Pedro Henrique Nascimento Castro

Reconciliação de chaves através de um canal público usando acelerômetros

Ouro Preto
2012

Pedro Henrique Nascimento Castro

Reconciliação de chaves através de um canal público usando acelerômetros

Dissertação apresentada ao Departamento de Computação da Universidade Federal de Ouro Preto, para a obtenção de Título de Mestre em Ciência da Computação.

Orientador: Ricardo Augusto Rabelo Oliveira

Co-orientador: Jeroen Antonius Maria van de Graaf

Ouro Preto
2012

C355r Castro, Pedro Henrique Nascimento.
Reconciliação de chaves através de um canal público usando acelerômetros [manuscrito] / Pedro Henrique Nascimento Castro. - 2012.
68f.: il.: color; grafs; tabs.

Orientador: Prof. Dr. Oliveira Ricardo Augusto Rabelo.
Coorientador: Prof. Dr. Van de Graaf Jeroen Antonius Maria.

Dissertação (Mestrado) - Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação. Programa de Pós-graduação em Ciência da Computação.
Área de Concentração .

1. Criptografia de dados (Computação) - Teses. 2. Acelerômetros - Teses. 3. Criptografia de chaves públicas - Teses. I. Oliveira, Ricardo Augusto Rabelo. II. Van de Graaf , Jeroen Antonius Maria. III. Universidade Federal de Ouro Preto. IV. Titulo.

CDU: 004.713

Catalogação: www.sisbin.ufop.br

MINISTÉRIO DA EDUCAÇÃO E DO DESPORTO
Universidade Federal de Ouro Preto
Instituto de Ciências Exatas e Biológicas – ICEB
Programa de Pós-Graduação em Ciência da Computação

Ata da Defesa Pública de Dissertação de Mestrado

Aos 19 dias do mês de outubro de 2012, às 10 horas na Sala de Seminários do Departamento de Computação do Instituto de Ciências Exatas e Biológicas (ICEB), reuniram-se os membros da banca examinadora composta pelos professores: **Prof. Dr. Ricardo Augusto Rabelo Oliveira (presidente e orientador), Prof. Dr. Joubert de Castro Lima, Prof. Dr. Jeroen Antonius Maria van de Graaf e Prof. Dr. Tiago Garcia de Senna Carneiro**, aprovada pelo Colegiado do Programa de Pós-Graduação em Ciência da Computação, a fim de arguirem o mestrando **Pedro Henrique Nascimento Castro**, com o título **"Reconciliação de chaves através de um canal público usando acelerômetros"**. Aberta a sessão pelo presidente, coube ao candidato, na forma regimental, expor o tema de sua dissertação, dentro do tempo regulamentar, sendo em seguida questionado pelos membros da banca examinadora, tendo dado as explicações que foram necessárias.

Recomendações da Banca:

( ) Aprovada sem recomendações

( ) Reprovada

(X) Aprovada com recomendações: entrega em uma semana

Banca Examinadora:

Prof. Dr. Ricardo Augusto Rabelo Oliveira

Prof. Dr. Jeroen Antonius Maria van de Graaf

Prof. Dr. Joubert de Castro Lima

Prof. Dr. Tiago Garcia de Senna Carneiro

Haroldo G. Santos
Prof. Dr. Haroldo Gambini Santos
Coordenador do Programa de Pós-Graduação em Ciência da Computação
DECOM/ICEB/UFOP

Ouro Preto, 19 de outubro de 2012.

Castro, Pedro H. N.
Reconciliação de chaves através de um canal público usando acelerômetros
68 páginas
Dissertação (Mestrado) - Instituto de Ciências Exatas e Biológicas da Universidade Federal de Ouro Preto. Departamento de Computação.

1. reconciliação de chaves
2. acelerômetro
3. canal público

I. Universidade Federal de Ouro Preto. Instituto de Ciências Exatas e Biológicas. Departamento de Computação.

Comissão Julgadora:

Prof. Dr.
Ricardo Augusto Rabelo Oliveira

Prof. Dr.
Jeroen Antonius Maria van de Graaf

Prof. Dr.
Tiago Garcia de Senna Carneiro

Prof. Dr.
Joubert de Castro Lima

Aos meus pais.

Agradecimentos

Agradeço aos meus pais e ao meu irmão pelo incentivo;

Ao meu grande amigo, Xurek, por ter me alertado dos prazos;

À minha namorada, Letícia, pela paciência e compreensão;

Ao NTI que permitiu me ausentar quando necessário para conclusão deste trabalho;

Ao Prof. Marcone Jamilson Freitas Souza pelos conselhos;

Ao meus orientadores, Prof. Jeroen Antonius Maria van de Graaf e Prof. Ricardo Augusto Rabelo Oliveira, pela paciência, incentivo e confiança.

Resumo

A maioria dos sistemas criptográficos se baseiam em chaves para alcançar a segurança desejada. Essas chaves podem ser obtidas a partir de fontes de dados aleatórias. Quanto maior a aleatoriedade, maior é a sua entropia e, consequentemente, a segurança da chave. Uma possível fonte de entropia seriam os dados de um acelerômetro, que é um instrumento de medição da aceleração de um dispositivo nas três dimensões. Desta forma, é possível criar uma conexão segura entre estes dispositivos a partir da similaridade de movimento dos mesmos. A leitura destes equipamentos não é muito precisa e, além disso, pode sofrer perturbações do meio. Assim, pode-se utilizar os protocolos de reconciliação de chaves para gerar uma chave comum entre os dispositivos, sem comprometer a segurança. Este trabalho aplica esses protocolos aos dados dos acelerômetros para prover uma chave criptográfica segura.

Palavras-chave: reconciliação de chaves, acelerômetro, autenticação

Abstract

Most cryptographic systems rely on secret keys to achieve the desired security. These keys can be obtained from random data sources. The greater the randomness of the source, the greater its entropy and hence key security. One possible source of entropy would be the accelerometer data, which is an instrument for measuring the acceleration of a device in three dimensions. Thus, it is possible to create a secure connection between these devices from the similarity of their motion. Readings from these facilities are not very precise and, moreover, may be disturbed. Thus, protocols for reconciliation of keys can be used to generate a common key between devices, without compromising security. This work applies these protocols to the accelerometers data to provide a secure cryptographic key.

Keywords: key reconciliation, accelerometer, authentication

Lista de Figuras

2.1 Leitura nas 3 dimensões 5
2.2 Exemplo de medição 5
2.3 2 dispositivos alinhados 6
2.4 Reconciliação de chaves 6
2.5 Exemplo de Universal Hashing Function 9

3.1 Arquitetura para o protocolo de autenticação 14
3.2 Diffie-Hellman com Interlock* 16
3.3 Joint Fuzzy Hashing 16
3.4 Exemplo de geração de chaves preliminares 17

4.1 Diagrama de tarefas 19
4.2 Sincronização temporal nos dispositivo 1 e 2 21
4.3 Arquitetura: Diagrama de Classes 25
4.4 Permutacao 26

5.1 Gráfico: BBBSS - Motorola 38
5.2 Gráfico: BBBSS - Samsung 39
5.3 Gráfico: Cascade - Motorola 39
5.4 Gráfico: Cascade - Samsung 40

A.1 Teste 1 44
A.2 Teste 2 45
A.3 Teste 3 45
A.4 Teste 4 46
A.5 Teste 5 46
A.6 Teste 6 47
A.7 Teste 7 47
A.8 Teste 8 48
A.9 Teste 9 48
A.10 Teste 11 49
A.11 Teste 12 49
A.12 Teste 13 50
A.13 Teste 14 50

A.14 Teste 15 51
A.15 Teste 16 51

Lista de Tabelas

4.1 Conversão de decimal para binário - 3 bits 22
4.2 Conversão de decimal para binário - 4 bits 22
4.3 Conversão de decimal para binário - 5 bits 23

5.1 Especificações 32
5.2 Extração de bits - Taxa de erro (Motorola) 33
5.3 Extração de bits - Taxa de erro (Samsung) 33
5.4 BBBSS - Motorola 34
5.5 BBBSS - Samsung 35
5.6 Cascade - Motorola 35
5.7 Cascade - Samsung 36
5.8 BBBSS - Motorola: troca de mensagens 36
5.9 BBBSS - Samsung: troca de mensagens 37
5.10 Cascade - Motorola: troca de mensagens 37
5.11 Cascade - Samsung: troca de mensagens 38
5.12 Análise de desempenho 40

C.1 Taxa de erros (4 bits) - Motorola Xoom 63
C.2 Taxa de erros (4 bits) - Samsung Galaxy 64
C.3 Quantidade de bits errados após BBBSS (13%) - Motorola Xoom 65
C.4 Quantidade de bits errados após BBBSS (13%) - Samsung Galaxy . . . 66
C.5 Quantidade de bits errados após Cascade (15%) - Motorola Xoom 67
C.6 Quantidade de bits errados após Cascade (15%) - Samsung Galaxy . . . 68

List of Algorithms

1 Método *permutacao()* 26
2 Método *binary(BitArray K)* 27
3 Método *reconciliacao()* 28
4 Método *biconf()* 28
5 Método *eliminaBitsRevelados()* 29
6 Método *reconciliacao()* 29
7 Método *biconf(int p)* 30
8 Método *cascade(int p, int posicaoBitErrado)* 30
9 Método *privacyAmplification(BitArray K, int m, long seed)* 31

Sumário

1 Introdução **1**
1.1 Motivação 3
1.2 Objetivos 3

2 Referencial Teórico **4**
2.1 Acelerômetro 4
2.2 Reconciliação de chaves 5
2.2.1 Primitivas 7
2.2.2 BBBSS 7
2.2.3 BBBSS Otimizado 7
2.2.4 Shell 8
2.2.5 Cascade 8
2.3 Privacy Amplification 8
2.4 Wavelet 10
2.4.1 Probriedade de Bases e Momentos Nulos 11
2.5 Entropia 12

3 Trabalhos Relacionados **13**

4 Metodologia **18**
4.1 Extração de bits 18
4.1.1 Aquisição de dados 18
4.1.2 Alinhamento temporal 19
4.1.3 Alinhamento espacial 20
4.1.4 Decimal para binário 21
4.2 Reconciliação de Chaves 22
4.2.1 Arquitetura 24
4.2.2 Métodos Comuns 24
4.2.3 BBBSS Otimizado 26
4.2.4 Cascade 29

5 Experimentos e análise de resultados **32**
5.1 Experimentos 32
5.1.1 Extração de bits 33
5.1.2 Reconciliação de chaves 34
5.1.3 Análise de desempenho 40

6 Considerações Finais **41**

Referências Bibliográficas **42**

A Comparação visual de medições **44**

B Código-fonte **52**

C Taxa de erros **62**

Capítulo 1

Introdução

À medida que as redes de computadores evoluem e eliminam a necessidade de fios para a transmissão de informações, estas se tornam cada vez mais vulneráveis a ataques que visam quebrar a confidencialidade e autenticidade [24] [23]. Tecnologias sem fio como Wi-Fi e Bluetooth estão mais expostas com relação ao vazamento de informações, pois, qualquer dispositivo dentro do alcance dessas redes poderia interceptar essas informações. Além disso, elas também são mais suscetíveis a perturbações causadas pelo ambiente ou maliciosamente. Assim, deve-se garantir que a troca de informações seja protegida por meio de criptografia.

Para garantir a segurança na autenticação e confidencialidade de informações trocadas em uma rede são utilizados sistemas criptográficos simétricos e de chave pública [18]. Estes sistemas devem ser baseados na força de pelo menos uma chave secreta e não apenas em seus métodos de embaralhamento [9]. As chaves devem ser obtidas através de fontes de dados aleatórios, como uma senha digitada por um usuário, ou dados dos diversos tipos de sensores. Quanto mais aleatória a fonte geradora da chave, maior é a quantidade de incerteza que ela contém. Essa quantidade é medida através da entropia. Estima-se que uma quantidade de entropia segura seja formada por chaves de no mínimo 128 bits (2^{128} chaves possíveis) [19].

Muitos dos algoritmos para troca de chaves estão sujeitos a ataques do tipo *Man-in-the-middle*. Neste ataque, o adversário ouve uma comunicação entre dois interlocutores e falsifica as trocas a fim de fazer-se passar por uma das partes. Ele é muito comum em redes sem fio por estas não estarem protegidas fisicamente contra invasores. O protocolo padrão utilizado para autenticação em redes Bluetooth [18] utiliza uma chave secreta inserida por um usuário para eliminar ataques do tipo *Man-in-the-middle*. Esta chave pode conter de 4 à 16 caracteres numéricos. Na melhor das hipóteses, a chave gerada conteria 10^{16} (aproximadamente 2^{53}) chaves diferentes. Essa senha possui baixa entropia e pode comprometer a segurança do sistema. Similarmente, a eliminação do ataque MITM na tecnologia Wi-Fi [18] [21] também é baseada em uma senha fornecida pelo usuário. Uma solução trivial em ambos os casos seria aumentar o número de caracteres da senha de forma a alcançar o mínimo de segurança. No entanto isso forçaria os usuários a digitar e lembrar enormes cadeias de caracteres, o que tornaria

inviável ou trabalhosa a utilização de uma conexão segura.

