UNIVERSIDADE DO EXTREMO SUL CATARINENSE - UNESC
CURSO DE CIÊNCIA DA COMPUTAÇÃO

DIEGO MOTTA DA SILVA

ANÁLISE DE TÉCNICAS DE CRIPTOGRAFIA ENVOLVIDAS NA UTILIZAÇÃO DO BITCOIN

CRICIÚMA
2016

DIEGO MOTTA DA SILVA

ANÁLISE DE TÉCNICAS DE CRIPTOGRAFIA ENVOLVIDAS NA UTILIZAÇÃO DO BITCOIN

Trabalho de Conclusão de Curso, apresentado para obtenção do grau de bacharel no curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC.

Orientador: Prof. Esp. Valter Blauth Junior

CRICIÚMA

Diego Motta da Silva

ANÁLISE DE TÉCNICAS DE CRIPTOGRAFIA ENVOLVIDAS NA UTILIZAÇÃO DO BITCOIN

Trabalho de Conclusão de Curso aprovado pela Banca Examinadora para obtenção do Grau de Bacharel, no Curso de Ciência da Computação da Universidade do Extremo Sul Catarinense, UNESC, com Linha de Pesquisa em Segurança da Informação

Criciúma, 01 de dezembro de 2016.

BANCA EXAMINADORA

Prof. Valter Blauth Júnior - Esp - UNESC - Orientador

Prof. Paulo João Martins - MSc - UNESC

Prof. Kristian Madeira - PhD - UNESC

Dedico este trabalho a meu pai,
que dedicou tanto de seu tempo a mim.

AGRADECIMENTOS

Primeiramente agradeço minha noiva e minha mãe por suportar todo o meu estresse durante a redação deste trabalho.

Agradeço meu pai, mesmo não estando aqui, sempre viverá dentro dos que o amam.

Agradeço meu orientador professor Valter e a banca examinadora, professores Kristian e Paulo, juntamente com a professora Merisandra por todas as contribuições ao trabalho.

Por fim, agradeço Alan Turing, pois sem sua contribuição à ciência, este trabalho jamais teria sido realizado.

“Nós só podemos ver um pouco do futuro, mas o suficiente para perceber que há muito a fazer”

Alan Turing

RESUMO

O *Bitcoin* apresenta um desafio chamado *nonce* para os nós concorrentes resolverem afim de obter aceitação do bloco calculado ao minerar, este desafio nada mais é do que um cálculo de *hash* parametrizado com uma quantidade de bits zero ao inicio do mesmo, desta forma, este trabalho realiza o cálculo de *hashes* em um protótipo e logo uma análise utilizando três algoritmos de *hashes* sendo eles: SHA-256, SHA-1 e MD5, trata-se de uma análise estatística com auxílio do software SPSS apresentando conclusões sobre a quantidade de *hashes* necessária para cada algoritmo calcular o *nonce* bem como o tempo que levou para alcançar o objetivo de calcular um *nonce*. Após as aplicações dos testes de kruskal-wallis e mann-whitney, foi possível perceber se havia diferença estatisticamente significativa, os únicos testes que não apresentaram diferença estatisticamente significativa foram nas quantidades de hashes quando comparado com mesma parametrização, tendo valor p de 0,850 para parametrização 20 zeros e valor p 0,398 para parametrização de 25 zeros, nos demais testes foi possível verificar um valor p inferior a 0,001, concluindo a existência de diferença estatisticamente significativa onde em uma mesma parametrização o SHA-256 possui maior custo que o SHA-1 que por sua vez tem custo maior que o MD5, quando comparados entre as parametrizações de 20 e 25 zeros foi obtido que a parametrização de 25 zeros é, em todos os casos testados, estatisticamente mais custosa nas métricas observadas.

Palavras-chave: Bitcoin.Hash.SHA-256.SHA-1.MD5.

ABSTRACT

The Bitcoin presents a challenge called nonce to all concurrent nodes to obtain acceptance of the calculated block at the mining process, this challenge is the calculation of a hash parameterizated with a certain number of zero bits at the start of the hash, this arcticle calculates the hashes in a prototype followed by a analysis using three hash algorithms: SHA-256, SHA-1 and MD5, that's a statistic analysis using the IBM SPSS software, presenting conclusion about the quantity of hashes and the time necessary to each algorithm calculate a nonce. After the application of kuskal-wallis' and mann-whitney's testes, it was possible to see if there was significant statistic difference between the quantity of hashes when compared with the same parameterization, having p value of 0,850 to 20 zeros parameterization and p value 0,398 to the 25 zeros parameterization, the other tests shown a p value of 0,001, concluding the existence of significant statistic difference, where in the same parameterization the SHA-256 have a greater cost than the SHA-1, which has a greater cost than MD5, when compared between the 20 zeros and the 25 zeros parameterizations it was concluded that the 25 zeros is, in all tested cases, statistically more costly in the observed metrics.

LISTA DE ILUSTRAÇÕES

Figura 1 – Ataques passivos 14
Figura 2 – Ataques ativos 1 15
Figura 3 – Ataques ativos 2 15
Figura 4 – Criptografia simétrica 17
Figura 5 – Tabela de Viginère. 19
Figura 6 – Transposição de colunas 20
Figura 7 – *rail fence* 20
Figura 8 – MAC 25
Figura 9 – SHA-1 27
Figura 10 – Transação 31
Figura 11 – Timestamp Server 32
Figura 12 – Bloco de transação 33
Figura 13 – Markle Tree 36
Figura 14 – Obtenção do ramo para verificação do pagamento 37
Figura 15 – Combinação e separação de moedas 38
Figura 16 – Relação entre modelo tradicional e novo de privacidade 39
Figura 17– Console *Spring-boot* 47
Figura 18 – Página inicial JHipster 48
Figura 19 – Diálogo de geração 49
Figura 20 – Processamento de dados 49
Figura 21 – Iteração de bits 50
Figura 22 – Classe de utilidades 51
Figura 23 – Classe de cálculo 52
Figura 24 – Comparação de pares para Tempo de Processamento com 20 zeros 58
Figura 25 – Comparação de pares para Tempo de Processamento com 25 zeros 63

LISTA DE TABELAS

Tabela 1 – Resumo do processamento de caso para quantidade de hashes 53
Tabela 2 – Descritivos para quantidade de hashes com 20 zeros 54
Tabela 3 – Descritivos para tempo de processamento com 20 zeros 55
Tabela 4 – Percentis para quantidade de hashes com 20 zeros 56
Tabela 5 – Percentis para tempo de processamento com 20 zeros 56
Tabela 6 – Testes de normalidade para quantidade de hashes com 20 zeros 56
Tabela 7 – Testes de normalidade para tempo de processamento com 20 zeros 57
Tabela 8 – Teste de Kruskal-Wallis para quantidade de hashes com 20 zeros 57
Tabela 9 – Teste de Kruskal-Wallis para tempo de processamento com 20 zeros ... 57
Tabela 10 – Descritivos quantidade de hashes com 25 zeros 59
Tabela 11 – Descritivos tempo de processamento com 25 zeros 60
Tabela 12 – Percentis para quantidade de hashes com 25 zeros 60
Tabela 13 – Percentis para tempo de processamento com 25 zeros 61
Tabela 14 – Testes de normalidade para quantidade de hashes com 25 zeros 61
Tabela 15 – Testes de normalidade para tempo de processamento com 25 zeros .. 61
Tabela 16 – Teste de Kruskal-Wallis para quantidade de hashes com 25 zeros 62
Tabela 17 – Teste de Kruskal-Wallis para tempo de processamento com 25 zeros . 62
Tabela 18 – Descritivos quantidade de hashes com SHA-256 64
Tabela 19 – Descritivos tempo de processamento com SHA-256 65
Tabela 20 – Percentis quantidade de hashes com SHA-256 65
Tabela 21 – Percentis tempo de processamento com SHA-256 66
Tabela 22 – Testes de normalidade quantidade de hashes com SHA-256 66
Tabela 23 – Testes de normalidade tempo de processamento com SHA-256 66
Tabela 24 – Teste U de Mann-Whitney para quantidade de hashes com SHA-256 . 67
Tabela 25 – Teste U de Mann-Whitney tempo de processamento com SHA-256 67
Tabela 26 – Descritivos quantidade de hashes com SHA-1 68
Tabela 27 – Descritivos tempo de processamento com SHA-1 69
Tabela 28 – Percentis quantidade de hashes com SHA-1 69
Tabela 29 – Percentis tempo de processamento com SHA-1 70
Tabela 30 – Testes de normalidade quantidade de hashes com SHA-1 70
Tabela 31 – Testes de normalidade tempo de processamento com SHA-1 70

Tabela 32 – Teste U de Mann-Whitney para quantidade de hashes com SHA-1 71
Tabela 33 – Teste U de Mann-Whitney para tempo de processamento com SHA-1 71
Tabela 34 – Descritivos quantidade de hashes com MD5 72
Tabela 35 – Descritivos tempo de processamento com MD5 73
Tabela 36 – Percentis quantidade de hashes com MD5 73
Tabela 37 – Percentis tempo de processamento com MD5 74
Tabela 38 – Testes de normalidade quantidade de hashes com MD5......................74
Tabela 39 – Testes de normalidade tempo de processamento com MD5 74
Tabela 40 – Teste U de Mann-Whitney para quantidade de hashes com MD5 75
Tabela 41 – Teste U de Mann-Whitney para tempo de processamento com MD5 ...75
Tabela 42 – Quantidade de hashes com 20 zeros 75
Tabela 43 – Tempo de processamento com 20 zeros 76
Tabela 44 – Quantidade de hashes com 25 zeros 76
Tabela 45 – Tempo de processamento com 25 zeros 77
Tabela 46 – Comparações de mesmo algoritmo com parametrizações diferentes ...77
Tabela 47 – Amostra dos dados obtidos 78

LISTA DE ABREVIATURAS E SIGLAS

AES	*Advanced Encryption Standard*
ASIC	*Application-specific integrated circuit*
DES	*Data Encryption Standard*
EFF	*Electronic Frontier Foundation*
FIPS	*Federal Information Processing Standards*
IBM	*International Business Machine*
MD	*Message Digest*
NIST	*National Institute of Standards and Technology*
NSA	*National Security Agency*
OSI	*Open Systems Interconnection*
RSA	Rivest-Shamir-Adleman
SPSS	*Statistical Package for the Social Sciences*

SUMÁRIO

INTRODUÇÃO 9
1.1 OBJETIVO GERAL 9
1.2 OBJETIVOS ESPECÍFICOS 10
1.3 JUSTIFICATIVA 10
1.4 ESTRUTURA DO TRABALHO 11
2 SEGURANÇA DE REDES 12
2.1 ATAQUES DE REDE 14
2.2 CRIPTOGRAFIA 16
2.2.1 Cifras simétricas 16
2.2.2 Algoritmos de chave simétrica 21
2.2.3 Algoritmos de chave pública 22
2.2.4 Assinaturas digitais 24
2.2.5 MD5 26
2.2.6 SHA-1 27
3 BITCOIN 30
3.1 TRANSAÇÕES 31
3.2 TIMESTAMP SERVER 32
3.3 PROVA REAL 33
3.4 REDE DE PROCESSAMENTO 34
3.5 GERAÇÃO DE MOEDAS 35
3.6 RECUPERANDO ESPAÇO EM DISCO 35
3.7 VERIFICAÇÃO DE PAGAMENTO SIMPLIFICADA 36
3.8 COMBINANDO E SEPARANDO VALORES DE MOEDAS 37
3.9 PRIVACIDADE 38
3.10 CÁLCULOS 39
3.11 CONCLUSAO DE NAKAMOTO 42
4 TRABALHOS CORRELATOS 43
5 ANÁLISE DOS ALGORITMOS 45
5.1 AMOSTRAGEM ESTATÍSTICA 45
5.2 METODOLOGIA 45
5.2.1 Tecnologia 46

5.2.2 Protótipo 47
5.2.3 Análises estatísticas com o SPSS 52
5.3 RESULTADOS 75
5.4 DISCUSSÃO DOS RESULTADOS OBTIDOS 79
6 CONCLUSÃO 80
REFERÊNCIAS 82

INTRODUÇÃO

O *Bitcoin* é uma moeda corrente eletrônica com visão descentralizada, ou seja, livre de influências de autoridades reguladoras; implementada usando criptografia e tecnologia *peer-to-peer,* documentada em 2008 e desenvolvida no ano seguinte pelo pseudônimo Satoshi Nakamoto, não sendo certa a sua real identidade até pouco tempo (LITTLE, 2014).

Em junho de 2016, o australiano Craig Wright se sentiu forçado a revelar sua identidade de principal criador do *Bitcoin* para algumas mídias internacionais por causa da perseguição de algumas pessoas sobre serem possíveis inventores do *Bitcoin*, afirma que nunca foi de sua vontade obter qualquer tipo de fama, apenas seguiu sua ideologia liberal (GLOBO, 2016).

Sobre formas de uso, o *Bitcoin* pode ser utilizado como qualquer moeda física, basta ter uma carteira e obter um valor em moeda, podendo ser minerando ou por qualquer transação envolvendo dois usuários (venda de produtos ou pagamento de serviços por exemplo), tendo saldo em carteira se pode realizar compras de qualquer natureza, desde que o vendedor aceite a moeda como pagamento. Outra forma de uso observada no mercado econômico é o investimento em *Bitcoin*, de mesma forma como comprar uma moeda estrangeira afim de lucrar, pode-se comprar *Bitcoin* e vender após ter sido valorizado, apresentando os riscos e custos que mercado acomoda.

Para o *Bitcoin* funcionar, o uso de criptografia é mandatório e o seu desempenho deve estar de acordo com as expectativas; como foi abordado neste trabalho, a geração de um bloco não pode ser muito rápida e nem muito lenta, ela deve possuir o tempo ideal e isto depende diretamente de como a estratégia lida com os algoritmos utilizados; para tal, uma visão constante do desempenho dos algoritmos envolvidos é essencial.

1.1 OBJETIVO GERAL

Analisar estatisticamente os mais importantes algoritmos de criptografia envolvidos no *Bitcoin* quanto ao desempenho.

1.2 OBJETIVOS ESPECÍFICOS

Os objetivos específicos deste trabalho são dados por:

a) conhecer o funcionamento das técnicas de criptografia envolvidas no *Bitcoin* e como são empregadas;
b) apresentar como funciona a criptografia relacionada à mineração do *Bitcoin* e aos processos econômicos envolvidos;
c) demonstrar o funcionamento dos mais importantes algoritmos de criptografia utilizados no *Bitcoin*;
d) utilizar o software SPSS para analisar estatisticamente os algoritmos de criptografia utilizados no *Bitcoin*.

1.3 JUSTIFICATIVA

A utilização de *Bitcoin* se iniciou em 2008, em meio à crise mundial econômica, tomando sua real proeminência global em 2012, ficando sob holofotes desde então. A capacidade de processamento coletiva somada do *Bitcoin* chega a ser cem vezes superior ao desempenho da lista dos 500 melhores supercomputadores do mundo somados, mais de 50 mil petaflops ($50.000\ x\ 10^{15}$ operações de ponto flutuante). Não obstante, o sucesso do *Bitcoin* mostrou alguns problemas, sendo estes: Não é tão seguro e anônimo quanto se pensava, o sistema de distribuição está ficando pesado e levou a uma "corrida armamentista" insustentável em busca de melhores máquinas mineradoras (*Bitcoin under pressure*, 2013).

Aproveitando a conexão do artigo *Bitcoin under pressure* com a guerra fria, é possível fazer um comparativo da mineração de dados do *Bitcoin* com o avanço tecnológico durante as guerras; afinal, foi na necessidade da guerra que Alan Turing iniciou seu trabalho para o serviço secreto britânico, onde sua pesquisa lhe rendeu o título de "o pai da computação". Assim também como a própria corrida armamentista trouxe inovações tecnológicas usadas pela humanidade até hoje (SATO, 2000; HODGES, 2001).

Desta forma, tem-se o desafio de manter a estabilidade do *Bitcoin*, para tal, é de vital importância analisar constantemente os métodos de criptografia que

envolvem a economia da moeda virtual. Nakamoto (2008), criador do *Bitcoin*, comenta em seu artigo sobre o aumento automático da complexidade do cálculo ao incrementar o número de zeros necessários do *nonce* calculado para o bloco, este *nonce* é um *hash* calculado através do algoritmo SHA-1 e todo o processo de mineração depende desta técnica para que a geração de blocos não seja nem muito demorada e nem rápida demais.

Baseado inteiramente em dois pilares: a economia e a criptografia, o *Bitcoin* caminha sobre as incertezas do mercado com toda a segurança que o poder de processamento de seus nós pode oferecer através da criptografia. A certeza que as partes precisam para realizar suas transações está confiada aos processos eletrônicos e estes precisam sempre se provar preparados para o mercado, através de estatísticas que passem confiança. É com essa ideologia que este trabalho é escrito, para demonstrar se os algoritmos selecionados por Nakamoto em 2008 conseguem atingir, quase dez anos depois, as espectativas.

1.4 ESTRUTURA DO TRABALHO

Este trabalho é dividido em cinco capítulos, onde o primeiro comenta sobre a segurança de redes, começando com as estratégias de segurança e os ataques, logo a seguir comentando sobre os tipos de criptografia existentes e alguns algoritmos que são usados para tal.

O segundo capítulo é sobre o *Bitcoin*, comentando sobre como sua estrutura é formada, como foi pensado em sua criação e sobre a relevância que a criptografia tem sobre o funcionamento da moeda.

O terceiro capítulo são os trabalhos correlatos, onde são descritos alguns trabalhos com natureza semelhantes a este.

O quarto capítulo é a análise dos algoritmos, é onde o objetivo deste trabalho é alcançado, demonstrando como foi feito através da metodologia e os seus resultados.

O quinto e último capítulo é a conclusão, que comenta sobre os fins obtidos com a realização dos testes feitos.

2 SEGURANÇA DE REDES

Afim de abordar o tema criptografia, primeiramente se vê necessário um esclarecimento sobre a área em que o mesmo está situado: a segurança de redes.

A maioria dos problemas de segurança se dão intencionalmente, por indivíduos de caráter malicioso, visando algum lucro próprio; desta forma, garantir a segurança de uma rede consiste em, muitas vezes, enfrentar pessoas inteligentes, dedicadas e, algumas vezes, muito bem subsidiadas. Ainda assim deve-se levar em conta que a maioria dos casos não vêm de indivíduos aleatórios, mas sim por elementos descontentes com a organização vitimada, como demonstram os registros policiais (TANENBAUM, 2003).

Para entender um pouco melhor, Tanenbaum (2003) separa os problemas de segurança em quatro áreas, as quais um sistema seguro deve garantir, são elas: sigilo, autenticação, não-repúdio e controle de integridade.

Para Tanenbaum (2003), o sigilo é garantir a não liberação de informação para indivíduos não autorizados, tendo relação direta com controle de acesso e confidencialidade dos dados trazido por Stallings (2008), onde o controle de acesso permite ou não que um usuário acesse aquele recurso e quais poderes este terá ao acessá-lo; sendo assim, esta confidencialidade possui algumas divisões:

a) uma destas divisões é a confidencialidade da conexão, a qual visa proteger todos os dados de um usuário em uma conexão;
b) outro ponto é a confidencialidade sem conexão, que por sua vez protege os dados em um único bloco de dados;
c) ainda existe a confidencialidade por campo selecionado, onde esta aspira proteger dados seletos, seja em uma conexão ou em um bloco de dados;
d) por fim, temos a confidencialidade do fluxo de tráfego, esta última lida com a proteção de informações que podem ser escutadas durante o tráfego na rede.

O segundo tipo, a autenticação, visa determinar se a entidade cuja intenção é se comunicar é aquela quem alega ser. A autenticação existe de duas formas, uma delas é a autenticação de entidade par, onde se associa a uma conexão lógica para confiabilidade da identidade das partes conectadas, e a outra

forma é a autenticação da origem de dados, ou seja, durante um tráfego sem conexão, garante que a origem da informação recebida é realmente a que afirma ser (STALLINGS, 2008).

Já o não-repúdio ou irretratabilidade é aquele que protege contra a negação, pode acontecer de duas formas, o não-repúdio de origem, onde prova que aquela mensagem foi realmente enviada por aquela entidade e também o não-repúdio de destino, o qual prova que a mensagem foi realmente recebida pelo destinatário (STALLINGS, 2008).

Por último, deve-se também garantir a integridade, evitando modificações, inserções, exclusões ou repetições na mensagem. Este controle de integridade é dividido em cinco campos, que seguem:

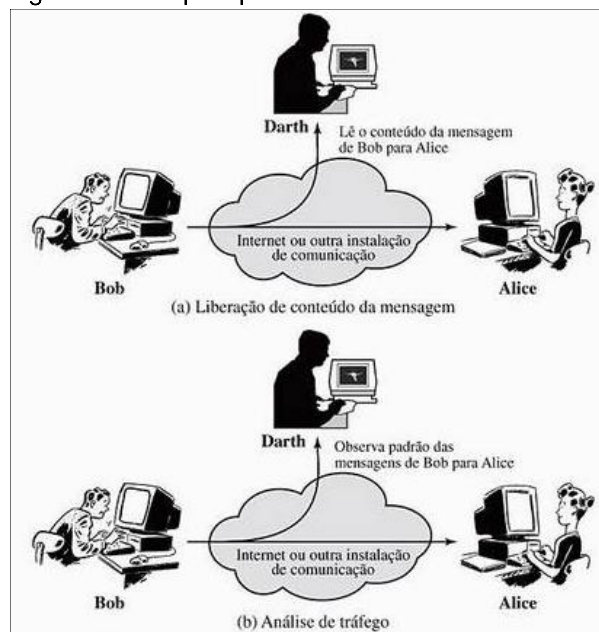
a) o primeiro é a integridade da conexão com recuperação, em que esta garante a integridade dos dados em uma conexão contra modificação, inserção, exclusão ou repetição de quaisquer dados, tentando a recuperação destes;
b) o segundo é a integridade da conexão sem recuperação, funcionando de mesma forma, apenas não apresenta tentativa de recuperar a mensagem;
c) o terceiro ponto é a integridade da conexão com campo selecionado, a qual visa a não alteração de determinados campos em um bloco transferido por uma conexão;
d) prosseguindo temos a integridade sem conexão, onde esta prevê a integridade de um único bloco de dados sem conexão;
e) por último temos a integridade sem conexão com campo seletivo, trabalhando de mesma forma a anterior, todavia com campos selecionados a serem verificados quando à alteração (STALLINGS, 2008).

Tendo em vista tais intenções a serem seguidas para manter uma comunicação segura, a próxima etapa é verificar quais ataques podem atentar contra estas estratégias e quais maneiras podem ser alcançadas para burlar estas seguranças aspiradas por redes seguras.

2.1 ATAQUES DE REDE

Vale aqui apresentar alguns tipos de ataques que podem ocorrer na rede, divididos em duas áreas: ataques passivos e ataques ativos. Os ataques passivos podem vir em forma de liberação de conteúdo de mensagem ou análise de tráfego, apresentados na figura 1, onde a primeira consiste em descobrir o conteúdo de alguma mensagem sigilosa sem alterar nada, já a segunda vem de forma mais sutil, a análise de tráfego apenas observa o que está no fluxo de dados e guarda as informações, este tipo de ataque é difícil de ser identificado, uma vez que não altera ativamente nada do que está sendo transmitido, uma boa forma de se precaver de um ataque passivo é o uso de criptografia (STALLINGS, 2008).

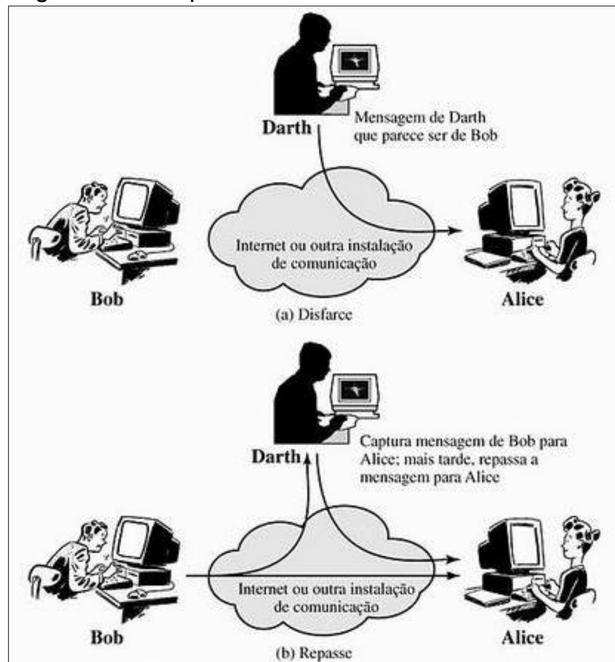
Figura 1 – Ataques passivos



Fonte: Stallings (2008).

Por conseguinte, temos os ataques ativos, onde se pode encontrar estratégias como o disfarce (figura 2), onde o malfeitor se passa por outra pessoa para acessar dados que não possui autorização para ver; também temos o ataque de repetição (figura 2), já este consiste basicamente em repetir uma mensagem para produzir um efeito não autorizado;

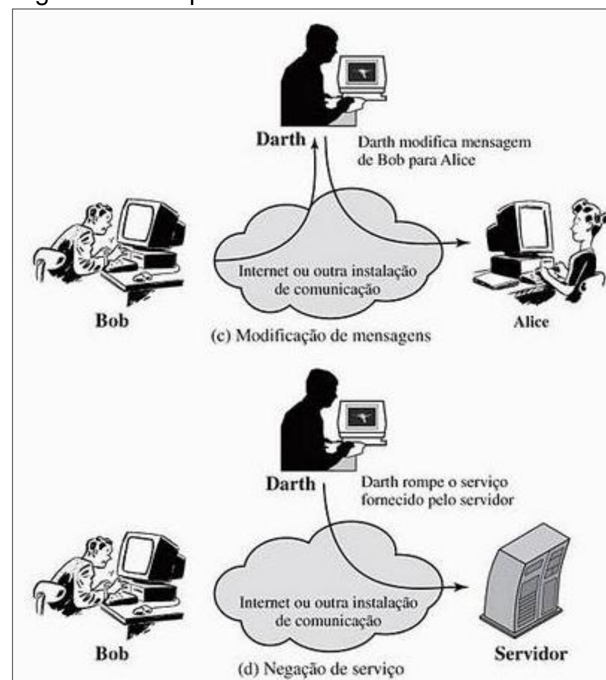
Figura 2 – Ataques ativos 1



Fonte: Stallings (2008).

Outro aspecto de ataque é a modificação de mensagens (figura 3), onde este visa alterar, adiar ou reordenar a mensagem para seu benefício; por último temos a negação de serviço (figura 3), o qual visa desativar ou sobrecarregar uma rede a fim de não permitir seu funcionamento (STALLINGS, 2008).

Figura 3 – Ataques ativos 2



Fonte: Stallings (2008).

