na sua posterior classificação quanto sua correspondência, tendo como arbitrador uma rede neural artificial. O desempenho da metodologia está bem caracterizado através da avaliação experimental que, além de

confirmar uma taxa muito baixa de falsos positivos (alta especificidade), também oferece uma quantidade razoável de acerto em identificar verdadeiros positivos (alta sensibilidade).

Resultados da metodologia abrangendo conjuntos de avaliação gerados a partir de códigos modificados através de códigos maliciosos mostraram uma correspondência em torno de 90% para a rastreabilidade dos códigos binários com uma baixa taxa de falsos positivos (~4%), como presente nos artigos publicados (BOCCARDO et al. 2010; CARMO et al., 2010). Apesar de a metodologia focar-se em rastreabilidade de códigos binários nesta dissertação, a mesma pôde ser generalizada para mapeamento de programas gerados por ambientes de compilação diferenciados, ou até mesmo por linguagens de compilação distintas. Esta generalização, apesar de conter resultados preliminares, quanto ao escopo do método, está publicada em Nascimento (et al., 2010).

A partir dos resultados apresentados pôde-se verificar a viabilidade do uso da metodologia proposta com RNA para rastrear códigos binários, o que é fundamental para a Metrologia Legal no processo de certificar que um dispositivo de medição em campo possui embarcada a versão compilada de um software que foi previamente avaliado.

5.3 TRABALHOS FUTUROS

Alguns aspectos relacionados aos assuntos abordados foram identificados durante o desenvolvimento do trabalho, mas não foram tratados com a profundidade necessária. Esses aspectos são apresentados como trabalhos futuros, importantes para a continuidade do trabalho.

Um dos possíveis caminhos para melhorar a abrangência e desempenho do método, consiste na utilização conjunta da proposta aqui apresentada com métodos de análise de fluxo de dados, buscando-se propriedades não somente na lógica dos programas, como também nos valores que as variáveis podem assumir durante a execução do programa e no relacionamento entre as mesmas. Pode-se também, através desta técnica, averiguar os parâmetros de função e valores de retorno. Uma das possíveis abordagens é a utilização da técnica *Value Set Analysis* (BALAKRISHNAN, 2007), que consiste em buscar estes valores de variáveis através da técnica de análise estática de interpretação abstrata (COUSOT, 2007).

Outro possível caminho é a avaliação do impacto da utilização de técnicas de transformação de código por ofuscação no método de rastreabilidade proposto. Ofuscações têm sido comumente utilizadas para proteção da propriedade intelectual (COLLBERG;

NAGRA, 2010; LINN; DEBRAY, 2003), contudo, também podem ser aplicadas com o intuito de esconder um comportamento malicioso (CHRISTODORESCU; JHA, 2003). Estas ofuscações podem ser aplicadas tanto no fluxo de controle como de dados, impactando de certa forma as propriedades recolhidas diante do método aqui proposto. Análise de código ofuscado, como no trabalho de Boccardo (2009), pode nortear caminhos para que propriedades sejam obtidas da lógica da execução independente se o código está ofuscado ou não.

Outra possibilidade de trabalho futuro envolve a investigação de outras propriedades que possam ser obtidas através de variantes de grafos – números cruzados, ciclos, isomorfismo etc. – que sejam significativas para estabelecer a correspondência entre os códigos fonte e os códigos binários.

REFERÊNCIAS

ABOELFOTOH, H. M. F. ; AL-SUMAIT, L. S. A Neural approach to topological optimization of communication networks, with reliability constraints**. IEEE Transactions on Reliability**, New York, v. 50, n. 4, p. 397-408, 2001.

ANDONIE, R. ; SASU, L. ; BEIU, V. Fuzzy ARTMAP with relevance factor. international In: JOINT CONFERENCE ON NEURAL NETWORKS, 2003, Portland. **Proceedings …** New York: IEEE, 2003.v.3, p. 1975-1980.

ANGULO, C. et al. Multi-classification by using tri-class SVM. **Neural Processing Letters**, Amsterdam, v. 23, n. 1, p. 89–101, 2006.

ASADI, R. ; MUSTAPHA, N. ; SULAIMAN, N. New supervised multi layer feed forward neural network model to accelerate classification with high accuracy. **European Journal of Scientific Research**, Victoria, v. 33, n. 1, p. 163-178. 2009.

BALAKRISHNAN, G. **WYSINWYX:** What you see is not what you eXecute. 2007. Thesis (PhD of Phylosophy in Computer Science) — Computer Science Department, University of Wisconsin, Madison, 2007.

BOCCARDO D. **Context-sensitive analysis of x86 obfuscated executables.** 2009. Tese (Doutorado.em Engenharia Eletrica) - Universidade Estadual Paulista, São Paulo, 2009.

BOCCARDO D.et al. Traceability of executable codes using neural networks. In: BURMESTER, M. et al. (Eds). **Information Security**. – 13th International Conference on Information Security. Berlin: Springer, 2011. (Lecture Notes in Computer Science, v. 6531). ISBN 978-3-642-18177.

BRUMLEY D. ; NEWSOME J. **Alias analysis for assembly**. Pittisburgh: Carnegie Mellon University, School of Computer Science, 2006. (Technical Report, CMU-CS- 06-180).

BURKARD, J. C**ódigos fonte em linguagem C**. Disponível em: http://people.sc.fsu.edu/~jburkardt/. Acesso em: jun. 2011.

BUTTLE, D. L. **Verification of compiled code**. 2001. Tesis (PhD of Phylosophy in Computer Science) - University of York, York UK, 2001.

CARMO L. et al. Rastreabilidade de códigos executáveis usando redes neurais. In: SIMPÓSIO BRASILEIRO EM SEGURANÇA DA INFORMAÇÃO E DE SISTEMAS COMPUTACIONAIS, 10., 2010, Fortaleza. **Anais ...** Fortaleza: SBC, 2010. p. 297-310. CDROM.

CIOCOIU, I. B. Hybrid feedforward neural networks for solving classification problems. **Neural Processing Letters**, Amsterdam, v. 16, n. 1, p. 81–91, 2002.

COLLBERG , C. ; NAGRA J. **Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection**. Reading: Addison Wesley, 2010.

CORTES, C. ; VAPNIK, V. Support-vector networks**. Machine Learning,** Boston, v. 20, n. 3, p. 273-297, 1995.

COUSOT, P. Abstract interpretation. **ACM Computing Surveys,** New York, v. 28, n. 2, p.324-328, 1996.

CHRISTODORESCU, M. ; JHA, S. Static analysis of executables to detect malicious patterns. In: USENIX SECURITY SYMPOSIUM, 12., 2003, Washington, DC. **Proceedings …** Washington, DC: Advanced Computing Systems Association, 2003.

CRUZ, A. J. O. **Códigos em linguagem C** .Disponível em: http://equipe.nce.ufrj.br/adriano/c/exemplos.htm. Acesso em: jun. 2011.

FLAKE, H. Structural comparison of executable objects. In: CONFERENCE ON ETECTION OF INTRUSIONS AND MALWARE & VULNERABILITY ASSESSMENT (DIMVA), 2004, Dortmund, Germany. **Proceedings …** Dortmund, Germany: IEEE.

GNU. **GCC, the GNU compiler collection.** Disponível em: http://gcc.gnu.org/. Acesso em: jul. 2011.

GOODMAN, J. E. ; O'ROURKE, J. **Handbook of discrete and computational geometry** 2. ed**.** Upper Saddle River**:** Chapman and Hall/CRC, 2004.

HASSAN, A. E. ; JIANG, Z. M. ; HOLT, R. C. Source versus object code extraction for recovering software architecture. In: WORKING CONFERENCE ON REVERSE ENGINEERING, 12., 2005, Pittsburgh. **Proceedings …** Pittsburgh: IEEE, 2005. P. 67–76.

HATTON, L. **Estimating source lines of code from object code:** windows and embedded control systems 2005. Disponível em: http://www.leshatton.org/Documents/LOC2005.pdf. Acesso em: jun. 2011

HAYKIN, S. **Neural networks:** a comprehensive foundation. Upper Saddle River: Prentice Hall, 1998

HERTZ, J. A. ; KROGH, A. S. ; PALMER, R. G. **Introduction to the theory of neural computation**. Reading: Addison-Wesley, 1991.

IDAPRO, **Ida pro – disassembler.** Disponível em: http://www.hex-rays.com/idapro/. Acesso em: jun. 2011.

INMETRO. **Portaria software Inmetro, portaria Inmetro nº 011 de 13 de Janeiro de 2009.** Rio de Janeiro: Instituto Nacional de Metrologia, Normalização e Qualidade Industrial, 2009.

LENIC, M. et al. Using cellular automata to predict reliability of modules. In: SOFTWARE ENGINEERING AND APPLICATIONS, 2004. Cambridge. **Proceedings …** Cambridge: MIT, 2004.

LINN, C. ; DEBRAY, S. Obfuscation of executable code to improve resistance to static disassembly. In: ACM Conference. on Computer and Communications Security, 1., 2003, Washington, DC. **Proceedings …** Washington, DC: ACM, 2003.

MA, R. P. ; TSAI, K., Artificial neuralnNetworks for distributed adaptive routing on dynamic topology networks**.** In: IJCNN INTERNATIONAL JOINT CONFERENCE ON NEURAL NETWORK, 1992, Baltimore. **Proceedings …** Baltimore: IEEE, 1992. v. 4, p. 468 – 473.

MCDONALD, J. **Delphi falls prey**. Disponível em: http://www.symantec.com/connect/blogs/delphi-falls-prey 2009. Acesso em: jun. 2011.

MEN, H. et al. Application of support vector machine to heterotrophic bacteria colony recognition. In: INTERNATIONAL CONFERENCE ON COMPUTER SCIENCE AND SOFTWARE ENGINEERING, 2008. Wuan, China. **Proceedings …** Wuan, China: IEEE, 2008. v. 1. p. 830–833.

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

MORETTI, E. ; CHANTEPERDRIX, G. ; OSORIO, A. New algorithms for controlflow graph structuring. In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING, 5., 2001. Lisboa. **Proceedings …** Lisboa: IEEE, 2001. p-184-187.

MURPHY, G. C. et al. An empirical study of static call graph extractors. **ACM Transactions on Software Engineering and Methodology**, New York, v. 7, n. 2, p 158–191, 1998.

NAGATOISHI, S. ; ARAKI, O. Effect of refractoriness on learning performance of a pattern sequence In: INTERNATIONAL JOINT CONFERENCE PUBLICATION ON NEURAL NETWORKS, 2009, Atlanta. **Proceedings …** Atlanta: IEEE, 2009.

NASCIMENTO, T. et al. **Program equivalence using neural networks**, In: INTERNATIONAL ICST CONFERENCE ON BIO-INSPIRED MODELS OF NETWORK, INFORMATION, AND COMPUTING SYSTEMS, 5., 2010, Boston. **Proceedings …** Boston: ACM, 2010.

NIELSON, F. ; NIELSON, H. R. ; HANKIN, C. **Principles of program analysis**, 2. ed. Berlin: Springer, 2005.