Acelerômetros são boas fontes de entropia, uma vez que, com um simples movimento, pode-se obter uma grande quantidade de informação. Dispositivos com acelerômetros movimentados por pessoas diferentes geram informações suficientemente diferentes para se chegar a uma chave com segurança de modo que um não consiga reproduzir o movimento executado pelo outro [14]. Além disso, se executado por um tempo mínimo, a quantidade de informação obtida é bem maior do que a fornecida por uma senha de 16 caracteres. Assim, é possível produzir uma chave comum a dois dispositivos a partir dos dados obtidos movendo-se ambos simultaneamente.

Sabe-se que a sequência obtida pelos sensores de movimento dos dispositivos não é muito precisa [11] e pode sofrer muitas perturbações, intencionais ou não. Desta forma, é necessário que as medições realizadas nos aparelhos sejam transformadas de maneira que se tornem idênticas. Isso pode ser feito através da troca de informações entre eles. No entanto, neste ponto, o canal de comunicação entre eles ainda não é seguro, pois é suscetível a interceptações, fornecendo a um adversário informações suficientes para obtenção da chave e comprometimento da informação.

Existem protocolos criptográficos que visam essa troca entre as partes, fornecendo apenas uma pequena informação (ou nenhuma) da chave final. Esses protocolos são chamados de reconciliação de chaves. Os mais comuns são o BBBSS [1], o Shell [4] e o Cascade [4]. Eles são capazes de corrigir ou ignorar algumas diferenças entre as chaves iniciais, desde que essas sejam correlatas. As chaves finais são idênticas, porém menores do que as iniciais.

Infelizmente, dispositivos móveis possuem severas limitações de recursos, como baixo processamento e memória, e também escassez de energia. Neste sentido, deve-se buscar protocolos que consumam uma menor quantidade recursos, sem detrimento da segurança. Neste trabalho foram utilizados o BBBSS Otimizado [22] e o Cascade, protocolos de reconciliação de chaves bastante conhecidos por serem bastante eficiente e possuir uma baixa taxa de troca de mensagens pela rede [4]. No entanto, não foram utilizados para dados de acelerômetros.

Neste trabalho, os protocolos de reconciliação de chaves foram aplicados aos dados obtidos dos dispositivos, sendo capaz de gerar chaves de 128 *bits* em menos de 5s, com taxa de acerto de 96%. Para garantir a segurança do método, foram testados experimentos com chaves diferentes e a taxa de erro foi de 0%, com determinados parâmetros de segurança.

Este trabalho está divido da seguinte maneira: o Capítulo 2 tem uma breve descrição dos métodos utilizados neste trabalho. O Capítulo 3 contém os trabalhos anteriores que lidam com acelerômetros para geração de chaves. O Capítulo 4 exibe a divisão das tarefas para alcançar a construção e reconciliação das chaves. No 5 encontram-se informações sobre a implementação do trabalho, bem como os testes realizados e a comparação com os trabalhos anteriores para análise de desempenho. Por último, o Capítulo 6 com a conclusão do trabalho.

1.1 Motivação

Existem muitos trabalhos que utilizam o acelerômetro como fonte de aleatoriedade e a transformam em uma chave criptográfica. Porém, alguns deles utilizam protocolos de criptografia sem uma prova forte e formal de segurança. Outros utilizam protocolos que exigem um alto poder computacional por não serem especializados em geração de chaves a partir de dados correlatos. Este trabalho utiliza o protocolo Cascade que, além de ter uma boa segurança, possui pouco processamento e baixa troca de informações pela rede.

1.2 Objetivos

O propósito deste trabalho é implementar os protocolos de reconciliação de chaves BBBSS Otimizado e Cascade aos dados obtidos a partir de dois acelerômetros e comparar com as soluções existentes em relação a quantidade de bits da chave gerada e no tempo gasto para isso. Com uma chave de confirmação maior, evita-se que o atacante teste todas as combinações possíveis de senhas, bem como facilita para o usuário a conexão e autenticação entre os dispositivos. O usuário não precisa digitar senhas ou palavras-chave. No entanto, é para alcançar tal objetivo é necessário que as chaves sejam similares.

Adicionalmente, é possível reduzir o consumo desnecessário de recursos causado por protocolos de segurança que utilizam excessivamente a troca de informações pela rede.

Capítulo 2

Referencial Teórico

2.1 Acelerômetro

Um acelerômetro é um componente eletrônico capaz de medir a aceleração aplicada a um dispositivo em 1, 2 ou 3 dimensões (figura 2.1). Existem várias formas de se obter a aceleração aplicada. Dentre elas estão a voltagem gerada em cristais sob estresse da aceleração e variação de capacitância. Porém, de forma geral essa aceleração é dada pelo somatório de todas as forças aplicadas a um dispositivo dividido pela sua massa, visualizada pela equação 2.1 [6]. A unidade é m/s^2.

$$Ad = -\sum F_s/massa \tag{2.1}$$

A especificação desses componentes utiliza a aceleração da gravidade ($g \approx 9.81m/s^2$) como medida padrão. As capacidades máximas de medição mais comuns são 2, 4, 6, 8 e 10 G. Outras variações incluem a frequência de medição e a precisão. Uma das famílias de componentes mais utilizadas é a ADLXxxx, na qual se enquadra os dispositivos testados neste trabalho.

Existem várias formas de realizar a leitura destes componentes. Neste trabalho, utilizou-se uma API de Java para Android chamada SensorManager. Esta API fornece um método que retorna um vetor contendo as medições (em m/s^2) de cada eixo no momento de leitura. Como a medição está sempre influenciada pela força da gravidade, pode se usar a relação 2.1, onde G corresponde a aceleração.

$$Ad = -G - \sum F_s/massa \tag{2.2}$$

Portanto, um dispositivo calibrado estacionado, com o eixo Z alinhado com a gravidade irá produzir a tupla de medição (0,0,9.81). Se o dispositivo estiver em queda livre, a marcação mudará para (0,0,0).

Desta forma, se este dispositivo é "agitado" por um ser humano, espera-se que os dados gerados sejam suficientemente randômicos para uso como chaves.

A figura 2.2 mostra um exemplo de medição onde o eixo y está alinhado com a gravidade enquanto o dispositivo é deslocado através do eixo x para a direita e para a

esquerda, alternadamente.

2.2 Reconciliação de chaves

Informações obtidas através de sensores como acelerômetro, temperatura, câmeras entre outros não são muito precisas e estão sujeitas a interferências, intencionais ou não. Mesmo que dois aparelhos (chamados de B = Bob e A = Alice) fiquem emparelhados corretamente (figura 2.3), e o movimento dos dois simultaneamente seja perfeito, a medição em cada um deles será diferente.

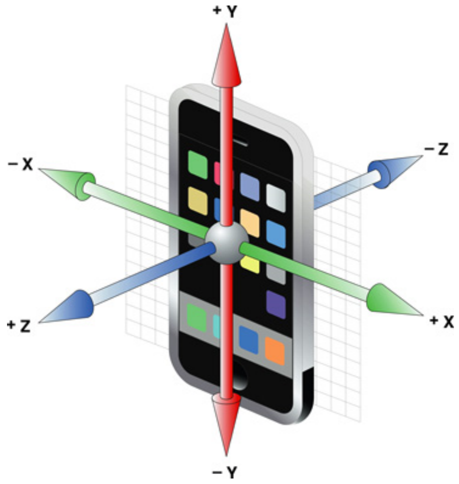

Figura 2.1: Leitura nas 3 dimensões

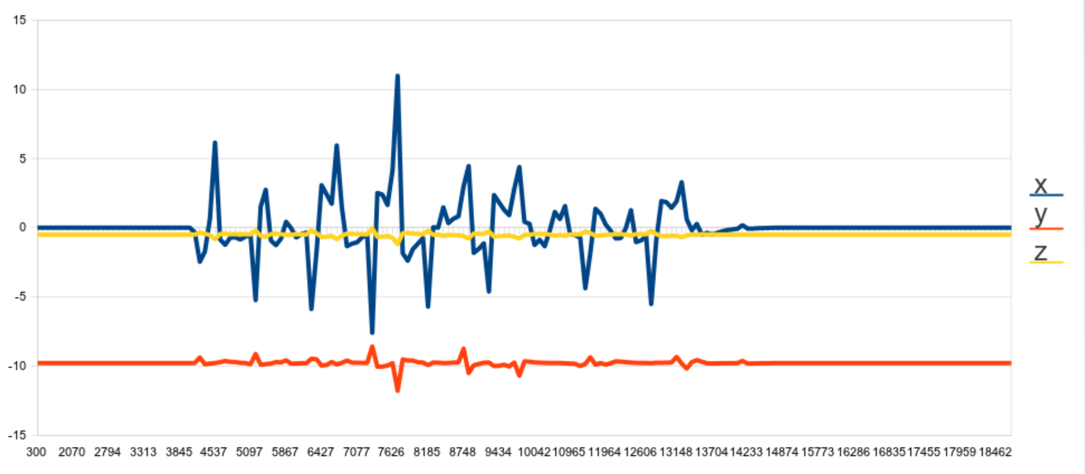

Figura 2.2: Exemplo de medição

Uma vez que se possui 2 dispositivos sendo agitados pela mesma pessoa, é possível utilizar os dados de ambos para se gerar uma chave comum entre eles. A leitura desses componentes pode sofrer alterações devido a própria natureza do mesmo e ao ambiente. Portanto é preciso remover essas diferenças para que as chaves tornem-se iguais.

Assim, tem-se duas chaves diferentes: KA e KB. Para que ambos possam trocar informações encriptadas, é necessário encontrar uma chave comum entre eles. O processo de reconciliação de chaves consiste em transformar as duas chaves criptográficas correlatas, mas diferentes, KA e KB em uma única chave KC que será menor que as chaves iniciais devido às partes descartadas durante o processo de reconciliação. A figura 2.4 exemplifica:

Essa reconciliação é baseada na troca de informações. Considerando que o canal para essa troca seja público (todos têm acesso ao *wireless*), é preciso que o protocolo utilize o mínimo de informações a respeito da chave.

Existem alguns métodos dedicados a essa tarefa como o BBBSS, BBBSS Otimizado, Shell e Cascade.

Antes de entrar nos detalhes dos protocolos, é importante detalhar algumas primitivas comuns a vários deles.

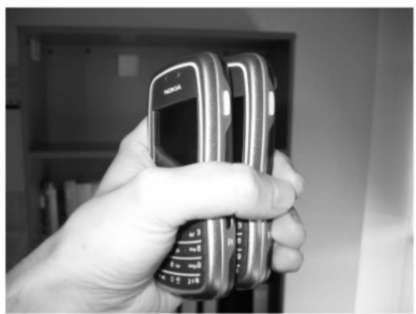

Figura 2.3: 2 dispositivos alinhados
FONTE - [16].

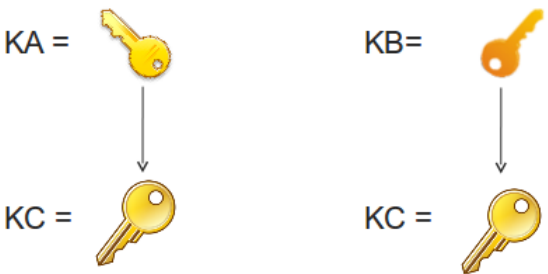

Figura 2.4: Reconciliação de chaves

2.2.1 Primitivas

Uma das primitivas adotadas pelos protocolos é a *BINARY* [4]. Se Alice e Bob possuem Strings A e B com número ímpar de erros, Alice envia a Bob a paridade da primeira metade de A. Bob compara com a paridade da mesma metade de B para identificar se o erro ocorreu na primeira ou segunda metade e avisa a Alice. O processo é repetido tomando-se a metade com erro como String até que o erro seja encontrado.

CONFIRM [4] é outra primitiva que indica, com probabilidade 1/2, quando A e B são diferentes. Caso elas sejam iguais, a primitiva o informa com probabilidade 1. Para realizá-lo, Alice e Bob escolhem um subconjunto de bits. Então comparam as suas paridades. Este processo pode ser repetido k vezes para assegurar com probabilidade de erro de 2^{-k} que A e B são iguais. O tamanho do subconjunto deve ser escolhido adequadamente de forma que a probabilidade de haver um número par ou ímpar de erros no intervalo seja alta.

BICONF [4] é uma combinação das primitivas anteriores. Toda vez que for verificado com *CONFIRM* que as Strings são diferentes, então executa-se *BINARY* para encontrar e corrigir o erro. Se vier acompanhado de um valor (ex.: *BICONF(s)*), ele deve ser executado esse número de vezes (s vezes).

2.2.2 BBBSS

Inicialmente, o BBBSS aplica a Tansformação de Uniformização [3] a toda String A e B. Ela permite distribuir os erros igualmente por toda a chave. Para isso, Alice escolhe uma função de permutação $\pi : 0{,}1^N \rightarrow 0{,}1^N$ e envia a descrição a Bob. Ambos aplicam a função de permutação π nas suas chaves KA e KB gerando chaves KA' e KB'. Deve-se observar que o número de erros (bits diferentes) continua o mesmo. Essa primitiva ajuda a evitar problemas de erros em rajada.

Após esse processo, A e B são divididos em blocos de tamanho k. Para cada bloco, *BICONF* é executado para busca e correção de erros. Como esta primitiva está sujeita a falhas quando o número de erros é par, o procedimento é repetido várias vezes aumentando-se o tamanho do bloco.