Estes ataques, não se enquadram especificamente em uma das áreas citadas no capítulo anterior, mas no conjunto delas, tendo em vista que as áreas estão interligadas, ou seja, uma brecha em uma delas pode comprometer todo o sistema, o tornando vulnerável a qualquer um dos ataques citados acima. Ainda no topo disto, a melhor estratégia de segurança de rede não garante todas essas questões, basta que o elemento humano falhe para toda a estratégia de segurança se tornar irrelevante (TANENBAUM, 2003).

Tendo assim introduzido a segurança de redes, pode-se para tal fim, iniciar a apresentação das técnicas que existem acerca da maneira a qual podemos nos precaver destes ataques.

2.2 CRIPTOGRAFIA

Fora a camada física, segundo o modelo *open systems interconnection* (OSI), praticamente toda a segurança tem como base a criptografia (TANENBAUM, 2003).

Seguindo a cronologia, as primeiras cifras encontradas na nossa história são as simétricas, devendo seu nome ao uso da mesma chave para realizar a cifra e para a desfazer.

2.2.1 Cifras simétricas

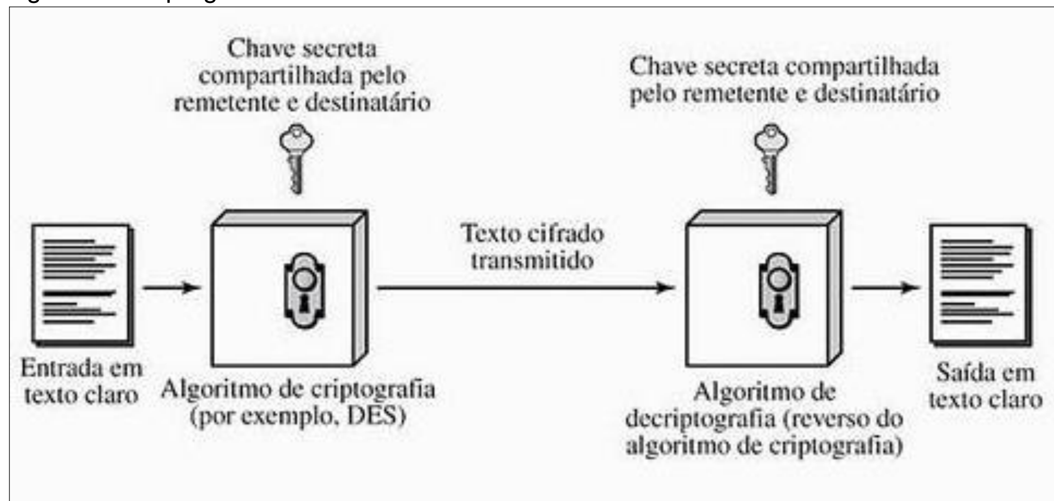
Para iniciar a abordagem sobre a criptografia, fala-se sobre o tipo de cifra mais usado, a cifra simétrica utiliza a mesma chave para encriptar e decriptar, e essa chave pode ser usada para comunicação bidirecional, razão pela qual ser denominada simétrica (FOROUZAN, 2013).

Também conhecida como criptografia convencional ou de chave única, a cifra simétrica era o único tipo de criptografia até a criação da chave pública em 1970. Este tipo de criptografia é dividido em cinco elementos, sendo eles:

a) texto claro, como sendo a mensagem original;
b) algoritmo de criptografia, responsável por fazer substituições e transformações no texto claro;

c) chave secreta, é um valor independente do texto claro, todavia o texto cifrado sofre alterações dependentes da chave;
d) texto cifrado, sendo este um texto ininteligível originado da combinação do texto claro e da chave secreta com o algoritmo de criptografia;
e) algoritmo de decriptografia, realiza o trabalho inverso do algoritmo de criptografia (STALLINGS, 2008).

Figura 4 – Criptografia simétrica



Fonte: Stallings (2008).

Como ilustrado na figura 4, a criptografia simétrica depende do conhecimento de um algoritmo de criptografia e decriptografia públicos e de uma chave secreta compartilhada apenas pelo emissor e receptor da mensagem, qualquer compartilhamento da chave com terceiros pode acarretar em leitura da mensagem por indivíduos indesejados (STALLINGS, 2008).

As cifras de chave simétrica podem ser divididas em cifras tradicionais e cifras modernas. As tradicionais são simples, orientadas a caracteres e que não mais podem ser consideradas seguras nos padrões atuais. Por outro lado, as cifras modernas são complexas, orientadas a bits, o que as torna mais seguras.

Começa-se o estudo das cifras simétricas pelas tradicionais, abrindo caminho para a discussão de cifras mais complexas. Primeiramente apresentando a cifra de substituição, a qual possui uma das mais antigas cifras, chamada de cifra de César.

2.2.1.1 Cifras de substituição

O funcionamento da cifra de substituição consiste em pegar as letras do texto claro e simplesmente as substituir por outras, como por exemplo na cifra de César, que consiste em substituir a letra pela sua terceira seguinte no alfabeto, ou seja, *a* se torna *D*, *b* se torna *E*, *c* se torna *F* e assim por diante, por exemplo a palavra *redes* passa a ser UHGHV utilizando a cifra de César. Generalizando um pouco a cifra de César, obtém-se que a letra seja deslocada *k* letras ao invés de três, desta forma *k* passa a ser a chave desta cifra simétrica (TANENBAUM, 2003).

Ainda outro tipo de cifra de substituição é a cifra monoalfabética. Nesta cifra, um caractere sempre é trocado por um mesmo caractere em toda a cifra, independentemente de sua posição, ou seja, digamos que o algoritmo troque a letra *A* pela letra *D*, em todas as palavras do texto a letra *A* se tornará *D*, a relação é de um para um (FOROUZAN, 2013).

Por seguinte, temos outra cifra que se enquadra na substituição é a cifra polialfabética, onde esta consiste em um conjunto de regras de substituição monoalfabéticas e uma chave; o modelo mais conhecido deste tipo é a cifra de Viginère, a qual possui 26 regras monoalfabéticas variantes da cifra César (deslocando-se de 0 a 25), cada uma delas indicada por uma letra chave; como por exemplo no algoritmo de César sendo chave 3, teríamos a chave *d* em Viginère. Para ilustrar, será utilizado a tabela de Viginère (figura 5) para encriptar o exemplo *disfarcedescoberto*, utilizando a chave espião, entretanto, o uso de uma chave menor que o texto claro leva a um problema de frequência de substituições, por ter que repetir a chave *n* vezes até alcançar o tamanho da mensagem, por esta razão, Viginère propôs o sistema de autochave, concatenando o próprio texto na chave, no exemplo a chave fica *espiaodisfarcedesc*, desta forma, o texto cifrado fica: *hahuaffmvjxtqfhvlq* (STALLINGS, 2008).

Figura 5 – Tabela de Viginère.

	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
a	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
b	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
c	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
d	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
e	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
f	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
g	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
h	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
i	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
j	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
k	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
l	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
m	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
n	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
o	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
p	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
r	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
s	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
t	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
u	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
v	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
w	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
x	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Fonte: Stallings (2008).

2.2.1.2 Cifras de transposição

Diferentemente das cifras de substituição, as de transposição não trocam um símbolo por outro todavia trocam suas posições. Um símbolo na primeira posição na cifra pode aparecer na décima posição, mas é certo que aparecerá no exemplo abaixo (figura 6) é apresentado a cifra de transposição de colunas com a chave *redes* e a mensagem sendo *cifradetransposicaodecolunas* (FOROUZAN, 2013).

Figura 6 – Transposição de colunas

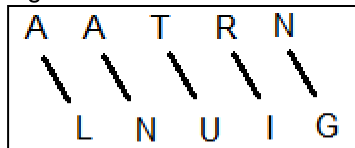
R	E	D	E	S
4	2	1	3	5
c	i	f	r	a
d	e	t	r	a
n	s	p	o	s
i	s	a	o	d
e	c	o	l	u
n	a	s		

Fonte: adaptado de Stallings (2008).

Após ordenar a mensagem de acordo com as colunas, o texto cifrado possui a seguinte forma: *ftpaosiesscarroolcdnienaasdu*.

Outra cifra de transposição de simples compreensão é a *rail fence*, onde a mensagem é escrita em uma diagonal e lida de forma convencional, por exemplo na mensagem Alan Turing na figura 7 (STALLINGS, 2008).

Figura 7 – *rail fence*



Fonte: adaptado de Stallings (2008).

Lendo de maneira convencional, temos o texto cifrado *aatrnlnuig*.

Para o uso de tais cifras, vê-se necessário o uso de certos algoritmos para padronizar o uso da criptografia, surge então um algoritmo de chave simétrica desenvolvido pela *International Business Machine* (IBM) para suprir esta necessidade.

2.2.2 Algoritmos de chave simétrica

Os algoritmos de chave simétrica, como visto anteriormente, são aqueles que usam a mesma chave para encriptar e decriptar uma mensagem, destes algoritmos, um exemplo bastante utilizado é o *Data Encryption Standard* (DES).

2.2.2.1 Data Encryption Standard

Desenvolvido pela IBM a partir do LUCIFER, com chave de 128 bits, o DES foi criado com uma chave de 56 bits com a visão de caber em um único chip, em 1977 foi adotado pelo governo dos Estados Unidos como uma cifra padrão oficial para informações não confidenciais. Este Algoritmo consiste em criptografar 64 bits de dados em 19 etapas de transformação do texto, onde a primeira e a última são transposições inversas independentes da chave, a penúltima é uma permuta dos 32 bits da esquerda com os 32 da direita, os 16 cálculos restantes são funções repetidas baseadas na chave (TANENBAUM, 2003).

O DES se provou inseguro em 1998, quando a *Electronic Frontier Foundation* (EFF) anunciou a quebra da criptografia DES através de uma máquina especial custando 250 mil dólares, tornando a arquitetura desta máquina pública para ser montada por terceiros. Sendo assim, o AES e o triplo DES são as alternativas mais importantes utilizadas (STALLINGS, 2008).

2.2.2.2 DES triplo

O 3DES consiste no uso de duas chaves e três estágios, no primeiro, uma criptografia com a chave 1, o segundo é uma descriptografia com a chave 2 e por último uma nova encriptação com a chave 1. Desta forma, duas questões são levantadas, por que usar duas chaves e não três? E por que não usar encriptação nas três etapas? Primeiramente, o uso de somente duas chaves se dá por conta de que o tamanho de 168 bits teria um custo maior do que o benefício trazido, onde 112 já seria considerado o suficiente para uso comercial por um tempo considerável. Já o motivo de não usar a criptografia três vezes seguidas é a compatibilidade com o

DES convencional, isto pode não influenciar na academia, mas era muito importante comercialmente para a IBM na época (TANENBAUM, 2003).

2.2.2.3 Advanced Encryption Standard

Com o objetivo de substituir o DES, o *Advanced Encription Standard* (AES) foi publicado em 2001 pelo *National Institute of Standars and Technology* (NIST). Quando comparado com algoritmos de chave pública, como o RSA por exemplo, a estrutura do AES é muito mais complexa (STALLINGS, 2008).

A definição do AES veio de um concurso feito pelo NIST, onde o mais votado foi o Rijndael, que aceita chaves e blocos de 128, 192 e 256 bits, destas as mais usadas seriam as de 128 e 256 bits. Assim como o DES, Rijndael utiliza permutações, substituições e rodadas na sua estrutura, sendo 10 rodadas para chaves de 128 bits e 14 rodadas para chaves ou blocos maiores, todavia difere do DES ao utilizar operações com bytes inteiros para melhor eficiência em implementações (TANENBAUM, 2003).

Dependendo da aplicação, o uso de chaves simétricas pode ser complicado, por exemplo a manutenção do sigilo da chave quando se tem muitos destinatários. É justamente para resolver problemas desta natureza que pesquisadores foram em busca de alternativas, como algoritmos que utilizassem duas chaves ao invés de uma.

2.2.3 Algoritmos de chave pública

Em 1976, Diffie e Hellman, pesquisadores da universidade de Stanford propuseram um esquema de criptografia totalmente novo, onde a chave para a criptografia era diferente da chave de decriptografia, e uma não deriva da outra, desta forma os algoritmos chaveados E e D, encriptação e decriptação respectivamente, podem ser declarados da seguinte forma:

a) D(E(P)) = P;

b) é extremamente difícil deduzir D a partir de E;

c) E não pode ser decifrado por um ataque de texto simples escolhido;

A primeira declaração afirma que a função de decriptografar D aplicada em uma mensagem criptografada E(P) gera novamente a mensagem original P, as restantes são autoexplicativas (TANENBAUM, 2003).

Pertencente a esta classificação, temos um algoritmo que possui como nome os respectivos sobrenomes de seus criadores: Rivest-Shamir-Adleman (RSA).

2.2.3.1 RSA

No RSA, segundo Stallings (2008), o texto claro é dividido em blocos, onde cada um destes deve possuir tamanho menor ou igual a $\log_2 n$, para um bloco de texto claro Me um bloco de texto cifrado C, temos:

$$C = M^e \, mod \, n$$

$$M = C^d \, mod \, n = (M^e)^d \, mod \, n = M^{ed} \, mod \, n$$

Ambos emissor e receptor conhecem n. O emissor conhece e e o receptor conhece d, sendo a chave pública $PU = \{e,n\}$ e privada $PR = \{d,n\}$, atendendo os seguintes requisitos:

a) ser possível encontrar e, d, n tais que $M^{ed} \, mod \, n = M$ para todo $M < n$;
b) ser relativamente fácil calcular $M^e \, mod \, n$ e $C^d \, mod \, n$ para todos os valores $M < n$;
c) ser inviável determinar d dados e e n;

Obedecendo estes requisitos, Tanenbaum (2003) traz uma sucinta da utilização do método RSA, baseando-se nestes passos:

a) primeiramente, dois números primos extensos são escolhidos, são eles p e q (geralmente de 1.024 bits);
b) calcula-se $n = p \times q$ e $z = (p - 1) \times (q - 1)$;
c) escolhe-se um número d tal que z e d sejam primos entre si;
d) encontra-se e de forma que $e \times d = 1 \, mod \, z$;

Tendo estas funções calculadas, divide-se o texto em blocos obedecendo a regra e se aplica as funções acima explicitadas, para criptografar um texto M, calcule $C = M^e \, mod \, n$, para descriptografar C, calcule $M = C^d \, mod \, n$. Utilizando-se das chaves definidas anteriormente segundo Stallings (2008) PU e PR.

Uma das aplicações dos algoritmos de chave pública é o reconhecimento de autenticidade através das assinaturas digitais.

2.2.4 Assinaturas digitais

Tanenbaum (2003) traz que em documentos oficiais, impressos, a assinatura firmada é utilizada para reconhecer a autenticidade do mesmo, já para que um documento digital tome o lugar da tinta e papel, é mandatório que esta assinatura não possa ser forjada; partindo deste princípio, é complexa a tarefa de criar um substituto para a assinatura física, tendo em vista que uma das partes deve ser capaz de enviar a mensagem assinada para a outra parte de forma a atender as seguintes regras:

a) o receptor deve ser apto a confirmar a identidade alegada pelo transmissor;
b) em futura auditoria, não deve ser possível ao transmissor repudiar o conteúdo da mensagem;
c) o receptor não deve poder forjar a mensagem;

As assinaturas digitais podem acontecer de três formas, criptografia convencional, utilizando tanto chaves públicas quanto simétricas, através de MAC, código de autenticação de mensagens, ou então *hash* (STALLINGS, 2008).

O uso da criptografia para a assinatura digital através de chave simétrica pode acontecer de duas formas, ou ambas as partes compartilham da chave, onde uma das partes saberá que foi enviado pela outra, todavia, não é possível garantir a autenticação para um terceiro, por conta que as duas partes possuem a chave secreta, por esta razão é sugerido uma autoridade central que garanta essa autenticidade, supõe-se Big Brother (BB), sendo assim, a parte A envia a mensagem criptografada com sua chave secreta e envia para BB, que por sua vez também possui a chave secreta de A, decriptografa a mensagem e a repassa ao destinatário garantindo que foi A quem a enviou, tendo assim validade de autenticidade e não-repúdio (STALLINGS, 2008; TANENBAUM, 2003).

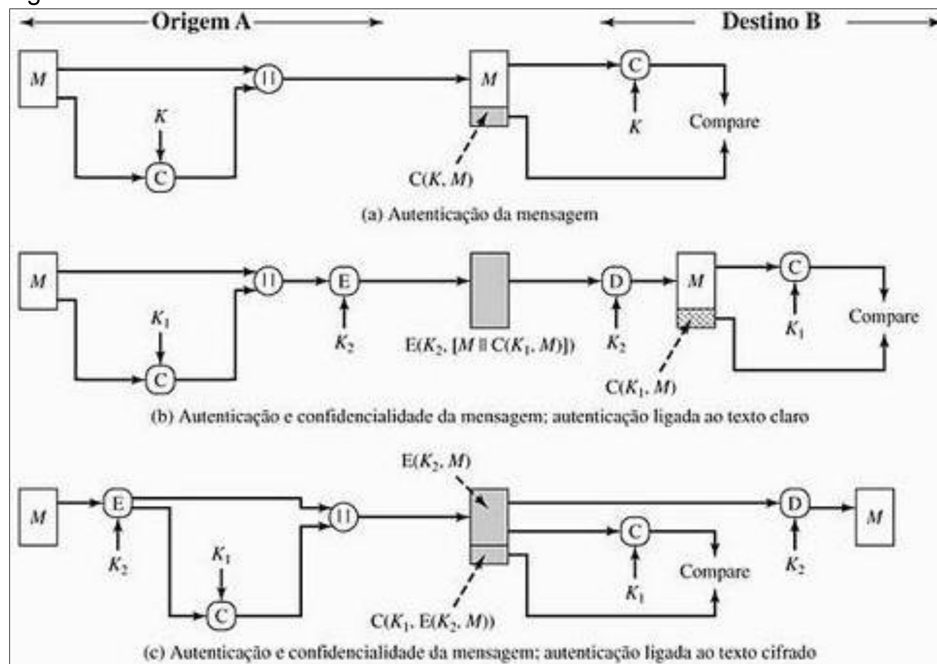
Assim como a simétrica, a criptografia por chave pública também pode ser utilizada para assinar uma mensagem, basta A criptografar a mensagem com sua

chave privada e qualquer destinatário poderá ler utilizando a chave pública de A, tendo a garantia de autenticidade (STALLINGS, 2008).

As formas acima representam o uso de criptografia aplicada na mensagem para a assinar, enquanto o MAC e o *hash* enviam juntamente com a mensagem um sumário de mensagens (*message digest*); que por sua vez, facilita a aplicação para uso juntamente com o envio de texto claro, tendo em vista que a criptografia em toda a mensagem pode gerar um processamento de dados desnecessário, dependendo da aplicação (STALLINGS, 2008; TANENBAUM, 2003).

O MAC também possui o uso de uma chave para criptografar, entretanto, o comprimento do *message digest* (MD) é de tamanho fixo e esta operação não precisa ser reversível. O envio de uma mensagem utilizando esta estratégia consiste na própria mensagem acompanhada do MAC, podendo ou não criptografar a mensagem, tendo ilustrado três possíveis na figura 8 (STALLINGS, 2008).

Figura 8 – MAC



Fonte: Stallings (2008).

A primeira forma na imagem traz o texto claro M juntamente com a aplicação de um algoritmo na mensagem com a chave K para gerar um MAC de tamanho fixo C, onde M e C são enviados juntos da origem A para o destino B e então B gera novamente C com a chave K a partir do texto claro M e compara com C recebido da origem A.

A segunda forma é exatamente igual a primeira, exceto que, antes do envio é utilizado uma nova criptografia com uma chave K_2 e logo após o recebimento é realizado uma descriptografia com esta mesma chave.

A terceira e última forma é baseada no cálculo de C sendo feito em cima da mensagem já criptografada com uma chave K_2.

A última forma de assinatura digital trazida por este trabalho é o uso de hash, onde este, da mesma forma como o MAC, consiste no envio dele simultâneo à mensagem, alternando apenas no fato em que o *hash* não utiliza chave para ser gerado, uma vez que é totalmente unidirecional e obedece às seguintes regras trazidas por Tanenbaum (2003):

a) se P for fornecido, o cálculo de $MD(P)$ será fácil;
b) se $MD(P)$ for fornecido, será efetivamente impossível encontrar P;
c) dado P, outra mensagem que gere o mesmo resultado que $MD(P)$ não pode ser encontrado;
d) qualquer mudança na entrada produz uma saída diferente.

Para atender o terceiro critério, o *hash* deve possuir um mínimo de 128 bits. Para atender o quarto critério, o *hash* deve embaralhar completamente os bits da mensagem (TANENBAUM, 2003).

Quando comentado sobre *hash*, Tanenbaum (2003) traz o MD5 e o SHA-1 como os mais amplamente utilizados.

2.2.5 MD5

Advindo a nomenclatura de ser o quinto sumário de mensagens criado por Ronald Rivest, este algoritmo opera tão complexamente na mistura de seus bits que todos os bits de saída são influenciados por todos os bits de entrada. Sucintamente, o MD5 aumenta o tamanho da entrada para 448 bits e une ao tamanho original da mensagem como um inteiro de 64 bits, gerando desta forma um múltiplo de 512, logo após isto é inicializado um buffer de 128 bits com um valor fixo (TANENBAUM, 2003).

Para os cálculos, é feito ao início de cada rodada, a alocação de um bloco de 512 ao buffer, aliado a isto, também é criado uma tabela a partir da função seno, aspirando uma maior precisão nos cálculos que acontecerão a seguir. O objetivo do

uso de uma função matemática conhecida, como o seno, é para garantir a não existência de algo secretamente escondido pelo criador para uso malicioso, tendo em vista que a IBM, ao criar o DES, não revelou o projeto da caixa S, gerando certa especulação a este respeito. Existem quatro rodadas para cada bloco, tendo em vista que o buffer tem 128 bits e o bloco, 512, desta forma, repete-se até que todos os blocos tenham sido processados. O conteúdo do buffer de 128 bits forma o resultado (TANENBAUM, 2003).

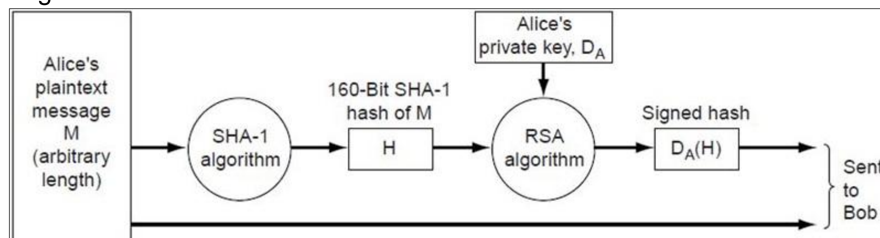
Mesmo vítima de ataques constantes, o MD5 resiste há décadas, foram expostas várias vulnerabilidades tanto de ataques de pré-imagem, que é quando o atacante consegue descobrir a entrada para a geração do *hash*, quanto para ataques de colisão, que é quando duas mensagens geram o mesmo *hash* (FORTE, 2009).

Outra função *hash* de grande importância, alternativa ao MD5, é o SHA-1, abordado a seguir.

2.2.6 SHA-1

Desenvolvido pela *National Security* Agency (NSA) e aprovado pelo NIST no *Federal Information Processing* Standars (FIPS) 180-1, similarmente ao MD5, o SHA-1 utiliza blocos de 512, entretanto, o resultado gerado possui 160 bits. Ilustrado abaixo na figura 9 está o envio de Alice a Bob de uma mensagem assinada e não-secreta, ocorrendo da seguinte maneira: a mensagem é processada pelo SHA-1, formando um *hash* de 160 bits e então é criptografado com sua chave privada e enviado a Bob junto à mensagem clara, como pode ser observado na figura 6 (TANENBAUM, 2003).

Figura 9 – SHA-1



Fonte: Tanenbaum (2003).

Ao receber a mensagem e o *hash* assinado, Bob aplica a chave pública de Alice no *hash* e também o algoritmo SHA-1 na mensagem para validar se os resultados são iguais, se sim, a mensagem é verdadeira (TANENBAUM, 2003).

Sobre o funcionamento do SHA-1, ele inicia preenchendo o fim da mensagem com um bit 1 e o restante em bits 0 até formar um múltiplo de 512, logo a seguir, pega-se um número de 64 bits que representa o tamanho da mensagem (antes do preenchimento) e realiza uma operação OR com os últimos 64 bits. Durante a operação, o SHA-1 mantém cinco variáveis de 32 bits, são nominadas de H_0 a H_4, nestas o *hash* vai se acumulando até obter o resultado final de 160 bits. Os blocos de 512 são chamados de M_0 a M_{n-1}, tendo cada um dezesseis palavras de 32 bits, e agora é o momento de processá-los. Para cada bloco, as 16 palavras são copiadas para o início de um *array* auxiliar de 80 palavras, chamado de W, o restante das palavras neste array é preenchido pela seguinte fórmula:

$$W_i = S^1(W_{i-3}\ xor\ W_{i-8}\ xor\ W_{i-14}\ xor\ W_{i-16})\ (16 \leq i \leq 79)$$

Assim, o cálculo real pode ser expresso em pseudo-C como:

for (i = 0; i<80; i++) {

$temp = S^5(A) + f_i(B,C,D) + E + W_i + K_i;$

$E = D;$

$D = C;$

$C = S^{30}(B);$

$B = A;$

$A = temp;$

}

Onde a função $S^b(W)$ representa a rotação circular à esquerda de W em b bits. As variáveis de A até E são a representação temporária dos valores de H_0 a H_4. K_i são constantes definidas no padrão. As funções f_i são definidas logicamente como:

$f_i(B,C,D) = (B\ e\ C)\ ou\ (não\ B\ e\ D)$ para $(0 \leq i \leq 19)$

$f_i(B,C,D) = B\ xor\ C\ xor\ D$ para $(20 \leq i \leq 39)$

$f_i(B,C,D) = (B\ e\ C)\ ou\ (B\ e\ D)\ ou\ (C\ e\ D)$ para $(40 \leq i \leq 59)$

$f_i(B,C,D) = B\ xor\ C\ xor\ D$ para $(60 \leq i \leq 79)$

Após as 80 iterações, as variáveis de *A* até *E* são somadas aos valores de H_0 a H_4. Repete-se os passos acima para o próximo bloco *M* reiniciando o array

W e mantendo os valores das variáveis H, onde esta última será a saída de 160 bits após todos os blocos serem processados (TANENBAUM, 2003).

3 BITCOIN

Este capítulo foi escrito baseado na publicação de novembro de 2008 do criador do *Bitcoin* utilizando o pseudônimo Satoshi Nakamoto, intitulado *Bitcoin: A Peer-to-Peer Electronic Cash System*, em janeiro do ano seguinte, a moeda começou a funcionar. Este artigo está disponível no site bitcoin.org e é citado neste trabalho com tradução nossa.