OH, J. Fight against 1-day exploits: diffing binaries vs anti-diffing binaries**.** In: BLACK HAT TECHNICAL SECURITY CONFERENCE, 2009, Las Vegas. **Proceedings …** Las Vegas: Black Hat, 2009.

POZNYAKOFF, S. **Gnu cflow**. Disponível em: http://savannah.gnu.org/projects/cflow. Acesso em: jun. 2011.

QUINLAN, D. ; PANAS, T. Source code and binary analysis of software defects. In: ANNUAL WORKSHOP ON CYBER SECURITY AND INFORMATION INTELLIGENCE RESEARCH, 5., 2009, Knoxville, TN, **Proceedings …** New York: ACM, 2009. p. 1–4.

REDDY, C. S. et al. Faultprone module prediction of a web application using artificial neural networks. In: SOFTWARE ENGINEERING AND APPLICATIONS, 2007, Cambridge, MA. **Proceedings …** Cambridge, MA: IAESTED, 2007. Track 591 - 149

STONE, M. Cross-validation **:** a review. **Mathematische Operationsforschung Statitischen, Serie Statitics,** Berlin, v.9, p. 127-139, 1978.

SUBRAMANIAN, S. ; COOK, J.V. Automatic verification of object code against source code computer assurance. In: COMPASS '96 - ANNUAL CONFERENCE ON COMPUTER ASSURANCE, 11., 1996, Gaithersburg, Maryland. **Proceedings …** Piscataway NJ: IEEE, 1996.

THOMPSON, K. Reflections on trusting trust. **Communications of the. ACM**, New York, v. 27, n. 8, p. 761-763, 1984.

VXHEAVENS, **VX heavens**. Disponível em: vx.netlux.org. Acesso em: jun. 2011.

WAHAB, M. **Object code verification**, 1998. Thesis (Doctored of Philosophy) - Department of Computer Science, University of Warwick, Coventry, UK, 1998.

WANG, Z. ; PIERCE, K. ; MCFARLING, S. Bmat - a binary matching tool for stale profile propagation. **The Journal of Instruction-Level Parallelism**, Raleigh, NC, v. 2, 2000.

ZENG, H. ; RINE, D. A neural network approach for software defects fix effort estimation. In: SOFTWARE ENGINEERING AND APPLICATIONS, 2004, Cambridge, MA. **Proceedings …** Cambridge, MA: IAESTED, 2004. Track 436 – 139.

ZHANG, Q. ; FENG, F. ; LIU F. Neurodynamic approach for generalized eigenvalue problems. In: INTERNATIONAL CONFERENCE ON COMPUTATIONAL INTELLIGENCE AND SECURITY, 2006, Guangzhou, China, **Proceedings …** Hong Kong: IEEE, 2006. p. 345-350.

ZHENGA, J. Predicting software reliability with neural network ensembles, **Expert Systems with Applications,** New York, v.36, n. 2, pt. 1, p. 2116-2122, Mar. 2009.

ZHENGA, J. A digital image encryption algorithm based on hyper-chaotic cellular neural network. **Journal Fundamenta Informaticae**, Warsaw, Poland, v. 90, n. 3, 2009.

APÊNDICE A - Traceability of Executable Codes Using Neural Networks

Traceability of Executable Codes using Neural Networks

Davidson R. Boccardo[†], Tiago M. Nascimento[†*], Raphael C. Machado[†], Charles B. Prado[†], and Luiz F. R. C. Carmo[†]

[†] INMETRO - National Institute of Metrology, Normalization and Industrial Quality
[*] UFRJ - Federal University of Rio de Janeiro
Rio de Janeiro - Brazil
{drboccardo, tmnascimento, rcmachado, cbprado, lfrust}@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 embedded in a meter corresponds to a version that was previously approved by the Legal Metrology Authority. In this paper, we propose a novel approach for correlating source and object codes using artificial neural networks. Our approach correlates the source code with the object 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 correspondence around 90% for the traceability of the executable codes with very low rate of false positives.

1 Introduction

In recent years, there has been a large increase in the number of embedded software based devices. These softwares bring a completely new dimension of validation approaches, requiring a better understanding of the new possibilities of errors, inaccuracies and security flaws. A critical risk associated with the presence of software in devices is the interest of individuals in tampering the software in order to obtain personal advantages. For example, in the automotive area, an attacker may want to reduce the mileage to obtain a higher resale value or increase the mileage to reduce the incoming tax; in commercial relations, the owner of a weighting machine could change the software in such a way that the new (wrong) measurements benefit him.

In several areas, the devices involved in human relations must be subject to some kind of "external control". One of such areas — which motivated the present work — is the area of Legal Metrology. The above example of commercial relation is a typical example of human relation on the Legal Metrology domain: it is important, for the sake of the fairness of the transactions between seller and buyer, to ensure that the weighting machine correctly exhibits the weight of the product being commercialized. Of course, the only way to ensure the correct

working of that weighting machine is to perform some kind of external verification: an entity non-committed with seller nor buyer evaluates the weighting machine and provides some kind of certification of its "good working". Typically, such certification entities are government or non-profit organizations.

Traditionally, quality certification processes involved several steps of "physical" tests of the device under evaluation. Such test should verify that the device works properly under several different conditions — for example, in distinct conditions of heat and humidity. With the advent of the embedded software devices, the certification process should involve, additionally, the validation of the software embedded in the devices. In our weighting machine example, the certification entity should guarantee, for example, that the software embedded in the weighting machine does not contain some backdoor that activates — under the command of the machine owner — some spurious function that decrease or increase the exhibited weight.

Binary code verification is the only true way to detect hidden capabilities, as demonstrated by Thompson in his Turing Award Lecture [1]. Lest Thompson's paper be considered theoretical, his ideas have been put into practice by the malware W32.Induc.A [2]. On large and complex systems, however, binary verification can require long and laborious work to integrally track variable manipulations and to perform vulnerability analysis, so that a usual approach is to conduct such verification on the source code. Depending on the architectural complexity of the embedded software, this software can be explicitly submitted to a white-box approach entailing a source code analysis. However, source code verification is not sufficient enough to give any guarantee about the behavior of the related embedded software in the device. To certify that the binary code being executed in some device works properly, one must guarantee that such binary code was, in fact, generated from the approved source code through a honest compilation process. The two validation approaches are depicted in Figure 1.

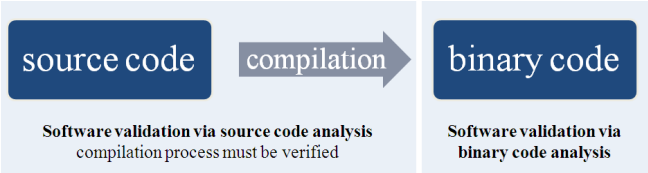


Fig. 1. Validation approaches: source code analysis with compilation verification × binary code analysis.

Source code verification does not preclude the verification as to whether the object code corresponds to its source code. Traceability of object codes is the process for establishing the correspondence between the evaluated source code and the object code embedded in the device. That is, once the source code of a given software version is analyzed, evaluated and approved, it is necessary to

verify whether a given binary code — which will be, in fact, in execution on the device — corresponds to that source code.

A simple and direct way of performing such "traceability" is to reproduce exactly the development environment which is used by the software developer and just compile the approved source, verifying whether the generated binary code is as expected. Such an approach, however, presents some disadvantages. The first drawback is related to the cost and the complexity required to keep several software development environments. A second drawback is related to Thompson Turing Award Lecture [1]: unless the "language transformation" performed by the compiler can be completely characterized — and this would require a binary analysis of the compiler code — it is not possible to guarantee that the compilation process, itself, does not introduce some kind of flaw or malicious behavior into the software.

Another way of performing the software traceability to which we refer is to audit the software development environment of the software developer. Such an approach presents disadvantages similar to those described in the previous paragraph, and it is likely to be ineffective when dealing with a malicious developer.

In the present work we propose a different strategy to verify whether two programs written in different languages (typically source code and binary machine code) describe the same software behavior. This paper presents a novel approach for performing traceability of object codes by using artificial neural network (ANN). More specifically, we collected properties of the source and object codes such as number of edges and number of nodes of the control flow graphs, number of functions of the call graph and the size of the codes (in bytes) to feed an ANN to discover the degree of similarity between source and object codes.

The rest of the paper is structured as follows. Section 2 discusses the related works on traceability. Section 3 describes our proposed method by characterizing the properties extraction of source and object codes and by showing the application of an artificial neural network to discover the degree of similarity between a source code and an object code based on the collected properties. Section 4 presents empirical evaluation of the method presented, followed by our concluding remarks.

2 Related works

There are not many contributions related with traceability of object codes in the literature. However, there were works for verifying and analyzing of source and object codes that may assist in achieving the proposed approach.

Quinlan *et al.* [3] proposed a framework for software defects verification (object or source). However, it does not compare source and object codes. Hassan *et al.* [4] observed that the architecture of some programs is intrinsically related with the their source and object codes. They used two types of extractors: a transfer control extractor of a code object (LDX) and a label extractor of a C source code (CTAGX). After the extraction process, they conducted a compari-

son of the obtained results in order to infer the software architecture. Hatton [5] investigated the defect density as a relationship between an object code and a source code. For so, he used the size (number of rows) of the source code and its defect per 1,000 lines to seek the relationship with the object code. Neither of these works addresses traceability of an object code nor are we aware of the existence of these works.

Buttle [6] utilizes the program logical structure (control flow graph) of the object code to match with the program logical structure of the source code. We also use program flow characteristics, however, we use other relationships that may coexist between source and object codes in order to obtain a better matching.

On a tangential direction there has been significant work in binary differing with the intent to review sequential versions of the same piece of software, to analyze malware variants of the same high-level language and to analyze security updates [7, 8]. Most of these works use graph matching comparison to compare the binaries. A good summary of these works may be found in [9], which also introduces anti-differing techniques with the intent to thwart algorithms based on graph matching. As far as we are concerned, there are not works tracking the traceability problem. However, research results from differing binaries [7–9], may thus be borrowed for the traceability problem for analogous constraints.

Some contributions in security use neural networks, for cryptography approaches. A new digital image encryption algorithm using neural networks is presented in [10]. Such algrithm employs a hyper-chaotic cellular neural network using chaotic characteristics of dynamical systems.

Artificial intelligence based methods for software validation, verification and reliability can be found on the literature. The approaches in in [11] and [12] propose the use neural networks for software reliability prediction. The former uses the prediction for software defects fix effort, while the second the prediction system is based on neural network ensembles. In [13], a neural network is used to predict a fault-prone module in a web application. In [14], a self-organizing system for reliability of modules is constructed.

3 Proposed approach

Our approach for software traceability involves two steps. In the first step, we use software analysis tools to extract some characteristics of both source code and binary code. Such characteristics could be simple ones, such as size, or more sophisticated ones, such as those derived from the control flow graphs or call graphs. In the second step, we use a nonlinear nondeterministic arbitrageur to determine the correspondence between source code and binary code. In this Section, we show how this approach was put in practice with the use of four characteristics (size, number of procedures of the call graph, number of vertices and edges of the control flow graph) and an artificial neural network as arbitrageur.