2.2.3 BBBSS Otimizado

O protocolo BBBSS descrito inicialmente em [3] não define o tamanho do bloco nem o seu aumento. Então, [22] desenvolveu uma regra para determinar o tamanho dos blocos em cada iteração [12].

Para cada iteração i, a probabilidade de erro p_i nessa iteração é dada pela equação 2.3, onde n é o tamanho da chave original, p é a probabilidade de erro original, n_i é o tamanho da chave no passo i (menor devido aos bits descartados) e z é o total de bits corrigidos até o momento.

$$p_i = \frac{np - z}{n_i} \tag{2.3}$$

Esta é a versão do BBBSS a ser usada neste trabalho.

2.2.4 Shell

O protocolo Shell [4] divide inicialmente (passo $s = 1$) as chaves KA e KB em blocos de tamanho k. Aplica-se então BICONF. Se for detectado um erro, todos os bits do bloco errado de Bob são sobrescritos com os correspondentes de Alice. No passo seguinte ($s = s + 1$), concatena-se pares de blocos adjacentes e *BICONF(s)* é aplicado. Novamente, caso um erro seja detectado, os bits do bloco problemático de Bob são substituídos pelos de Alice. Repete-se o processo até que o tamanho do bloco seja igual ao tamanho inicial das chaves.

2.2.5 Cascade

Neste protocolo [20], Alice e Bob devem decidir antecipadamente o número de passos p. Em cada passo i, ambos permutam randomicamente as chaves KA e KB e dividem em blocos de tamanho $k(i)$. Após calcular as paridades de cada bloco, Alice as envia à Bob para que o mesmo faça uma busca do tipo *BINARY* para encontrar e corrigir possíveis erros. Nesse ponto, o número de erros em cada bloco se torna par.

Então, para cada passo $i \geq 2$, aumenta-se o tamanho do bloco para $k(i) = 2k(i - 1)$ [4]. Deve-se armazenar todos os blocos de bits em cada passo, com suas devidas permutações. Agora seja l o bit corrigido no passo i. E seja A o conjunto de todos os blocos de passos $[1..i - 1]$ que contém o bit l. Deve-se corrigir todos os blocos A. Sabe-se então que agora todos esses blocos contém um número ímpar de erros. Então Alice e Bob escolhem o menor desses blocos e usam *BINARY* novamente para encontrar o outro erro no bit l'.

Seja B o conjunto de todos os blocos de passos $[1..i]$ contendo l'. Corrige-se B com *BINARY*. Todos os blocos em $(B \cup A) \backslash (B \cap A)$ tem um número ímpar de erros. Então atualiza-se A com $A \leftarrow (B \cup A) \backslash (B \cap A)$ e repete-se o processo até que A fique vazio. Neste ponto, o passo i terminou. O protocolo encerra de acordo com o número de passos determinado inicialmente.

Este protocolo é parecido com os anteriores com a diferença de que a cada erro corrigido (tornando o número de erros par), busca-se o mesmo bloco no passo anterior, que agora possui um número ímpar de erros e processa apenas esse bloco. Realiza-se esse método recursivamente até o primeiro passo. Neste trabalho será usado o cascade como protocolo de reconciliação.

2.3 Privacy Amplification

Durante o processo de reconciliação de chaves, muitos bits de paridade são revelados pelo canal de comunicação. Isso poderia dar alguma informação a um adversário e comprometer a segurança da chave.

Uma das estratégias usadas para se evitar esse problema é descartar o último bit de cada bloco cuja paridade foi transmitida pela rede. Assim, elimina-se qualquer informação que o adversário possui, uma vez que, não conhecendo o bit descartado, de

nada vale o bit de paridade que ele possui. Este método é muito utilizado no BBBSS e no Shell.

No entanto, o Cascade possui muitas operações de trocas de bits de paridade pois a cada troca, ele itera sobre os passos anteriores executando o mesmo processo. Desta forma, existe uma forma mais elaborada, e bastante segura, de se eliminar qualquer vantagem concedida a um terceiro elemento. Ela é chamada de Privacy Amplification*Privacy Amplification* [2] [3].

Esta técnica consiste em utilizar uma função de transformação da chave original K de tamanho m para uma chave K' de tamanho menor $n = m - r$, onde r é o número de bits revelados. Esta função é descrita pela equação 2.4.

$$F = \{0,1\}^M \rightarrow \{0,1\}^N \tag{2.4}$$

Existem muitas formas de se construir esta função. A mais comum é utilizar uma *Universal Hashing Function* [5].

Universal Hashing é uma função *hash* [17] [7] tomada aleatoriamente de uma família de funções com certas propriedades matemáticas. Isto garante uma expectativa de um baixo número de colisões, mesmo se os dados são escolhidos por um adversário. Muitas famílias universais são conhecidas (para *hashing* de inteiros, vetores, *strings*), e seu desempenho é frequentemente muito eficiente.

Uma função de hashing universal deve ser capaz de ser construida a partir da definição em tempo computacionalmente viável. Por exemplo, a definição exige que a função transforme uma string de tamanho m para n. Então deve ser possível construir a função que o faça de forma simples.

Uma das formas existentes é através da multiplicação de matrizes. Suponha que a chave original K tenha tamanho m. Seja n o tamanho da chave a ser gerada K' para eliminar os r bits revelados durante o processo de reconciliação de chaves. Para Alice e Bob, é gerada uma matriz aleatória de n linhas e m colunas (igual para ambos). Cada linha da matriz é multiplicada por K, produzindo um elemento da chave K'. A figura 2.5 exemplifica o processo.

$$\underset{\text{Matriz}}{\begin{vmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{vmatrix}} \text{ x } \underset{K}{\begin{vmatrix} 1 \\ 0 \\ 1 \\ 1 \end{vmatrix}} = \underset{K'}{\begin{vmatrix} 0 \\ 1 \\ 0 \end{vmatrix}}$$

Figura 2.5: Exemplo de Universal Hashing Function

Ao final do próximo, Alice e Bob possuem strings iguais de tamanho n com nenhuma informação por parte de um possível adversário.

2.4 Wavelet

Nesta seção apresentamos as *wavelets*. Bases *wavelet* revelam a regularidade do sinal através de coeficientes usando um algoritmo eficiente computacionalmente. Os bancos de filtros requerem somente $O(N)$ operações para sinais de tamanho N.

Através desta seção, mostraremos os parâmetros corretos para a regularidade do sinal.

Considerando que $f(x) \in L(\mathbb{R}^2)$ seja o sinal observado, aqui nós consideramos a função $\hat{f}(x)$ como um vetor no espaço de Hilbert. Supondo-se que existe uma família de bases (Φ, ψ) onde

$$\lim_{N->\infty} ||f(x) - \sum_{n=0}^{N} \hat{f} < \phi,\psi > || = 0$$

Seja V'_j uma seqüência de subespaços fechados no espaço de Hilbert, de $L(\mathbb{R}^2)$. Cada V'_j representa a aproximação sucessiva do sinal original, considerando-se uma resolução de 2^j. Uma resolução suave ocorre quando o menor valor de j é usado.

Os detalhes da projeção entre 2^j e 2^{j-1}, denotada por W_j, é definida por $W_j \oplus V'_j = V'_{j-1}$, onde $\oplus$ denota a soma direta de dois espaços vetoriais. Desta forma, V'_j pode ser decomposto através da soma direta de subespaços, $V'_j = W_{j+1} \oplus W_{j+2} \oplus \cdots \oplus W_J \oplus V'_J$, onde $j < J$ e todos os subespaços são ortogonais.

Isto significa que a aproximação de $f(x)$ na resolução $2 - j$ é definida como uma projeção ortogonal em V_j. Esta projeção pode ser descrita como um processo de amostragem:

$$\hat{f}_{2-J} = f(x) * \phi_{2j}$$

Então, $\hat{f}$ é a melhor aproximação linear na projeção V_j. Para esta aproximação, considere dois conjuntos de funções: a escala de funções $(\phi(t))$; e as funções *wavelet* $(\psi(t))$. Ambos estão relacionados com filtros passa-baixa e passa-alta denominados, respectivamente, *wavelet* e vetores de escala.

Desta forma, $f(x)$ pode ser aproximado pela seguinte expansão

$$f(x) = \sum_{n} s_{i_0}[n]\phi_{i_0,n}(x) + \sum_{i=i_0}^{i_1} w_i[n]\,\psi_{i,n}(x)$$

onde $s_{i_0}[n]$ são os coeficientes de escala,

$$s_{i_0}[n] = \int f(x)\,\phi_{i_0,n}(x)\,dx$$

e $w_i[n]$ são os coeficientes *wavelet*,

$$w_i[n] = \int f(x)\,\psi_{i,n}(x)\,dx$$

2.4.1 Probriedade de Bases e Momentos Nulos

A maioria das aplicações de bases *wavelet* usa sua capacidade de aproximar funções com poucos coeficientes. O conjunto de bases, ψ, é concebido para produzir uma quantidade máxima de coeficientes próximos de zero. Isto depende da regularidade de $f(x)$ e a quantidade de momentos nulos das bases. Considere que L é o número de momentos nulos da *wavelet*.

A transformada *wavelet* de uma função $f(x)$ é uma função bidimensional $\gamma(s,\tau)$. As variáveis s e τ são as novas dimensões, escala e translação, respectivamente.

A transformada *wavelet* é composta por funções $\phi(t)$ and $\psi(t)$, as quais a condição de aceitação $\int \psi(t)dt = 0$ detém. Esta condição indica que a *wavelet* tem um suporte compacto, uma vez que a maior parte do seu valor está limitado a um intervalo finito, o que significa que ele tem um valor zero exato fora deste intervalo. Além disso, $\psi_{i,k}(t) = 2^{\frac{i}{2}}\psi(2^i - k)$, para as versões não compactas e transladadas de $\psi(t)$. Ambos os casos caracterizam a propriedade de localização espacial *wavelet*.

Outra propriedade é a suavização *wavelet*. Se $f(x)$ é localmente suavizada, então, em um pequeno intervalo, é bem aproximada por um polinômio de Taylor de grau k. Se $k < L$, então *wavelets* são ortogonais a este polinômio de Taylor.

Considere a expansão de $\gamma(s,\tau)$, cerca de $\tau = 0$, em um polinômio de Taylor de ordem n,

$$\gamma(s,0) = \frac{1}{\sqrt{s}}\left[\sum_{p=0}^{n} f^{(p)}(0)\int \frac{t^p}{p!}\,\psi(t/s)dt + O(n+1)\right],$$

onde $\mathrm{f}^{(p)}$ é a p-ésima derivada de f e $O(n+1)$ representa o resto da expansão. Uma *wavelet*, $\psi(t)$, tem L momentos nulos se

$$\int_{-\infty}^{\infty} t^k\psi(t)dt = 0,$$

para $0 \leq k < L$. Se

$$M_k = \int t^k\psi(t)dt,$$

nós temos

$$\gamma(s,0) = \frac{1}{\sqrt{s}}\left[\frac{f^{(0)}(0)}{0!}M_0 s^1 + \cdots + \frac{f^{(n)}(0)}{n!}M_n s^{n+1} + O(s^{n+2})\right]$$

Os momentos nulos são

$$M_n(0,l) = 0, \text{ para } l = 0,1,...,L-1$$

A partir da condição de aceitação, temos que o momento $0 - simo$, M_0, é igual a zero. Se os outros momentos M_n são zero, então $\gamma(s,\tau)$ converge para uma função suave $f(t)$.

Então, se $f(t)$ é descrito por uma função polinomial de grau até $(L-1)$, o termo $O(s^{n+2})$ será zero e valores pequenos aparecerão como uma combinação linear de ψ na função $\gamma(s,\tau)$.

O grau de regularidade e a taxa decrescente de transformações *wavelet* estão relacionados com o número de momentos nulos. Esta propriedade é importante para inferir a propriedade de aproximações no espaço multiresolução. Quando uma *wavelet* tem vários momentos nulos, haverá coeficientes com valores baixos. Em regiões onde $f(x)$ é uma função suave, os coeficientes wavelet com escalas finas em $f(x)$ são nulos. Isso torna as séries *wavelet* uma representação esparsa de $f(t)$.

Ao escolher uma *wavelet* particular, existe uma troca entre o número de momentos nulos e o tamanho do suporte. Se $f(x)$ tem poucas singularidades e muita suavidade entre as singularidades, é melhor escolher um *wavelet* com muitos momentos nulos para produzir um grande número de coeficientes nulos. Se as singularidades aumentam, é melhor reduzir o número de momentos nulos e o tamanho do suporte.

2.5 Entropia

Seja X uma variável aleatória tomada de um conjunto finito de valores $x_1,x_2,...,x_n$ com probabilidade $P(X = x_i) = p_i$, onde $0 \leq p_i \leq 1$ para cada i, $1 \leq i \leq n$, e $\sum_{i=1}^{n} p_i = 1$.