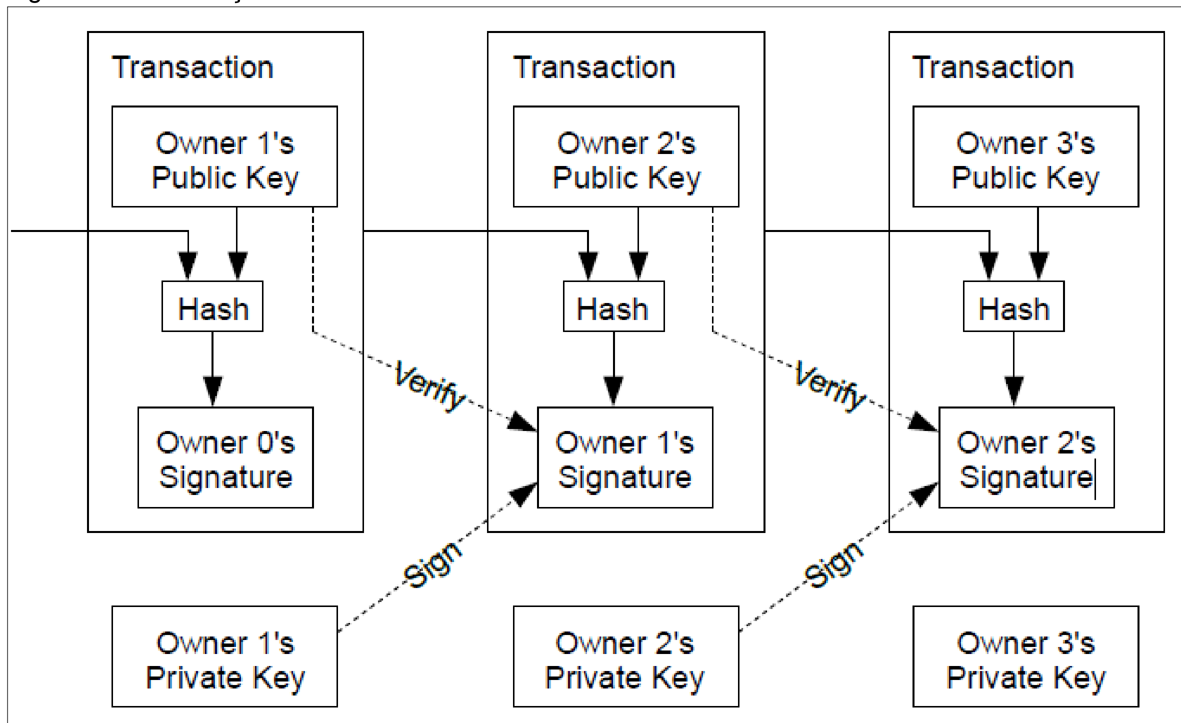
O comércio pela Internet se baseia muito em instituições financeiras intermediando transações de pagamento eletrônico, ao mesmo tempo que este sistema funciona bem para a maioria dos casos, também apresenta sua fraqueza no modelo baseado na confiança. Transações completamente não reversíveis não são possíveis de verdade, sendo que estas instituições mediadoras não podem evitar mediar tais disputas, o custo destas mediações aumenta o custo de cada transação, limitando o valor mínimo a ser praticado, inibindo assim pequenas e casuais transferências, além do fato de ter um custo alto ao não se garantir o pagamento de um serviço que não pode ser revertido. Com a possibilidade de reversão, vem a necessidade da confiança, trazendo assim instituições que de toda forma tentam obter informações dos consumidores, muitas vezes obtendo mais informações do que realmente precisariam. Pelas mesmas razões, é tido como inevitável uma certa porcentagem de perda. Estes custos e incertezas podem ser evitados pessoalmente por uso de moeda corrente física, todavia a rede não oferece mesmo mecanismo sem um terceiro de alta confiança.

O necessário é um meio eletrônico de pagamento baseado em criptografia ao invés de confiança, permitindo duas partes distintas realizarem transações entre si sem a necessidade de um terceiro. Transações que são computacionalmente impraticáveis de reverter protegeriam os vendedores de fraude e um mecanismo de garantia poderia ser facilmente implementado para proteger os compradores. O sistema do *Bitcoin* é seguro enquanto os nós honestos coletivamente possuírem mais poder de processamento que qualquer outro grupo de nós atacantes unidos.

3.1 TRANSAÇÕES

Uma moeda eletrônica é definida por uma corrente de assinaturas digitais, onde cada dono da moeda transfere a mesma para outra pessoa assinando o *hash* da transação anterior e a chave pública do próximo dono, adicionando estes à moeda, desta forma, o receptor da moeda pode verificar as assinaturas para conhecer a cadeia de donos, ilustrado pela figura 10.

Figura 10 – Transação



Fonte: Nakamoto (2008).

Partindo deste princípio, o problema que se pode constatar em um primeiro momento é que o receptor da moeda não consegue comprovar que o antigo dono já não havia repassado esta moeda para outro, isto é chamado de *double-spending*. Uma solução amplamente conhecida para isto é o uso de um terceiro, como a casa da moeda, onde esta é responsável por checar se o dinheiro não foi gasto duas vezes. Neste modo de operação, após cada transação, a moeda volta à fonte e se solicita uma nova moeda, apenas moedas solicitadas direto da casa da moeda podem ser ditas como comprovadamente não gastas duplamente. Com esta solução vem o problema da dependência de uma central controladora de todas as

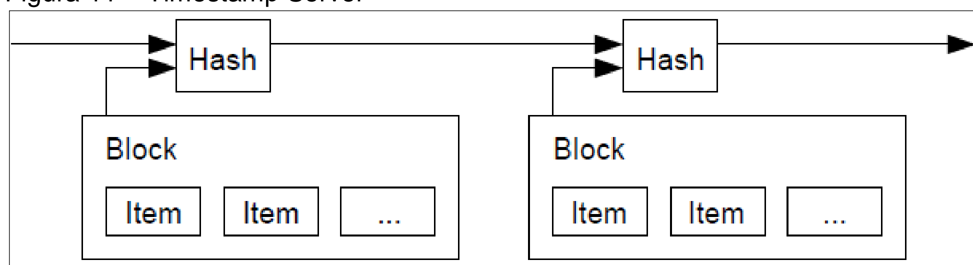
transações monetárias, colocando assim todo o sistema econômico à mercê de uma única entidade, como um banco.

Para resolver este problema é preciso que o receptor da moeda tenha a garantia de que esta não tenha sido gasta anteriormente, para este propósito, basta conhecer a primeira transação da moeda, não é necessário levar em consideração quaisquer tentativas posteriores de gasto duplicado desta moeda. Sendo assim, a única forma de confirmar que realmente não houve nenhuma transação anterior é conhecendo todas as transações, no exemplo da casa da moeda, ela detém todas as transações e poderia decidir por si só aquela que aconteceu antes; para obter isso sem uma central controladora é preciso tornar todas as transações públicas, e complementarmente de um sistema em que os participantes concordem em apenas um histórico de transações, respeitando a ordem cronológica em que foram recebidas. Desta forma, a prova que o receptor precisa em cada transação é dada pela maioria dos nós reconhecendo ela como sendo a primeira transação a ser recebida.

3.2 TIMESTAMP SERVER

A primeira parte da solução oferecida pelo *Bitcoin* é chamado de *timestamp server*. Como ilustrado pela figura 11, o funcionamento deste serviço consiste no cálculo do *hash* de um bloco para ser datado e massivamente publicado, esta data estampada no *hash* prova que o dado existiu naquela ordem para entrar no *hash*, cada entrada possui o *hash* do bloco anterior, formando uma corrente.

Figura 11 – Timestamp Server



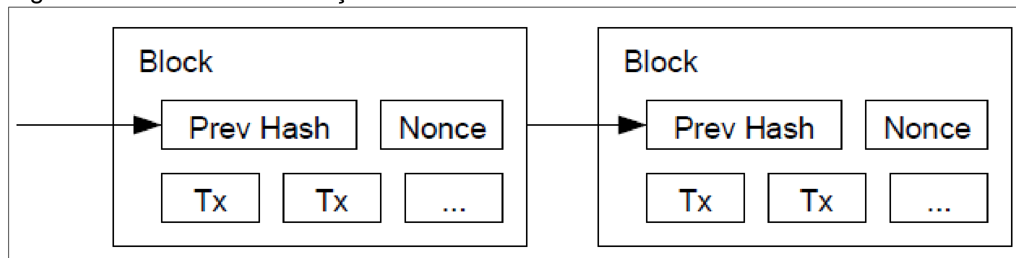
Fonte: Nakamoto (2008).

3.3 PROVA REAL

Para implementar o *timestamp server* é necessário algo similar ao *Hashcash* de Adam Back, ver apêndice A para mais detalhes; onde a prova real envolve a busca de um valor que, ao ser calculado o *hash*, utilizando SHA-256 por exemplo, obtenha-se um *hash* que começa com um certo número de bits zero. O trabalho necessário para este cálculo é exponencialmente relacionado ao número de zeros e pode ser verificado com o simples cálculo de um *hash*.

Para a rede *timestamp* do *Bitcoin*, a prova real é implementada por aumentar o *nonce* do bloco até que o valor encontrado dê o número de zeros necessário. Uma vez que este cálculo seja realizado para atender satisfatoriamente a prova real, este bloco não pode ser alterado sem que todo o trabalho seja refeito, partindo do princípio que todos os blocos seguintes são encadeados a este, a mudança deste acarretaria no recálculo de todos os blocos que se seguiram. Este processo é ilustrado na figura 12.

Figura 12 – Bloco de transação



Fonte : Nakamoto (2008).

A prova real também resolve o problema de determinar a representação da maioria ao tomar a decisão, se a maioria fosse baseada em um-IP-um-voto, alguém poderia alocar vários IPs para si e comprometer a decisão, todavia a prova real é dotada por um-CPU-um-voto, desta forma a decisão da maioria é composta pela maior corrente, a qual possui o maior esforço investido na prova real. Se a maioria do poder de processamento for composta por nós honestos, estes criarão uma corrente honesta mais rapidamente do que algum atacante poderia criar, tendo em vista que, como comentado anteriormente, para modificar um bloco passado se tem que modificar todos os seguintes, isto diminuiria as chances de um atacante exponencialmente e se tornaria computacionalmente impraticável.

Outro ponto importante deste tema é o aumento do poder de processamento conforme a tecnologia computacional evolui, para resolver este problema a dificuldade do cálculo é baseada na velocidade média de geração de blocos por minuto, portanto, se for rápido demais, a dificuldade aumenta.

3.4 REDE DE PROCESSAMENTO

Os passos para rodar uma rede são os seguintes:

a) novas transações são enviadas em broadcast para todos os nós;
b) cada nó coleta as novas transações e as insere em um bloco;
c) cada nó trabalha para processar a prova real do bloco;
d) assim que a prova real é encontrada, o bloco é enviado em broadcast para todos os outros nós;
e) os nós aceitam o bloco recebido apenas se todas as transações não houverem sido gastas anteriormente;
f) os nós aceitam o bloco por começar a processar o próximo bloco utilizando o hash do bloco aceito vinculado como anterior.

Os nós sempre aceitarão a corrente maior como sendo a correta e sempre trabalharão nela para a aumentar. Se dois nós simultaneamente enviarem duas versões diferentes de blocos, pode variar qual bloco cada nó vai receber primeiro, sendo assim, cada um trabalhará naquele qual recebeu primeiro, todavia guardará o outro, para o caso deste outro se tornar o maior, se isso ocorrer, o nó abandona a corrente menor e toma como correta a maior, a qual a maioria do poder de processamento estava trabalhando.

O *broadcast* de novas transações não necessariamente precisam alcançar a todos os nós, bastando que alcance um número considerável, elas entrarão na cadeia logo. Os blocos também são tolerantes a perdas, se um nó não recebeu um bloco, ele irá notar quando receber o próximo e o irá solicitar para completar sua corrente.

Tendo isto em vista, qual razão um nó teria para realizar todos estes cálculos de geração de blocos que requerem gastos em energia e investimentos em novas tecnologias? A resposta para esta pergunta está no incentivo dado ao gerar novas moedas.

3.5 GERAÇÃO DE MOEDAS

Por convenção, a primeira transação em um bloco é especial, esta começa uma nova moeda que pertence ao gerador do bloco. Este fator gera um incentivo para os nós continuarem dando subsídio à rede, tanto quanto cria um modo de inicialmente gerar moedas para circularem, tendo em vista que não existe uma autoridade central para gerar tais moedas. A adição constante de novas moedas é análogo aos recursos gastos pelos mineradores, em que, neste caso são recursos como eletricidade e tempo de processamento.

Uma outra maneira de incentivo são taxas de transação. Se o valor de saída da transação é menor que o de entrada, têm-se uma taxa de transação, esta é adicionada ao valor de incentivo do bloco em que ela está contida. Uma vez que o número de moedas pré-determinado está inteiramente em circulação, o incentivo se tomará inteiramente das taxas de transação e o sistema se torna livre de inflação.

Este incentivo ajuda a encorajar os nós a se manterem honestos, uma vez que um atacante ganancioso é capaz de controlar mais do que a metade do poder computacional total, ele terá de escolher entre fraudar o sistema e tomar de volta seus pagamentos ou então usar este poder para minerar novas moedas, desta forma ele poderá encontrar a maneira honesta mais lucrativa e criar mais moedas do que todo o resto somado ao invés de comprometer todo o sistema, afinal, ele perderia todas as moedas que possui.

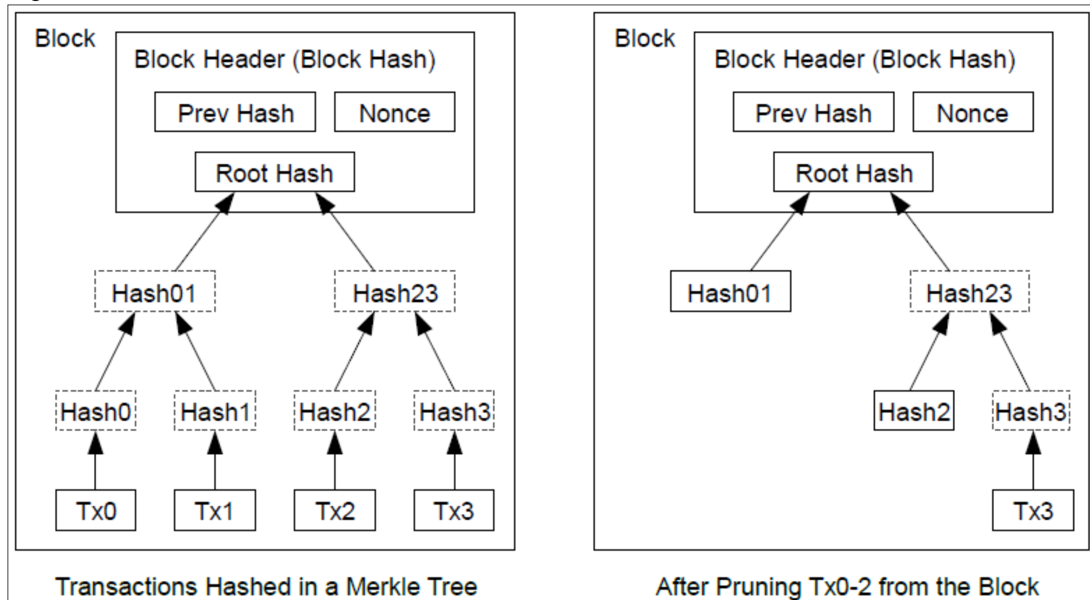
Mas todo esse incentivo compensa o investimento que precisa ser feito? Todos estes blocos não vão se acumular infinitamente gerando uma altíssima carga de armazenamento? Logicamente, é necessário existir uma estratégia para economizar o espaço em disco.

3.6 RECUPERANDO ESPAÇO EM DISCO

Uma vez que a última transação de uma moeda está enterrada abaixo de suficientes blocos, as transações gastas anteriormente a esta podem ser descartadas para economizar espaço em disco. Para facilitar o processo sem quebrar o *hash* do bloco, estas transações são armazenadas em uma *Merkle Tree*,

com apenas a raiz incluída no bloco do *hash*. Blocos mais antigos podem ser compactados arrancando ramos da árvore, os *hashes* interiores não precisam permanecer armazenados, como na figura 13.

Figura 13 – Markle Tree



Fonte: Nakamoto (2008).

Um cabeçalho de bloco (*block header)* sem transação será de aproximadamente 80 bytes, supondo que blocos são gerados a cada 10 minutos, são 80 bytes multiplicados por 6 porções de 10 minutos em uma hora, multiplicado por 24 horas no dia, multiplicado por 365 dias no ano, temos 4,2 MB por ano. Com a memória RAM dos computadores atuais isso não viria a ser um problema nem se os blocos precisassem ficar na memória volátil.

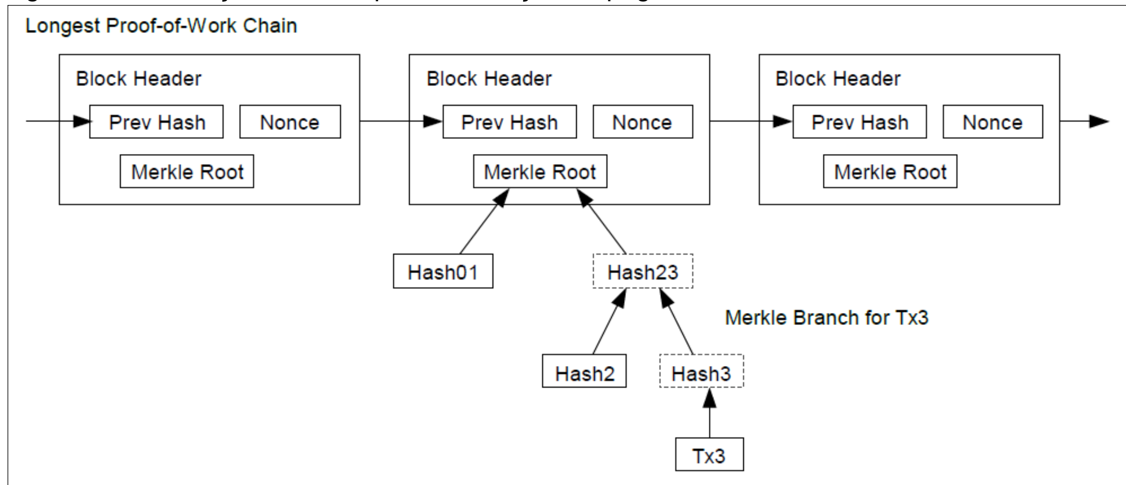
Mesmo assim, existe uma simplificação para a varredura da rede, não sendo viável e tampouco necessário percorrer todos os blocos para se verificar um pagamento.

3.7 VERIFICAÇÃO DE PAGAMENTO SIMPLIFICADA

Para verificar um pagamento, basta manter uma cópia dos cabeçalhos dos blocos da maior corrente de prova-real, que por sua vez pode ser obtida navegando na rede até que o nó esteja convencido de possuir a mais longa corrente, obtendo assim o ramo *Markle* relacionando a transação ao bloco a qual

está inserida. Não se pode verificar a transação em si, mas encontrar sua posição na corrente, verificando que o nó a aceitou como verdadeira e os blocos subsequentes também a confirmam como verdadeira, ilustrado na Figura 14.

Figura 14 – Obtenção do ramo para verificação do pagamento



Fonte: Nakamoto (2008).

Levando em consideração que a honestidade dos nós é de extrema importância para verificar o pagamento, uma das estratégias para prevenir atacantes de forjar o pagamento ao controlar maior parte do poder de processamento da rede é aceitar avisos de outros nós sobre problemas ou inverdades nos blocos, sugerindo ao usuário que baixe o bloco completo e verifique a inconsistência. Estabelecimentos que tratam frequentemente com o recebimento de transações provavelmente optarão por possuir seus próprios nós com uma verificação independente para maior segurança e rápida verificação.

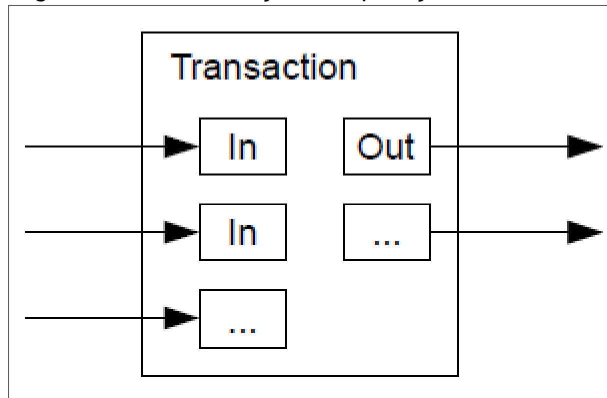
Também podem existir pagamentos que foram originados de várias transações, desta forma foi desenvolvido uma maneira de reagrupar estas transações para que não sejam necessários novas múltiplas transações para repassar essas moedas para um novo dono.

3.8 COMBINANDO E SEPARANDO VALORES DE MOEDAS

Mesmo que possível tratar individualmente cada moeda, seria intratável realizar uma transação para cada centavo de moeda. Para possibilitar essa fragmentação, as transações possuem múltiplas entradas e saídas, ilustrado na

figura 15. Geralmente as transações possuem como entrada uma grande ou várias menores combinando pequenas quantias, quanto às saídas, são normalmente duas: uma para o pagamento e outra para o troco, que se houver, volta ao remetente.

Figura 15 – Combinação e separação de moedas



Fonte: Nakamoto (2008).

É importante tomar nota de que uma transação que depende de muitas outras e estas ainda de muitas outras não é um problema, tendo em vista que nunca será necessário extrair uma cópia do histórico independente de uma única transação.

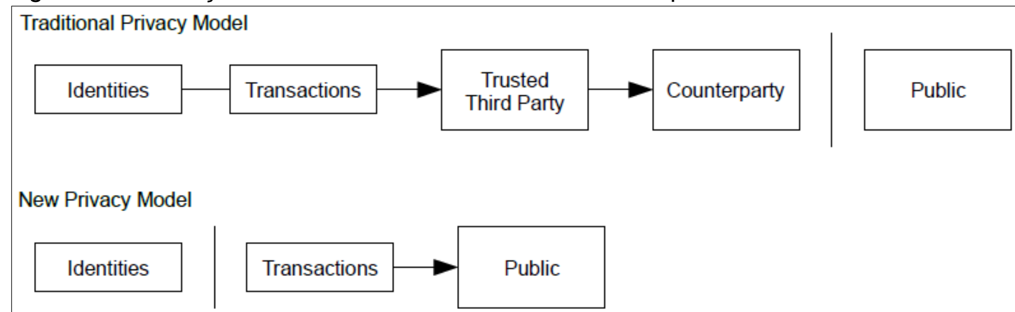
Mesmo com todas essas combinações e a publicação das transações, não necessariamente significa que a privacidade das partes será comprometida, ela ainda pode permanecer anônima, como explicado a seguir.

3.9 PRIVACIDADE

O modelo de finanças tradicional dos bancos atinge seu desejado nível de privacidade por manter as transações privadas entre as partes envolvidas e uma terceira confiável e reguladora. A necessidade da publicação das transações para todos verem é nativo deste método, todavia não é de mesma natureza perder a privacidade, ela pode ser mantida por tornar as chaves públicas anônimas, onde o público em geral pode ver que a quantia foi de uma parte para outra, mas sem que essa transação esteja diretamente relacionada à uma identidade. Esta relação é um pouco parecida com o que acontece na divulgação de informação da bolsa de valores, onde o tempo e o tamanho das trocas individuais é publicado, mas sem

dizer quem realizou tais trocas. A relação entre o modelo tradicional e o novo modelo é ilustrado respectivamente na figura 16.

Figura 16 – Relação entre modelo tradicional e novo de privacidade



Fonte: Nakamoto (2008).

Ainda temos uma importante parte relacionado ao ataque de maior poder de processamento, como ele se relaciona com os nós honestos e o que exatamente o atacante pode fazer, existem algumas limitações que serão abordadas a seguir.

3.10 CÁLCULOS

Considerando um cenário onde um atacante tenta gerar uma cadeia alternativa de blocos mais rapidamente que os nós honesto; mesmo que obtenha sucesso, não é possível realizar mudanças arbitrárias no sistema, como criar moedas do nada ou pegar uma moeda que não era originalmente sua, os nós simplesmente não aceitarão transações inválidas como pagamento e nós honestos de maneira nenhuma aceitarão blocos contendo tais transações como verdadeiros. O que o atacante pode realmente fazer de maneira maliciosa é tomar de volta alguma transação feita por ele e gastar esta moeda novamente em outra transação.

A corrida entre a corrente honesta e a corrente atacante pode ser expressa como um binômio de caminho randômico, onde o evento de sucesso é a corrente honesta se estender por um bloco, incrementando sua liderança em +1, a situação de falha é a corrente atacante se entender por um bloco, diminuindo a distância em -1.

A probabilidade do atacante alcançar o topo a partir de um déficit é análoga ao problema *Gambler's Ruin*, ou A Ruína do Apostador. Supondo que um apostador com crédito ilimitado começa em déficit e joga potencialmente um número

infinito de tentativas para tentar alcançar um empate; pode-se calcular as chances deste apostador alcançar tal empate, ou que este atacante alcance a corrente honesta, seguindo abaixo:

$p =$ probabilidade de um nó honesto encontrar o próximo bloco.

$q =$ probabilidade do atacante encontrar o próximo bloco.

$q_z =$ probabilidade de que o atacante alcance o topo de z blocos atrás.

$$q_z = \begin{Bmatrix} 1 & if\ p \leq q \\ (q/p)^z & if\ p > q \end{Bmatrix}$$

Assumindo que $p > q$, a probabilidade cai exponencialmente a medida que os blocos que o atacante tem de alcançar aumentam. Com as chances contra ele, se ele não der uma aposta certeira logo, suas chances serão esmagadas a medida que ele fica cada vez mais para trás.

Deve-se agora considerar o quanto um novo beneficiário de uma nova transação tem de esperar antes que esteja suficientemente certo de que o remetente não pode alterar a transação. Assume-se que o remetente é um atacante que deseja fazer acreditar que realizou o pagamento ao receptor por um tempo, entretanto logo deseja retomar esta transação para si, o receptor será alertado sobre este fato e o remetente espera que seja tarde demais.

O remetente pode preparar toda a corrente de blocos antes do tempo trabalhando incansavelmente até que tenha a sorte de estar suficientemente a frente e então; neste momento, executar a transação, para impedir isto, o receptor pode criar um novo par de chaves e dar ao remetente a chave publica logo antes de receber a transação. Então, só neste momento o atacante pode começar a processar em segredo uma corrente paralela onde existe uma transação alternativa à enviada para tomar ela de volta.