3.1 Extraction process

In the compilation process there is a lot of lost information that should be taken into account when designing traceability of object codes. The amount of available information of the compiled code is totally dependent on the compilation process. In the following, we mention some properties regardless of the source and object codes, which may be explicitly or implicitly available, in order to give insights about the complexity of designing a traceability method.

Considering the fact that most embedded softwares are written in imperative language, properties such as variable names, variable types and procedure names may be lost during the compilation process since the compiler goal is to maximize the performance. This process normally decreases the legibility of the object code, so these properties represent low confidence to use in our traceability problem.

Data contents are not explicitly available in the source and object codes. Nonetheless, these contents may be computed by data-flow analysis. The scope of the data-flow analysis for source and object is faintly different. In the source code, the scope is at the variable level, meanwhile, in the object it is at memory and registers level. This difference certainly increases the number of instructions contained into the object in comparison to the respective number of the source code. Besides being positive for compiler data optimizations since it tracks fine-grained transitions, it requires more memory to analyze more code lines. Since the scope is different, the mapping of source and object codes using data contents becomes complex, then it is also not suitable for our traceability problem.

The control sequentiality is certainly kept in the object, albeit, its tree of execution is not clearly structured as such in the source code. Preliminary algorithms to structure the control flow graph are needed, and for so, there are different algorithms in the literature concerned about this topic. A good summary of these works may be found in [15]. The control sequentiality describes the program logic of a certain code, and it can be characterized by call graphs and the individual function flow graphs. The call graphs show the caller-callee relationship. The individual function flow graphs represent the basic blocks and its flow of information based on conditional and unconditional branches.

The characteristics used in our traceability problem are based on the program logic of the code, so we collected the number of nodes, edges, functions, and the size of the codes (in bytes). The size of the codes was obtained in a straightforward manner. The number of nodes and edges were extracted from the control flow graphs obtained by using gcc debugger option "-fdump-tree-cfg" for the source codes and IDA disassembler [16] for the object codes. The number of the functions was obtained from the call graphs using GNU cflow [17] for the source codes and IDA debugger for the object codes.

3.2 Artificial neural network

Assuming that source code and object code are provided by manufacturers before a model approval process, the fundamental point is to establish an association

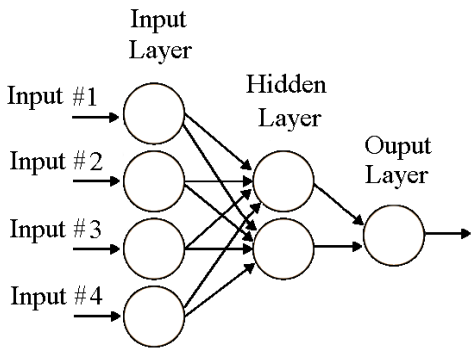


Fig. 2. Neural Network Topology.

between object code and source code by linking their intrinsic logical characteristics. For such, it is fundamental to find an efficient approach that combines the four different parameters extracted from logical program code (number of nodes, edges, functions, and the size of the codes (in bytes)).

As first analysis, some linear separation methods could be investigated to solve this problem. However, as will be showed in the future sections, this problem is both nonlinear and highly complex to solve it using a linear method. For these reasons, we examine the use of artificial neural networks.

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. Obviously, there are many kinds of neural networks applied to a large set of different problems, like: classification, recognition, prediction and others.

At this work, the main focus will be on neural networks applied to classification problem. For that, we propose the use of Cascade-Forward Backpropagation Neural Networks, since they are widely used in this context [18–20].

Neural Network Implementation. To design neural network, the four characteristics (number of nodes, edges, functions, and the size of the codes (in bytes)) were fed into the input nodes of the one fully-connected cascade forward network using the Backpropagation training procedure [21]. From the empirical analysis, the best neural configuration was built with two neurons in the hidden layer. For the output layer, only one neuron was used (see Figure 2).

In order to simulate the neural network, the Matlab Neural Network Toolbox [22] was used. The selected activation function for the neurons was the

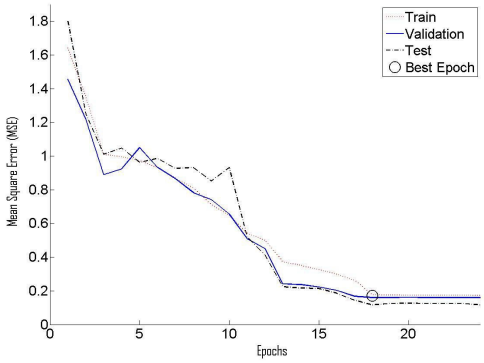


Fig. 3. MSE Convergence using Backpropagation Algorithm.

hyperbolic tangent sigmoid. The target vector for the training phase was defined by establishing a target value of 1 when the input parameters represent that object code corresponds to its source code (called class 0), otherwise the target value was set to -1 (called class 1).

For training process, 100 samples of both true association (class 0) and false association (class 1) between object code and its source code were used. For both classes, the percentage used to training phase was around 70% and the remaining samples were used to testing phase.

Figure 3 shows the convergence of the SME (Square Mean Error) during the training process, the point 17 represents the best epoch obtained (marked by circle). Table 1 shows the neural network results (testing phase) with respect to the number of hits and mistakes for both classes. Details of this experiment are further explained in the Section 4, which also includes a more refined experiment involving malware modified binaries.

Table 1. Neural Network Results

Class	# of hits	# of mistakes
0	27	3
1	30	0

4 Experimental evaluation

We now present the results of an empirical evaluation of object codes traceability. Our evaluation is divided in two parts: 1) selection of meaningful characteristics and 2) evaluation of malware modified binaries.

To perform the evaluation we collected one-hundred C source codes, taken from [23, 24] and compiled using gcc compiler. The implementation, done on the Matlab Neural Network Toolbox, was configured with the number of characteristics in the input layer, two neurons in the intermediate layer and one neuron in the output layer with the hyperbolic tangent sigmoid function to determine equivalence.

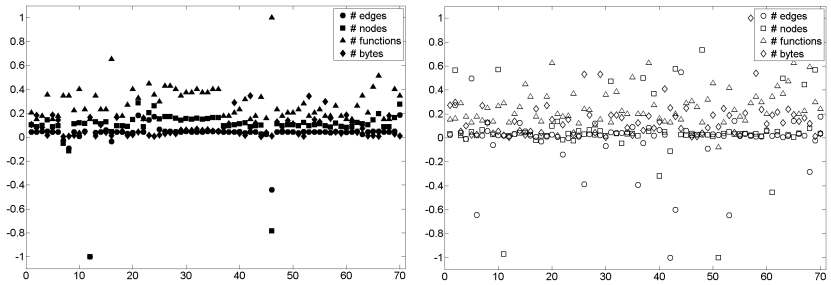
The control flow graph and the call graph parameters extracted from the source code and object code were correlated by subtracting the source code characteristic from the object code characteristic. The size characteristic was correlated by dividing the object code size by the source code size. Before inputing such parameters into the ANN, a normalization step was necessary since our ANN only accepts values in the interval [-1,1]. The normalization of the parameters was calculated by dividing all data of each parameter by the largest value of the same parameter. As usual, the training phase was realized with 70% of all data as shown in Figure 4, while the 30% left is used to check the model sensibility in face of a variable set of errors injection. The x-axis represents the programs and the y-axis their normalized characteristics. The characteristics of the true association samples, *i.e.*, corresponding source a object codes are represented by bolded symbols (circle, square, triangle, diamond) and the characteristics of the false association samples by unbolded symbols.

We studied the improvements of our method by comparing it against linear manual introspection. Any incidence of false positives is obviously a critical issue for software evaluation purposes. However, false negatives must be also treated as a major issue, since it can entail a reworking process of traceability. Our empirical evaluation shows that our method produces more precise results than linear manual introspection.

4.1 Selection of meaningful characteristics

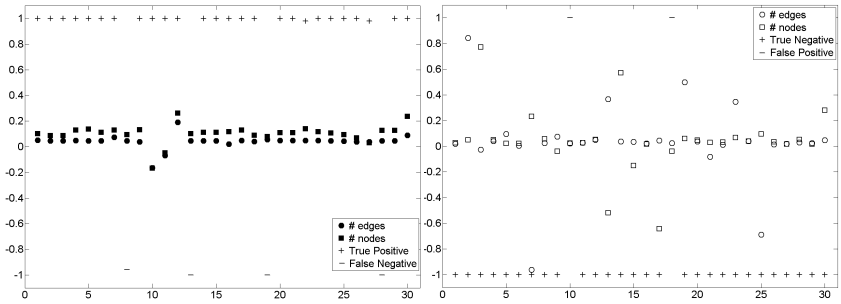
In order to verify meaningfulness of the set of input parameters from both source code and object code, we used an individual training strategy, verifying the subsequent system contribution of each parameter collected from the source code and object code. We considered four program characteristics related to size, call graph and control flow graph. Moreover, we trained and simulated the artificial neural network for all subsets of the set of these four characteristics, verifying that each of them are, in fact meaningful. In Figures 5(a) and 5(b) we show the best simulation results using two characteristics, while in Figures 6(a) and 6(b) we show the best simulation results using three characteristics. Figures 7(a) and 7(b) contain the best simulation results using all four characteristics.

We now present the comparison of the performance of our proposed approach by first starting with the control flow graph characteristics, followed by adding the call graph and the size characteristics. Figure 5(a) presents the answers of our approach from training and analyzing the control flow characteristics for a set of true pairs (source and object codes), and Figure 5(b) presents the answers for a set of false pairs. As we can notice in Figures 5(a) and 5(b), the data shows



(a) Positive set. (b) Negative set.

Fig. 4. Training of positive and negative sets.



(a) Positive set. (b) Negative set.

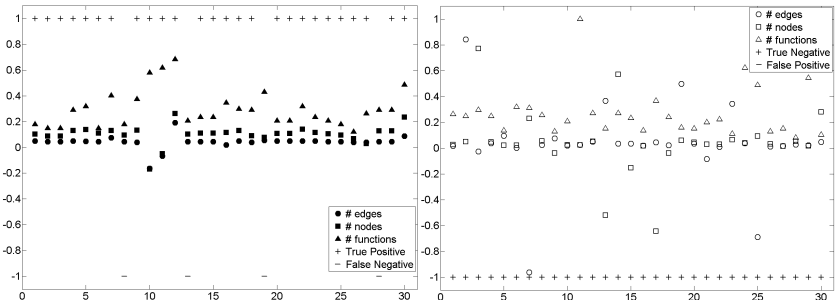
Fig. 5. Evaluation of the ANN using number of nodes and edges of the control flow graph.

that the neural network produces 4 of 30 (≈13%) false negatives and 2 of 30 (≈6%) false positives.

Figures 6(a) and 6(b) present the same evaluation by adding call graph characteristic into the procedure. The data shows that the extra input improved the neural network by reducing the false positives to zero. Finally, by adding the size in bytes as a characteristic, the evaluation resulted in less false negatives 3/30 (10%) in comparison with earlier evaluations 4/30 (≈13%), as shown in Figures 7(a) and 7(b). It is clear that the non linearity answer of the neural network is more precise than manual introspection. In the following, we present the results of the evaluation in the presence of the same set of programs, however, infected by the Win32.Cabanas.a malware.