Entropia é a quantidade de incerteza existente em uma informação provida pela observação de X [17]. É medida em *bits* [13] e pode ser usada para aproximar a média de *bits* necessária para codificar os elementos de X. Pode ser calculada através da equação 2.5.

$$H(x) = -\sum_{i=1}^{n} p_i \ln p_i \tag{2.5}$$

Capítulo 3

Trabalhos Relacionados

Existem muitos trabalhos envolvidos com a coleta de dados de sensores para aplicação na criptografia. Um estudo inicial mostrou a relação entre os dados obtidos em um acelerômetro e o tipo de movimento das pessoas para determinar se dois aparelhos são carregados pela mesma pessoa [11]. Dois dispositivos são acoplados juntos ao corpo de uma pessoa e enquanto ela realiza uma atividade qualquer (andar, subir e descer no elevador e etc) são monitorados os dados dos acelerômetros dos dispositivos.

Para os testes foram utilizados 3 modelos de acelerômetros MEMS: ADXL202E, LIS3L02 e CXL02LF3. Eles medem de 2 a $6G$.

Em cada medição, é tomada a magnitude do vetor resultante das dimensões como na equação 3.1.

$$A_{mag} = \sqrt{(A_x)^2 + (A_y)^2 + (A_z)^2} \tag{3.1}$$

Neste momento, uma FFT é aplicada ao conjunto de magnitudes para passá-las ao domínio da frequência. Esta transformação tem o intuito de reduzir o efeito de problemas como: latências na comunicação, limites de medição de cada dispositivo e alto custo da análise de modelos mais complexos.

Depois de obtidas as frequências, utiliza-se um método baseado em uma medida de correlação linear para determinar se os dispositivos são carregados pela mesma pessoa ou não. Essa coerência é dada pela equação 3.2 onde S_{xy} é a espectral cruzada de dois sinais e S_{xx} e S_{yy} é a densidade espectral de cada sinal.

$$C_{xy}(f) = |\gamma_{xy}(f)|^2 = \frac{|S_{xy}(f)|^2}{S_{xx}(f)S_{yy}(f)} \tag{3.2}$$

Correlações próximas de 1 indicam que os sinais são bastante correlatos, enquanto as próximas de 0 afirmam o contrário. A partir de um determinado parâmetro, o protocolo decide se são similares ou não. Com uma medição de pelo menos $8s$ é possível determinar com 100% de precisão que os dispositivos estão sendo movimentados pela mesma pessoa.

No entanto, este trabalho não foi aplicado a nenhuma técnica de segurança. Não pode-se garantir que os passos serão utilizados para autenticar dois dispositivos sem

falhas. Isso porque não há definições de como as informações a respeito da coerência são transmitidas. Se não há um meio de encriptá-las antes do processo de transmissão, ela pode revelar dados significativos para um terceiro elemento reproduzir o protocolo e se autenticar sem dificuldades.

Sendo assim, posteriormente esse protocolo foi aplicado à criptografia [14], autenticando os dispositivos e gerando uma chave comum entre as partes a partir das informações dos acelerômetros. Como instrumento de medição, foram utilizados acelerômetros ADXL202JE com capacidade de leitura de até $2G$ em cada um dos eixos. Estes componentes foram conectados a um computador (onde ocorre todo o processamento) pela porta paralela. Mais tarde, os mesmos métodos foram testados em aparelhos com precisão de leitura inferior [15], como o celular Nokia 5500. Além disso, todo o processamento foi realizado nos aparelhos, conectados via Bluetooth. No entanto, o método é igual para ambos os trabalhos. Ele é dividido em 5 tarefas de acordo com a figura 3.1.

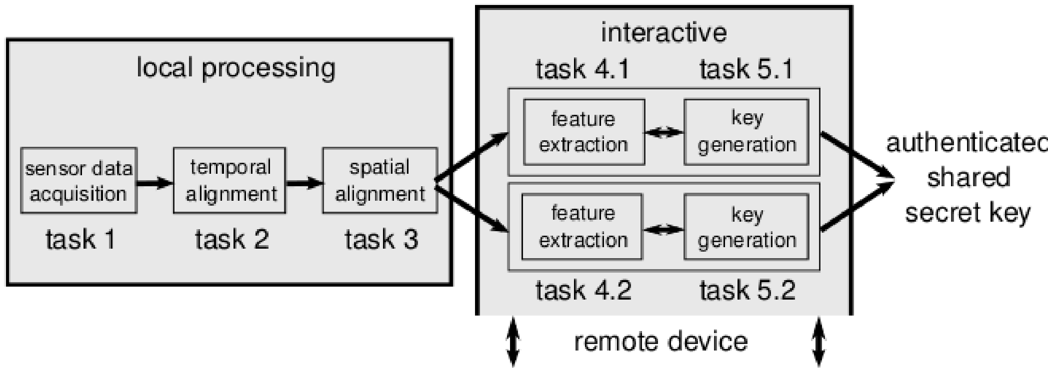

Figura 3.1: Arquitetura para o protocolo de autenticação

A primeira tarefa consiste na aquisição de dados do sensor, definindo a taxa de amostragem. Estes dados devem ser coletados localmente e, por questões de segurança, não devem trafegar pela rede sem fio. Uma taxa de amostragem de 100 a $600Hz$ foi identificada como apropriada.

A segunda é responsável pelo alinhamento temporal. É necessário que haja um consenso sobre o início da medição e isso pode ser alcançado por meio de disparo manual (explícito) ou detecção de uma movimentação brusca (implícito). Existem diversos modelos na literatura para identificar esses movimentos [11] [8]. Contudo, este trabalho utiliza a forma implícita, onde os dispositivos começam a registrar os dados do acelerômetro assim que detecta um movimento brusco. Isso elimina a necessidade de sincronização entre os aparelhos. Além disso, é considerado que a taxa de amostragem é precisa o suficiente para não requerer a sincronização entre os intervalos de medição.

A próxima tarefa realiza o alinhamento espacial. Isso porque a medição sofre interferência da orientação dos aparelhos. É muito difícil realizar a coleta sem provocar mudanças devido ao ângulo de inclinação. Além disso, a própria diferença de precisão

dos dispositivos gera ruídos e distorções na medição. Então é utilizado o vetor resultante das 3 dimensões (magnitude do vetor) (equação 3.1) como no trabalho de [11]. Desta forma, elimina-se a diferença devido a orientação.

Para as tarefas 4 e 5 (extração de características e geração de chave secreta) foram propostos dois métodos, *Diffie-Hellman com Interlock* e *Candidate Key Protocol*, que são descritos abaixo.

O primeiro método, resumido na figura 3.2, utiliza *Diffie-Hellman* para geração de chaves de sessão e autenticação. Para eliminar o ataque MITM próprio deste protocolo [19], é utilizado o *Interlock*. Este faz com que A envie a primeira metade de uma mensagem a encriptada a B usando a chave de autenticação. B também envia a primeira metade da mensagem b encriptada a A. Enquanto A e B não receberem a primeira metade do seu correspondente, eles não enviam a segunda. Então, se houver um terceiro elemento E recebendo e modificando as mensagens, este não conseguirá decriptar e encriptá-las novamente para repassar a mensagem ao seu destino, pois não é possível fazê-lo com apenas metade das chaves. Logo, deixará as mensagens seguirem seu curso normalmente. Quando A e B possuirem as duas metades, eles podem decriptá-las e verificar se é uma mensagem válida.

Para determinar a validade A e B utilizam suas leituras dos acelerômetros como mensagens a e b. Desta forma, ao reconstruir as mensagens originais, ambos realizam uma comparação para identificar a similaridade das mesmas. A comparação é realizada do mesmo modo que o trabalho de [11], com correlação linear (coerência). Se forem similares, eles utilizam a chave gerada inicialmente.

O *Interlock* originalmente exige um tamanho fixo da mensagem. Como os tamanhos de n e m das mensagens encriptadas são diferentes, houve uma pequena modificação de modo a dividí-las em blocos de $128bits$, tomar a primeira metade de cada um destes blocos e concatená-los para formar as mensagens $A1$ e $B1$ e a segunda metade para $A2$ e $B2$. Com esta alteração o método foi chamado de *Interlock**.

O segundo método divide os dados obtidos do acelerômetro em várias partes e gera um hash de cada uma delas. Cada hash é enviado do dispositivo A para B, que irá escolher quais partes da chave irá usar para concatenar e construir a chave (*matching key parts*). Quando alcançar uma quantidade segura de entropia, B concatena a chave gerada a um sal C e envia o hash (*candidate key*) para A. Se A conseguir gerar uma chave com mesmo hash, então ambos compartilham uma chave secreta. Caso não seja possível, o processo é refeito e B irá escolher outras partes da chave. Este método é menos custoso computacionalmente, porém não há muitas provas formais de sua segurança em relação ao primeiro.

Estes dois métodos passaram a ser chamados de ShaVe e ShaCK respectivamente [16]. No restante deste trabalho eles foram referenciados por estes nomes.

Mais tarde, foi desenvolvido o protocolo MartiniSync [10]. O primeiro passo deste protocolo é a notificação, onde os dispositivos marcam o início da medição. Pode ser feito botão manual, teste de proximidade ou movimentação brusca. Em seguida, eles são movimentados aleatoriamente (*Martini shake*) para produzir entropia. Quando um dos dispositivos estima que a entropia alcançada é suficiente, ele envia um sinal para o

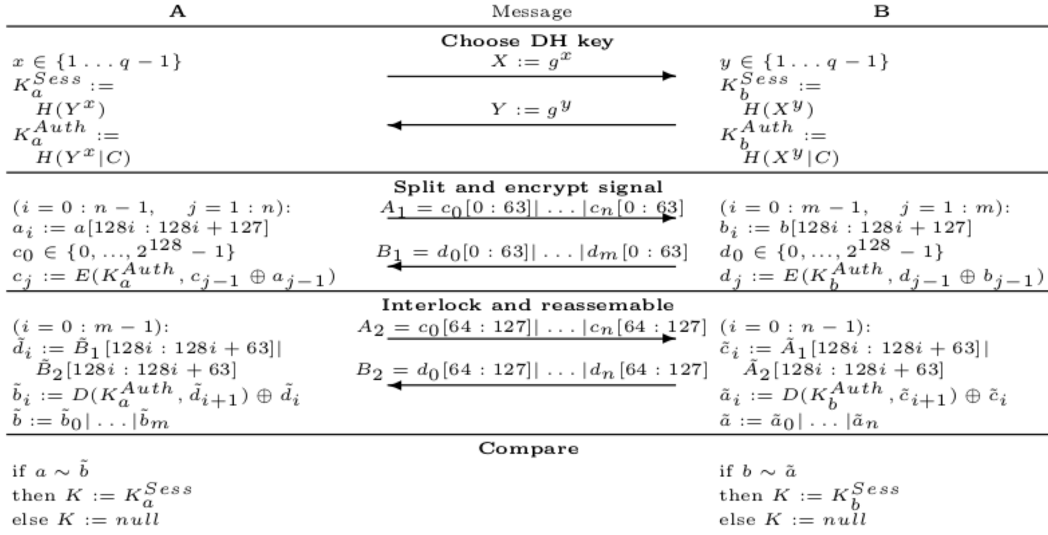

Figura 3.2: Diffie-Hellman com Interlock*

outro para que as medições terminem.

Simultaneamente ao *Martini shake*, é realizado o *Joint Fuzzy Hashing*, que é responsável pela geração das chaves. Ele será descrito abaixo e pode ser resumido na figura 3.3.

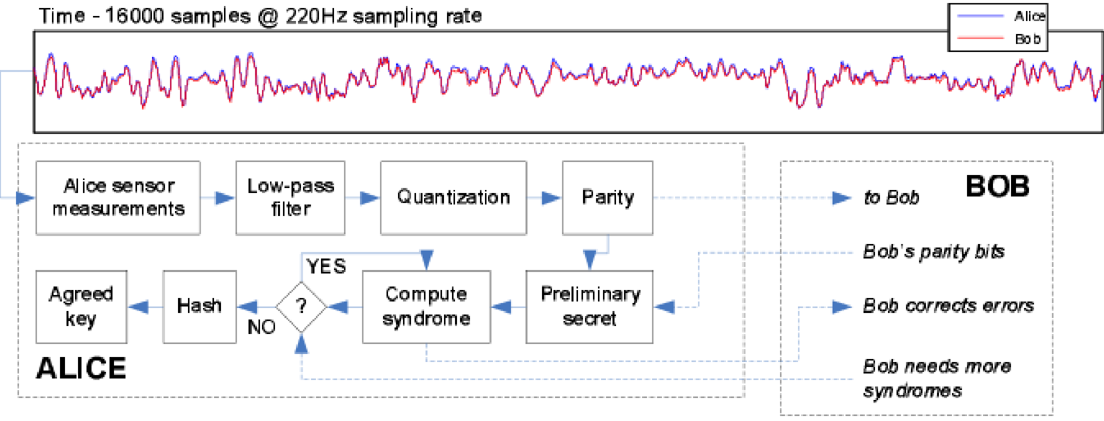

Figura 3.3: Joint Fuzzy Hashing

Inicialmente os dados passam por um filtro passa-baixo para reduzir o ruído. E Alice e Bob quantizam os seus sinais dividindo por um passo de quantização Q e ar-

rendondando para o inteiro mais próximo. Então eles enviam um ao outro as paridades destes inteiros. Quanto maior o passo de quantização, menor será a diferença entre as paridades dos dispositivos.

