FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

DENIS RENATO DE MORAES PIAZENTIN

TROCA DE CHAVES CRIPTOGRÁFICAS UTILIZANDO CRIPTOGRAFIA NEURAL

MARÍLIA
2011

DENIS RENATO DE MORAES PIAZENTIN

TROCA DE CHAVES CRIPTOGRÁFICAS UTILIZANDO CRIPTOGRAFIA NEURAL

Trabalho de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador:
Prof. Me. MAURÍCIO DUARTE

MARÍLIA
2011

Piazentin, Denis Renato de Moraes
Troca de Chaves Criptográficas utilizando Criptografia Neural / Denis Renato de Moraes Piazentin; orientador: Maurício Duarte. Marília, SP:[s.n], 2011 66 f.

Trabalho de Curso (Graduação em Ciência da Computação) – Curso de Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, Marília, 2011.

1. Redes Neurais Artificiais 2. Troca de Chaves Criptográficas 3. Criptografia Neural

CDD:005.82

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Denis Renato de Moraes Piazentin

ALGORITMO PARA SEGURANÇA DE INFORMAÇÃO UTILIZANDO CRIPTOGRAFIA

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 10,0 (Dez)

Orientador: Mauricio Duarte

1°. Examinador: Fabio Lucio Meira

2°. Examinador: Fábio Dacêncio Pereira

Marília, 29 de novembro de 2011.

Aos meus pais, Vânia Cristina de Moraes e Gabriel Bosquê Neto,
por todo o apoio durante minha vida.

À minha tia, Lúcia Helena Bosquê,
por toda a ajuda e incentivo.

À minha avó, Odila Fabris de Moraes,
pela sua presença e ajuda desde sempre.

Agradecimentos

Ao meu orientador Prof. Ms. Maurício Duarte, pelo auxílio e orientação durante a produção do trabalho.

A todos os professores que lecionaram durante minha graduação em Bacharelado em Ciência da Computação, obrigado por todo o conhecimento compartilhado.

Agradeço aos meus amigos Raphael Negrisoli Batista, Luis Fernando Martins Carlos Junior e Jonathan Schneider, pelo apoio durante todo o curso.

Agradeço à Mayara Monteiro, por todo o apoio e ajuda.

Por último, obrigado a todos os grandes cientistas e pessoas que possibilitaram esse trabalho.

“Com grandes poderes vêm grandes responsabilidades”
Tio Ben

“DON’T PANIC!”
O Guia do Mochileiro das Galáxias

“Sobre os ombros de gigantes”
Isaac Newton

PIAZENTIN, Denis Renato de Moraes. **Troca de Chaves Criptográficas utilizando Criptografia Neural.** 2011 66 f. Trabalho de Curso (Graduação em Ciência da Computação) – Curso de Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, Marília, 2011.

RESUMO

Há uma necessidade constante de proteger informações importantes do acesso indevido e para isso são utilizados algoritmos avançados de criptografia baseados em modelos matemáticos que embaralham e tornam ilegível a informação. Esses complexos algoritmos trabalham com o conceito de chaves criptográficas, um dado usado pelo algoritmo para o embaralhamento dos dados e posterior restauração dos mesmos através da descriptografia. Redes Neurais Artificiais podem se sincronizar por aprendizado mútuo, ajustando seus pesos, e essa sincronização é muito mais rápida que o treinamento das Redes Neurais Artificiais por meio de exemplos. Dada esta propriedade, pode-se usar dessa sincronização e dos pesos ajustados como protocolo de troca de chaves criptográficas. Este trabalho é o estudo e o desenvolvimento de um algoritmo que utiliza esta técnica, conhecida como Criptografia Neural.

Palavras-chave: Redes Neurais Artificiais, Troca de Chaves Criptográficas, Criptografia Neural

PIAZENTIN, Denis Renato de Moraes. **Troca de Chaves Criptográficas utilizando Criptografia Neural.** 2011 66 f. Trabalho de Curso (Graduação em Ciência da Computação) – Curso de Ciência da Computação, Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, Marília, 2011.

ABSTRACT

There is a continuing need to protect sensitive information from unauthorized access and advanced encryption algorithms based on mathematical models to scramble and make the information unreadable are used for that. These complex algorithms work with the concept of cryptographic keys, a data used by a given algorithm for shuffling data and subsequent restoration of them through decryption. Artificial Neural Networks can synchronize by mutual learning, adjusting their weights, and this synchronization is much faster than training the Artificial Neural Networks through examples. Given this property, you may use this synchronization and the adjusted weights as a protocol for exchanging cryptographic keys. This work is the study and the development of an algorithm that uses this technique, known as Neural Cryptography.

Keywords: Artificial Neural Networks, Key Exchange, Neural Cryptography

Lista de Figuras

Sumário

Introdução

A criptografia usa algoritmos criptográficos para transformar texto plano em texto cifrado e utiliza um dado chamado chave criptográfica para criptografar e descriptografar esses textos. Fazer com que ambas as partes da comunicação possuam essa mesma chave é um problema conhecido em criptografia, que já teve propostas e implementadas soluções como o uso de um terceiro confiável, a troca com antecedência e uso de chaves públicas. A sincronização de redes neurais e o uso de seus pesos como chaves criptográficas é uma alternativa ao problema de troca de chave.

Com a descoberta da sincronização entre redes neurais por um processo conhecido como aprendizagem mútua, onde os pesos são ajustados até que convirjam e com a criação de redes neurais com uma estrutura diferenciada onde há uma sincronização muito mais rápida que o treinamento comum, foi possível propor um protocolo de troca de chaves que utiliza os pesos dessas redes sincronizadas como chaves criptográficas, criando uma alternativa ao problema de troca de chave.

Este trabalho tem como objetivo estudar o uso da sincronização de redes neurais como solução para o problema de troca de chaves criptográficas, realizando uma revisão bibliográfica sobre criptografia e segurança da informação, inteligência artificial, em específico redes neurais artificiais, sua estrutura e funcionamento e sobre a criptografia neural em si e criar e testar uma implementação funcional da técnica.

No Capítulo 1 é descrito os princípios da segurança da informação e em seguida é apresentado a criptografia, descrevendo os aspectos e tipos de criptografia, os algoritmos criptográficos de chave simétrica e o problema e soluções existentes para a troca de chaves criptográfica.

O segundo capítulo se inicia com a definição de inteligência artificial e em seguida são apresentados os papéis da inteligência artificial na segurança de informação. Então é iniciada uma revisão bibliográfica sobre redes neurais artificiais, com um histórico, as relacionando com redes neurais biológicas e apresentando o modelo de neurônio artificial, arquitetura de redes neurais e o treinamento das redes neurais.

No terceiro capítulo é apresentada a criptografia neural, descrevendo-a e em seguida, apresentando a estrutura de rede neural utilizada pela técnica, com as regras de aprendizado utili-

zadas e a descrição do fluxo do algoritmo de sincronização da rede. Também são apresentadas diferentes técnicas de ataque documentadas sobre a criptografia neural, como o ataque simples, geométrico, de maioria e genético e as melhorias que já foram propostas para a segurança do algoritmo.

No último capítulo é descrito a implementação do protocolo de troca de chaves, descrevendo a implementação da rede neural utilizada e do algoritmo que executa o fluxo de sincronização entre as redes neurais. Também é descrito o fluxo do algoritmo adaptado para sincronização entre duas máquinas em rede. Nesse capítulo também foram realizados testes no algoritmo de sincronização, demonstrando o tempo e a variação deste quanto a alteração dos parâmetros da rede.

1 Segurança da Informação e Criptografia

1.1 Princípios da Segurança da Informação

A segurança da informação tem como fim proteger a informação pertencente a um indivíduo ou corporação, objetivando a confidencialidade, integridade e disponibilidade, buscando garantir um equilíbrio entre esses objetivos (PFLEEGER; PFLEEGER, 2006) e segundo Menezes, Vanstone e Oorschot (1996, p. 2, tradução nossa) "todas as partes de uma transação devem ter confiança de que certos objetivos relacionados à segurança da informação devem ter sido cumpridos".

Confidencialidade visa garantir que apenas agentes autorizados possam acessar o conteúdo da informação. A necessidade de se garantir a confidencialidade de uma informação surge do armazenamento de informações sensíveis, como dados estratégicos de novos produtos e informações financeiras de empresas ou informações militares e governamentais referentes à segurança nacional, dados que causariam prejuízo a estes, caso se tornassem acessíveis a terceiros com intenções hostis. A confidencialidade também se aplica a tornar obscura a própria existência da informação e de recursos, já que por vezes o próprio fato de se possuir um recurso tecnológico ou uma informação privilegiada é sensível e deve ser mantido em sigilo. Mecanismos de acesso existem para assegurar a confidencialidade da informação, com a criptografia sendo um desses (BISHOP, 2003).

A integridade trata da confiabilidade dos dados, e usualmente refere-se à garantia e prevenção de que ocorram mudanças impróprias ou não autorizadas na informação. Integridade inclui a integridade dos dados (o conteúdo da informação) e integridade da origem (a origem dos dados, também conhecida como autenticação) (BISHOP, 2003). Os mecanismos para garantia da integridade da informação caem em duas categorias distintas:

- Mecanismos de prevenção que tentam manter a integridade dos dados bloqueando qual-

quer tentativa não autorizada de alteração dos dados e qualquer tentativa de alteração dos dados de forma não autorizada (BISHOP, 2003). A primeira forma de tentativa ocorre quando um usuário tenta alterar um dado que ele não tem permissão para alterar, por exemplo, um usuário não autorizado tentando acessar o sistema de home banking de outro. A segunda forma ocorre quando um usuário com permissões para efetuar certas alterações tenta manipular os dados de uma forma não permitida, como um usuário de home banking tentando transferir mais dinheiro do que possui em conta. Enquanto o primeiro tipo de tentativa de comprometer a integridade do sistema pode ser resolvido através de uma simples autenticação, o segundo exige uma série de controles e validações do sistema (BISHOP, 2003).

- Mecanismos de detecção que não tentam manter a integridade da informação, apenas reportam se ela foi comprometida ou não. Estes mecanismos podem analisar eventos do sistema para detectar problemas ou analisar os dados em si para verificar se determinadas condições e verificações permanecem cumpridas (BISHOP, 2003).

A disponibilidade se refere à possibilidade de usar a informação ou recurso desejado. Para a segurança da informação, a informação ou serviço está disponível se está presente de forma que possa ser utilizada, responde em tempo adequado, tem os recursos alocados de forma justa, sem favorecer indevidamente alguns; possui tolerância a falhas, não havendo perda abrupta de informação em caso de falhas no serviço; pode ser usado da forma pretendida e tem a concorrência em casos de acessos de múltiplos usuários controlada, sem que ocorram deadlocks (PFLEEGER; PFLEEGER, 2006). A disponibilidade pode ser comprometida, por exemplo, com um ataque de negação de serviço (*denial of service*, DoS) (BISHOP, 2003).

Menezes, Vanstone e Oorschot (1996) subdivide os objetivos buscados pela segurança da informação em uma maior quantidade de itens, dentre os quais estão listados e descritos:

- Privacidade ou confidencialidade, visando manter a informação secreta de todos, exceto os autorizados a vê-la.
- Integridade dos dados, que garante que a informação não foi alterada por pessoas não autorizadas ou de forma não prevista.
- Identificação ou autenticação de entidades, garantindo a identidade de uma entidade (por exemplo, uma pessoa, um computador, um cartão de crédito).
- Autenticação de mensagem ou autenticação de origem dos dados, corroborando a origem dos dados.

- Assinatura, um meio de criar uma ligação entre informação a uma entidade.
- Autorização, a garantia, à outra entidade, de uma sanção oficial para fazer ou ser alguma coisa.
- Validação, um meio garantir o uso da autorização para usar ou manipular informações ou recursos.
- Controle de acesso, uma forma de restringir acesso a recursos apenas para entidades privilegiadas.
- Certificação, endossamento da informação por uma autoridade confiável.
- Timestamping, a gravação do tempo de existência ou de criação da informação;
- Witnessing ou testemunho, verificação da criação ou existência por uma entidade outra que o criador;
- Recebimento, a confirmação do recebimento da informação;
- Confirmação, o reconhecimento de que os serviços foram prestados;
- Propriedade, uma forma de garantir o direito legal de usar ou transferir um recurso a outros;
- Anonimato, preservar a identidade de uma entidade envolvida em algum processo;
- Não repudiação, prevenção da negação de autorizações ou ações anteriores;
- Revogação, o cancelamento de uma autorização ou certificação.

A segurança da informação se manifesta de muitas formas, de acordo com a situação e com as necessidades (MENEZES; VANSTONE; OORSCHOT, 1996, p. 2) e comumente os objetivos da segurança da informação não podem ser alcançados através de algoritmos matemáticos e protocolos. Uma assinatura, por exemplo, é uma forma de garantir a integridade da origem, enquanto um envelope selado e as leis que proíbem a abertura de correspondências destinadas a terceiros é uma maneira de garantir a confidencialidade da informação.

Para garantir a segurança da informação no ambiente computacional, incluindo a rede Internet onde se encontram diversos serviços como home banking, comércio eletrônico, serviços de e-mail e mensagens instantâneas, trafegando a todo tempo através da rede, são necessárias diversas capacidades técnicas e legais. As capacidades técnicas para garantir a segurança da informação nesse ambiente são providas através da criptografia (MENEZES; VANSTONE; OORSCHOT, 1996).

1.2 Criptografia

Segundo Menezes, Vanstone e Oorschot (1996, p. 4, tradução nossa) "Criptografia é o estudo das técnicas matemáticas relacionadas a aspectos da segurança da informação como a confidenciabilidade, a integridade dos dados, a autenticação de entidades e a autenticação da origem dos dados". Goldreich (2000, p. 1, tradução nossa) diz no livro Foundations of Cryptography: Basic Tools que "historicamente, o termo 'criptografia' foi associado com o problema de desenhar e analisar métodos de criptografia (métodos que provêem comunicação secreta sobre um meio de comunicação inseguro)" e continua que desde que problemas como a construção de assinaturas digitais e o desenho de protocolos tolerantes a falhas caíram no domínio da criptografia "a criptografia pode ser vista como preocupada com o projeto de qualquer sistema que precisa resistir a tentativas maliciosas de ataque".