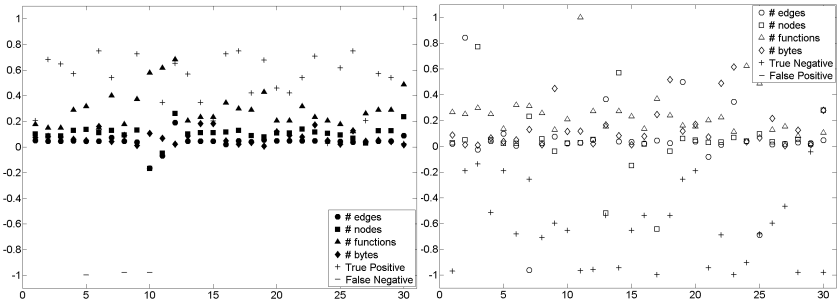
4.2 Evaluation of malware modified binaries

In order to perform this evaluation we setup a secure virtual laboratory system with Windows XP OS using Sun VirtualBox [25]. We carefully disconnected



(a) Positive set. (b) Negative set.

Fig. 6. Evaluation of the ANN using number of nodes and edges of the control flow graph, and number of functions of the call graph.



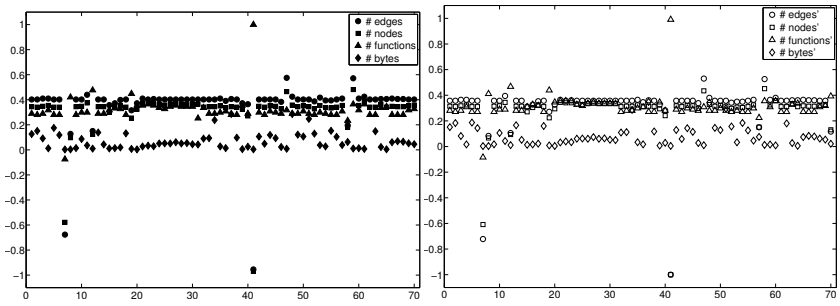
(a) Positive set. (b) Negative set.

Fig. 7. Evaluation of the ANN using number of nodes and edges of the control flow graph, number of functions of the call graph and the size of the codes.

all network connections before infecting the system with the Win32.Cabanas.a malware. After the system was infected we extracted the characteristics of the malware modified binaries. However, only 93 of 100 files were infected with the malware. We used 70 of these in the training phase (see Figure 8) and 23 in the evaluation (see Figure 9). Figures 9(a) and 9(b) present the evaluation of the ANN using malware modified binaries. The data shows that the hit rate of the ANN is 18/23 (≈78%) with 1/23 false positives (≈4%).

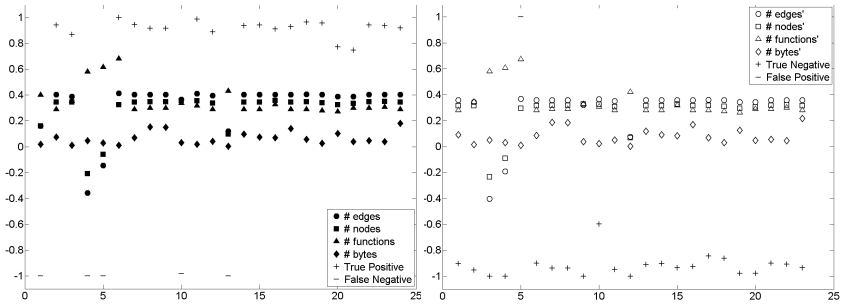
5 Conclusions

This paper deals with the issue of traceability of object codes. The problem is fundamental for software validation: since the software evaluation is frequently based on source code analysis, it is important to guarantee that the compilation process did not introduce security flaws, backdoors or unwanted behaviors.



(a) Positive set. (b) Negative set.

Fig. 8. Training set using Win32.Cabanas.a modified binaries.



(a) Positive set. (b) Negative set.

Fig. 9. Evaluation of the ANN using Win32.Cabanas.a modified binaries.

In the present work, we tackle the problem of traceability by extracting meaningful characteristics of source code and binary code, obtaining such characteristics from program call graphs and control flow graphs. We use such characteristics to feed a nondeterministic arbitrageur that decides whether a binary code corresponds to a given source code. The originality of the our approach lies on the extraction of characteristics of the call graphs and control flow graphs of both source code and binary code, and on the use of an artificial neural network to decide about the legitimacy of an object code based on those characteristics. The performance of the proposed approach is well characterized through an experimental evaluation that, besides confirming a very low rate of false positives (considered as a basic requirement), also provides a reasonable amount of false negatives.

It could be argued that the proposed approach would not work to detect very simple binary code modifications that do not alter call graphs and control flow graphs, such as the sole change of a constant. We observe, however, that such modification would be immediately noted by the conventional functional tests performed on devices under verification. Returning to the example of the

weighting machine given in Section 1, a simple doubling of a constant in the binary code could cause that device to double all exhibited weights. Such behavior would be immediately noticed by a simple test: put a standard weight of 1kg on the device and observe the indicated weight. The kind of modification our approach propose to counter are more subtle. For example, a malicious manufacturer could left a backdoor with password, and only when this backdoor is activated is that the sporious behavior — for example, doubling weight — occur. Such kind of malicious modification would hardly be noticed by functional tests such as the one previously described (the "stardard weight" test). However, it would not be possible to the manufacturer to include a backdoor without modifying the call graph and the control flow graph of the binary code. This make the malicious modifications exactly the ones ammenable to our approach based on call graph and control flow graph characteristics.

An open question in the proposed approach, and subject of ongoing research, concerns the necessity of also considering obfuscated codes during the training phase, to better understand its implications. For example, control-flow obfuscation alters the flow of control of the application by reordering statements, procedures, loops, obscuring flow of control using opaque predicates and replacing transfer flow instructions. Using such obfuscation some properties used in our approach may change, so violating our results. Other possible avenues of research involve merging our current technique with data-flow strategies to circumvent attacks such as modification of variable contents. Finally, in future works we plan to investigate which other graph invariants — crossing number, cycle covering, cromatic number etc. — are meaningful to establish the correspondence between software source codes and binary codes.

References

1. Thompson, K.: Reflections on trusting trust. Commun. ACM **27**(8) (1984) 761–763
2. McDonald, J.: Delphi falls prey. http://www.symantec.com/connect/blogs/delphi-falls-prey (Last accessed October 2009)
3. Quinlan, D., Panas, T.: Source code and binary analysis of software defects. In: CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research, New York, NY, USA, ACM (2009) 1–4
4. Hassan, A.E., Jiang, Z.M., Holt, R.C.: Source versus object code extraction for recovering software architecture. In: WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering, Washington, DC, USA, IEEE Computer Society (1995) 67–76
5. Hatton, L.: Estimating source lines of code from object code. In: Windows and Embedded Control Systems, 2005, at: www.leshatton.org/Documents/LOC2005.pdf. (2005)
6. Buttle, D.L.: Verification of Compiled Code. PhD thesis, University of York, UK (2001)
7. Wang, Z., Pierce, K., McFarling, S.: Bmat - a binary matching tool for stale profile propagation. In: The Journal of Instruction-Level Parallelism. (2002)
8. Flake, H.: Structural comparison of executable objects. In: Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), IEEE Computer Society (2004)

9. Oh, J.: Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In: Blackhat technical Security Conference. (2009)
10. Zhenga, J.: A digital image encryption algorithm based on hyper-chaotic cellular neural network. Journal Fundamenta Informaticae (2009)
11. Zeng, H., Rine, D.: A neural network approach for software defects fix effort estimation. In: IASTED Conf. on Software Engineering and Applications. (2004) 513–517
12. Zhenga, J.: Predicting software reliability with neural network ensembles. Expert Systems with Applications (36) (2007) 2116–2122
13. Reddy, C.S., Raju, K.V.S.V.N., Kumari, V.V., Devi, G.L.: Fault-prone module prediction of a web application using artificial neural networks. In: Proceeding (591) Software Engineering and Applications. (2007)
14. Lenic, M., Povalej, P., Kokol, P., Cardoso, A.: Using cellular automata to predict reliability of modules. In: Proceeding (436) Software Engineering and Applications. (2004)
15. Moretti, E., Chanteperdrix, G., Osorio, A.: New algorithms for control-flow graph structuring. In: CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, Washington, DC, USA, IEEE Computer Society (2001) 184
16. IdaPro: Ida pro - disassembler. http://www.hex-rays.com/idapro/ (Last accessed January 2010)
17. Poznyakoff, S.: Gnu cflow. http://savannah.gnu.org/projects/cflow (Last accessed January 2010)
18. Ciocoiu, I.B.: Hybrid feedforward neural networks for solving classification problems. Neural Processing Letters. **16**(1) (2002) 81–91
19. Asadi, R., Mustapha, N., Sulaiman, N.: New supervisioned multi layer feed forward neural network model to accelerate classification with high accuracy. European Journal of Scientific Research. **33**(1) (2009) 163–178
20. Haykin, s.: Neural Networks: A Comprehensive Foundation. Prentice Hall (1998)
21. Hertz, J.A., Krogh, A.S., Palmer, R.G.: Introduction to the Theory of Neural Computation. Addison-Wesley, Redwood City, CA, USA (1991)
22. Moler, C.B.: MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science (1980)
23. Burkard, J.: C software. http://people.sc.fsu.edu/ burkardt/ (Last accessed January 2010)
24. Oliveira Cruz, A.J.: C software. http://equipe.nce.ufrj.br/adriano/c/exemplos.htm (Last accessed January 2010)
25. Microsystems, S.: Virtualbox. http://www.virtualbox.org (Last accessed January 2010)

APÊNDICE B - Rastreabilidade de Códigos Executáveis usando Redes Neurais

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].

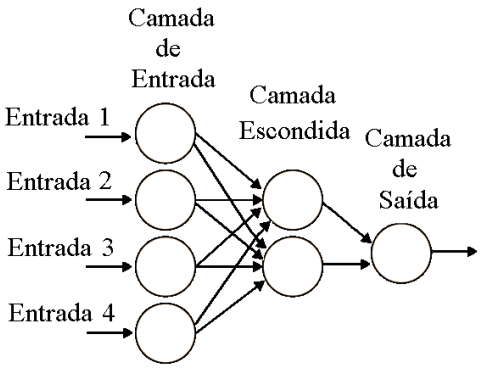


Figura 2. Topologia de rede neural.