O beneficiário espera até que a transação seja adicionada a um bloco e z blocos a frente se tenham relacionado a este. Ele não sabe o quanto de progresso o atacante possui, todavia assumindo que os nós honestos tiveram o progresso médio esperado por bloco, o potencial do atacante será uma distribuição de Poisson com o valor esperado a seguir:

$$\lambda = z\frac{q}{p}$$

Para definir a probabilidade de que o atacante ainda consiga alcançar o topo agora, multiplica-se a densidade de Poisson por cada quantidade de progresso

que ele possa já ter realizado pela probabilidade de que ele possa alcançar o topo daquele momento:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} . \begin{Bmatrix} (q\ /\ p)^{(z-k)} & se\ k\ \leq z \\ 1 & se\ k > z \end{Bmatrix}$$

Modificando para evitar somar a infinita calda da distribuição, temos:

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - (q/p)^{(z-k)}\right)$$

Convertendo para código C, temos:

```
#include <math.h>
double AttackerSuccessProbability(double q, int z) {
   double p = 1.0 - q;
   double lambda = z * (q / p);
   double sum = 1.0;
   int i, k;
   for (k = 0; k <= z; k++) {
      double poisson = exp(-lambda);
      for (i = 1; i <= k; i++)
         poisson *= lambda / i;
      sum -= poisson * (1 - pow(q / p, z - k));
   }
   return sum;
}
```

Após executar e obter alguns resultados, Nakamoto comenta que se pode ver a probabilidade despencar exponencialmente com z:

```
q=0.1
z=0 P=1.0000000
z=1 P=0.2045873
z=2 P=0.0509779
z=3 P=0.0131722
z=4 P=0.0034552
z=5 P=0.0009137
z=6 P=0.0002428
z=7 P=0.0000647
z=8 P=0.0000173
z=9 P=0.0000046
z=10 P=0.0000012
q=0.3
z=0 P=1.0000000
z=5 P=0.1773523
z=10 P=0.0416605
z=15 P=0.0101008
z=20 P=0.0024804
z=25 P=0.0006132
```

z=30 P=0.0001522
z=35 P=0.0000379
z=40 P=0.0000095
z=45 P=0.0000024
z=50 P=0.0000006

Resolvendo para p menor que 0.1%:

P < 0.001
q=0.10 z=5
q=0.15 z=8
q=0.20 z=11
q=0.25 z=15
q=0.30 z=24
q=0.35 z=41
q=0.40 z=89
q=0.45 z=340

3.11 CONCLUSAO DE NAKAMOTO

Ao fim do artigo o autor escreve um resumo geral do que foi abordado no mesmo e de quais conclusões foram alcançadas a partir de seu estudo.

Foi proposto um sistema eletrônico sem dependência de confiança. Começou-se com o *framework* usual de moedas feitas de assinaturas digitais, o qual prevê um forte controle de posse, todavia incompleto sem um meio de prevenir o duplo-gasto. Para resolver este problema, foi proposto uma rede *peer-to-peer* usando prova real para gravar um histórico público de transações, que rapidamente se torna computacionalmente impraticável para o atacante modificar este histórico se a maioria do poder de processamento estiver contido em nós honestos. A rede é robusta em sua simplicidade não estruturada. Nós trabalham simultaneamente com pouca coordenação. Não precisam ser identificados, partindo do princípio que as mensagens não são roteadas para nenhum lugar em particular, só precisam ser entregues baseadas no melhor esforço. Nós podem sair e reentrar na rede a qualquer momento, bastando aceitar a prova real da corrente como sendo prova do que aconteceu durante a sua ausência. Vota-se com seu poder de CPU, expressando sua aceitação de blocos válidos por trabalhar em estendê-los e rejeitando blocos tidos como inválidos por recusar trabalhar nestes. Qualquer regra e incentivo podem ser estimulados utilizando este mecanismo de consenso.

4 TRABALHOS CORRELATOS

Neste capítulo, observar-se-á alguns trabalhos que possuem temas parecidos com o abortado neste documento, que foram publicados em veículos acadêmicos públicos.

Começando pelo curso de ciência da computação da UNESC, temos o trabalho da acadêmica Juliana da Luz de Campos, de 2008, intitulado Avaliação de Desempenho dos Algoritmos de Criptografia: Data Encryption Standard e Algoritmo Assimétrico (RSA). Este trabalho inicia com uma introdução sobre a segurança de dados e logo a seguir discorre sobre criptografia, mais especificamente através de alguns exemplos de cifras. Logo adentra ao tema das cifras simétricas, citando suas vantagens e desvantagens, seguido da explanação de seus objetos de estudo: DES e RSA, explanando mais profundamente seu funcionamento. Por fim, utiliza o software JR Cripto versão 1.0 para medir os tempos dos algoritmos DES e RSA, demonstrando seus resultados e comentando sobre os tempos. Sucintamente, a conclusão obtida por este trabalho foi que o algoritmo simétrico DES é muito mais rápido de processar do que o assimétrico RSA, todavia menos seguro; outra conclusão de tamanha importância foi que os testes com chaves maiores não aumentaram tanto o tempo de processamento, mas a segurança aumenta consideravelmente.

O segundo trabalho foi publicado em 2015 na Universidade Federal de Santa Catarina, por Ismael Cabral, intitulado Segurança da Informação em Bibliotecas Universitárias Federais: Um levantamento sobre ferramentas e técnicas utilizadas. Iniciando com um breve histórico de segurança da informação e sobre seus princípios, segue para algumas políticas a serem utilizadas para melhorar a segurança, como senhas e backups, também assinalando as ameaças à segurança e algumas ferramentas de apoio que podem ser utilizadas. Por fim, apresenta através de uma metodologia caracterizada como bibliográfica e documental, descritiva e de caráter quali-quantitativo, seus resultados obtidos através de questionários enviados à 27 bibliotecas das universidades federais brasileiras, tendo 21 respostas, sua pesquisa mostrou que a tecnologia de segurança está a pouco se desenvolvendo e que a maioria das bibliotecas ainda possui o mínimo de segurança em sua rede.

O terceiro trabalho a ser descrito foi publicado na revista Eventos Pedagógicos em dezembro de 2012, com autoria de Rafael Santos Andrade, defendido em 2009 na Universidade Estadual do Sudoeste da Bahia e é intitulado Algoritmo de Criptografia RSA: análise entre a segurança e velocidade. Inicia trazendo um breve embasamento nas primeiras cifras e logo inicia sobre o RSA e seu funcionamento. A metodologia do trabalho foi a implementação do RSA na linguagem C, gerando aleatoriamente 20 chaves para os testes, cifrando e decifrando 50 mensagens de 100 caracteres aleatórios, logo após, outro teste, dessa vez aumentando a mensagem, tendo 102 caracteres para um módulo de 1024 bits, 205 caracteres para um módulo de 2048 bits e 410 caracteres para um módulo de 4096 bits. Diferentemente do primeiro trabalho comentado aqui, os testes deste trabalho se basearam no aumento do módulo da chave, e não do texto, desta forma foi obtido um resultado exponencial de aumento conforme o tamanho do módulo aumentou.

O quarto trabalho foi enviado em 2014 e aceito em 2015 pela *Wiley Periodicals*, com autoria de Noam Goldberg, Sven Leyffer e Ilya Safro, intitulado *Optimal Response to Epidemics and Cyber Attacks in Networks*, citado com tradução nossa. Inicia demonstrando matematicamente os modelos de espalhar vírus de computadores motivados pela epidemiologia, então apresenta modelos de otimização de respostas de rede determinísticos. Logo a seguir é realizado uma análise preliminar e são apresentadas técnicas computacionais. Tendo por fim a apresentação de seus resultados, sendo feito seu experimento em uma rede randômica gerada através de um modelo gráfico Erdós-Rényi, para cada rede foram gerados 10 diferentes valores para suas variáveis e seus resultados foram apresentados em tabelas. Foi proposto uma formulação probabilística para responder ataques cibernéticos e biológicos à redes, problemas com otimizações similares foram levados em consideração, todavia com uma gama diferente de restrições que falta à motivação de frameworks probabilísticos que são considerados acerca deste assunto. Ao final do trabalho é provado a proposição feita.

5 ANÁLISE DOS ALGORITMOS

Aborda-se aqui o desenvolvimento do protótipo, as tecnologias utilizadas, bem como os resultados obtidos e suas amostragens estatísticas.

5.1 AMOSTRAGEM ESTATÍSTICA

A teoria da amostragem é o estudo onde torna possível a obtenção de dados de apenas uma amostra da população total e seus resultados estatísticos representem significativamente o todo envolvido. Uma das suas utilidades é determinar se a diferença no resultado obtido foi uma casualidade ou realmente é algo significativo (SPIEGEL; STEPHENS, 2009).

5.2 METODOLOGIA

Primeiramente através de um levantamento bibliográfico sobre a segurança de dados, partindo logo após para um aprofundamento em criptografia, com ênfase nos principais algoritmos de criptografia. O segundo capítulo aborda o artigo publicado pelo criador do *Bitcoin*, onde descreve detalhadamente o funcionamento da moeda. Por fim, este trabalho implementa testes de algoritmos utilizados no funcionamento da moeda, utilizando a linguagem java, e então alcança seu objetivo de medir estes tempos e mostrar as estatísticas dos resultados obtidos.

Para a pesquisa bibliográfica foram usados artigos publicados em revistas nacionais e internacionais, livros, teses, trabalhos de conclusão de curso e demais pesquisas que possuem valor para agregar ao trabalho.

Para a obtenção dos resultados acima foram aplicados diversos testes utilizando o software IBM *Statistical Package for the Social Sciencies* (SPSS) versão 21, os resultados explanados de cada teste podem ser encontrados mais a seguir neste capítulo.

Para a montagem das tabelas de resultados, o primeiro teste a ser aplicado é da estatística descritiva, obtendo dados como média, desvio padrão, valor mínimo, valor máximo e outros, junto a este, foi obtida a tabela de percentis, para poder visualizar como a distribuição cresce conforme é executada.

Após a obtenção destes resultados e a montagem da tabela, foi realizado o teste de normalidade, este teste consiste em verificar se a distribuição se desvia de uma distribuição normal, os testes de Kolmogorov-Smirnov e de Shapiro-Wilk fazem justamente isso, comparam escores de uma amostra com os de uma distribuição normal modelo com a mesma média e variância dos valores da amostra. Se o teste é não significativo ($p > 0,05$), significa que os dados não diferem de uma distribuição normal. (FIELD, 2009).

Tais desvios de normalidade informam que não se pode utilizar um teste paramétrico, pois a hipótese de normalidade não se verifica. Desta forma, pode-se considerar testes não paramétricos como forma de verificar a hipótese de interesse (FIELD, 2009).

Os testes não paramétricos irão informar se existe diferença significativa entre os grupos de variávels quantitativas, neste trabalho é utilizado o teste U de Mann-Whitney quando se tem 2 amostras independentes e o teste H de Kruskal-Wallis quando há 3 amostras independentes (DEVORE, 2006; ZAR, 2010).

5.2.1 Tecnologia

Para facilitar o desenvolvimento do protótipo, foi utilizado um gerador baseado no Yeoman, denominado JHipster, este, por sua vez utiliza em sua camada superior AngularJS e por trás Java, suportado por um serviço Spring-boot, diferenciando-se dos servidores de aplicação comuns por sua arquitetura leve e facilidade de manuseio. O banco de dados utilizado foi o PostgreSQL e a conexão com o banco para conversão para .CSV feita pelo software SQuirreL. Todas as tecnologias citadas acima são de distribuição gratuita e algumas de código aberto.

Afim de melhor demonstração estatística, para a amostragem final foi utilizado o software da IBM denominado SPSS, onde este possui período de testes gratuito suficiente para realização dos cálculos necessários para este trabalho.

5.2.2 Protótipo

A configuração do banco de dados e instalação de programas de suporte secundários não serão abordados por conta de existirem tutoriais oficiais dos fabricantes de como realizar as instalações e configurações dos mesmos.

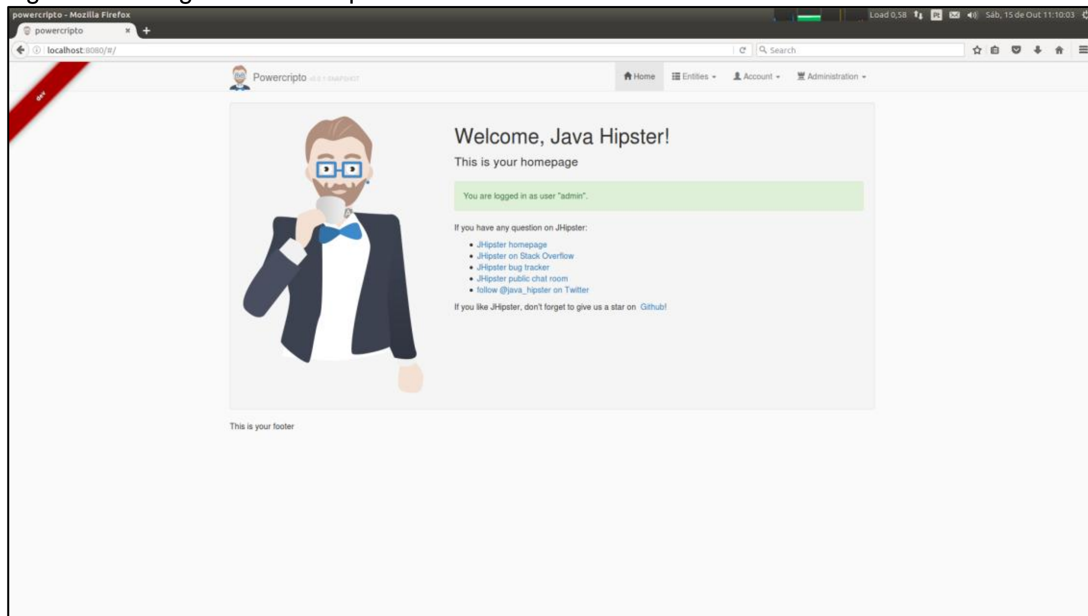
Após desenvolvido e configurado, o primeiro passo para utilizar o protótipo é subir a aplicação utilizando o *Spring-boot*, através do comando Maven “mvn spring-boot:run” como demonstrado na figura 17, a partir disto, informa-se no endereço do navegador o endereço informado pelo console.

Figura 17– Console *Spring-boot*

Fonte: gerado a partir do JHipster.

Por se tratar de um protótipo, foi rodado localmente, o que não impede a configuração para acesso via web normalmente. Desta forma basta abrir um navegador web qualquer e acessar a porta 8080 da máquina local, abrindo a tela demonstrada na figura 18 no menu “*Entities”* existe o item “*Cripto*” onde este abrirá a tela de cadastro, como explanado a seguir.

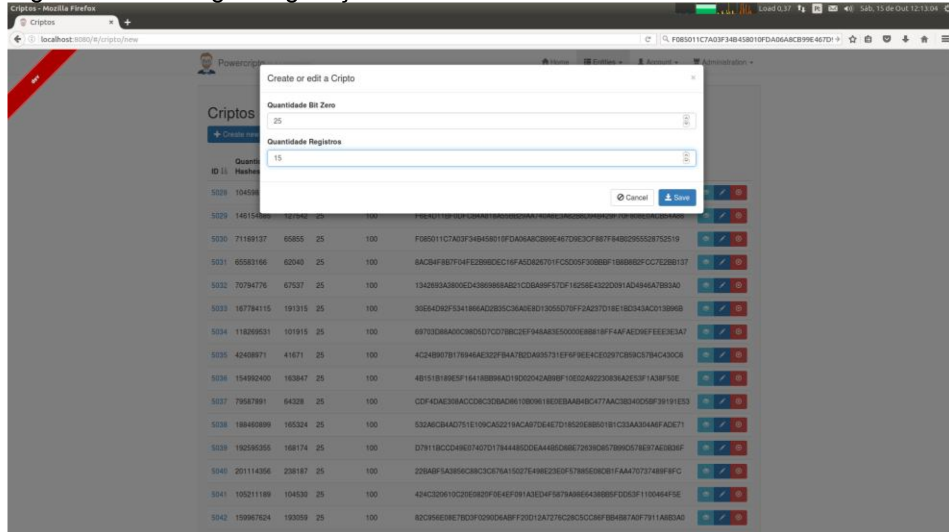
Figura 18 – Página inicial JHipster



Fonte: gerado a partir do JHipster.

Quando aberto, o diálogo de geração aparecerá sobre a tela como demonstrado na figura 19, nele é solicitado a quantidade de bits zeros ao fim do hash que será necessário e a quantidade de hashes que terão de ser encontrados com esta quantidade de bits zeros, quando salvo, iniciam-se os cálculos.

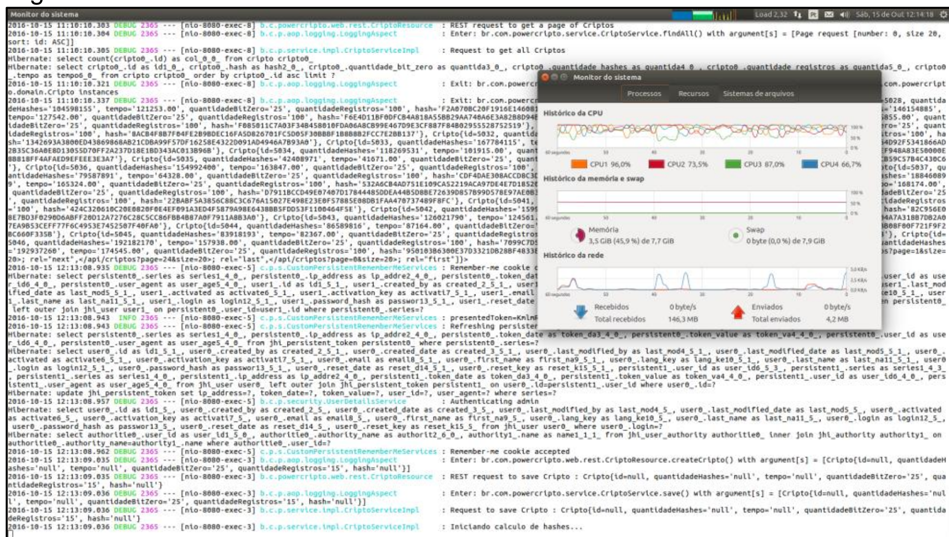
Figura 19 – Diálogo de geração



Fonte: gerado a partir do JHipster.

Na figura 20 podemos ver pelo console que o cálculo de *hashes* foi iniciado, e pode ser comprovado pelo monitor de recursos do computador onde demonstra que todos os núcleos do processador estão sendo utilizados, assim se pode verificar como estes cálculos estão sendo feitos.

Figura 20 – Processamento de dados



Fonte: gerado a partir do JHipster.

Na figura 21 se tem a maneira utilizada para iterar um *array* de bytes, onde este array é o *hash* calculado, afim de poder verificar o número de zeros ao fim do hash. Esta classe java é a implementação de um *iterator* da mesma forma como já funciona o *iterator* nativo das *Collections* java, com os métodos *next()* e *hasNext()*, respectivamente servindo para pegar o valor do próximo bit e para verificar se existe um próximo bit no *array*.

Figura 21 – Iteração de bits

```java
package br.com.powercripto.service.util;

import java.util.Iterator;

/**
 * Diego 03/08/16.
 * Originally taken from StackOverflow website
 */
public class ByteArrayBitIterable {
    private final byte[] array;

    public ByteArrayBitIterable(byte[] array) {
        this.array = array;
    }

    public Iterator<Boolean> iterator() {
        return new Iterator<Boolean>() {
            private int bitIndex = 0;
            private int arrayIndex = 0;

            @Override
            public boolean hasNext() { return (arrayIndex < array.length) && (bitIndex < 8); }

            @Override
            public Boolean next() {
                Boolean val = (array[arrayIndex] >> (7 - bitIndex) & 1) == 1;
                bitIndex++;
                if (bitIndex == 8) {
                    bitIndex = 0;
                    arrayIndex++;
                }
                return val;
            }

        };
    }

}
```

Fonte: do autor.

Na figura 22 se tem a classe "*CriptoUtil*" que por sua vez utiliza a classe demonstrada na figura 21 em um de seus métodos e possui outros de uso essencial para o processamento, os métodos são: "verificaQuantidadeBitsZero", "gerarNovoHash" e "bytesToHex".

Figura 22 – Classe de utilidades

```
package br.com.powercripto.service.util;

import ...

/**
 * Diego 03/08/16.
 */
public class CriptoUtil {

    public static boolean verificaQuantidadeBitsZeros(byte[] b, Long quantidadeBitsZeros) {
        Long quantidadeIteracoes = 0L;
        Iterator<Boolean> iterator = new ByteArrayBitIterable(b).iterator();
        while (iterator.hasNext()) {
            Boolean bit1 = iterator.next();
            if (bit1) {
                return true;
            } else {
                quantidadeIteracoes++;
                if (quantidadeIteracoes >= quantidadeBitsZeros) {
                    return false;
                }
            }
        }

        return true;
    }

    public static byte[] gerarNovoHash() throws NoSuchAlgorithmException {
        byte[] b = new byte[300];
        new Random().nextBytes(b);

        return MessageDigest.getInstance("SHA-256").digest(b);
    }

    final protected static char[] hexArray = "0123456789ABCDEF".toCharArray();
    public static String bytesToHex(byte[] bytes) {
        char[] hexChars = new char[bytes.length * 2];
        for ( int j = 0; j < bytes.length; j++ ) {
            int v = bytes[j] & 0xFF;
            hexChars[j * 2] = hexArray[v >>> 4];
            hexChars[j * 2 + 1] = hexArray[v & 0x0F];
        }
        return new String(hexChars);
    }
}
```

Fonte: do autor.

O Método "verificaQuantidadeBitsZeros" utiliza o *iterator* mostrado anteriormente para percorrer o *hash* gerado e validar se possui a quantidade de zeros necessária, retornando verdadeiro se não possuir e falso se possuir.

O método "gerarNovoHash" gera uma mensagem randômica de 300 bytes e gera um *hash* com base nesta mensagem utilizando a classe *MessageDigest* nativa do java 8, onde no caso da figura 21 é utilizado o algoritmo SHA-256.

O método "bytesToHex" é uma simples conversão de um *array* de bytes para um número hexadecimal para melhor visualização.

Na figura 23 é onde são ordenados os cálculos dos *hashes*, mantendo a *thread* calculando *hashes* enquanto o número de zeros não é alcançado. A partir deste momento, aborda-se as análises após os cálculos aqui explicados.

Figura 23 – Classe de cálculo

```java
package br.com.powercripto.service.util;

import ...

/**
 * Diego 03/08/16.
 */
public class ThreadHashCalculation extends Thread {
    private final Logger log = LoggerFactory.getLogger(CriptoUtil.class);

    private String hash;
    private Long quantidadeBitZero;
    private Long quantidadeHashesCalculado = 0L;

    private volatile boolean threadRunning = true;

    public ThreadHashCalculation(Long quantidadeBitZero) { this.quantidadeBitZero = quantidadeBitZero; }

    public String getHash() { return hash; }

    public void setHash(String hash) { this.hash = hash; }

    public boolean isThreadRunning() { return threadRunning; }

    @Override
    public void run() {
        try {
            byte[] digest;
            do {
                digest = CriptoUtil.gerarNovoHash();
                quantidadeHashesCalculado++;
            } while (CriptoUtil.verificaQuantidadeBitsZeros(digest, quantidadeBitZero) && threadRunning);


            if (!CriptoUtil.verificaQuantidadeBitsZeros(digest, quantidadeBitZero)) {
                hash = CriptoUtil.bytesToHex(digest);
            }

            threadRunning = false;
        } catch (NoSuchAlgorithmException e) {
            log.error(e.getMessage(), e);
        }
    }

    public Long stopThread() {
        threadRunning = false;
        return quantidadeHashesCalculado;
    }
}
```

Fonte: do autor.

5.2.3 Análises estatísticas com o SPSS

Logo após os dados serem gerados e capturados, foi utilizado o software SQuirreL para conectar com o banco de dados e converter os dados capturados para .CSV, desta forma, foi usado o Libre Office Calc versão 5.1.4.2 para manipular os dados em .CSV e fazer o transporte para o SPSS versão 21, quando pertinente também foi utilizado o formato dados .xlsx através do Microsoft Office Excel 2007 devidamente licenciado nos computadores da UNESC, apenas por maior comodidade por conta de nem todas as máquinas da universidade possuírem previamente instalado o Sistema Operacional Linux.

Uma vez que as variáveis foram criadas no software IBM SPSS versão 21, os dados transportados e alimentados no software, pode-se iniciar as análises descritas no início do item 5.2 deste trabalho.

Os testes foram divididos em dois tipos, quanto a quantidade de hashes necessária para alcançar o resultado e quanto ao tempo de processamento para alcançar o mesmo. Após esta divisão será abordado a subdivisão por algoritmo e pela parametrização da quantidade de zeros solicitado pelo problema, conforme a regra de Nakamoto.

A inicio, temos as análises de estatística descritiva, iniciando pela quantidade de *hashes* entre os algoritmos com parametrização de 20 zeros:

Tabela 1 – Resumo do processamento de caso para quantidade de hashes

Algoritmo	N válidos	% válidos	N ausentes	% ausentes	N Total	% total
SHA-256	1000	100%	0	0%	1000	100%
SHA-1	1000	100%	0	0%	1000	100%
MD5	1000	100%	0	0%	1000	100%

Fonte: do autor.

De acordo com a tabela 1, absorve-se que foram calculados *N* de 1000 para cada algoritmo quanto a quantidade de *hashes* calculados e não foi obtido nenhum ausente, tendo aproveitamento de 100% de *N* válidos, onde *N* é o número de amostras para cada distribuição.

Por ser exatamente igual, as demais tabelas de resumo de processamento de caso serão omitidas, incluindo as de parametrizações diferentes, as conclusões são exatamente a mesma acima.

Inicia-se então a demonstração dos resultados dos testes descritivos na tabela 2 a seguir. Salienta-se que todas as tabelas abaixo descritas que contemplam o tempo de processamento estão descritas em segundos.

Tabela 2 – Descritivos para quantidade de hashes com 20 zeros

Algoritmo	Descrição	Estatística	Erro Padrão
SHA-256	Média	4103955,57	65986,904
	Intervalo de confiança de 95% para	3974466,74	
	média	4233444,41	
	5% da média aparada	3983925,46	
	Mediana	3796470,00	
	Variância	4,354E+12	
	Desvio Padrão	2086689,109	
	Mínimo	294178	
	Máximo	12709116	
	Range	12414938	
	Amplitude interquantil	2741403	
	Assimetria	0,901	0,077
	Kurtosis	0,933	0,155
SHA-1	Média	4048292,74	63170,518
	Intervalo de confiança de 95% para	3924330,61	
	média	4172254,86	
	5% da média aparada	3931930,72	
	Mediana	3662829,50	
	Variância	3,991E+12	
	Desvio Padrão	1997627,164	
	Mínimo	605002	
	Máximo	13182126	
	Range	12577124	
	Amplitude interquantil	2704145	
	Assimetria	0,948	0,077
	Kurtosis	1,068	0,155
MD5	Média	4066137,24	64807,950
	Intervalo de confiança de 95% para	3938961,91	
	média	4193312,57	
	5% da média aparada	3940383,07	
	Mediana	3776953,00	
	Variância	4,200E+12	
	Desvio Padrão	2049407,325	
	Mínimo	358941	
	Máximo	16221330	
	Range	15862389	
	Amplitude interquantil	2436958	
	Assimetria	1,096	0,077
	Kurtosis	2,054	0,155

Fonte: do autor.

Observa-se na tabela 2 a diferença entre as médias de quantidade de *hashes*, induzindo o observador a crer que o algoritmo SHA-256 precisa calcular mais *hashes* em média para alcançar o objetivo, o que posteriormente será provado uma falsa impressão inicial.