Agora eles irão gerar a chave preliminar. Para entender este processo deve-se observar a figura 3.4. As duas primeiras linhas representam as informações quantizadas de Alice e Bob. A terceira e quarta, mostram os bits de paridade de cada quantização enviado pela rede. Então Alice e Bob comparam os bits de paridade recebidos com os seus. Toda vez que eles concordarem em uma paridade e essa paridade for diferente da última utilizada, ambos armazenam nas linhas 5 e 6 a diferença entre o valor quantizado atual e o último, onde D significa que o valor diminuiu em relação a última marcação e U indica que aumentou.

Por fim, os valores armazenados (em cinza) são utilizados como chave preliminar.

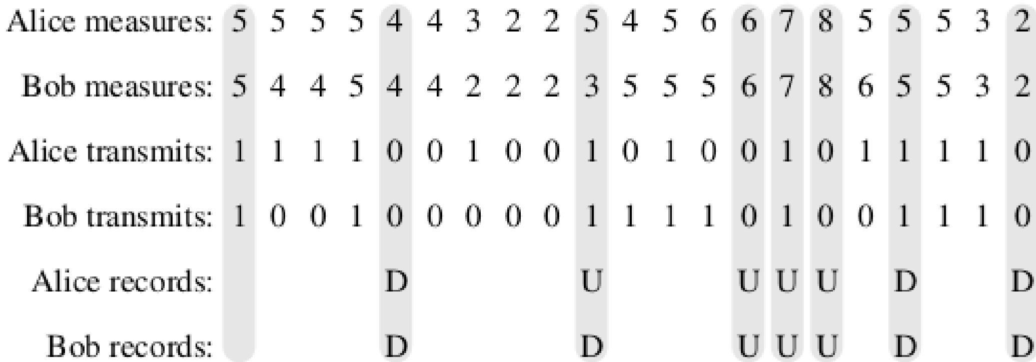

Figura 3.4: Exemplo de geração de chaves preliminares

Deve-se notar que ainda se possui alguns erros. Então é aplicado um protocolo de correção de erros várias vezes até que as chaves tornem-se idênticas. Quando Alice e Bob estão convencidos de que possuem a mesma chave, então elas geram um hash que será usado como a chave final.

Os testes foram feitos em acelerômetros WiTilt de 3 eixos que são capazes de medir até $6G$. Comparado aos trabalhos anteriores, este foi capaz de produzir mais bits de entropia pois em vez de usar o vetor de magnitude dos 3 eixos, ele concatena as medições de cada eixo para formar uma única *string*.

Capítulo 4

Metodologia

O presente trabalho se diferencia dos anteriores por aplicar os métodos de reconciliação de chaves BBBSS Otimizado e Cascade às chaves similares para correção de erros e geração de uma chave comum. Além disso, a extração das chaves utiliza Wavelets para normalização dos valores obtidos de cada eixo do acelerômetro. Também é importante salientar que no lugar de dispositivos especializados em dados de aceleração (como MEMS), foram utilizados aparelhos comerciais, como o Motorola Xoom e o Samsung Galaxy, com todas as suas limitações de recursos e ruídos intrínsecos a eles.

Também é possível explorar melhor a entropia fornecida pelos acelerômetros, pois não há perda de informação devido a normalização dos eixos pela magnitude, se comparado à solução de [14], [15] e [16].

O trabalho foi dividido de acordo com o diagrama 4.1 em 5 etapas: aquisição de dados, alinhamento temporal, alinhamento espacial, extração de bits e reconciliação de chaves. As 3 primeiras consistem em capturar os dados e corrigir as perturbações causadas por falhas na medição, falta de sincronismo, ruído, capacidade do dispositivo e interferências do ambiente. A quarta é responsável por transformar os dados obtidos em cadeias binárias. A última gera as chaves criptográficas e provê a segurança necessária. Estas tarefas são explicadas adiante.

4.1 Extração de bits

Foi proposto um modelo para tratamento dos dados obtidos pelo acelerômetro [14]. Este tratamento é dividido em 3 tarefas: aquisição de dados do sensor, alinhamento temporal e alinhamento espacial. Neste trabalho foi adicionada a transformação de decimal para binário. Isso é necessário para a aplicação do protocolo Cascade.

4.1.1 Aquisição de dados

A tarefa de aquisição de dados consiste em não apenas coletar os dados, mas também definir a taxa de amostragem. Estes dados devem ser coletados localmente e, por questões de segurança, não devem trafegar pela rede sem fio. No trabalho de [14], uma

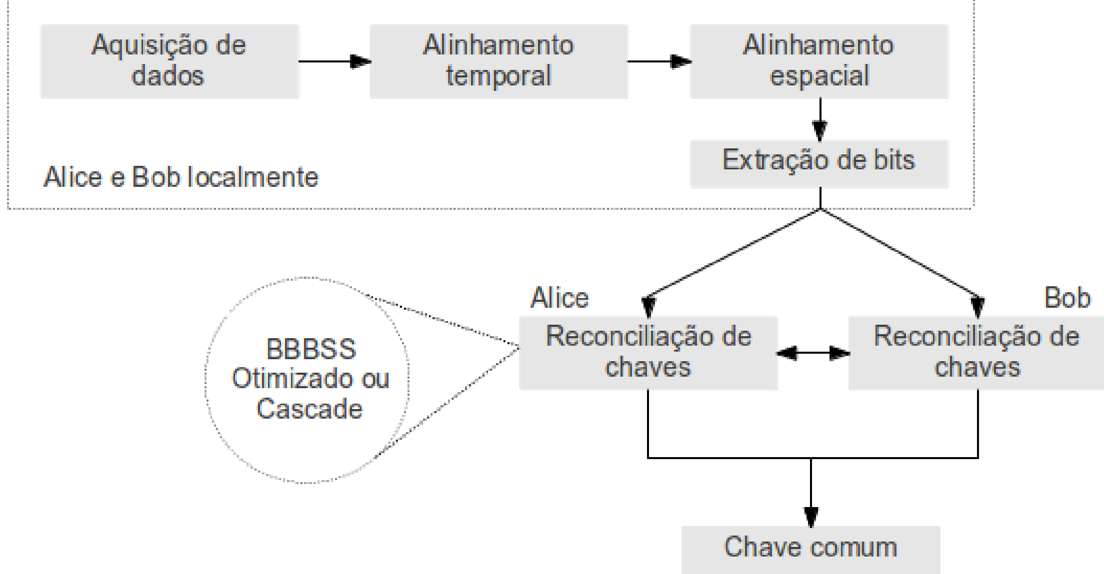

Figura 4.1: Diagrama de tarefas

taxa de amostragem de 100 a 600Hz foi identificada como apropriada. Em [10], foi utilizado 220Hz. Porém, no presente trabalho, foi utilizado a taxa de 50Hz devido a maior velocidade de medição dos dispositivos atuais. Contudo, esta diminuição na taxa não provoca grandes efeitos nas medições pois uma taxa abaixo de 100Hz não possui diferença significante no movimento provocado pelo usuário.

Para a leitura utilizou-se a API de Java SensorManager com tipo de acelerometro *TYPE_ACCELEROMETER*. Para taxa de amostragem o parâmetro foi *SENSOR_DELAY_FASTEST*. Os dados são armazenados no diretório raiz dos aparelhos sem nenhum cálculo adicional e processados posteriormente em um computador.

4.1.2 Alinhamento temporal

A segunda tarefa diz respeito à sincronização temporal para comparação. Para que as chaves sejam correlatas, é necessário que haja um consenso do sobre o início da medição e que as medições sejam amostradas no mesmo intervalo de tempo.

A inicialização da medição pode ser feito por gatilho manual (explícito) ou detecção de uma movimentação brusca (implícito). Existem diversos modelos na literatura para identificar esses movimentos [11] [8]. Contudo, este trabalho utiliza o disparo, onde o usuário pressiona um botão na tela do dispositivo.

Como esse disparo é feito manualmente pelo usuário, pode ocorrer de um dos dispositivos começar a medição um pouco antes do outro. Uma maneira de resolver isso é fazer com que um deles envie uma mensagem ao outro indicando o início da medição.

Neste trabalho, utilizou-se o disparo manual em ambos os aparelhos e uma correção do início através da troca de mensagens. Bob envia a Alice um trecho do início de sua medição (no mínimo $400ms$). Alice então realiza uma comparação entre os trechos (com uma certa tolerância a erros) para identificar o começo da gravação. Uma vez localizado, Alice e Bob irão considerar apenas as medições depois do trecho enviado pela mensagem, descartando a parte enviada pela rede. Na figura 4.2 a área hachurada representa a parte enviada por Bob à Alice. Toda esta área (e a parte anterior a ela) será descarta para garantir a segurança da chave. Para determinar o fim da gravação, os dispositivos trocam informações sobre quanto tempo de medição eles vão considerar. Isso é feito para cada eixo separadamente e, caso não consigam sincronizar, a leitura do eixo inteiro é descartada.

A variação da taxa de amostragem entre os aparelhos pode fazer com que as medições distoem umas das outras por diferença de alguns milissegundos ao longo do tempo. Por exemplo, se um aparelho está medindo a uma taxa de $49Hz$ e outro a $51Hz$, a comparação dos últimos valores lidos terá uma divergência muito grande. Para evitá-la, é feita a média de todos os valores medidos dentro do intervalo de $100ms$.

4.1.3 Alinhamento espacial

A última tarefa é responsável por normalizar os dados de cada eixo. Isso porque a medição sofre interferência da orientação dos aparelhos. É muito difícil realizar a coleta sem provocar mudanças devido ao ângulo de inclinação. Além disso, a própria diferença de precisão dos dispositivos gera ruídos e distorções na medição. [14] propôs utilizar o vetor resultante das 3 dimensões (magnitude do vetor). Desta forma, elimina-se a diferença devido a orientação. No entanto, uma parte da informação é perdida neste método pois cada eixo deixa de ser apresentado independente. O cálculo da magnitude do vetor resultante na posição i ($w[i]$) é dado pela equação abaixo e x, y e z são os vetores de medição de cada eixo.

$$w[i] = \sqrt{x[i]^2 + y[i]^2 + z[i]^2}$$

Neste trabalho não utilizou-se a magnitude do vetor com o intuito de produzir chaves com maior entropia. Desta forma, foi necessário dispor os aparelhos com a orientação bem próxima (um aparelho sobre o outro). Pode ser necessário calibrar os aparelhos se as medições forem discrepantes. Essa calibragem pode ser feita a partir da troca de informações a respeito do direcionamento do aparelho no início da medição. Para evitar problemas com ruídos e normalizar a amplitude da medição, foram utilizadas transformadas Wavelets. O operador Wavelet utilizado foi o Haar. A string final é obtida concatenando-se todos os eixos:

$$w = x||y||z$$

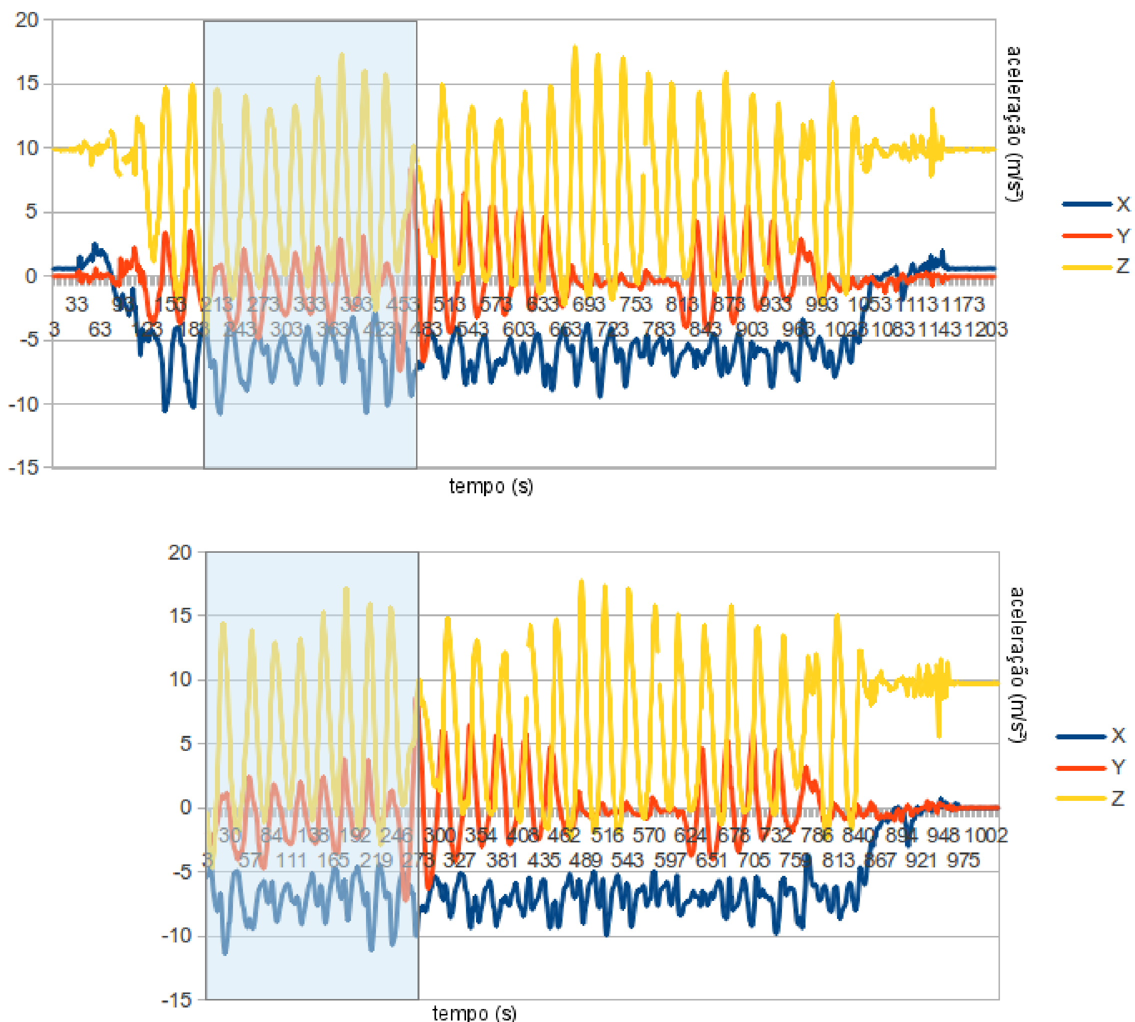

Figura 4.2: Sincronização temporal nos dispositivo 1 e 2

4.1.4 Decimal para binário

Neste momento, os dados estão correlatos, mas ainda são números decimais. O protocolo de reconciliação de chaves utiliza *strings* binárias. Portanto, é necessário transformar os dados em um conjunto de *bits*. Para tanto, foi utilizada uma tabela de conversão para cada intervalo. São 3 tabelas (4.1, 4.2 e 4.3). Cada tabela transforma o decimal para uma cadeia binária de 3, 4 e 5*bits* respectivamente. Na seção de experimentos a utilização de cada uma destas tabelas será analisada para identificar a mais apropriada.

