

UNIVERSIDADE FEDERAL DE GOIÁS
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA
E DE COMPUTAÇÃO

**CORINDA: HEURÍSTICAS CONCORRENTES PARA
QUEBRA DE SENHAS**

Bernardo Araujo Rodrigues

[UFG] & [EMC]
[Goiânia - Goiás - Brasil]
28 de setembro de 2018

**TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR
VERSÕES ELETRÔNICAS DE TESES E DISSERTAÇÕES
NA BIBLIOTECA DIGITAL DA UFG**

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou download, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: ☒ Dissertação ☐ Tese

2. Identificação da Tese ou Dissertação:

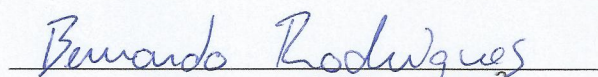
Nome completo do autor: Bernardo Araujo Rodrigues

Título do trabalho: Corinda: Heurísticas Concorrentes para Quebra de Senhas

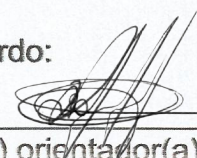
3. Informações de acesso ao documento:

Concorda com a liberação total do documento ☒ SIM ☐ NÃO¹

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.


Assinatura do(a) autor(a)²

Ciente e de acordo:


Assinatura do(a) orientador(a)²

Data: 28 / 09 / 18

¹ Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

Casos de embargo:

- Solicitação de registro de patente;
- Submissão de artigo em revista científica;
- Publicação como capítulo de livro;
- Publicação da dissertação/tese em livro.

² A assinatura deve ser escaneada.

UNIVERSIDADE FEDERAL DE GOIÁS
PROGRAMA DE PÓS-GRADUAÇÃO EM
ENGENHARIA ELÉTRICA
E DE COMPUTAÇÃO

**CORINDA: HEURÍSTICAS CONCORRENTES PARA
QUEBRA DE SENHAS**

Bernardo Araujo Rodrigues

Dissertação apresentada a Banca Examinadora como exigência parcial para a obtenção do título de Mestre em Engenharia Elétrica e de Computação pela Universidade Federal de Goiás (UFG), Escola de Engenharia Elétrica, Mecânica e de Computação (EMC), sob a orientação do Prof. Dr. Wesley Pacheco Calixto.

[UFG] & [EMC]
[Goiânia - Goiás - Brasil]
28 de setembro de 2018

Dados Internacionais de Catalogação na Publicação (CIP)
Sistemas da Bibliotecas da UFG, GO - Brasil

C331s Rodrigues, Bernardo Araujo, 31/07/90.

Corinda: Heurísticas Concorrentes para Quebra de Senhas [manuscrito]/ Bernardo Araujo Rodrigues. – [Goiânia - Goiás - Brasil]: [UFG] & [EMC], 28 de setembro de 2018.
130 f. : il.

Orientador: Wesley Pacheco Calixto - UFG

Dissertação - Universidade Federal de Goiás - UFG,
Escola de Engenharia Elétrica, Mecânica e de Computação
- EMC

Inclui bibliografia.

1.Senhas - Teses. 2.*Hashes* - Teses. 3.Segurança da Informação - Teses. I. Calixto, Wesley Pacheco; II. Universidade Federal de Goiás. Programa de Pós-Graduação em Engenharia Elétrica e de Computação. III. Corinda: Quebrando *Hashes* de Senhas Concorrentemente com a Linguagem de Programação Go

CDU 000.0.000:000.0

Copyright © 28 de setembro de 2018 by Federal University of Goiás - UFG, Brazil. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, microfilming, recording or otherwise, without written permission from the Library of UFG, with the exception of any material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use of the reader of the work.

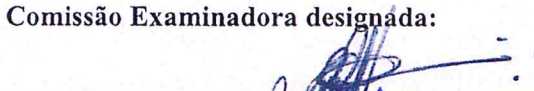


Ata de Defesa de Dissertação de Mestrado

Ata da sessão de julgamento da Dissertação de Mestrado em Engenharia Elétrica e de Computação, área de concentração Engenharia de Computação, do candidato **Bernardo Araújo Rodrigues**, realizada em 27 de agosto de 2018.

Aos vinte e sete dias do mês de agosto de dois mil e dezoito, às 14:00 horas, na sala Caryocar brasiliensis, bloco "A" da Escola de Engenharia Elétrica, Mecânica e de Computação (EMC), Universidade Federal de Goiás (UFG), reuniram-se os seguintes membros da Comissão Examinadora designada pela Coordenadoria do Programa de Pós-graduação em Engenharia Elétrica e de Computação: os Doutores Wesley Pacheco Calixto – Orientador (EMC/UFG), Marcos Antônio de Sousa – (ENG/PUCGoiás), Ricardo Soares Bôaventura – (FEELT/UFU), Regina Célia Bueno da Fonseca – (MAT/IFG), Eduardo Noronha de Andrade Freitas – (INF/IFG) e Rodrigo Pinto Lemos – (EMC/UFG), para julgar a Dissertação de Mestrado de **Bernardo Araújo Rodrigues**, intitulada "**Corinda: heurísticas concorrentes para quebra de senhas**", apresentada pelo candidato como parte dos requisitos necessários à obtenção do grau de Mestre, em conformidade com a regulamentação em vigor. O Professor Doutor Wesley Pacheco Calixto, Presidente da Comissão, abriu a sessão e apresentou o candidato que discorreu sobre seu trabalho, após o que, foi arguido pelos membros da Comissão na seguinte ordem: Marcos Antônio de Sousa, Ricardo Soares Bôaventura, Regina Célia Bueno da Fonseca, Eduardo Noronha de Andrade Freitas e Rodrigo Pinto Lemos. A parte pública da sessão foi então encerrada e a Comissão Examinadora reuniu-se em sessão reservada para deliberar. A Comissão julgou então que o candidato, tendo demonstrado conhecimento suficiente, capacidade de sistematização e argumentação sobre o tema de sua Dissertação, foi considerado **aprovado** e deve satisfazer as exigências listadas na Folha de Modificação, em anexo a esta Ata, no prazo máximo de 30 dias, ficando o professor orientador responsável por atestar o cumprimento destas exigências. Os membros da Comissão Examinadora descreveram as justificativas para tal avaliação em suas respectivas Folhas de Avaliação, anexas a esta Ata. Nada mais havendo a tratar, o presidente da Comissão declarou encerrada a sessão. Nos termos do Regulamento Geral dos Cursos de Pós-graduação desta Universidade, a presente Ata foi lavrada, lida e julgada conforme segue assinada pelos membros da Comissão supracitados e pelo candidato. Goiânia, 27 de agosto de 2018.

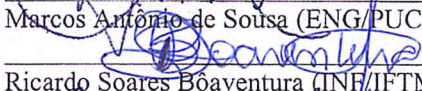
Comissão Examinadora designada:


Wesley Pacheco Calixto – Orientador (EMC/UFG)

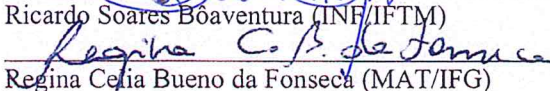
(Avaliação: Aprovado)


Marcos Antônio de Sousa (ENG/PUCGoiás)

(Avaliação: Aprovado)


Ricardo Soares Bôaventura (INF/IFTM)

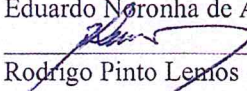
(Avaliação: Aprovado)


Regina Célia Bueno da Fonseca (MAT/IFG)

(Avaliação: Aprovado)


Eduardo Noronha de Andrade Freitas (INF/IFG)

(Avaliação: H)


Rodrigo Pinto Lemos (EMC/UFG)

(Avaliação: APROVADO)

Candidato:


Bernardo Araújo Rodrigues

“Argumentar que não te importas com o direito à privacidade por não ter nada a esconder é o mesmo que dizer que não te importas com a liberdade de expressão por não ter nada a dizer.”

EDWARD SNOWDEN

*Dedico este trabalho a meus pais, pelo amor e dedicação
incondicional.*

AGRADECIMENTOS

Agradeço ao orientador e amigo Wesley Pacheco Calixto, por sua dedicação e compreensão, mesmo nos momentos mais difíceis. Por ter acreditado em mim desde o início, e ter permitido que eu seguisse o caminho onde pude florescer.

Aos colegas do NExT (Núcleo de Estudos Experimentais e Tecnológicos), pelo companheirismo, dedicação e amizade.

A minha família, pelo amor e incentivo incondicional, seja em momentos de dificuldade ou de alegria.

Aos colegas de trabalho e amigos da Data Traffic, pela compreensão, incentivo, e risadas.

A CAPES pelo aporte para o desenvolvimento deste trabalho.

RESUMO

Propõe-se neste trabalho o desenvolvimento de software para quebra de senhas baseado em algoritmos heurísticos concorrentes, teoria dos modelos de primeira ordem e inferência estatística. A metodologia proposta é desenvolvida na linguagem de programação Go, que possui foco em processos sequenciais comunicantes. O software utiliza algoritmos heurísticos concorrentes criados a partir de padrões estatísticos identificados em conjuntos de amostras para realizar a quebra de senhas. São detectadas distintas distribuições estatísticas em conjuntos amostrais de senhas. O software é capaz de quebrar porções significativas de três diferentes conjuntos de senhas em curto período de tempo. Conclui-se que heurísticas concorrentes são alternativa viável para realizar a quebra de senhas em Unidades Centrais de Processamento, podendo ser utilizada para conscientizar usuários sobre a importância de senhas de alta entropia e privacidade digital.

CORINDA: CONCURRENT HEURISTICS FOR PASSWORD CRACKING

ABSTRACT

This work proposes the development of a software with the purpose of cracking passwords based on concurrent heuristic algorithms, first order model theory, and statistical inference. The proposed methodology is developed on the Go programming language, with focus on communicating sequential processes. The software uses concurrent heuristic algorithms based on statistical patterns derived from sample sets to perform password cracking. Distinct statistical distributions are detected on sample sets. The software is able to crack significant portions of three different password sets in a short period of time. It is concluded that concurrent heuristic algorithms are a viable alternative to perform Central Processing Unit password cracking and can be used to raise awareness amongst users about the importance of high entropy passwords and digital privacy.

SUMÁRIO

Pág.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

LISTA DE ABREVIATURAS E SIGLAS

CAPÍTULO 1 INTRODUÇÃO 25

CAPÍTULO 2 QUEBRA DE SENHAS E LINGUAGEM GO . . . 29

2.1 Autenticação 29

2.2 Quebra das Senhas 29

2.3 Heurística 30

2.4 *Passfault* 31

2.5 Casos de Vazamentos 31

2.5.1 *RockYou* 32

2.5.2 *LinkedIn* 32

2.5.3 *AntiPublic* 32

2.6 Linguagem de Programação Go 33

2.6.1 Processos Sequenciais Comunicantes 33

2.6.2 Gorrotinas 33

2.6.3 Canais 34

2.6.4 *Array* 35

2.6.5 Mapa 35

2.6.6 Estrutura, Tipo e Classe 36

2.7 Considerações 37

CAPÍTULO 3 MODELAGEM DE SENHAS 39

3.1 String 39

3.1.1 Conjunto Amostral 40

3.1.2 Operações com Strings 40

3.2 Modelo 41

3.3 Entropia 42

3.4	Função de Dispersão Criptográfica	42
3.5	Processo de Quebra	43
3.6	Considerações	44
CAPÍTULO 4 METODOLOGIA		45
4.1	Modelagem do Corinda	45
4.1.1	Modelo Elementar e Modelo Composto	45
4.1.2	<i>Token</i>	47
4.1.3	Modelo Crítico	47
4.1.4	Frequência Relativa	48
4.1.5	Entropia de Modelo	48
4.2	Corinda	48
4.3	Pacote Modelo Elementar	50
4.3.1	Método elementary.Model.UpdateEntropy()	50
4.3.2	Método elementary.Model.UpdateTokenFreq()	50
4.3.3	Método elementary.Model.SortedTokens()	51
4.4	Pacote Modelo Composto	51
4.4.1	Método composite.Model.UpdateProb()	51
4.4.2	Método composite.Model.UpdateFreq()	51
4.4.3	Método composite.Model.UpdateEntropy()	52
4.4.4	Método composite.Model.recursive()	52
4.4.5	Método composite.Model.Guess()	52
4.5	Pacote de Treinamento	52
4.5.1	Gorrotina train.Train.generator()	53
4.5.2	Gorrotina train.Train.batchAnalyzer()	53
4.5.3	Gorrotina train.Train.mapsMerger()	53
4.5.4	Gorrotina train.Train.mapsSaver()	54
4.5.5	Fluxograma do Pacote de Treinamento	54
4.6	Pacote de Quebra de Senhas	55
4.6.1	Gorrotina de geração de palpites	56
4.6.2	Gorrotina crack.Crack.guessLoop()	57
4.6.3	Gorrotina composite.Model.digest()	57
4.6.4	Gorrotina crack.Crack.searcher()	58
4.6.5	Gorrotina crack.Crack.saver()	58
4.6.6	Gorrotina crack.Crack.monitor()	58
4.6.7	Gorrotina crack.Crack.reporter()	58
4.6.8	Fluxograma do Pacote de Quebra de Senhas	58
4.7	Pacote da Interface de Comandos	59

4.7.1	Comando train	59
4.7.2	Comando crack	59
4.8	Experimentos e Validação	61
4.8.1	Validação do Corinda	61
4.9	Considerações	63
CAPÍTULO 5 RESULTADOS		65
5.1	Experimentos Preliminares	65
5.2	Código Fonte	66
5.3	Validação do Corinda	70
5.4	Processos de treinamento	71
5.4.1	<i>RockYou</i>	71
5.4.2	<i>LinkedIn</i>	72
5.4.3	<i>AntiPublic</i>	73
5.5	Quebra de Senhas	74
5.6	Limitações do Corinda	77
5.7	Comentários	78
CAPÍTULO 6 CONCLUSÃO		79
6.1	Contribuições do Trabalho	80
6.2	Sugestões para Trabalhos Futuros	81
APÊNDICE A Pacote Modelo Elementar		83
APÊNDICE B Pacote Modelo Composto		87
APÊNDICE C Pacote de Treinamento		91
APÊNDICE D Pacote de Quebra de Senhas		103
APÊNDICE E Pacote de Interface de Comandos		111
APÊNDICE F Exemplo de Produto Cartesiano		115
APÊNDICE G Exemplo de Parametrizacao de Carga Computacional		119
REFERÊNCIAS BIBLIOGRÁFICAS		125
GLOSSÁRIO		129

LISTA DE FIGURAS

	<u>Pág.</u>
2.1 Impressão no <i>console</i> utilizando o Algoritmo 2.1.	35
2.2 Impressão no <i>console</i> utilizando o Algoritmo 2.2.	35
2.3 Impressão no <i>console</i> utilizando o Algoritmo 2.3.	36
2.4 Impressão no <i>console</i> utilizando o Algoritmo 2.4.	37
3.1 Entropia $H(x_i)$ de base 10 em função da probabilidade $P(x_i)$	43
4.1 Fluxograma do Processo de Treinamento.	55
4.2 Legendas do Fluxograma do Processo de Treinamento.	56
4.3 Fluxograma do Processo de Quebra de Senhas.	60
4.4 Legendas do Fluxograma do Processo de Quebra de Senhas.	61
5.1 Experimentos preliminares com o conjunto <i>RockYou</i> : (a) ranking de frequências dos modelos compostos no conjunto, (b) histograma dos ta- manhos das senhas.	67
5.2 Exemplo de modelo elementar retirado de espaço amostral.	67
5.3 Exemplo de modelo composto retirado de espaço amostral.	69
5.4 Histogramas: (a) frequências relativas nos modelos compostos da lista <i>RockYou</i> , (b) entropias dos modelos elementares na lista <i>RockYou</i> , (c) entropias dos modelos compostos na lista <i>RockYou</i>	72
5.5 Histogramas: (a) frequências relativas nos modelos compostos da lista <i>LinkedIn</i> , (b) entropias dos modelos elementares na lista <i>LinkedIn</i> , (c) entropias dos modelos compostos na lista <i>LinkedIn</i>	73
5.6 Histogramas: (a) frequências relativas nos modelos compostos da lista <i>AntiPublic</i> , (b) entropias dos modelos elementares na lista <i>AntiPublic</i> , (c) entropias dos modelos compostos na lista <i>AntiPublic</i>	74
5.7 Saída do método <code>composite.Model.recursive()</code>	78

LISTA DE TABELAS

	<u>Pág.</u>
2.1 Dez senhas mais frequentes da lista <i>LinkedIn</i>	30
4.1 Modelos elementares reconhecidos pelo <i>Passfault</i>	62
5.1 Vinte modelos compostos frequentemente utilizados no conjunto <i>RockYou</i>	66
5.2 Conjuntos amostrais.	71
5.3 Dez modelos compostos críticos frequentes na lista <i>RockYou</i>	71
5.4 Dez modelos compostos críticos frequentes na lista <i>LinkedIn</i>	72
5.5 Dez modelos compostos críticos frequentes na lista <i>AntiPublic</i>	73
5.6 Média e desvio padrão das entropias dos modelos compostos.	74
5.7 Experimentos de quebra de <i>hashes</i> SHA1.	75
5.8 Experimentos de quebra de <i>hashes</i> SHA1 com conjuntos amostrais trun- cados.	75
5.9 Experimentos de quebra de <i>hashes</i> SHA256.	75
5.10 Experimentos de quebra de <i>hashes</i> SHA256 com conjuntos amostrais truncados.	76

LISTA DE SÍMBOLOS

c	Caractere
A	Alfabeto
s	<i>String</i>
\tilde{s}	<i>String</i> composta
$\Psi(\tilde{s})$	Partição da <i>string</i> composta
$\Psi^{-1}(s_i)$	Concatenação de <i>strings</i>
Γ	Multiconjunto de <i>strings</i>
$\alpha(s)$	Predicado lógico
$\phi(s)$	Fórmula lógica
m	Modelo
λ	Domínio de m
$\mathcal{C}(m)$	cardinalidade de m
\tilde{m}	Modelo composto
$\tilde{\lambda}$	Domínio de \tilde{m}
$\hat{m}(\tilde{s})$	Modelo crítico de \tilde{s} no conjunto de amostras Γ
$\hat{\lambda}$	Domínio de \hat{m}
\mathcal{M}_Γ	Multiconjunto de modelos críticos das strings do conjunto de amostras Γ
$\theta(\hat{m})$	Frequência relativa de \hat{m} em \mathcal{M}_Γ
$H(\hat{m}(\tilde{s}))$	Entropia do modelo crítico \hat{m} de \tilde{s} no conjunto de amostras Γ
$F(\tilde{s})$	<i>Hash</i> calculado com a função de dispersão criptográfica (FDC) aplicada à <i>string</i> \tilde{s}
$F(\Gamma)$	Conjunto de <i>hashes</i> calculados com a FDC aplicada às <i>strings</i> do conjunto de amostras Γ
N_e	Número de entradas no arquivo de conjunto amostral
S_f	Somatório total das frequências de cada senha no conjunto amostral
C_s	Tamanho médio (de caracteres) das <i>strings</i> do conjunto
h	Número de <i>hashes</i> quebrados no experimento
$\%h$	Porcentagem de <i>hashes</i> quebrados no experimento
S	Número de senhas quebradas no experimento
$\%S$	Porcentagem de senhas quebradas no experimento

LISTA DE ABREVIATURAS E SIGLAS

FDC	–	Função de Dispersão Criptográfica
CPU	–	<i>Central Processing Unit</i>
GPU	–	<i>Graphics Processing Unit</i>
DSP	–	<i>Digital Signal Processor</i>
FPGA	–	<i>Field Programmable Gate Array</i>
ASIC	–	<i>Application Specific Integrated Circuit</i>
NIST	–	<i>National Institute of Standards and Technology</i>
NSA	–	<i>National Security Agency</i>
OWASP	–	<i>Open Web Application Security Project</i>
SHA1	–	<i>Secure Hashing Algorithm 1</i>
SHA256	–	<i>Secure Hashing Algorithm 2, 256 bits</i>
PSC	–	Processos Sequenciais Comunicantes
POO	–	Programação Orientada a Objetos
CSV	–	<i>Comma Separated Value</i>
JSON	–	<i>JavaScript Object Notation</i>
ASCII	–	<i>American Standard Code for Information Interchange</i>
GB	–	<i>Giga Bytes</i>
RAM	–	<i>Random Access Memory</i>
LTS	–	<i>Long Term Support</i>

CAPÍTULO 1

INTRODUÇÃO

Internet das Coisas, Computação na Nuvem e Redes Sociais possuem papel cada vez mais fundamental no desenvolvimento social e econômico da sociedade atual. Isso faz com que o tema da Segurança da Informação seja cada vez mais importante (HUTCHENS, 2014). Com isto, qualquer vulnerabilidade inerente a tecnologias amplamente utilizadas torna-se alvo de interesse ao público em geral.

Senhas são objetos importantes em sistemas computacionais, sendo essenciais para garantir a segurança de informações pessoais ou sigilosas, bem como acesso a variedade de dispositivos inteligentes (CISAR; CISAR, 2007). Enquanto diversos tipos alternativos de autenticação têm sido implementados, senhas baseadas em sequências de caracteres ainda são o tipo de autenticação mais comum (SHEN; KHANNA, 1997; SINGH; YAMINI, 2013; JAYAMAHA et al., 2008; ZHANG; KOUSHANFAR, 2016).

Combinações frágeis de *username* e senhas fazem com que dispositivos e contas sejam facilmente comprometidas por agentes maliciosos (FLORENCIO; HERLEY, 2007; AMICO et al., 2010; BISHOP; KLEIN, 1995). Além de senhas fracas, a reutilização de senhas é prática comum entre usuários (GAW; FELTEN, 2006). Tais práticas abrem espaço para técnicas de engenharia social e escalada de privilégios, tais como as utilizadas no vazamentos de dados do serviço de armazenamento de arquivos *Dropbox* em 2016, onde agentes maliciosos se aproveitaram de senhas reutilizadas para obter acesso ao sistema da empresa (CONGER; LYNLEY, 2017).

Funções de dispersão criptográficas (FDC) geram longas e complexas sequências de caracteres (chamadas *hash*) a partir da informação de entrada, sendo matematicamente impossível recuperar a informação original a partir do *hash* calculado. FDC são funções unidirecionais unívocas, o que significa que a única forma de obter determinado *hash* é fornecendo a informação original como entrada da função (MENEZES et al., 2001; STALLINGS, 2017). Atualmente, sistemas computacionais armazenam apenas o *hash* da senha do usuário. A autenticação ocorre quando o *hash* calculado a partir da senha digitada pelo usuário é comparada com o *hash* armazenado no banco de dados, permitindo o acesso caso os valores sejam iguais (MENEZES et al., 2001).