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

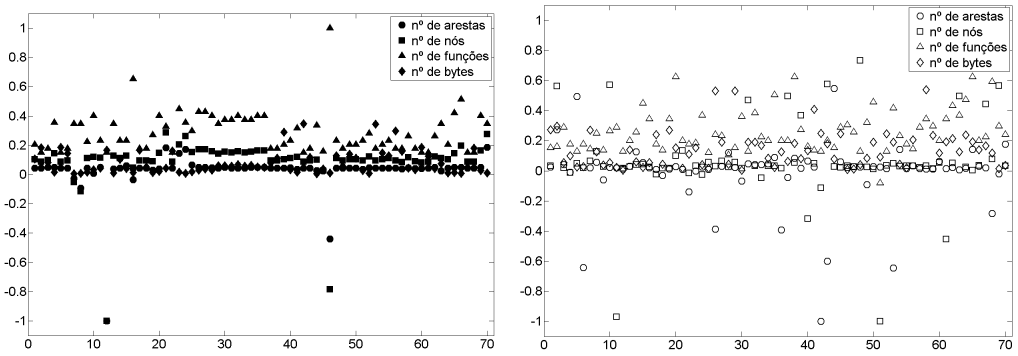
As redes neurais, com suas capacidades de extração de significado em dados complexos ou imprecisos, podem ser usadas para extrair padrões e detectar tendências, que são complexas para serem notadas manualmente ou por técnicas computacionais. Assim, uma rede neural treinada pode ser caracterizada como "*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.

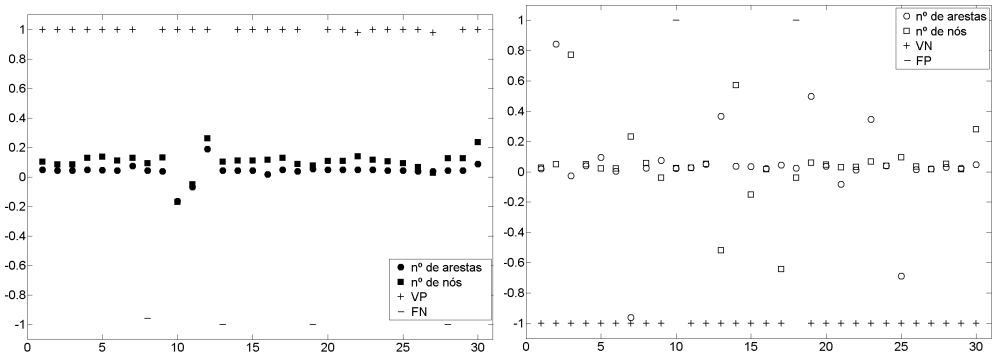
Para o primeiro processo de treinamento, 100 amostras para ambas associações: verdadeira (classe 0) e falsa (classe 1) entre o código executável e seu código fonte foram utilizadas. Para ambas as classes, o percentual usado para fase de treinamento foi de 70% e as amostras restantes foram utilizadas 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



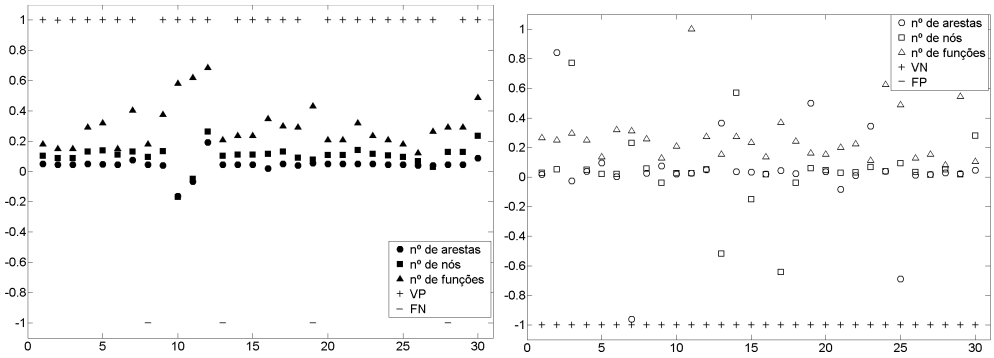
(a) Conjunto verdadeiro. (b) Conjunto falso.

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.

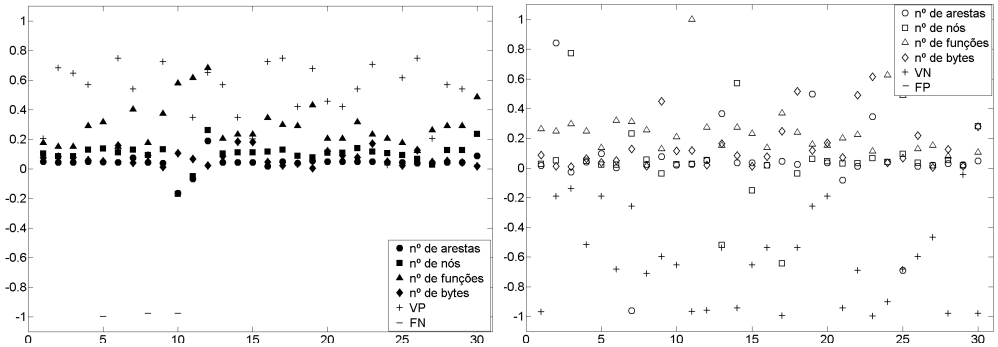
Primeiramente, para um conjunto de dois parâmetros, os mais relevantes foram os extraídos do grafo de controle de fluxo, ou seja, número de nós e de arestas de cada função do programa. Para três parâmetros, a melhor alternativa foi a adição do número de funções extraído dos grafos de chamadas, e por 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 (≈13%) falsos negativos (FN) e 26/30 (≈87%) verdadeiros positivos (VP), e a Figura 4(b) exibe 2 de 30 (≈6%) falsos positivos (FP) e 28/30 (≈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 (≈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.



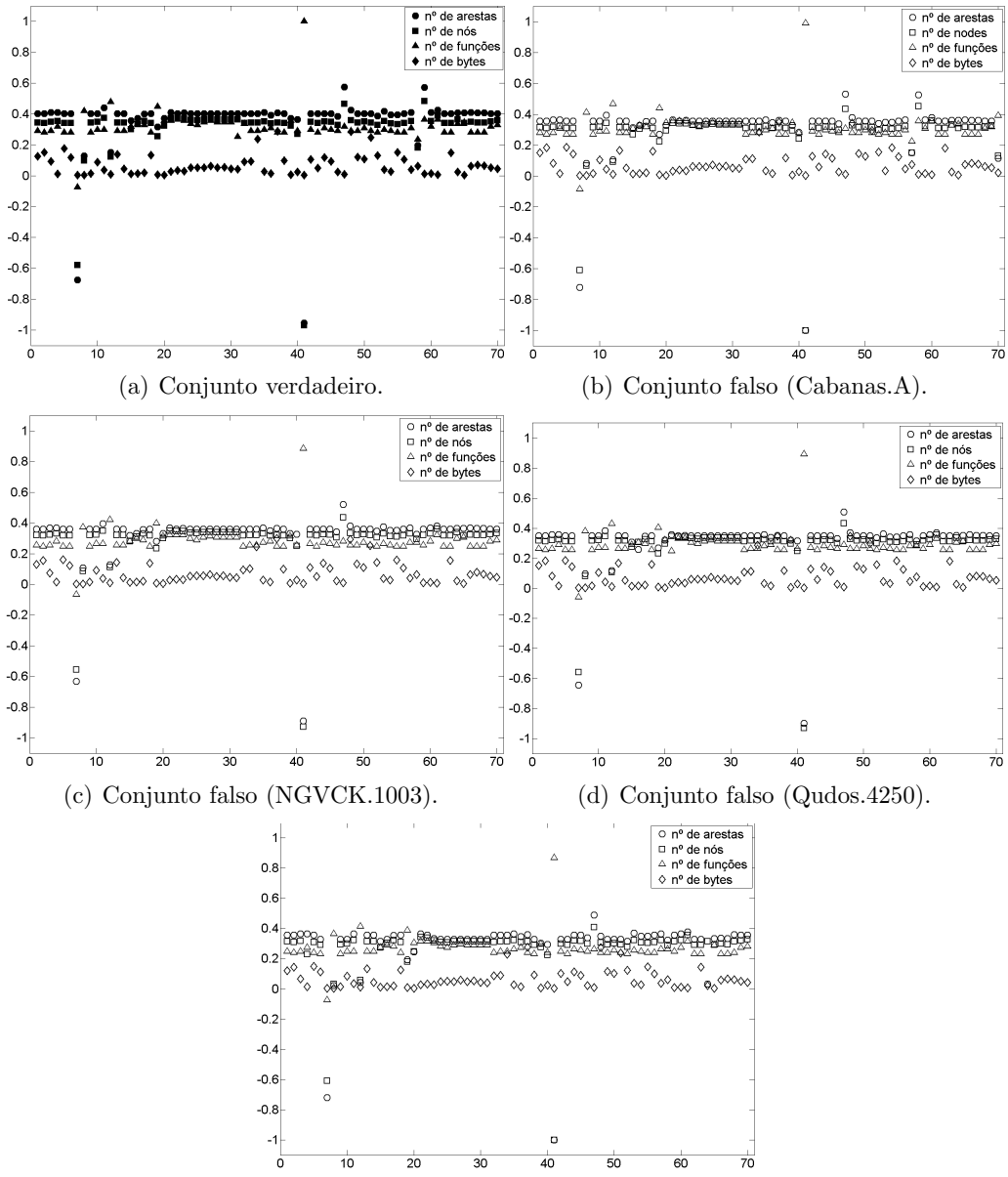
(a) Conjunto verdadeiro. (b) Conjunto falso.

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).

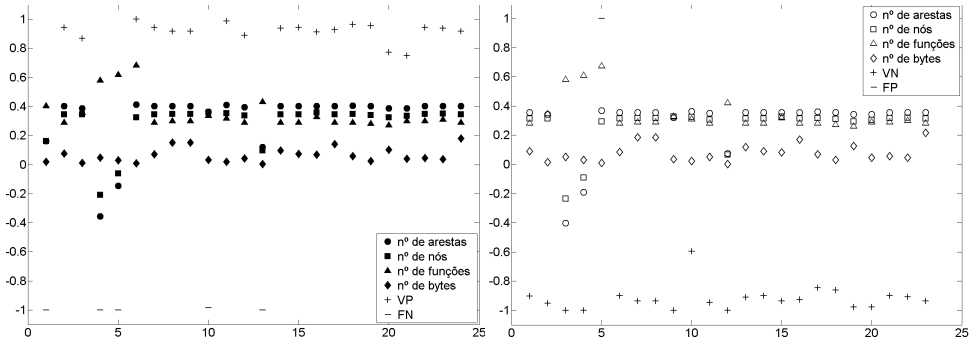
(d) Conjunto falso (Qudos.4250).

(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.