A criptografia é composta de várias primitivas, como métodos de encriptação, funções de hash e métodos de assinatura digital. Na figura 1.1 se encontra uma esquematização das primitivas e de seu relacionamento segundo Menezes, Vanstone e Oorschot (1996).

1.3 Criptografia de Chave Simétrica

Quando se converte informações sensíveis em textos ilegíveis, está se criptografando os dados. Quando se converte esses textos ilegíveis de volta, está se descriptografando. Para criptografar e descriptografar esses dados utiliza-se um algoritmo (BURNETT; PAINE, 2001). O algoritmo criptográfico age sobre um texto plano e o converte para o chamado texto cifrado. O texto plano pode ser um arquivo de texto humanamente legível ou um arquivo binário que pode ser lido por um computador através de um programa específico. O conceito está diretamente ligado ao fato de a informação poder ser lida ou não, seja por humanos ou computadores. Em paralelo, o texto cifrado é a informação ilegível para ambos; humanos e computadores.

Para o algoritmo criptografar o texto plano em texto cifrado ele precisa de uma chave. Na criptografia computadorizada a chave é sempre um número ou um conjunto de números (BURNETT; PAINE, 2001).

Terada (2008, p. 20), no livro Segurança de Dados: Criptografia em rede de computador, nos dá o exemplo:

> em que Alice, com chave K, *criptografa* um *texto legível* x obtendo outro *texto ilegível* $f_K(x) = y$. O texto y é transmitido para o computador destino do Beto onde y é *descriptografado* pelo algoritmo inverso $f_K^{-1}(y)$ obtendo-se x se e

Fonte: Menezes, Vanstone e Oorschot (1996)

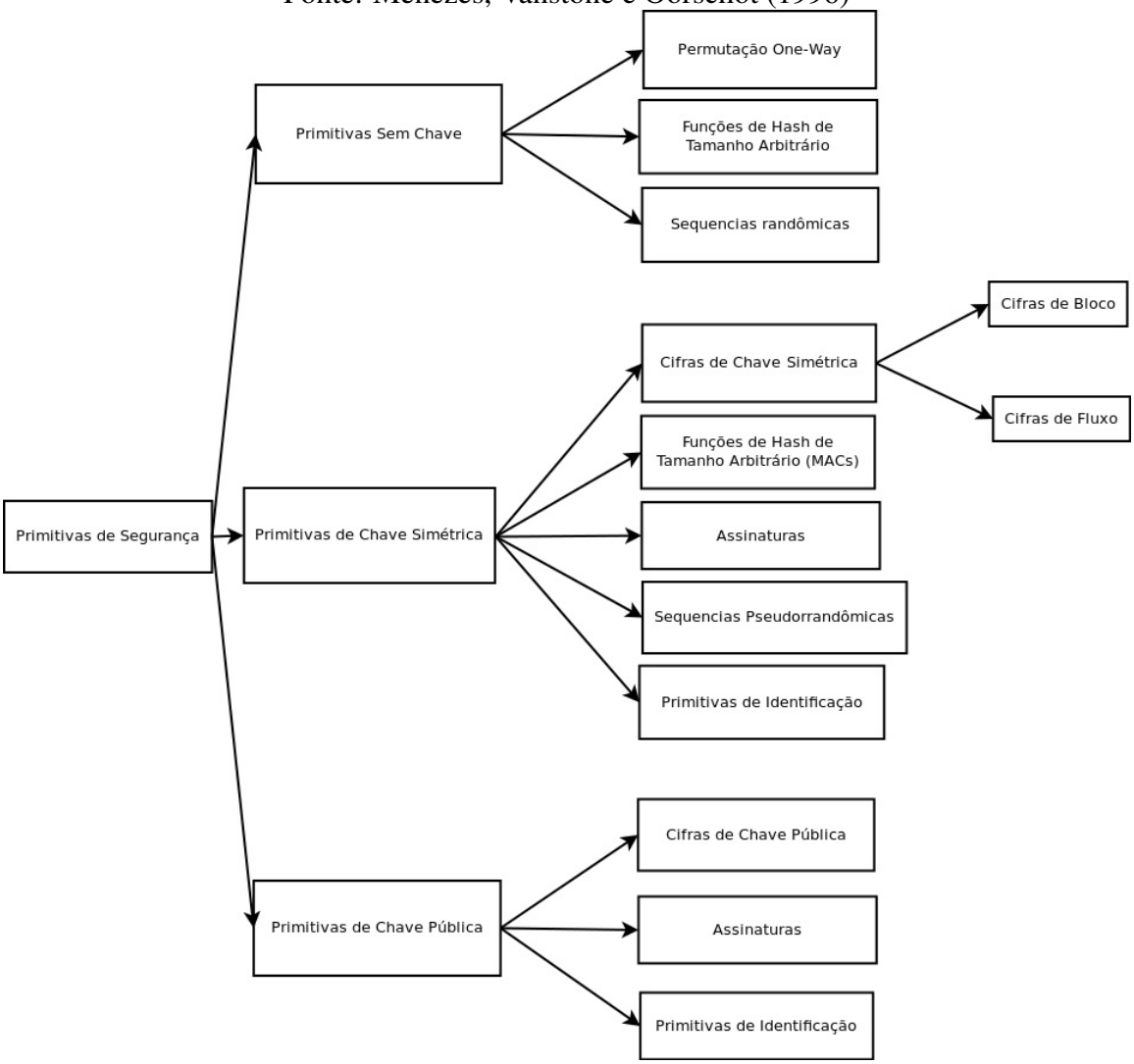

Figura 1.1: Primitivas criptográficas

> somente se o destinatário Beto conhece a chave K. Para quem desconhece a chave K é computacionalmente difícil obter-se y a partir do conhecimento de x, se o algoritmo for bem projetado, isto é, se for *seguro*.

Estes algoritmos são chamados de algoritmos de chave secreta ou de algoritmos simétricos porque a chave K é a mesma para Alice e Beto (TERADA, 2008, p. 43), isto é, a chave usada para criptografar e descriptografar a informação é a mesma.

Essa chave, porém, não é intercambiável entre diferentes algoritmos. Por exemplo, Alice use uma chave K para encriptar um texto plano x utilizando o algoritmo DES, obtendo como resultado o texto cifrado y. O texto y é enviado para Beto, e mesmo conhecendo a chave K ele não conseguirá descriptografar y se utilizar o algoritmo RC6, pois os algoritmos processam o texto e a chave de formas diferentes (TERADA, 2008).

Os algoritmos de chave simétrica são ainda classificados em duas categorias: os de cifra de

bloco e de cifra de fluxo.

Uma cifra de bloco trabalha com um bloco de dados. Quando se fornece dados para criptografar ou descriptografar, o algoritmo quebra os dados em blocos e processa cada um independentemente. As cifras de fluxo processam cada caractere ou byte de informação individualmente, criptografando ou descriptografando e normalmente utilizam a chave apenas uma vez (BURNETT; PAINE, 2001). Cifras de fluxo geralmente são mais apropriadas, e às vezes necessárias, quando não é possível reter a informação, quando os caracteres precisam ser processados individualmente conforme são recebidos ou quando a chance de ocorrerem erros de transmissão é alta, já que a propagação de erros das cifras de fluxo é nula. Infelizmente há poucas especificações completas de cifras de fluxo abertas, em comparação com as cifras de bloco (TERADA, 2008).

Cifras de fluxo são mais rápidas e normalmente usam menos código. O algoritmo mais conhecido de cifras de fluxo, o RC4, pode ser escrito em até 30 linhas de código e é pelo menos duas vezes mais rápido que o mais rápido algoritmo de cifra de bloco. As cifras de bloco tem como vantagens o fato de serem padronizadas e de suas chaves serem reusáveis. É tido como quase garantido que a entidade com que está se comunicando possua os algoritmos DES e AES, mas não é possível afirmar o mesmo sobre a existência do RC4 na outra parte (BURNETT; PAINE, 2001).

1.4 Algoritmos de Criptografia de Chave Simétrica

Alguns dos algoritmos criptográficos que utilizam o esquema de chave simétrica são apresentados na sequência, com uma breve descrição:

1.4.1 Data Encryption Standart – DES e Triple DES

É o algoritmo mais amplamente usado internacionalmente. Foi um avanço científico significativo no sentido de ser o primeiro algoritmo de criptografia cujo conhecimento se tornou público. A segurança do algoritmo DES se baseia exclusivamente no conhecimento da chave secreta. O DES foi projetado pela IBM e publicado pelo National Bureau of Standarts – NBS – em 1977 e foi adotado como padrão para troca de informações comerciais (TERADA, 2008).

Utiliza blocos de entrada de 64 bits, chave de 56 bits de comprimento e saída de 64 bits, com a mesma chave para descriptografar. Porém, O DES é amplamente criticado devido ao pequeno comprimento da chave e a algumas vulnerabilidades em potencial descobertas no algoritmo

(BURNETT; PAINE, 2001) (TERADA, 2008).

Em 1999 na RSA Conference, a Eletronic Frontier Foundation quebrou uma chave DES em menos de 24 horas (BURNETT; PAINE, 2001).

O Triple DES é amplamente utilizado como substituição ao DES. Como o nome sugere, o Triple DES usa o algoritmo DES três vezes, com três chaves distintas. Isso torna o algoritmo muito mais seguro já que, apesar de uma chave DES poder ser quebrada em menos de 24 horas, é impossível para um atacante descobrir que quebrou a primeira chave porque os dados ainda estarão ilegíveis após a primeira descriptografia, e como não há texto legível não é possível inferir que a chave utilizada é a correta. Dessa forma, só é possível saber que a primeira chave é correta após combiná-la com a segunda e terceira chaves também corretas (BURNETT; PAINE, 2001).

Triple DES utiliza três chaves de 56 bits, que em teoria cria uma chave de 168 bits, sendo necessário um ataque de força bruta – uma forma de ataque que tenta todas as combinações possíveis de chaves – para todas as combinações possíveis de 168 bits de dados. Porém, pesquisadores descobriram que é possível reduzir o número de combinações de um ataque de força bruta para o equivalente a uma chave de 108 bits, um comprimento de chave ainda segura, mas a descoberta de tal vulnerabilidade causa dúvidas sobre a possível existência de maiores problemas com a segurança do algoritmo (BURNETT; PAINE, 2001). O segundo problema existente é a lentidão. O algoritmo DES gasta um tempo longo para criptografar e descriptografar dados. O Triple DES gasta o triplo desse tempo (BURNETT; PAINE, 2001).

1.4.2 RC4

É um algoritmo de cifra de fluxo criado em 1987 pela RSA Data Security e nunca teve sua implementação publicada por razões financeiras – a companhia queria ser a única a vender o algoritmo – e apesar disso é provavelmente a cifra simétrica mais utilizada, até mais que o DES. RC4 é também usada como parte do Secure Socket Layer (SSL), o protocolo de comunicação segura da World Wide Web (BURNETT; PAINE, 2001).

1.4.3 Advanced Encryption Standart – AES

Em 2 de Janeiro de 1997 o NIST – National Institute of Standarts and Technology – dos Estados Unidos anunciou uma competição convidando qualquer um a submeter um novo algoritmo como uma padrão para a industria, para ser conhecido como AES, com a condição de desistir dos direitos de toda a propriedade intelectual do algoritmo escolhido (BURNETT;

PAINE, 2001). O objetivo era escolher um substituto para o DES (TERADA, 2008).

Muitos algoritmos foram submetidos para a competição e em agosto de 1998 o NIST nomeou 15 candidatos. Esses candidatos então foram analisados quanto à segurança (sem fraquezas no algoritmo), performance (teria de ser rápido em muitas plataformas) e tamanho (não poderia ocupar muito espaço ou memória). Em 1999 foram anunciados os cinco finalistas e no ano subsequente os algoritmos foram testados por pesquisadores, criptanalistas, fabricantes de hardware e software. Muitos artigos foram publicados e muitas estatísticas foram geradas comparando os finalistas (BURNETT; PAINE, 2001).

Em outubro de 2000 o algoritmo Rijndael foi selecionado e adotado com pequenas alterações no comprimento permitido das chaves como AES (TERADA, 2008).

O AES utiliza blocos de 128 bits de entrada e saída, chaves de 128, 192 ou 256 bits e foi escolhido por ser eficientemente implementável em software, hardware e smart-cards (TERADA, 2008).

1.5 Troca de chaves criptográficas

Criptografia de chave simétrica pode manter a informação secreta a salvo, mas se é necessário trocar informações secretas com outras pessoas, também é necessário trocar as chaves. Se um atacante pode interceptar a mensagem, também pode interceptar a chave que a descriptografa, surge então o problema de distribuição de chaves criptográficas (BURNETT; PAINE, 2001).

Foram criados vários protocolos para solucionar esse problema, e alguns dos subconjuntos desses protocolos são apresentados aqui. Para fins de definição, tem-se que um protocolo é, segundo Menezes, Vanstone e Oorschot (1996, tradução nossa), “um algoritmo de múltiplas partes, definido por uma seqüência de passos especificando precisamente as ações requeridas de duas ou mais partes para alcançar um objetivo especifico”.

1.5.1 Compartilhamento de chaves com antecedência

Neste protocolo as duas partes que desejam se comunicar devem trocar as chaves com antecedência. Por exemplo, Alice deseja transferir alguma informação criptografada para Beto. Alice gera uma chave K que usará para criptografar as informações que vai transferir. Alice então se encontra com Beto e transfere para ele a chave gerada. É importante que Alice e Beto se encontrem para que a chave não seja transferida por nenhum meio de comunicação não

confiável. Após a troca de chaves, Alice pode utilizar a chave K para criptografar um texto x e enviá-lo para Beto, onde Beto poderá utilizar a chave K previamente compartilhada para descriptografá-lo (BURNETT; PAINE, 2001).