No contexto de modelagem de ameaças, o indivíduo malicioso tentando obter a senha em questão é chamado de atacante. O administrador do sistema ou especialista

que tenta previnir que a senha seja comprometida é chamado de defensor (SWIDERSKI; SNYDER, 2009). De forma a obter a senha, o atacante deve encontrar a sequência de caracteres cuja FDC corresponde ao *hash* roubado do banco de dados. Tal processo de adivinhação é chamado de quebra da senha, e existem ferramentas específicas capazes de realizar milhões de palpites por segundo, tais como *John The Ripper* e *Hashcat* (MURAKAMI et al., 2010; BINNIE, 2016). *Hashcat* é a ferramenta mais avançada e popular atualmente, tornando possível o uso de técnicas de paralelização baseadas no uso de *Open Computing Language* (OpenCL), descrita em Munshi (2009). O *Hashcat* pode ser utilizado em diversas plataformas de *hardware*, tais como Unidades de Processamento Central (CPU), Unidades de Processamento Gráfico (GPU), Processadores Digitais de Sinais (DSP), e Arranjos de Portas Programáveis em Campo (FPGA).

Técnicas de quebra de senhas tem sido amplamente pesquisadas, não apenas pela comunidade científica, mas também por comunidades *online* e fóruns (WEIR et al., 2009; BISHOP; KLEIN, 1995; WANG et al., 2016). Em 2016, mais de 96% dos *hashes* vazados dos bancos de dados da rede social de negócios *LinkedIn*, descritos em Gosney (2016), foram quebrados menos de cinco meses depois de terem sido disponibilizados *online*. Se as senhas forem simples o bastante, mesmo FDC robustas não resistem a técnicas modernas de quebra de senha.

Quando os usuários criam suas senhas, a maioria tende a utilizar padrões que possam ser facilmente lembrados. Isto faz com que seja possível que o atacante consiga adivinhá-la. Alguns autores trabalharam com a tentativa de encontrar padrões na forma que os usuários criam suas senhas. Bishop e Klein (1995) procuram por padrões comuns tais como números de caracteres e palavras de dicionários. Outros pesquisadores também tentaram elaborar métricas confiáveis capazes de quantificar o quão robusta determinada senha é frente a tentativas de quebra. Castelluccia et al. (2016) utilizam Modelos de Markov para modelar a força de senhas.

Shannon (2009) estabelece o conceito de entropia como quantificador da incerteza de variáveis aleatórias. No contexto de quebra de senhas, a entropia é utilizada como indicador da robustez do modelo da senha quando submetida a processos de quebra (WHEELER, 2016). O padrão definido por Burr et al. (2004), Burr et al. (2013) nas diretrizes de autenticação *online* do *National Institute of Standards and Technology* (NIST) propõe a entropia de caractere como métrica de força. Kelley et al. (2011) avaliam a métrica do NIST como útil, porém limitada na maioria dos casos. Shay et al. (2010) propõem como métrica a entropia empírica baseada em

senhas previamente coletadas, também com resultados limitados.

De forma a estabelecer *framework* generalizado para a quantificação da resistência de determinada senha contra técnicas de quebra, Sahin et al. (2015) formaliza os conceitos de **complexidade** e **força** com sólidas definições matemáticas que enfatizam suas diferenças. Estes conceitos possuem importância fundamental no contexto de quebra de senhas, uma vez que eles derivam da compreensão fundamental de que:

...o sucesso do atacante ao quebrar uma senha deve ser definido pelos seus recursos computacionais disponíveis, tempo disponível, FDC utilizada, bem como a topologia que define o espaço de busca (SAHIN et al., 2015).

Rodrigues et al. (2017) avaliam os conceitos de complexidade e força estabelecidos por Sahin et al. (2015). Os autores realizam experimentos preliminares no trabalho intitulado *Passfault: an Open Source Tool for Measuring Password Complexity and Strength*. Neste trabalho é observada a existência de padrões estatísticos distintos na distribuição das senhas analisadas. A descoberta destes padrões de forma a anteceder a quebra da senha, pode ser útil na redução do tempo de quebra. A possibilidade de criação de algoritmos heurísticos que reflitam tais padrões estatísticos nos processos de quebra de senhas justifica este trabalho.

Este trabalho visa explorar a hipótese: se os conjuntos amostrais de senhas vazadas apresentam distintos padrões estatísticos e se a programação concorrente pode ser utilizada para otimizar o processo de quebra de senhas em CPU, então a entropia e a frequência relativa dos modelos de senhas podem ser utilizadas como parâmetros na otimização dos processos de quebra.

O objetivo principal deste trabalho é desenvolver software para quebra de senhas com processos sequenciais comunicantes em CPU, que recebe o nome de **Corinda**. Ainda como objetivos têm-se: i) investigar os padrões estatísticos dos modelos encontrados em conjuntos amostrais de senhas; ii) investigar os parâmetros entropia e frequência relativa dos modelos encontrados em conjuntos amostrais de senhas; iii) utilizar os parâmetros entropia e frequência relativa para criação de heurísticas concorrentes para quebra de senhas; iv) realizar experimentos para análise da performance do software de quebra de senhas em CPU.

Este trabalho é organizado em seis capítulos: o Capítulo 2 introduz o problema de quebra de senhas e a linguagem de programação Go. O Capítulo 3 apresenta a modelagem formal do problema de quebra de senhas. No Capítulo 4 é apresentada a

metodologia do trabalho e dos experimentos realizados e no Capítulo 5 são apresentados os resultados dos experimentos. Por fim, o Capítulo 6 apresenta as conclusões, e os Apêndices dispõem o código fonte do software proposto.

CAPÍTULO 2

QUEBRA DE SENHAS E LINGUAGEM GO

Este capítulo apresenta a autenticação e a quebra de senhas, bem como o fenômeno do vazamento de dados de autenticação. Apresenta algumas listas conhecidas com casos de vazamentos importantes e a ferramenta de avaliação de complexidade de senhas chamada *Passfault*. Descreve ainda a linguagem de programação **Go**.

2.1 Autenticação

Na maioria das organizações, as informações relativas à autenticação de usuários ficam armazenados em servidores. Tais servidores encontram-se instalados em complexas infraestruturas de rede, com diversos pontos que podem servir de entrada para agentes maliciosos, caso sejam mal configurados ou estejam desatualizados (HUTCHENS, 2014). Por razões que passam por ativismo político, crime organizado e até guerras cibernéticas entre nações, tais servidores são invadidos e informações de autenticação de usuários são furtadas.

Funções de dispersão criptográfica são funções unidirecionais que tomam como entrada *string* arbitrária e retornam como saída *string* de tamanho fixo, chamada *hash*. A inversão da FDC é considerada irrealizável, o que significa que não é possível encontrar a *string* original a partir do *hash* resultante (FERGUSON et al., 2010).

De forma a mitigar os danos no caso do vazamento dos dados de autenticação, a utilização de FDC para criptografar senhas armazenadas é prática comum (CISAR; CISAR, 2007). Apenas o *hash* da senha é armazenado, e a autenticação ocorre quando o *hash* calculado a partir da senha digitada pelo usuário é comparada com aquele armazenado no banco de dados, permitindo o acesso caso os valores sejam idênticos.

2.2 Quebra das Senhas

Denomina-se quebra da senha o processo de encontrar a *string* que quando utilizada como entrada da função de dispersão criptográfica, produz o *hash* alvo. Uma vez encontrada a *string*, a senha é considerada quebrada. Diferentes plataformas de hardware podem ser utilizadas para a quebra das senhas. Entre as arquiteturas utilizadas para paralelizar os processos de quebra de senha encontram-se Processadores de Vídeo (GPU), Arranjos de Portas Programáveis em Campo (FPGA), Processadores Digitais de Sinais (DSP), Circuitos Integrados de Aplicação Específica (ASIC), Aglomerados de Computadores (*Clusters*) e Unidades de Processamento Central

(CPU) (PICOLET, 2017).

Apesar de não possuir mecanismo de aceleração para o cálculo de *hashes* como as GPU, FPGA e ASIC, as CPU são de fácil acesso e uso, fazendo com que sejam escolha viável de *hardware* para a investigação do problema de quebra de senhas. As CPU apresentam-se como plataformas simples e neutras para explorar as dinâmicas de carga computacional de maneira controlada.

2.3 Heurística

A abordagem mais ingênua para se quebrar senhas é a força bruta. O processo de quebra varre todo o espaço de busca de maneira linear, e a complexidade do espaço de busca tende a superar a capacidade computacional na maioria dos casos (BINNIE, 2016). Apesar de ser solução completa, no sentido de que todo o espaço de busca é varrido, a força bruta possui tempo de execução subótimo. Algoritmos heurísticos sacrificam a completude da solução em troca da otimização do tempo de execução (PEARL, 1986). Apesar de não garantir sucesso em 100% das vezes que são executados, algoritmos heurísticos apresentam-se como alternativas no esforço de viabilizar a otimização do processo de quebra dos *hashes*.

As tendências dos usuários ao criarem senhas seguindo padrões frequentes permitem abordagem heurística para o processo de quebra de *hashes*. Por exemplo, a Tabela 2.1 dispõe as dez senhas que aparecem com maior frequência na lista furtada da rede social *LinkedIn*. Nota-se que sequência numérica é padrão recorrente. Palavras associadas ao contexto da senha (**linkedin** e **password**) também apresentam-se populares. Estes são exemplos dos padrões que limitam o espaço de busca para senhas prováveis, reduzindo sua ordem de grandeza de trilhões de senhas para algumas dezenas de milhares (RODRIGUES et al., 2017).

Tabela 2.1: Dez senhas mais frequentes da lista *LinkedIn*.

Senha	Frequência
123456	753.305
linkedin	172.523
password	144.458
123456789	94.314
12345678	63.769
111111	57.210
1234567	49.652
sunshine	39.118
qwerty	37.538
654321	33.854

2.4 *Passfault*

Criada em 2001, a *Open Web Application Security Project (OWASP)* é a organização com o objetivo de produzir artigos, metodologias, documentação, ferramentas e tecnologias no campo da segurança de aplicações *web*.

Mantido pela OWASP, o *Passfault* é a ferramenta de código aberto, implementada em *Java* e distribuída sob licença *Apache Licence 2.0*. Seu desenvolvimento começou em 2011, com o objetivo de fazer com que a complexidade e força das senhas sejam facilmente entendidas pela população. *Passfault* recebe como entrada uma senha, uma opção de FDC e uma opção de *hardware* adversário. Ele analisa a estrutura da senha, tentando encontrar o modelo que melhor descreve a mesma. Então, o *Passfault* informa o usuário da complexidade da senha (tamanho do espaço de busca), bem como o tempo necessário para quebrá-la, baseado na FDC e no *hardware* escolhido (RODRIGUES et al., 2017).

2.5 Casos de Vazamentos

Quando as listas de *usernames* e *hashes* são furtadas, várias vezes seus destinos são mercados negros *online*. Eventualmente, estas listas acabam emergindo para a superfície da internet. Comunidades inteiras se formam para compartilhar técnicas e resultados de quebra de *hashes*. Exemplo disto é a ferramenta *Hashcat*, projeto de código aberto resultado de esforço puramente comunitário. Quando ocorre o vazamento público dos dados de alguma corporação, curtos períodos de tempo se decorrem até que elevadas porcentagens dos *hashes* sejam quebradas.

Tabelas de consulta com *hashes* previamente computados são utilizados para reduzir o tempo de quebra dos mesmos. Caso considerável número de usuários utilizem a mesma senha em determinada lista, basta que a primeira senha seja quebrada para que todas iguais a ela sejam comprometidas. De forma a diminuir a eficácia das tabelas com os *hashes* pré computados, a técnica de *salting* é prática de segurança recomendada por especialistas. A técnica consiste em concatenar caracteres aleatórios (*salt*) à senha antes do cálculo do *hash* via FDC. O *salt* deve ser armazenado de forma segura. Durante a autenticação, a senha fornecida pelo usuário mais o *salt* armazenado são usadas no cálculo do *hash*. Contudo, caso os *salts* também sejam furtados junto aos *hashes*, tal técnica não impede que os *salts* sejam usados para computar a nova tabela de consulta. Existem algumas listas de *hashes* furtados que são conhecidas, como: i) *RockYou*, ii) *LinkedIn*, e iii) *AntiPublic*.

2.5.1 *RockYou*

A *RockYou* fornece serviços de jogos *on-line*. Em Dezembro de 2009, empresas de segurança da informação emitiram notificações sobre falhas de segurança (SQL Injection) nos servidores da *RockYou*, que alegou ter resolvido o problema. Este tipo de falha já havia sido amplamente documentada e estudada por cerca de uma década. Apesar das alegações da *RockYou* sobre a correção do problema, agentes maliciosos foram capazes de explorar tal falha e furtar cerca de 32 milhões de senhas na forma *plaintext*, ou seja, sem nenhuma proteção por FDC (CUBRILOVIC, 2009).

Este acontecimento é amplamente divulgado na mídia logo após o furto. Especialistas de segurança aconselharam aos usuários do *site* que trocassem todas suas senhas imediatamente, uma vez que as senhas furtadas poderiam ser utilizadas para acessar contas em outros *sites* (CUBRILOVIC, 2009). Desde então, esta lista de senhas tem sido amplamente utilizada para fins de pesquisa.

2.5.2 *LinkedIn*

O *LinkedIn* fornece serviços como rede social de contatos profissionais. Em Junho de 2012, os *hashes* das senhas de cerca de 6,5 milhões de usuários são furtados por criminosos russos. As senhas são protegidas pela FDC *Secure Hashing Algorithm 1* (SHA1), sem *salting*. Estes usuários não puderam mais acessar suas contas, e a empresa encorajou que todos seus usuários mudassem suas senhas. No mesmo dia, os *hashes* foram quebrados e postados em fóruns *online* (MURPHY, 2012).

Em Maio de 2016, mais 117 milhões de *hashes* aparecem na internet. Especula-se que esta lista foi furtada no mesmo incidente de 2012, contudo permanecendo em mercados negros *online* antes de ser descoberta. Como não havia proteção por *salting*, mais de 90% da lista de *hashes* é quebrada em menos de 72 horas. Como resposta, o *LinkedIn* invalidou as senhas de todos os usuários que não mudaram a senha a partir de 2012 (FRANCESCHI-BICCHIERAI, 2016).

2.5.3 *AntiPublic*

A lista conhecida como *AntiPublic* surge na internet em Dezembro de 2016 por meio de fóruns russos. A lista *AntiPublic* é amplamente comercializada em mercados negros. Quase nada se sabe sobre sua verdadeira origem, mas especula-se que ela seja o compilado geral de diversos vazamentos. Tais listas são comumente chamadas de listas *combo* (HUNT, 2017).

A lista contém cerca de 458 milhões de *e-mails* diferentes, alguns com múltiplas senhas. A lista também é utilizada em prática conhecida como *credential stuffing*, que consiste em automatizar a autenticação em diversos *websites* utilizando diferentes credenciais da lista (HUNT, 2017).

2.6 Linguagem de Programação Go

A linguagem de programação *Golang* ou linguagem de programação Go é criada pelo Google em 2009, tendo como autores: R. Griesmer, R. Pike e K. Thompson (DONOVAN; KERNIGHAN, 2015). A linguagem de programação Go é compilada, estaticamente tipada e baseada na linguagem C. O compilador, as ferramentas e o código fonte são distribuídos como software livre. Suporta diversas arquiteturas (*x86*, *ARM*), bem como diversos sistemas operacionais (*GNU/Linux*, *FreeBSD*, *MacOS*, *Windows*). A linguagem de programação Go possui proteção de memória, coleta de lixo, sistema de tipagem estrutural e implementa a programação concorrente no formato de processos sequenciais comunicantes (PSC), permitindo também a programação orientada à objetos (POO) (ORNBO, 2018).

2.6.1 Processos Sequenciais Comunicantes

No paradigma de computação por Processos Sequenciais Comunicantes (PSC), programas atuam como composições paralelas de processos. Na linguagem Go, processos do programa são chamados gorrotinas e não possuem estado compartilhado entre si. Toda comunicação e sincronização entre processos ocorre via canais (DONOVAN; KERNIGHAN, 2015).

2.6.2 Gorrotinas

Threads são a forma de processos computacionais dividirem a si mesmos em duas ou mais linhas de execução concorrente. O suporte à *threads* é fornecido a nível de sistema operacional e *kernel*. Em plataformas de hardware com arquitetura de CPU única, cada *thread* é processada de forma intercalada, porém aparentemente simultânea. Em arquiteturas de hardware com CPU *multi-core*, as *threads* podem ser executadas de maneira verdadeiramente simultânea (ROSEN, 2007).

Na linguagem Go, cada *thread* é chamada **gorrotina**. Gorrotinas são implementadas como funções ou métodos e podem ser vistas como *threads* minimalistas. Quando comparadas às *threads* de outras linguagens de programação, as gorrotinas possuem ambientes de execução simples. Isto permite que programas possam gerenciar milhares ou milhões de gorrotinas sendo executadas concorrentemente.

2.6.3 Canais

Canais são as conexões entre gorrotinas concorrentes. Cada canal age como sistema de comunicação que a gorrotina usa para enviar mensagens para outras gorrotinas. Cada canal conduz valores de determinado tipo, chamado de **tipo de elemento** do canal. O Algoritmo 2.1 apresenta duas gorrotinas concorrentes se comunicando via canal. A gorrotina `gerador()` é responsável por gerar dez números inteiros aleatórios de maneira sequencial, enquanto a gorrotina principal `main()` drena o canal ao imprimir os números no *console*. A gorrotina `main()` lança instância de `gerador()` antes de drenar o canal. A execução do Algoritmo 2.1 gera a saída ilustrada na Figura 2.1.

Algoritmo 2.1: Exemplo do uso de canal em linguagem Go.

```
package main

import (
    "math/rand"
    "fmt"
)

func gerador(c chan int) {
    // 10 iterações
    for i := 0; i < 10; i++ {
        // gera inteiro aleatório entre 0 e 100
        r := rand.Intn(100)

        // envia pro canal
        c <- r
    }

    // fecha canal
    close(c)
}

func main() {
    c := make(chan int) // inicializa canal
    go gerador(c) // lança gorrotina

    // drena canal
    for r := range c {
        fmt.Printf("%d, ", r) // imprime
    }
}
```

```
81, 87, 47, 59, 81, 18, 25, 40, 56, 0,
```

Figura 2.1: Impressão no *console* utilizando o Algoritmo 2.1.

2.6.4 *Array*

Arrays são sequências de tamanho fixo de zero ou mais elementos de determinado tipo. O Algoritmo 2.2 apresenta a construção do *array* utilizando a linguagem de programação Go, onde a execução do mesmo gera a saída ilustrada na Figura 2.2.

Algoritmo 2.2: Exemplo do uso de *array* em linguagem Go.

```
package main

import "fmt"

func main() {

    // declaração do array a
    var a []int = []int{100, 20, 3}

    // varre elementos de a
    for i, v := range a{
        fmt.Printf("(%d, %d), ", i, v)
    }
}
```

```
(0, 100), (1, 20), (2, 3),
```

Figura 2.2: Impressão no *console* utilizando o Algoritmo 2.2.

2.6.5 *Mapa*

Na linguagem Go ocorre o mapeamento de chaves a valores, onde as chaves são utilizadas como índices da tabela de conteúdo. O Algoritmo 2.3 apresenta a declaração e uso dos mapas. A execução do Algoritmo 2.3 gera a saída ilustrada na Figura 2.3.

Algoritmo 2.3: Exemplo do uso de mapa em linguagem Go.

```
package main

func main() {

    // declaração do mapa
    m := make(map[string]int)

    m["chave1"] = 7
    m["chave2"] = 13

    // acesso via chave
    fmt.Println(m["chave1"])
    fmt.Println(m["chave2"])
}
```

```
7
13
```

Figura 2.3: Impressão no *console* utilizando o Algoritmo 2.3.

2.6.6 Estrutura, Tipo e Classe

Na linguagem Go, estruturas representam tipos de dado agregado que agrupa zero ou mais valores de diversos tipos. Cada valor é chamado de campo. A linguagem permite que novos **tipos** sejam definidos como estruturas análogas às **classes** da programação orientada a objetos (POO). Os campos do tipo são referenciados pela sintaxe `pacote.Tipo.campo`. O Algoritmo 2.4 exemplifica a abstração do vetor como estrutura e tipo em Go. Os campos `vetor.Vetor.name`, `vetor.Vetor.X`, e `vetor.Vetor.Y` contém os dados necessários para representar vetores bidimensionais. A Figura 2.4 ilustra a saída oriunda da execução do Algoritmo 2.4.

Estruturas podem ter métodos (funções, gorrotinas) atribuídas a si, fazendo com que se comportem como classes. Métodos são referenciados pela nomenclatura `pacote.Tipo.metodo()`. Por exemplo, o tipo `vetor.Vetor` pode ter o método `vetor.Vetor.incX()`, responsável por incrementar o campo `vetor.Vetor.X` em uma unidade, como apresentado no Algoritmo 2.5.

Algoritmo 2.4: Exemplo de declaração de estrutura em linguagem Go.

```
package vetor

import "fmt"

type Vetor struct {
    name  string
    X      int
    Y      int
}

func main() {
    exemplo := Vetor{"ex", 1, 2}
    fmt.Println(exemplo.name)
    fmt.Println(exemplo.X)
    fmt.Println(exemplo.Y)
}
```

```
ex
1
2
```

Figura 2.4: Impressão no *console* utilizando o Algoritmo 2.4.

Algoritmo 2.5: Exemplo de declaração de método em linguagem Go.

```
func (vetor Vetor) incX(){
    vetor.X = vetor.X + 1
}
```

2.7 Considerações

Várias organizações não utilizam as práticas de segurança como o *salting*. Ainda assim, heurísticas de quebra de *hashes* continuam em constante evolução e os vazamentos de credenciais continuam acontecendo com frequência cada vez maior. Desta forma, é importante a conscientização dos usuários para que utilizem senhas fortes de forma a dificultar a quebra do *hash*. A linguagem de programação Go possibilita a utilização de processos sequenciais comunicantes e programação orientada a objetos para modelar e implementar processos da quebra. O capítulo a seguir apresenta a modelagem teórica do problema de quebra de senhas.

CAPÍTULO 3

MODELAGEM DE SENHAS

Este capítulo formaliza as principais definições pertinentes à modelagem do problema de quebra de senhas. O embasamento teórico dos conceitos como *string*, conjunto amostral, modelo, frequência relativa, função de dispersão criptográfica, processo de quebra e entropia do modelo são apresentados, baseando-se em teoria dos conjuntos, teoria dos modelos de primeira ordem e inferência estatística.

3.1 String

Seja c o símbolo pertencente ao conjunto universo de símbolos A . Chama-se c de **caractere**. Seja A o conjunto universo de todos os caracteres possíveis. Por exemplo, A pode ser o conjunto de caracteres ASCII. Chama-se A de **alfabeto**.

$$A = \{c_{i=1}, \dots, c_N\} \quad (3.1)$$

Seja s o multiconjunto bem ordenado de caracteres. Chama-se s de **string**.

$$s = \{c_{j=1}, \dots, c_M | c_j \in A\} \quad (3.2)$$

Seja \tilde{s} o multiconjunto bem ordenado de *strings*. Chama-se \tilde{s} de **string composta**.

$$\tilde{s} = \{s_{k=1}, \dots, s_L\} \quad (3.3)$$

Seja $\mathbb{X}(A)$ o conjunto de todas *strings* possíveis sobre A .

$$\mathbb{X}(A) = \wp(A^n), n \rightarrow \infty \quad (3.4)$$

Aqui, $\wp(x)$ representa o conjunto potência de x , e A^n representa o n -ésimo produto cartesiano de A (HALMOS, 2017).