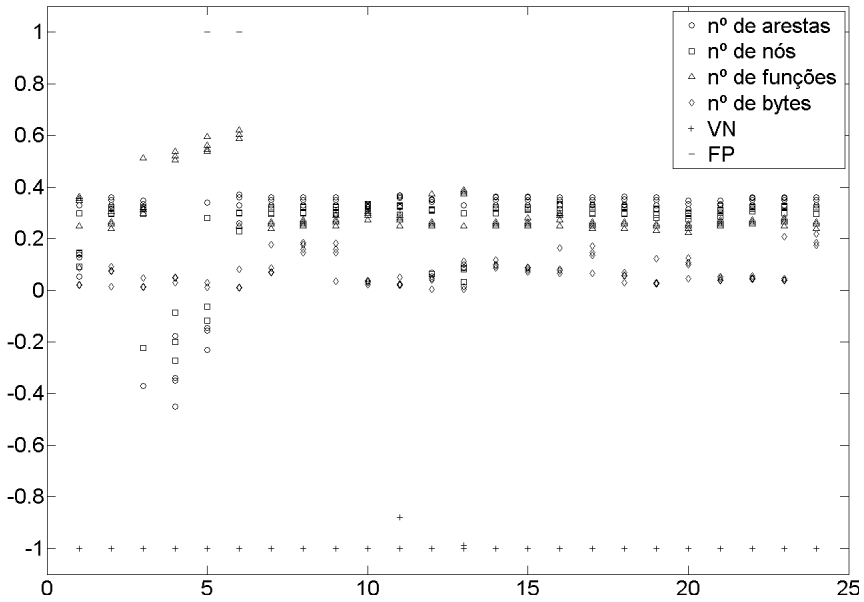


Figura 9. Avaliação da RNA diante de códigos infectados.

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*.

APÊNDICE C - Program Equivalence using Neural Networks

Program Equivalence using Neural Networks

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

[1] INMETRO - National Institute of Metrology, Normalization and Industrial Quality
Rio de Janeiro - Brazil
[2] UFRJ - Federal University of Rio de Janeiro - Brazil
{tmnascimento,cbprado,drboccardo,lfrust,rcmachado}@inmetro.gov.br

Abstract. Program equivalence refers to the mapping between equivalent codes written in different languages – including high-level and low-level languages. In the present work, we propose a novel approach for correlating program codes of different languages using artificial neural networks and program characteristics derived from control flow graphs and call graphs. Our approach correlates the program codes of different languages by feeding the neural network with logical flow characteristics. Our evaluation using real code examples shows a typical correspondence rate between 62% and 100% with the very low rate of 4% false positives.

1 Introduction

Equivalence between two programs can be characterized by their executing behavior. The behavior equivalence can be established between distinct languages – including high-level and low-level languages. The *equivalence problem* refers to, given two programs, deciding if they present the same executing behavior or not. In the following, we present some scenarios of Software Engineering concerned with the equivalence problem.

1. **Platform migration**. Consider a developer who, for historic reasons, has a small part of a certain system written in a distinct programming language from the rest of the system. For maintainability reasons, this developer may track the homogeneity of the system by rewriting the routines in the predominant language of the system. One way to determine whether the "rewritten routines" were properly coded is to verify if they are equivalent to the "older routines".
2. **Legacy software recovering**. Consider a developer who has a software system in production that, due to difficulties in its software configuration management, needs to "recover the baseline" of the software system in production, *i.e.*, for each software module that has a binary code in execution, the developer must identify among several versions of source code, which corresponds to that binary in execution. One possibility for this task could be recompiling the distinct source code modules, comparing the generated binaries with the ones in the production environment. This approach however, besides being impractical – given the huge amount of versions and

compilations that should be conducted, still has the possibility of failure in the case that the setting parameters are not exactly the same as those used in the compilation of the modules currently in the production.

3. **Introduction of non-intentioned behavior in the compilation process**. This scenario refers to the fact that the compilation process is a part of the software development and must be validated. In fact, the compilation process may introduce *bugs* and non-intentioned behavior in the software [1]. Besides, the software code and the compiler may be deliberately corrupted by the insertion of a malicious behavior (backdoor). Hence, it becomes interesting to have tools to directly compare the behavior described by source and binary code, independently of the compilation process.
4. **Software acquisition management**. Assume that a software manufacturer wishes to outsourcing the development of some libraries in a project. For so, the manufacturer gives some specifications to an independent developer, which returns the specified product developed. Naturally, such a product should be submitted to a battery of tests in order to characterize it according to the specifications. However, these tests normally are ineffective with hidden malicious behavior. Such a behavior is typically activated by the insertion of hidden undocumented commands — for instance, by using a counterintuitive sequence of keystrokes[1]. The detection of hidden behavior only can be done through of code analysis. In this case, the manufacturer may require not only the binary code but also the associated source code. Once the source code be analyzed, further step requires the mapping (equivalence) between the source and the binary code.

In the previous examples, we see that frequently the equivalence problem refers to the mapping of a binary code from a source code or of a binary code from another binary code, which we term "traceability". A simple and direct way of performing such "traceability" is to reproduce exactly the development environment of the software developer and to compile the source code, verifying whether the generated binary code is as expected. For this approach we must assure that the compilation environment is the same. Such a hypothesis, however, may be impractical — as the case (2), in which the compilation settings may be lost due to flaws in the configuration management, or as the case (4), which would restrict the developer to a unique environment. Another drawback is related to the cost and the complexity required to keep several software development environments. Moreover, as demonstrated in the Thompson Turing Award Lecture [1]: unless the "language transformation" performed by the compiler can be completely characterized — which would require a binary analysis of the compiler code — it is not possible to guarantee that the compilation process, itself, does not introduce some kind of flaw or malicious behavior into the software (this is closely related to the example in case (3)). Another way of performing the software traceability is to audit the software development environment of the software developer. Such an approach presents disadvantages

[1] These sequences of commands are sometimes called *easter eggs*.

similar to those described in the previous paragraph, and it is likely to be ineffective when dealing with a malicious developer.

Summarizing, binary code verification is the only true way to detect hidden capabilities, as demonstrated by Thompson in his Turing Award Lecture [1]. Lest Thompson's paper be considered theoretical, his ideas have been put into practice by the malware W32.Induc.A [2]. On large and complex systems, however, binary verification can require long and laborious work to integrally track variable manipulations and to perform vulnerability analysis, so that a usual approach is to conduct such verification on the source code. Depending on the architectural complexity of the software, this software can be explicitly submitted to a white-box approach entailing a source code analysis. However, source code verification is not sufficient enough to give any guarantee about the behavior of the related binary code. To certify that the binary code being executed works properly, one must guarantee that such binary code was, in fact, generated from the approved source code through an honest compilation process.

Source code verification does not preclude the verification as to whether the binary code corresponds to its source code. Traceability of executable codes is the process for establishing the correspondence between source and object codes. That is, once the source code of a given software version is analyzed, evaluated and approved, it is necessary to verify whether a given binary code — which will be, in fact, in execution — corresponds to that source code.

In the present work we propose a different strategy to verify whether two programs of different languages (typically source code and binary machine code) describe the same software behavior. This paper presents a novel approach for performing program equivalence of program codes of different languages by using artificial neural network (ANN). More specifically, we collected properties of the control flow graphs and call graph, such as number of edges, number of nodes and number of functions, and we used an ANN to discover the degree of similarity of the program languages based on the collected properties.

The rest of the paper is structured as follows. Section 2 discusses the related works on program equivalence. Section 3 describes our proposed method by characterizing the properties extraction of program languages and by showing the application of an artificial neural network to discover the degree of similarity of the program languages. Section 4 presents the empirical evaluation of the method presented, followed by our concluding remarks.

2 Related Works

There are not many contributions related with program equivalence in the literature. However, there were works for verifying and analyzing of source and binary codes that may assist in achieving the proposed approach.

Quinlan *et al.* [3] proposed a framework for software defects verification (binary or source). However, it does not compare source and binary codes. Hassan *et al.* [4] observed that the architecture of some programs is intrinsically related with the their source and binary codes. They used two types of extractors: a

transfer control extractor of a code binary (LDX) and a label extractor of a C source code (CTAGX). After the extraction process, they conducted a comparison of the obtained results in order to infer the software architecture. Hatton [5] investigated the defect density as a relationship between a binary code and a source code. For so, he used the size (number of rows) of the source code and its defect per 1,000 lines to seek the relationship with the binary code. Neither of these works addresses the program equivalence problem.

Buttle [6] utilizes the program logical structure (control flow graph) of the binary code to match with the program logical structure of the source code. We also use program flow characteristics to match binary codes, however, in our approach other relationships that may coexist between source and binary codes in order to obtain a better matching were considered.

On a tangential direction there has been significant work in binary differing with the intent to review sequential versions of the same piece of software, to analyze malware variants of the same high-level language and to analyze security updates [7, 8]. Most of these works use graph matching to compare the binaries. A good summary of these works may be found in [9], which also introduces anti-differing techniques with the intent to thwart algorithms based on graph matching. Research results from differing binaries [7, 8, 9], may thus be borrowed for the program equivalence problem for analogous constraints.

Some contributions in security use neural networks for cryptography approaches. A new digital image encryption algorithm using neural networks is presented in [10]. Such algorithm employs a hyper-chaotic cellular neural network using chaotic characteristics of dynamic systems.

Artificial intelligence based methods for software validation, verification and reliability can be found on the literature. The approaches in [11] and [12] propose the use neural networks for software reliability prediction. The former uses the prediction for software defects fix effort, while the second the prediction system is based on neural network ensembles. In [13], a neural network is used to predict a fault-prone module in a web application. In [14], a self-organizing system for reliability of modules is constructed.

The use of artificial neural networks was previously considered in [15], which described a method for verifying the correspondence between source and binary codes using artificial neural networks. The approach described in this paper improves upon that work by refining the extracted properties and by applying the method for program equivalence of distinct languages.

3 Proposed approach

Our approach for program equivalence involves two steps. In the first step, we use software tools to extract characteristics of distinct program languages. Such characteristics could be simple ones, such as size, or more sophisticated ones, such as those derived from the control flow graphs or call graphs. In the second step, we use a nonlinear nondeterministic classifier to determine the correspondence of the program languages. In this section, we show how this approach was put in

practice with the use of four characteristics (size, number of procedures, number of nodes of control flow graph and number of edges of control flow graph) and an artificial neural network as classifier.

3.1 Extraction process

In the compilation process there is a lot of lost information that should be taken into account when designing a program equivalence approach. The amount of available information of the compiled code is platform and compiler-specific. In the following, we mention some properties regardless of the source and binary codes, which may be explicitly or implicitly available, in order to give insights about the complexity of designing a program equivalence method.

Considering the fact that most embedded softwares are written in imperative language, properties such as variable names, variable types and procedure names may be lost during the compilation process since the compiler goal is to maximize the performance. This process normally decreases the legibility of the binary code, so representing low confidence for the mentioned properties to use in our program equivalence problem.

Data contents are not explicitly available in the source and binary codes. Nonetheless, these contents may be computed by data-flow analysis. The scope of the data-flow analysis for source and binary is faintly different. In the source code, the scope is at the variable level, meanwhile, in the binary code it is at memory and registers. This difference certainly increases the number of instructions contained in the binary in comparison to the respective number of the source code. Besides being positive for compiler data optimizations since it tracks fine-grained transitions, it requires more memory to analyze more code lines. Since the extraction of this property is complex, it is not suitable for our program equivalence problem.