Este é um esquema funcional, um atacante que interceptar o texto criptografado não poderá recuperar a informação por não ter acesso à chave (BURNETT; PAINE, 2001).

O protocolo, porém, tem problemas, considerando que para cada pessoa com que se deseja trocar a chave é necessário um novo encontro. Se as pessoas estiverem em locais geograficamente distantes ou for necessária a troca com muitas pessoas diferentes, muitos encontros serão necessários, tornando o esquema inviável em termos de logística. Foi calculado que para que 1000 pessoas troquem chaves entre si, 499500 encontros são necessários (BURNETT; PAINE, 2001).

1.5.2 Uso de um terceiro confiável

No caso do compartilhamento de chaves com antecedência não ser uma opção, é possível usar uma terceira parte confiável (trusted third party, TTP), que compartilha uma chave com cada uma das outras partes. Nesse esquema, cada parte possui uma chave diferente compartilhada com o TTP, que precisa ser confiável porque tem a chave de todas as outras partes (BURNETT; PAINE, 2001).

Supondo que Alice e Beto tenham criado chaves criptográficas com o TTP e querem se comunicar, Alice manda uma mensagem para o TTP solicitando uma chave de sessão. O TTP gera uma chave de sessão para ser usada e a criptografa usando a chave compartilhada com Alice e a envia, dessa forma qualquer terceiro que intercepte a comunicação não poderá identificar a chave recém gerada. Alice então descriptografa a chave de sessão usando sua chave compartilha com o TTP, que também criptografa a chave de sessão com a outra compartilhada com Beto e a envia para ele. Agora Alice e Beto compartilham da mesma chave de sessão e podem criptografar a informação e trocá-la utilizando essa chave (BURNETT; PAINE, 2001).

O primeiro problema relacionado a esta técnica é que o TTP pode ler todas as mensagens trocadas, o que torna a sua confiabilidade extremamente crítica. Outro problema é que caso o TTP seja perdido ou de alguma forma comprometido, é necessário obter outro TTP e reiniciar todo o processo de geração de chaves (BURNETT; PAINE, 2001).

1.5.3 Criptografia de Chave Pública

Em 1976 Diffie e Hellman publicaram um artigo onde foi proposto um modelo de sistema criptográfico em que cada usuário possui um par de chaves, sendo uma particular e outra pública (TERADA, 2008). Estas duas chaves são diferentes e estão ligadas por um relacionamento matemático. Uma delas, a chave pública, é usada para criptografar os dados e somente a segunda, a chave privada, é capaz de descriptografar os dados criptografados pela primeira. A chave pública é conhecida e não é mantida em segredo e sua contraparte, a chave privada, é conhecida apenas pelo dono do par de chaves. Por esse motivo a criptografia de chave assimétrica é também chamada de criptografia de chave pública (BURNETT; PAINE, 2001).

O protocolo Diffie-Hellman, como ficou conhecido, usava uma troca de mensagens sobre um canal público para gerar uma chave secreta (BURNETT; PAINE, 2001). As duas partes que queriam se comunicar conhecem publicamente dois inteiros e tem dois números aleatórios privados. Uma chave secreta igual para ambas as partes é gerada envolvendo cálculos matemáticos baseados nesses números. A versão básica desse protocolo era segura contra adversários passivos (observadores), mas era sucessivo à atacantes ativos, do tipo man- in-the-middle, onde o atacante não apenas observa como também insere informações da linha de comunicação (MENEZES; VANSTONE; OORSCHOT, 1996) (TERADA, 2008). Uma versão modificada do protocolo evita esse tipo de ataque (TERADA, 2008). O protocolo Diffie-Hellman resolve o problema de troca de chave criptográfica, mas não é um algoritmo de criptografia (BURNETT; PAINE, 2001).

Em 1978 é publicado o algoritmo RSA por Ron Rivest, Adi Shamir e Len Adleman em que o esquema de cada usuário possuir sua chave pública e privada é efetivamente usado (TERADA, 2008). Por exemplo, Alice deseja se comunicar com Beto. Alice gera uma chave de sessão e a criptografa utilizando a chave pública disponibilizada por Beto e um algoritmo de criptografia assimétrica. Alice usa a chave de sessão gerada para criptografar o texto que deseja enviar utilizando um algoritmo de criptografia simétrica e então envia a chave de sessão criptografada com a chave pública e o texto criptografado com a chave de sessão para Beto. Como a chave usada para criptografar o texto está criptografada com a chave de Beto, e apenas Beto pode descriptografá-la com sua chave privada, um possível atacante que intercepte os dados não poderá ler o conteúdo da mensagem. Quando Beto recebe a mensagem, Beto utiliza sua chave privada para descriptografar a chave de sessão e então utiliza a chave de sessão para descriptografar o texto recebido, obtendo o texto legível original (BURNETT; PAINE, 2001).

O motivo pelo qual todo o conteúdo da mensagem, ao invés de apenas a chave de sessão, não é criptografado utilizando algoritmo de chave assimétrica e a chave pública é porque algoritmos

de chave pública em geral são muito mais lentos que os de chave simétrica. Tomando como exemplo o algoritmo de chave assimétrica RSA, o algoritmo de chave simétrica de bloco RC5 é 500 vezes mais rápido e o algoritmo de chave simétrica de fluxo RC4 é 700 vezes mais rápido (BURNETT; PAINE, 2001). Dessa forma, é mais eficiente utilizar o algoritmo de chave assimétrica apenas para criptografar a chave de sessão usada no texto.

1.5.4 Criptografia Neural

Em 2002 os físicos Kanter, Kinzel e Kanter, no artigo Secure exchange of information by synchronization of neural networks, propuseram um novo protocolo de troca de chaves criptográficas entre duas partes, baseando-se na capacidade de sincronização de redes neurais artificiais. Esse protocolo de troca de chaves criptográficas foi nomeado Criptografia Neural.

As redes neurais artificiais são organizadas na forma de Tree Parity Machines e trocam mensagens contendo vetores de entradas e saídas, usados no treinamento das redes neurais, através de um meio não seguro (KANTER; KINZEL; KANTER, 2002).

Utilizando o exemplo em que Alice e Beto queiram se comunicar, ambos iniciam o processo conhecido como sincronização de redes neurais. As redes neurais de ambos recebem vetores de entrada idênticos, geram uma saída e a partir daí ocorre o treino mútuo utilizando os resultados da rede parceira. Os pesos sinápticos das redes neurais artificiais convergem para um vetor de pesos idênticos (KINZEL; KANTER, 2002). Esses vetores de pesos idênticos podem ser usados como chaves criptográficas por Alice e Beto e utilizados pelos algoritmos de criptografia simétrica que desejarem.

Foi observado que redes neurais artificiais sincronizam-se muito mais rápido, utilizando o treinamento mútuo, do que um possível atacante que utilize uma rede neural e a treine com as mensagens trocadas, o que torna a técnica segura contra observadores externos (RUTTOR, 2007).

Criptografia neural é uma área nova nos campos da criptografia e da inteligência artificial e se mostrou uma alternativa muito mais veloz do que a criptografia de chave pública (RUTTOR, 2007). Ademais, desde 2002 tem havido várias publicações tratando desse tema, analisando a técnica, estudando formas de ataque e vulnerabilidades da criptografia neural (SHACHAM et al., 2003) (RUTTOR et al., 2006) e propondo melhorias na segurança do protocolo (RUTTOR; KINZEL; KANTER, 2004) (ALLAM; ABBAS, 2010).

2 Inteligência Artificial e Redes Neurais Artificiais

A criptografia neural utiliza Redes Neurais Artificiais, um dos subcampos de inteligência artificial, para gerar as chaves criptográficas entre as partes de uma comunicação. Neste capítulo são apresentados os conceitos de inteligência artificial e sua participação atual na segurança da informação, as redes neurais biológicas e artificiais, o modelo de neurônio artificial, a arquitetura e os modelos de treinamento das redes.

2.1 Inteligência Artificial

A inteligência artificial (IA) é um dos mais novos campos na ciência e engenharia. Os trabalhos na área se iniciaram logo após a segunda guerra mundial e o nome do campo em si foi criado em 1956 (RUSSELL; NORVIG, 2010).

As definições de IA, segundo Russell e Norvig (2010), são divididas em quatro diferentes abordagens:

- agir humanamente, definida por Kurzweil (apud RUSSELL; NORVIG, 2010, pag. 2) como "a arte de criar máquinas que executem funções que necessitam de inteligência quando executadas por pessoas";
- pensar humanamente, em que IA consiste da automação das "atividades que nós associamos com o pensamento humano, atividades como tomadas de decisão, resolução de problemas, aprender..." (BELLMAN, 1978 apud RUSSELL; NORVIG, 2010, pag. 2);
- pensar racionalmente, definida por Winston (apud RUSSELL; NORVIG, 2010, pag. 2) como "o estudo da computação que torna possível perceber, raciocinar e agir";
- e agir racionalmente, definido por Nilsson (apud RUSSELL; NORVIG, 2010, pag. 2) como tendo que a inteligência artificial "é preocupada com comportamentos inteligentes em artefatos", e por Poole, Mackworth e Goebel (apud RUSSELL; NORVIG, 2010,

pag. 2) que nos diz que “Inteligência Computacional é o estudo do desenho de agentes inteligentes”.

Pesquisas em IA são feitas em vários subcampos, como em computação evolutiva, campo baseado no conceito de seleção natural, reprodução e mutação de organismos, inteligência coletiva, originada do estudo de organismos sociais como formigas e abelhas, sistemas fuzzy que lidam com incertezas e razão aproximada, e redes neurais artificiais, campo surgido a partir do modelo de neurônio humano (ENGELBRECHT, 2002), assim como é aplicada em diversas áreas, como no mercado financeiro, comércio eletrônico, medicina, transportes e na segurança da informação.

2.2 Inteligência Artificial na Segurança da Informação

Técnicas envolvendo inteligência artificial são utilizadas no campo de segurança de informação, como detecção de intrusão em sistemas, biometria para autenticar usuários, quebra de CAPTCHA e na criptografia neural.

Na detecção de intrusão em sistemas usa-se RNAs para filtrar e agrupar o grande volume de dados de uso do sistema, já que esses dados tornariam uma análise integral muito custosa. RNAs também são usadas para detectar anomalias nesses dados, causadas por corportamentos inesperados do usuário. O uso de IA para filtrar e identificar anomalias nessa informação é especialmente importante quando se deseja detectar intrusões em tempo real (FRANK; MDAC, 1994).

A IA também é usada na biometria para prover métodos de autenticação de usuário, por exemplo, para detectar e reconhecer rostos (ROWLEY, 1999).

Algoritmos de inteligência artificial para reconhecimento de caracteres são utilizados por atacantes e aprimorados constantemente para quebrar CAPTCHAS (AHN et al., 2003). CAPTCHA é um programa que gera testes fáceis de serem resolvidos por humanos mas difíceis de serem resolvidos computacionalmente, e consistem comumente de imagens com letras distorcidas, como visto na Figura 2.1 ou aúdios com ruídos. Os CAPTCHA visam assegurar que o usuário do sistema é humano, e assim criar uma barreira contra programas enviadores de spam, contra ataques de dicionário em formulários de autenticação, ou ainda criar uma barreira contra ataques de negação de serviço (AHN et al., 2003).

Fonte: próprio autor

Figura 2.1: CAPTCHA usado para assegurar que o usuário não é uma máquina.

2.3 Redes Neurais Artificiais

Redes Neurais Artificiais (RNAs) são sistemas paralelos distribuídos por unidades de processamento simples (neurônios artificiais) que calculam determinadas funções matemáticas (BRAGA; CARVALHO; LUDERMIR, 2007). Em RNAs, o procedimento normal para a resolução de problemas consistem em primeiro treinar a rede, extraindo as características necessárias para representar a informação fornecida e em seguida usar essas características para gerar respostas para o problema (BRAGA; CARVALHO; LUDERMIR, 2007).

O primeiro trabalho envolvendo neurônios artificiais foi apresentado em 1943 por McCulloch e Pitts. Em 1949, Donald Hebb publica um trabalho que traz consideráveis avanços à área, demonstrando uma técnica de aprendizado baseada na variação dos pesos nas entradas dos neurônios. Em 1958 novos avanços são conseguidos graças ao modelo de perceptron introduzido por Rosenblatt, tornando possível classificar certos padrões através de sinapses ajustáveis (HAYKIN, 1999) (BRAGA; CARVALHO; LUDERMIR, 2007).

Em 1969, Minsky e Papert expuseram deficiências no perceptron e apontaram um problema de crescimento explosivo nas RNAs, que causou, junto com a limitação do poder computacional na época, a estagnação dos trabalhos na área, até serem retomados em 1982 após a publicação de um trabalho de Hoppfield sobre as propriedades associativas das RNAs e posteriormente em 1986, com o desenvolvimento do algoritmo de back-propagation por Rumelhart, Hinton e Willians (HAYKIN, 1999) (BRAGA; CARVALHO; LUDERMIR, 2007).

As RNAs são aplicadas basicamente em problemas em que existem dados experimentais ou gerados por modelos, pelos quais a rede poderá se adaptar para executar determinada tarefa (BRAGA; CARVALHO; LUDERMIR, 2007).

As principais tarefas nas quais RNAs se aplicam são:

- classificação, que envolve atribuir um padrão desconhecido entre várias classes conhecidas;

- categorização, que envolve a descoberta de categorias e classes definidas nos dados de entrada;
- aproximação, em que os pesos da rede são adaptados para mapear as relações entre entradas e saídas, para mapear funções contínuas das variáveis de entrada;
- previsão, onde a rede é treinada com estados passados, atuais e futuros para aprender a estimar situações futuras;
- e otimização, onde se utilizam RNAs para minimizar ou maximizar uma função de custo (BRAGA; CARVALHO; LUDERMIR, 2007).