3.1.1 Conjunto Amostral

Seja Γ o multiconjunto de *strings* resultado do processo de amostragem. Chama-se Γ de multiconjunto de amostras, ou **conjunto amostral** (CASELLA; BERGER, 2010).

3.1.2 Operações com Strings

A partir das operações de partição e concatenação, é possível transformar *strings* compostas em *strings* elementares e vice-versa. Define-se a operação de **partição** de *string* composta $\Psi(\tilde{s})$, como a partição do multiconjunto \tilde{s} em subconjuntos s_i , dada por:

$$\Psi(\tilde{s}) = s_{i=1} | \cdots | s_N \quad (3.5)$$

Por exemplo, seja $\tilde{s} = \text{"psword1"}$. Possíveis partições de \tilde{s} são:

$$\Psi_1(\tilde{s}) = \{\{\text{"p"}\}, \{\text{"s"}\}, \{\text{"word"}\}, \{\text{"1"}\}\} \quad (3.6)$$

$$\Psi_2(\tilde{s}) = \{\{\text{"ps"}\}, \{\text{"word"}\}, \{\text{"1"}\}\} \quad (3.7)$$

$$\Psi_3(\tilde{s}) = \{\{\text{"psword"}\}, \{\text{"1"}\}\} \quad (3.8)$$

Define-se a operação de **concatenação** de *strings* $\Psi^{-1}(s_{i=1} | \cdots | s_N)$ como a operação inversa da partição, tal que:

$$\Psi^{-1}(s_{i=1} | \cdots | s_N) = \tilde{s} \quad (3.9)$$

As concatenações dos subconjuntos em (3.6), (3.7), (3.8) são dados por:

$$\Psi_1^{-1}(\{\{\text{"p"}\}, \{\text{"s"}\}, \{\text{"word"}\}, \{\text{"1"}\}\}) = \text{"psword1"} \quad (3.10)$$

$$\Psi_2^{-1}(\{\{\text{"ps"}\}, \{\text{"word"}\}, \{\text{"1"}\}\}) = \text{"psword1"} \quad (3.11)$$

$$\Psi_3^{-1}(\{\{\text{"psword"}\}, \{\text{"1"}\}\}) = \text{"psword1"} \quad (3.12)$$

Como qualquer *string* com mais de um caractere pode ser considerada *string* composta, utiliza-se os termos *string* e *string* composta indistintamente.

3.2 Modelo

Seja $\alpha(s)$ o **predicado lógico**, isto é, a função booleana que define a relação entre o termo s e o conjunto $\lambda \subseteq \mathbb{X}(A)$. Por exemplo, se λ é dicionário de palavras da língua inglesa, tem-se o seguinte predicado:

$$\alpha(s) \implies s \in \lambda \quad (3.13)$$

Seja $\phi(s)$ a **fórmula lógica** formada por um ou mais predicados α_i unidos por conectores lógicos, por exemplo:

$$\phi(s) : (\alpha_{i=1}(s) \wedge \cdots \wedge \alpha_N(s)) \quad (3.14)$$

Define-se K como o conjunto formado por constantes (no caso, *strings*) $s_i \in \lambda$ e predicados α_j . Seguindo a definição de verdade na teoria dos modelos segundo Tarski em [Patterson \(2008\)](#), define-se o modelo m como a estrutura de assinatura K , onde todos elementos s_i de λ satisfazem $\phi(s)$, formada pelos predicados α_j . Desta forma, escreve-se:

$$m \models \phi(s) \quad (3.15)$$

para expressar que $\phi(s)$ é verdade em m , ou seja, que m é **modelo** de $\phi(s)$ ([CHANG; KEISLER, 2012](#)). Por exemplo, seja λ_a o dicionário de palavras inglesas, e λ_b composto por todas possíveis *strings* iniciadas com caractere em caixa-alta, então:

$$\alpha_a(s) \implies s \in \lambda_a \quad (3.16)$$

$$\alpha_b(s) \implies s \in \lambda_b \quad (3.17)$$

$$\phi(s) : (\alpha_a(s) \wedge \alpha_b(s)) \implies s \in \lambda = \lambda_a \cap \lambda_b \quad (3.18)$$

$$m \models \phi(s) \Rightarrow \text{"Example"} \in \lambda \quad (3.19)$$

No contexto da teoria dos modelos de primeira ordem, o conjunto de constantes (no caso, *strings*) λ da assinatura K de m é chamado de **domínio** de m . O número de elementos de λ é chamado de **cardinalidade** de m (CHANG; KEISLER, 2012). Denota-se a cardinalidade de m_i por \mathcal{C}_i e $\mathcal{C}(m_i)$. Observa-se que a cardinalidade é equivalente ao conceito de complexidade estabelecido por Sahin et al. (2015).

Quando seres humanos criam suas senhas, eles tendem a utilizar de padrões que possam ser lembrados posteriormente. A teoria dos modelos de primeira ordem permite a formalização de tais padrões, restringindo o espaço de busca das senhas mais prováveis.

3.3 Entropia

A Teoria da Informação estuda a transmissão e codificação de mensagens em conjuntos de símbolos (MACKAY, 2004). De acordo com Shannon (2009), a entropia de determinada variável aleatória discreta X definida sobre o conjunto de símbolos $\chi = \{x_{i=1}, \dots, x_N\}$ é dada por:

$$H(x_i) = -P(X = x_i) \log_{10} P(X = x_i) \quad (3.20)$$

A Figura 3.1 ilustra a relação entre a entropia da variável aleatória X com a probabilidade desta variável assumir determinado valor x_i . Observa-se na Figura 3.1 que a entropia pode ser utilizada como indicador de incerteza da variável aleatória X . Observa-se ainda que os símbolos com probabilidades de ocorrência altas ($P(x_i) \approx 1$), ou baixas ($P(x_i) \approx 0$) praticamente não alteram a entropia do sistema ($H(x_i) \approx 0$). Por outro lado, símbolos cuja ocorrência é difícil de prever tendem a adicionar entropia ao sistema ($H(x_i) > 0$).

3.4 Função de Dispersão Criptográfica

A função de dispersão criptográfica (FDC) é a função cuja inversão é considerada praticamente irrealizável, ou seja, é quase impossível recriar os valores de entrada,

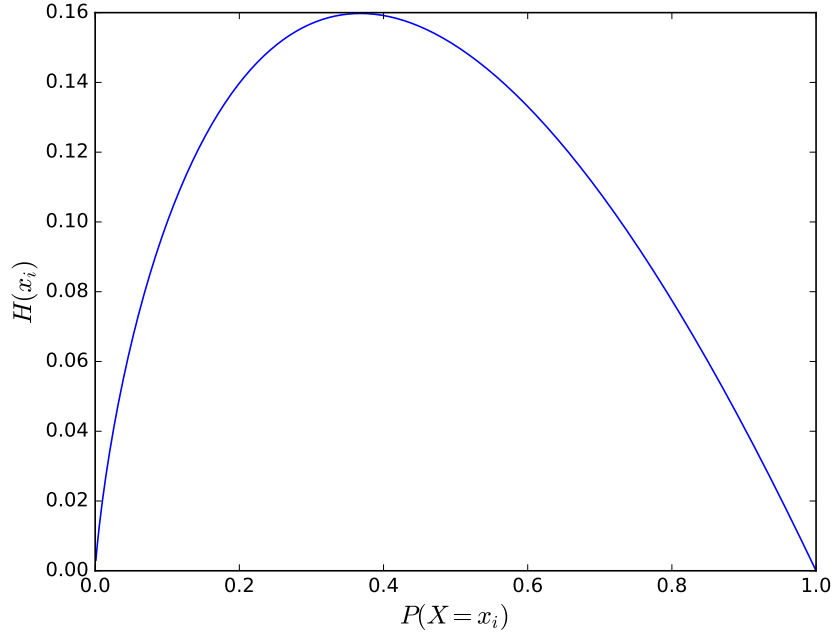


Figura 3.1: Entropia $H(x_i)$ de base 10 em função da probabilidade $P(x_i)$.

utilizando somente o valor resultante da dispersão. Define-se a FDC $F(\tilde{s})$ como função unidirecional que toma como entrada *string* de tamanho arbitrário \tilde{s} e retorna como saída *string* de tamanho fixo h , chamada **hash** (FERGUSON et al., 2010; MENEZES et al., 2001). A FDC possui as seguintes características: i) dado \tilde{s} , é trivial computar $h = F(\tilde{s})$; ii) dado h , é computacionalmente difícil encontrar \tilde{s} tal que $h = F(\tilde{s})$ e iii) dado \tilde{s} , é computacionalmente difícil encontrar \tilde{s}' tal que $F(\tilde{s}) = F(\tilde{s}')$. Por computacionalmente difícil entende-se que o processo levaria tempo demasiadamente longo para sua conclusão, na ordem de anos, décadas ou séculos.

3.5 Processo de Quebra

O **processo de quebra** do *hash* h , consiste da tarefa de encontrar *string* \tilde{s} tal que $F(\tilde{s}) = h$. Para isto, o Processo consiste em calcular a FDC $F(\tilde{s}_i)$ de cada *string* \tilde{s}_i pertencente ao domínio $\hat{\lambda}$ do modelo crítico \hat{m} , até que seja encontrada $F(\tilde{s}_i) = h$.

Dependendo da FDC escolhida e da capacidade computacional utilizada no processo de quebra, caso o domínio $\hat{\lambda}$ seja grande o suficiente, o processo pode estender-se por período de tempo extremamente longo, fazendo com que o processo torne-se inviável.

3.6 Considerações

A definição de *string* estabelece representação formal de senhas. As definições de modelo e seus derivados formalizam os conceitos que permitem a otimização do processo de quebra de senhas. A definição de FDC estabelece a noção de *hash*. Por fim, a formalização do conceito de entropia permite quantificar quão robusta é determinada senha quando submetida a processos de quebra. Com base nestas definições, o próximo capítulo estabelece a metodologia utilizada na implementação do **Corinda**.

CAPÍTULO 4

METODOLOGIA

Neste capítulo é apresentada a metodologia para o desenvolvimento do software denominado **Corinda**, que é um quebrador de senhas. As estratégias de geração de palpites de senhas são baseadas nos conceitos de modelos, concorrência e entropia. Os *hashes* alvo são previamente conhecidos, o que faz do **Corinda** o quebrador *offline*, implementado na linguagem de programação **Go**. Ainda é realizado neste capítulo a descrição dos experimentos para validar o desempenho do **Corinda**.

4.1 Modelagem do Corinda

Com o intuito de construir o software para quebras de senhas com processos sequenciais comunicantes em unidades de processamento central (CPU), desenvolve-se o **Corinda**. A metodologia implementada para o desenvolvimento do **Corinda** parte da investigação dos padrões estatísticos dos modelos encontrados em conjuntos amostrais de senhas, da investigação dos parâmetros entropia e frequência relativa dos modelos encontrados nestes conjuntos amostrais e na utilização da entropia e da frequência relativa para a criação de heurística concorrente para quebra das senhas. O escopo deste trabalho não inclui o conceito de *salting*, prática comum para dificultar a quebra de *hashes*. De forma a incorporar a operação de partição de *strings*, a metodologia proposta expande o conceito de modelo apresentado na seção 3.2, para a forma de modelo elementar e modelo composto.

4.1.1 Modelo Elementar e Modelo Composto

A definição de modelo elementar e de modelo composto consiste na adaptação estabelecida para os fins deste trabalho e não constam na literatura clássica da teoria dos modelos de primeira ordem (CHANG; KEISLER, 2012). No desenvolvimento do **Corinda**, utiliza-se o termo modelo elementar para representar as subdivisões atômicas nas partições da estrutura da *string*.

O modelo composto é definido como, seja Ξ o conjunto bem ordenado de modelos elementares, dado por:

$$\Xi = \{m_{i=1}, \dots, m_N\} \quad (4.1)$$

Define-se o **modelo composto** \tilde{m} como o modelo cujo domínio $\tilde{\lambda}$ é formado pelo

produto cartesiano dos domínios λ_i dos modelos simples $m_i \in \Xi$, dado por:

$$\tilde{\lambda} = \lambda_{i=1} \times \cdots \times \lambda_N \quad (4.2)$$

Por exemplo, seja $\Xi = \{m_a, m_b\}$, seja λ_a o dicionário de nomes de cidadãos brasileiros, e λ_b o conjunto de *strings* composta por números em formato de data, então:

$$\alpha_a(s) \implies s \in \lambda_a \quad (4.3)$$

$$\phi_a(s) : \alpha_a(s) \implies \text{"bernardo"} \in \lambda_a \quad (4.4)$$

$$\alpha_b(s) \implies s \in \lambda_b \quad (4.5)$$

$$\phi_b(s) : \alpha_b(s) \implies \text{"310790"} \in \lambda_b \quad (4.6)$$

portanto:

$$\tilde{\lambda} = \lambda_a \times \lambda_b \quad (4.7)$$

$$\tilde{s} = \Psi^{-1}(s_a \in \lambda_a | s_b \in \lambda_b) \quad (4.8)$$

$$\tilde{\alpha}(\tilde{s}) \implies \tilde{s} \in \tilde{\lambda} \quad (4.9)$$

$$\tilde{\phi}(\tilde{s}) : \tilde{\alpha}(\tilde{s}) \quad (4.10)$$

$$\tilde{m} \models \tilde{\phi}(s) \implies \text{"bernardo310790"} \in \tilde{\lambda} \quad (4.11)$$

Observa-se que a cardinalidade do modelo composto é definida pelo produto das cardinalidades de cada $m_i \in \Xi$, dado por:

$$\mathcal{C}(\tilde{m}) = \prod_{i=1}^N \mathcal{C}(m_i) \quad (4.12)$$

Assim, se $\mathcal{C}(m_a) = 30.000$ e $\mathcal{C}(m_b) = 930.000$, então $\mathcal{C}(\tilde{m}) = 27.900.000.000$.

4.1.2 *Token*

Na composição estrutural do modelo composto, refere-se como **token** a *substring* correspondente a determinado modelo elementar. Desta forma, a *string* “bernardo310790” é composta pelos *tokens* “bernardo” e “310790”.

4.1.3 Modelo Crítico

Propõe-se a criação do termo **modelo crítico**, definido como: seja $M(\tilde{s})$ o conjunto de todos os modelos (elementares e compostos) cujos domínios possuem \tilde{s} como membro.

$$M(\tilde{s}) = \{\tilde{m}_{i=1}, \dots, \tilde{m}_N\} | \forall \tilde{\lambda}_i, \tilde{s} \in \tilde{\lambda}_i$$

Seja \hat{m} o modelo com menor cardinalidade em $M(\tilde{s})$. Chama-se $\hat{m}(\tilde{s})$ de **modelo crítico** de \tilde{s} , dado por:

$$\hat{m}(\tilde{s}) = \arg \min_{\tilde{m}} \mathcal{C}(\tilde{m}_i), \forall \tilde{m}_i \in M(\tilde{s}) \quad (4.13)$$

Seja \mathcal{M}_Γ o **multiconjunto de modelos críticos** capazes de gerar cada *string* no multiconjunto de amostras Γ , dado por:

$$\Gamma = \{\tilde{s}_{j=1}, \dots, \tilde{s}_L\} \implies \mathcal{M}_\Gamma = \{\hat{m}(\tilde{s}_{j=1}), \dots, \hat{m}(\tilde{s}_L)\} \quad (4.14)$$

Utiliza-se $\hat{m}(\tilde{s}_j)$ e \hat{m}_j com o mesmo significado, denotando o domínio do modelo crítico \hat{m}_j por $\hat{\lambda}_j$.

4.1.4 Frequência Relativa

Como \mathcal{M}_Γ é multiconjunto, podem ocorrer membros \hat{m}_i repetidos. Seja n_i o número de ocorrências de cada \hat{m}_i em \mathcal{M}_Γ , e seja $|\mathcal{M}_\Gamma|$ o número total de membros de \mathcal{M}_Γ , incluindo repetições. Define-se então θ_i como a **frequência relativa** de \hat{m}_i em \mathcal{M}_Γ , dado por:

$$\theta(\tilde{m}_i) = \frac{n_i}{|\mathcal{M}_\Gamma|} \quad (4.15)$$

4.1.5 Entropia de Modelo

A entropia pode ser definida com diversas bases para o logaritmo. Neste trabalho é utilizado o logaritmo de base 10 em função da ordem de grandeza dos dados. Portanto, como $X \in \chi = \{x_{i=1}, \dots, x_N\}$, tem-se:

$$H(X) = \sum_i^N H(x_i) = - \sum_i^N P(x_i) \log_{10} P(x_i) \quad (4.16)$$

onde $H(X)$ é a entropia da variável aleatória discreta X . Neste trabalho, a entropia do modelo elementar $m(s)$ é dada por:

$$H(m(s)) = - \sum_{i=1}^N P(s_i) \log_{10} P(s_i) \quad (4.17)$$

onde $s_i \in \lambda = \{s_{i=1}, \dots, s_N\}$, ou seja, o somatório leva em conta todas as probabilidades de ocorrência de cada *string (token)* contida no domínio do modelo elementar. No caso do modelo composto crítico $\hat{m}(\tilde{s})$, a entropia é definida como o somatório das entropias de cada modelo elementar contido no mesmo, dado por:

$$H(\hat{m}(\tilde{s})) = H(m_a(s)) + H(m_b(s)) + H(m_c(s)) + \dots \quad (4.18)$$

4.2 Corinda

A arquitetura do software baseia-se na análise de espaços amostrais de senhas. Cada espaço amostral consiste de arquivo no formato *comma-separated value* (CSV), compactado no formato GZIP, que é do gênero compactador de arquivos, que gera re-

apresentação eficiente de vários arquivos dentro de único arquivo, ocupando menos espaço em mídia. Cada entrada do arquivo consiste de uma senha, acompanhada pela respectiva frequência de ocorrência. As entradas são ordenadas de forma decrescente de frequência. Os arquivos consistem das listas *RockYou*, *LinkedIn*, e *AntiPublic*.

A análise dos conjuntos amostrais tem como objetivo encontrar os modelos críticos correspondentes a cada senha da lista. Parâmetros como entropia, complexidade, e frequência relativa dos modelos elementares e compostos são coletados no processo denominado **treinamento**.

Uma vez que a etapa de treinamento foi concluída, o **Corinda** pode ser utilizado para efetivamente **quebrar** *hashes*. Para cada modelo composto detectado na etapa de treinamento, é lançada uma instância da gorrotina `composite.Model.Guess()`, responsável por gerar palpites de acordo com o modelo. A carga computacional alocada a cada gorrotina é proporcional à força do respectivo modelo.

O **Corinda** é implementado na forma de lista de arquivos:

- 1) **corinda/elementary/elementary.go**: contendo o pacote de modelo elementar.
- 2) **corinda/composite/composite.go**: contendo o pacote de modelo composto.
- 3) **corinda/train/train.go**: contendo o pacote de treinamento.
- 4) **corinda/crack/crack.go**: contendo o pacote de quebra de senhas.

A interação com o usuário no **Corinda** acontece com auxílio da biblioteca Cobra, que é biblioteca utilizada em linguagem **Go** para a criação de interface de comandos via *console*. Cada arquivo sob o diretório `corinda/cmd/` representa o respectivo comando na interface de usuário.

- a) **corinda/cmd/train.go**: treina modelos estatísticos a partir de conjuntos amostrais.
- b) **corinda/cmd/crack.go**: usa modelos treinados para quebrar listas de *hashes*.
- c) **corinda/main.go**: para execução da gorrotina principal.

Opta-se por subdividir a construção do **Corinda** em Pacotes, facilitando o entendimento da metodologia proposta.

4.3 Pacote Modelo Elementar

Modelos elementares são implementados no pacote `elementary`, na forma da estrutura `elementary.Model`. O campo `elementary.Model.Name` representa o nome do modelo. O campo `elementary.Model.Entropy` representa a entropia do modelo, e o campo `elementary.Model.TokenFreqs` representa o mapa de frequências dos *tokens* no espaço amostral.

Os modelos elementares implementam os métodos:

- a) `elementary.Model.UpdateEntropy()`
- b) `elementary.Model.UpdateTokenFreq()`
- c) `elementary.Model.SortedTokens()`

4.3.1 Método `elementary.Model.UpdateEntropy()`

Este método é responsável por calcular a entropia do modelo elementar m . Primeiro, é calculado o somatório das frequências dos diferentes tokens s_i no domínio `elementary.Model.TokenFreqs` do modelo. Então, as frequências relativas $P(s_i)$ de todos elementos de `elementary.Model.TokenFreqs` são computadas e a entropia `elementary.Model.Entropy` é calculada como o somatório, dada por:

$$H(m) = - \sum P(s_i) \log_{10} P(s_i) \quad (4.19)$$

4.3.2 Método `elementary.Model.UpdateTokenFreq()`

Este método é responsável por atualizar o campo `elementary.Model.TokenFreqs`. Caso o *token* já exista previamente no mapa, o valor da frequência é adicionado. Caso o *token* ainda não exista, a nova entrada é adicionada, com a *string* do *token* agindo como chave e a frequência agindo como valor no mapa `elementary.Model.TokenFreqs`.

4.3.3 Método `elementary.Model.SortedTokens()`

Este método é responsável por retornar o *array* de *strings* com os *tokens* do modelo elementar, organizados em ordem decrescente de frequência de ocorrência no conjunto amostral.

4.4 Pacote Modelo Composto

Modelos compostos são implementados no pacote `composite`, na forma da estrutura `Model`. O campo `Name` representa o nome do modelo composto. O campo `Freq` representa a frequência de ocorrência do modelo composto no espaço amostral. O campo `Prob` representa a probabilidade de ocorrência (frequência relativa) do modelo composto no espaço amostral. O campo `Entropy` representa a entropia do modelo composto. O campo `Models` consiste do *array* de *strings* relativos aos nomes dos modelos elementares que compõem o modelo composto.

Os modelos compostos implementam os métodos:

- a) `composite.Model.UpdateProb()`
- b) `composite.Model.UpdateFreq()`
- c) `composite.Model.UpdateEntropy()`
- d) `composite.Model.recursive()`
- e) `composite.Model.Guess()`
- f) `composite.Model.digest()`

4.4.1 Método `composite.Model.UpdateProb()`

Este método é responsável por atualizar a frequência relativa $\theta(\hat{m})$ do modelo composto, representada pelo objeto `cm.Prob`. O parâmetro `cm.Freq` representa a frequência de ocorrência do modelo composto crítico no conjunto amostral, enquanto o parâmetro `freqSum` representa o somatório das frequências de todos os modelos compostos no conjunto amostral.