Reforça ainda a tese da falsa impressão quando observado os valores de 5% da média aparada, demonstrando uma conversão de valores mais próximos um do outro, mesmo que não seja uma prova definitiva, indica uma dúvida quanto ao algoritmo SHA-256 ser realmente mais custoso em quantidade de *hashes* calculados.

Logo abaixo é apresentado na tabela 3 os resultados dos testes descritivos para o tempo de processamento, em segundos.

Tabela 3 – Descritivos para tempo de processamento com 20 zeros

Algoritmo	Descrição	Estatística	Erro Padrão
SHA-256	Média	4,153666	0,0698862
	Intervalo de confiança de 95% para	4,016525	
	média	4,290807	
	5% da média aparada	4,005043	
	Mediana	3,842000	
	Variância	4,884	
	Desvio Padrão	2,2099968	
	Mínimo	0,2940	
	Máximo	15,8480	
	Range	15,5540	
	Amplitude interquantil	2,8070	
	Assimetria	1,156	0,077
	Kurtosis	2,130	0,155
SHA-1	Média	3,437700	0,0546151
	Intervalo de confiança de 95% para	3,330527	
	média	3,544873	
	5% da média aparada	3,325494	
	Mediana	3,079500	
	Variância	2,983	
	Desvio Padrão	1,7270803	
	Mínimo	0,4910	
	Máximo	12,5530	
	Range	12,0620	
	Amplitude interquantil	2,2593	
	Assimetria	1,095	0,077
	Kurtosis	1,793	0,155
MD5	Média	2,921129	0,486600
	Intervalo de confiança de 95% para	2,825641	
	média	3,016617	
	5% da média aparada	2,810527	
	Mediana	2,662000	
	Variância	2,368	
	Desvio Padrão	1,5387640	
	Mínimo	0,2780	
	Máximo	10,9460	
	Range	10,6680	
	Amplitude interquantil	1,8103	
	Assimetria	1,305	0,077
	Kurtosis	2,794	0,155

Fonte: do autor.

Na tabela 3 também temos valores bem divergentes nas médias, também neste caso o SHA-256 é apresentado como o mais custoso em tempo de processamento para encontrar o resultado, observar-se-á que este resultado é realmente relevante estatisticamente, diferentemente da quantidade de *hashes*.

A seguir são apresentados os percentis obtidos para observar a progressão dos testes.

Tabela 4 – Percentis para quantidade de hashes com 20 zeros

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
SHA-256	294178,00	2567827,00	3796470,00	5307197,00	12709116,00
SHA-1	605002,00	2525146,00	3662829,50	5227138,00	13182126,00
MD5	358941,00	2664215,50	3776953,00	5099764,50	16221330,00

Fonte: do autor.

Observando os percentis apresentados na tabela 4, percentis para quantidade de *hashes*, temos que com 25% dos valores apurados o algoritmo MD5 teria calculado uma quantidade maior de *hashes* para obter o resultado, com 50% dos valores apurados o algoritmo SHA-256 toma a frente e com 75% dos valores apurados o SHA-256 se mantém com o maior número e o SHA-1 fica na segunda posição.

Tabela 5 – Percentis para tempo de processamento com 20 zeros

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
SHA-256	0,2940	2,5250	3,8420	5,3320	15,8480
SHA-1	0,4910	2,1547	3,0795	4,4140	12,5530
MD5	0,2780	1,8560	2,6620	3,6662	10,9460

Fonte: do autor.

Na tabela 5, onde apresenta os percentis para tempo de processamento, não é apresentado mudança entre os três algoritmos estudados, em todos os três percentis apresentados, o algoritmo SHA-256 possui um tempo de processamento maior em relação aos outros dois.

A seguir dos testes descritivos se observa a normalidade das distribuições, ambos os resultados dos testes de Kolmogorov-Smirnov e Shapiro-Wilk são apresentados nas tabelas abaixo.

Tabela 6 – Testes de normalidade para quantidade de hashes com 20 zeros

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
SHA-256	0,073	1000	< 0,001	0,933	1000	< 0,001
SHA-1	0,083	1000	< 0,001	0,935	1000	< 0,001
MD5	0,097	1000	< 0,001	0,921	1000	< 0,001

Fonte: do autor.

De acordo com os dados apresentados na tabela 6, pode-se concluir que a hipótese de normalidade não é sustentada ao encontrar significância inferior a 0,001, portanto a distribuição de quantidade de *hashes* para todos os três algoritmos difere de uma distribuição normal.

Tabela 7 – Testes de normalidade para tempo de processamento com 20 zeros

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
SHA-256	0,505	1000	< 0,001	0,144	1000	< 0,001
SHA-1	0,509	1000	< 0,001	0,131	1000	< 0,001
MD5	0,519	1000	< 0,001	0,239	1000	< 0,001

Fonte: do autor.

Observando a tabela 7, obtém-se a mesma conclusão sobre o tempo de processamento, as distribuições dos três algoritmos diferem de uma distribuição normal.

Uma vez obtido a informação de divergência da normalidade de todas as distribuições, foi realizado o teste H de Kruskal-Wallis de amostras independentes para verificar a hipótese nula que afirma sobre as distribuições não terem diferença estatisticamente significativa entre os algoritmos.

Tabela 8 – Teste de Kruskal-Wallis para quantidade de hashes com 20 zeros

Hipótese nula	Teste	Sig.	Decisão
A distribuição de Quantidade de Hashes é a mesma entre as categorias de Algoritmos	Teste de Kruskal-Wallis de Amostras Independentes	0,850	Reter a hipótese nula.

Fonte: do autor.

Quanto a quantidade de *hashes*, a tabela 8 demonstra a decisão de reter a hipótese nula, tendo em vista que a significância é 0,850, portanto se pode concluir que as distribuições de quantidade de *hashes* não possuem diferença estatisticamente significativas entre si.

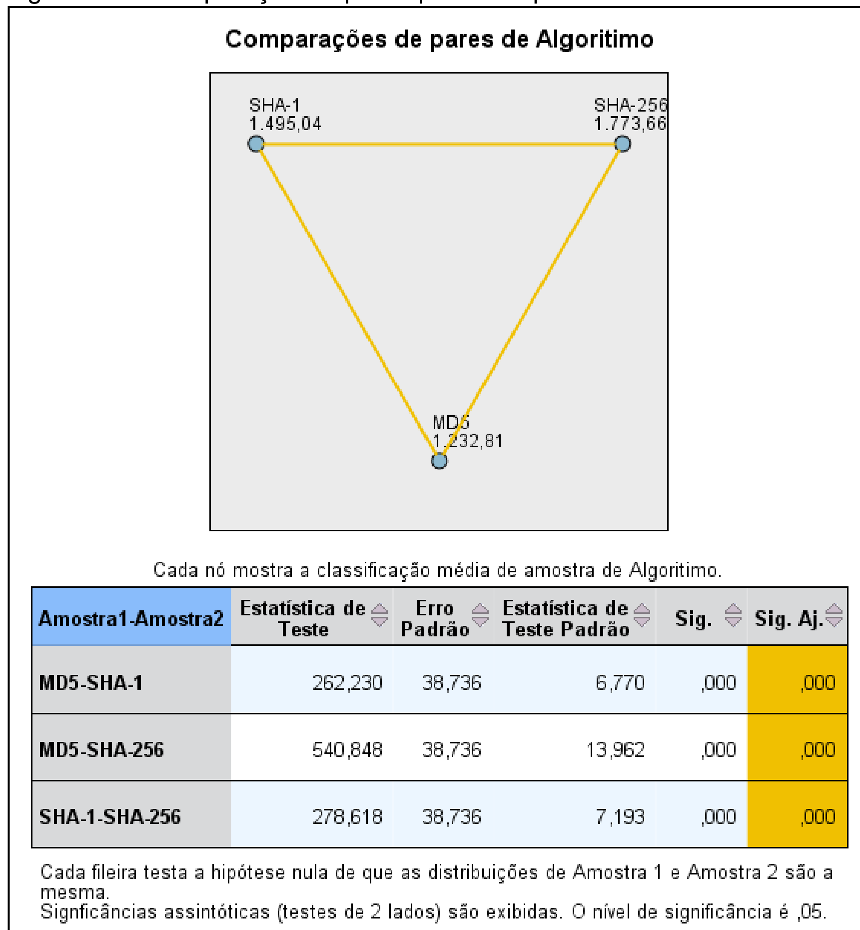
Tabela 9 – Teste de Kruskal-Wallis para tempo de processamento com 20 zeros

Hipótese nula	Teste	Sig.	Decisão
A distribuição de Tempo é a mesma entre as categorias de Algoritmos	Teste de Kruskal-Wallis de Amostras Independentes	< 0,001	Rejeitar a hipótese nula.

Fonte: do autor.

Diferentemente da quantidade de *hashes*, a tabela 9 demonstra a rejeição da hipótese nula, com uma significância inferior a 0,001, conclui-se portanto que as distribuições de tempo de processamento possuem diferença estatisticamente significativas entre si. Uma vez observado a rejeição de hipótese nula podemos observar a comparação de pares na figura 24.

Figura 24 – Comparação de pares para Tempo de Processamento com 20 zeros



Amostra1-Amostra2	Estatística de Teste	Erro Padrão	Estatística de Teste Padrão	Sig.	Sig. Aj.
MD5-SHA-1	262,230	38,736	6,770	,000	,000
MD5-SHA-256	540,848	38,736	13,962	,000	,000
SHA-1-SHA-256	278,618	38,736	7,193	,000	,000

Cada fileira testa a hipótese nula de que as distribuições de Amostra 1 e Amostra 2 são a mesma.
Signficâncias assintóticas (testes de 2 lados) são exibidas. O nível de significância é ,05.

Fonte: do autor.

A figura 23 compara cada par de distribuição entre si para verificar onde se encontra a diferença estatisticamente significativa, fica desta forma demonstrado que todos os pares possuem diferença estatisticamente significativa entre si.

Os resultados demonstrados até aqui foram entre os três algoritmos utilizando uma parametrização de 20 zeros, agora será apresentado os mesmos testes para uma parametrização de 25 zeros.

Tabela 10 – Descritivos quantidade de hashes com 25 zeros

Algoritmo	Descrição	Estatística	Erro Padrão
SHA-256	Média	134225834,0	2169546,261
	Intervalo de confiança de 95%	129968443,4	
	para média	138483224,6	
	5% da média aparada	129965491,9	
	Mediana	122121062,0	
	Variância	4,707E+15	
	Desvio Padrão	68607076,74	
	Mínimo	11230077	
	Máximo	4E+008	
	Range	415217168	
	Amplitude interquantil	90735107	
	Assimetria	0,931	0,077
	Kurtosis	0,986	0,155
SHA-1	Média	130523071,5	2143545,632
	Intervalo de confiança de 95%	126316703,0	
	para média	134729439,9	
	5% da média aparada	126511040,0	
	Mediana	120503495,0	
	Variância	4,595E+15	
	Desvio Padrão	67784864,65	
	Mínimo	12522789	
	Máximo	5E+008	
	Range	523050891	
	Amplitude interquantil	90564944	
	Assimetria	0,963	0,077
	Kurtosis	1,411	0,155
MD5	Média	131663708,5	2189491,772
	Intervalo de confiança de 95%	127367178,1	
	para média	135960239,0	
	5% da média aparada	126847495,0	
	Mediana	117679351,0	
	Variância	4,794E+15	
	Desvio Padrão	69237809,18	
	Mínimo	12885044	
	Máximo	5E+008	
	Range	448407602	
	Amplitude interquantil	84984260	
	Assimetria	1,072	0,077
	Kurtosis	1,431	0,155

Fonte: do autor.

Observado a tabela 10, os valores de médias e medianas possuem valores próximos, sendo por ora não passível de opiniões prévias ao resultado, prossegue-se assim para o teste de tempo de processamento.

Tabela 11 – Descritivos tempo de processamento com 25 zeros

Algoritmo	Descrição	Estatística	Erro Padrão
SHA-256	Média	134,142170	2,2369242
	Intervalo de confiança de 95% para média	129,752561 138,531779	
	5% da média aparada	129,574239	
	Mediana	122,721000	
	Variância	5003,830	
	Desvio Padrão	70,7377542	
	Mínimo	10,6750	
	Máximo	439,5870	
	Range	428,9120	
	Amplitude interquantil	92,5688	
	Assimetria	0,982	0,077
	Kurtosis	1,149	0,155
SHA-1	Média	106,699740	1,7831787
	Intervalo de confiança de 95% para média	103,200534 110,198946	
	5% da média aparada	103,297160	
	Mediana	97,034500	
	Variância	3179,726	
	Desvio Padrão	56,3890629	
	Mínimo	9,0440	
	Máximo	366,5290	
	Range	357,4850	
	Amplitude interquantil	78,1495	
	Assimetria	0,939	0,077
	Kurtosis	1,033	0,155
MD5	Média	89,249677	1,5026610
	Intervalo de confiança de 95% para média	86,300943 92,198411	
	5% da média aparada	85,996576	
	Mediana	80,665500	
	Variância	2257,990	
	Desvio Padrão	47,5183133	
	Mínimo	7,4830	
	Máximo	309,6070	
	Range	302,1240	
	Amplitude interquantil	58,9703	
	Assimetria	1,080	0,077
	Kurtosis	1,569	0,155

Fonte: do autor.

Já na tabela 11, pode-se observar um padrão nas médias, no corte de 5% e nas medianas, todas seguem com os valores de SHA-256 > SHA-1 > MD5, levando a crer que o SHA-256 leve em média mais tempo processar um resultado.

Após os descritivos, é apresentado os percentis da parametrização com 25 zeros.

Tabela 12 – Percentis para quantidade de hashes com 25 zeros

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
SHA-256	11230077,00	83960161,50	122121062,0	174634448,5	4E+008
SHA-1	12522789,00	77994333,00	120503495,0	168394264,0	5E+008
MD5	12885044,00	83298417,00	117679351,0	168202992,0	5E+008

Fonte: do autor.

De acordo com os percentis na tabela 12, teríamos que até 25% das amostras apuradas o SHA-256 necessitaria de mais *hashes* calculados, seguido pelo MD5, todavia o SHA-1 ultrapassa o MD5 na coluna de 50% de amostras apuradas e esta relação se mantém após isto.

Tabela 13 – Percentis para tempo de processamento com 25 zeros

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
SHA-256	10,6750	81,995500	122,721000	174,519000	439,5870
SHA-1	9,0440	62,519500	97,0345000	140,578500	366,5290
MD5	7,4830	51,945000	80,665500	113,881500	309,6070

Fonte: do autor.

Apresentada na tabela 13, temos uma relação constante dos três algoritmos a medida que as amostras são apuradas, desta forma o SHA-256 levou mais tempo para ser processado em todo o decorrer das apurações.

Agora será apresentado os testes de normalidade da parametrização 25 zeros.

Tabela 14 – Testes de normalidade para quantidade de hashes com 25 zeros

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
SHA-256	0,076	1000	< 0,001	0,948	1000	< 0,001
SHA-1	0,073	1000	< 0,001	0,945	1000	< 0,001
MD5	0,086	1000	< 0,001	0,936	1000	< 0,001

Fonte: do autor.

A tabela 14 apresenta que as distribuições de quantidade de *hashes* com parametrização de 25 zeros diferem de uma distribuição normal, tendo significância inferior a 0,001.

Tabela 15 – Testes de normalidade para tempo de processamento com 25 zeros

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
SHA-256	0,082	1000	< 0,001	0,944	1000	< 0,001
SHA-1	0,072	1000	< 0,001	0,945	1000	< 0,001
MD5	0,088	1000	< 0,001	0,936	1000	< 0,001

Fonte: do autor.

De mesma forma a anterior, a tabela 15 demonstra que as distribuições de tempo de processamento com a parametrização 25 zeros diferem de uma distribuição normal, tendo significância inferior a 0,001.

Prossegue-se portanto para o teste H de Kruskal-Wallis para verificar a significância estatística entre as distribuições.

Tabela 16 – Teste de Kruskal-Wallis para quantidade de hashes com 25 zeros

Hipótese nula	Teste	Sig.	Decisão
A distribuição de Quantidade de Hashes é a mesma entre as categorias de Algoritmos	Teste de Kruskal-Wallis de Amostras Independentes	0,398	Reter a hipótese nula.

Fonte: do autor.

Assim como o resultado com parametrização de 20 zeros, o teste H de Kruskal-Wallis feito nas distribuições de quantidade de *hashes* com parametrização de 25 zeros apresentaram decisão de reter a hipótese nula, com significância de 0,398 temos que as distribuições não possuem diferenças estatisticamente significativas.

Tabela 17 – Teste de Kruskal-Wallis para tempo de processamento com 25 zeros

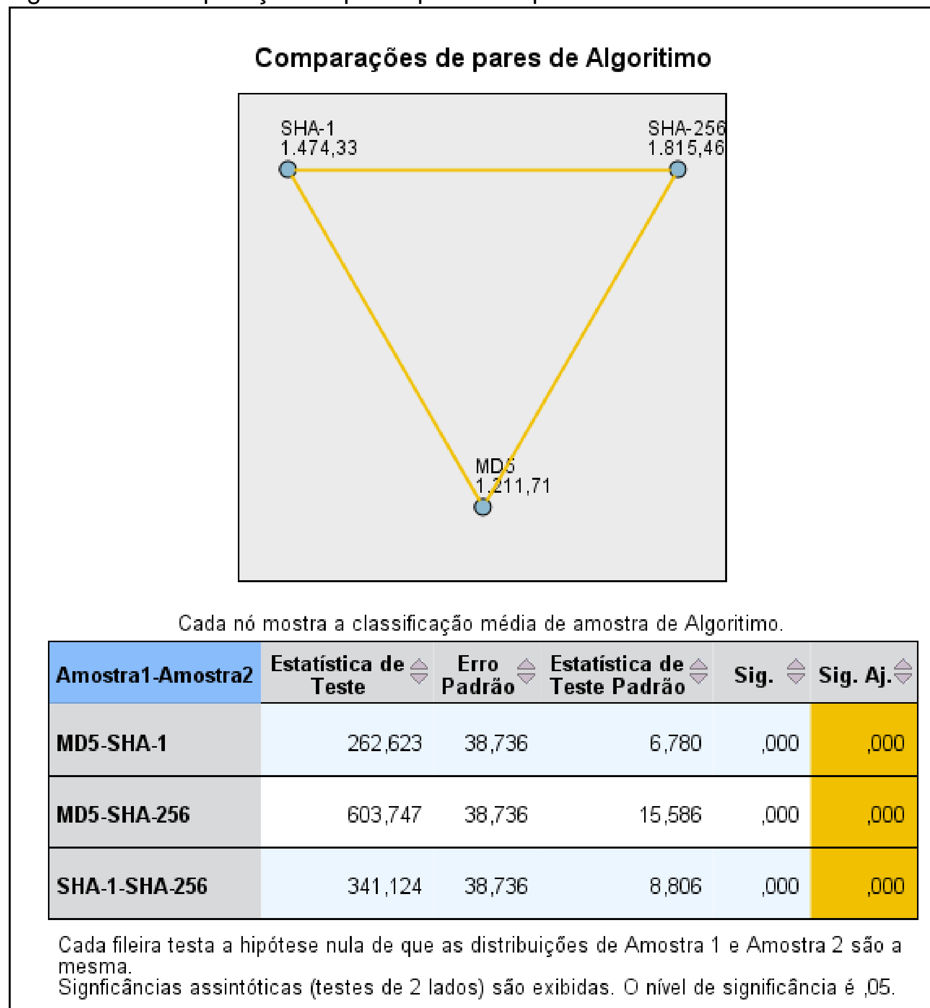
Hipótese nula	Teste	Sig.	Decisão
A distribuição de Tempo de Processamento é a mesma entre as categorias de Algoritmos	Teste de Kruskal-Wallis de Amostras Independentes	0,000	Rejeitar a hipótese nula.

Fonte: do autor.

Também indo de encontro com o teste de parametrização 20 zeros, as distribuições de tempo de processamento com parametrização de 25 zeros apresentam decisão de rejeitar a hipótese nula, ou seja, possuem entre si diferença estatisticamente significativa, com significância inferior a 0,001, como observado no teste descritivo, mantém-se a visão inicial de tempos SHA-256 > SHA-1 > MD5.

Comprovado também da mesma forma como na parametrização com 20 zeros, na figura 25, a significância estatística entre todos os pares.

Figura 25 – Comparação de pares para Tempo de Processamento com 25 zeros



Amostra1-Amostra2	Estatística de Teste	Erro Padrão	Estatística de Teste Padrão	Sig.	Sig. Aj.
MD5-SHA-1	262,623	38,736	6,780	,000	,000
MD5-SHA-256	603,747	38,736	15,586	,000	,000
SHA-1-SHA-256	341,124	38,736	8,806	,000	,000

Cada fileira testa a hipótese nula de que as distribuições de Amostra 1 e Amostra 2 são a mesma.
Signficâncias assintóticas (testes de 2 lados) são exibidas. O nível de significância é ,05.

Fonte: do autor.

Até aqui foram apresentados comparações estatísticas entre algoritmos diferentes com mesma parametrização de zeros, agora será observado o mesmo algoritmo, sendo comparado entre parametrizações de 20 e 25 zeros.

Primeiramente são apresentados as análises descritivas.

Tabela 18 – Descritivos quantidade de hashes com SHA-256

Algoritmo	Descrição	Estatística	Erro Padrão
20 zeros	Média	4103955,57	65986,904
	Intervalo de confiança de 95% para média	3974466,74 4233444,41	
	5% da média aparada	3983925,46	
	Mediana	3796470,00	
	Variância	4,354E+12	
	Desvio Padrão	2086689,109	
	Mínimo	294178	
	Máximo	12709116	
	Range	12414938	
	Amplitude interquantil	2741403	
	Assimetria	0,901	0,077
	Kurtosis	0,933	0,155
25 zeros	Média	134225834,0	2169546,261
	Intervalo de confiança de 95% para média	129968443,4 138483224,6	
	5% da média aparada	129965491,9	
	Mediana	122121062,0	
	Variância	4,707E+15	
	Desvio Padrão	68607076,74	
	Mínimo	11230077	
	Máximo	4E+008	
	Range	415217168	
	Amplitude interquantil	90735107	
	Assimetria	0,931	0,077
	Kurtosis	0,986	0,155

Fonte: do autor.

Na tabela 18 é possível observar uma grande diferença nos valores de média e mediana, induzindo que a parametrização de 25 zeros necessita de mais *hashes* para obter o resultado.

Quanto ao tempo de processamento do SHA-256, observa-se a seguir:

Tabela 19 – Descritivos tempo de processamento com SHA-256

Algoritmo	Descrição	Estatística	Modelo Padrão
20 zeros	Média	4,153666	0,0698862
	Intervalo de confiança de 95% para média	4,016525 4,290807	
	5% da média aparada	4,005043	
	Mediana	3,842000	
	Variância	4,884	
	Desvio Padrão	2,2099968	
	Mínimo	0,2940	
	Máximo	15,8480	
	Range	15,5540	
	Amplitude interquantil	2,8070	
	Assimetria	1,156	0,077
	Kurtosis	2,130	0,155
25 zeros	Média	134,142170	2,2369242
	Intervalo de confiança de 95% para média	129,752561 138,531779	
	5% da média aparada	129,574239	
	Mediana	122,721000	
	Variância	5003,830	
	Desvio Padrão	70,7377542	
	Mínimo	10,6750	
	Máximo	439,5870	
	Range	428,9120	
	Amplitude interquantil	92,5688	
	Assimetria	0,982	0,077
	Kurtosis	1,149	0,155

Fonte: do autor.

Assim como na tabela anterior, na tabela 19 pode ser visto uma diferença nas médias e medianas induzindo a crer que a parametrização de zeros também leva mais tempo para ser processada.

São abordados agora, os percentis.

Tabela 20 – Percentis quantidade de hashes com SHA-256

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
20 zeros	294178,00	2567827,00	3796470,00	5307197,00	12709116,00
25 zeros	11230077, 00	83960161,50	122121062,0	174634448,5	4E+008

Fonte: do autor.

Na tabela 20, é possível verificar que durante todo o decorrer do processamento, a distribuição com parametrização de 25 zeros demandou uma quantidade maior de *hashes* para encontrar um resultado.

Tabela 21 – Percentis tempo de processamento com SHA-256

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
20 zeros	0,2940	2,5250	3,8420	5,3320	15,8480
25 zeros	10,6750	81,9955	122,7210	174,5190	439,5870

Fonte: do autor.

De mesma forma, na tabela 21 se pode verificar a necessidade de mais tempo de processamento para a parametrização de 25 zeros.

Apresenta-se neste momento os testes de normalidade do SHA-256

Tabela 22 – Testes de normalidade quantidade de hashes com SHA-256

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
20 zeros	0,071	1000	< 0,001	0,952	1000	< 0,001
25 zeros	0,076	1000	< 0,001	0,948	1000	< 0,001

Fonte: do autor.

Assim como todos os testes anteriores, pode-se verificar na tabela 22 que as distribuições de SHA-256 quanto a quantidade de *hashes* diferem de uma distribuição normal com significância inferior a 0,001.

Tabela 23 – Testes de normalidade tempo de processamento com SHA-256

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
20 zeros	0,073	1000	< 0,001	0,933	1000	< 0,001
25 zeros	0,082	1000	< 0,001	0,944	1000	< 0,001

Fonte: do autor.

Também na tabela 23 se pode observar que as distribuições do SHA-256 quanto ao tempo de processamento diferem de uma distribuição normal, com significância inferior a 0,001.

Agora será verificado se estas distribuições possuem diferença estatisticamente significativa entre si, por se tratar de apenas duas amostras a serem comparadas, não é utilizado o teste H de Kruskal-Wallis, mas o teste U de Mann-Whitney.

Tabela 24 – Teste U de Mann-Whitney para quantidade de hashes com SHA-256

Hipótese nula	Teste	Sig.	Decisão
A distribuição de Quantidade de Hashes é a mesma entre as categorias de Número de Zeros	Teste U de Mann-Whitney de Amostras Independentes	< 0,001	Rejeitar a hipótese nula.

Fonte: do autor.

De acordo com a tabela 24, pode-se reparar que a decisão é de rejeitar a hipótese nula, portanto se constata uma diferença estatisticamente significativa entre as distribuições de SHA-256 para a quantidade de *hashes*, com significância inferior a 0,001, onde observando as medias e medianas, tem-se que a parametrização de 25 zeros demanda mais *hashes* para chegar a um resultado.

Tabela 25 – Teste U de Mann-Whitney tempo de processamento com SHA-256

Hipótese nula	Teste	Sig.	Decisão
A distribuição de tempo de processamento é a mesma entre as categorias de Número de Zeros	Teste U de Mann-Whitney de Amostras Independentes	< 0,001	Rejeitar a hipótese nula.

Fonte: do autor.

De acordo com a tabela 25, conclui-se que a decisão é a rejeição da hipótese nula, ou seja, se constata uma diferença estatisticamente significativa entre as distribuições de SHA-256 para tempo de processamento, com significância inferior a 0,001, onde observando as medias e medianas, tem-se que a parametrização de 25 zeros demanda mais tempo para chegar a um resultado.