2.4 Redes Neurais Biológicas

O cérebro humano contém em torno de 10^{11} neurônios e cada um deles se comunica com de 10 a 100,000 de outros continuamente e em paralelo (RUSSELL; NORVIG, 2010) (BRAGA; CARVALHO; LUDERMIR, 2007). E é na estrutura, topologia de conexões e comportamento conjuntos desses neurônios que se forma a base para o estudo das RNAs (BRAGA; CARVALHO; LUDERMIR, 2007).

As RNAs buscam implementar o comportamento funcional e a dinâmica das redes biológicas (BRAGA; CARVALHO; LUDERMIR, 2007).

Um neurônio biológico é dividido, tipicamente, em três partes (BRAGA; CARVALHO; LUDERMIR, 2007). O corpo da célula, ou soma, e do corpo da célula saem várias fibras, chamadas dendritos e uma única e longa fibra chamada axônio (RUSSELL; NORVIG, 2010). O corpo da célula mede apenas alguns milésimos de milimetros, e os dendritos alguns poucos milimetros e o axonio medindo de 1 cm a até 1 metro (BRAGA; CARVALHO; LUDERMIR, 2007) (RUSSELL; NORVIG, 2010). A figura 2.2 mostra a estrutura de um neurônio biologico.

A função dos dendritos é receber as informações, os impulsos nervosos, de outros neurônios e conduzi-las até o corpo celular, onde a informação é processada e novos impulsos são gerados. Esses impulsos são transmitidos do axônio da célula até os dendritos dos neurônios seguintes. O ponto de contato entre o axonio da célula e um dentrito do neurônio seguinte é chamado de sinapse, e é por elas que os neurônios se unem formando as redes neurais biológicas (BRAGA; CARVALHO; LUDERMIR, 2007) (HAYKIN, 1999).

As sinapses funcionam como válvulas. Os sinais recebidos pelos dendritos são enviados para o corpo do neurônio e combinados com os outros sinais recebidos. Se a excitação do

Fonte: Braga, Carvalho e Ludermir (2007)

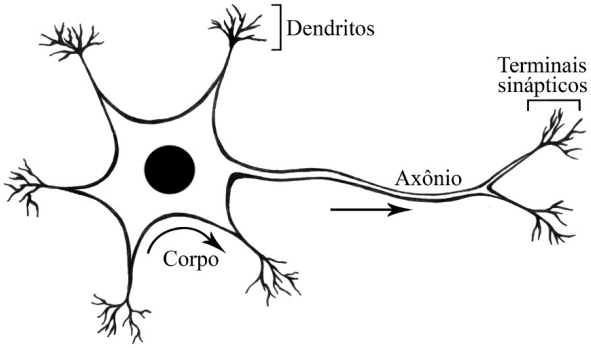

Figura 2.2: Estrutura simplificada de um neurônio biológico.

neurônio for suficientemente alta, a célula dispara um impulso para as células seguintes através do axônio. É esse sistema simples que é responsavel pela maioria das funções cerebrais (BRAGA; CARVALHO; LUDERMIR, 2007).

Combinar sinais recebidos pelos dendritos é uma das funções básicas da célula. Caso essa combinação esteja acima do limiar de excitação no neurônio, o impulso elétrico é produzido e propagado através do axônio para os neurônios seguintes. São concentrações de íons de sódio e íons de potássio que dão o potencial elétrico ao neurônio, e para que a célula dispare é necessário que os impulsos recebidos por ele reduzam esse nível de potencial até um valor de limiar, onde ocorre uma inversão de polaridade que faz com que o impulso se propague pelo axônio. No terminal de um axônio os canais controlados por tensão se abrem e são liberadas na sinapse moléculas neurotransmissoras, cujo tipo determinará a polarização ou despolarização do corpo seguinte, agindo como excitatória ou inibidora. A combinação de todos os impulsos recebidos dessa forma irá determinar a geração ou não de um impulso nervoso nesse outro neurônio (BRAGA; CARVALHO; LUDERMIR, 2007).

2.5 Modelo de Neurônios Artificiais

O neurônio artificial proposto por McCulloch e Pitts em 1943 é uma simplificação do conhecimento na época sobre o neurônio biologico. A descrição matemática do neurônio resultou num modelo com n terminais de entrada (dendritos) que recebem os valores $x_1, x_2, \ldots, x_n$ representando as ativações dos neurônios anteriores e um terminal de saída y que representa o axônio (BRAGA; CARVALHO; LUDERMIR, 2007) (HAYKIN, 1999). Representando as sinapses, os terminais de entrada do neurônio possuem pesos acoplados $w_1, w_2, \ldots, w_n$ com valores que podem ser positivos ou negativos, dependendo das sinapses serem excitatórias ou inibitórias. O efeito de uma sinapse i então é dado por $x_i w_i$. O neurônio então dispara quando a soma dos

impulsos ultrapassa seu limiar de excitação (*threshold*), dado pela somatória dos valores $x_i w_i$ e por uma função de ativação $f(u)$ e o disparo é dado pela saída nos valores 1 ou 0 (BRAGA; CARVALHO; LUDERMIR, 2007) (RUSSELL; NORVIG, 2010). A representação do neurônio proposto pode ser vista na figura 2.3.

Fonte: Braga, Carvalho e Ludermir (2007)

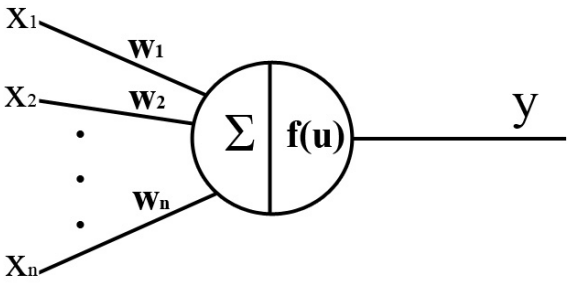

Figura 2.3: Modelo do neuronio MCP, com as entradas $x_1, x_2, \ldots, x_n$, pesos $w_1, w_2, \ldots, w_n$, saída y, soma ponderada Σ e função de ativação $f(u)$.

A função de ativação gera a saída y do neurônio a partir do vetor de pesos e entradas $x_i w_i$. Neurônios MCP usam uma função de ativação degrau deslocada do limiar de ativação Θ em relação à origem, conforme apresentado na equação 2.1.

$$f(u) = \begin{cases} 0, & \sum_{i=1}^{n} x_i w_i \geq \Theta \\ 1, & \sum_{i=1}^{n} x_i w_i < \Theta \end{cases} \tag{2.1}$$

Também podem ser usadas funções degrau com aproximação continua, como na 2.2

$$f(u) = \frac{1}{1 + e^{-\beta u}} \tag{2.2}$$

onde β é a inclinação da função, e ainda funções de ativação lineares como apresentado na equação 2.3. (BRAGA; CARVALHO; LUDERMIR, 2007).

$$f(u) = u \tag{2.3}$$

2.6 Arquiteturas de Redes Neurais Artificiais

Neurônios individuais possuem capacidade computacional limitada, porém, um conjunto de neurônios conectados em forma de uma rede é capaz de resolver problemas de complexidade elevada (BRAGA; CARVALHO; LUDERMIR, 2007).

Existem basicamente dois modelos de arquitetura de redes neurais artificiais, o modelo de rede com camadas alimentadas para frente (*feedforward*) e um modelo com conexões recorrentes, em que há pelo menos um retorno na alimentação dos neurônios. As arquiteturas de RNAs também diferem no número de camadas de neurônios usadas (RUSSELL; NORVIG, 2010) (HAYKIN, 1999) (BRAGA; CARVALHO; LUDERMIR, 2007).

As redes feedforward possuem conexões somente em uma direção, cada nó (neurônio) recebe entradas de neurônios da camada superior e envia sua saída para nós da camada inferior, sem haver nenhuma recursão (RUSSELL; NORVIG, 2010). A figura 2.4 mostra a estrutura de uma rede feedforward. Por não possuírem recorrência, suas saídas em um determinado instante dependem apenas das entradas atuais, e por tal razão são consideradas estáticas (BRAGA; CARVALHO; LUDERMIR, 2007).

Fonte: Braga, Carvalho e Ludermir (2007)

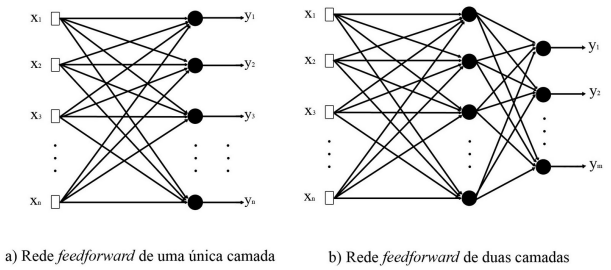

Figura 2.4: Estrutura de RNAs com arquitetura feedforward com uma (a) e duas (b) camadas.

Redes recursivas por sua vez enviam suas saídas também para neurônios de camadas superiores ou da própria camada. A figura 2.5 mostra a estrutura de uma rede com recursão. Essa estrutura de rede é usada na resolução de problemas que envolvem processamento temporal, já que suas saídas dependem também do estado interno dos neurônios da rede (BRAGA; CARVALHO; LUDERMIR, 2007) (RUSSELL; NORVIG, 2010).

Segundo Braga, Carvalho e Ludermir (2007), na escolha da estrutura de uma RNA devem ser analisadas a complexidade do problema, a dimensionalidade do espaço de entrada, as características dinâmicas ou estáticas, o conhecimento prévio sobre o problema e a representatividade dos dados.

2.7 Aprendizado e Treinamento de Redes Neurais

Uma das propriedades mais significantes para as RNAs é a capacidade da rede de aprender e assim melhorar sua performance (HAYKIN, 1999).

Fonte: Braga, Carvalho e Ludermir (2007)

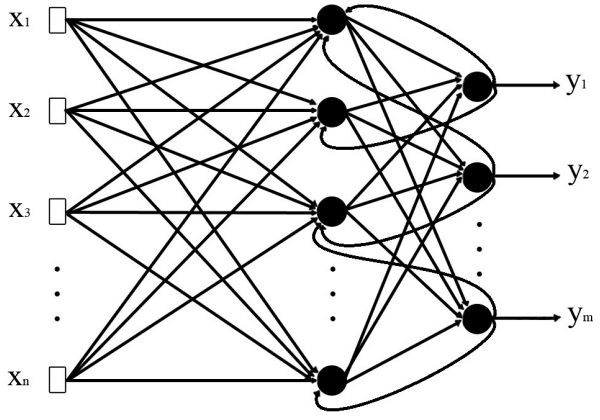

Figura 2.5: Arquitetura de rede com recursão entre a camada de saída e a camada intermediária.

A aprendizagem consiste em um processo iterativo de ajuste dos pesos das conexões que guardam, ao final do processo, o conhecimento adquirido pela RNA (BRAGA; CARVALHO; LUDERMIR, 2007).

Braga, Carvalho e Ludermir (2007) e Haykin (1999) usam a definição de aprendizagem dada por Mendel e McClaren (1970):

> Aprendizado é o processo pelo qual os parâmetros livres de uma rede neural são ajustados por meio de uma forma continuada de estímulo pelo ambiente externo, sendo o tipo específico de aprendizado definido pela maneira particular como ocorrem os parâmetros livres.

O vetor de parâmetros livres (pesos) $w(t+1)$ no instante $t+1$ pode ser escrito conforme a equação

$$w(t+1) = w(t) + \Delta w(t) \tag{2.4}$$

Onde $w(t)$ representa o valor dos pesos no instante t e $w(t+1)$ no instante $t+1$. $\Delta w(t)$ é o ajuste aplicado aos pesos (BRAGA; CARVALHO; LUDERMIR, 2007) (HAYKIN, 1999).

Os algoritmos de aprendizado diferem, basicamente, na forma como $\Delta w(t)$ é calculado (BRAGA; CARVALHO; LUDERMIR, 2007). Há dois principais paradigmas dividindo os algoritmos: aprendizado supervisionado e aprendizado não-supervisionado (BRAGA; CARVALHO; LUDERMIR, 2007).

Aprendizado supervisionado implica na existência de um supervisor que estimula as entradas da rede e compara a saída calculada com a saída desejada, ajustando os pesos para aproximar a saída calculada da saída que se deseja (BRAGA; CARVALHO; LUDERMIR, 2007). O

aprendizado supervisionado ocorre basicamente em duas formas: *off-line*, em que os conjuntos de treinamentos são fixos e, uma vez obtida uma solução para a rede, esta deve permanecer fixa e *on-line*, em que os dados mudam continuamente e a rede deve estar em um processo contínuo de adaptação (BRAGA; CARVALHO; LUDERMIR, 2007).

No aprendizado não-supervisionado não há um supervisor externo que acompanha o processo de treinamento e apenas padrões de entrada estão disponíveis para a rede (BRAGA; CARVALHO; LUDERMIR, 2007). Durante o aprendizado os padrões são apresentados continuamente à rede, e a existência de regularidades nesses dados torna o aprendizado possível. O aprendizado não-supervisionado se aplica a problemas que visam descobrir características estatisticamente relevantes nos dados de entrada, como agrupamentos ou classes (BRAGA; CARVALHO; LUDERMIR, 2007). Segundo Braga, Carvalho e Ludermir (2007) os modelos mais conhecidos de aprendizado não-supervisionado são os mapas auto-organizativos de Kohonen e os modelos ART. Algoritmos baseados na regra de Hebb também são classificados como não-supervisionados, já que apesar de necessitarem de pares de entrada e saída para o treinamento, não há a figura de um supervisor externo (BRAGA; CARVALHO; LUDERMIR, 2007).

A regra de aprendizado de Hebb propõe que o peso de uma conexão sináptica deve ser ajustado caso haja sincronismo entre as atividades de entrada e saída. Caso dois neurônios em lados distintos da sinapse sejam ativados sincronamente, essa sinapse é fortalecida. Caso essa ativação seja assíncrona, porem, a sinapse é enfraquecida ou eliminada (BRAGA; CARVALHO; LUDERMIR, 2007) (HAYKIN, 1999).