4.4.2 Método `composite.Model.UpdateFreq()`

O método `composite.Model.UpdateFreq()` é responsável por atualizar o parâmetro `composite.Model.Freq` do modelo composto.

4.4.3 Método `composite.Model.UpdateEntropy()`

O método `composite.Model.UpdateEntropy()` é responsável por atualizar o parâmetro `composite.Model.Entropy`. O mapa `elementaries` contém ponteiros para os modelos elementares que compõem o modelo composto \hat{m} em questão, referenciado por `cm`.

A entropia do modelo composto é estabelecida como o somatório das entropias dos modelos elementares que o compõem.

4.4.4 Método `composite.Model.recursive()`

Este método é responsável pela geração recursiva do produto cartesiano $\tilde{\lambda}$ dos domínios dos modelos elementares. Por exemplo, sejam os conjuntos λ_1 , λ_2 e λ_3 dos domínios dos modelos elementares que compõem \hat{m} . O algoritmo recursivo produz o produto cartesiano $\tilde{\lambda} = \lambda_1 \times \lambda_2 \times \lambda_3$.

4.4.5 Método `composite.Model.Guess()`

O método `composite.Model.Guess()` é responsável por gerar os palpites de senhas. Ele invoca o método `composite.Model.recursive()` para enviar *strings* de palpites pelo canal de saída. As *strings* são compostas de acordo com o produto cartesiano dos domínios dos modelos elementares listados no *array* `composite.Model.Models`.

4.5 Pacote de Treinamento

O pacote de treinamento fornece as rotinas necessárias para análises dos conjuntos amostrais. Este pacote está dividido em:

- a) `train.Train.generator()`
- b) `train.Train.batchAnalyzer()`
- c) `train.Train.batchDecoder()`
- d) `train.Train.mapsMerger()`
- e) `train.Train.mapsSaver()`

O arquivo de entrada `input.csv` contém a lista de pares ordenados `freq`, `password` no formato *Comma Separated Value* (CSV).

4.5.1 Gorrotina `train.Train.generator()`

A gorrotina `train.Train.generator()` é responsável por transformar cada linha do arquivo de entrada em objetos do tipo `input` e enviar vetores do tipo `inputBatch` pelo canal de saída. O número de objetos `input` em cada vetor `inputBatch` é determinado pelo parâmetro `batchSize`.

4.5.2 Gorrotina `train.Train.batchAnalyzer()`

A gorrotina `train.Train.batchAnalyzer()` é responsável por invocar a *Java Virtual Machine* (JVM) com a *thread* do *Passfault*. Objetos do tipo `result` compõem os vetores `resultBatch`, com o mesmo número de elementos `batchSize`. Cada vetor `resultBatch` é enviado pelo canal `out`, enquanto a variável `c` indica o número de senhas processadas no tempo. Cada elemento do tipo `result` contém a frequência de ocorrência da senha, bem como as informações de modelos elementares e crítico encontradas pelo *Passfault*. O *Passfault* organiza tais informações no formato *JavaScript Object Notation* (JSON) e as codifica na forma de *arrays* de *bytes*.

A gorrotina `batchDecoder()` é responsável por separar as informações de cada objeto `result` em pares de objetos do tipo `elementaryJSON` e `compositeJSON`. As informações contidas em cada par de objetos JSON é adicionada ao objeto `trainedMaps`, onde dois mapas são utilizados para referenciar os diferentes modelos elementares e compostos encontrados na senha. Para cada `resultBatch` recebida por `train.Train.batchDecoder()`, um objeto `trainedMaps` é enviado pelo canal `tmChan`.

4.5.3 Gorrotina `train.Train.mapsMerger()`

Enquanto cada objeto `train.Train.trainedMaps` contém pares de mapas com informações de `batchSize` senhas, o objeto `finalMaps` contém pares de mapas responsável por todas as senhas do conjunto amostral. A gorrotina `crack.Crack.mapsMerger()` é responsável por receber os diversos objetos `trainedMaps` e concatená-los no par de mapas do objeto `finalMaps`. A variável `m` é utilizada para contabilizar o número de mapas processados no tempo.

As operações de união dos diferentes mapas provenientes dos lotes de senhas possui alto custo computacional. De forma a evitar que a gorrotina `crack.Crack.mapsMerger()` seja gargalo no processamento de informação, é importante que lotes inteiros de `batchSize` senhas sejam processados a cada iteração. Assim, a gorrotina `crack.Crack.mapsMerger()` está sempre unindo mapas em lotes

de `batchSize`.

4.5.4 Gorrotina `train.Train.mapsSaver()`

Por fim, a gorrotina `train.Train.mapsSaver()` é responsável por reorganizar as informações dos mapas finais dos modelos elementares e compostos (`finalMaps`) e salvá-las no arquivo de saída `output.json`.

Antes de enviar os modelos elementares e compostos, a gorrotina invoca os métodos `composite.Model.UpdateProb()`, `composite.Model.UpdateFreq()`, `composite.Model.UpdateEntropy()`, e `elementary.Model.UpdateEntropy()` para atualizar os parâmetros internos dos modelos elementares e compostos encontrados durante o processo de treinamento.

4.5.5 Fluxograma do Pacote de Treinamento

A Figura 4.1 apresenta o fluxograma do funcionamento do pacote de treinamento do **Corinda**. A Figura 4.2 dispõe as legendas dos objetos contidos no fluxograma. O arquivo `input.csv` é utilizado como entrada para a gorrotina `train.Train.generator()`. Objetos do tipo `input` são agrupados em objetos `inputBatch` (lote de entradas), e são enviados no canal de saída desta gorrotina.

A gorrotina `train.Train.batchAnalyze()` é responsável por inicializar a JVM e o *Passfault*. Os objetos `input` são recebidos e para cada entrada é gerado um objeto `result`. Objetos do tipo `result` são agrupados em `resultBatch` e enviados no canal de saída da gorrotina. A variável *c* computa o número de entradas processadas pelo *Passfault*. A gorrotina `train.Train.batchDecoder()` recebe os lotes de objetos `result` e os decodifica em objetos do tipo `elementaryJSON` e `compositeJSON`. Para cada lote `resultBatch` é criado o mapa `trainedMaps` contendo os modelos elementares e compostos identificados no lote. Os mapas `trainedMaps` são enviados no canal de saída.

A gorrotina `train.Train.mapsMerger()` é responsável por receber os diferentes `trainedMaps` relativos aos lotes de entradas e criar o único mapa contendo todos os modelos elementares e compostos do conjunto amostral. A variável *m* computa o número de mapas processados. A gorrotina `train.Train.reporter()` é responsável por ler os ponteiros das variáveis *c* e *m* e imprimir no *console* o progresso do processo de treinamento. A gorrotina `train.Train.mapsSaver()` é responsável por receber o mapa final e salvá-lo no arquivo de saída `models.json`.

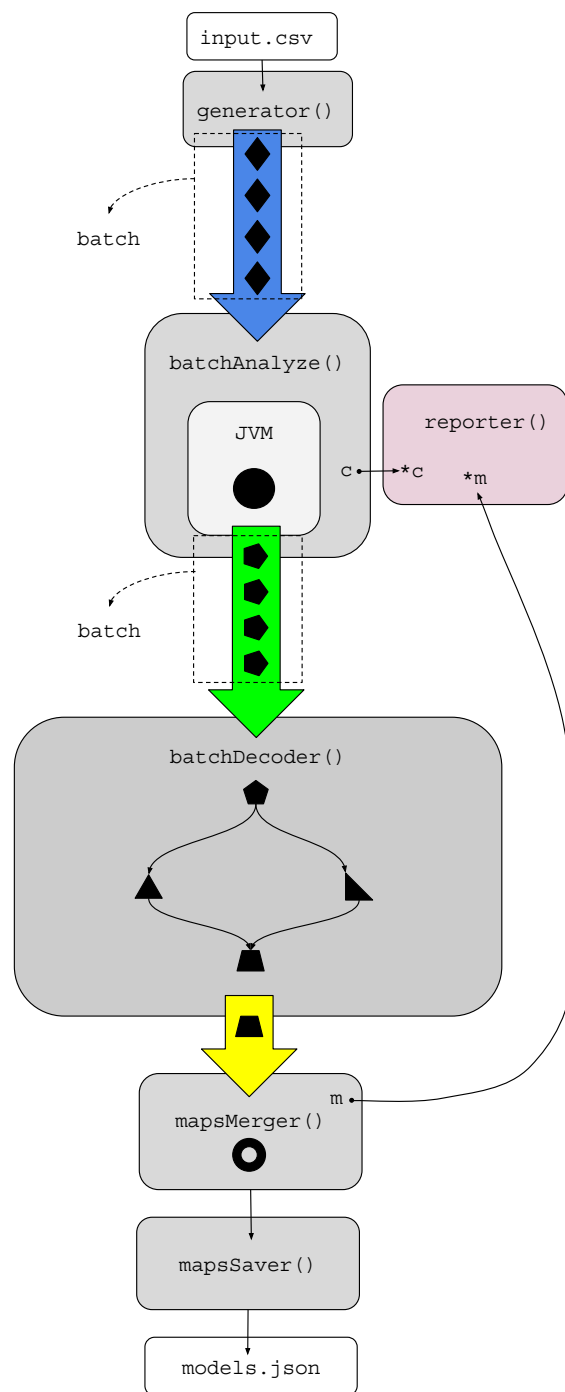


Figura 4.1: Fluxograma do Processo de Treinamento.

4.6 Pacote de Quebra de Senhas

O pacote de quebra de senhas fornece o código necessário para a implementação da metodologia heurística de quebra de senhas. A gorrotina `crack.Crack()` gerencia os processos de quebra de senhas. O arquivo de entrada `models.json` contém os mapas

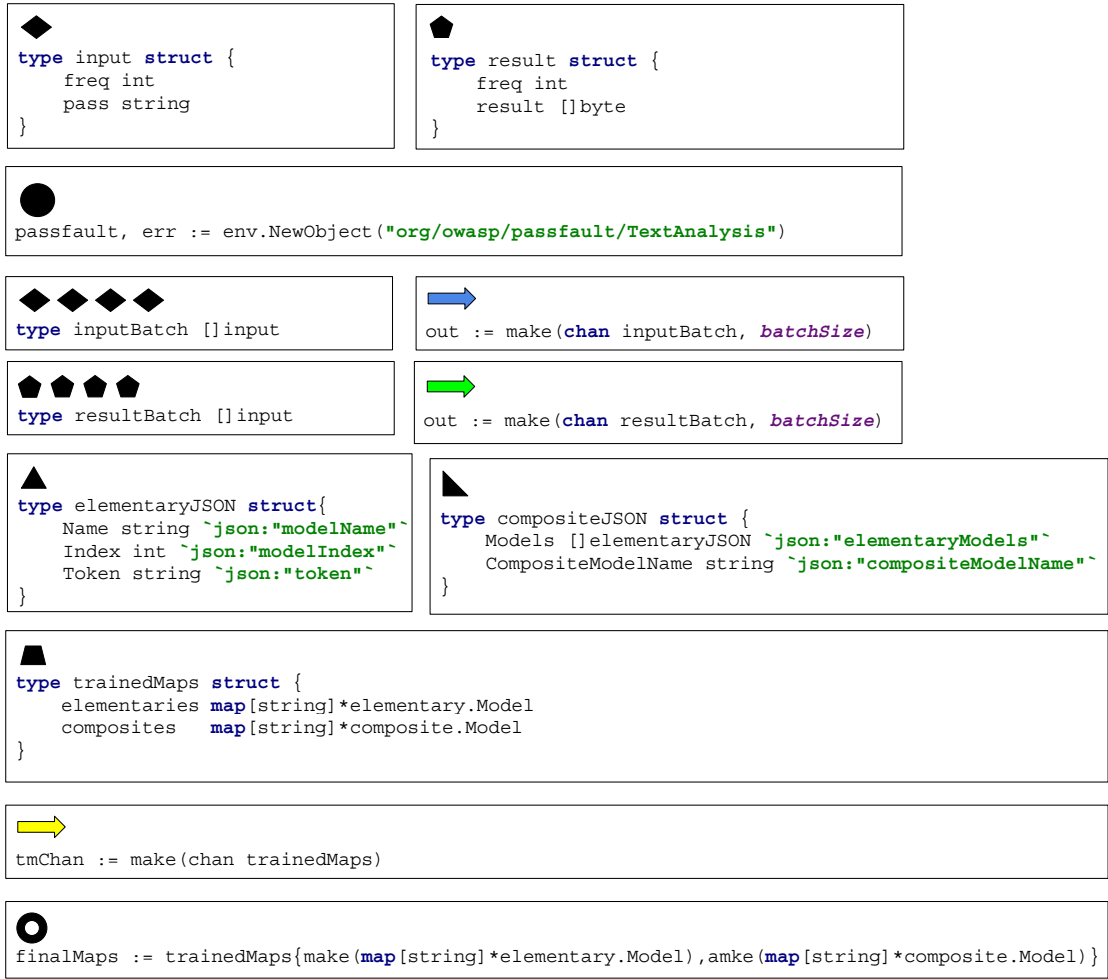


Figura 4.2: Legendas do Fluxograma do Processo de Treinamento.

com ponteiros para os modelos elementares e compostos encontrados no conjunto amostral durante o processo de treinamento.

4.6.1 Gorrotina de geração de palpites

Para cada modelo composto do arquivo de entrada, é lançada a gorrotina `composite.Model.Guess()`, responsável por gerar palpites na forma de *strings*. A cada gorrotina é atribuído o parâmetro `nGuesses`, descrito por:

$$\text{nGuesses}(\hat{m}_i) = \theta(\hat{m}_i) \cdot H(\hat{m}_i) \quad (4.20)$$

O termo $\theta(\hat{m}_i)$ representa a frequência relativa do modelo composto \hat{m}_i no conjunto

amostral Γ , enquanto o termo $H(\hat{m}_i)$ representa a entropia de \hat{m}_i em Γ .

Esta modelagem iterativa permite que modelos com altos valores de $\theta(\hat{m}_i)$ compo-
nham maior parcela de palpites `guess` no lote `batch` de palpites. Da mesma forma,
modelos com altas entropias $H(\hat{m}_i)$ também geram mais palpites por lote. Assim,
a cada iteração, um único lote é processado, e a carga computacional efetivamente
direcionada a cada modelo \hat{m}_i é parametrizada pelo indicador `nGuesses`(\hat{m}_i).

A organização das diversas gorrotinas geradoras de palpites é produzida de maneira
iterativa. A cada iteração, cada modelo composto gera seus respectivos `nGuesses`
palpites. Tal organização garante que a carga computacional alocada a cada modelo
composto seja proporcional à entropia e frequência relativa do mesmo. Cada lote de
palpites gerado ao fim de cada iteração é descrito pelo elemento `batch`. Cada lote
possui total T de palpites:

$$T = \sum_{i=0}^N \text{nGuesses}(\hat{m}_i) \quad (4.21)$$

Assim, a carga computacional alocada a cada modelo \hat{m}_i é dada por:

$$\text{CPUload}(\hat{m}_i) = \frac{\text{nGuesses}(\hat{m}_i)}{T} \quad (4.22)$$

4.6.2 Gorrotina `crack.Crack.guessLoop()`

A gorrotina `crack.Crack.guessLoop()` é responsável por convergir o fluxo de pal-
pites das diversas gorrotinas em canal único. A cada iteração do processo de quebra,
o número de objetos `password` gerados por cada modelo composto é condicionado
pela entropia e frequência relativa do mesmo.

4.6.3 Gorrotina `composite.Model.digest()`

A gorrotina `composite.Model.digest()` é responsável por transformar os diferen-
tes palpites na forma de *strings* para o tipo `password`. Tal processo consiste em
computar a FDC (SHA1 ou SHA256) calculada a partir da *string* e armazenar o
hash resultante na forma de *array* de *bytes*.

4.6.4 Gorrotina `crack.Crack.searcher()`

A gorrotina `crack.Crack.searcher()` é responsável por comparar os *hashes* dos diferentes palpites gerados contra os *hashes* alvo. Para cada palpite bem sucedido (palpite que produz *hash* contido na lista de alvos), o respectivo objeto `password` é encaminhado para o canal de saída da gorrotina. Esta gorrotina atua como filtro, e o total de palpites bem sucedidos é igual ao número t de *hashes* de palpites que foram encontrados na lista de *hashes* alvo.

4.6.5 Gorrotina `crack.Crack.saver()`

A gorrotina `crack.Crack.saver()` é responsável por salvar os resultados das senhas efetivamente quebradas. Ela recebe os objetos `password` e os salva no arquivo `output.csv`. Os resultados são salvos na forma de pares ordenados `password, hash`, onde o *hash* é representado no formato ASCII para visualização.

4.6.6 Gorrotina `crack.Crack.monitor()`

A gorrotina `monitor()` é responsável por monitorar o tempo decorrido desde o início da sessão. O objeto `wg` do tipo `sync.WaitGroup` atua como primitiva de sincronização. O valor em horas da duração total do experimento é comparada com o limite `durationH`. Caso o limite de tempo seja excedido, o objeto `wg` é utilizado para finalizar todos os processos de quebra e sair do programa.

4.6.7 Gorrotina `crack.Crack.reporter()`

A gorrotina `crack.Crack.reporter()` é responsável por armazenar o progresso da sessão no diretório `log_sessions`.

4.6.8 Fluxograma do Pacote de Quebra de Senhas

A Figura 4.3 apresenta o fluxograma do funcionamento do pacote de quebra de senhas do **Corinda**. A Figura 4.4 dispõe as legendas dos objetos contidos no fluxograma. O arquivo `models.json` é utilizado como entrada para a gorrotina `crack.Crack.Crack()`. A gorrotina carrega os diferentes objetos `composite.Model` referentes a cada modelo composto detectado no processo de treinamento. Cada objeto `composite.Model` é responsável por inicializar sua respectiva gorrotina `composite.Model.Guess()`, que gera `nGuesses(m)` palpites e os envia no canal de saída. Cada palpite é representado pelo objeto `guess`.

A gorrotina `crack.Crack.guessLoop()` é responsável por receber os objetos `guess`

dos diferentes modelos compostos e os enviar em único canal de saída. A cada iteração, são recebidos T palpites para compor o lote, indicado pela linha traçada no objeto `batch` da Figura 4.3. A gorrotina `crack.Crack.digest()` é responsável por calcular o *hash* de cada palpite recebido. Para cada objeto `guess` recebido do lote de palpites, a gorrotina encaminha um objeto `password` no canal de saída.

O objeto `targetsMap` contém o mapa dos *hashes* alvo a serem quebrados. A gorrotina `crack.Crack.searcher()` é responsável por comparar os *hashes* dos objetos `password` com os *hashes* alvo de `targetsMap`. Para cada lote de palpites, a gorrotina encaminha os t objetos `password` (correspondentes às senhas quebradas) no canal de saída, onde $t \leq T$. A gorrotina `crack.Crack.saver()` é responsável por receber os objetos `password` correspondentes às senhas quebradas no arquivo de saída `result.csv`.

A gorrotina `crack.Crack.monitor()` é responsável por monitorar o tempo de execução do processo de quebra. Quando o critério de parada tempo limite para o experimento é atingido, esta gorrotina interrompe o fluxo do programa. A gorrotina `crack.Crack.reporter()` é responsável por contabilizar o tempo decorrido desde o início do experimento, bem como o número total de senhas quebradas até o momento. O progresso do experimento é impresso no *console* e armazenado no diretório `/log_sessions`.

4.7 Pacote da Interface de Comandos

A biblioteca Cobra é utilizada para organizar e abstrair os comandos para o usuário. Aplicações baseadas na biblioteca Cobra possuem estruturas de comandos, argumentos, e *flags*. Os comandos disponíveis são `crack`, `help`, e `train`.

4.7.1 Comando train

O comando `train` é responsável por invocar o pacote `train/train.go`, executado por: `corinda train <conjunto amostral>`. O argumento `<conjunto amostral>` indica qual conjunto amostral deve ser utilizado no processo de treinamento (`rockyou`, `linkedin` ou `antipublic`). Por exemplo: `corinda train rockyou`.

4.7.2 Comando crack

O comando `crack` é responsável por invocar o pacote `crack/crack.go`, executado por: `corinda crack <conjunto amostral> <alvo> <fdc>` onde o argumento `<conjunto amostral>` indica o conjunto amostral a ser usado para gerar os palpites,

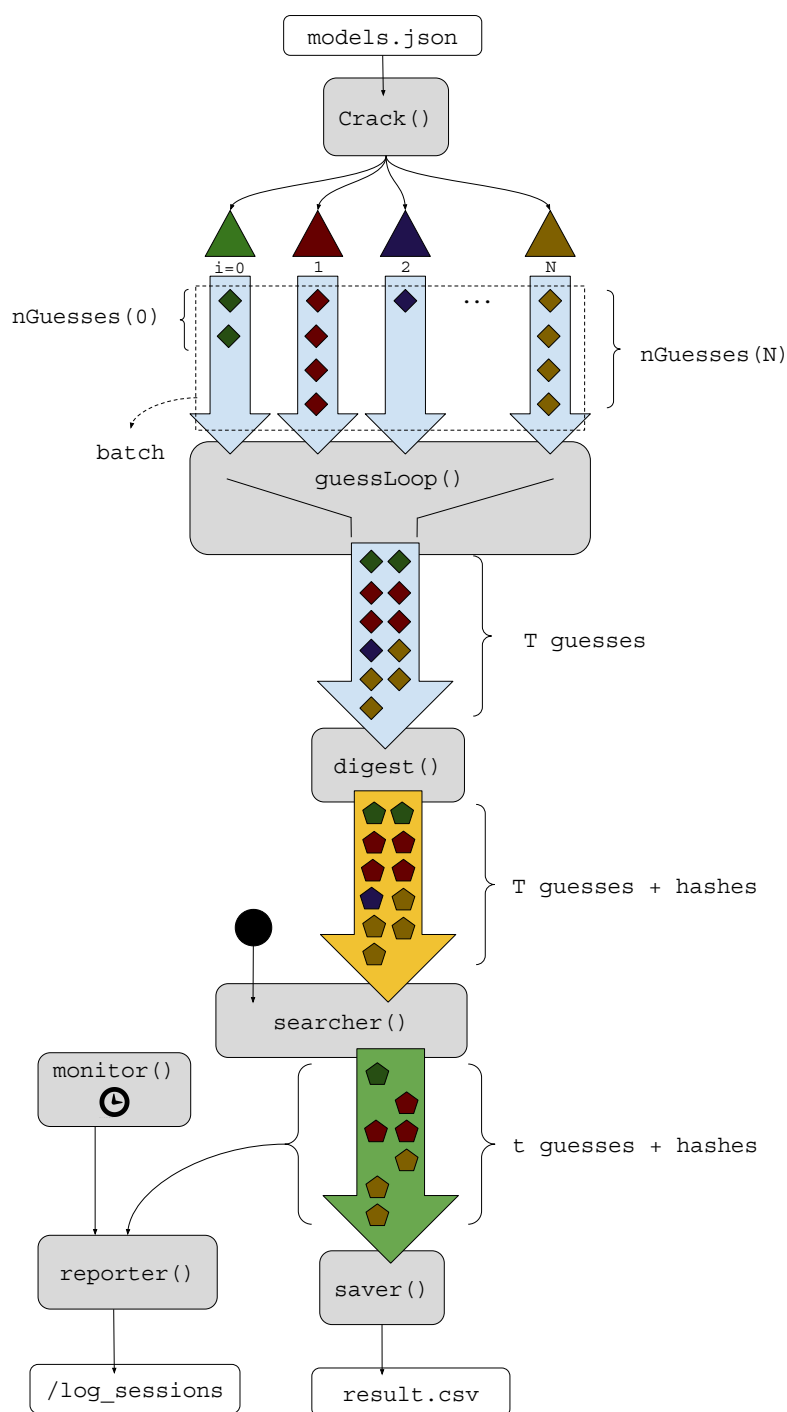


Figura 4.3: Fluxograma do Processo de Quebra de Senhas.

e <alvo> indica a lista de *hashes* alvo, e o argumento <fdc> indica o tipo de FDC a ser utilizada (*sha1* ou *sha256*). Ambos argumentos podem ser *rockyou*, *linkedin* ou *antipublic*. Por exemplo, para utilizar os modelos extraídos da lista *rockyou* para quebrar os *hashes* SHA1 da lista *linkedin*, utiliza-se o comando: *corinda*

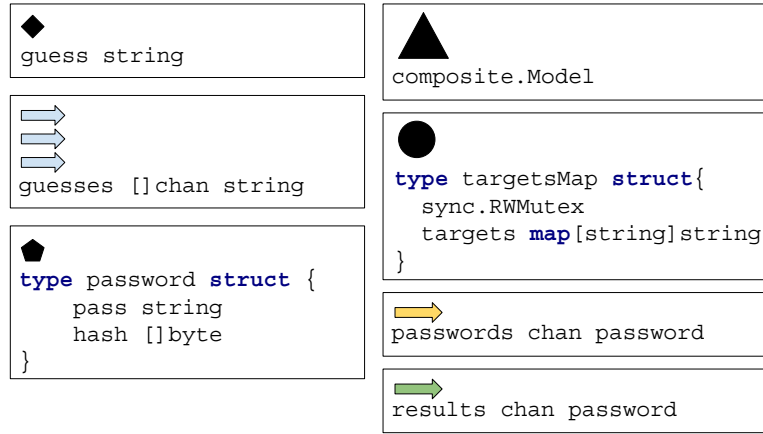


Figura 4.4: Legendas do Fluxograma do Processo de Quebra de Senhas.

crack rockyou linkedin sha1.