A seguir se tem o algoritmo SHA-1 comparado quanto parametrização 20 e 25 zeros, começando com os testes descritivos.

Tabela 26 – Descritivos quantidade de hashes com SHA-1

Algoritmo	Descrição	Estatística	Erro Padrão
20 zeros	Média	4048292,74	63170,518
	Intervalo de confiança de 95%	3924330,61	
	para média	4172254,86	
	5% da média aparada	3931930,72	
	Mediana	3662829,50	
	Variância	3,991E+12	
	Desvio Padrão	1997627,164	
	Mínimo	605002	
	Máximo	13182126	
	Range	12577124	
	Amplitude interquantil	2704145	
	Assimetria	0,948	0,077
	Kurtosis	1,068	0,155
25 zeros	Média	130523071,5	2143545,632
	Intervalo de confiança de 95%	126316703,0	
	para média	134729439,9	
	5% da média aparada	126511040,0	
	Mediana	120503495,0	
	Variância	4,595E+15	
	Desvio Padrão	67784864,65	
	Mínimo	12522789	
	Máximo	5E+008	
	Range	523050891	
	Amplitude interquantil	90564944	
	Assimetria	0,963	0,077
	Kurtosis	1,411	0,155

Fonte: do autor.

Na tabela 26 se pode observar as diferenças das médias e medianas induzindo que a parametrização de 25 zeros necessita de mais *hashes* calculados para obter o resultado.

Tabela 27 – Descritivos tempo de processamento com SHA-1

Algoritmo	Descrição	Estatística	Erro Padrão
20 zeros	Média	3,437700	0,0546151
	Intervalo de confiança de 95% para média	3,330527 3,544873	
	5% da média aparada	3,325494	
	Mediana	3,079500	
	Variância	2,983	
	Desvio Padrão	1,7270803	
	Mínimo	0,4910	
	Máximo	12,5530	
	Range	12,0620	
	Amplitude interquantil	2,2593	
	Assimetria	1,095	0,077
	Kurtosis	1,793	0,155
25 zeros	Média	106,699740	1,7831787
	Intervalo de confiança de 95% para média	103,200534 110,198946	
	5% da média aparada	103,297160	
	Mediana	97,034500	
	Variância	3179,726	
	Desvio Padrão	56,3890629	
	Mínimo	9,0440	
	Máximo	366,5290	
	Range	357,4850	
	Amplitude interquantil	78,1495	
	Assimetria	0,939	0,077
	Kurtosis	1,033	0,155

Fonte: do autor.

Assim como na tabela anterior, a tabela 27 apresenta diferença nas medias e medianas induzindo que a parametrização de 25 zeros levaria mais tempo para ser processada.

Neste momento, os percentis.

Tabela 28 – Percentis quantidade de hashes com SHA-1

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
20 zeros	605002,00	2525186,00	3662829,50	5227138,00	13182126,00
25 zeros	12522789,00	77994333,00	120503495,0	168394264,0	5E+008

Fonte: do autor.

Na tabela 28 pode ser verificado uma maior quantidade de *hashes* necessária para obter um resultado durante todo o decorrer do processamento com parametrização de 25 zeros.

Tabela 29 – Percentis tempo de processamento com SHA-1

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
20 zeros	0,4910	2,1547	3,0795	4,4140	12,5530
25 zeros	9,0440	62,5195	97,0345	140,5785	366,5290

Fonte: do autor.

Assim como na anterior, a tabela 29 demonstra um maior tempo de processamento durante todo o processo para a parametrização com 25 zeros.

Agora são apresentados os testes de normalidade do SHA-1.

Tabela 30 – Testes de normalidade quantidade de hashes com SHA-1

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
20 zeros	0,087	1000	< 0,001	0,945	1000	< 0,001
25 zeros	0,073	1000	< 0,001	0,945	1000	< 0,001

Fonte: do autor.

De acordo com o apresentado na tabela 30 se tem que as distribuições de SHA-1 quanto a quantidade de *hashes* diferem de uma distribuição normal, com significância inferior a 0,001.

Tabela 31 – Testes de normalidade tempo de processamento com SHA-1

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
20 zeros	0,083	1000	< 0,001	0,935	1000	< 0,001
25 zeros	0,072	1000	< 0,001	0,945	1000	< 0,001

Fonte: do autor.

Assim como na tabela anterior, a tabela 31 apresenta que as distribuições do algoritmo SHA-1 para o tempo de processamento diferem de uma distribuição normal, apresentando uma significância inferior a 0,001.

Desta forma, pode-se apresentar os testes U de Mann-Whitney para verificar a significância estatística das distribuições.

Tabela 32 – Teste U de Mann-Whitney para quantidade de hashes com SHA-1

Hipótese nula	Teste	Sig.	Decisão
A distribuição de Quantidade de Hashes é a mesma entre as categorias de Número de Zeros	Teste U de Mann-Whitney de Amostras Independentes	< 0,001	Rejeitar a hipótese nula.

Fonte: do autor.

De acordo com a tabela 32, é apresentado que as distribuições de SHA-1 quanto a quantidade de *hashes* possuem diferenças estatisticamente significativas, com significância inferior a 0,001, conclui-se que a parametrização de 25 zeros requer uma quantidade maior de *hashes* calculados para chegar ao resultado.

Tabela 33 – Teste U de Mann-Whitney para tempo de processamento com SHA-1

Hipótese nula	Teste	Sig.	Decisão
A distribuição de Quantidade de Hashes é a mesma entre as categorias de Número de Zeros	Teste U de Mann-Whitney de Amostras Independentes	< 0,001	Rejeitar a hipótese nula.

Fonte: do autor.

Assim como na tabela anterior, pode ser visto na tabela 33 que existem diferenças estatisticamente significativas nas distribuições de SHA-1 para o tempo de processamento, com significância inferior a 0,001, pode ser concluído que a parametrização de 25 zeros demanda mais tempo de processamento para alcançar o resultado.

Tem-se a seguir os testes do algoritmo MD5, iniciando-se pelos testes descritivos.

Tabela 34 – Descritivos quantidade de hashes com MD5

Algoritmo	Descrição	Estatística	Erro Padrão
20 zeros	Média	4066137,24	64807,950
	Intervalo de confiança de 95%	3938961,91	
	para média	4193312,57	
	5% da média aparada	3940383,07	
	Mediana	3776953,00	
	Variância	4,200E+12	
	Desvio Padrão	2049407,325	
	Mínimo	358941	
	Máximo	16221330	
	Range	15862389	
	Amplitude interquantil	2436958	
	Assimetria	1,096	0,077
	Kurtosis	2,054	0,155
25 zeros	Média	131663708,5	2189491,772
	Intervalo de confiança de 95%	127367178,1	
	para média	135960239,0	
	5% da média aparada	126847495,0	
	Mediana	117679351,0	
	Variância	4,794E+15	
	Desvio Padrão	69237809,18	
	Mínimo	12885044	
	Máximo	5E+008	
	Range	448407602	
	Amplitude interquantil	84984260	
	Assimetria	1,072	0,077
	Kurtosis	1,431	0,155

Fonte: do autor.

Na tabela 34, também se pode ver uma diferença nas medias e medianas das distribuições, induzindo uma quantidade de *hashes* maior para parametrização de 25 zeros.

Tabela 35 – Descritivos tempo de processamento com MD5

Algoritmo	Descrição	Estatística	Erro Padrão
20 zeros	Média	2,921129	5,2231619
	Intervalo de confiança de 95%	2,825641	
	para média	3,016617	
	5% da média aparada	2,810527	
	Mediana	2,662000	
	Variância	2,368	
	Desvio Padrão	1,5387640	
	Mínimo	0,2780	
	Máximo	10,9460	
	Range	10,6680	
	Amplitude interquantil	1,8103	
	Assimetria	1,305	0,077
	Kurtosis	2,794	0,155
25 zeros	Média	89,249677	0,0546151
	Intervalo de confiança de 95%	86,300943	
	para média	92,198411	
	5% da média aparada	85,996576	
	Mediana	80,665500	
	Variância	2257,990	
	Desvio Padrão	47,5183133	
	Mínimo	7,4830	
	Máximo	309,6070	
	Range	302,1240	
	Amplitude interquantil	58,9703	
	Assimetria	1,080	0,077
	Kurtosis	1,569	0,155

Fonte: do autor.

Assim como na tabela anterior, pode-se verificar na tabela 35 a existência de diferenças nas medias e medianas induzindo que a parametrização de 25 zeros leva mais tempo para processar.

São apresentados agora os percentis.

Tabela 36 – Percentis quantidade de hashes com MD5

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
20 zeros	358941,00	2664215,50	3776953,00	5099764,50	16221330,00
25 zeros	12885044,00	83298417,00	117679351,0	168202992,0	5E+008

Fonte: do autor.

Assim como nos anteriores, na tabela 36 pode ser verificado a demanda maior de quantidade de *hashes* na parametrização com 25 zeros.

Tabela 37 – Percentis tempo de processamento com MD5

Algoritmo	Mínimo	P_{25}	P_{50}	P_{75}	Máximo
20 zeros	0,2780	1,8560	2,6620	3,6662	10,9460
25 zeros	12885044,00	54,9450	80,6655	113,8815	5E+008

Fonte: do autor.

Seguindo o mesmo padrão, pode ser constatado na tabela 37 a necessidade de maior tempo de processamento para a distribuição com parametrização de 25 zeros.

Segue-se abaixo os testes de normalidade para o algoritmo MD5.

Tabela 38 – Testes de normalidade quantidade de hashes com MD5

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
20 zeros	0,088	1000	< 0,001	0,940	1000	< 0,001
25 zeros	0,086	1000	< 0,001	0,936	1000	< 0,001

Fonte: do autor.

Com significância inferior a 0,001, a tabela 38 demonstra que as distribuições de quantidade de hashes com o algoritmo MD5 também diferem de uma distribuição normal.

Tabela 39 – Testes de normalidade tempo de processamento com MD5

Algoritmo	Kolmogorov-Smirnov Estatística	Kolmogorov-Smirnov df	Kolmogorov-Smirnov Sig.	Shapiro-Wilk Estatística	Shapiro-Wilk sf	Shapiro-Wilk Sig.
20 zeros	0,097	1000	< 0,001	0,921	1000	< 0,001
25 zeros	0,088	1000	< 0,001	0,936	1000	< 0,001

Fonte: do autor.

Também na tabela 39 é indicado um desvio da normalidade nas distribuições de tempo de processamento com MD5, verificando uma significância inferior a 0,001.

Agora se pode verificar com o teste U de Mann-Whitney se as distribuições com MD5 possuem diferenças significativas.

Tabela 40 – Teste U de Mann-Whitney para quantidade de hashes com MD5

Hipótese nula	Teste	Sig.	Decisão
A distribuição de Quantidade de Hashes é a mesma entre as categorias de Número de Zeros	Teste U de Mann-Whitney de Amostras Independentes	< 0,001	Rejeitar a hipótese nula.

Fonte: do autor.

Pode-se observar uma significância inferior a 0,001 na tabela 40, indicando uma diferença estatisticamente significativa entre as distribuições de quantidade de *hashes* com MD5, portanto se conclui que quando parametrizado com 25 zeros, é necessário mais *hashes* para chegar a um resultado.

Tabela 41 – Teste U de Mann-Whitney para tempo de processamento com MD5

Hipótese nula	Teste	Sig.	Decisão
A distribuição de Quantidade de Hashes é a mesma entre as categorias de Número de Zeros	Teste U de Mann-Whitney de Amostras Independentes	< 0,001	Rejeitar a hipótese nula.

Fonte: do autor.

Na tabela 41 também é visualizado a decisão de rejeitar a hipótese nula, onde se pode comprovar a diferença estatisticamente significativa entre as distribuições de tempo de processamento com MD5, desta forma se tem que leva mais tempo para processar quando parametrizado com 25 zeros.

Acima descritos, foram feitos os testes necessários para obter as conclusões que este trabalho visa, compilando no capítulo abaixo os resultados.

5.3 RESULTADOS

Abaixo seguem as tabelas com os resultados obtidos após a alimentação dos valores no software SPSS e as análises realizadas:

Tabela 42 – Quantidade de hashes com 20 zeros

Algoritmo	n	$\bar{x} \pm DP$	Mín	P_{25}	Md	P_{75}	Máx
SHA-256	1000	4103955.57±2086689.109	294178	2567827.00	3796470.00	5307197.00	12709116
SHA-1	1000	4048292.74±1997627.164	605002	2525146.00	3662829.50	5227138.00	13182126
MD5	1000	4066137.24±2049407.325	358941	2664215.50	3776953.00	5099764.50	16221330

Fonte: do autor.

Embora demonstrado na tabela 42 onde a amostra sugere que o número de *hashes* calculados nos algoritmos SHA-1 e MD5 seja menor que o SHA-256, não foi encontrado diferença estatisticamente significativa, tendo valor p = 0,850, desta forma a conclusão é de que a quantidade de *hashes* calculada para obter o

resultado desejado independe do algoritmo utilizado para uma mesma parametrização.

Tabela 43 – Tempo de processamento com 20 zeros

Algoritmo	n	$\bar{x} \pm DP$	Mín	P_{25}	Md	P_{75}	Máx
SHA-256	1000	$4.143666 \pm 2.2099968^{a}$	0.2940	2.5250	3.8420	5.3320	15.8
SHA-1	1000	$3.437700 \pm 1.7270803^{b}$	0.4910	2.1547	3.0795	4.4140	12.5
MD5	1000	$2.921129 \pm 1.5387640^{c}$	0.2780	1.8560	2.8120	3.6662	10.9

Fonte: do autor.

[a,b,c] Letras distintas indicam diferenças estatisticamente significativas.

Verifica-se diferença estatisticamente significativas após aplicação do teste de Kruskal-Wallis, tendo valor $p < 0.001$, portanto se pode concluir que o tempo de processamento varia conforme o algoritmo utilizado, assim, observamos a mediana na tabela 43 para classificar, tendo a seguinte relação de tempo: SHA-256 > SHA-1 > MD5.

Isto nos diz quanto a parametrização de 20 zeros, é demonstrado agora com parametrização de 25 zeros.

Tabela 44 – Quantidade de hashes com 25 zeros

Algoritmo	n	$\bar{x} \pm DP$	Mín	P_{25}	Md	P_{75}	Máx
SHA-256	1000	134225834,0±68607076,74	11230077	83960161,50	122121062,0	174634448,5	4E+008
SHA-1	1000	130523071,5±67784864,65	12522789	77994333,00	120503495,0	168394264,0	5E+008
MD5	1000	131663708,5±69237809,18	12885044	83298417,00	117679351,0	168202992,0	5E+008

Fonte: do autor.

Da mesma forma como na parametrização de 20 zeros, é demonstrado na tabela 44 uma amostra que sugere um número de *hashes* calculados nos algoritmos SHA-1 e MD5 inferior ao SHA-256, entretanto não foi encontrado diferença estatisticamente significativa, tendo valor $p = 0{,}398$, desta forma a conclusão é reforçada de que a quantidade de *hashes* calculada para obter o resultado desejado independe do algoritmo utilizado para uma mesma parametrização.

Tabela 45 – Tempo de processamento com 25 zeros

Algoritmo	n	$\bar{x} \pm DP$	Mín	P_{25}	Md	P_{75}	Máx
SHA-256	1000	134,142170±70,7377542 [a]	10,6750	81,995500	122,721000	174,519000	439,5870
SHA-1	1000	106,699740±56,3890629 [b]	9,0440	62,519500	97,034500	140,578500	366,5290
MD5	1000	89,249677±47,5183133 [c]	7,4830	54,945000	80,665500	113,881500	309,6070

Fonte: do autor.
[a,b,c] Letras distintas indicam diferenças estatisticamente significativas.

Verifica-se diferença estatisticamente significativas após aplicação do teste de Kruskal-Wallis, tendo valor $p < 0,001$, portanto se pode concluir que o tempo de processamento varia conforme o algoritmo utilizado, assim observamos a mediana na tabela 45 para classificar, tendo a seguinte relação de tempo: SHA-256 > SHA-1 > MD5.

Acima foram demonstrados as comparações de algoritmos diferentes com a mesma parametrização, primeiro com 20 zeros e logo após com 25 zeros, será visto a seguir comparações em que o algoritmo é o mesmo, mas a parametrização é diferente.

Tabela 46 – Comparações de mesmo algoritmo com parametrizações diferentes

Algoritmo	Tipo	Normal	Md 20 zeros	Md 25 zeros	Valor-p
SHA-256	*Hashes*	Não	3796470,00	122121062,0	< 0,001
	Tempo	Não	3,8420	122,721000	< 0,001
SHA-1	*Hashes*	Não	3662829,50	120503495,0	< 0,001
	Tempo	Não	3,0795	97,034500	< 0,001
MD5	*Hashes*	Não	3776953,00	117679351,0	< 0,001
	Tempo	Não	2,6620	80,665500	< 0,001

Fonte: do autor.

O valor-p foi obtido com a aplicação do teste U de Mann-Whitney para duas amostras independentes.

Tem-se valor-p inferior a 0,001 em todos os pares, desta forma se observa significância estatística em todas as comparações, concluindo que, em todos os casos a parametrização de 25 zeros possui valores maiores, ou seja, demanda maior quantidade de *hashes* calculados e também maior tempo de processamento para alcançar um resultado.

Abaixo se pode observar alguns dos dados utilizados para alimentar o SPSS afim de obter as estatísticas estudadas.

Tabela 47 – Amostra dos dados obtidos

Algoritmo	Parametrização	Quantidade de Hashes	Tempo(s)	Hash resultado
MD5	20 zeros	1291762	1,086	00000DCA31A64BC210430E045FCE8916
	20 zeros	2761927	2,137	000000FE757832E7B13EC7503234D808
	20 zeros	10503236	9,96	000005F4AA3EE2E02C66F4A71F921FB7
	20 zeros	9922819	8,825	00000233A5660D2392D8015BA1DB8374
	20 zeros	5302120	4,715	00000E7844D0CDD6BCAABB3082F1990E
	25 zeros	271985213	211,117	0000001C0DD1036B2A87E42B5249FB76
	25 zeros	196964075	130,25	0000005C01D444D420B0E0B0D9697B8F
	25 zeros	135613653	107,758	000000633967696FFB6DA8C30608099E
	25 zeros	96984259	68,578	00000075B9B94640E10BE13A75C51EF3
	25 zeros	181763842	146,396	00000033380E478392BEA0AFE494E563
SHA-1	20 zeros	2835941	2,627	00000AB56754DBE8A9C95572DDE309CE681244CE
	20 zeros	1810763	1,92	00000EA3B8EB99013228F10ECC244780E5664C34
	20 zeros	3396662	3,199	0000030D62F9042A499160011C65F42E18BB103A
	20 zeros	6934945	6,682	00000311964609CF386385156321F5C03D504118
	20 zeros	2109329	1,593	00000ABD6B545DF5673E5B999715FA0C57A4DED4
	25 zeros	70795550	62,028	0000005C492371BA33056FD736BC333254E7BE9F
	25 zeros	74307478	68,421	00000031E371EB90B591CB80FFAD0FA283F81964
	25 zeros	112724420	93,175	00000032301465AB62285BA17A6FC43CC5472FD3
	25 zeros	109847537	112,974	0000001AF6EF9B216BA456E554BDD144D4741B58
	25 zeros	54285406	52,702	0000001AA2DC9723A73736A8C82DB4667432B4FC
SHA-256	20 zeros	2808143	3,26	0000087654D76BEA49E48E69DE3D71B0CE398355FF0CAFB08156A61B858032A6
	20 zeros	4956959	4,361	00000611EEDFBA97BC073456B6DBEE0F4F2E1936E360173C8920B2A831C611A1
	20 zeros	5140739	5,372	000002D0412ABAD4FF12F591F326A887A7EEB630A661D7AE921BC229F5492197
	20 zeros	2441506	2,279	000007ACA59CDCFA05ECB4B50D107F88A557CAC0A0F51AF37D9E69F066024F24
	20 zeros	3040213	2,802	0000043BC19740B38F17914B5F8A08A7AAD5535894E49D62345B371C0CD5C4D6
	25 zeros	104598155	121,253	0000000A8E62AB77C359366016D3D9E384C282D663FC2560D9E928F4E306D59B
	25 zeros	146154885	127,542	0000004E4D4DAA910C4DC2AE90C18D6355EC7FF79F973C5EB45C384B89DBAE26
	25 zeros	71169137	65,855	0000002EB5D82BB8E8FD0A16B08F81C56537B6AC1AA232672209E5930F6CF408
	25 zeros	65583166	62,04	0000001080041B13ED18E1BE80D323B32F67225F55BDDFF604A87F80493E43D7
	25 zeros	70794776	67,537	0000000C4F68ED1E5796F1927C4A5B1A395BC5D5A215F31B89CD2ACD5AAA04E9

Fonte: do autor.

Os dados da tabela 47 são apenas demonstrados 5 de cada parametrização, enquanto foram realmente calculados 1000 para cada uma, o total foi omitido devido ao grande número de dados, pode ser encontrado na mídia digital anexa ao trabalho.

5.4 DISCUSSÃO DOS RESULTADOS OBTIDOS

Neste capítulo será abordado a visão da literatura sobre os resultados obtidos neste trabalho e como se convergem para uma mesma visão ou divergência de resultados.

Foram obtidos, quando estatisticamente significativos, resultados demonstrando um custo maior para SHA-256 do que para SHA-1, onde o MD5 é o mais veloz para processar, tendo em vista estes resultados, discute-se.

Como visto no capítulo 2 deste trabalho o MD5 é usado há décadas e já apresentou várias vulnerabilidades, desta forma, retém-se a conclusão de ser mais rápido de processar, todavia com custos em sua segurança, sendo desaconselhado pelo autor seu uso em temas tão sensíveis como transações financeiras.

Quanto ao SHA-256 e o SHA-1, como visto no artigo de Satoh e Inoue (2007) foi obtido o melhor desempenho entre os *Secure Hashes* no algoritmo SHA-1 no circuito integrado de aplicação específica, do inglês *application-specific integrated circuit* (ASIC), também obtendo bons resultados com o SHA-256, indo de encontro com os resultados obtidos neste trabalho, também é comentado sobre o desempenho do MD5, todavia a implementação do hardware para este algoritmo necessitou de especificidades que levaram a queda de eficiência do hardware, reforçando a tese de que é preferível utilizar um *Secure Hash* ao MD5.

Vale ressaltar que se pode utilizar um ASIC especificamente para mineração de *Bitcoin*, tendo em vista que foi desenvolvido um hardware específico para cálculo de *hashes* no artigo citado.

Desta forma, a literatura estudada para este trabalho se encontra nas conclusões ao obter o SHA-256 como mais indicado para a parametrização do problema do cálculo de *nonce* do *Bitcoin*, por apresentar maior desafio ao ser processado todavia mantendo a eficiência e menor vulnerabilidade quando comparado com o MD5.

6 CONCLUSÃO

Foram abordados neste trabalho análises estatísticas comparando amostras de cálculos de *hashes* com uma parametrização de número de bits zeros ao início do *hash*, os quais representam o cálculo do *nonce* utilizado para minerar um bloco de transação no *Bitcoin* de acordo com o trabalho de Nakamoto.

Estas análises foram feitas com os algoritmos SHA-256, SHA-1 e MD5, utilizando a parametrização de bits zeros de 20 zeros e 25 zeros. Foram analisados os algoritmos entre si com parametrização de 20 zeros, a seguir novamente entre si com parametrização de 25 zeros e por fim pares do mesmo algoritmo comparado entre parametrização de 20 zeros e 25 zeros, todos estes testes foram divididos entre quantidade de *hashes* calculados necessários para chegar em um resultado e tempo de processamento para mesmo fim.

Primeiramente, nas análises entre os algoritmos com parametrização de 20 zeros foi concluído:

a) para quantidade de *hashes* não existe diferença estatisticamente significativa;
b) para tempo de processamento se tem SHA-256 > SHA-1 > MD5.

Assim, nas análises entre os algoritmos com parametrização de 25 zeros foi concluído:

a) para quantidade de *hashes* não existe diferença estatisticamente significativa;
b) para tempo de processamento se tem SHA-256 > SHA-1 > MD5.

Por fim a comparação de mesmo algoritmo entre as parametrizações de 20 e 25 zeros foi concluído:

a) para quantidade de *hashes* se tem a parametrização de 25 zeros > 20 zeros em todos os três algoritmos;
b) para tempo de processamento de mesma forma se tem a parametrização de 25 zeros > 20 zeros em todos os três algoritmos.

Tem-se portanto a conclusão de que, dos algoritmos estudados, o SHA-256 apresenta a melhor escolha para o cálculo do problema de *nonce* do *Bitcoin*, aumentando o parâmetro de zeros a medida que o poder de hardware dos nós aumenta, é tida como solução suficiente para a manutenção da estrutura do desafio

na mineração da moeda até que se crie ou venha a ser necessário um novo algoritmo de cálculo de *hash* que não os abordados.

Desta forma podem ser apresentados maiores estudos futuros e aqui são apresentados visões de trabalhos que podem abordar tais temas:

a) Realização de testes com algoritmos de *hash* mais custosos que o SHA-256, sugerindo-se o SHA-512 ou demais derivados dos *Secure Hashes* com palavras maiores a 256 bits, bem como novos algoritmos não existentes até a presente data;
b) Estudos afim de, com os dados obtidos neste trabalho, prever matematicamente com margens aceitáveis de erro os custos que parametrizações maiores poderiam ocasionar;
c) Uso do software FlexSim para criar casos de testes e simulações entre os algoritmos;
d) Clusterização dos testes em alto desempenho;
e) Implementação de um protótipo minerador.

Também se pode verificar, devido ao custo de tempo ao processar com parametrização de 25 zeros, considera-se uma limitação a de não ser viável incrementar a parametrização para níveis mais altos, onde os trabalhos futuros descritos podem auxiliar com esta limitação.

REFERÊNCIAS

ANDRADE, R. S. **Algoritmo de Criptografia RSA**: análise entre a segurança e velocidade. Eventos Pedagógicos, Jequié, v. 3, n. 3, p. 438-457, 2012.

CABRAL, I. **Segurança da Informação em Bibliotecas Universitárias Federais:** Um levantamento sobre ferramentas e técnicas utilizadas. Florianópolis: Universidade Federal de Santa Catarina, 2015, 80p.

CAMPOS, J. L. **Avaliação de Desempenho dos Algoritmos de Criptografia:** Data Encryption Standard e Algoritmo Assimétrico (RSA). Criciúma: Universidade do Extremo Sul Catarinense, 2008, 105p.

DEVORE, J. L. **Probabilidade e Estatística para Engenharia e Ciências**. Tradução Joaquim Pinheiro Nunes da Silva. São Paulo: Pioneira Thomson Learning, 2006.

FIELD, A. **Descobrindo a estatística usando o SPSS**. Tradução Lorí Viali. 2 ed. Porto Alegre: Artmed, 2009. 688p.