Matematicamente, temos que o ajuste dos pesos no tempo t deve ser proporcial ao produto dos valores de entrada e saída da rede, portanto temos que $\Delta w_{ij}(t) \propto y_i(t)x_j(t)$ onde $w_{ij}(t)$ é o valor do peso j do neurônio i, $x_j(t)$ é o valor da entrada j e $y_i(t)$ é a saída do neurônio i. A forma mais simples para ajustes dos pesos pela regra de Hebb é então dada pela equação

$$\Delta w_{ij}(t) = n y_i(t) x_j(t) \tag{2.5}$$

onde n é uma constante positiva que determina a taxa de aprendizado.

3 Criptografia Neural

Foi observado um interessante fenômeno de sincronização de redes neurais através do aprendizado mútuo. Também foi observado que certas redes neurais com uma estrutura especial são capazes de sincronizar muito mais rápido do que é possível para uma rede neural aprender simplesmente usando as mensagem para treinar. Baseado nesse fenômeno de sincronização, foi proposto um protocolo de troca de chaves criptográficas conhecido como criptografia neural.

A sincronização de redes neurais é um caso especial de aprendizado onde duas redes neurais são iniciadas com pesos aleatórios e, a cada passo do processo de sincronização, recebem uma lista de entradas comum e calculam e comunicam suas saídas. Caso o mapeamento entre a entrada atual e a saída de ambas as redes não seja igual, os pesos da rede são atualizados de acordo com uma das regras aplicáveis (RUTTOR, 2007).

No caso de redes neurais perceptrons, não se observa diferença significativa entre o tempo necessário para a rede aprender por exemplos, do tempo para sincronizar. Porém, redes neurais *Tree Parity Machines* (TPM) sincronizam mais rápido do que uma terceira rede pode aprender observando a comunicação, e essa diferença de tempo é usada pelo protocolo para resolver o problema de troca de chaves (RUTTOR, 2007).

3.1 Tree Parity Machine

A arquitetura da Tree Parity Machine foi apresentada no artigo *Secure exchange of information by synchronization of neural networks* (KANTER; KINZEL; KANTER, 2002) e pode ser vista na Figura 3.1.

A TPM é composta por três camadas: a de entrada, a escondida e a de saída, respectivamente. A camada escondida possui K unidades, representados na Figura 3.1 por o_i onde $i = 1, \ldots, K$, com cada unidade possuindo N unidades da camada de entradas x_j com peso associado w_j, onde $j = 1, \ldots, N$. A camada de saída possui apenas uma unidade y.

Fonte: Ruttor (2007)

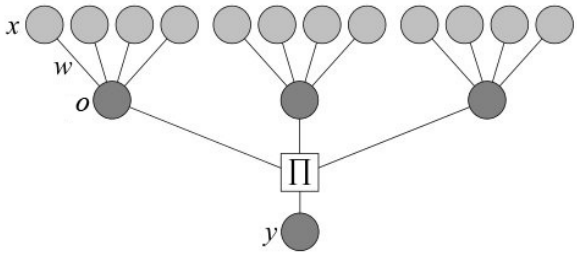

Figura 3.1: Estrutura de uma Tree Parity Machine, com $K = 3$ e $N = 4$.

Tem-se que todos os valores de entradas são binários, tais que

$$x_{i,j} \in \{-1, +1\} \tag{3.1}$$

e os pesos que definem o mapeamento de entradas para saída são números discretos entre $-L$ e $+L$,

$$w_{i,j} \in \{-L, -L+1, \ldots, +L-1, L\} \tag{3.2}$$

como em outras redes neurais, tem-se também que o estado de cada neurônio é dado pelo somatório de $x_j w_j$,

$$h_i = \frac{1}{\sqrt{N}} x_i w_i = \frac{1}{\sqrt{N}} \sum_{j=1}^{N} x_{i,j} w_{i,j} \tag{3.3}$$

com a saída o_i sendo definida pela função sinal de h_i,

$$o_i = sgn(h_i) \tag{3.4}$$

com o caso especial de $h_i = 0$ sendo mapeado para -1 para garantir um valor de saída binário. Tem-se então que a saída total y da TPM é dado pelo produto (paridade) das unidades escondidas o_i,

$$y = \prod_{i=1}^{K} o_i \tag{3.5}$$

De tal forma que a saída y apenas indica se o número de unidades escondidas inativas é par ($y = +1$) ou ímpar($y = -1$) e, consequentemente, há 2^{K-1} diferentes representações internas $(o_1, o_2, \ldots, o_k)$ que resultam no mesmo valor de y (RUTTOR, 2007).

3.2 Sincronização de Redes Neurais

No início do processo de sincronização, os pesos das TPM A e B são inicializados independentemente com valores aleatórios, não relacionados e secretos.

Após isso, para cada passo da sincronização, A e B tem as entradas alimentadas com um vetor aleatório gerado publicamente de tamanho $K \times N$ e as saídas y^A e y^B são calculadas (RUTTOR, 2007).

Caso $y^A \neq y^B$, nenhuma ação é tomada. Caso contrário, para $y^A = y^B$, é aplicada uma das regras de aprendizado, que são (PRABAKARAN; P., 2010):

- Regra de aprendizado de Hebb:

$$w_i^{A/B}(t+1) = w_i^{A/B}(t) + x_i y^{A/B} \Theta(y^{A/B} o_i^{A/B}) \Theta(y^A y^B) \tag{3.6}$$

- Regra de aprendizado anti-Hebb:

$$w_i^{A/B}(t+1) = w_i^{A/B}(t) - x_i o_i \Theta(y^{A/B} o_i^{A/B}) \Theta(y^A y^B) \tag{3.7}$$

- Regra de aprendizado Passeio Aleatório

$$w_i^{A/B}(t+1) = w_i^{A/B}(t) + x_i \Theta(y^{A/B} o_i^{A/B}) \Theta(y^A y^B) \tag{3.8}$$

Onde Θ é a função degrau,

$$\Theta(x) = \frac{1 + sgn(x)}{2} = \begin{cases} 0, & x < 0 \\ \frac{1}{2}, & x = 0 \\ 1, & x > 0 \end{cases} \tag{3.9}$$

dessa forma, apenas são atualizadas as unidades onde $o_i = y^{A/B}$ quando $y^A = y^B$. Essa restrição na atualização dos pesos é especialmente útil, já que torna impossível saber quais pesos foram atualizados sem conhecer os seus valores na camada escondida (FILHO, 2009).

Os passos da sincronização devem ser repetidos até que as duas redes estejam sincronizadas.

O processo de sincronização é baseado na competição entre forças aleatórias atrativas e repulsivas. Um passo atrativo ocorre quando $y^A = o_i^A = o_i^B = y^B$, situação onde os pesos de ambas as redes são atualizados. Com os pesos das redes entre $-L$ e $+L$, a distância $d_i = |w_i^A - w_i^B|$ não será modificada, exceto caso o valor de um dos pesos ultrapasse L, caso em que atribuído ao mesmo o valor limitante, o que faz com que a distância d_i diminua, até que $d_i = 0$ (FILHO, 2009).

Já um passo repulsivo ocorre quando $y^A = y^B$ mas $o_i^A \neq o_i^B$, e nessa situação apenas um dos pesos é atualizado, o que aumenta a distância $d_i = |w_i^A - w_i^B|$.

Ruttor (2007, p. 37) define que, dado a influência da comunicação entre A e B para a sincronização, e considerando que uma rede atacante E não interfira nessa comunicação, a probabilidade da ocorrência de passos atrativos entre A e B é sempre maior que entre qualquer uma das redes parceiras e E, da mesma forma que a probabilidade de passos repulsivos é menor entre A e B que entre uma das redes parceiras e E.

Para medir a sincronização entre duas TPMs em máquinas distintas, porém, não é possível usar os pesos $w_i^{A/B}$, já que o estado atual dos pesos é secreto para cada TPM e não pode ser transmitido por meio inseguro.

Para determinar o momento de sincronização, é proposto que seja feito um teste cifrando uma mensagem pré-determinada com um algoritmo criptográfico usando como chave o estado dos pesos de A e B e comparando-os, de forma que se a mensagem cifrada m^A seja igual à m^B, então A e B estão sincronizados. Como otimização desse algoritmo, acrescenta-se a regra de que o teste de sincronização deve ser executado apenas caso a condição $y^A = y^B$ tenha ocorrido nos últimos M passos.

3.3 Ataques à Criptografia Neural

O principal problema para um atacante E é que a representação interna $(o_1, o_2, \ldots, o_i)$ de A e B lhe é desconhecida. Como as alterações nos pesos depende dos valores de o_i, é importante para um ataque bem sucedido que o estado das unidades ocultas seja adivinhado corretamente.

Ataques de força bruta são computacionalmente inviáveis contra o protocolo, pois para determinada TPM há $(2L+1)^{KN}$ diferentes configurações possíveis de pesos.

Há quatro principais formas de ataque; simples, geométrico, de maioria e o genético.

3.3.1 Ataque Simples

No ataque simples, E treina uma terceira TPM com os vetores públicos x e com as saídas y^A, que são facilmente obtidas, pois são transmitidas publicamente.

A TPM de E deve ter a mesma estrutura de A e B e inicia com pesos aleatórios (RUTTOR, 2007).

A rede neural de E é treinada por uma das seguintes equações:

- Regra de aprendizado de Hebb:

$$w_i^E(t+1) = w_i^E(t) + x_i y^E \Theta(y^A o_i^E)\Theta(y^A y^B) \tag{3.10}$$

- Regra de aprendizado anti-Hebb:

$$w_i^E(t+1) = w_i^E(t) - x_i o_i \Theta(y^A o_i^E)\Theta(y^A y^B) \tag{3.11}$$

- Regra de aprendizado Passeio Aleatório

$$w_i^E(t+1) = w_i^E(t) + x_i \Theta(y^A o_i^E)\Theta(y^A y^B) \tag{3.12}$$

3.3.2 Ataque Geométrico

O ataque geométrico é o com taxa de sucesso mais alta entre os ataques onde E possui uma única TPM (RUTTOR, 2007).

É similar ao ataque simples, porém a regra de aprendizado só é aplicada caso $y^E = y^A = y^B$. Caso $y^E \neq y^A = y^B$, o atacante não poderá impedir a atualização dos pesos nas outras redes, mas tentará corrigir sua representação interna para obter a mesma saída, antes de atualizar seus pesos, trocando o valor de sua saída y^E pelo valor da saída de um neurônio da camada escondida (FILHO, 2009, p. 30).

3.3.3 Ataque de Maioria

No ataque de maioria, o atacante usa m TPMs para melhorar sua capacidade de predição. As m redes são inicializadas com pesos aleatórios e quando a saída de uma determinada rede y_i^E for diferente de $y^{A/B}$, E tenta corrigir sua representação da mesma forma que o ataque geométrico. Após a correção, o atacante E seleciona a representação interna mais comum, e esta será adotada por todas as redes na regra de aprendizagem (FILHO, 2009).

Para reduzir a correlação que surge entre as TPMs de E devido às atualizações idênticas e aumentar a eficiência, o atacante pode usar o ataque de maioria e o ataque geométrico alternadamente (RUTTOR, 2007).

3.3.4 Ataque Genético

O ataque genético é baseado em um algoritmo evolucionário, em que E começa com apenas uma TPM, mas pode usar até m redes neurais. Quando $y^A = y^B$ o seguinte algoritmo genético é

aplicado:

- Caso E tenha até $\frac{m}{2^{K-1}}$ Tree Parity Machines, ele determina todas as 2^{K-1} representações internas $(o_1, o_2, \ldots, o_i)$ que produzem a saída y^A e os usa para atualizar os pesos de acordo com a regra de aprendizado. Assim E cria 2^{K-1} variantes de cada TPM nesse *passo de mutação* (RUTTOR, 2007).

- Caso E já tenha mais que $\frac{m}{2^{K-1}}$ TPMs, só as mais aptas devem ser mantidas. E isso é obtido descartando todas as redes que predisseram menos que U saídas y^A nos últimos V passos de aprendizagem em que $y^A = y^B$ no *passo de seleção*. Como regra adicional, E mantém ao menos 20 TPMs.

3.4 Melhoria na Segurança da Criptografia Neural

A segurança da criptografia neural é determinada pelo valor de L, onde a probabilidade de sucesso de um atacante cai exponencialmente com o incremento de L enquanto o tempo de sincronização cresce apenas linearmente (RUTTOR, 2007).

Porém, essa segurança é comprometida no caso do ataque de maioria, onde foi mostrado que a probabilidade de sucesso é constante, independente do valor de L. Para recuperar a segurança da criptografia neural, Ruttor, Kinzel e Kanter (2005) propôs melhorias para o algoritmo usando *aprendizado por queries*.

Os parceiros A e B podem tornar a estratégia de aprendizado do ataque geométrico inválida e, por consequência, aumentar a segurança da criptografia neural usando um parametro extra H, com valores de entrada selecionados alterando a frequência de passos repulsivos na sincronização.

Dessa forma, é possível para A e B ajustarem a frequência de passos repulsivos, regulando a dificuldade de sincronização e aprendizado.

Outra melhoria ao protocolo foi proposta por Allam e Abbas (2010) com o uso de transmissões errôneas entre as TPMs parceiras. Nessa proposta, as TPMs enviam ocasionalmente, com probabilidade baseada na distância estimada entre os pesos de A e B, o valor errado de saída da rede e a rede parceira tentaria predizer o envio dessa informação e corrigi-lo. Como a probabilidade é definida por parâmetros internos das redes sincronizantes, A e B tem uma chance maior de predizer o envio da mensagem errônea e corrigir do que é possível para um atacante E.