4.8 Experimentos e Validação

Os experimentos a serem realizados são divididos em duas categorias: i) experimentos preliminares e ii) validação do **Corinda**. De forma a obter avaliação inicial dos dados sobre listas de senhas, utiliza-se o *Passfault* como ferramenta de identificação de modelos críticos de senhas e suas cardinalidades. A Tabela 4.1 dispõe os possíveis modelos elementares que o *Passfault* é capaz de encontrar.

Listas com palavras de sete diferentes idiomas são utilizadas como dicionários: alemão, inglês, espanhol, italiano, holandês e português. Sabe-se que idiomas seguem distribuições de Zipf, o que significa que a frequência de cada palavra é inversamente proporcional à sua posição no ranking de frequências. Assim, para cada idioma, dois dicionários são utilizados: um com a **cabeça** da distribuição (80% mais frequentes), e outra com a **cauda longa** da distribuição.

4.8.1 Validação do Corinda

De forma a validar o desempenho do **Corinda**, são realizados experimentos que simulam cenários onde o atacante utiliza o conjunto de amostras Γ para treinar o conjunto de modelos críticos \mathcal{M}_Γ com o objetivo de quebrar os *hashes* pertencentes ao conjunto $F(\Gamma')$.

São utilizados os conjuntos amostrais: i) *RockYou*, ii) *LinkedIn* e iii) *AntiPublic*. Cada conjunto amostral consiste de arquivo no formato CSV, onde cada linha contém o

Tabela 4.1: Modelos elementares reconhecidos pelo *Passfault*.

Modelo Crítico	Descrição	Exemplo
Correspondência Exata em Dicionário	Sequência de caracteres encontrada no dicionário.	<i>john</i>
Correspondência Invertida em Dicionário	Sequência de caracteres encontrada no dicionário, porém com ordem invertida.	<i>nhoj</i>
Formato de Data	Sequência numérica em formato de data.	<i>073190</i>
Sequência Aleatória de Caracteres	Sequência de caracteres sem padrão reconhecido.	<i>a7mk0s</i>
Padrões de Teclado	Sequência de caracteres obedecendo a Padrões de disposição espacial comumente encontrados em teclados.	<i>zxcvbnm</i>
Caracteres Repetidos	Sequência de caracteres repetidos.	<i>aaaa</i>
Repetição de Sequência de Caracteres	Sequência de caracteres repete algum padrão já reconhecido.	<i>johnjohn</i>

valor {frequencia, senha}.

A natureza do algoritmo recursivo de geração dos palpites possui a limitação de não priorizar *tokens* de alta probabilidade. Assim, caso algum dos domínios dos modelos elementares em questão seja demasiadamente elevado, palpites com baixa probabilidade de sucesso podem ser gerados. De forma a investigar a influência do tamanho de Γ na eficácia de \mathcal{M}_Γ para gerar os palpites durante o processo de quebra, para cada conjunto amostral, também são criadas versões truncadas, limitadas a 1 milhão de entradas. Para cada lista, apenas o primeiro milhão de senhas mais populares são utilizadas. Os conjuntos amostrais são referidos como: i) *RockYou_1M*, ii) *LinkedIn_1M*, e iii) *AntiPublic_1M*.

De forma a gerar as listas de *hashes* alvo, os conjuntos amostrais (completos) são convertidos em *hashes*, utilizando as FDC:

- **SHA1:** o *Secure Hash Algorithm 1* é a FDC estabelecida pela Agência de Segurança Nacional dos Estados Unidos (NSA) em 1993 (DANG, 2013). O SHA1 produz *hash* de 20 *bytes*. Estudos publicados desde 2005 comprovam a possibilidade de colisão de *hashes* SHA1, fazendo com que esta não seja mais considerada FDC segura (WANG et al., 2005).
- **SHA256:** o SHA256 pertence à família de FDC conhecida como *Secure Hash Algorithm 2*, também estabelecida pela NSA. Produz *hash* de 256

bits. As fraquezas encontradas no SHA1 ainda não foram comprovadas no SHA256, fazendo com que este ainda seja considerado FDC segura (GLABB et al., 2007), (SANADHYA; SARKAR, 2007).

A estratégia dos experimentos consiste em escolher o conjunto de amostras para treinar \mathcal{M}_Γ e utilizá-lo para quebrar os *hashes* gerados a partir dos outros conjuntos. O mesmo processo é repetido para todas combinações possíveis dos três conjuntos *RockYou*, *LinkedIn* e *AntiPublic* e suas versões truncadas, bem como das FDC: SHA1 e SHA256.

4.9 Considerações

Neste capítulo foi apresentada a metodologia para o desenvolvimento do software denominado **Corinda**. Os conceitos de modelo elementar e modelo composto crítico complementam a teoria dos modelos de primeira ordem de forma a descrever a estrutura das senhas. A arquitetura do **Corinda** é dividida em quatro pacotes: `elementary`, `composite`, `train` e `crack`. Além disto, a interface de comandos é implementada sob o diretório `corinda/cmd`. O próximo capítulo apresenta os resultados obtidos decorrentes da metodologia proposta.

CAPÍTULO 5

RESULTADOS

Esta seção apresenta os resultados dos experimentos descritos na seção anterior. Primeiro, são apresentados os resultados dos experimentos preliminares. Então, são apresentadas as estatísticas encontradas durante os processos de treinamento com cada conjunto amostral. Em seguida, são apresentados os resultados dos experimentos de quebra de senhas com os conjuntos amostrais parciais. Por fim, são apresentados os resultados dos experimentos de quebra de senhas com os conjuntos amostrais completos.

5.1 Experimentos Preliminares

O total de 848.645 modelos críticos diferentes foi encontrado pelo *Passfault* no conjunto *RockYou*. Estes modelos compostos críticos consistem de permutações de 1.533 modelos elementares diferentes. A Tabela 5.1 apresenta os vinte modelos críticos frequentemente encontrados no conjunto. O tempo para quebra t_q corresponde ao cenário onde o *hash* SHA1 é atacado utilizando placa de vídeo NVidia GRID K520 a 423,4 milhões de *hashes* por segundo. Estes vinte modelos críticos foram encontrados em cerca de 10 milhões de senhas, o que corresponde a cerca de 27% dos 36,9 milhões de senhas contidas no conjunto.

O domínio definido por estes vinte modelos críticos possui cardinalidade igual a 206,8 milhões, o que significa que para FDC obsoleta como o SHA1, um atacante poderia obter as credencias de cerca de um quarto dos usuários em período de tempo na ordem de milisegundos.

O *ranking* de frequências dos modelos compostos é visualizado na Figura 5.1(a) onde o eixo das abcissas é o *ranking* dos modelos. A primeira posição é do modelo mais frequente e a última posição é do modelo menos frequente. O eixo das ordenadas apresenta o número de ocorrências de cada modelo. Os modelos seguem a distribuição de Zipf (Lei de Potência). Uma das hipóteses para explicar este comportamento está relacionada ao fato de que senhas seguem padrões linguísticos, que são conhecidos por seguir a distribuição de Zipf.

Se a quantidade de caracteres na senha (tamanho da senha) é baixo, tal como seis caracteres ou menos, a sua cardinalidade está limitada a valores relativamente pequenos, uma vez que até ataques de força bruta nesta magnitude são possíveis em curtos períodos de tempo. O histograma dos tamanhos das senhas é apresentado na

Figura 5.1(b), onde o eixo das abcissas é o número de caracteres na senha, enquanto que as barras paralelas ao eixo das ordenadas são as frequências de ocorrência de cada número de caracteres. O tamanho médio das senhas da lista *RockYou* é 7,86 caracteres, indicado pela linha cinza tracejada. O total de 9.970.733 senhas (27,97% da lista completa) possui seis caracteres ou menos.

Tabela 5.1: Vinte modelos compostos frequentemente utilizados no conjunto *RockYou*.

Modelo Crítico	Frequência	Cardinalidade	t_q
Correspondência em Dicionário: John The Ripper	1.423.805	3.545	8,3727 μs
Formato de Data	1.098.073	930.000	2,1965 ms
Correspondência em Dicionário: 500 Piores Senhas	998.980	500	1,1809 μs
Correspondência em Dicionário: 10k Piores Senhas	939.639	10.000	23,618 μs
Correspondência em Dicionário: “Cabeça” dos Nomes Americanos	736.001	926	2,1871 μs
6 Números Aleatórios	510.875	1.000.000	2,3618 ms
Correspondência em Dicionário: Nomes de Animais de Estimação	419.748	400	0,9447 μs
Correspondência em Dicionário: “Cauda Longa” Alemão	412.485	97.212	0,2296 ms
Correspondência em Dicionário: John The Ripper	354.587	354.500	0,8373 ms
-			
2 Números Aleatórios	351.678	10.000.000	23,6183 ms
7 Números Aleatórios			
Sequência Horizontal de 6 Caracteres em Teclado Americano	335.550	278	0,6567 μs
Correspondência em Dicionário: 500 Piores Senhas	319.673	50.000	0,1181 ms
-			
2 Números Aleatórios	315.149	35.450	83,7270 μs
Correspondência em Dicionário: 500 Piores Senhas			
-	314.824	93.000.000	219,6504 ms
1 Número Aleatório			
Formato de Data	290.586	92.600	0,2187 ms
-			
2 Números Aleatórios	286.861	1,000,000	2,3618 ms
Correspondência em Dicionário: “Cabeça” dos Nomes Americanos			
-	244.682	24.700	58,3372 μs
2 Números Aleatórios			
Correspondência em Dicionário: 10k Piores Senhas	237.737	100.000	0,2362 ms
-			
1 Número Aleatório	229,563	100.000.000	236.1833 ms
8 Números Aleatórios			
Correspondência em Dicionário: “Cauda Longa” Italiano	210.625	97.292	0,2298 ms
Total	10.031.121	206.797.403	488,4208 ms

5.2 Código Fonte

Apresenta-se alguns trechos do código fonte dos pacotes de modelos elementar e composto do **Corinda**. O código fonte completo de todos os pacotes pode ser encontrado no Apêndice A ao Apêndice G. O Algoritmo 5.1 ilustra a declaração do modelo elementar como estrutura `elementary.Model`. Os parâmetros `Name`, `Entropy` e `TokenFreqs` são declarados como campo da estrutura. A Figura 5.2 apresenta

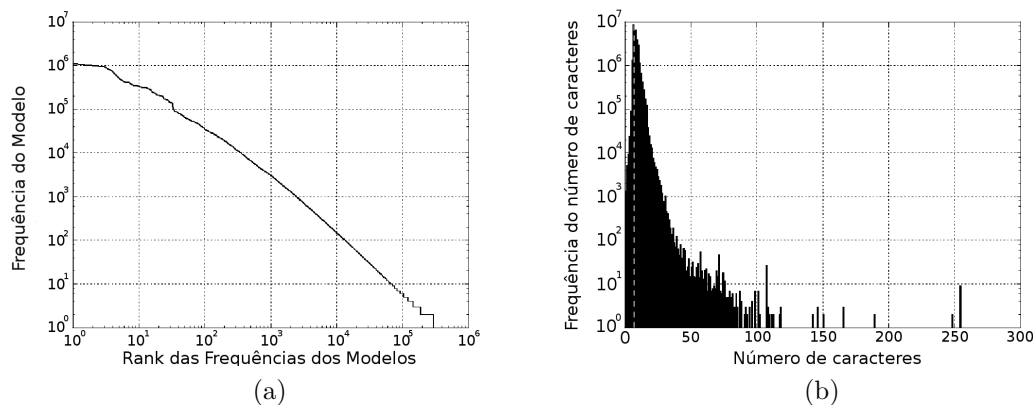


Figura 5.1: Experimentos preliminares com o conjunto *RockYou*: (a) ranking de frequências dos modelos compostos no conjunto, (b) histograma dos tamanhos das senhas.

exemplo de modelo elementar retirado de espaço amostral.

Algoritmo 5.1: Declaração da estrutura modelo elementar.

```
type Model struct {
    Name          string
    Entropy        float64
    TokenFreqs     map[string]int
}
```

Figura 5.2: Exemplo de modelo elementar retirado de espaço amostral.

```
Name: "10 Random Character(s):Numbers"
Entropy: 5.323614327609053
TokenFreqs: map[0837134726:1 0839526140:1 0857168644:1 ...]
```

O Algoritmo 5.2 apresenta o funcionamento do método `elementary.Model.UpdateEntropy()`. O primeiro *loop* iterativo calcula a soma das frequências de ocorrência de cada *token* do modelo elementar. O segundo *loop* realiza o cálculo da frequência relativa (probabilidade) de cada *token*, bem como a entropia, adicionando os valores da entropia ao somatório total. Por fim, o valor calculado da entropia do modelo elementar é associada ao campo `Entropy` do objeto `elementary.Model`.

Algoritmo 5.2: Método `elementary.Model.UpdateEntropy()`.

```
func (em *Model) UpdateEntropy(){

    sum := 0
    for _, freq := range em.TokenFreqs {
        sum += freq
    }

    entropy := float64(0)
    for _, freq := range em.TokenFreqs {
        f := freq
        p := float64(f)/float64(sum)
        e := -p*math.Log10(p)
        entropy += e
    }

    em.Entropy = entropy
}
```

O Algoritmo 5.3 apresenta o funcionamento do método `elementary.Model.UpdateTokenFreq()`, responsável por atualizar o campo `elementary.Model.TokenFreqs`. Como o campo é do tipo mapa, primeiro é verificado se o *token* já existe previamente no mesmo. Em caso positivo, o valor da frequência é adicionado ao valor existente e a entrada do mapa é atualizada. Caso o *token* ainda não exista, o *token* é adicionado como nova entrada no mapa.

Algoritmo 5.3: Método `elementary.Model.UpdateTokenFreq()`.

```
func (em *Model) UpdateTokenFreq(freq int, token string){

    if _, ok := em.TokenFreqs[token]; ok{
        f := em.TokenFreqs[token]
        em.TokenFreqs[token] = freq + f
    }else{
        em.TokenFreqs[token] = freq
    }

}
```

O Algoritmo 5.4 ilustra a declaração do modelo composto como estrutura `composite.Model`. Os parâmetros `Name`, `Freq`, `Prob`, `Entropy` e `Models` são declarados como campo da estrutura. O objeto `composite.Model.Freq` representa a frequência total de ocorrências do modelo composto no conjunto amostral, enquanto

`composite.Model.Prob` representa a frequência relativa do modelo composto. A Figura 5.3 apresenta exemplo de modelo composto crítico retirado de espaço amostral.

Algoritmo 5.4: Declaração da estrutura modelo composto.

```
type Model struct{
    Name      string
    Freq      int
    Prob      float64
    Entropy   float64
    Models    []string
}
```

Figura 5.3: Exemplo de modelo composto retirado de espaço amostral.

```
Name: "3 Random Character(s):Latin|Exact Match:JohnTheRipper"
Freq: 1586
Prob: 4.8660946479340633e-05
Entropy: 6.598703202825462
```

O Algoritmo 5.5 apresenta o funcionamento do método `composite.Model.UpdateFreq()`. O parâmetro de entrada `freq` é utilizado para atualizar o objeto `composite.Model.Freq`.

Algoritmo 5.5: Método `composite.Model.UpdateFreq()`.

```
func (cm *Model) UpdateFreq(freq int){
    cm.Freq = cm.Freq + freq
}
```

O Algoritmo 5.6 apresenta o funcionamento do método `composite.Model.UpdateProb()`. O parâmetro de entrada `freqSum` é utilizado para calcular a divisão de `composite.Model.Freq` e objeto `composite.Model.Prob`, é atualizado.

Algoritmo 5.6: Método `composite.Model.UpdateProb()`.

```
func (cm *Model) UpdateProb(freqSum int){
    cm.Prob = float64(cm.Freq)/float64(freqSum)
}
```

O Algoritmo 5.7 apresenta o funcionamento do método

`composite.Model.UpdateEntropy()`. O parâmetro de entrada `elementaries` é o mapa que contém todos modelos elementares encontrados no conjunto amostral. O *loop* iterativo varre os modelos elementares que compõem o modelo composto em questão, adicionando os valores da entropia de cada modelo elementar ao somatório. Por fim, o valor do somatório é atribuído ao campo `composite.Model.Entropy`.

Algoritmo 5.7: Método `composite.Model.UpdateEntropy()`.

```
func (cm *Model) UpdateEntropy(
    elementaries map[string]*elementary.Model){

    entropy := float64(0)

    for _, em := range cm.Models {
        entropy += elementaries[em].Entropy
    }

    cm.Entropy = entropy
}
```

5.3 Validação do Corinda

A estratégia dos experimentos consiste em escolher o conjunto de amostras para treinar \mathcal{M}_Γ e utilizá-lo para quebrar os *hashes* gerados a partir dos outros conjuntos. O mesmo processo é repetido para todas combinações possíveis dos três conjuntos (*RockYou*, *LinkedIn* e *AntiPublic*) e suas versões truncadas, como das FDC (SHA1 e SHA256). Cada \mathcal{M}_Γ é testado contra as três listas de *hashes*, totalizando nove sessões. Cada sessão é repetida utilizando cada uma das duas FDC, totalizando assim dezoito experimentos.

As listas de *hashes* alvo ficam armazenadas no diretório `/targets` do repositório do **Corinda**, nos subdiretórios `/sha1` e `/sha256`. Os resultados dos experimentos são armazenados no diretório `/result`. Os experimentos foram realizados no serviço de computação na nuvem *Google Cloud*. A instância utilizada possui dezesseis núcleos de CPU e 60 GB de Memória RAM. O sistema operacional utilizado foi o GNU/Linux Ubuntu 16.04.4 LTS.

A Tabela 5.2 dispõe as informações gerais sobre os conjuntos amostrais utilizados nos experimentos. O número de entradas N_e representa o total de linhas no arquivo, onde cada linha contém uma senha e sua respectiva frequência de ocorrência. O somatório de frequências S_f representa o somatório total das frequências de cada

senha e os caracteres por senha C_s representa o tamanho médio (de caracteres) das *strings* do conjunto.

Tabela 5.2: Conjuntos amostrais.

Γ	N_e	S_f	C_s
<i>RockYou</i>	14.336.603	31.453.096	7,867
<i>LinkedIn</i>	60.001.147	114.932.331	9,635
<i>AntiPublic</i>	192.360.478	561.046.229	9,073

5.4 Processos de treinamento

Os resultados relacionados ao processos de treinamento são baseados nos três conjuntos amostrais (*RockYou*, *LinkedIn* e *AntiPublic*). O processo de treinamento consiste em utilizar o *Passfault* para identificar o modelo crítico de cada senha do conjunto amostral. São apresentados os dez modelos compostos críticos frequentes, bem como os histogramas das distribuições estatísticas dos parâmetros entropia e frequência dos modelos identificados nos três conjuntos.

5.4.1 *RockYou*

A frequência relativa representa a probabilidade de ocorrência de determinado modelo composto crítico dentro do multiconjunto \mathcal{M}_Γ . A Tabela 5.3 apresenta os dez modelos compostos frequentemente encontrados na lista *RockYou*. Nota-se a predominância de modelos baseados em dicionários.

Tabela 5.3: Dez modelos compostos críticos frequentes na lista *RockYou*.

$\Theta(\hat{m}(s))$	$\hat{m}(s)$
0,0461	Dicionário: <i>JohnTheRipper</i>
0,0345	Formato de Data
0,0323	Dicionário: <i>500-worst-passwords</i>
0,0302	Dicionário: <i>10k-worst-passwords</i>
0,0294	Dicionário: <i>usFirstNamesLongTail</i>
0,0227	Dicionário: <i>usFirstNamesPopular</i>
0,0160	6 Números Aleatórios
0,0136	Dicionário: <i>petNames</i>
0,0115	Dicionário: <i>JohnTheRipper</i> 2 Números Aleatórios
0,0110	7 Números Aleatórios

A Figura 5.4(a) apresenta o histograma da distribuição das frequências relativas dos modelos compostos críticos encontrados na lista *RockYou*. Nota-se que o padrão da distribuição tende a assumir a forma de Lei de Potência.

A entropia atua como indicador de incerteza do modelo. Modelos com altas entropias geram senhas de difícil previsão, enquanto modelos com entropias tendendo a *zero* geram senhas de fácil previsibilidade. A Figura 5.4(b) apresenta o histograma da distribuição das entropias dos modelos elementares do conjunto *RockYou*. A Figura 5.4(c) apresenta o histograma da distribuição das entropias dos modelos compostos do conjunto *RockYou*. Nota-se que o padrão da distribuição das entropias dos modelos compostos tende a assumir a forma de distribuição Normal.