Para cada valor, verifica-se na tabela o código binário correspondente para gerar a *string*. É importante notar que a variação entre dois intervalos vizinhos é de apenas um *bit*. Esta estratégia foi adotada para minimizar os efeitos de uma possível mudança no

intervalo devido a falha de leitura. Assim, um erro pequeno deveria provocar apenas um *bit* de diferença.

Tabela 4.1: Conversão de decimal para binário - 3 bits

Intervalo	Binário
≤ -2.5	000
> -2.5 e ≤ -1.5	001
> -1.5 e ≤ -0.5	011
> -0.5 e ≤ 0.5	010
> 0.5 e ≤ 1.5	110
> 1.5 e ≤ 2.5	111
> 2.5 e ≤ 3.5	101
> 3.5	100

Tabela 4.2: Conversão de decimal para binário - 4 bits

Intervalo	Binário
≤ -3.5	1000
> -3.5 e ≤ -2.5	1001
> -2.5 e ≤ -1.5	1011
> -1.5 e ≤ -0.5	1010
> -0.5 e ≤ 0.5	0010
> 0.5 e ≤ 1.5	0011
> 1.5 e ≤ 2.5	0001
> 2.5 e ≤ 3.5	0000
> 3.5	0100

4.2 Reconciliação de Chaves

Como citado anteriormente, o BBBSS e o Cascade são protocolos para gerar uma chave comum entre duas partes que possuem chaves parecidas sem comprometer a segurança da chave. Isso significa que as partes possuem um canal de comunicação inseguro para troca de informações. Este canal pode ser entendido de várias formas e, neste trabalho, ele é representado pela rede de comunicação entre dois dispositivos móveis (Bluetooth, Wi-Fi e etc).

Desta forma, antes de entrar nos detalhes do protocolo, é necessário definir uma arquitetura abstrata o suficiente para se adaptar a qualquer tipo de comunicação.

Nesta Seção, são detalhados a arquitetura geral, os métodos comuns aos dois protocolos e em seguida os métodos do BBBSS e do Cascade individualmente.

Tabela 4.3: Conversão de decimal para binário - 5 bits

Intervalo	Binário
≤ -15.5	11110
> -15.5 e ≤ -14.5	11111
> -14.5 e ≤ -13.5	11101
> -13.5 e ≤ -12.5	11100
> -12.5 e ≤ -11.5	10100
> -11.5 e ≤ -10.5	10101
> -10.5 e ≤ -9.5	10111
> -9.5 e ≤ -8.5	10110
> -8.5 e ≤ -7.5	10010
> -7.5 e ≤ -6.5	10011
> -6.5 e ≤ -5.5	10001
> -5.5 e ≤ -4.5	10000
> -4.5 e ≤ -3.5	11000
> -3.5 e ≤ -2.5	11001
> -2.5 e ≤ -1.5	11011
> -1.5 e ≤ -0.5	11010
> -0.5 e ≤ 0.5	01010
> 0.5 e ≤ 1.5	01011
> 1.5 e ≤ 2.5	01001
> 2.5 e ≤ 3.5	01000
> 3.5 e ≤ 4.5	00000
> 4.5 e ≤ 5.5	00001
> 5.5 e ≤ 6.5	00011
> 6.5 e ≤ 7.5	00010
> 7.5 e ≤ 8.5	00110
> 8.5 e ≤ 9.5	00111
> 9.5 e ≤ 10.5	00101
> 10.5 e ≤ 11.5	00100
> 11.5 e ≤ 12.5	01100
> 12.5 e ≤ 13.5	01101
> 13.5 e ≤ 14.5	01111
> 14.5	01110

4.2.1 Arquitetura

Foi utilizado um modelo *Cliente/Servidor*(figura 4.3) onde ambos possuem uma classe *Conexao*. Esta conexão pode ser utilizada com qualquer tipo suportado por Java, desde que *Streams* de entrada e saída sejam passados como parâmetro para o construtor. Esta arquitetura é necessária porque nos protocolos de reconciliação de chaves a execução dos métodos ocorre de forma diferente em cada um dos lados. Por exemplo, se um bit precisa ser corrigido, então apenas um dos lados vai fazê-lo.

O método *envia(String msg)*, envia uma String ao outro lado da conexão, enquanto os métodos *recebe()* e *recebe(String msg)* ficam aguardando até que a conexão receba alguma mensagem. Todas as mensagens são do tipo "cabeçalho : parametro1 : parametro2 : ... : parametroN". Se o método *recebe(String msg)* for utilizado, a aplicação irá aguardar uma mensagem com o cabeçalho indicado no parâmetro, descartando todas as outras. Além disso, todos os métodos para envio de mensagem incrementam o contador *mensagensEnviadas* sempre que uma mensagem é trafegada pela rede. Esta será uma das métricas nos experimentos.

Os protocolos BBBSS e Cascade possuem alguns métodos em comum. Por isso, eles herdam a classe *Comum*. Esta classe armazena o tamanho inicial dos blocos, a taxa de erro esperada, a quantidade de bits corrigidos e trafegados pela rede. Além disso, também possui um vetor de chaves, que são do tipo *BitArray*. Em muitos protocolos se usa apenas uma chave. Porém, no caso do Cascade, é necessário manter o histórico das mesmas em cada passo. Desta forma, se foi definido como 5 passos, então tem-se um vetor de 5 chaves. É também nesta classe que ficam as conexões.

BitArray é uma classe auxiliar correspondente a um vetor de booleanos onde cada elemento corresponde a um bit da chave. Ela contém métodos para manipulação dos elementos da chave como *and*, *xor*, *subList*, *parity* e etc. Também, para cada elemento, existe um inteiro indicando a posição dele na chave. Isso facilita o trabalho pois, se houver uma permutação na chave (o que é comum nos protocolos de reconciliação), a posição original do elemento é mantida.

4.2.2 Métodos Comuns

Nesta subseção os métodos da classe *Comum* são explicados individualmente. Para quase todos os métodos existentes no cliente existe um correspondente no servidor que é responsável por responder as requisições. Estes são marcados com _ (*underscore*) e, salvo quando necessário, não serão detalhados pois executam a mesma função do método no lado requisitante.

método *permutacao()*

No cliente é responsável por gerar um vetor de números randômicos e enviá-lo ao servidor. Então, em ambos os lados, ordena-se o vetor de randomicos aplicando a mesma ordenação a chave. A figura 4.4 mostra um exemplo de permutação onde K é a chave antes da permutação e VR é o vetor randômico antes da ordenação. A 3ª e 4ª linha

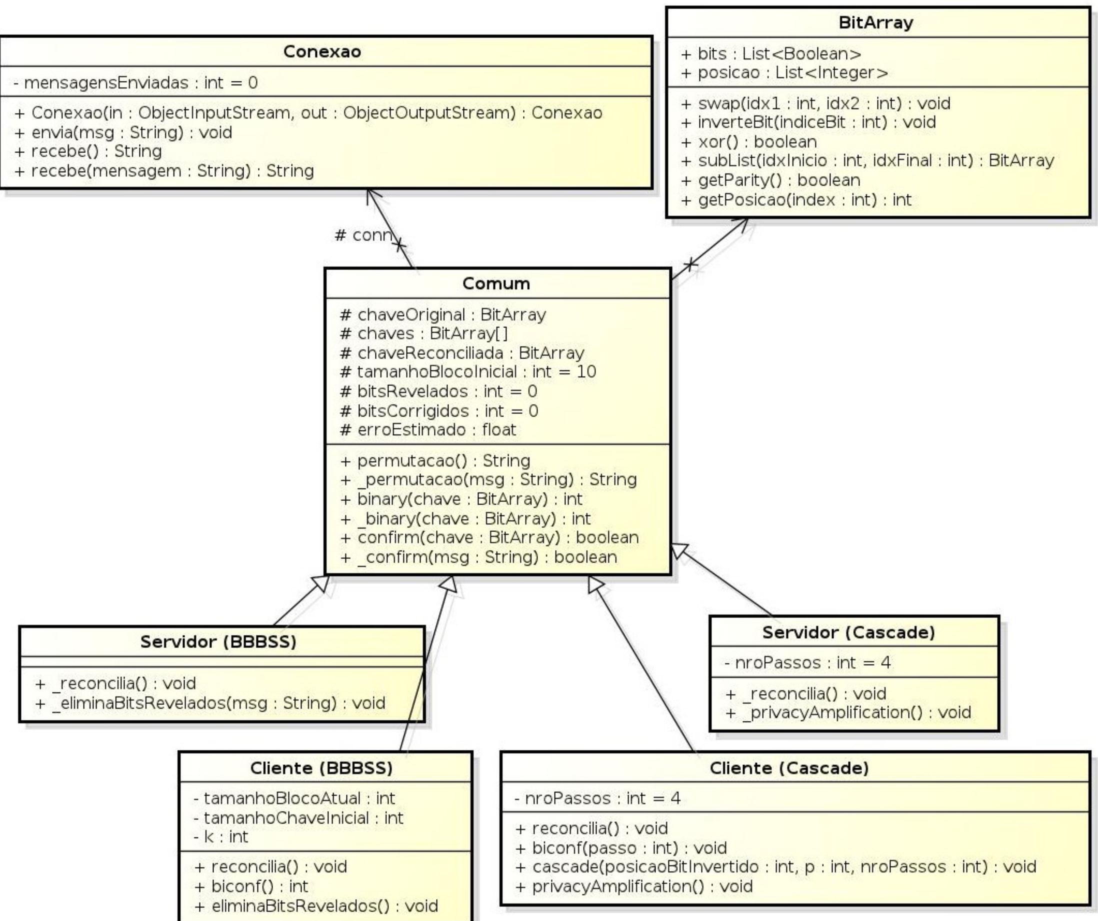

Figura 4.3: Arquitetura: Diagrama de Classes

representam a chave permutada e o vetor ordenado. Os valores destacados são apenas para visualização de parte do bloco.

O pseudocódigo 1 detalha o método.

método *confirm(BitArray K)*

O cliente calcula a paridade de um determinado bloco de chave K. Então solicita ao servidor que faça o mesmo e lhe envie o resultado. Para isso, o cliente deve enviar o indice inicial do bloco e o tamanho da cada bloco ao servidor. No caso do Cascade, ele também deve fornecer o passo em que o método se encontra para o servidor saber qual chave utilizar.

K	1	0	0	1	1	0	1	0	0	0	1	1	0
VR	4	5	3	2	1	9	5	9	7	2	5	1	4

K permutado	1	1	1	0	0	1	0	0	1	1	0	0	0
VR ordenado	1	1	2	2	3	4	4	5	5	5	7	9	9

Figura 4.4: Permutacao

Algorithm 1 Método *permutacao()*

for $i = 1 \rightarrow |K|$ **do**
  $permutacao_i \leftarrow$ random()
envia("permutacao:"+ $permutacao$)
for $i = 1 \rightarrow |K|$ **do**
  for $j = i + 1 \rightarrow |K|$ **do**
    if $permutacao_j \leq permutacao_i$ **then**
      swap($permutacao_i, permutacao_j$)
      swap(K_i, K_j)

método *binary(BitArray K)*

O cliente divide o bloco de chave K em duas partes e envia a paridade da primeira metade (à esquerda) ao servidor. Então o servidor compara com a paridade da mesma parte e retorna *true* caso sejam iguais ou *false* se diferentes. Então o cliente sabe em qual das partes está o número ímpar de erros. Neste momento, ele utiliza a parte que está com erro e aplica *binary* recursivamente até que o erro seja encontrado. O algoritmo 2 contém os detalhes.