4 Implementação e Resultados

Para validar o funcionamento do protocolo de troca de chaves com criptografia neural, foi criada uma implementação do algoritmo na linguagem Python. Nenhuma biblioteca adicional à padrão da linguagem foi necessária para a implementação das redes TPM. Para a implementação da sincronização em rede, porém, foram utilizadas as bibliotecas Twisted e PyCrypto, ambas de código aberto e gratuita, que forneceu a implementação dos algoritmos de criptografia utilizados no trabalho. O ambiente utilizado para os testes foi uma máquina com processador *AMD Athlon II X2 processor M300 2.0 GHz*, com 4 gigabytes de memória RAM e placa gráfica *ATI Radeon HD4200 Graphics* com o sistema operacional *Windows 7 Professional 64 bits*, configurado no modo de segurança e usando a versão 2.7.2 do Python.

A implementação principal consiste de uma rede neural TPM, que é iniciada com os parâmetros N, K e L que definem, respectivamente: o número de neurônios, a quantidade de entradas de cada neurônio, e o valor limite dos neurônios.

4.1 Desenvolvimento da Tree Parity Machine

A *Tree Parity Machine* foi programada seguindo os passos descritos no capítulo 3. Assim, foram criados os neurônios da camada oculta, os quais foram agrupados na estrutura da TPM. Em sequência, foi implementado o algoritmo de sincronização usando duas instâncias das redes neurais.

4.1.1 Regras de Aprendizado

Foram apresentadas no capítulo 3 três regras de aprendizado que podem ser utilizadas em conjunto com a técnica de criptografia neural. Essas regras foram implementadas no projeto pelas classes *Hebbian*, *AntiHebbian* e *RandomWalk*, disponíveis no Apêndice A.

Cada classe implementa um método em que recebem os parâmetros w, x, y e o e retornam

o valor do peso ajustado, onde os parâmetros correspondem respectivamente ao peso, o valor da entrada correspondente a esse peso, à saída da TPM e à saída do neurônio correspondente. Esses métodos implementam os cálculos descritos nas equações 3.6, 3.7 e 3.8.

A classe de aprendizado é definida para uma TPM no momento de inicialização da mesma.

4.1.2 Unidades Ocultas

Cada neurônio, ou unidade oculta, é representado na implementação por um objeto da classe *Unit* e é instanciado com os parâmetros N, L e *lrule*, que definem, respectivamente, a quantidade de entradas do neurônio, os limites mínimo e máximo para os valores dos pesos e a regra de aprendizado que será aplicada ao neurônio.

As unidades ocultas são criadas pela TPM na sua inicialização na quantidade definida por K. No momento da criação, a unidade tem seu vetor de pesos w criado com tamanho N e inicializado com valores aleatórios entre $-L$ e $+L$ e a regra de aprendizado *lrule* definida como a mesma da TPM.

A implementação das unidades ocultas consta no Apêndice A, onde foram implementadas com métodos que:

- Processam um vetor de entradas e retornam a saída do neurônio
- Realizam o treinamento do neurônio, que irá ajustar os pesos da unidade aplicando a regra de aprendizado sobre os parâmetros x, vetor de entradas, e y, saída da TPM, recebidos.

4.1.3 Tree Parity Machine

As TPMs são implementadas pela classe *TreeParityMachine* e são instanciadas com os parâmetros K, N, L e $lrule$, que são, respectivamente, a quantidade de unidades ocultas que a rede possuirá, a quantidade de entradas que cada unidade oculta irá ter, o valor máximo e mínimo que cada peso das unidades ocultas poderá assumir e a regra de aprendizado que será utilizada. Durante a inicialização do objeto as unidades ocultas correspondentes ao mesmo também são instanciadas e inicializadas.

As TPMs são capazes de processar a saída da rede com base em um vetor de entradas e de realizar o treinamento e ajustar seus pesos se necessário através do vetor de entradas e da saída da rede parceira.

A implementação da TPM encontra-se no Apêndice A.

4.1.4 Fluxo de sincronização

No início do processo de sincronização, duas instâncias de *TreeParityMachine* são criadas, representando as redes parceiras A e B.

A partir de então, para cada iteração, A gera um vetor de entradas, A e B processam o vetor entradas e calculam suas saídas. As saídas são então comparadas e se forem diferentes, uma nova iteração se inicia, caso contrário as redes são treinadas. É então verificado se as TPM estão sincronizadas e caso não estejam, uma nova iteração é inciada, caso contrário, a sincronização está concluída.

A partir desse instante, é possível gerar a chave criptográfica a partir dos valores dos pesos, por exemplo, os concatenando.

A implementação desse fluxo de sincronização pode ser encontrada no Apêndice B e visualizada no diagrama de atividades da Figura 4.1.

Para a sincronização via rede o algoritmo deve ser modificado para ser distribuído em duas máquinas parceiras, e a detecção de sincronização das redes deve ser alterada para possibilitar que se detecte a sincronização sem haver conhecimento dos pesos da rede parceira.

Para a detecção dessa sincronização foi criado um contador que é incrementado sempre que $y^A = y^B$ e zerado sempre que $y^A \neq y^B$. Quando esse contador atinge determinado limiar, um teste de sincronização é feito com A enviando uma mensagem pré-definida e criptografada com seus pesos para B, que a descriptografa e verifica se o conteúdo descriptografado é igual ao conteúdo esperado.

Para criar a chave com base nos pesos foi gerado um hash sobre os pesos da TPM usando o algoritmo SHA-256 e esse hash resultante foi usado como chave juntamente com o algoritmo AES para criptografar a mensagem pré definida.

O texto criptografado é então transmitido para a rede parceira, que deve executar o mesmo procedimento de aplicar o algoritmo SHA-256 sobre os pesos e então usar o algoritmo AES para descriptografar a mensagem. Assim, se, e somente se, os pesos de A e de B forem iguais, o hash gerado será igual e, por consequência, o algoritmo AES terá a chave correta e conseguirá descriptografar o texto cifrado.

O fluxo completo de sincronização em rede ocorre da seguinte forma: o lado da comunicação do *cliente*, com a TPM A, gera um vetor de entradas, as processa e envia para o *servidor*, com a TPM B. A TPM B então processa as entradas e envia sua saída de volta para A, que a compara com sua própria saída, incrementa o contator e treina caso sejam iguais, e envia sua

Fonte: próprio autor

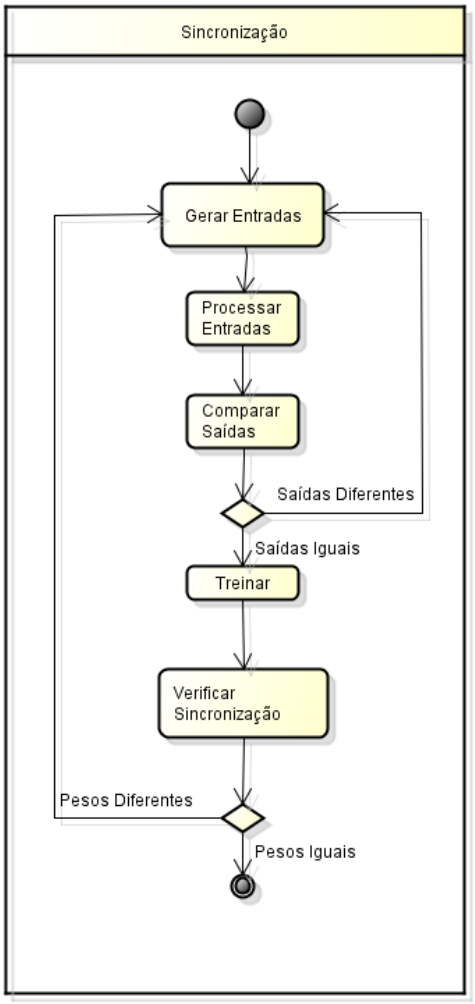

Figura 4.1: Diagrama de Atividades do Fluxo de Sincronização das redes Tree Parity Machine parceiras.

saída para B.

No servidor ocorre novamente uma comparação entre as saídas e treina a TPM B caso sejam iguais, retornando uma mensagem "*Ack*" após o processo.

Quando o cliente recebe o "*Ack*", verifica se o contador de teste de sincronização atingiu o limiar definido. Caso não tenha atingido, uma nova iteração é iniciada. Caso contrário, o contador é zerado e é gerada uma chave criptográfica a partir dos pesos, que criptografa uma mensagem pré-definida e a envia para o servidor, gerando uma chave a partir de seus pesos e tenta descriptografar a mensagem. Caso essa descriptografia ocorra com sucesso, o servidor dá a sincronização como concluída e alerta o cliente do resultado. Caso contrário, uma nova iteração se inicia.

A partir desse momento, ambas as redes dão a sincronização como concluída e é possível

usar as chaves geradas para trocar mensagens criptografadas de forma segura.

O fluxo de troca de mensagens para a sincronização em rede pode ser observado com mais detalhes no diagrama de atividades da Figura 4.2 e encontra-se implementado no Apêndice C.

Fonte: próprio autor

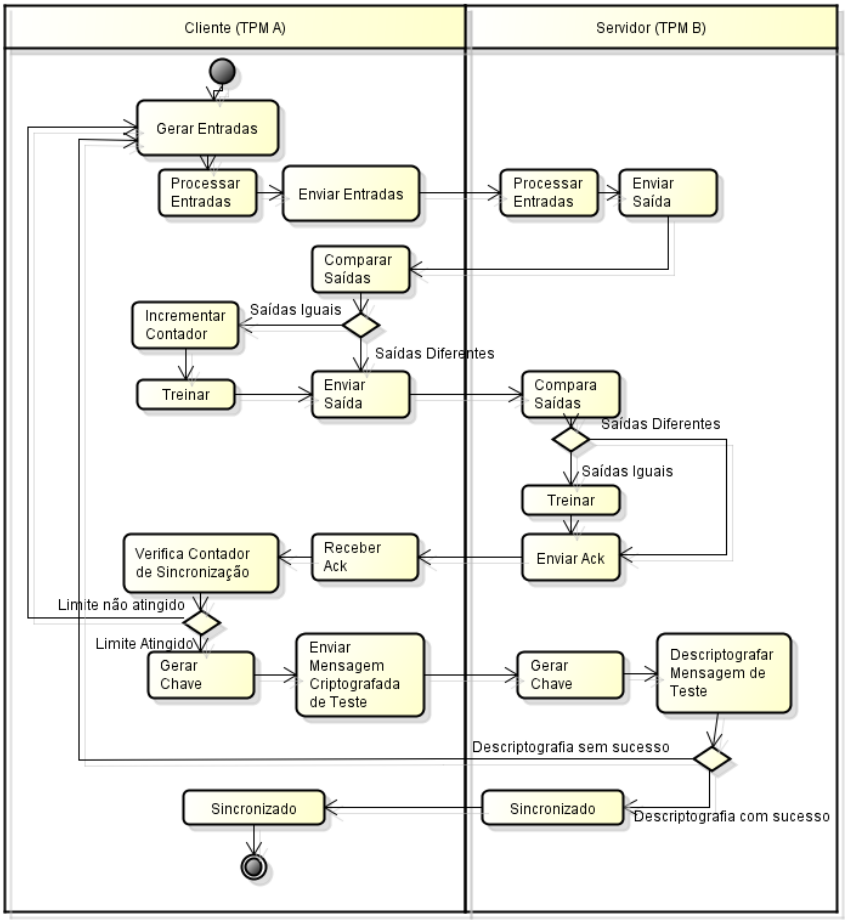

Figura 4.2: Diagrama de Atividades do Fluxo de Sincronização em rede das Tree Parity Machine parceiras.

4.2 Análise de performance da sincronização entre TPM

Foi analisada a performance de sincronização das redes neurais parceiras, através de 200 testes para cada situação, em relação aos parâmetros K: número de neurônios, N: número de entradas por neurônio e L: limites máximos e mínimos do neurônio e regra de aprendizado de Hebb. Para cada parâmetro foi medido o número de mensagens trocadas até que ocorresse a sincronização e o tempo de processamento total gasto para a ela.

A Figura 4.3 ilustra a variação no número de mensagens trocadas quando se altera o valor de L, e os reflexos dessa alteração no tempo de processamento encontram-se na Figura 4.4.

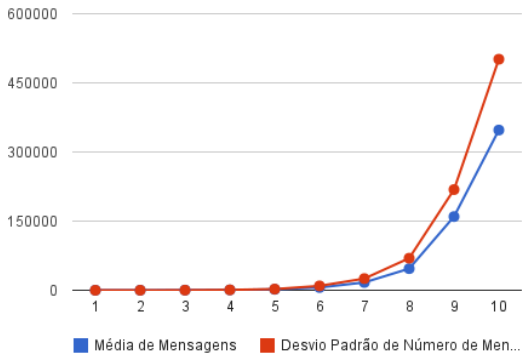

Figura 4.3: Número de mensagens trocadas até a sincronização com $K = 3$, $N = 4$ e variando em L.

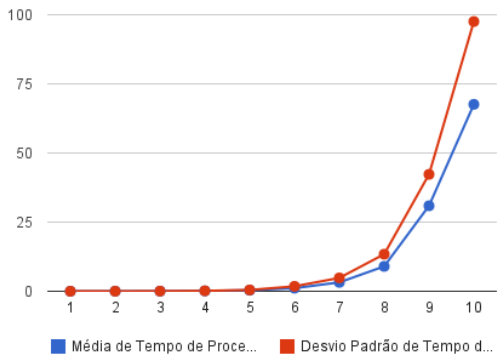

Figura 4.4: Tempo de processamento até a sincronização com $K = 3$, $N = 4$ e variando em L.

Nos testes é possível observar um aumento polinomial das mensagens trocadas, com o tempo de processamento acompanhando na mesma proporção, conforme o aumento de L. Cabe ressaltar que foi observado por Ruttor (2007, pg. 55) que a taxa de sucesso de um atacante cai exponencialmente com o incremento do parâmetro L.

O desvio padrão do número de mensagens aumenta consideravelmente quando o único parâmetro alterado é o L, superando a média em todos os casos verificados.