The control sequentiality is certainly kept in the binary, albeit, its tree of execution is not clearly structured as such in the source code. A good summary of algorithms to structure the control sequentiality may be found in [16]. The control sequentiality describes the program logic of a certain code, and it can be characterized by call graphs and the individual function flow graphs. The call graphs show the caller-callee relationship. The individual function flow graphs represent the basic blocks and its flow of information based on conditional and unconditional branches.

The characteristics used in our program equivalence method are based on the program logic of the code (control sequentiality). These characteristics are number of nodes, edges and functions. The sizes in bytes of the codes were also used in our method. The sizes of the codes were obtained in a straightforward manner, however, the number of nodes and edges of the control flow graph for the source code were obtained by using a shell script that counted these properties. Such a control flow graph of the source code was built from the Gnu Compiler Collection with the parameter “fdump-tree-cfg”. For the extraction of the same characteristics from the control flow graph in the binary code, we used an idapython script over the IDA disassembler [17]. The number of the

functions for the source codes and binary codes were extracted from the call graphs, generated by the GNU cflow [18] tool for the source code and by the IDA disassembler for the binary code.

3.2 Artificial neural network

The fundamental point in program equivalence is to establish an association of the program languages by linking their intrinsic logical characteristics. For such, it is fundamental to find an efficient approach that combines the four different parameters extracted from logical program code (number of nodes, edges, functions, and the sizes of the codes (in bytes)).

At first analysis, some linear separation methods could be investigated to solve this problem. However, as will be shown in the further sections, this problem is both nonlinear and highly complex to solve using a linear method. For these reasons, we examine the use of artificial neural networks.

Neural networks, with their remarkable ability to derive meaning from complicated or imprecise data, can be used to extract patterns and detect trends that are too complex to be noticed by either humans or other computer techniques. A trained neural network can be thought of as an "expert" in the category of information it has been given to analyze. Obviously, there are many kinds of Back-propagation networks applied to a large set of different problems, like: classification, recognition, prediction and others.

In this work, the main focus will be on neural networks applied to the classification problem. For that, we propose the use of Cascade-Forward Back-propagation Neural Networks, since they are widely used in this context [19, 20, 21]. In the next section, the details of neural network implementation will be presented.

Neural Network Implementation. To design a neural network, four characteristics (number of nodes, edges, functions, and the sizes of the codes (in bytes)) were fed into the input nodes of the one fully-connected cascade forward network using the Back-propagation training procedure [22]. From the empirical analysis, the best neural configuration was built with two neurons in the hidden layer. For the output layer, only one neuron was used (see Figure 1).

In order to simulate the neural network, the Matlab Neural Network Toolbox [23] was used. The selected activation function for the neurons was the hyperbolic tangent sigmoid. The target vector for the training phase was defined by establishing a target value of 1 when the input parameters represent equivalent codes (called class of true association), otherwise the target value was set to -1 (called class of false association).

4 Experimental evaluation

We now present the results of an empirical evaluation for mapping equivalent codes written in distinct languages. Three evaluations were performed: 1) evalua-

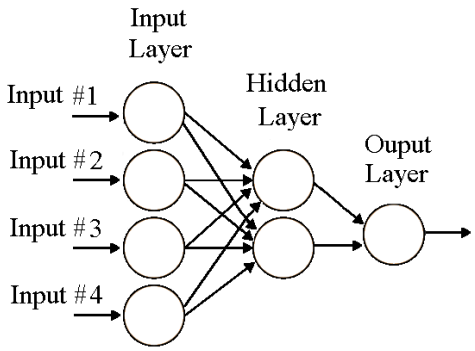


Fig. 1. Neural Network Topology.

tion of malware modified binaries, 2) evaluation of binaries of different platforms and 3) evaluation of binaries generated from different compilers.

For all the evaluations, the percentage used for training phase was ≈70% and the remaining samples were used to verify the capacity of generalization of the ANN. We studied the improvements of our method by comparing it against Support Vector Machine (SVM) — a technique typically applied to the construction of classifiers [24, 25]. Our empirical evaluation shows that our method produces more precise results than SVM.

4.1 Evaluation of malware modified binaries

Since most malwares are predominant for Windows environment, we performed the evaluation in this environment. Before starting to infect the system in order to extract the characteristics of the infected codes, we established some security policies to avoid malware dissemination. We setup an isolated machine for the malware infection and characteristics extraction. The extracted characteristics of the infected codes were used to build the set of false association.

For this evaluation, we collected 94 source codes, taken from [26, 27] and compiled them using the gcc compiler of Windows platform. Figure 2 shows the training set of true association, *i.e.*, characteristics extracted from 70 pairs of correlated source and binary codes. The 24 remaining were utilized in the evaluation phase. The histograms show the distribution of the normalized variables: edges, nodes, functions and size, respectively, in the x-axis and their frequency in the y-axis. The control flow graph characteristics (edges and nodes) were fed into input #1 and input #2 of the ANN, respectively. The number of functions (extracted from the call graph) was fed into input #3, and the size into input #4. All the characteristics were correlated by subtracting the source code characteristic from the binary code characteristic except for the size characteristic,

in which a division was applied. Before inputing such parameters into the ANN, a normalization step was necessary since our ANN only accepts values in the interval [-1,1]. The normalization of the parameters was calculated by dividing all data of each parameter by the largest value of the same parameter.

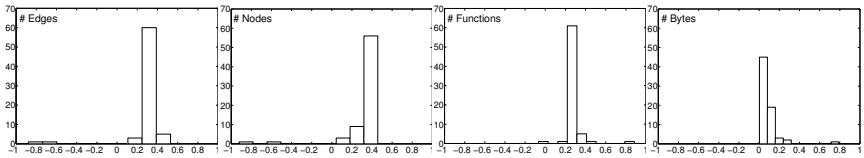


Fig. 2. Training set of the true association using correlated binary and source codes.

The false association was created by infecting the 94 codes with four malwares (Virus.Cabanas.a, Virus.Win32.NGVCK.1003, Virus.Win32.Qudos.4250, Virus.Win32.Artelad.2173). Figure 3 shows the training set of false association using characteristics extracted from 280 infected samples (70 for each malware), in which the characteristics were correlated by subtracting the source code characteristic from the infected binary code except for the size characteristic, in which a division was applied. All the characteristics were normalized before inputing into the ANN. The 96 remaining infected samples were utilized in the evaluation phase.

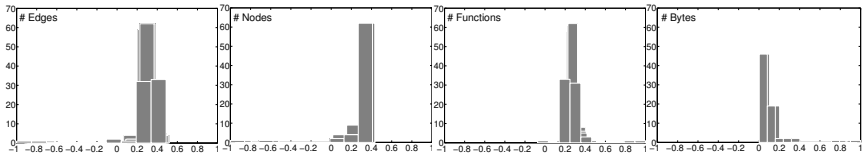


Fig. 3. Training set of the false association using malware modified binaries.

Figures 4 and 5 show the sets utilized for the evaluation of the ANN. Table 1 shows the neural network results with respect to the number of hits and mistakes for both classes (true and false association). The data shows ≈4% (4/96) false positives and ≈37% (9/24) false negatives. The results of our approach are promising since the program equivalence is mainly concerned with a low rate of false positives.

Table 1. Neural network results of the evaluation of malware modified binaries

Class	True association	False association
True association	15	9
False association	4	92

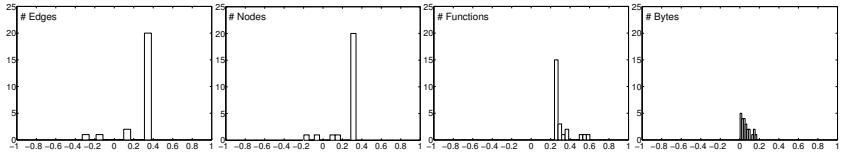


Fig. 4. Evaluation of the ANN using correlated binary and source codes.

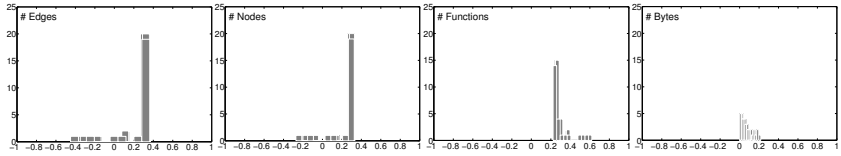


Fig. 5. Evaluation of the ANN using non-correlated binary and source codes.

We compare the results of our proposed ANN with the results achieved using a Support Vector Machine (SVM). The classifier used was a two-norm and soft-margin SVM. The kernel function that maps the training data into kernel space was linear and the method to find the separating hyperplane was quadratic programming. However, the SVM did not converge to any separating hyperplane for the training sets exhibited in Figures 2 and 3. However, using only one malware (Virus.Cabanas.a) for the false association we found a division of the 4d-space — that is, a 3-d hyperplane that divides the 4-d space into two semi-spaces. In this experiment, the SVM correctly identified 19 among the 24 pairs of corresponding codes, with 7/23 (≈30%) ratio of false positives. Observe that the SVM method achieved a reasonable accuracy regarding the identification of corresponding codes (exactly the same 19/24 as the ANN), but with a much higher number of false positives (7/23 against the 1/24 ratio of the ANN).

4.2 Evaluation of binaries of different platforms

For this evaluation, we compiled 81 source codes using both gcc compiler on the Windows environment and gcc compiler on a Linux-like environment to build the true association. Figure 6 shows the training set of true association, *i.e.*, characteristics extracted from 60 pairs of correlated Windows binary and Linux binary. The 21 remaining were utilized in the evaluation phase. The normalization and extraction processes are analogous to the ones of the previous evaluation.

Figure 7 shows the training set for the false association, created naively by random values. Figures 8 and 9 show the sets utilized for the evaluation of the ANN. Table 2 shows the neural network results with respect to the number of hits and mistakes for both classes (true and false association). The data shows ≈9% (2/21) false positives and zero false negatives. We used the same training sets exhibited in Figures 6 and 7 to feed the SVM. The SVM correctly identified all the 21 pairs of corresponding codes as the results of our ANN, with 3/21 (≈14%) ratio of false positives against 2/21 ratio of our ANN approach.

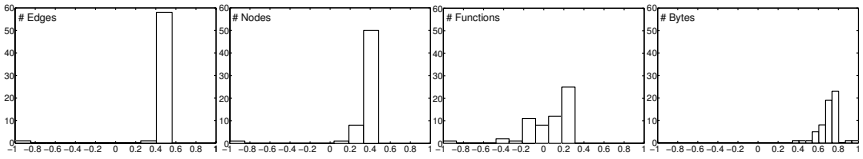


Fig. 6. Training set of true association using correlated binaries of different platforms.

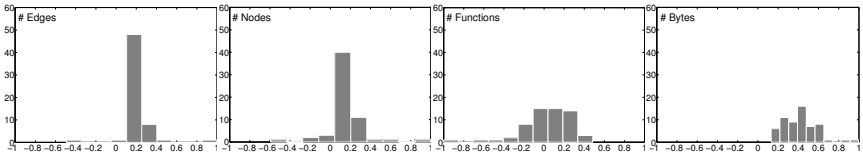


Fig. 7. Training set of the false association using non-correlated binaries of different platforms.