4.2.3 BBBSS Otimizado

método *reconcilia()*

No cliente, este método define o tamanho inicial do bloco w_0. Este é calculado de acordo com a taxa de erro esperada de forma que em cada bloco contenha 1 erro. Por exemplo, se a taxa de erro, representada por e_0 é de 15%, então o bloco terá 7 bits (6.7 arredondado para cima). O tamanho inicial da chave deve ser armazenado (em n) para ser usado mais tarde na fórmula de cálculo do tamanho do bloco. Então os envia ao servidor que entrará em modo de espera pelos comandos do cliente.

Agora o cliente entra em um loop com k iterações, onde k define a probabilidade $\frac{1}{2^k}$

Algorithm 2 Método *binary(BitArray K)*

```
if |K| = 1 then
    return K[1]
else
    paridadeEsquerda ← paridade(K[1, meio])
    result ← envia("binary:"+ paridadeEsquerda)
    if result = true then                    ▷ O bit invertido está na segunda metade
        return binary(K[meio, fim])
    else                                     ▷ O bit invertido está na primeira metade
        return binary(K[1, meio])
```

de o método não encontrar um erro. Neste trabalho utilizou-se $k = 10$.

Para cada iteração, aplica-se uma permutação na chave para distribuir os erros. Então executa-se o método *biconf()* para encontrar e corrigir erros nos blocos. O método *reconcilia()* termina se em k iterações seguidas não encontrar erros.

Neste ponto o método *eliminaBitsRevelados()* é chamado para remover o último bit de cada bloco cuja paridade foi trafegada pela rede.

Por fim, sempre que se encontra e corrige pelo menos um erro, o contador de iterações retorna ao estado inicial. Neste caso, a taxa de erros diminui e, portanto, o tamanho dos blocos é calculado novamente de acordo com a nova probabilidade. A taxa de erro e o tamanho do bloco na iteração i são chamados de e_i e w_i respectivamente.

O *reconcilia()* também pode ser interrompido caso o tamanho dos blocos da chave seja maior do que a própria chave. Isso pode ocorrer porque a chave diminui ao longo do algoritmo por causa dos descartes. Adicionalmente, o tamanho dos blocos da chave aumenta conforme são corrigidos erros.

Ao receber o comando *fim*, o servidor encerra suas atividades. O algoritmo 3 contém a descrição do método.

método *biconf()*

O método pode ser conferido no pseudocódigo 4. A chave é dividida em $n = |K|/w_i$ blocos de tamanho w_i (tamanho do bloco na iteração i), onde $bloco_b$ é o bloco da posição b. Para cada bloco, executa-se o *confirm()* para verificar se existe um número ímpar de erros no bloco. Caso se confirme, então executa-se o *binary()* para encontrar e corrigí-lo. O método retorna o número de erros corrigidos nesta iteração.

método *eliminaBitsRevelados()*

Durante a reconciliação de chaves muitos bits são revelados. Apesar de serem bits de paridade e não os bits da chave, eles podem comprometer a segurança revelando um mínimo de informação das chaves. Uma das formas de diminuir esse problema é descartar o último bit de um bloco cuja paridade foi revelada. Assim, um adversário não teria nenhuma informação a respeito.

Algorithm 3 Método *reconciliacao()*

```
w_0 ← ⌈1/e_0⌉
w_i ← w_0
n ← |K|
envia("inicio")
for i = 1 → k do
    if |K| ≥ w_i then
        break
    permutacao()
    errosEncontrados ← biconf()
    eliminaBitsRevelados()
    if errosEncontrados > 0 then
        bitsCorrigidos ← bitsCorrigidos + errosEncontrados
        e_i ← (n * e − bitsCorrigidos)/|K|
        w_i ← ⌈1/e_i⌉
        i ← 1
envia("fim")
```

Algorithm 4 Método *biconf()*

```
errosEncontrados ← 0
for b = 1 → |K|/w_i do
    if ¬ confirm(bloco_b) then
        posicaoBitErrado ← binary(bloco_b)
        inverteBitErrado(posicaoBitErrado)
        errosEncontrados ← errosEncontrados + 1
```

O pseudocódigo 5 demonstra como fazê-lo. O bloco é percorrido de $|K|/w_i$ até 1, que correspondem aos índices último e do primeiro bloco respectivamente. Para cada bloco é removido o último bit ($idxBit$). O método realiza a operação do final para o ínicio para não ter que lidar com os blocos seguintes após a remoção de um bit.

Algorithm 5 Método *eliminaBitsRevelados()*

```
for b = |K|/w_i → 1 do
    posicaoBitErrado ← binary(bloco_b)
    idxBit ← b * w_i
    removeBit(K,idxBit)
```

4.2.4 Cascade

método *reconcilia()*

Assim como no BBBSS, este método define o tamanho inicial do bloco (w), calculado a partir do erro estimado (e). Adicionalmente, o número máximo de passos do Cascade é configurado.

Então os envia ao servidor que entrará em modo de espera.

Depois de enviados os parâmetros ao servidor, o cliente entra em um loop de NRO_MAX_PASSOS iterações. Em cada iteração, a chave do passo p é clonada (copiada) da chave $p-1$. Então aplica-se a *permutacao* e *biconf*. O pseudocódigo 6 ilustra esse procedimento.

Após o término do protocolo, *privacyAmplification* é executado para eliminar qualquer informação de um adversário.

Algorithm 6 Método *reconciliacao()*

```
w ← ⌈1/e⌉
envia("inicio:"+NRO_MAX_PASSOS + ":"+ w)
p ← 1
while p ≤ NRO_MAX_PASSOS do
    K_p ← K_{p-1}.clone()
    permutacao(p)
    biconf(p)
    p ← p + 1
envia("fim")
privacyAmplification()
```

método *biconf(int p)*

Este método encontra-se apenas no cliente e tem como finalidade executar a primitiva BICONF para cada passo p. A primitiva é simplesmente a combinação de CONFIRM e

BINARY. Então ele divide a chave K_p em n blocos, onde $n = |K|/w_p$ e w_p é o tamanho do bloco no passo p.

Para cada bloco, ele executa $confirm()$ para verificar se existe um número ímpar de erros no bloco. Caso exista, ele executa o $binary()$ para encontrá-lo e corrigir. Essa correção é feita invertendo-se o bit errado, mas poderia ser feito por meio do descarte de bit. Toda vez que um bit é corrigido, o método $cascade()$ é acionado para correção dos bits nos passos anteriores. O algoritmo 7 mostra uma descrição completa de $biconf()$. Os métodos $confirm()$, $binary()$ e $cascade()$ são detalhados a seguir.

Algorithm 7 Método *biconf(int p)*

$w_p \leftarrow w * 2^{p-1}$
for $b = 1 \rightarrow |K|/w_p$ **do**
  if $\neg$ confirm($bloco_b$, p) **then**
    $posicaoBitErrado \leftarrow$ binary($bloco_b$, p)
    for $i = 1 \rightarrow p$ **do**
      inverteBitErrado(K_i,$posicaoBitErrado$)
    cascade($posicaoBitErrado$, p)

método *cascade(int p, int posicaoBitErrado)*

Este é o principal método (algoritmo 8) que diferencia o protocolo Cascade dos outros de reconciliação de chaves. Ele corrige os erros dos passos anteriores em cascata. Por exemplo, considere que o número de erros no passo p é par, não sendo detectado pelo $CONFIRM$. Se no passo $p + 1$ algum erro for corrigido, então pode-se corrigir esse erro nos passos $p, p - 1, p - 2, ..., 1$ e, consequentemente torná-los com número de erros ímpar novamente. Desta forma, executa-se o cascade recursivamente para todos os passo anteriores.

Algorithm 8 Método *cascade(int p, int posicaoBitErrado)*

if $p < 1$ **then**
  return
$w_p \leftarrow w * 2^{p-1}$
$idxInicial \leftarrow \lfloor posicaoBitErrado/w_p \rfloor * w_p$
$idxFinal \leftarrow idxInicial + tamanhoBloco$
$subK \leftarrow K_p[idxInicial, idxFinal]$
if $\neg$confirm($subK$, p) **then**
  $posicaoBitErrado2 \leftarrow$ binary($subK$)
  for $i = 1 \rightarrow passoAtual$ **do**
    ▷ A variável passoAtual armazena o passo que está sendo executado
    inverteBitErrado(K_i,$posicaoBitErrado2$)
  cascade($p - 1$, $posicaoBitErrado2$)

método *privacyAmplification()*

Para eliminar informações fornecidas ao adversário através dos bits de paridade trafegados pela rede, o BBBSS utiliza a técnica de descarte descrita anteriormente. No entanto, como o Cascade revela mais bits de paridade que os outros métodos, haveria muito descarte de bits, dificultando o processo. Ao invés disso, utiliza-se o Privacy Amplification, que gera uma chave de tamanho $m = n - r$, onde n é o tamanho original da chave e r o número de bits revelados.

Para realizar esse processo, um contador de bits revelados é incrementado toda vez que um bit é trafegado pela rede nos métodos *confirm* e *binary* do Cascade. Então esse contador será usado como parâmetro no método *privacyAmplification*.

O algoritmo precisa de uma sequencia randômica que será utilizada tanto no cliente e servidor. Não há problemas na segurança se a sequência for conhecida porque será usada como uma espécie de função *hash*. Desta forma ela pode ser trafegada pela rede.

No entanto, para chaves grandes, a cadeia de bits seria muito grande pois é uma matriz mxn. Assim, para agilizar o processo, utilizou-se uma *seed* de números randômicos que será transmitida do cliente ao servidor, fazendo com que ambos produzam a mesma sequência.

O pseudocódigo 9 descreve o método. K é a chave original e nK é a nova chave de tamanho m. A matriz de tamanho mxn gerada a partir de uma semente é de booleanos. Então, cada linha i da matriz é somada a chave K e então o *xor* da linha resultante é atribuído ao elemento i da chave nK. Este processo é na verdade uma multiplicação de matrizes. Ao final de m iterações, obtem-se a nova chave.

Algorithm 9 Método *privacyAmplification(BitArray K, int m, long seed)*

$matrizRandomica[m][|K|] \leftarrow \text{random}(seed)$
for $i = 0 \rightarrow m$ **do**
  $nK_i \leftarrow \otimes(matrizRandomica[m] \wedge K)$
return nK

Capítulo 5

Experimentos e análise de resultados

Este capítulo contém detalhes dos testes realizados como ambiente de execução, especificação dos equipamentos e metodologia de testes. Os testes são divididos em extração de bits e reconciliação de chaves. Neste último há uma comparação de desempenho com os trabalhos anteriores [14] [15] [16] [10].

5.1 Experimentos

Para a extração dos dados do acelerômetro, usou-se dois aparelhos Motorola Xoom e dois Samsung Galaxy. As especificações estão na tabela 5.1. Para obtenção dos dados, foi utilizada uma API de Java para Android, a Accelerometer Sensor Manager, que é capaz medir a aceleração do aparelho nos 3 eixos. O início e o fim de cada extração são acionados por um botão na tela dos Smartphones.

Tabela 5.1: Especificações

	Galaxy (GT-P7510)	Xoom (MZ605)	Xoom (MZxoom605)
Processador	Dual Core 1.0 GHz	Dual Core 1.0 GHz	Dual Core 1.0 GHz
Memória	1GB	1GB	1GB
Versão do Android	3.2	3.2	4.0.4
Kernel	2.6.36.3	2.6.36.3	2.6.39.4-g42a0480

Apesar de a coleta ser local, o processamento foi realizado em um computador Intel Core 2 Duo com 4 GB de memória.

Foram 50 experimentos no total (25 com cada par de aparelhos), com duração de 5 segundos cada. Os 50 testes foram chamados de amostras correlatas. Isso significa que foram movimentados simultaneamente pela mesma pessoa, com um dispositivo posicionado sobre o outro.

Ao combinar uma das amostras de um teste com outra amostra diferente (balançados não simultaneamente), pode-se obter um teste que foi chamado de não-correlato. Testes não-correlatos não devem conseguir gerar uma chave comum. Desta forma, é possível combinar os 25 testes de um aparelho entre si totalizando 600 testes ($(25 * 25 - 25)$).

Os testes foram divididos em duas categorias: extração de dados e reconciliação de chaves.

5.1.1 Extração de bits

O primeiro teste, referenciado nas tabelas 5.2 e 5.3, contém a quantidade bruta de bits (coluna 2) gerados antes de se aplicar a reconciliação de chaves. Estes bits foram usados para gerar a chave final que, obviamente, será de um tamanho menor devido às partes descartadas para garantir a segurança. Foram testados com as 3 tabelas de conversão de decimal para binário apresentadas no capítulo anterior individualmente.

Tabela 5.2: Extração de bits - Taxa de erro (Motorola)

	Bits		Taxa de erro	
	média	desvio padrão	média	desvio padrão
Correlatos (3 bits)	441.000	0.000	0.106	0.032
Não correlatos (3 bits)	-	-	0.329	0.048
Correlatos (4 bits)	588.000	0.000	0.090	0.028
Não correlatos (4 bits)	-	-	0.305	0.046
Correlatos (5 bits)	735.000	0.000	0.126	0.036
Não correlatos (5 bits)	-	-	0.308	0.038

Tabela 5.3: Extração de bits - Taxa de erro (Samsung)

	Bits		Taxa de erro	
	média	desvio padrão	média	desvio padrão
Correlatos (3 bits)	441.120	4.013	0.109	0.037
Não correlatos (3 bits)	-	-	0.358	0.038
Correlatos (4 bits)	588.960	3.322	0.090	0.030
Não correlatos (4 bits)	-	-	0.329	0.036
Correlatos (5 bits)	736.200	4.153	0.124	0.043
Não correlatos (5 bits)	-	-	0.341	0.037

Como nesta fase ainda não foi aplicada nenhuma reconciliação, é esperado que haja uma diferença entre os bits. Esta diferença encontra-se na taxa de erro da coluna 4. A maioria dos testes correlatos possuem taxa abaixo de 12%, contra 25% dos testes não correlatos. Esta taxa é um bom indicativo do parâmetro a ser usado no protocolo de reconciliação como taxa de erro.