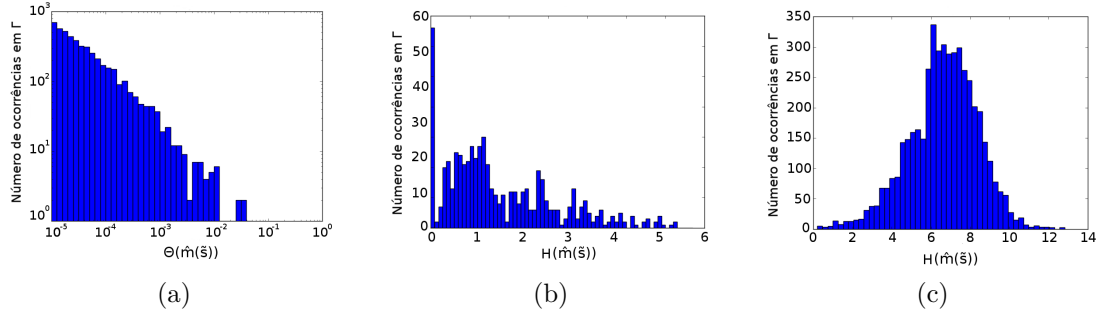


Figura 5.4: Histogramas: (a) frequências relativas nos modelos compostos da lista *RockYou*, (b) entropias dos modelos elementares na lista *RockYou*, (c) entropias dos modelos compostos na lista *RockYou*.

5.4.2 *LinkedIn*

A Tabela 5.4 apresenta os dez modelos compostos frequentemente encontrados na lista *LinkedIn*. Nota-se a predominância de modelos baseados em dicionários.

Tabela 5.4: Dez modelos compostos críticos frequentes na lista *LinkedIn*.

$\Theta(\hat{m}(s))$	$\hat{m}(s)$
0,0258	Dicionário: <i>10k-worst-passwords</i>
0,0229	Dicionário: <i>JohnTheRipper</i>
0,0228	15 Números Aleatórios
0,0217	Formato de Data
0,0173	Dicionário: <i>500-worst-passwords</i>
0,0173	Dicionário: <i>usFirstNamesLongTail</i>
0,0137	6 Números Aleatórios
0,0117	Dicionário: <i>deLongTail</i>
0,0101	Dicionário: <i>10k-worst-passwords</i>
0,0095	Dicionário: <i>usFirstNamesPopular</i>

A Figura 5.5(a) apresenta o histograma da distribuição das frequências relativas dos modelos compostos críticos encontrados na lista *LinkedIn*. A Figura 5.5(b) apresenta

o histograma da distribuição das entropias dos modelos elementares do conjunto *LinkedIn*. A Figura 5.5(c) apresenta o histograma da distribuição das entropias dos modelos compostos do conjunto *LinkedIn*. Nota-se que o padrão da distribuição das entropias dos modelos compostos tende a assumir a forma de distribuição Normal.

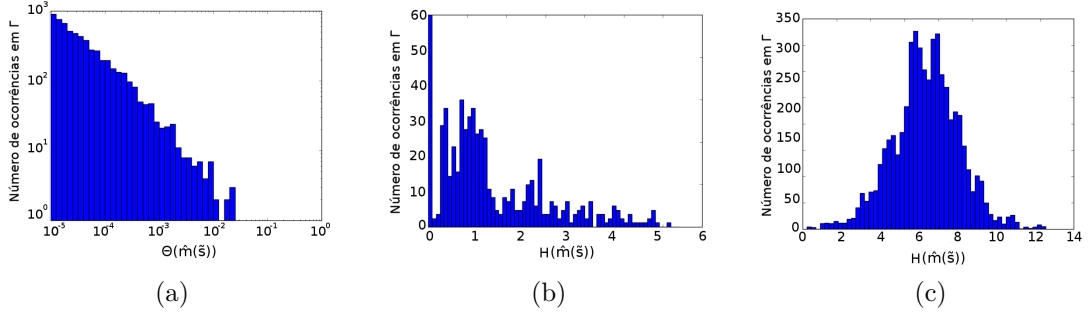


Figura 5.5: Histogramas: (a) frequências relativas nos modelos compostos da lista *LinkedIn*, (b) entropias dos modelos elementares na lista *LinkedIn*, (c) entropias dos modelos compostos na lista *LinkedIn*.

5.4.3 *AntiPublic*

A Tabela 5.5 apresenta os dez modelos compostos frequentemente encontrados na lista *AntiPublic*. Nota-se a predominância de modelos baseados em dicionários.

Tabela 5.5: Dez modelos compostos críticos frequentes na lista *AntiPublic*

$\Theta(\hat{m}(s))$	$\hat{m}(s)$
0,0241	Dicionário: <i>JohnTheRipper</i>
0,0216	Dicionário: <i>10k-worst-passwords</i>
0,0165	Formato de Data
0,0152	Dicionário: <i>10k-worst-passwords</i> 2 Números Aleatórios
0,0148	Dicionário: <i>JohnTheRipper</i> 1 Número Aleatório
0,0142	Dicionário: <i>10k-worst-passwords</i> 1 Números Aleatório
0,0141	Dicionário: <i>500-worst-passwords</i> 2 Números Aleatórios
0,0140	Dicionário: <i>JohnTheRipper</i> 2 Números Aleatórios
0,0129	8 Caracteres Aleatórios

A Figura 5.6(a) apresenta o histograma da distribuição das frequências relativas dos modelos compostos críticos encontrados na lista *AntiPublic*. A Figura 5.6(b) apresenta o histograma da distribuição das entropias dos modelos elementares do conjunto *AntiPublic*. A Figura 5.6(c) apresenta o histograma da distribuição das entropias dos modelos compostos do conjunto *AntiPublic*. Nota-se que o padrão da distribuição das entropias dos modelos compostos tende a assumir a forma de

distribuição Normal.

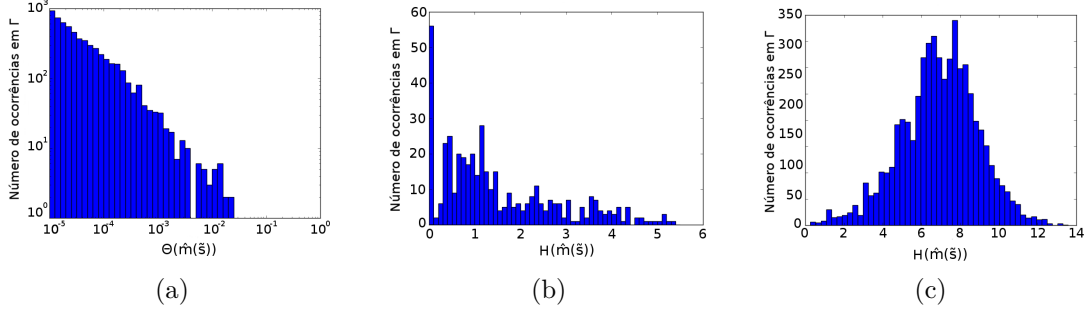


Figura 5.6: Histogramas: (a) frequências relativas nos modelos compostos da lista *AntiPublic*, (b) entropias dos modelos elementares na lista *AntiPublic*, (c) entropias dos modelos compostos na lista *AntiPublic*.

A Tabela 5.6 apresenta as médias \bar{H} e desvios padrão σ das distribuições das entropias dos modelos compostos.

Tabela 5.6: Média e desvio padrão das entropias dos modelos compostos.

Γ	\bar{H}	σ
<i>RockYou</i>	6,626	1,765
<i>LinkedIn</i>	7,279	1,972
<i>AntiPublic</i>	6,917	1,991

5.5 Quebra de Senhas

A partir da metodologia proposta foram realizados os experimentos de quebra de senhas, onde os modelos identificados nos conjuntos amostrais são utilizados para gerar palpites para quebra dos *hashes* da lista alvo.

As Tabelas 5.7 à Tabela 5.10 apresentam os números de *hashes* e senhas quebradas. Em Γ tem-se o nome do conjunto amostral. Em h tem-se o número de *hashes* efetivamente quebrados, enquanto que $\%h$ apresenta a porcentagem de *hashes* quebrados em relação ao número total de *hashes* da lista alvo. Em S tem-se o somatório das frequências de ocorrência de cada *hash* quebrado, enquanto $\%S$ apresenta a porcentagem que S representa do somatório total S_f de frequências das senhas na lista.

A Tabela 5.7 apresenta os resultados dos experimentos de quebra dos *hashes* SHA1.

Os conjuntos amostrais utilizados foram completos, sem truncamento. A Tabela 5.8 apresenta os resultados dos experimentos de quebra dos *hashes* SHA1, com a utilização de conjuntos amostrais truncados em 1 milhão de entradas.

A Tabela 5.9 apresenta os resultados dos experimentos de quebra dos *hashes* SHA256. Os conjuntos amostrais utilizados foram completos, sem truncamento. A Tabela 5.10 apresenta os resultados dos experimentos de quebra dos *hashes* SHA256, com a utilização de conjuntos amostrais truncados em 1 milhão de entradas.

Tabela 5.7: Experimentos de quebra de *hashes* SHA1.

Γ	Alvo	h	$\%h$	S	$\%S$
<i>RockYou</i>	<i>RockYou</i>	4.936.670	34,43%	18.273.376	58,09%
<i>RockYou</i>	<i>LinkedIn</i>	7.494.658	12,49%	42.181.515	36,70%
<i>RockYou</i>	<i>AntiPublic</i>	17.029.973	8,85%	217.868.495	38,83%
<i>LinkedIn</i>	<i>RockYou</i>	19.319	0,13%	128.826	0,40%
<i>LinkedIn</i>	<i>LinkedIn</i>	43.519	0,07%	29.563	0,02%
<i>LinkedIn</i>	<i>AntiPublic</i>	63.446	0,03%	3.957.835	0,70%
<i>AntiPublic</i>	<i>RockYou</i>	1.250.004	8,71%	9.278.497	29,49%
<i>AntiPublic</i>	<i>LinkedIn</i>	2.456.815	4,09%	19.747.476	17,18%
<i>AntiPublic</i>	<i>AntiPublic</i>	10.020.378	5,20%	127.760.736	22,77%

Tabela 5.8: Experimentos de quebra de *hashes* SHA1 com conjuntos amostrais truncados.

Γ	Alvo	h	$\%h$	S	$\%S$
<i>RockYou_1M</i>	<i>RockYou</i>	4.180.384	29,15%	18.755.420	59,62%
<i>RockYou_1M</i>	<i>LinkedIn</i>	10.857.288	18,09%	51.201.646	44,54%
<i>RockYou_1M</i>	<i>AntiPublic</i>	24.868.569	12,92%	258.029.148	45,99%
<i>LinkedIn_1M</i>	<i>RockYou</i>	3.965.650	27,66%	18.056.129	57,40%
<i>LinkedIn_1M</i>	<i>LinkedIn</i>	11.506.404	19,17%	53.929.283	46,92%
<i>LinkedIn_1M</i>	<i>AntiPublic</i>	24.545.005	12,75%	254.855.321	45,42%
<i>AntiPublic_1M</i>	<i>RockYou</i>	305.720	2,13%	7.611.826	24,20%
<i>AntiPublic_1M</i>	<i>LinkedIn</i>	370.333	0,61%	15.694.084	13,65%
<i>AntiPublic_1M</i>	<i>AntiPublic</i>	488.621	0,25%	87.647.732	15,62%

Tabela 5.9: Experimentos de quebra de *hashes* SHA256.

Γ	Alvo	h	$\%h$	S	$\%S$
<i>RockYou</i>	<i>RockYou</i>	4.983.057	34,75%	18.366.066	58,39%
<i>RockYou</i>	<i>LinkedIn</i>	7.323.122	12,20%	41.805.411	36,37%
<i>RockYou</i>	<i>AntiPublic</i>	18.375.159	9,55%	222.931.995	39,73%
<i>LinkedIn</i>	<i>RockYou</i>	66.237	0,46%	802.763	2,55%
<i>LinkedIn</i>	<i>LinkedIn</i>	94.960	0,15%	394.473	0,34%
<i>LinkedIn</i>	<i>AntiPublic</i>	148.275	0,07%	3.278.458	0,58%
<i>AntiPublic</i>	<i>RockYou</i>	1.320.170	9,20%	9.626.939	30,60%
<i>AntiPublic</i>	<i>LinkedIn</i>	2.679.042	4,46%	21.091.053	18,35%
<i>AntiPublic</i>	<i>AntiPublic</i>	11.702.087	6,08%	139.183.407	24,80%

Tabela 5.10: Experimentos de quebra de *hashes* SHA256 com conjuntos amostrais truncados.

Γ	Alvo	h	$\%h$	S	$\%S$
<i>RockYou_1M</i>	<i>RockYou</i>	4.180.053	29,15%	18.754.979	59,62%
<i>RockYou_1M</i>	<i>LinkedIn</i>	10.850.888	18,08%	51.192.632	44,54%
<i>RockYou_1M</i>	<i>AntiPublic</i>	24.839.515	12,91%	257.954.290	45,97%
<i>LinkedIn_1M</i>	<i>RockYou</i>	3.962.676	27,64%	18.051.536	57,39%
<i>LinkedIn_1M</i>	<i>LinkedIn</i>	11.480.743	19,13%	53.890.454	46,88%
<i>LinkedIn_1M</i>	<i>AntiPublic</i>	24.187.270	12,57%	254.050.325	45,28%
<i>AntiPublic_1M</i>	<i>RockYou</i>	271.950	1,89%	7.210.868	22,92%
<i>AntiPublic_1M</i>	<i>LinkedIn</i>	331.870	0,55%	14.580.465	12,68%
<i>AntiPublic_1M</i>	<i>AntiPublic</i>	429.904	0,22%	81.897.307	14,59%

A divisão (razão) entre os números de *hashes* quebrados é utilizada para comparar a influência do tipo de FDC e do truncamento dos conjuntos amostrais. Por exemplo, calcula-se a razão r entre o número de *hashes* SHA1 pertencentes à lista alvo *RockYou* quebrados pelos modelos do conjunto amostral *RockYou* na sua versão truncada h' pelo número de *hashes* SHA1 pertencentes à lista alvo *RockYou* quebrados pelos modelos do conjunto amostral *RockYou* na sua versão completa h . Assim, tem-se a razão $r = \frac{h'}{h}$.

O truncamento dos conjuntos amostrais apresentou comportamentos distintos para cada conjunto amostral. A média das razões \bar{r} entre os números de *hashes* quebrados pelo conjunto amostral *RockYou* na sua versão truncada pela versão completa foi de 1,23802276. Já a \bar{r} entre os números de *hashes* quebrados pelo conjunto amostral *LinkedIn* na sua versão truncada pela versão completa foi de 200,06450108. Por fim, a \bar{r} entre os números de *hashes* quebrados pelo conjunto amostral *AntiPublic* na sua versão truncada pela versão completa foi de 0,135114154.

- $\bar{r} = \left(\frac{h'(\text{RockYou_1M})}{h(\text{RockYou})} \right) = 1,23802276$
- $\bar{r} = \left(\frac{h'(\text{LinkedIn_1M})}{h(\text{LinkedIn})} \right) = 200,06450108$
- $\bar{r} = \left(\frac{h'(\text{AntiPublic_1M})}{h(\text{AntiPublic})} \right) = 0,135114154$

O tipo de FDC dos *hashes* alvo também teve influências distintas para cada conjunto amostral. A \bar{r} entre os números de *hashes* do tipo SHA256 quebrados pelo conjunto amostral *RockYou* completo pelo número de *hashes* do tipo SHA1 foi de 1,021832656. Já a \bar{r} entre os números de *hashes* do tipo SHA256 quebrados pelo conjunto amostral *RockYou* truncado pelo número de *hashes* do tipo SHA1 foi de 0,999387684.

A \bar{r} entre os números de *hashes* do tipo SHA256 quebrados pelo conjunto amostral

LinkedIn completo pelo número de *hashes* do tipo SHA1 foi de 2,649218603. Já a \bar{r} entre os números de *hashes* do tipo SHA256 quebrados pelo conjunto amostral *LinkedIn* truncado pelo número de *hashes* do tipo SHA1 foi de 0,994148418. A \bar{r} entre os números de *hashes* do tipo SHA256 quebrados pelo conjunto amostral *AntiPublic* completo pelo número de *hashes* do tipo SHA1 foi de 1,104804936. Já a \bar{r} entre os números de *hashes* do tipo SHA256 quebrados pelo conjunto amostral *AntiPublic* truncado pelo número de *hashes* do tipo SHA1 foi de 0,888503355.

- $\bar{r} = \left(\frac{h'(\text{RockYou}, \text{SHA256})}{h(\text{RockYou}, \text{SHA1})} \right) = 1,021832656$
- $\bar{r} = \left(\frac{h'(\text{RockYou_1M}, \text{SHA256})}{h(\text{RockYou_1M}, \text{SHA1})} \right) = 0,999387684$
- $\bar{r} = \left(\frac{h'(\text{LinkedIn}, \text{SHA256})}{h(\text{LinkedIn}, \text{SHA1})} \right) = 2,649218603$
- $\bar{r} = \left(\frac{h'(\text{LinkedIn_1M}, \text{SHA256})}{h(\text{LinkedIn_1M}, \text{SHA1})} \right) = 0,994148418$
- $\bar{r} = \left(\frac{h'(\text{AntiPublic}, \text{SHA256})}{h(\text{AntiPublic}, \text{SHA1})} \right) = 1,104804936$
- $\bar{r} = \left(\frac{h'(\text{AntiPublic_1M}, \text{SHA256})}{h(\text{AntiPublic_1M}, \text{SHA1})} \right) = 0,888503355$

5.6 Limitações do Corinda

A principal limitação da implementação atual do **Corinda** é a ordem em que os palpites são gerados pelo método `composite.Model.recursive()`. O espaço de busca do domínio do modelo composto é varrido de maneira subótima, uma vez que conforme a cardinalidade do domínio dos modelos elementares aumenta, *tokens* populares são penalizados. Por exemplo, sejam os conjuntos dos domínios dos modelos elementares que compõem \hat{m} .

- $\lambda_1 = \{\text{"banana"}, \text{"maçã"}\}$
- $\lambda_2 = \{\text{"1"}, \text{"2"}, \text{"3"}\}$
- $\lambda_3 = \{\text{"rodrigo"}, \text{"igor"}, \text{"regina"}, \text{"marcos"}\}$

A ordem na qual os *tokens* são listados representa a popularidade dos mesmos, ou seja, *"banana"* ocorre com frequência maior que *"maçã"*, *"1"* ocorre com frequência maior que *"3"* e *"rodrigo"* ocorre com frequência maior que *"marcos"*. Idealmente, a *string* *"maçã1rodrigo"* deveria ser gerada antes de *"banana3marcos"*. Contudo, o

algoritmo recursivo produz o produto cartesiano $\tilde{\lambda} = \lambda_1 \times \lambda_2 \times \lambda_3$ como apresentado na Figura 5.7. O código fonte utilizado para a geração deste exemplo encontra-se no Apêndice F.

```
frutas x nums x nomes
banana1rodrigo
banana1igor
banana1regina
banana1marcos
banana2rodrigo
banana2igor
banana2regina
banana2marcos
banana3rodrigo
banana3igor
banana3regina
banana3marcos
maçã1rodrigo
maçã1igor
maçã1regina
maçã1marcos
maçã2rodrigo
maçã2igor
maçã2regina
maçã2marcos
maçã3rodrigo
maçã3igor
maçã3regina
maçã3marcos
|frutas x nums x nomes| = 24
```

Figura 5.7: Saída do método `composite.Model.recursive()`.

5.7 Comentários

Vários experimentos foram realizados para validar o **Corinda**. Os experimentos preliminares apresentaram padrões estatísticos distintos na lista *RockYou*. As frequências relativas dos modelos compostos dos três conjuntos amostrais tendem a assumir a forma da distribuição de Lei de Potência. As entropias dos modelos compostos dos três conjuntos amostrais tendem a assumir a forma da distribuição Normal.

Além da escrita e depuração dos diferentes módulos do software, o maior desafio para a realização dos experimentos foi a preparação dos dados dos conjuntos amostrais para processamento. Como as listas foram obtidas em diferentes fóruns *online*, elas originalmente possuíam diferentes estilos de formatação. Foi necessário formatar as listas de forma padronizada para que o **Corinda** pudesse ler os dados dos conjuntos amostrais.

CAPÍTULO 6

CONCLUSÃO

Este trabalho apresentou nova metodologia de quebra de senhas via heurísticas concorrentes. Os experimentos preliminares indicaram a existência de padrões distintos na distribuição estatística dos modelos de senhas em conjuntos amostrais. Os resultados encontrados sugerem que as distribuições estatísticas dos modelos compostos críticos tendem a seguir a Lei de Potência. Isto significa que pequenos conjuntos de modelos críticos populares são capazes de gerar porções majoritárias das senhas contidas na lista. Caso os domínios destes modelos críticos possuam cardinalidades suficientemente pequenas, elevados números de senhas podem ser facilmente comprometidos, mesmo quando FDC robustas são utilizados.

O software foi treinado a partir de três diferentes conjuntos amostrais: *RockYou*, *LinkedIn*, e *AntiPublic*. As distribuições estatísticas das frequências relativas dos modelos compostos dos três conjuntos amostrais assumem a forma de Lei de Potência. Já as distribuições estatísticas das entropias dos modelos compostos dos três conjuntos amostrais tendem a assumir forma de distribuição Normal.

Quando comparados às suas versões completas, o número de modelos detectados em conjuntos amostrais truncados é inferior, uma vez que há quantidade menor de senhas disponíveis para o processo de treinamento. Além disto, os modelos elementares encontrados nos conjuntos amostrais truncados também possuem menor número de *tokens*, e portanto, geram modelos críticos com domínios de menor cardinalidade durante o processo de treinamento.

Em termos de porcentagem de *hashes* e senhas quebradas (eficiência), o conjunto *RockYou* apresentou os melhores resultados. O truncamento dos conjuntos amostrais apresentou comportamentos distintos para cada conjunto amostral. O conjunto *RockYou* não apresentou diferença significativa de performance entre as versões truncada e completa. Já o conjunto *LinkedIn* na versão truncada apresentou eficiência 200 vezes maior que a versão completa. A versão truncada do conjunto *AntiPublic* apresentou performance significativamente inferior à sua versão completa. Tal fato pode ser explicado pelo tamanho extenso deste conjunto, bem como sua natureza não homogênea, uma vez que este conjunto consiste da compilação de diversas listas com perfis de usuários distintos. A diversidade dos resultados indica a necessidade de investigação adicional para melhor entendimento sobre a relação do tamanho do conjunto amostral e a eficiência dos processos de quebra.