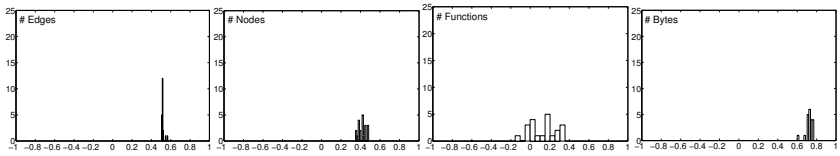


Fig. 8. Evaluation of the ANN using correlated binaries of different platforms.

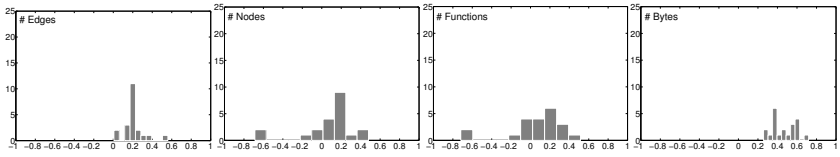


Fig. 9. Evaluation of the ANN using non-correlated binaries of different platforms.

Table 2. Neural network results of the evaluation of binaries of different platforms

Class	True association	False association
True association	21	0
False association	2	19

4.3 Evaluation of binaries generated from different compilers.

For this evaluation, we compiled 83 source codes using the gcc compiler and the Borland C++ compiler, both on the Windows environment. Figure 10 shows the training set of true association, *i.e.*, characteristics extracted from 63 pairs of correlated binary generated from the compilers above. The 20 remaining were utilized in the evaluation phase. The normalization and extraction processes are analogous to the ones of the previous evaluation.

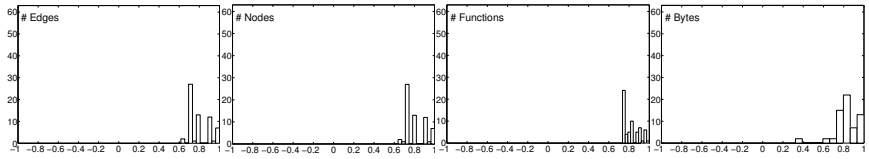


Fig. 10. Training set using correlated binaries generated by different compilers.

Figure 11 shows the training set for the false association, created naively by random values. Figures 12 and 13 show the sets utilized for the evaluation of the ANN. Table 3 shows the neural network results with respect to the number of hits and mistakes for both classes (true and false association). The data shows 5% (1/20) false positives and 5% (1/20) false negatives. We used the training sets exhibited in Figures 10 and 11 to feed the SVM. The SVM correctly identified 18/20 pairs of corresponding codes against 19/20 of our ANN, with 10/20 (50%) ratio of false positives against 1/20 ratio of our ANN approach.

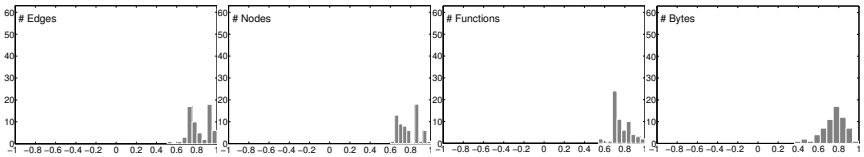


Fig. 11. Training set using different non-correlated compiled binaries.

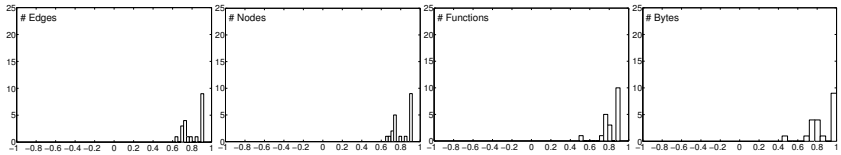


Fig. 12. Evaluation of the ANN using correlated binaries generated by different compilers.

Table 3. Neural network results of the evaluation of binaries compiled with distinct compilers.

Class	True association	False association
True association	19	1
False association	1	19

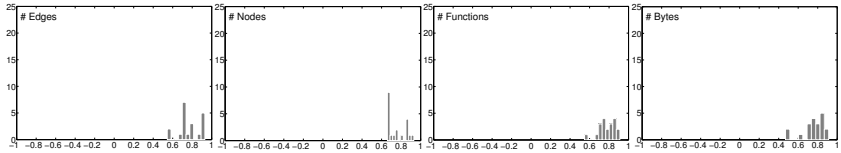


Fig. 13. Evaluation of the ANN using non-correlated binaries generated by different compilers.

5 Conclusions

This paper deals with the issue of program equivalence of different languages — including high-level and low-level languages. The problem is fundamental for software validation: since the software evaluation is frequently based on source code analysis, it is important to guarantee that the compilation process did not introduce security flaws, backdoors or unwanted behaviors.

In the present work, we tackle the problem of program equivalence by extracting meaningful characteristics of program codes, obtaining such characteristics from program call graphs and control flow graphs. We use such characteristics to feed a nondeterministic classifier that decides whether a binary code corresponds to a given source code or whether a binary corresponds to another binary of a different platform or compiler. The originality of our approach lies on the extraction of characteristics of the call graphs and control flow graphs of the program codes, and on the use of an artificial neural network to decide the legitimacy of a binary code based on those characteristics. The performance of the proposed approach is well characterized through an experimental evaluation that, besides confirming a very low rate of false positives (considered as a basic requirement), also provides a reasonable amount of false negatives.

It could be argued that the proposed approach would not work in detecting simple binary code modifications that do not alter call graphs and control flow graphs, such as a change of a constant. We observe, however, that such modification would be immediately noted by the conventional functional tests performed on devices under verification. The kind of modification that our approach proposes to encounter is more subtle. For example, a malicious manufacturer could leave a backdoor that when activated transfers the execution control to a malicious behavior. Such a malicious modification would hardly be noticed by functional tests. However, it would not be possible for the manufacturer to include a backdoor without modifying the call graph and the control flow graph of the binary code. This makes the malicious modifications exactly the ones amenable to our approach based on call graph and control flow graph parameters.

An open question in the proposed approach, and the subject of ongoing research, concerns the necessity of also considering obfuscated codes during the training phase, to better understand its implications. For example, control-flow obfuscation alters the flow of control of the application by reordering statements, procedures, loops, obscuring flow of control using opaque predicates and replac-

ing transfer flow instructions. Using such obfuscation some properties used in our approach may change, so violating our results. Other possible avenues of research involve merging our current technique with data-flow strategies to circumvent attacks such as modification of variable contents. Finally, in future works we plan to investigate which other graph invariants — crossing number, cycle covering, chromatic number etc. — are meaningful to improve the correspondence of codes written in different languages.

References

1. Thompson, K.: Reflections on trusting trust. Commun. ACM **27**(8) (1984) 761–763
2. McDonald, J.: Delphi falls prey. http://www.symantec.com/connect/blogs/delphi-falls-prey (Last accessed October 2009)
3. Quinlan, D., Panas, T.: Source code and binary analysis of software defects. In: CSIIRW '09: Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research, New York, NY, USA, ACM (2009) 1–4
4. Hassan, A.E., Jiang, Z.M., Holt, R.C.: Source versus object code extraction for recovering software architecture. In: WCRE '05: Proceedings of the 12th Working Conference on Reverse Engineering, Washington, DC, USA, IEEE Computer Society (1995) 67–76
5. Hatton, L.: Estimating source lines of code from object code. In: Windows and Embedded Control Systems, 2005, at: www.leshatton.org/Documents/LOC2005.pdf. (2005)
6. Buttle, D.L.: Verification of Compiled Code. PhD thesis, University of York, UK (2001)
7. Wang, Z., Pierce, K., McFarling, S.: Bmat - a binary matching tool for stale profile propagation. In: The Journal of Instruction-Level Parallelism. (2002)
8. Flake, H.: Structural comparison of executable objects. In: Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), IEEE Computer Society (2004)
9. Oh, J.: Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In: Blackhat technical Security Conference. (2009)
10. Zhenga, J.: A digital image encryption algorithm based on hyper-chaotic cellular neural network. Journal Fundamenta Informaticae (2009)
11. Zeng, H., Rine, D.: A neural network approach for software defects fix effort estimation. In: IASTED Conf. on Software Engineering and Applications. (2004) 513–517
12. Zhenga, J.: Predicting software reliability with neural network ensembles. Expert Systems with Applications (36) (2007) 2116–2122
13. Reddy, C.S., Raju, K.V.S.V.N., Kumari, V.V., Devi, G.L.: Fault-prone module prediction of a web application using artificial neural networks. In: Proceeding (591) Software Engineering and Applications. (2007)
14. Lenic, M., Povalej, P., Kokol, P., Cardoso, A.I.: Using cellular automata to predict reliability of modules. In: Proceeding (436) Software Engineering and Applications. (2004)
15. Boccardo, D.R., Nascimento, T.M., Machado, R.C., Prado, C.B., Carmo, L.F.R.C.: Traceability of executable codes using neural networks. In: Proceedings of the Information Security Conference. (2010) (to appear)

16. Moretti, E., Chanteperdrix, G., Osorio, A.: New algorithms for control-flow graph structuring. In: CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, Washington, DC, USA, IEEE Computer Society (2001) 184
17. IdaPro: Ida pro - disassembler. http://www.hex-rays.com/idapro/ (Last accessed January 2010)
18. Poznyakoff, S.: Gnu cflow. http://savannah.gnu.org/projects/cflow (Last accessed January 2010)
19. Ciocoiu, I.B.: Hybrid feedforward neural networks for solving classification problems. Neural Processing Letters. **16**(1) (2002) 81–91
20. Asadi, R., Mustapha, N., Sulaiman, N.: New supervisioned multi layer feed forward neural network model to accelerate classification with high accuracy. European Journal of Scientific Research. **33**(1) (2009) 163–178
21. Haykin, s.: Neural Networks: A Comprehensive Foundation. Prentice Hall (1998)
22. Hertz, J.A., Krogh, A.S., Palmer, R.G.: Introduction to the Theory of Neural Computation. Addison-Wesley, Redwood City, CA, USA (1991)
23. Moler, C.B.: MATLAB — an interactive matrix laboratory. Technical Report 369, University of New Mexico. Dept. of Computer Science (1980)
24. Men, H., Wu, Y., Gao, Y., Kou, Z., Xu, Z., Yang, S.: Application of support vector machine to heterotrophic bacteria colony recognition. In: CSSE (1). (2008) 830–833
25. Angulo, C., Ruiz, F., González, L., Ortega, J.A.: Multi-classification by using tri-class svm. Neural Processing Letters **23**(1) (2006) 89–101
26. Burkard, J.: C software. http://people.sc.fsu.edu/ burkardt/ (Last accessed January 2010)
27. Oliveira Cruz, A.J.: C software. http://equipe.nce.ufrj.br/adriano/c/exemplos.htm (Last accessed January 2010)

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

CCMN - Bloco C - Cidade Universitária - Ilha do Fundão
Rio de Janeiro - RJ CEP: 21941-916
www.ppgi.ufrj.br