Mantendo o valor $L = 3$, $N = 4$ e variando o parâmetro K, obtemos os tempos de sincronização exibidos na Figura 4.5 e Figura 4.6, onde podemos observar que o aumento no número de mensagens trocadas para a sincronização não aumenta consideravelmente com o aumento de K, mantendo uma taxa estável com o crescimento desacelerando para valores maiores, enquanto o desvio padrão se manteve na mesma faixa de valores, com exceção de uma elevação para quando $K = 2$.

Ruttor (2007, pg. 58) verificou que a probabilidade de sucesso de um atacante E é menor para $K > 1$. Porém, quando $K > 3$, tem-se que o número de flutuações na sincronização, com a ocorrência de muitos passos repulsivos, é afetado fortemente com o aumento de L, tornando impraticáveis valores maiores que este. Devido a tanto, obtem-se os valores ideais para K com $K = 2$ e $K = 3$, onde $K = 3$ oferece a maior segurança.

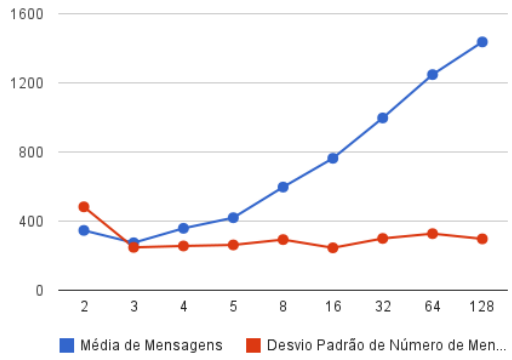

Figura 4.5: Número de mensagens trocadas até a sincronização com $L = 3$, $N = 4$ e variando em K.

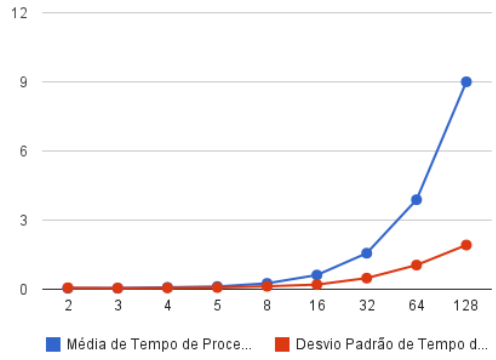

Figura 4.6: Tempo de processamento até a sincronização com $L = 3$, $N = 4$ e variando em K.

O tempo de processamento gasto na sincronização aumentou proporcionalmente com o aumento de K, sem variações.

De forma similar, nos testes sobre N foi possível observar que o tempo de processamento gasto na sincronização também aumentou proporcionalmente com o incremento do parâmetro, como pode ser visto na Figura 4.7.

A variação no número de mensagens trocadas para diferentes valores de N, como pode ser vista na Figura 4.8, é pequena, com uma média de 30 mesmo em saltos maiores como de 128

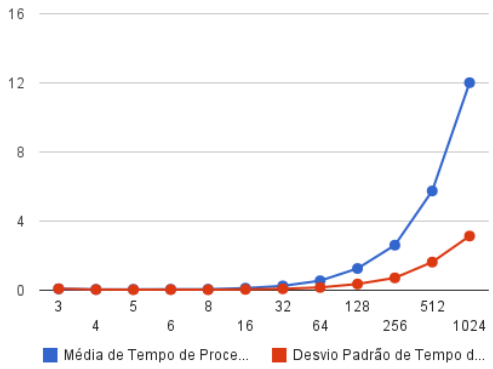

Figura 4.7: Tempo de processamento até a sincronização com $L = 3$, $K = 3$ e variando em N.

para 256 ou de 512 para 1024, o que indica uma tendência à estabilidade em valores maiores. Apesar disso, é notável um pico no número de mensagens para sincronização quando $N = 3$, com uma mudança brusca na transição para $N = 4$.

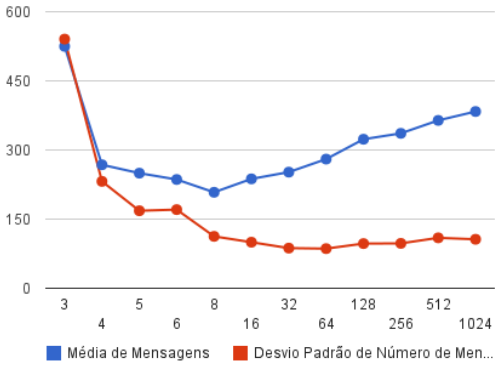

Figura 4.8: Número de mensagens trocadas até a sincronização com $L = 3$, $K = 3$ e variando em N.

A variação de K e N se reflete no tamanho do vetor de pesos gerado, que é diretamente ligado à segurança do protocolo, onde um vetor maior de pesos reflete em um maior número possível de configurações da rede.

Como temos definido que o valor ideal de K é 3, podemos alterar o parâmetro N para controlar o tamanho da chave gerada sem afetar negativamente a sincronização. A variação do tamanho da chave, porém, afeta a segurança em menor escala que a de L, em relação as técnicas de ataque especializadas apresentadas.

Por último, considerando as redes neurais Tree Parity Machine configuradas com $K = 3$ e $N = 16$, foram realizados testes comparando a performance da rede com as três diferentes

regras de treinamento, variando L. O gráfico com o resultado dos testes pode ser observado na Figura 4.9.

Fonte: próprio autor

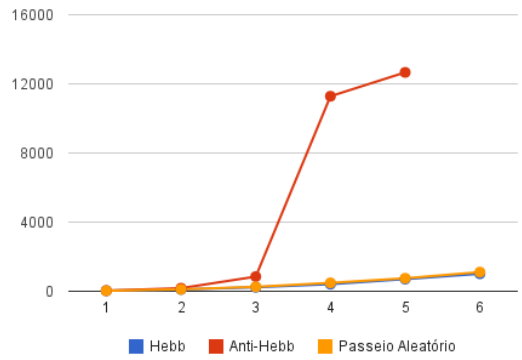

Figura 4.9: Número de mensagens trocadas com diferentes regras de aprendizado com variância de L.

É possível observar na Figura 4.9 que a regra de treinamento de Hebb é a mais performática, seguida pela regra do Passeio Aleatório por uma diferença média de 9% a mais de mensagens.

A regra Anti-Hebb apresentou um comportamento diferente, com um salto no número de mensagens de 470% entre $L = 2$ e $L = 3$ e de 13300% entre $L = 3$ e $L = 4$. Com $L = 5$ as redes tiveram 99,5% de problemas com sincronia e 100% de falhas na sincronização com $L = 6$, numa situação em que as redes parceiras produziam saídas diferentes em 100% das vezes, que inviabiliza o treinamento e posterior ajuste dos pesos.

Foi possível observar, através da realização de testes adicionais, que essa proporção de falhas pode ser controlada e reduzida com o aumento do parâmetro N. Com $K = 3$, $N = 100$ e $L = 5$ as redes conseguiram sincronizar em 100% das vezes usando a regra de aprendizado Anti-Hebb.

As simulações não consideraram o tempo gasto na comunicação para o transporte de mensagens via rede porque esse tempo é muito susceptível a variações na qualidade da comunicação, onde situações como perda de pacotes devido a baixa qualidade da comunicação, alta latência ou limitações de velocidade de banda possuem forte influência.

O tempo de comunicação, porem, deve ser considerado na escolha dos parâmetros das redes, já que influi fortemente no tempo total de comunicação principalmente quando se varia L, onde o número de mensagens necessárias para sincronização pode aumentar consideravelmente.

Em testes realizados na mesma máquina, utilizando a implementação do algoritmo de crip-

tografia de chave pública RSA, com tamanho de chave de 1024 bits, da biblioteca PyCrypto, obteve-se a média de tempo de 0,032 segundo para transferir uma chave criptográfica simétrica, com a melhor aproximação ao tempo de sincronização sendo de duas TPM configuradas com $K = 3$, $N = 4$ e $L = 3$, com média de tempo de 0,048.

Cabe ressaltar que trata-se de uma implementação do algoritmo RSA já otimizada e implementada em linguagem C, que tende a ser até 700 vezes mais rápidas que Python em algumas simulações (SCYPY.ORG, 2011). Portanto uma melhor comparação poderia ser feita caso o algoritmo de sincronização de redes neurais fosse desenvolvido também em linguagem C.

4.3 Conclusões

Foi possível verificar a sincronização de redes neurais através da implementação criada. A implementação criada foi capaz de utilizar duas redes neurais na arquitetura Tree Parity Machine, conforme proposto por Kinzel e Kanter (2002) e sincronizar seus pesos através do algoritmo proposto.

Foram realizados testes com diversas configurações de TPMs, onde pode-se observar uma grande elevação no número de mensagens e tempo de processamento gasto em valores maiores de L, que inviabiliza valores maiores deste parâmetro em futuras aplicações.

Pode-se concluir através da comparação de tempo entre as três variações de treinamento que a regra de aprendizado de Hebb é a que possui melhor performance.

No desenvolvimento dos testes observou-se grande variação, tanto no tempo de processamento requerido para a sincronização como na quantidade de mensagens trocadas entre as redes. Caberia um estudo posterior também para definir configurações da rede e sua equivalência em termos de segurança quando comparado à outras soluções de troca de chave.

O algoritmo desenvolvido foi trabalhado com uma aplicação simples, sendo capaz apenas de sincronizar as redes neurais e criar uma chave criptográfica a partir do peso das mesmas. O uso dessa chave, porém, é de utilização livre entre as partes comunicantes, podendo ser usado como chave criptográfica de um algoritmo de criptografia simétrico, como o AES utilizado no trabalho. A partir de então, cabe a aplicação usuária do protocolo utilizar a chave gerada para cifrar as mensagens conforme seu propósito.

Nos testes executados não foi possível comparar efetivamente a velocidade do algoritmo implementado com a de algoritmos de criptografia de chave pública, já que o trabalho foi desenvolvido na linguagem Python e as implementações encontradas de outros algoritmos são em

linguagem C, reconhecidamente mais performática.

O algoritmo de criptografia neural implementado é uma das alternativas ao problema de troca de chaves. Tem como característica precisar, em geral, de um número mais baixo de cálculos que alternativas de chave pública (KANTER; KINZEL; KANTER, 2002). Adicionalmente, a criptografia neural não possui algumas das desvantagens encontradas em outras técnicas, como a troca de chave com antecedência, que é logísticamente inviável, e que o uso de um terceiro confiável, que depende exclusivamente de uma máquina mestre com acesso a todas as informações.

Como trabalho futuro, sugere-se a implementação do protocolo em linguagem C para a realização de comparação entre a performance e segurança da criptografia neural em relação à criptografia de chave pública juntamente com a implementação das melhorias já propostas ao protocolo, sendo o uso de queries e a transmissão de informações errôneas com predição. Também se sugere a implementação do protocolo em microcontroladores e dispositivos computacionais mais limitados, como por exemplo, telefones celulares.

Referências Bibliográficas

AHN, L. V. et al. Captcha: using hard ai problems for security. In: *Proceedings of the 22nd international conference on Theory and applications of cryptographic techniques*. Berlin, Heidelberg: Springer-Verlag, 2003. (EUROCRYPT'03), p. 294–311. ISBN 3-540-14039-5. Disponível em: <http://portal.acm.org/citation.cfm?id=1766171.1766196>.

ALLAM, A. M.; ABBAS, H. M. On the improvement of neural cryptography using erroneous transmitted information with error prediction. *Trans. Neur. Netw.*, IEEE Press, Piscataway, NJ, USA, v. 21, p. 1915–1924, December 2010. ISSN 1045-9227. Disponível em: <http://dx.doi.org/10.1109/TNN.2010.2079948>.

BELLMAN, R. E. An introduction to artificial intelligence: Can computers think? Boyd & Fraser Publishing Company, 1978.

BISHOP, M. *Computer security: art and science*. [S.l.]: Addison-Wesley, 2003. ISBN 9780201440997.

BRAGA, A.; CARVALHO, A.; LUDERMIR, T. *Redes Neurais Artificiais: Teoria e Aplicações*. 2. ed. [S.l.]: LTC, 2007. ISBN 9788521615644.

BURNETT, S.; PAINE, S. *The RSA Security's Official Guide to Cryptography*. Berkeley, CA, USA: Osborne/McGraw-Hill, 2001. ISBN 0072194049.

ENGELBRECHT, A. *Computational Intelligence: An Introduction*. New York, NY, USA: Halsted Press, 2002. ISBN 0470848707.

FILHO, J. F. Implementação e análise de desempenho dos protocolos de criptografia neural e diffie-hellman em sistemas rfid utilizando uma plataforma embarcada. *Universidade Federal do Rio Grande do Norte*, p. 61, 2009.

FRANK, J.; MDA-C, N. U. Artificial intelligence and intrusion detection: Current and future directions. In: *In Proceedings of the 17th National Computer Security Conference*. [S.l.: s.n.], 1994.

GOLDREICH, O. *Foundations of Cryptography: Basic Tools*. New York, NY, USA: Cambridge University Press, 2000. ISBN 0521791723.

HAYKIN, S. *Neural networks: a comprehensive foundation*. [S.l.]: Prentice Hall, 1999. ISBN 9780132733502.

KANTER, I.; KINZEL, W.; KANTER, E. Secure exchange of information by synchronization of neural networks. *Europhysics Letters*, IOP Publishing, v. 57, n. 1, p. 11, 2002. Disponível em: <http://arxiv.org/abs/cond-mat/0202112>.

KINZEL, W.; KANTER, I. Neural cryptography. In: *in Proc. of the 9th International Conference on Neural Information Processing*. [S.l.: s.n.], 2002. p. 18–22.

KURZWEIL, R. The age of intelligent machines. MIT Press, 1990.

MENEZES, A. J.; VANSTONE, S. A.; OORSCHOT, P. C. V. *Handbook of Applied Cryptography*. 1st. ed. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN 0849385237.

NILSSON, N. J. Artificial intelligence: A new synthesis. Morgan Kaufmann, 1998.

PFLEEGER, C. P.; PFLEEGER, S. L. *Security in Computing (4th Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006. ISBN 0132390779.

POOLE, D.; MACKWORTH, A. K.; GOEBEL, R. *Computational Intelligence: A logical approach*. [S.l.]: Oxford University Press, 1998.