Em termos de segurança da FDC, quanto mais tempo necessário para o cálculo do *hash*, mais segura a FDC, uma vez que o processo de quebra será mais demorado. A eficiência dos processos de quebra de *hashes* do tipo SHA256 foram similares aos processos de quebra de *hashes* do tipo SHA1, independentemente do truncamento ou não dos conjuntos amostrais. Tal comportamento indica que o tempo de cálculo dos *hashes* SHA256 não é significativamente maior do que o tempo de cálculo dos *hashes* SHA1. Investigações adicionais são necessárias para entender como outros tipos de FDC se comportam.

Idealmente, o algoritmo de geração de palpites deve priorizar a combinação de *tokens* populares no cálculo dos produtos cartesianos dos domínios dos modelos elementares. A principal limitação da implementação atual do **Corinda** é a ordem em que os palpites são gerados pelo método `composite.Model.recursive()`. O espaço de busca do domínio do modelo composto é varrido de maneira subótima, uma vez que conforme a cardinalidade do domínio dos modelos elementares aumenta, *tokens* populares são penalizados.

Quando comparado a outros trabalhos, a principal diferença do **Corinda** é a utilização de processos concorrentes na geração dos palpites. O *John The Ripper* utiliza apenas uma linha de execução para realizar a quebra das senhas em CPU, enquanto que o *Hashcat* paraleliza apenas o cálculo dos *hashes* em GPU, contudo com apenas uma linha de execução responsável pela geração dos palpites relativos a apenas um modelo por vez.

6.1 Contribuições do Trabalho

A principal contribuição deste trabalho é a criação de nova metodologia para quebra de senhas baseada em heurísticas concorrentes. Também são apresentados padrões estatísticos de como senhas são formadas em diferentes conjuntos amostrais.

Artigos em congresso:

RODRIGUES, B. A.; PAIVA, J. R. B.; GOMES, V. M.; MORRIS, C.; CALIXTO, W. P. Passfault: an open source tool for measuring password complexity and strength. International Multi-Conference on Complexity, Informatics and Cybernetics, 2017. 25, 29.

6.2 Sugestões para Trabalhos Futuros

1. Incorporar diferentes FDC à funcionalidade do software. Entre exemplos de FDC modernas estão SHA3, *scrypt*, *bcrypt*, e *PBKDF2*. A incorporação de diferentes tipos de FDC permitirá a investigação do impacto de cada uma no tempo de quebra dos *hashes*.
2. Otimizar a performance do software de forma a conceber e implementar novo algoritmo de geração de palpites que priorize a frequência de ocorrência dos diferentes *tokens* durante o cálculo do produto cartesiano dos domínios dos modelos elementares.
3. Realizar novos experimentos para investigação do impacto do tamanho e da natureza dos conjuntos amostrais na eficiência dos processos de quebra.
4. Disponibilizar o **Corinda** em servidor para acesso via *web*. O acesso do software via interface *web* permitirá a popularização do mesmo, fazendo com que o público possa se conscientizar sobre a importância de senhas de alta entropia.

APÊNDICE A

Pacote Modelo Elementar

elementary.go

```
package elementary

import (
    "math"
    "sort"
)

// this struct represents an Elementary Model
// a map[string]ElementaryModel is later saved into a gob file
type Model struct {
    Name          string
    Entropy       float64
    TokenFreqs    map[string]int
}

type TokenFreq struct {
    Token string
    Freq  int
}

func (em *Model) UpdateEntropy(){

    // sum for normalization (frequencies to probabilities)
    sum := 0
    for _, freq := range em.TokenFreqs {
        sum += freq
    }

    // entropy calculation
    entropy := float64(0)
    for _, freq := range em.TokenFreqs {
        f := freq
        p := float64(f)/float64(sum)
        e := -p*math.Log10(p)
        entropy += e
    }

    em.Entropy = entropy
}
```



```

// updates the frequency of a token in some elementary.Model
func (em *Model) UpdateTokenFreq(freq int, token string){

    //token already in TokenFreqs?
    if _, ok := em.TokenFreqs[token]; ok{
        f := em.TokenFreqs[token]
        em.TokenFreqs[token] = freq + f
    }else{
        em.TokenFreqs[token] = freq
    }
}

// temporary structure used to sort tokens
type tokenSort struct{
    tokenFreqs map[string]int
    tokenSlice []string
}

//returns a sorted slice with tokens in descending frequency order
func (em *Model) SortedTokens() []string{
    tokenSlice := make([]string, 0)

    for token, _ := range em.TokenFreqs{
        tokenSlice = append(tokenSlice, token)
    }

    tokenSort := tokenSort{em.TokenFreqs, tokenSlice}
    tokenSort.Sort()

    return tokenSort.tokenSlice
}

// sorts TokenFreqs in descendent order
func (ts *tokenSort) Sort(){
    sort.Sort(ts)
}

// We implement 'sort.Interface' - 'Len', 'Less', and
// 'Swap' - on our type so we can use the 'sort' package's
// generic 'Sort' function. 'Len' and 'Swap'
// will usually be similar across types and 'Less' will
// hold the actual custom sorting logic.
func (ts *tokenSort) Len() int {
    return len(ts.tokenFreqs)
}

```

```
func (ts *tokenSort) Swap(i, j int) {  
    ts.tokenSlice[i], ts.tokenSlice[j] =  
        ts.tokenSlice[j], ts.tokenSlice[i]  
}  
func (ts *tokenSort) Less(i, j int) bool {  
    tokenI := ts.tokenSlice[i]  
    tokenJ := ts.tokenSlice[j]  
  
    return ts.tokenFreqs[tokenI] > ts.tokenFreqs[tokenJ]  
}
```


APÊNDICE B

Pacote Modelo Composto

composite.go

```
package composite

import (
    "github.com/bernardoaraujor/corinda/elementary"
)

// this struct represents a Composite Model
// a map[string]CompositeModel is later saved into a gob file
type Model struct{
    Name          string
    Freq          int
    Prob          float64
    Entropy       float64
    Models        []string
}

// gets Probability from normalized Frequency
func (cm *Model) UpdateProb(freqSum int){
    cm.Prob = float64(cm.Freq)/float64(freqSum)
}

// updates the frequency of the Composite Model
func (cm *Model) UpdateFreq(freq int){
    cm.Freq = cm.Freq + freq
}

// updates the total entropy of the Composite Model
func (cm *Model) UpdateEntropy(elementaries
                                map[string]*elementary.Model){
    entropy := float64(0)

    for _, em := range cm.Models {
        entropy += elementaries[em].Entropy
    }

    cm.Entropy = entropy
}
```

```
// returns channel with password guesses belonging to the
```

```

//cartesian product between the Composite Model's Elementary Models
func (cm *Model) Guess(tokenLists [][]string) chan string{
    out := make(chan string)

    go cm.recursive(0, nil, nil, out, tokenLists)

    return out
}

// sends elements of the cartesian product of TokensNFreqs of all
//cm's ems to out channel
func (cm *Model) recursive(depth int, counters []int,
    lengths []int, out chan string, tokenLists [][]string){

    // max depth to be processed recursively
    n := len(cm.Models)

    // first depth level of recursion. init counters and lengths
    if depth == 0{

        // init counters (all 0)
        counters = make([]int, n)

        // init lengths
        lengths = make([]int, n)
        for i, _ := range cm.Models {
            //emName := cm.Models[i]
            lengths[i] = len(tokenLists[i])
        }
    }

    // last depth level of recursion
    if depth == n{
        result := ""
        for d := 0; d < n; d++){
            i := counters[d]

            result += tokenLists[d][i]
        }

        // send result to out channel
        out <- result
    }

    // any other depth that isn't the last
}else{

```

```

        // go through current depth
        for counters[depth] = 0; counters[depth] <
            lengths[depth]; counters[depth]++){

            // recursively process next depth
            cm.recursive(depth+1, counters,
                lengths, out, tokenLists)

        }
    }
    // time to close the channel?
    // analyzes counters of all EMs... if all are
    // equal to the respective lengths,
    // then every element of the cartesian product
    // have been calculated, and the channel can be closed
    closer := true
    for i := 0; i < n; i++){
        if counters[i] != lengths[i]{
            closer = false
        }
    }

    // close channel
    if closer{
        close(out)
    }
}

```


APÊNDICE C

Pacote de Treinamento

train.go

```
package train

import (
    "runtime"
    "fmt"
    "os"
    "github.com/bernardoaraujor/corinda/elementary"
    "github.com/bernardoaraujor/corinda/composite"
    "compress/gzip"
    "encoding/csv"
    "strconv"
    "github.com/timob/jnigi"
    "time"
    "encoding/json"
)

const passfaultClassPath = "-Djava.class.path=passfault_corinda/out\n/artifacts/passfault_corinda_jar/passfault_corinda.jar"
const bufSize = 10000000

type input struct {
    freq int
    pass string
}

type inputBatch []input

type result struct {
    freq    int
    result []byte
}

type resultBatch []result

type trainedMaps struct {
    elementaries map[string]*elementary.Model
    composites   map[string]*composite.Model
}

// used only for parsing JSON into elementary.Model
```



```

type elementaryJSON struct{
    Name    string `json:"modelName"`
    Index   int    `json:"modelIndex"`
    Token    string `json:"token"`
}

// used only for parsing JSON into composite.Model
type compositeJSON struct {
    Models []elementaryJSON `json:"elementaryModels"`
    CompositeModelName string `json:"compositeModelName"`
}

// checks for error
func check(e error) {
    if e != nil {
        _, file, line, _ := runtime.Caller(1)
        fmt.Println(line, "\t", file, "\n", e)
        os.Exit(1)
    }
}

func countLines(list string) int{
    path := "csv/" + list + ".csv.gz"

    f, err := os.Open(path)
    check(err)
    defer f.Close()

    gr, err := gzip.NewReader(f)
    check(err)
    defer gr.Close()

    cr := csv.NewReader(gr)

    //fmt.Println("Counting lines in list...")
    listSize := 0
    for records, err := cr.Read(); records != nil;
        records, err = cr.Read(){
        check(err)
        listSize++
    }

    return listSize
}

```

```
func generator(list string, batchSize int) (int, chan inputBatch){
    listSize := countLines(list)
```

```
    out := make(chan inputBatch, bufSize)

    path := "csv/" + list + ".csv.gz"
    f, err := os.Open(path)
    check(err)

    gr, err := gzip.NewReader(f)
    check(err)

    cr := csv.NewReader(gr)

    go func(){
        defer f.Close()
        defer gr.Close()
        end := false

        for{
            ib := make([]input, 0)

            for i := 0; i < batchSize; i++){
                row, _ := cr.Read()

                if row != nil{
                    freq, err :=
                        strconv.Atoi(row[0])
                    pass := row[1]
                    check(err)

                    input := input{freq, pass}
                    ib = append(ib, input)
                }else{ //end of list
                    end = true
                    break
                }
            }

            out <- ib
            if end{
                close(out)
                return
            }
        }
    }()
```

```

    }
    }()

    return listSize, out
}

```

```

func batchAnalyzer(c *int, ibChan chan inputBatch) chan resultBatch{
    out := make(chan resultBatch, bufSize)

    go func(){

        jvm, _, err := jnigi.CreateJVM(jnigi.NewJVMInitArgs(
            false, true, jnigi.DEFAULT_VERSION,
            []string{passfaultClassPath}))

        check(err)

        for ib := range ibChan{

            // attach this routine to JVM
            env := jvm.AttachCurrentThread()

            obj, err := env.NewObject(
                "org/owasp/passfault/TextAnalysis")
            check(err)

            rb := resultBatch{}

            // range over inputBatch
            for _, input := range ib{
                freq := input.freq
                pass := input.pass

                // filter out long weird passwords
                if len(pass) < 30{
                    str, err := env.NewObject(
                        "java/lang/String",
                        []byte(pass))
                    check(err)

                    // call passwordAnalysis
                    // on password
                    v, err := obj.CallMethod(
                        env, "passwordAnalysis",
                        "java/lang/String", str)

```

```

        check(err)

        //format result from JVM
        //into byte array (probably
        //not the most elegant way)

        resultJVM, err := v.(
            *jnigi.ObjectRef).
            CallMethod(

env, "getBytes",

            jnigi.Byte|jnigi.Array)

        resultString := string(
            resultJVM.([]byte))

        resultBytes := []byte(
            resultString)

        rb = append(rb,result{freq,
            resultBytes})

        // increment counter
        *c++

        //env.DeleteGlobalRef(str)
        env.DeleteLocalRef(str)
    }
}

out <- rb

jvm.DetachCurrentThread()
}

close(out)
}()

return out
}

func reporter(m *int, c *int, total int, done bool){
    start := time.Now()

    lastCount := 0

```

```

// report progress every second
for !done{
    time.Sleep(1000 * time.Millisecond)

    since := time.Since(start)
    speed := *c - lastCount

    progress := float64(*c*100)/float64(total)

    fmt.Println("Maps merged: " + strconv.Itoa(*m) +
"; Speed: " + strconv.Itoa(speed)+" P/s; Progress:"
+ strconv.FormatFloat(progress, 'f', 2, 64) +
" %; Processed passwords: " + strconv.Itoa(*c) +
"; Total time: " + since.String())

    lastCount = *c
}
}

func batchDecoder(rbChan chan resultBatch) chan trainedMaps{
    tmChan := make(chan trainedMaps, bufSize)

    go func(){
        for rb := range rbChan{
            tm := trainedMaps{make(
                map[string]*elementary.Model), make(map
                [string]*composite.Model))

            for _, r := range rb{
                freq := r.freq
                result := r.result

                // parse JSON
                var cmFromJSON compositeJSON
                json.Unmarshal(result, &cmFromJSON)

                // update elementaryModel map
                for _, emFromJSON := range
cmFromJSON.Models {
                    //elementary.Model already
//in tm only update frequency

                    if emFromMap, ok :=
tm.elementaries[
emFromJSON.Name]; ok {

```

```

emFromMap.UpdateTokenFreq(
    freq, emFromJSON.Token)
// elementary.Model not in
//map, create new instance and insert into the map
}else{
    t :=
    emFromJSON.Token
    f := freq

```

```

tokenFreqs := make(
    map[string]int)
tokenFreqs[t] = f

newEM :=
    elementary.Model{
        emFromJSON.Name,
        0, tokenFreqs}

tm.elementaries[
    emFromJSON.Name] =
    &newEM
}

}

//composite.Model already in map,
// only update frequency
if cmFromMap, ok := tm.composites[
    cmFromJSON.CompositeModelName]; ok{
    cmFromMap.UpdateFreq(freq)
// composite.Model not in map,
// create new instance and insert into the map
}else{
    compModelName :=
        cmFromJSON.CompositeModelName

    elementaryModels :=
        make([]string, 0)
    for _, emFromJSON :=
        range cmFromJSON.Models {
        elementaryModels =
            append(
                elementaryModels,
                emFromJSON.Name)
    }

```

```

        // new Composite Model
        cm := composite.Model{
            compModelName, freq, 0,
            0, elementaryModels}

        // add to map
        tm.composites[compModelName] =
            &cm
    }
}

```

```

        tmChan <- tm
    }

    close(tmChan)
}()

return tmChan
}

func mapsMerger(m *int, tmChan chan trainedMaps) trainedMaps{
    finalMaps := trainedMaps{make(map[string]*elementary.Model),
        make(map[string]*composite.Model)}

    for tm := range tmChan{

        // process tm.elementaries
        for k, emFrom := range tm.elementaries {

            //emFrom already in finalMaps.elementaries
            if emTo, ok := finalMaps.elementaries[k]; ok{

                //verify every TokenFreq in emFrom
                for token, freq :=
                    range emFrom.TokenFreqs{

                    // is t already in
                    //emTo.TokenFreqs??

                    if _, ok :=
                        emTo.TokenFreqs[token]; ok{
                        f :=
                            emTo.TokenFreqs[token]

                            emTo.TokenFreqs[token] =
                                f + freq
                    }
                }
            }
        }
    }
}

```

```

                                }else{
                                    emTo.TokenFreqs[token] =
                                        freq
                                }
                            }
                        //emFrom not in finalMaps.elementaries,
// create new entry
                    }else{
                        finalMaps.elementaries[k] = emFrom
                    }
                }
            }

```

```

// process composite.Models
for k, cmFrom := range tm.composites {

    //cm already in finalMaps.composites
    if cmTo, ok := finalMaps.composites[k]; ok{
        cmTo.UpdateFreq(cmFrom.Freq)

    //cm not in finalMaps.composites
// create new entry
    }else{

        //insert cm
        finalMaps.composites[k] = cmFrom
    }

}

*m++

}

for _, em := range finalMaps.elementaries{
    em.UpdateEntropy()
}

sum := 0
for _, cm := range finalMaps.composites{
    sum += cm.Freq
}

for _, cm := range finalMaps.composites{
    cm.UpdateProb(sum)
    cm.UpdateEntropy(finalMaps.elementaries)
}

```



```

        return finalMaps
    }

func saveMaps(finalMaps trainedMaps, list string){
    fmt.Println("Saving maps...")
    emArray := make([]elementary.Model,
        len(finalMaps.elementaries))

    i := 0
    for _, em := range finalMaps.elementaries{
        emArray[i] = *em
        i++
    }
}

```

```

    jsonData, err := json.Marshal(emArray)

    emFile, err := os.Create("maps/" + list +
        "Elementaries.json")
    check(err)
    defer emFile.Close()

    emFile.Write(jsonData)
    emFile.Sync()

    cmArray := make([]composite.Model,
        len(finalMaps.composites))
    i = 0
    for _, cm := range finalMaps.composites{

        // ignore improbable cms
        if cm.Prob > 0.00001{
            cmArray[i] = *cm
            i++
        }
    }

    jsonData, err = json.Marshal(cmArray)

    cmFile, err := os.Create("maps/" + list + "Composites.json")
    check(err)
    defer cmFile.Close()

    cmFile.Write(jsonData)
    cmFile.Sync()
}

```

```
func Train(list string){
    total, inputBatches := generator(list, 1000000)
    c := 0
    m := 0
    resultBatches := batchAnalyzer(&c, inputBatches)
    go reporter(&m, &c, total, false)
    trainedMaps := batchDecoder(resultBatches)
    finalMaps := mapsMerger(&m, trainedMaps)
    saveMaps(finalMaps, list)
    os.Exit(0)
}
```


APÊNDICE D

Pacote de Quebra de Senhas

crack.go

```
package crack

import (
    "crypto/sha256"
    "crypto/sha1"
    "runtime"
    "hash"
    "sync"
    "fmt"
    "os"
    "io/ioutil"
    "encoding/hex"
    "github.com/bernardoaraujo/corinda/composite"
    "github.com/bernardoaraujo/corinda/elementary"
    "encoding/json"
    "compress/gzip"
    "encoding/csv"
    "math"
    "time"
    "strconv"
)

const Rockyou = "rockyou"
const Linkedin = "linkedin"
const Antipublic = "antipublic"

const SHA1 = "sha1"
const SHA256 = "sha256"

const minProb = 0.00015

type targetsMap struct{
    sync.RWMutex
    targets map[string]string
}

type password struct {
    pass string
    hash []byte
}
```

```

type Crack struct {
    alg                string
    composites         map[string]*composite.Model
    elementaries       map[string]*elementary.Model
    targetsMap         targetsMap
    targetName         string
    trainedName        string
    durationH          float64
}

// crack session
func (crack Crack) Crack(){
    composites := crack.composites
    elementaries := crack.elementaries

    var wg sync.WaitGroup

    // initialize channels
    fmt.Println("Initializing Guess Channels")

    guesses := make([]chan string, 0)
    nGuesses := make([]int, 0)
    for _, cm := range composites{
        if cm.Prob > minProb{
            tokenLists := make([][]string, 0)
            for _, elementaryName := range cm.Models{
                elementary :=
                    elementaries[elementaryName]

                tokenLists = append(
                    tokenLists, elementary.SortedTokens())
            }

            guesses = append(
                guesses, cm.Guess(tokenLists))

            // n = p*10^E
            n := int(cm.Prob * math.Pow(10, cm.Entropy))
            nGuesses = append(nGuesses, n)
        }
    }

    fmt.Println(nGuesses)
    guessLoop := crack.guessLoop(guesses, nGuesses)

```

```

passwords := crack.digest(guessLoop)

wg.Add(1)
results := crack.searcher(passwords)
count := 0

go monitor(wg, crack.durationH)
go reporter(&count)
go save(crack.trainedName, crack.targetName, crack.alg,
    results, wg, &count)

fmt.Println("Cracking...")

wg.Wait()
}

// Constructor
func NewCrack(trained string, target string, alg string,
    durationH float64) Crack{
    var crack Crack

    crack.targetName = target
    crack.trainedName = trained
    crack.alg = alg
    crack.durationH = durationH

    f, err := os.Open("maps/" + trained + "Elementaries.json.gz")
    check(err)

    gr, err := gzip.NewReader(f)
    check(err)

    raw, err := ioutil.ReadAll(gr)
    check(err)
    var elementaries []*elementary.Model
    err = json.Unmarshal(raw, &elementaries)
    check(err)
    crack.elementaries = make(map[string]*elementary.Model)

    for _, em := range elementaries{
        crack.elementaries[em.Name] = em
    }

    f, err = os.Open("maps/" + trained + "Composites.json.gz")
    check(err)

```

```

gr, err = gzip.NewReader(f)
check(err)

raw, err = ioutil.ReadAll(gr)
check(err)
var composites []*composite.Model
err = json.Unmarshal(raw, &composites)

crack.composites = make(map[string]*composite.Model)

for _, cm := range composites{
    crack.composites[cm.Name] = cm
}

f, err = os.Open("targets/" + alg + "/"
    + target + ".csv.gz")
check(err)
defer f.Close()

gr, err = gzip.NewReader(f)
check(err)
defer gr.Close()

cr := csv.NewReader(gr)

fmt.Println("Loading target list...")
crack.targetsMap.targets = make(map[string]string)
for records, err := cr.Read(); records != nil;
    records, err = cr.Read(){
    check(err)

```

```

        hash := records[0]
        crack.targetsMap.targets[hash] = hash
    }

    return crack
}

// Decode Gob file
func load(path string, object interface{}) error {
    file, err := os.Open(path)
    if err == nil {
        decoder := json.NewDecoder(file)
        err = decoder.Decode(object)
    }
}

```

```

        file.Close()
        return err
    }

    // checks for error
    func check(e error) {
        if e != nil {
            _, file, line, _ := runtime.Caller(1)
            fmt.Println(line, "\t", file, "\n", e)
            os.Exit(1)
        }
    }

    // returns channel with hashes in string format
    func (crack Crack) digest(in chan string) chan password {
        out := make(chan password)

        go func(out chan password) {
            defer close(out)
            // reads in channel
            for guess := range in {
                // temporary conditional... avoiding weird
                // bug that receives empty guess
                if guess != "" {
                    // digest
                    var hasher hash.Hash
                    switch crack.alg {
                    case SHA1:
                        hasher = sha1.New()
                    case SHA256:
                        hasher = sha256.New()
                }

```

```

                    hasher.Write([]byte(guess))
                    digest := hasher.Sum(nil)

                    // spits out digest
                    out <- password{guess, digest}
                }
            }

        }(out)

        return out
    }
}