FOROUZAN, B. A. **Data Communication and Networking**. Tradução nossa. 5 ed. Nova Iorque: McGrall-Hill, 2013. 1269p.

FORTE, D. **The Death of MD5**. Network Security, Elsevier, p. 18-20, 2009.

GOLDBERG, N.; LEYFFER, S.; SAFRO, I. **Optimal Response to Epidemics and Cyber Attacks in Networks**. Tradução nossa. Wiley Periodicals, 2015, 14p.

LITTLE, E. **Bitcoin**. Tradução nossa. The Investment Lawyer, v. 21, n. 5, 2014, 5p.

NAKAMOTO, S. **Bitcoin: A Peer-to-Peer Electronic Cash System**. 2008, 9 p.

O GLOBO. **Australiano revela que é o inventor do bitcoin:** Engenheiro exibe chaves criptográficas utilizadas na criação da moeda digital. Rio de Janeiro, 2016.

SATOH, A.; INOUE, T. **ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS**. tradução nossa. Integration, the VLSI Journal 40, Elsevier, p. 3-10, 2007.

SPIEGEL, M. E.; STEPHENS, L. J. **Estatística**. Tradução José Lucimar do Nascimento. 4. ed. Porto Alegre: Bookman, 2009.

STALLINGS, W. **Criptografia e segurança de redes:** Princípios e práticas. Tradução de Daniel Vieira. 4. ed. São Paulo: Pearson Education do Brasil, 2008. 494 p.

TANENBAUM, A. S. **Redes de Computadores**. 3. ed. Rio de Janeiro: Campus Ltda., 1997. 923 p.

TANENBAUM, A. S. **Redes de Computadores**. Tradução de Vanderberg D. de Souza. 4 ed. Rio de Janeiro: Elsevier Editora Ltda, 2003. 945 p.

ZAR, J. H. **Biostatistical Analysis**. Tradução Nossa. 5 ed. New Jersey: Pearson Hall, Inc. 2010.

APÊNDICE(S)

APÊNDICE A – Hashcash: Contramedida para a negação de serviço

O *hashcash* foi originalmente proposto como um mecanismo de redução de abuso sistemático de recursos da web não mensuráveis, como o email e encaminhadores anônimos (*anonymous remailers*) em maio de 1997. Passados cinco anos, este documento agrupa várias aplicações, melhorias sugeridas e publicações subsequentes relatadas, e descreve experiências iniciais de uso do *hashcash*.

A função-custo de CPU do *hashcash* processa um token que pode ser usado como prova-real. Variantes interativas e não-interativas da função-custo podem ser construídas, o que pode ser usado quando o servidor pode requisitar um desafio (conexões orientadas a protocolo interativo), e onde não o pode (onde a comunicação é *store-and-forward*, ou orientada a pacotes) respectivamente.

Na época da publicação original, em 1997, o autor desconhecia o trabalho prévio de Dwork e Naor em *Pricing via processing or combatting junk mail*, 1992, o qual propusera uma função de precificação para o custo de CPU das aplicações que combatem emails considerados lixo eletrônico. Subsequentemente, aplicações de função custo foram ainda discutidas por Juels e Brainard em *Client puzzles: A cryptographic countermeasure against connection depletion attacks*, 1999. Jakobsson e Juels propuseram um duplo propósito para o trabalho gasto em funções-custo: para além disso executar um cálculo de outro método útil em *Proofs of work and bread pudding protocols*, 1999.

Funções-Custo

Uma função-custo deve ser facilmente verificável, todavia difícil de processar (esta dificuldade deve ser parametrizável). Usa-se a seguinte notação para definir uma função-custo.

Dentro do contexto de funções-custo, definimos *client* para se referir ao usuário que computa o *token* (denotado τ) usando uma função-custo MINT() cujo uso é criar *tokens* para participar em um protocolo com um *server*. Usa-se o termo *mint* para a função-custo por causa da analogia entre criar *tokens* de custo e a cunhagem (do ingles *minting*) de moedas físicas.

O *server* irá verificar o valor do *token* utilizando uma função VALUE() e só prosseguirá com o protocolo se o *token* tiver o valor requerido.

As funções são parametrizadas pela quantidade de trabalho ω que o usuário terá de gastar em média para cunhar um *token*.

Através de funções-custo interativas, o *server* solicita um desafio $\mathbb{C}$ para o *client*; o *server* utiliza uma função CHAL() para computar o desafio. A função de desafio também é parametrizada pelo fator trabalho.

$$\begin{cases} \mathbb{C} \leftarrow CHAL(s, \omega) \\ \tau \leftarrow MINT(\mathbb{C}) \\ V \leftarrow VALUE(\tau) \end{cases}$$

Já nas funções não interativas, o *client* escolhe seu próprio desafio ou inicia randomicamente um valor para MINT(), não existindo a função CHAL().

$$\begin{cases} \tau \leftarrow MINT(s, \omega) \\ V \leftarrow VALUE(\tau) \end{cases}$$

Logicamente, uma função não interativa pode ser usado em uma configuração interativa, mas o inverso não é válido.

Publicamente auditável, custo probabilístico

a) Uma função-custo publicamente auditável pode ser eficientemente verificável por qualquer terceiro sem precisar ter acesso a nenhuma função de direção única (*trapdoor*) ou informação secreta. (Quando dito que uma função-custo é publicamente auditável, refere-se implicitamente que a função é eficientemente publicamente auditável quando comparada ao custo de cunhar um *token*, ao invés de auditável no sentido mais baixo em que o auditor pode repetir o trabalho feito pelo *client*);
b) uma função-custo de custo fixo possui um número fixo de recursos para computar. O mais rápido algoritmo para cunhar um *token* de custo fixo é um algoritmo determinístico;
c) uma função-custo de custo probabilístico é onde o custo para cunhar o *token* possui um tempo previsível esperado, todavia se torna um tempo mais randômico partindo do princípio que o *client* pode computar mais eficientemente a função-custo por iniciar de um número randômico.

Algumas vezes o *client* pode ter sorte e iniciar de um valor próximo da solução da função; existem dois tipos de custo probabilístico: o custo probabilístico restrito e o custo probabilístico irrestrito:

1. o custo probabilístico irrestrito de uma função-custo pode, em teoria, nunca terminar de processar, apesar de que as chances do tempo aumentar significantemente em relação ao esperado diminui a quase zero. Um exemplo seria uma função-custo de ser requerido tirar cara ao jogar uma moeda honesta; em teoria, o usuário poderia ter o azar de acabar por obter várias vezes o lado da coroa, entretanto, na prática a probabilidade de não atingir o lado cara em k jogadas tende a 0 rapidamente tendo $\lim_{k\to\infty}\left(\frac{1}{2}\right)^k = 0$;
2. com o custo probabilístico restrito de uma função-custo existe um limite do quanto de azar um *client* pode ter em sua busca pela solução; por exemplo quando o *client* tem de encontrar um espaço chave para uma solução conhecida; o tamanho do espaço chave impõe um limite no custo de buscar a solução.

Livre de *trapdoors*

Uma desvantagem de uma função-custo de solução conhecida é que o desafiado pode facilmente criar *tokens* de valores arbitrários. isso se opõe a uma auditoria pública onde o *server* pode ter um conflito de interesses, por exemplo em um contador web de cliques, o *server* pode ter o interesse de inflar o numero de cliques em uma página onde está sendo pago por cliques por um anunciante.

uma função livre de *trapdoor* é aquela em que o *server* não recebe vantagens ao cunhar um *token*.

Um exemplo de função-custo livre de *trapdoor* é o h*ashcash*. A função-custo *client-puzzle* de Juels e Brainard é um exemplo de função de solução conhecida onde o *server* possui vantagem em cunhar *tokens*. *Client-puzzles* como especificado no documento não são publicamente auditáveis, apesar de ser por causa da otimização de armazenamento e não nativo de sua arquitetura.

A função-custo Hashcash

O *hashcash* é uma função-custo não-interativa, publicamente auditável, livre de *trapdoors* com custo probabilístico irrestrito.

Primeiramente se introduz uma notação: considerando o bitstring $s = \{0,1\}*$, define-se $[s]_i$ para significar o bit em *offset* i, onde $[s]_1$ é o bit mais à esquerda e $[s]_{|s|}$ é o bit mais à direita. $[s]_{i\ldots j}$ significa a substring de bits entre e incluindo os bits i e j, $[s]_{i\ldots j} = [s]_i \| \ldots \| [s]_j$. Então se tem $s = [s]_{1\ldots|s|}$.

Define-se um operador comparador infixo binário $\overset{esq}{=}b$ onde b é o tamanho da substring esquerda das duas substrings de bits.

$$x \overset{esq.}{=} 0\, y \quad [x]_1 \neq [y]_1$$

$$x \overset{esq.}{=} b\, y \quad \forall_{i=1\ldots b}\, [x]_i = [y]_i$$

Hashcash é computado relativo ao *server* nominal *s*, para prevenir *tokens* cunhados para um *server*, usados em outro (*servers* apenas aceitam *tokens* cunhados para o seu *server* nominal). A nomenclatura do *server* pode ser qualquer bit de string que unicamente define o *server* (ex. IP, email, etc).

A função hashcash é definida como (note que é uma versão melhorada da inicial, como descrita em um capítulo posterior deste apêndice):

$$\begin{cases} PUBLICO: & função\ hash\ H(\cdot) com\ saída\ k\ bits \\ \tau \leftarrow MINT(s,\omega) & \boldsymbol{encontre}\ x\ \in R\ \{0,1\} *\ \boldsymbol{st}\ H(s||x) \overset{esq.}{=} \omega 0^k\ \boldsymbol{retorne}\ (s,x) \\ V \leftarrow VALUE(\tau) & H(s||x) \overset{esq.}{=} v 0^k\ \boldsymbol{retorne}\ v \end{cases}$$

A função-custo *hashcash* é baseada em encontrar *hashes* parciais de colisão em todos os bits 0 na string de k bits 0^k. O mais rápido algoritmo para calcular colisões de *hash* é a força bruta. Não há desafio, portanto o *client* pode escolher seu próprio desafio randômico, portanto o *hashcash* é livre de *trapdoor* e não-interativo. Ainda mais, o *hashcash* é publicamente auditável, ou seja, qualquer um pode eficientemente verificar qualquer *token* publicado. (Na prática $|x|$ deve ser escolhido para ser grande o suficiente para fazer a probabilidade de *clients* reusarem um valor de partida igual ser desprezível; $|x| = 128$ bits deve ser suficiente, mesmo para servidores ocupados).

O *server* precisa manter um banco de dados de duplo-gasto para *tokens* já gastos, para detectar e recusar tentativas de gastar o mesmo *token* novamente. Para prevenir que esse banco de dados cresça indefinidamente, o *server* pode armazenar a data em que o *token* foi cunhado, isso possibilita a exclusão deste registro após sua data de expiração. Um período de expiração razoável deve ser tomado levando em consideração erros em horários, tempo de processamento e tempo de espera em transmissões.

Hashcash foi originalmente proposto como contra medida a *spam* de email, abuso sistemático e reencaminhamento anônimo de mensagens. É necessário utilizar funções-custo não interativas para estes casos por conta de não existir um canal para um desafio ser enviado. Todavia uma vantagem de métodos interativos é ser possível a prevenção de ataques pré-computados. Por exemplo, se existe um custo associado ao envio de cada email, isto pode ser suficiente para limitar o numero de abusos por email perpetuados por *spammers*; entretanto, para ataques puramente motivados por DoS, um atacante pode gastar um ano pré-computando *tokens* para todos serem válidos em um único dia, e neste dia ser capaz de sobrecarregar o sistema.

É possível reduzir o escopo de ataques pré-computados utilizando um sinalizador que muda de valor (*broadcast* não previsível e não autenticável de valores mudando de acordo com o tempo), digamos por exemplo os números da loteria desta semana. Neste evento o sinalizador atual é incluído no início da *string*,

limitando os ataques pré-computados a este intervalo de tempo em que o sinalizador muda.

Hashcash interativo

Com uma forma interativa de *hashcash*, para uso em configurações de conexão estabelecida como: TCP, TLS, SSH, IPSEC dentre outros, um desafio é eleito pelo *server*. O objetivo do *hashcash* interativo é defender os recursos de um *server* de esgotamento prematuro, além de prover uma atrativa degradação de serviço com uma justa alocação entre os usuários quando enfrentando um ataque de DoS. Em casos de protocolos de segurança como TLS, SSH e IPSEC com fases de conexões custosamente estabelecidas envolvendo criptografia de chave pública o recurso a ser defendido é o tempo de CPU do *server*.

A função *hashcash* interativa é definida por:

$$\begin{cases} C \leftarrow CHAL(s,\omega) & \textbf{escolha } c \in R\ \{0,1\}^k \textbf{ retorne } (s,\omega,c) \\ \tau \leftarrow MINT(C) & \textbf{encontre } \boldsymbol{x} \in R\ \{0,1\}^* \textbf{ st } H(s||c||x) \overset{esq.}{=}_{\omega} 0^k \ \textbf{retorne } (s,x) \\ V \leftarrow VALUE(T) & H\ (s||c||x) \overset{esq.}{=}_{v} 0^k \textbf{ retorne } v \end{cases}$$

Regulagem dinâmica

Com *hashcash* interativo é possível dinamicamente ajustar fator de trabalho requerido para o *client* baseado na carga de CPU do *server*. Esta abordagem também admite *hashcash* interativos com desafios-resposta serão apenas usados em períodos de alta carga. Isto faz com que seja possível criar protocolos resistentes ao DoS sem quebrar a compatibilidade com softwares mais antigos. Em períodos de alta carga, *clients* sem o conhecimento de *hashcash* serão impossibilitados de conectar ou então redirecionados para conexões limitadas, mais antigas e menos eficientes contra DoS, como perda de conexão randômica.

Cookie hashcash

Com ataques de esgotamento de *slots* de conexão, como o ataque *syn-flood*, e de conexões de via única, como o TCP, o recurso do *server* que está sendo consumido é espaço livre para o TCP armazenar estado por conexão.

Neste cenário pode ser desejável evitar manter um estado por conexão, até que o *client* tenha computado um *token* com uma função-custo *hashcash* interativa. Essa defesa é similar ao *syn-cookie* do ataque *syn-flood*, mas aqui se propõe impor um adicional de custo de CPU na máquina de conexão para reservar um *slot* de conexão TCP.

Para evitar guardar o desafio no estado da conexão (que consome espaço por si só) o *server* pode escolher computar um MAC chaveado que de outra forma armazenaria e o enviar ao *client* como parte do desafio, podendo assim verificar a autenticidade do desafio e do *token* quando o *client* os retornar. (Esta técnica genérica - de enviar um registro que de outra forma seria armazenado junto a um MAC para a entidade cuja informação é sobre - é denominada como certificado de chave simétrica.) Esta abordagem é análoga à técnica usada em *syn-cookie*, ainda Juels e Brainard propuseram uma abordagem relacionada, todavia em um nível de protocolo de aplicação, em seu *client-puzzle*.

Por exemplo, com uma função MAC chaveada M pela chave do *server* K o desafio do MAC pode ser expresso por:

$$\begin{cases} PUBLICO: & \qquad\qquad\qquad\qquad Função\ MAC\ M(\cdot,\cdot) \\ C \leftarrow CHAL(\omega) & \boldsymbol{escolha}\ c \in R\ \{0,1\}^k\ \boldsymbol{compute}\ m \leftarrow M\big(K, t\ ||s||p||\omega||c\big) \\ & \boldsymbol{retorne}\ (t, s, p, \omega, c, m) \end{cases}$$

O *client* deve enviar o MAC m, o desafio c e os parâmetros do desafio p com a resposta *token* para que o *server* possa verificar o desafio e a resposta. O *server* também deve incluir no MAC os parâmetros de conexão, o mínimo suficiente para identificar o *slot* de conexão e algumas vezes mesurando ou aumentando o valor t para que respostas de desafios anteriores não possam ser coletadas e reusadas depois que os *slots* de conexões estejam livres. O desafio e o MAC serão enviados em uma mensagem resposta TCP SYN-ACK, e o *client* irá incluir o *token* do *hashcash* interativo (desafio-resposta) no TCP ACK da mensagem. Assim como *syn-cookies*, o *server* pode não precisar manter nenhum estado de conexão anterior ao recebimento do TCP ACK.

Para retrocompatibilidade com *syn-cookies* sencientes de TCP, um *hashcash-cookie* senciente de TCP deve somente ativar *hashcash-cookies* quando detectar que foi sujeito a um ataque de depreciação de conexão TCP. Argumentos similares aos dados por Dan Bernstein em *syn-cookies* podem ser usados para mostrar que retrocompatibilidade é mantida, nominalmente sobre ataques *syn-flood* o argumetno de Bernstein mostra como prover retrocompatibilidade com implementações não *syn-cookies* sencientes; similarmente sobre ataques de depreciação de conexão, *hashcash-cookies* são somente ativados ao ponto onde o serviço não seria de outro modo disponível para um não *hashcash-cookie* senciente de TCP.

Enquanto o *flood* aumenta em severidade, o algoritmo de *hashcash-cookie* pode incrementar o tamanho de colisões necessárias para estar em uma mensagem TCP ACK. O *client hashcash-cookie* senciente ainda pode conectar com uma chance mais justa contra o atacante de DoS, presumindo que o atacante possui recurso de CPU limitado. O atacante DoS irá efetivamente opor seu CPU a qualquer outro (*hashcash-cookie* senciente) *client* que tente se conectar. Sem uma defesa de *hashcash-cookie* o atacante DoS pode *floodar* o *server* com estabelecimento de conexões e pode mais facilmente obter todos os *slots* por completar n conexões por time-out de conexão ociosa, onde n é o tamanho da tabela de conexão, ou pingar as conexões por time-out de conexão ociosa para convencer o *server* de que estão ativas.

Conexões serão entregues coletivamente para os usuários em crua proporção de poder de CPU, ou seja, sua justa proporção é baseada em recurso de CPU (presumindo que cada usuário está tentando abrir o máximo de conexões possíveis) então o resultado será favorável aos *clients* que possuírem processadores mais rápidos assim como podem processar mais *tokens* desafio-resposta de *hashcash* interativo por segundo.

Melhorias no hashcash

No esquema inicialmente publicado do *hashcash*, a string alvo é encontrar um hash de colisão justo por usar o hash do nome do serviço (e respectivamente o nome do serviço e o desafio em *hashcash* interativo). Uma subsequente melhoria foi independentemente sugerida por Hal Finney e Thomas Boschloo para o *hashcash* encontrar uma colisão de uma saída de string fixa. A observação deles é que uma colisão de uma string fixa é de mesma forma justa, mais simples e reduz o custo do fator de verificação em 2. Uma string fixa que é conveniente para comparar com tentativas de colisões é a string k-bit 0^k onde k é o tamanho de saída do hash.

Baixa variância

Tokens de funções-custo idealmente devem tomar um número de recursos computacionais previsível para serem computados. A construção do *client-puzzle* de Juels e Brainard provê um custo-restrito probabilístico por solicitar desafios com soluções conhecidas, todavia enquanto isto limita o tempo do pior caso teórico. A técnica de usar soluções conhecidas também não é aplicável para a configuração não interativa. É uma questão aberta se existe custo-restrito probabilístico, ou funções-custo não interativas de custo fixo com a mesma ordem e magnitude da verificação de custo como o *hashcash*.

A outra mais relevante melhoria devido a Juels e Brainard é a sugestão de usar múltiplos *sub-puzzles* com o mesmo custo esperado, mas com um custo em baixa variância. Esta técnica deve ser aplicável tanto para métodos interativos quanto para não interativos variantes do *hashcash*.

DoS distribuido e não paralelização

Roger Dingledine, Michael Freedman e David Molnar colocaram em moção o argumento de que funções-custo não paralelizáveis são menos vulneráveis a ataques distribuídos de DoS (DDoS) no capítulo de 16 de *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. Seu argumento é de que funções-custo não paralelizáveis frustram DDoS por conta do atacante não conseguir subdividir e cultivar o trabalho de computar um *token* individual.

O autor descreve uma função-custo de custo fixo usando o *puzzle time-lock* de Rivest, Shamir e Wagner, que também acontece de ser não paralelizável. A função-custo *puzzle time-lock* pode ser usada em configuração interativa e não interativa, assim como é seguro para o usuário escolher seu próprio desafio. A aplicabilidade desta função como função-custo foi também subsequentemente observada por Dingledine et. al.

Define-se uma função-custo *puzzle time-lock* baseada em custo fixo e não interativa como:

$$\begin{cases} \substack{PUBLICO: n=pq \\ PRIVADO: primes\ p\ and\ q,\ \emptyset(n)-(p-1)(q-1)} \\ C \leftarrow CHAL(s,\omega)\ \boldsymbol{escolha}\ c \in R\ [0,n)\ \boldsymbol{retorne}\ (s,c,\omega) \\ \tau \leftarrow MINT(C)\ \boldsymbol{computar}\ x \leftarrow H(s||c)\ \boldsymbol{computar}\ y \leftarrow x^{x^{\omega}}\ (mod\ n)\ \boldsymbol{retorne}\ (s,c,w,y) \\ V \leftarrow VALUE(\tau)\ \substack{\boldsymbol{computar}\ x \leftarrow H(s||c)\ \boldsymbol{computar}\ z \leftarrow x^{\omega}\ mod\ \emptyset(n) \\ \boldsymbol{se}\ x^{z}=y\ (mod\ n)\ \boldsymbol{retorne}\ \omega\ \ \boldsymbol{senao}\ \boldsymbol{retorne}\ 0} \end{cases}$$

O *client* não conhece $\emptyset(n)$, então o mais eficiente método para o *client* calcular MINT() é exponenciação repetida, cuja requer exponenciações ω. O desafiador sabe $\emptyset(n)$, o que o possibilita uma computação mais eficiente por reduzir o expoente $mod\emptyset(n)$, desta forma, o desafiador pode executar VALUE() com duas exponenciações modulares. O desafiador, como consequência, possui uma *trapdoor* em computar a função-custo por poder calcular MINT() eficientemente utilizando o mesmo algoritmo.

Argumenta-se portanto, que a proteção provida ao DDoS pela não paralelização da função-custo é marginal: a não ser que o *server* restrinja o número de desafios para um *client* reconhecível unicamente, o atacante DDoS pode cultivar múltiplos desafios tão facilmente como cultiva um único subdividido, consumindo recursos do *server* de mesma forma. De mesma forma, não é muito difícil para um usuário mascarar ser vários *clients* para um *server*.

Considera-se ainda que o atacante DDoS geralmente possui, devido à natureza de seu método de comandar nós com o mesmo número de nós conectados a uma rede a sua disposição como processadores. Ele pode portanto, em qualquer caso, ter cada nó atacante participando diretamente e indescritivelmente do protocolo normal de qualquer usuário legítimo. Essa estratégia de ataque é também otimizada de qualquer forma assim como os nós de ataque apresentarão uma variada gama de endereços de origem, que irá frustrar tentativas de estratégias justas de controle por conexão e roteadores baseados em contra medidas DDoS baseados em volume de tráfego sobre alcance de endereços de IP. Por conta disso, a combinação de padrões de funções-custo não paralelizáveis para o atacante natural oferece resistência adicional limitada.

Assim como o argumento contra a eficácia prática e valor de funções-custo não paralelizáveis. Tais funções, quando comparadas as mesmas mais atuais tem possuído ordem de magnitude mais devagar, isto se deve por serem relacionadas a *trapdoors* de criptografia de chave pública ser nativamente menos eficiente. É uma questão aberta se existe uma função-custo não paralelizável

baseada em chaves simétricas (ou públicas) com funções de verificação de mesma magnitude das funções-custo baseadas em criptografia simétrica.

Enquanto a aplicação de *puzzle time-lock* para funções-custo, uma chave pública de tamanho reduzido pode ser utilizada para aumentar a velocidade da função de verificação, esta abordagem apresenta um risco que o módulo será fatorado com o resultado de que o atacante ganha uma grande vantagem ao cunhar *tokens*. (Nota: Fatoração é por si só um processo amplamente paralelizável)

Para combater isso, o *server* pode trocar os parâmetros públicos periodicamente. Todavia neste caso particular do parâmetro público usado pelo *puzzle time-lock* (que é o mesmo utilizado no modulo RSA, usado na encriptação), esta operação é por si só moderadamente custosa, por isso não seria executada tão frequentemente. Provavelmente não seria aconselhável liberar softwares baseados em chaves com menos de 768 bits para esta aplicação, em adição ajudaria a troca de chaves periodicamente, digamos a cada hora.

A função custo *puzzle time-lock* também possui necessariamente *trapdoor*, uma vez que o *server* precisa de uma chave de verificação privada para permitir verificar eficientemente os *tokens*. A existência de uma chave de verificação apresenta um risco adicional de comprometimento da chave, permitindo ao atacante burlar a função-custo de proteção (a função-custo *hashcash* interativa é livre de *trapdoor*). Se de fato a chave de verificação for comprometida, pode ser substituída, todavia adiciona complexidade por conta deste evento ter de ser detectado e manualmente intervencionado, ou algum gatilho automático precisa ser implementado para a detecção e substituição da chave.

A função custo *puzzle time-clock* também tenderá a ter mensagens maiores uma vez que existe a necessidade de comunicar planejadamente e parâmetros públicos de rechaveamento emergencial. Para algumas aplicações, por exemplo os protocolos *syn-cookie* e *hashcash-cookie*, espaço está em um dever premiado para retrocompatibilidade e o tamanho do pacote é restritivamente imposto pela infraestrutura da rede.

Por fim, em síntese, argumenta-se que funções-custo não paralelizáveis são de valor prático questionável em proteger contra ataques DDoS, possui funções de verificações mais custosas, inclui o risco de comprometimento da chave de verificação e complexidade de a substituir, possui mensagens maiores e

significantemente mais complexas de implementar. Recomenda-se, portanto, funções com protocolo *hashcash* mais simples.

Aplicações

Além da proposta inicial do protocolo *hashcash* estrangular DoS contra redes de reenvio e reter *spam* de email, desde a publicação desta referência, as aplicações abaixo foram discutidas, exploradas e em alguns casos implementadas e distribuídas.

a) *hashcash-cookies*, uma potencial extensão de *syn-cookies;*
b) *hashcash* interativo;
c) *hashcash* usado para estrangular DoS em systemas como o Freenet;
d) *hashcash* usado para supressão de requisições de serviço em sistema de arquivo *self-certifying*;
e) *hashcash* usado para supressão de USENET *flooding*;
f) *hashcash* usado como uma mecanismo de cunhagem de moedas sem interferência de bancos.