PRABAKARAN, N.; P., V. A new security on neural cryptography with queries. *Int. J. of Advanced Networking and Applications*, p. 437–444, 2010.

ROWLEY, H. A. *Neural network-based face detection*. Tese (Doutorado), Pittsburgh, PA, USA, 1999. AAI9950035.

RUSSELL, S.; NORVIG, P. *Artificial intelligence: a modern approach*. [S.l.]: Prentice Hall, 2010. (Prentice Hall series in artificial intelligence). ISBN 9780136042594.

RUTTOR, A. Neural synchronization and cryptography. *arXiv*, p. 120, 2007. Disponível em: <http://arxiv.org/abs/0711.2411>.

RUTTOR, A.; KINZEL, W.; KANTER, I. Neural Cryptography with Queries. nov. 2004. Disponível em: <http://arxiv.org/abs/cond-mat/0411374>.

Ruttor, A.; Kinzel, W.; Kanter, I. Neural cryptography with queries. *Journal of Statistical Mechanics: Theory and Experiment*, v. 1, p. 9–+, jan. 2005.

RUTTOR, A. et al. Genetic attack on neural cryptography. *Physical Review E - Statistical, Nonlinear and Soft Matter Physics*, APS, v. 73, n. 3 Pt 2, p. 036121, 2006. Disponível em: <http://arxiv.org/abs/cond-mat/0512022>.

SCYPY.ORG. *Performance Python*. 6 2011. Disponível em: <http://www.scipy.org/PerformancePython>.

SHACHAM, L. N. et al. *Cooperating Attackers in Neural Cryptography*. [S.l.], Dec 2003.

TERADA, R. *Segurança de Dados: Criptografia em Rede de Computador*. 2nd. ed. [S.l.]: Edgard Blucher, 2008. ISBN 9788521204398.

WINSTON, P. H. *Artificial Intelligence*. 3. ed. [S.l.]: Addison-Wesley, 1992.

APÊNDICE A – Implementação da Tree Parity Machine

Código A.1: Implementação de uma Tree Parity Machine

```
1  # -*- coding: utf-8 -*-
2  import random
3  import math
4  from lrule import *
5  from unit import Unit
6
7
8  class TreeParityMachine(object):
9
10     def __init__(self, K=3, N=4, L=3, lrule=Hebbian):
11         self.units = []
12         self.K = K
13         self.N = N
14         self.y = None
15         for unit in range(K):
16             self.units.append(Unit(N, L, lrule))
17
18     def __call__(self, x):
19         self.y = 1
20         self.x = x
21         x = self._chunks(x, self.N)
22         for unit, xi in zip(self.units, x):
23             self.y = self.y * unit(xi)
24         return self.y
25
26     def _chunks(self, l, chunk_size):
27         offset = 0
28         chunks = []
29         for i in range(len(l)/chunk_size):
30             chunk = []
```

```
31              for j in range(chunk_size):
32                  chunk.append(l[offset + j])
33              offset = offset + j + 1
34              chunks.append(chunk)
35          return chunks
36
37      def activation(self, y):
38          return (self.y == y)
39
40      def train(self, x=None):
41          x = x or self.x
42          x = self._chunks(x, self.N)
43          for unit, xi in zip(self.units, x):
44              unit.train(xi, self.y)
45
46      def weights(self):
47          w = []
48          for unit in self.units:
49              for wi in unit.w:
50                  w.append(wi)
51          return w
52
53      def outputs(self):
54          w = []
55          for unit in self.units:
56                  w.append(unit.o)
57          return w
58
59      def generate_inputs(self):
60          self.x = [[-1,1][random.randint(0,1)] for whatever in range(self.K
                * self.N)]
61          return self.x
```

Código A.2: Implementação das unidades ocultas

```
1  # -*- coding: utf-8 -*-
2  import random
3  import math
4  from lrule import *
5
6  class Unit(object):
7
8      def __init__(self, N=4, L=3, lrule=Hebbian):
9          self.lrule = lrule()
```

```
10          self.w = [random.randint(-L, +L) for _ in range(N)]
11          self.L = L
12
13      def __call__(self, x):
14          h = 0
15          for wi, xi in zip(self.w, x):
16              h = h + (wi * xi)
17          self.o = self._sgn(h)
18          return self.o
19
20      def _sgn(self, v):
21          if v <= 0:
22              return -1
23          else:
24              return 1
25
26      def train(self, x, y):
27          for i in range(len(self.w)):
28              wi = self.lrule(self.w[i], x[i], y, self.o)
29              self.w[i] = self._adjust(wi)
30
31      def _adjust(self, wi):
32          if wi > self.L:
33              return self.L
34          elif wi < (-self.L):
35              return -self.L
36          else:
37              return wi
```

Código A.3: Implementação das regras de aprendizado

```
1  # -*- coding: utf-8 -*-
2  import random
3  import math
4
5  class LearningRule:
6
7      def step(self, y, o):
8          if (y * o) <= 0:
9              return 0
10         else:
11             return 1
12
13
```

```
14 class Hebbian(LearningRule):
15
16     def __call__(self, w, x, y, o):
17         return w + (x * y) * self.step(y, o)
18
19
20 class AntiHebbian(LearningRule):
21
22     def __call__(self, w, x, y, o):
23         return w - (x * o) * self.step(y, o)
24
25
26 class RandomWalk(LearningRule):
27
28     def __call__(self, w, x, y, o):
29         return w + x * self.step(y, o)
```

APÊNDICE B – Implementação do protocolo de Criptografia Neural local

Código B.1: Implementação do Fluxo de Sincronização de duas TPMs localmente

```
from tpm import *
import time

K = 3
N = 16
L = 3

c = 0
i = 0

print "----------------------------------------"
print "Sincronizando"
print "----------------------------------------"
A = TreeParityMachine(K, N, L, lrule)
B = TreeParityMachine(K, N, L, lrule)
start = time.time()

while A.weights() != B.weights():
        A.generate_inputs()
        t = t + 1
        A(A.x)
        B(A.x)

        if A.activation(B.y):
                A.train(A.x)
                B.train(A.x)

print t, "\t", (time.time() - start)
print "Sincronizado!"
```

APÊNDICE C – Implementação do protocolo de Criptografia Neural em rede

Código C.1: Classe NeuralCryptography implementa os métodos herdáveis do protocolo de Criptografia Neural para o cliente e servidor

```
1  from twisted.internet.protocol import Protocol
2  from twisted.internet.protocol import Factory
3  from twisted.internet.endpoints import TCP4ServerEndpoint
4  from twisted.internet import reactor
5  from Crypto.Cipher import AES
6  from Crypto.Hash import SHA256
7  from threading import Thread
8  from tpm import *
9  import json
10 import base64
11
12 # Used to access the index in the exchanged messages
13 MESSAGE_TYPE = 0
14 MESSAGE = 1
15
16 # The types of the messages exchanged
17 TYPE_INPUTS = 0
18 TYPE_OUTPUT_SERVER = 1
19 TYPE_OUTPUT_CLIENT = 2
20 TYPE_ACK = 3
21 TYPE_TEST = 4
22 TYPE_TEST_OK = 5
23 TYPE_MESSAGE = 6
24
25 SYNC_THRESHOLD = 20
26 TEST_MESSAGE = 'SYNCED'
27
28 class NeuralCryptography(Protocol):
29
```

```
30      def __init__(self):
31          self.tpm = TreeParityMachine(4,3);
32          self.count = 0
33          self.syncronized = False
34          self.key = None
35          self.cipher = None
36
37      def syncronizer(self, data):
38          data = json.loads(data)
39          if self.count == SYNC_THRESHOLD:
40              self.test_sync()
41          elif data[MESSAGE_TYPE] == TYPE_INPUTS:
42              self.receive_inputs(data[MESSAGE])
43          elif data[MESSAGE_TYPE] == TYPE_OUTPUT_SERVER:
44              self.receive_output_from_server(data[MESSAGE])
45          elif data[MESSAGE_TYPE] == TYPE_OUTPUT_CLIENT:
46              self.receive_output_from_client(data[MESSAGE])
47          elif data[MESSAGE_TYPE] == TYPE_ACK:
48              self.receive_ack()
49          elif data[MESSAGE_TYPE] == TYPE_TEST:
50              self.receive_test(data[MESSAGE])
51          elif data[MESSAGE_TYPE] == TYPE_TEST_OK:
52              self.receive_test_ok()
53
54      def receive_inputs(self, inputs):
55          self.tpm(inputs)
56          self.transport.write(json.dumps([TYPE_OUTPUT_SERVER, self.tpm.y]))
57
58      def receive_output_from_server(self, output):
59          self.transport.write(json.dumps([TYPE_OUTPUT_CLIENT, self.tpm.y]))
60          if self.tpm.y == output:
61              self.count += 1
62              self.tpm.train()
63          else:
64              self.count = 0
65
66      def receive_output_from_client(self, output):
67          if self.tpm.y == output:
68              self.count += 1
69              self.tpm.train()
70          else:
71              self.count = 0
72          self.transport.write(json.dumps([TYPE_ACK, 0]))
```

```
73
74      def receive_ack(self):
75          self.tpm.generate_inputs()
76          self.tpm(self.tpm.x)
77          self.transport.write(json.dumps([TYPE_INPUTS, self.tpm.x]))
78
79      def synced(self):
80          return self.syncronized
81
82      def test_sync(self):
83          self.count = 0
84          self.generate_key()
85          self.cipher = AES.new(self.key, AES.MODE_CBC)
86
87          ciphertext = self.cipher.encrypt(self.pad(TEST_MESSAGE.encode('utf
                -8')))
88          ciphertext = base64.b64encode(ciphertext)
89          self.transport.write(json.dumps([TYPE_TEST, ciphertext]))
90
91      def receive_test(self, ciphertext):
92          self.generate_key()
93          self.cipher = AES.new(self.key, AES.MODE_CBC)
94          ciphertext = base64.b64decode(ciphertext)
95          plaintext = self.cipher.decrypt(ciphertext)
96          plaintext = self.unpad(plaintext)
97          if plaintext == TEST_MESSAGE:
98              self.transport.write(json.dumps([TYPE_TEST_OK, TEST_MESSAGE]))
99              self.syncronized = True
100             print self.tpm.weights()
101             self.start_service()
102         else:
103             self.transport.write(json.dumps([TYPE_ACK, 0]))
104
105     def receive_test_ok(self):
106         self.syncronized = True
107         self.start_service()
108         print self.tpm.weights()
109
110     def generate_key(self):
111         seed = str(self.tpm.weights())
112         sha = SHA256.new()
113         sha.update(seed)
114         self.key = sha.digest()
```

```
115         return self.key
116
117     def pad(self, s):
118         BS = 16
119         return s + (BS - len(s) % BS) * chr(BS - len(s) % BS)
120
121     def unpad(self, s):
122         return s[0:-ord(s[-1])]
123
124     def call(self, target, args):
125         self.thread = Thread(target=target, args=(args))
126
127     def receive(self, target):
128         self.data_received = target
129
130     def start_service(self):
131         self.thread.start()
132
133     def received(self, data):
134         data = json.loads(data)
135         ciphertext = data[MESSAGE]
136         ciphertext = base64.b64decode(ciphertext)
137         plaintext = self.cipher.decrypt(ciphertext)
138         plaintext = self.unpad(plaintext)
139         self.data_received(plaintext)
140
141     def send_message(self, data):
142         ciphertext = self.cipher.encrypt(self.pad(data))
143         ciphertext = base64.b64encode(ciphertext)
144         self.transport.write(json.dumps([TYPE_MESSAGE, ciphertext]))
```

Código C.2: Classe NeuralCryptographyClient implementa o cliente na sincronização em rede

```
1  from twisted.internet.protocol import Protocol
2  from twisted.internet.protocol import Factory
3  from twisted.internet.endpoints import TCP4ClientEndpoint
4  from twisted.internet import reactor
5  from protocol import NeuralCryptography
6  from tpm import TreeParityMachine
7  import json
8
9  class NeuralCryptographyClient(NeuralCryptography):
10
11     def __init__(self):
```

```
12          NeuralCryptography.__init__(self)
13          self.call(target=chatLoop, args=(self,))
14          self.receive(target=printer)
15
16      def dataReceived(self, data):
17          if not self.synced():
18              self.syncronizer(data)
19          else:
20              self.received(data)
21
22      def connectionMade(self):
23          self.receive_ack()
24
25
26  def chatLoop(chat):
27      while True:
28          chat.send_message(str(raw_input()))
29
30  def printer(message):
31      print message
32
33  # Inicializa classe
34  factory = Factory()
35  factory.protocol = NeuralCryptographyClient
36  endpoint = TCP4ClientEndpoint(reactor, "localhost",1984)
37  endpoint.connect(factory)
38  reactor.run()
```

Código C.3: Classe NeuralCryptographyServer implementa o servidor na sincronização em rede

```
1  from twisted.internet.protocol import Protocol
2  from twisted.internet.protocol import Factory
3  from twisted.internet.endpoints import TCP4ServerEndpoint
4  from twisted.internet import reactor
5  from protocol import NeuralCryptography
6  from tpm import TreeParityMachine
7  import json
8
9  class NeuralCryptographyServer(NeuralCryptography):
10
11      def __init__(self):
12          NeuralCryptography.__init__(self)
```

```
13          self.call(target=chatLoop, args=(self,))
14          self.receive(target=printer)
15
16      def dataReceived(self, data):
17          if not self.synced():
18              self.syncronizer(data)
19          else:
20              self.received(data)
21
22  def chatLoop(chat):
23      while True:
24          chat.send_message(str(raw_input("Digite sua mensagem: ")))
25
26  def printer(message):
27      print "Mensagem recebida: ", message
28
29  # Inicializa classe
30  factory = Factory()
31  factory.protocol = NeuralCryptographyServer
32  endpoint = TCP4ServerEndpoint(reactor, 1984)
33  endpoint.listen(factory)
34  reactor.run()
```