```



```

// merge the flux from channels cs into out
func fanIn(cs []chan password) chan password {
    var wg sync.WaitGroup

    out := make(chan password)

    // Start an output goroutine for each input channel in cs.
output
    // copies values from c to out until c is closed
    // then calls wg.Done.
    output := func(c <-chan password) {
        for n := range c {
            out <- n
        }
        wg.Done()
    }

    // prepares wait group for the number of input channels
    wg.Add(len(cs))

    // start goroutines
    for _, c := range cs {
        go output(c)
    }

    // start goroutine that closes output channel when all
    // input channels have been closed
    go func() {
        wg.Wait()
        close(out)
    }()
}

```

```

    // returns output channel
    return out
}

// loops over guess channels generating guesses for digest
func (crack Crack) guessLoop(guessChans []chan string,
    ns []int) chan string {
    out := make(chan string)

    go func(guessChans []chan string, ns []int){
        for {
            for i, guessChan := range guessChans{

```

```

                                nGuesses := ns[i]

                                for j := 0; j < nGuesses; j++{
                                    guess := <- guessChan

                                    out <- guess
                                }
                            }
                        }(guessChans, ns)

                    return out
                }

func (crack *Crack) searcher(in chan password) chan password {
    out := make(chan password)

    go func(){
        for password := range in{
            //pass := ph.pass
            hash := hex.EncodeToString(password.hash)

            if _, ok :=
                crack.targetsMap.targets[hash]; ok{
                out <- password

                crack.targetsMap.Lock()
                delete(
                    crack.targetsMap.targets, hash)
                crack.targetsMap.Unlock()
            }
        }
    }()
}

```

```

        return out
    }

func save(trained string, target string, alg string,
    in chan password, wg sync.WaitGroup, c *int){
    defer wg.Done()

    resultFile, err := os.Create("results/" +
        trained + "_" + target + "_" + alg + ".csv")
    check(err)
    defer resultFile.Close()
}

```

```

        for ph := range in{
            pass := ph.pass
            hash := ph.hash

            line := pass + "," + hex.EncodeToString(hash)
            *c++

            fmt.Fprintln(resultFile, line)
        }
    }

func reporter(c *int){
    lastCount := 0
    // report progress every second
    for {
        time.Sleep(1000 * time.Millisecond)

        speed := *c - lastCount

        fmt.Println("Speed: " + strconv.Itoa(speed) +
            " P/s; Found Passwords: " + strconv.Itoa(*c))
        lastCount = *c
    }
}

func monitor(wg sync.WaitGroup, durationH float64){
    start := time.Now()

    for{
        since := time.Since(start)
        if since.Hours() > durationH{
            wg.Done()
            os.Exit(0)
        }
    }
}

```

APÊNDICE E

Pacote de Interface de Comandos

crack.go

```
package cmd

import (
    "fmt"

    "github.com/spf13/cobra"
)

// crackCmd represents the crack command

//Usage: corinda crack <trained> <target>
//
//example: corinda crack rockyou linkedin
var crackCmd = &cobra.Command{
    Use:     "crack",
    Short:   "Use trained maps to crack target list",
    Long:    "",
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("crack called")
    },
}

func init() {
    rootCmd.AddCommand(crackCmd)
}
```

root.go

```
package cmd

import (
    "fmt"
    "os"

    homedir "github.com/mitchellh/go-homedir"
    "github.com/spf13/cobra"
    "github.com/spf13/viper"
)

var cfgFile string
```



```

    viper.AutomaticEnv()

    // If a config file is found, read it in.
    if err := viper.ReadInConfig(); err == nil {
        fmt.Println("Using config file:",
            viper.ConfigFileUsed())
    }
}

```

train.go

```

package cmd

import (
    "os"

    "github.com/spf13/cobra"
    "github.com/bernardoaraujor/corinda/train"
    "fmt"
)

// trainCmd represents the train command
var trainCmd = &cobra.Command{
    Use:     "train",
    Short:   "Train models from csv list",
    Long:    ``,
    Run: func(cmd *cobra.Command, args []string) {
        if len(os.Args) <= 2 {
            fmt.Println("error: which input list should I use?")
            os.Exit(1)
        }

        list := os.Args[2]

        fmt.Println("Starting the training process based on "
            + list + ".csv")
        train.Train(list)
    },
}

func init() {
    rootCmd.AddCommand(trainCmd)
}

```


APÊNDICE F

Exemplo de Produto Cartesiano

produtoCartesiano.go

```
package main

import (
    "fmt"
    "strconv"
)

func main() {

    // inicializa as listas de tokens
    var frutas = []string{"banana", "maca", "uva"}
    var nums = []string{"1", "2", "3", "4"}
    var nomes =
        []string{"wesley", "rodrigo", "igor", "regina", "marcos"}

    fmt.Println(
        "-----frutas x nums-----")
    total := 0

    // drena canal com elementos do produto cartesiano
    // dos arrays em arrays1
    for s := range produtoCartesiano(frutas, nums){
        total++
        fmt.Println(s)
    }
    fmt.Println("|frutas x nums| = " + strconv.Itoa(total))

    fmt.Println(
        "-----frutas x nums x nomes-----")
    total = 0

    // drena canal com elementos do produto cartesiano
    for s := range produtoCartesiano(frutas, nums, nomes){
        total++
        fmt.Println(s)
    }
    fmt.Println("|frutas x nums x nomes| = "
        + strconv.Itoa(total))
}
```



```

}

//gera canal com fluxo de elementos do produto cartesiano dos arrays
func produtoCartesiano(arrays ...[]string) chan string{
    // inicializa canal de saida
    saida := make(chan string)

    // lanca gorrotina recursiva
    go recursao(arrays, 0, nil, nil, saida)

    // retorna canal de saida
    return saida
}

// lanca os valores do produto cartesiano
// dos arrays no canal de saida
func recursao(arrays [][]string, profundidade int,
    contadores []int, tamanhos []int, saida chan string){
    // profundidade maxima a ser processada recursivamente
    n := len(arrays)

    // primeiro nivel de recursao...
    // inicializa contadores e tamanhos
    if profundidade == 0{

        // inicializa contadores (todos 0)
        contadores = make([]int, n)

        // inicializa tamanhos
        tamanhos = make([]int, n)
        for i, _ := range arrays{
            tamanhos[i] = len(arrays[i])
        }
    }

    // ultimo nivel de profundidade na recursao
    if profundidade == n{
        resultado := ""
        for p := 0; p < n; p++){
            i := contadores[p]
            resultado += arrays[p][i]
        }
    }
}

```

```

        // envia elemento no canal de saida
        saida <- resultado
    // qualquer outra profundidade que nao seja a ultima
}else{
    // varre array da profundidade atual
    for contadores[profundidade] = 0;
        contadores[profundidade] < tamanhos[profundidade];
        contadores[profundidade]++{
        // processa proxima profundidade recursivamente
        recursao(arrays, profundidade+1, contadores,
            tamanhos, saida)
    }
}

// hora de fechar canal?
// analisa contadores de todos arrays... se todos forem
// iguais aos respectivos tamanhos,
// entao todos elementos do produto cartesiano foram
// calculados, e o canal pode ser fechado
fecha := true
for i := 0; i < n; i++){
    if contadores[i] != tamanhos[i]{
        fecha = false
    }
}

// fecha canal
if fecha{
    close(saida)
}
}

```


APÊNDICE G

Exemplo de Parametrizacao de Carga Computacional

cargaComputacional.go

```
/*
Exemplo de como o padrao de concorrencia funil, associado
a utilizacao de iteracoes mestras com gorrotinas de vida
limitada, podem ser utilizados para distribuir a carga
computacional ao processamento de diferentes canais geradores.
*/
package main

import (
    "sync"
    "encoding/hex"
    "fmt"
    "crypto/sha1"
)

func main() {
    // inicializa canais geradores
    chA := gerador("a")
    chB := gerador("b")
    chC := gerador("c")

    // delega o fechamento dos canais a funcao main
    defer close(chA)
    defer close(chB)
    defer close(chC)

    // inicializa contadores
    a := 0
    b := 0
    c := 0

    // gera array de canais geradores
    geradores := []chan string{chA, chB, chC}

    //gera array de indices
    ns := []int{1, 10, 100}

    // inicializa canal de lotes de hashes
    lotes := loteHashes(geradores, ns)
```

```

// k lotes de hashes
k := 100
for i := 0; i < k; i++){

    // recebe lote do canal lotes
    lote := <-lotes

    // processa lote
    for _, byte := range lote{
        // converte o hash de hex para string
        s := hex.EncodeToString(byte)

        // incrementa contador correspondente
        // ao valor lido
        switch s{
            case "86f7e437faa5a7fce15d1ddcb9eaeaea377667b8":
                a++
            case "e9d71f5ee7c92d6dc9e92ffdad17b8bd49418f98":
                b++
            case "84a516841ba77a5b4648de2cd0dfcb30ea46dbb4":
                c++
        }
    }
}

// imprime os resultados
fmt.Println(a)
fmt.Println(b)
fmt.Println(c)
}

// gera canal com fluxo continuo de strings identicas a s
func gerador(s string) chan string{

    // inicializa canal de saida
    out := make(chan string)

    // lanca gorrotina que envia copias de s ao canal de saida
    // indefinidamente, ate que este canal seja fechado
    go func(){
        for {
            out <- s
        }
    }()
}

```

```

        // retorna o canal de saida
        return out
    }

    // drena o canal in por n iteracoes
    func hash(entrada chan string, n int) chan []byte{

        // inicializa canal de saida
        out := make(chan []byte)

        // lanca gorrotina que repete por n iteracoes
        // o fechamento do canal de saida e delegado ao encerramento
        // da gorrotina (defer), o que acontece apos a execucao
        // das n iteracoes
        go func(n int, saida chan []byte){
            defer close(saida)
            for i := 0; i < n; i++ {
                // le o canal de entrada
                s := <-entrada

                // calcula o hash
                hasher := sha1.New()
                hasher.Write([]byte(s))
                hashBytes := hasher.Sum(nil)

                // envia o hash no canal de saida
                saida <- hashBytes
            }
        }(n, out)

        // retorna o canal de saida
        return out
    }

    // funde o fluxo dos canais cs no canal out
    func funil(cs []chan []byte) chan []byte {

        // declara o grupo de espera wg
        var wg sync.WaitGroup

        // inicializa canal de saida
        saida := make(chan []byte)

        // inicializa gorrotina de saida para cada canal de entrada
        // em cs. A gorrotina e responsavel por enviar para a saida

```

```

// copias dos valores drenados de c ate que c seja fechado,
// ate por fim chamar wg.Done
output := func(c <-chan []byte) {
    for n := range c {
        saida <- n
    }
    wg.Done()
}

// prepara o grupo de espera para o numero de gorrotinas
// a serem lancadas
wg.Add(len(cs))

// lança gorrotinas
for _, c := range cs {
    go output(c)
}

// lança gorrotina para fechar o canal de saída uma vez que
// todas gorrotinas de saidas estao finalizadas
go func() {
    wg.Wait()
    close(saida)
}()

// retorna canal de saida
return saida
}

// calcula lote de hashes a partir de um array de canais de entrada
func loteHashes(entradas []chan string, ns []int) chan [][]byte{

    // inicializa canal de saida
    saida := make(chan [][]byte)

    go func(entradas []chan string, ns []int){

        // processa lotes indefinidamente
        for {
            // gera array de canais de calculo de hashes
            // a serem utilizados como entrada para o funil
            hashes := make([]chan []byte, 0)
            for i, entrada := range entradas{
                n := ns[i]

```

```

        hashes = append(hashes, hash(entrada, n))
    }

    // gera canal funil para drenar canais
    // de processamento
    funil := funil(hashes)

    // inicializa lote de bytes
    lote := make([][]byte, 0)

    // drena canal funil
    for byte := range funil{
        lote = append(lote, byte)
    }

    // envia lote no canal de saida
    saida <- lote
    }
}(entradas, ns)

return saida
}

```

A execucao do codigo acima gera o resultado:

```

100
1000
10000

```


REFERÊNCIAS BIBLIOGRÁFICAS

- AMICO, M. D.; MICHIARDI, P.; ROUDIER, Y. Password strength: An empirical analysis. **2010 Proceedings IEEE INFOCOM**, 2010. [25](#)
- BINNIE, C. Password cracking with hashcat. **Linux Server Security**, p. 99 – 112, 2016. [26](#), [30](#)
- BISHOP, M.; KLEIN, D. V. Improving system security via proactive password checking. **Computers & Security**, v. 14, n. 3, p. 233 – 249, 1995. [25](#), [26](#)
- BURR, W. E.; DODSON, D. F.; NEWTON, E. M.; PERLNER, R. A.; POLK, W. T.; GUPTA, S.; EVANS, D. L.; NABBUS, E. A. Electronic authentication guideline. In: **NIST Special Publication No. 800-63-2**. [S.l.: s.n.], 2013. [26](#)
- BURR, W. E.; DODSON, D. F.; POLK, W. T.; EVANS, D. L. Electronic authentication guideline. In: **NIST Special Publication No. 800-63-1**. [S.l.: s.n.], 2004. [26](#)
- CASELLA, G.; BERGER, R. L. **Statistical inference**. [S.l.]: China Machine Press, 2010. [40](#)
- CASTELLUCCIA, C.; DURMUTH, M.; PERITO, D. Adaptive password-strength meters from markov models. **Proc. NDSS**, 2016. [26](#)
- CHANG, C. C.; KEISLER, H. J. **Model theory**. [S.l.]: Dover Publications, 2012. [41](#), [42](#), [45](#)
- CISAR, P.; CISAR, S. M. Password - a form of authentication. **2007 5th International Symposium on Intelligent Systems and Informatics**, 2007. [25](#), [29](#)
- CONGER, K.; LYNLEY, M. Dropbox employee's password reuse led to theft of 60m+ user credentials. **TechCrunch**, 2017. [25](#)
- CUBRILOVIC, N. Rockyou hack: From bad to worse. **TechCrunch**, 2009. [32](#)
- DANG, Q. Changes in federal information processing standard (fips) 180-4, secure hash standard. **Cryptologia**, v. 37, n. 1, p. 69–73, 2013. [62](#)
- DONOVAN, A. A. A.; KERNIGHAN, B. W. **The Go programming language**. [S.l.]: Addison-Wesley, 2015. [33](#)

- FERGUSON, N.; SCHNEIER, B.; KOHNO, T. **Cryptographysykes engineering: design principles and practical applicationns**. [S.l.]: Wiley, 2010. [29](#), [43](#)
- FLORENCIO, D.; HERLEY, C. A large-scale study of web password habits. **Proceedings of the 16th international conference on World Wide Web - WWW 07**, 2007. [25](#)
- FRANCESCHI-BICCHIERAI, L. Another day, another hack: 117 million linkedin emails and passwords. **Mortherboard**, 2016. [32](#)
- GAW, S.; FELTEN, E. W. Password management strategies for online accounts. **Proceedings of the second symposium on Usable privacy and security - SOUPS 06**, 2006. [25](#)
- GLABB, R.; IMBERT, L.; JULLIEN, G.; TISSERAND, A.; VEYRAT-CHARVILLON, N. Multi-mode operator for sha-2 hash functions. **Journal of Systems Architecture**, v. 53, n. 2-3, p. 127–138, 2007. [63](#)
- GOSNEY, J. M. How linkedin’s password sloppiness hurts us all. **Ars Technica**, 2016. [26](#)
- HALMOS, P. R. **NAIVE SET THEORY**. [S.l.]: STELLAR CLASSICS, 2017. [39](#)
- HUNT, T. Password reuse, credential stuffing and another billion records in have i been pwned. **Troy Hunt**, 2017. [32](#), [33](#)
- HUTCHENS, J. **Kali Linux network scanning cookbook: over 90 hands-on recipes explaining how to leverage custom scripts and integrated tools in Kali Linux to effectively master network scanning**. [S.l.]: Packt Publishing, 2014. [25](#), [29](#)
- JAYAMAHA, R. G. M. M.; SENADHEERA, M. R. R.; GAMAGE, T. N. C.; WEERASEKARA, K. D. P. B.; DISSANAYAKA, G. A.; KODAGODA, G. N. Voizlock - human voice authentication system using hidden markov model. **2008 4th International Conference on Information and Automation for Sustainability**, 2008. [25](#)
- KELLEY, P.; KOM, S.; MAZUREK, M. L.; SHAY, R.; VIDAS, T.; BAUER, L.; CHRISTIN, N.; CRANOR, L. F.; LÓPEZ, J. **Guess Again (and again and again): Measuring Password Strength by Simulating Password-Cracking Algorithms**. 2011. [26](#)

- MACKAY, D. J. C. Information theory, inference, and learning algorithms. **IEEE Transactions on Information Theory**, v. 50, n. 10, p. 2544–2545, 2004. [42](#)
- MENEZES, A. J.; C., V. O. P.; VANSTONE, S. A. **Handbook of applied cryptography**. [S.l.]: CRC Press, 2001. [25](#), [43](#)
- MUNSHI, A. The openssl specification. **2009 IEEE Hot Chips 21 Symposium (HCS)**, 2009. [26](#)
- MURAKAMI, T.; KASAHARA, R.; SAITO, T. An implementation and its evaluation of password cracking tool parallelized on gpgpu. **2010 10th International Symposium on Communications and Information Technologies**, 2010. [26](#)
- MURPHY, S. LinkedIn confirms, apologizes for stolen password breach. **Mashable**, 2012. [32](#)
- ORNBO, G. **Sams teach yourself Go in 24 hours: next generation systems programming with Golang**. [S.l.]: Sams, 2018. [33](#)
- PATTERSON, D. **Tarski and philosophy**. [S.l.]: Oxford University Press, 2008. [41](#)
- PEARL, J. Heuristics: Intelligent search strategies for computer problem solving. **Telematics and Informatics**, v. 3, n. 4, p. 305, 1986. [30](#)
- PICOLET, J. **Hash crack: password cracking manual**. [S.l.]: Netmux, 2017. [30](#)
- RODRIGUES, B. A.; PAIVA, J. R. B.; GOMES, V. M.; MORRIS, C.; CALIXTO, W. P. Passfault: an open source tool for measuring password complexity and strength. **International Multi-Conference on Complexity, Informatics and Cybernetics**, 2017. [27](#), [30](#), [31](#)
- ROSEN, K. H. **UNIX: the complete reference**. [S.l.]: McGraw-Hill, 2007. [33](#)
- SAHIN, C. S.; LYCHEV, R.; WAGNER, N. General framework for evaluating password complexity and strength. **CoRR**, abs/1512.05814, 2015. [27](#), [42](#)
- SANADHYA, S. K.; SARKAR, P. New local collisions for the sha-2 hash family. **Lecture Notes in Computer Science Information Security and Cryptology - ICISC 2007**, p. 193–205, 2007. [63](#)
- SHANNON, C. E. A mathematical theory of communication. 2009. [26](#), [42](#)

SHAY, R.; KOMANDURI, S.; KELLEY, P. G.; BAUER, L.; CHRISTIN, P. G. L. and Nicolas; MAZUREK, M. L.; CRANOR, L. F. Encountering stronger password requirements: User attitudes and behaviors. In: **In SOUPS 10: Proceedings of the Sixth Symposium on Usable privacy and Security**. ACM. [S.l.: s.n.], 2010. [26](#)

SHEN, W.; KHANNA, R. Prolog to iris recognition: An emerging biometric technology. **Proceedings of the IEEE**, v. 85, n. 9, p. 1347 – 1347, 1997. [25](#)

SINGH, S.; YAMINI, M. Voice based login authentication for linux. **2013 International Conference on Recent Trends in Information Technology (ICRTIT)**, 2013. [25](#)

STALLINGS, W. **Cryptography and network security: principles and practice**. [S.l.]: Pearson, 2017. [25](#)

SWIDERSKI, F.; SNYDER, W. **Threat Modeling**. [S.l.]: O'Reilly Media, Inc., 2009. [26](#)

WANG, D.; HE, D.; CHENG, H.; WANG, P. fuzzypsm: A new password strength meter using fuzzy probabilistic context-free grammars. **2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)**, 2016. [26](#)

WANG, X.; YU, H.; YIN, Y. L. Efficient collision search attacks on sha-0. **Advances in Cryptology - CRYPTO 2005 Lecture Notes in Computer Science**, p. 1–16, 2005. [62](#)

WEIR, M.; AGGARWAL, S.; MEDEIROS, B. D.; GLODEK, B. Password cracking using probabilistic context-free grammars. **2009 30th IEEE Symposium on Security and Privacy**, 2009. [26](#)

WHEELER, D. L. zxcvbn: Low-budget password strength estimation. In: **USENIX Security Symposium**. [S.l.: s.n.], 2016. p. 157–173. [26](#)

ZHANG, Y.; KOUSHANFAR, F. Robust privacy-preserving fingerprint authentication. **2016 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)**, 2016. [25](#)

GLOSSÁRIO

String - Sequência de caracteres.

Senha - *String* utilizada na autenticação de usuários em sistemas computacionais.

Função de Dispersão Criptográfica - Função unidirecional que toma como entrada *string* de tamanho e retorna como saída *string* de tamanho fixo, chamada *hash*.

Hash - Resultado do cálculo de determinada Função de Dispersão Criptográfica.

Rainbow Table - Tabelas que organizam senhas previamente quebradas para consulta rápida..

Heurística - Algoritmos que sacrificam a completude da solução em troca da otimização do tempo de execução.

Passfault - Passfault é a ferramenta com o objetivo de fazer com que a complexidade e força das senhas sejam facilmente entendidas pela população.

Go - Linguagem de programação compilada, estaticamente tipada e baseada em C, com foco em concorrência.

Processos Sequenciais Comunicantes - Padrões de interação em sistemas computacionais concorrentes.

Gorrotina - *Threads* minimalistas da linguagem Go que compõem os Processos Sequenciais Comunicantes.

Canal - Conexões entre Gorrotinas concorrentes.

Conjunto Amostral - Multiconjunto de *strings* que representam senhas de usuários em determinado sistema.

Modelo - Fórmulas lógicas e predicados lógicos que determinam a constituição de determinado grupo de *string*.

Domínio - Conjunto de *strings* que obedecem às fórmulas lógicas e predicados lógicos de determinado Modelo.

Cardinalidade - Número de elementos contidos no Domínio de determinado Modelo.

Modelo Elementar - Modelo que representa as subdivisões atômicas nas partições da estrutura da *string*.

Modelo Composto - Modelo cujo Domínio é formado pelo produto cartesiano de diferentes Modelos Elementares.

Modelo Crítico - Modelo Composto com Domínio de Cardinalidade mínima que descreve determinada *string* em determinado Conjunto Amostral.

Token - *Substring* correspondente a cada Modelo Elementar dentro da estrutura de determinado Modelo Composto.

Frequência Relativa - Probabilidade de ocorrência de determinado Modelo Crítico em determinado Conjunto Amostral.

Entropia - Indicador de incerteza atribuído a determinado Modelo Crítico.

Corinda - Software cujo objetivo é quebrar *hashes* de senhas baseado em heurísticas concorrentes.