Esquema de classificação de funções custo

Lista-se a classificação das características das funções custo, possuindo a seguinte notação para denotar suas propriedades:

$$(\left[e = \left\{1, \frac{1}{2}, 0\right\}\right], \left[\sigma = \left\{0, \frac{1}{2}, 1\right\}\right], [\{i, i'\}], [\{a, a'\}], [\{t, t'\}], \{[p, p'\}]$$

Onde e é a eficiência: valor $e = 1$ significa eficiência verificável. Verificável com o custo comparável para igual ou menor que o custo de verificar construções de chaves simétricas como o *hashcash* que consome uma rodada de compressão de uma função de compressão interativa baseada em função hash como SHA-1 ou MD5. Valor $e = \frac{1}{2}$ significa praticamente verificável, sendo menos verificável que o eficientemente verificável, todavia ainda suficientemente eficiente na prática de algumas aplicações, por exemplo, o autor considera o *puzzle time-lock* pertencente a esta categoria. Valor $e = 0$ significa verificável mas impraticável, que a função-custo

é verificável, todavia a função é tão impraticavelmente lenta que a sua existência serve apenas como conceito de prova para ser melhorada para uso prático.

σ é a caracterização do desvio padrão. Temos $\sigma = 0$ para custo fixo, $\sigma = \frac{1}{2}$ para custo probabilístico restrito e $\sigma = 1$ para custo probabilístico irrestrito.

i denota que a função custo é interativa enquanto i' denota sua não interatividade

a denota que a função é publicamente auditável, enquanto a' denota não publicamente auditável.

t denota a existência de *trapdoor*, enquanto t' denota ausência de *trapdoor*.

p denota que a função é paralelizável, enquanto p' denota não paralelismo.

	trapdoor-free	trapdoor
interactive	hashcash $(e = 1, \sigma = 1, i, a, \bar{t}, p)$	client-puzzles $(e = 1, \sigma = \frac{1}{2}, i, a, t, p)$ time-lock $(e = \frac{1}{2}, \sigma = 0, i, \bar{a}, t, \bar{p})$
non-interactive	hashcash $(e = 1, \sigma = 1, \bar{\imath}, a, \bar{t}, p)$	time-lock $(e = \frac{1}{2}, \sigma = 0, \bar{\imath}, \bar{a}, t, \bar{p})$

Problemas em aberto

a) Existência de função-custo eficientemente verificável, não interativa e de custo fixo($e = 1, \sigma = 0, i'$), tal qual com custo probabilístico restrito ($e = 1, \sigma = \frac{1}{2}, i'$);

b) Existência de função-custo eficientemente verificável, não interativa e não paralelizável ($e = 1, i', p'$), tal qual com interatividade ($e = 1, i, p'$);

c) Existência de função-custo publicamente auditável, não interativa e de custo fixo ($\sigma = 0, i', a$), tal qual com custo probabilístico restrito ($\sigma = \frac{1}{2}, i', a$).

APÊNDICE B - ARTIGO

Análise de Técnicas de Criptografia Envolvidas na Utilização do Bitcoin

Diego M. Silva[1], **Valter Blauth Júnior**[1]

[1]Departamento de ciência da Computação – Universidade do Extremo Sul Catarinense (UNESC)
Criciúma – SC – Brazil

diegomottas@gmail.com, valterblauth@unesc.net

Abstract. *The* Bitcoin *presents a challenge called* nonce *to all concurrent nodes to obtain acceptance of the calculated block at the mining process, this challenge is the calculation of a* hash *parametrizated with a certain number of zero bits at the start of the* hash*, this arcticle calculates the* hashes *in a prototype followed by a analysis using three* hash *algorithms: SHA-256, SHA-1 and MD5, that's a statistic analysis using the IBM SPSS software, presenting conclusion about the quantity of* hashes and the time *necessary to each algorithm calculate a* nonce.

Resumo. *O* Bitcoin *apresenta um desafio chamado* nonce *para os nós concorrentes resolverem afim de obter aceitação do bloco calculado ao minerar, este desafio nada mais é do que um cálculo de* hash *parametrizado com uma quantidade de bits zero ao início do mesmo, desta forma, este trabalho realiza o cálculo de* hashes *em um protótipo e logo uma análise utilizando três algoritmos de* hashes *sendo eles: SHA-256, SHA-1 e MD5, trata-se de uma análise estatística com auxílio do software IMB SPSS apresentando conclusões sobre a quantidade de* hashes *e o tempo necessários para cada algoritmo calcular um* nonce.

• 1. Introdução

O *Bitcoin* é uma moeda corrente eletrônica com visão descentralizada, ou seja, livre de influências de autoridades reguladoras; implementada usando criptografia e tecnologia *peer-to-peer,* documentada em 2008 e desenvolvida no ano seguinte pelo pseudônimo Satoshi Nakamoto, não sendo certa a sua real identidade até pouco tempo (LITTLE, 2014).

Em junho de 2016, o australiano Craig Wright se sentiu forçado a revelar sua identidade de principal criador do *Bitcoin* para algumas mídias internacionais por causa da perseguição de algumas pessoas sobre serem possíveis inventores do *Bitcoin*, afirma que nunca foi de sua vontade obter qualquer tipo de fama, apenas seguiu sua ideologia liberal (GLOBO, 2016).

Sobre formas de uso, o *Bitcoin* pode ser utilizado como qualquer moeda física, basta ter uma carteira e obter um valor em moeda, podendo ser minerando ou por qualquer transação envolvendo dois usuários (venda de produtos ou pagamento de serviços por exemplo), tendo saldo em carteira se pode realizar compras de qualquer natureza, desde que o vendedor aceite a moeda como pagamento. Outra forma de uso observada no mercado econômico é o investimento em *Bitcoin*, de mesma forma como comprar uma moeda estrangeira afim de lucrar, pode-se comprar *Bitcoin* e vender após ter sido valorizado, apresentando os riscos e custos que mercado acomoda.

Para o *Bitcoin* funcionar, o uso de criptografia é mandatório e o seu desempenho deve estar de acordo com as expectativas; como foi abordado neste trabalho, a geração de um bloco não

pode ser muito rápida e nem muito lenta, ela deve possuir o tempo ideal e isto depende diretamente de como a estratégia lida com os algoritmos utilizados; para tal, uma visão constante do desempenho dos algoritmos envolvidos é essencial.

- **2. O Bitcoin**

A utilização de *Bitcoin* se iniciou em 2008, em meio à crise mundial econômica, tomando sua real proeminência global em 2012, ficando sob holofotes desde então. A capacidade de processamento coletiva somada do *Bitcoin* chega a ser cem vezes superior ao desempenho da lista dos 500 melhores supercomputadores do mundo somados, mais de 50 mil petaflops (operações de ponto flutuante). Não $50.000\ x\ 10^{15}$ obstante, o sucesso do *Bitcoin* mostrou alguns problemas, sendo estes: Não é tão seguro e anônimo quanto se pensava, o sistema de distribuição está ficando pesado e levou a uma "corrida armamentista" insustentável em busca de melhores máquinas mineradoras (*Bitcoin under pressure*, 2013).

Aproveitando a conexão do artigo *Bitcoin under pressure* com a guerra fria, é possível fazer um comparativo da mineração de dados do *Bitcoin* com o avanço tecnológico durante as guerras; afinal, foi na necessidade da guerra que Alan Turing iniciou seu trabalho para o serviço secreto britânico, onde sua pesquisa lhe rendeu o título de "o pai da computação". Assim também como a própria corrida armamentista trouxe inovações tecnológicas usadas pela humanidade até hoje (SATO, 2000; HODGES, 2001).

Desta forma, tem-se o desafio de manter a estabilidade do *Bitcoin*, para tal, é de vital importância analisar constantemente os métodos de criptografia que envolvem a economia da moeda virtual. Nakamoto (2008), criador do *Bitcoin*, comenta em seu artigo sobre o aumento automático da complexidade do cálculo ao incrementar o número de zeros necessários do *nonce* calculado para o bloco, este *nonce* é um *hash* calculado através do algoritmo SHA-1 e todo o processo de mineração depende desta técnica para que a geração de blocos não seja nem muito demorada e nem rápida demais.

Baseado inteiramente em dois pilares: a economia e a criptografia, o *Bitcoin* caminha sobre as incertezas do mercado com toda a segurança que o poder de processamento de seus nós pode oferecer através da criptografia. A certeza que as partes precisam para realizar suas transações está confiada aos processos eletrônicos e estes precisam sempre se provar preparados para o mercado, através de estatísticas que passem confiança. É com essa ideologia que este trabalho é escrito, para demonstrar se os algoritmos selecionados por Nakamoto em 2008 conseguem atingir, quase dez anos depois, as espectativas.

- **3. Protótipo**

A configuração do banco de dados e instalação de programas de suporte secundários não serão abordados por conta de existirem tutoriais oficiais dos fabricantes de como realizar as instalações e configurações dos mesmos.

Após desenvolvido e configurado, o primeiro passo para utilizar o protótipo é subir a aplicação utilizando o *Spring-boot*, através do comando Maven “mvn spring-boot:run” como demonstrado na figura 1, a partir disto, informa-se no endereço do navegador o endereço informado pelo console.

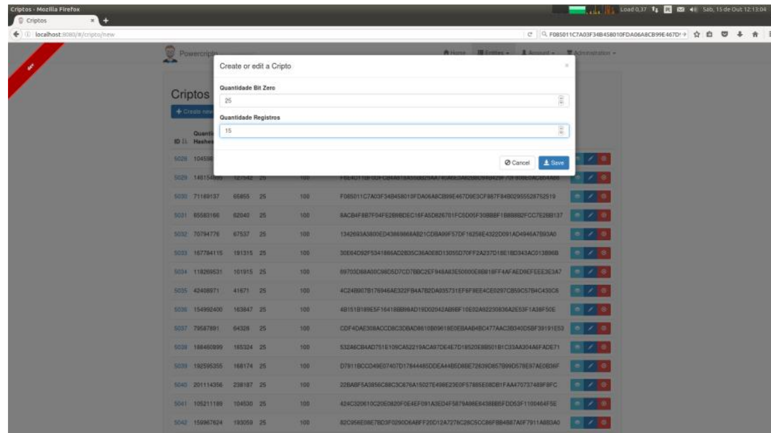
Figura 1 – Console *Spring-boot*

Por se tratar de um protótipo, foi rodado localmente, o que não impede a configuração para acesso via web normalmente. Desta forma basta abrir um navegador web qualquer e acessar a porta 8080 da máquina local, abrindo a tela demonstrada na figura 2 no menu "*Entities*" existe o item "*Cripto*" onde este abrirá a tela de cadastro, como explanado a seguir.

Figura 2 – Página inicial JHipster

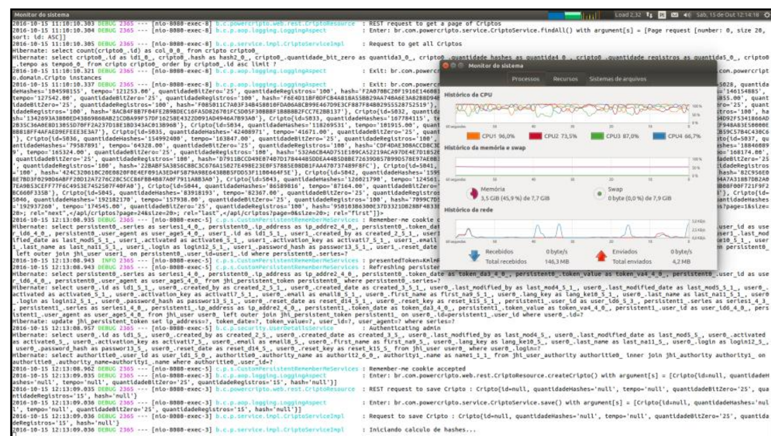
Quando aberto, o diálogo de geração aparecerá sobre a tela como demonstrado na figura 3, nele é solicitado a quantidade de bits zeros ao fim do hash que será necessário e a quantidade de hashes que terão de ser encontrados com esta quantidade de bits zeros, quando salvo, iniciam-se os cálculos.

Figura 3 – Diálogo de geração



Na figura 4 podemos ver pelo console que o cálculo de *hashes* foi iniciado, e pode ser comprovado pelo monitor de recursos do computador onde demonstra que todos os núcleos do processador estão sendo utilizados, assim se pode verificar como estes cálculos estão sendo feitos.

Figura 4 – Processamento de dados



Na figura 5 se tem a maneira utilizada para iterar um *array* de bytes, onde este array é o *hash* calculado, afim de poder verificar o número de zeros ao fim do hash. Esta classe java é a implementação de um *iterator* da mesma forma como já funciona o *iterator* nativo das *Collections* java, com os métodos *next()* e *hasNext()*, respectivamente servindo para pegar o valor do próximo bit e para verificar se existe um próximo bit no *array*.

Figura 5 – Iteração de bits

```java
package br.com.powercripto.service.util;

import java.util.Iterator;

/**
 * Diego 03/08/16.
 * Originally taken from StackOverflow website
 */
public class ByteArrayBitIterable {
    private final byte[] array;

    public ByteArrayBitIterable(byte[] array) {
        this.array = array;
    }

    public Iterator<Boolean> iterator() {
        return new Iterator<Boolean>() {
            private int bitIndex = 0;
            private int arrayIndex = 0;

            @Override
            public boolean hasNext() { return (arrayIndex < array.length) && (bitIndex < 8); }

            @Override
            public Boolean next() {
                Boolean val = (array[arrayIndex] >> (7 - bitIndex) & 1) == 1;
                bitIndex++;
                if (bitIndex == 8) {
                    bitIndex = 0;
                    arrayIndex++;
                }
                return val;
            }

        };
    }

}
```

Na figura 6 se tem a classe "*CriptoUtil*" que por sua vez utiliza a classe demonstrada na figura 5 em um de seus métodos e possui outros de uso essencial para o processamento, os métodos são: "verificaQuantidadeBitsZero", "gerarNovoHash" e "bytesToHex".

Figura 6 – Classe de utilidades

```java
package br.com.powercripto.service.util;

import ...

/**
 * Diego 03/08/16.
 */
public class CriptoUtil {

    public static boolean verificaQuantidadeBitsZeros(byte[] b, Long quantidadeBitsZeros) {
        Long quantidadeIteracoes = 0L;
        Iterator<Boolean> iterator = new ByteArrayBitIterable(b).iterator();
        while (iterator.hasNext()) {
            Boolean bit1 = iterator.next();
            if (bit1) {
                return true;
            } else {
                quantidadeIteracoes++;
                if (quantidadeIteracoes >= quantidadeBitsZeros) {
                    return false;
                }
            }
        }

        return true;
    }

    public static byte[] gerarNovoHash() throws NoSuchAlgorithmException {
        byte[] b = new byte[300];
        new Random().nextBytes(b);

        return MessageDigest.getInstance("SHA-256").digest(b);
    }

    final protected static char[] hexArray = "0123456789ABCDEF".toCharArray();
    public static String bytesToHex(byte[] bytes) {
        char[] hexChars = new char[bytes.length * 2];
        for ( int j = 0; j < bytes.length; j++ ) {
            int v = bytes[j] & 0xFF;
            hexChars[j * 2] = hexArray[v >>> 4];
            hexChars[j * 2 + 1] = hexArray[v & 0x0F];
        }
        return new String(hexChars);
    }
}
```

O Método "verificaQuantidadeBitsZeros" utiliza o *iterator* mostrado anteriormente para percorrer o *hash* gerado e validar se possui a quantidade de zeros necessária, retornando verdadeiro se não possuir e falso se possuir.

O método “gerarNovoHash” gera uma mensagem randômica de 300 bytes e gera um *hash* com base nesta mensagem utilizando a classe *MessageDigest* nativa do java 8, onde no caso da figura 21 é utilizado o algoritmo SHA-256.

O método “bytesToHex” é uma simples conversão de um *array* de bytes para um número hexadecimal para melhor visualização.

Na figura 7 é onde são ordenados os cálculos dos *hashes*, mantendo a *thread* calculando *hashes* enquanto o número de zeros não é alcançado. A partir deste momento, aborda-se as análises após os cálculos aqui explicados.

Figura 7 – Classe de cálculo

```java
package br.com.powercripto.service.util;

import ...

/**
 * Diego 03/08/16.
 */
public class ThreadHashCalculation extends Thread {
    private final Logger log = LoggerFactory.getLogger(CriptoUtil.class);

    private String hash;
    private Long quantidadeBitZero;
    private Long quantidadeHashesCalculado = 0L;

    private volatile boolean threadRunning = true;

    public ThreadHashCalculation(Long quantidadeBitZero) { this.quantidadeBitZero = quantidadeBitZero; }

    public String getHash() { return hash; }

    public void setHash(String hash) { this.hash = hash; }

    public boolean isThreadRunning() { return threadRunning; }

    @Override
    public void run() {
        try {
            byte[] digest;
            do {
                digest = CriptoUtil.gerarNovoHash();
                quantidadeHashesCalculado++;
            } while (CriptoUtil.verificaQuantidadeBitsZeros(digest, quantidadeBitZero) && threadRunning);


            if (!CriptoUtil.verificaQuantidadeBitsZeros(digest, quantidadeBitZero)) {
                hash = CriptoUtil.bytesToHex(digest);
            }

            threadRunning = false;
        } catch (NoSuchAlgorithmException e) {
            log.error(e.getMessage(), e);
        }
    }

    public Long stopThread() {
        threadRunning = false;
        return quantidadeHashesCalculado;
    }
}
```

- ## 4. Análise dos Algoritmos

Abaixo seguem as tabelas com os resultados obtidos após a alimentação dos valores no software IBM SPSS versão 21 e as análises realizadas:

Tabela 1 – Quantidade de hashes com 20 zeros

Algoritmo	n	$\bar{x} \pm DP$	Mín	P_{25}	Md	P_{75}	Máx
SHA-256	1000	4103955.57±2086689.109	294178	2567827.00	3796470.00	5307197.00	12709116
SHA-1	1000	4048292.74±1997627.164	605002	2525146.00	3662829.50	5227138.00	13182126
MD5	1000	4066137.24±2049407.325	358941	2664215.50	3776953.00	5099764.50	16221330

Embora demonstrado na tabela 1 onde a amostra sugere que o número de *hashes* calculados nos algoritmos SHA-1 e MD5 seja menor que o SHA-256, não foi encontrado diferença estatisticamente significativa, tendo valor p = 0,850 no teste H de Kruskal-Wallis, desta forma a conclusão é de que a quantidade de *hashes* calculada para obter o resultado desejado independe do algoritmo utilizado para uma mesma parametrização de 20 zeros.

Tabela 2 – Tempo de processamento com 20 zeros

Algoritmo	n	$\bar{x} \pm DP$	Mín	P_{25}	Md	P_{75}	Máx
SHA-256	1000	4.143666±2.2099968[a]	0.2940	2.5250	3.8420	5.3320	15.8
SHA-1	1000	3.437700±1.7270803[b]	0.4910	2.1547	3.0795	4.4140	12.5
MD5	1000	2.921129±1.5387640[c]	0.2780	1.8560	2.8120	3.6662	10.9

[a,b,c] Letras distintas indicam diferenças estatisticamente significativas.

Verifica-se diferença estatisticamente significativas após aplicação do teste H de Kruskal-Wallis, tendo valor $p < 0.001$, portanto se pode concluir que o tempo de processamento varia conforme o algoritmo utilizado, assim se observa a mediana na tabela 2 para classificar, tendo a seguinte relação de tempo: SHA-256 > SHA-1 > MD5.

Isto nos diz quanto a parametrização de 20 zeros, é demonstrado agora com parametrização de 25 zeros.

Tabela 3 – Quantidade de hashes com 25 zeros

Algoritmo	n	$\bar{x} \pm DP$	Mín	P_{25}	Md	P_{75}	Máx
SHA-256	1000	134225834,0±68607076,74	11230077	83960161,50	122121062,0	174634448,5	4E+008
SHA-1	1000	130523071,5±67784864,65	12522789	77994333,00	120503495,0	168394264,0	5E+008
MD5	1000	131663708,5±69237809,18	12885044	83298417,00	117679351,0	168202992,0	5E+008

Da mesma forma como na parametrização de 20 zeros, é demonstrado na tabela 3 uma amostra que sugere um número de *hashes* calculados nos algoritmos SHA-1 e MD5 inferior ao SHA-256, entretanto não foi encontrado diferença estatisticamente significativa, tendo valor $p = 0{,}398$, desta forma a conclusão é de que a quantidade de *hashes* calculada para obter o resultado desejado independe do algoritmo utilizado para uma mesma parametrização de 25 zeros.

Tabela 4 – Tempo de processamento com 25 zeros

Algoritmo	n	$\bar{x} \pm DP$	Mín	P_{25}	Md	P_{75}	Máx
SHA-256	1000	134,142170±70,7377542 [a]	10,6750	81,995500	122,721000	174,519000	439,5870
SHA-1	1000	106,699740±56,3890629 [b]	9,0440	62,519500	97,034500	140,578500	366,5290
MD5	1000	89,249677±47,5183133 [c]	7,4830	54,945000	80,665500	113,881500	309,6070

[a,b,c] Letras distintas indicam diferenças estatisticamente significativas.

Verifica-se diferença estatisticamente significativas após aplicação do teste de Kruskal-Wallis, tendo valor $p < 0{,}001$, portanto se pode concluir que o tempo de processamento varia conforme o algoritmo utilizado, assim observamos a mediana na tabela 4 para classificar, tendo a seguinte relação de tempo: SHA-256 > SHA-1 > MD5.

Acima foram demonstrados as comparações de algoritmos diferentes com a mesma parametrização, primeiro com 20 zeros e logo após com 25 zeros, será visto a seguir comparações em que o algoritmo é o mesmo, mas a parametrização é diferente.

Tabela 5 – Comparações de mesmo algoritmo com parametrizações diferentes

Algoritmo	Tipo	Normal	Md 20 zeros	Md 25 zeros	Valor-p
SHA-256	*Hashes*	Não	3796470,00	122121062,0	< 0,001
	Tempo	Não	3,8420	122,721000	< 0,001
SHA-1	*Hashes*	Não	3662829,50	120503495,0	< 0,001
	Tempo	Não	3,0795	97,034500	< 0,001
MD5	*Hashes*	Não	3776953,00	117679351,0	< 0,001
	Tempo	Não	2,6620	80,665500	< 0,001

O valor-p foi obtido com a aplicação do teste U de Mann-Whitney para duas amostras independentes.

Tem-se valor-p inferior a 0,001 em todos os pares, desta forma se observa significância estatística em todas as comparações, concluindo que, em todos os casos a parametrização de 25 zeros possui valores maiores, ou seja, demanda maior quantidade de *hashes* calculados e também maior tempo de processamento para alcançar um resultado.

- ## 5. Discussão dos resultados obtidos

Neste capítulo será abordado a visão da literatura sobre os resultados obtidos neste trabalho e como se convergem para uma mesma visão ou divergência de resultados.

Foram obtidos, quando estatisticamente significativos, resultados demonstrando um custo maior para SHA-256 do que para SHA-1, onde o MD5 é o mais veloz para processar, tendo em vista estes resultados, discute-se.

O o MD5 é usado há décadas e já apresentou várias vulnerabilidades, desta forma, retém-se a conclusão de ser mais rápido de processar, todavia com custos em sua segurança, sendo desaconselhado pelo autor seu uso em temas tão sensíveis como transações financeiras (FORTE, 2009).

Quanto ao SHA-256 e o SHA-1, como visto no artigo de Satoh e Inoue (2007) foi obtido o melhor desempenho entre os *Secure Hashes* no algoritmo SHA-1 no circuito integrado de aplicação específica, do inglês *application-specific integrated circuit* (ASIC), também obtendo bons resultados com o SHA-256, indo de encontro com os resultados obtidos neste trabalho, também é comentado sobre o desempenho do MD5, todavia a implementação do hardware para este algoritmo necessitou de especificidades que levaram a queda de eficiência do hardware, reforçando a tese de que é preferível utilizar um *Secure Hash* ao MD5.

Vale ressaltar que se pode utilizar um ASIC especificamente para mineração de *Bitcoin*, tendo em vista que foi desenvolvido um hardware específico para cálculo de *hashes* no artigo citado.

Desta forma, a literatura estudada para este trabalho se encontra nas conclusões ao obter o SHA-256 como mais indicado para a parametrização do problema do cálculo de *nonce* do *Bitcoin*, por apresentar maior desafio ao ser processado todavia mantendo a eficiência e menor vulnerabilidade quando comparado com o MD5.

- ## 6. Conclusão

Foram abordados neste trabalho análises estatísticas comparando amostras de cálculos de *hashes* com uma parametrização de número de bits zeros ao início do *hash*, os quais representam o cálculo do *nonce* utilizado para minerar um bloco de transação no *Bitcoin* de acordo com o trabalho de Nakamoto.

Estas análises foram feitas com os algoritmos SHA-256, SHA-1 e MD5, utilizando a parametrização de bits zeros de 20 zeros e 25 zeros. Foram analisados os algoritmos entre si com parametrização de 20 zeros, a seguir novamente entre si com parametrização de 25 zeros e por fim pares do mesmo algoritmo comparado entre parametrização de 20 zeros e 25 zeros, todos estes testes foram divididos entre quantidade de *hashes* calculados necessários para chegar em um resultado e tempo de processamento para mesmo fim.

Primeiramente, nas análises entre os algoritmos com parametrização de 20 zeros foi concluído:

a. para quantidade de *hashes* não existe diferença estatisticamente significativa;
b. para tempo de processamento se tem SHA-256 > SHA-1 > MD5.

Assim, nas análises entre os algoritmos com parametrização de 25 zeros foi concluído:

a. para quantidade de *hashes* não existe diferença estatisticamente significativa;
b. para tempo de processamento se tem SHA-256 > SHA-1 > MD5.

Por fim a comparação de mesmo algoritmo entre as parametrizações de 20 e 25 zeros foi concluído:

a. para quantidade de *hashes* se tem a parametrização de 25 zeros > 20 zeros em todos os três algoritmos;
b. para tempo de processamento de mesma forma se tem a parametrização de 25 zeros > 20 zeros em todos os três algoritmos.

Tem-se portanto a conclusão de que, dos algoritmos estudados, o SHA-256 apresenta a melhor escolha para o cálculo do problema de *nonce* do *Bitcoin*, aumentando o parâmetro de zeros a medida que o poder de hardware dos nós aumenta, é tida como solução suficiente para a manutenção da estrutura do desafio na mineração da moeda até que se crie ou venha a ser necessário um novo algoritmo de cálculo de *hash* que não os abordados.

Também se pode verificar, devido ao custo de tempo ao processar com parametrização de 25 zeros, considera-se uma limitação a de não ser viável incrementar a parametrização para níveis mais altos, onde os trabalhos futuros com clusterizações ou análises estatísticas prevendo os resultados poderiam auxiliar com esta limitação.

- **References**

FORTE, D. **The Death of MD5**. Network Security, Elsevier, p. 18-20, 2009.

LITTLE, E. **Bitcoin**. Tradução nossa. The Investment Lawyer, v. 21, n. 5, 2014, 5p.

NAKAMOTO, S. **Bitcoin: A Peer-to-Peer Electronic Cash System**. 2008, 9 p.

O GLOBO. **Australiano revela que é o inventor do bitcoin:** Engenheiro exibe chaves criptográficas utilizadas na criação da moeda digital. Rio de Janeiro, 2016.

SATOH, A.; INOUE, T. **ASIC-hardware-focused comparison for hash functions MD5, RIPEMD-160, and SHS**. tradução nossa. Integration, the VLSI Journal 40, Elsevier, p. 3-10, 2007.