Os dados em destaque correspondem a taxa de erro dos testes correlatos e não correlatos da tabela de conversão de 4 *bits*. Esta teve o melhor desempenho no que diz

respeito a taxa de erro média dos testes correlatos, de apenas 9%. Esta foi escolhida como tabela padrão para os teste de reconciliação.

5.1.2 Reconciliação de chaves

O segundo teste (tabelas 5.4, 5.5, 5.6 e 5.7) reconcilia as chaves obtidas no primeiro através dos protocolos Cascade e BBBSS. A taxa de erro utilizada variou de 5 a 16% para cada protocolo e o tamanho inicial do bloco foi de 7 a 20 *bits*, dependendo da taxa. As instâncias destacadas correspondem ao melhor resultado com relação a quantidade de falsos-negativos. No Cascade, o número máximo de passos foi 4 e no BBBSS o parâmetro k, que define a quantidade de rodadas para afirmar que não existe erros é 10, gerando uma probabilidade de 0.001 de falha. A cada passo, o tamanho do bloco é aumentado como explicado no capítulo anterior. Como citado anteriormente, todos os testes foram aplicados aos dados extraídos com a tabela de conversão de 4 *bits*.

Tabela 5.4: BBBSS - Motorola

Taxa de erro	Média de bits	Desvio padrão	Falsos-negativos	Falsos-positivos
0.05	464.5	30.4	0.92	0.0
0.06	444.7	60.0	0.84	0.0
0.07	427.7	54.3	0.68	0.0
0.08	389.6	54.4	0.48	0.0
0.09	304.3	55.2	0.36	0.0
0.10	314.5	61.6	0.32	0.0
0.11	302.7	75.7	0.16	0.0
0.12	270.1	86.7	0.04	0.0
0.13	228.7	75.6	0.04	0.0
0.14	203.4	49.1	0.16	0.0
0.15	187.2	41.3	0.28	0.0
0.16	171.5	32.5	0.52	0.0

Para classificar a reconciliação como sucesso, não só as chaves têm de ser idênticas (número de erros igual a 0), mas também devem ter um tamanho mínimo de 128*bits*. Isso porque algumas instâncias podem descartar muitos *bits* durante a fase de aumento de privacidade, tornando a chave tão pequena que não possua erros. Além disso, chaves menores que esse valor podem não ser seguras.

Pode-se observar que a quantidade *bits* gerados diminui a medida que a taxa de erro estimada aumenta. Isso ocorre porque o tamanho do bloco inicial dos blocos do protocolo diminuem, provocando um número maior de *bits* de paridade trocados pela rede. Desta forma, os métodos de ampliação da privacidade necessitam descartar mais *bits* para aumentar a segurança.

Outro efeito provocado pelo aumento da taxa de erro é a mudança na quantidade de falsos-negativos. O método se manteve seguro para ambos os dispositivos até uma taxa de 15%. Para o Cascade, 15% foi suficiente para gerar 4% de falsos-negativos, a

menor taxa. O BBBSS alcançou a mesma porcentagem utilizando a taxa de erro de 13%

Tabela 5.5: BBBSS - Samsung

Taxa de erro	Média de bits	Desvio padrão	Falsos-negativos	Falsos-positivos
0.05	473.0	27.8	0.88	0.0
0.06	438.0	34.6	0.76	0.0
0.07	402.8	52.6	0.64	0.0
0.08	374.7	46.6	0.60	0.0
0.09	316.4	69.2	0.32	0.0
0.10	317.7	75.9	0.28	0.0
0.11	300.1	78.0	0.12	0.0
0.12	267.3	68.4	0.20	0.0
0.13	238.6	74.1	0.04	0.0
0.14	200.1	51.2	0.08	0.0
0.15	181.5	48.8	0.24	0.0
0.16	172.7	36.0	0.44	0.0

Tabela 5.6: Cascade - Motorola

Taxa de erro	Média de bits	Desvio padrão	Falsos-negativos	Falsos-positivos
0.05	336.1	38.3	0.76	0.0
0.06	345.1	46.2	0.68	0.0
0.07	326.7	46.8	0.48	0.0
0.08	328.8	44.7	0.56	0.0
0.09	302.5	58.6	0.32	0.0
0.10	288.8	65.9	0.24	0.0
0.11	274.9	70.6	0.16	0.0
0.12	273.9	66.4	0.12	0.0
0.13	264.6	64.6	0.04	0.0
0.14	266.5	64.5	0.04	0.0
0.15	255.0	61.5	0.04	0.0
0.16	251.1	65.8	0.00	0.0

No Cascade, a taxa de 16% ocasionou 1 falso-positivo em 600 amostras (0.0016). Portanto, deve ser evitado a fim de garantir a segurança do método. No BBBSS, taxas acima de 14% provocaram altas taxas de falso-positivos, devendo estas não serem utilizadas também.

Houve um desvio padrão alto em relação a média de bits geradas. Isso porque uma pequena variação no ângulo entre os dispositivos pode causar muito impacto, uma vez que não é realizado nenhum procedimento de calibragem. No entanto, o importante é que todas as chaves tenha o tamanho mínimo de 128*bits* para garantir a segurança

Tabela 5.7: Cascade - Samsung

Taxa de erro	Média de bits	Desvio padrão	Falsos-negativos	Falsos-positivos
0.05	363.6	43.9	0.76	0.0
0.06	332.4	65.8	0.72	0.0
0.07	345.2	52.1	0.60	0.0
0.08	308.7	73.2	0.40	0.0
0.09	296.8	69.5	0.28	0.0
0.10	285.7	65.3	0.20	0.0
0.11	283.2	73.3	0.20	0.0
0.12	277.2	66.9	0.12	0.0
0.13	276.4	62.0	0.12	0.0
0.14	281.6	58.3	0.20	0.0
0.15	255.6	65.8	0.04	0.0
0.16	264.5	58.0	0.08	0.0016

desejada.

Entre as métricas deste experimento estão as trocas de mensagens pela rede dos protocolos. Toda vez que o cliente acessa a rede para enviar uma mensagem ao servidor ou vice-versa, um contador é incrementado em uma unidade, independente do tamanho da mensagem. As tabelas 5.8, 5.9, 5.10 e 5.11 contém o número de mensagens enviadas nos testes correlatos, não correlatos e ambos. Pode-se observar que o Cascade contém uma troca baixa de mensagens quando as amostras são parecidas. No entanto este número piora em testes não correlatos, onde o BBBSS se manteve estável.

Tabela 5.8: BBBSS - Motorola: troca de mensagens

Taxa de erro	Correlatos	Não correlatos	Todos
0.05	559.0	503.1	505.3
0.06	647.5	579.8	582.5
0.07	720.5	660.9	663.3
0.08	810.3	734.5	737.5
0.09	986.3	871.6	876.1
0.1	971.6	861.4	865.8
0.11	1034.7	941.8	945.5
0.12	1116.4	1003.8	1008.3
0.13	1190.1	1060.5	1065.6
0.14	1244.8	1139.1	1143.3
0.15	1291.3	1191.9	1195.9
0.16	1353.6	1255.2	1259.1

Tabela 5.9: BBBSS - Samsung: troca de mensagens

Taxa de erro	Correlatos	Não correlatos	Todos
0.05	571.3	504.1	506.8
0.06	655.6	584.2	587.1
0.07	744.6	662.3	665.6
0.08	806.5	737.8	740.5
0.09	977.8	874.6	878.7
0.1	978.5	865.0	869.5
0.11	1044.0	944.3	948.3
0.12	1110.4	1005.8	1010.0
0.13	1172.1	1069.8	1073.9
0.14	1243.0	1142.1	1146.1
0.15	1293.2	1194.6	1198.6
0.16	1357.0	1258.5	1262.4

Tabela 5.10: Cascade - Motorola: troca de mensagens

Taxa de erro	Correlatos	Não correlatos	Todos
0.05	590.0	626.6	625.1
0.06	572.0	728.1	721.9
0.07	615.8	806.9	799.3
0.08	609.8	915.5	903.3
0.09	672.1	1050.6	1035.5
0.1	705.1	1137.1	1119.8
0.11	737.8	1132.8	1117.0
0.12	740.5	1239.1	1219.2
0.13	764.4	1341.5	1318.5
0.14	759.7	1343.2	1319.9
0.15	782.5	1496.1	1467.6
0.16	792.4	1491.7	1463.8

Tabela 5.11: Cascade - Samsung: troca de mensagens

Taxa de erro	Correlatos	Não correlatos	Todos
0.05	526.6	636.2	631.8
0.06	599.4	728.9	723.7
0.07	571.5	796.6	787.6
0.08	658.9	921.2	910.8
0.09	687.2	1045.9	1031.6
0.1	715.0	1131.4	1114.7
0.11	719.4	1133.9	1117.3
0.12	734.8	1240.1	1219.9
0.13	737.0	1350.4	1325.9
0.14	722.9	1345.3	1320.4
0.15	783.8	1499.6	1471.0
0.16	761.8	1501.5	1471.9

Os gráficos 5.1, 5.2, 5.3 e 5.4 mostram a variação da quantidade de bits e o número de mensagens trafegadas na rede (ambos no eixo y primário) em relação à taxa de erro estimada. O eixo y secundário representa a taxa de falsos-negativos.

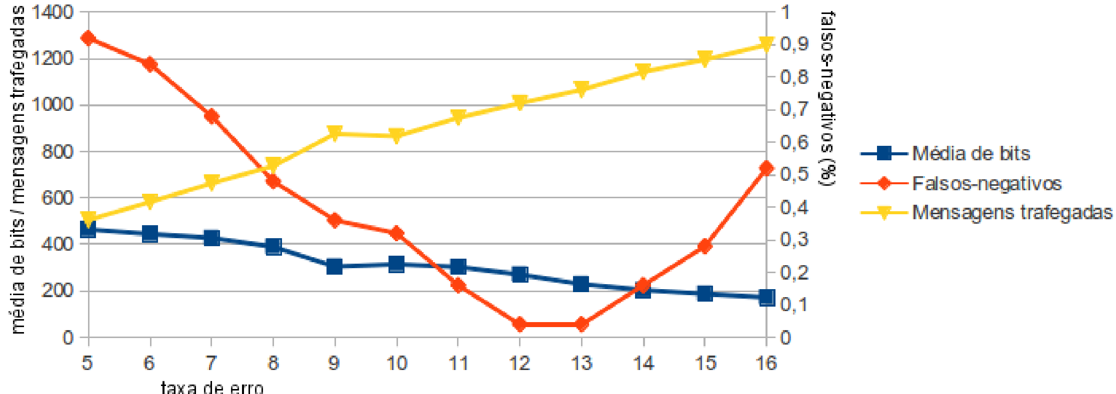

Figura 5.1: Gráfico: BBBSS - Motorola

A quantidade média de entropia gerados em 5s de teste no BBBSS foi de 233.65$bits$ para a instância de 13% de taxa de erro, enquanto no Cascade com 15% foi de 255.30$bits$. No entanto é possível aumentar essa média diminuindo-se a taxa. Porém, como consequência, as taxa de falsos-negativos também aumentam.

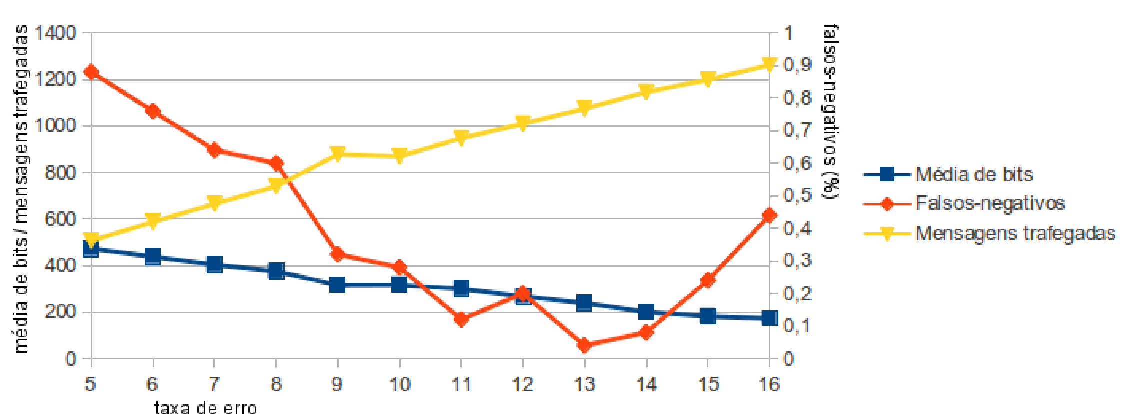

Figura 5.2: Gráfico: BBBSS - Samsung

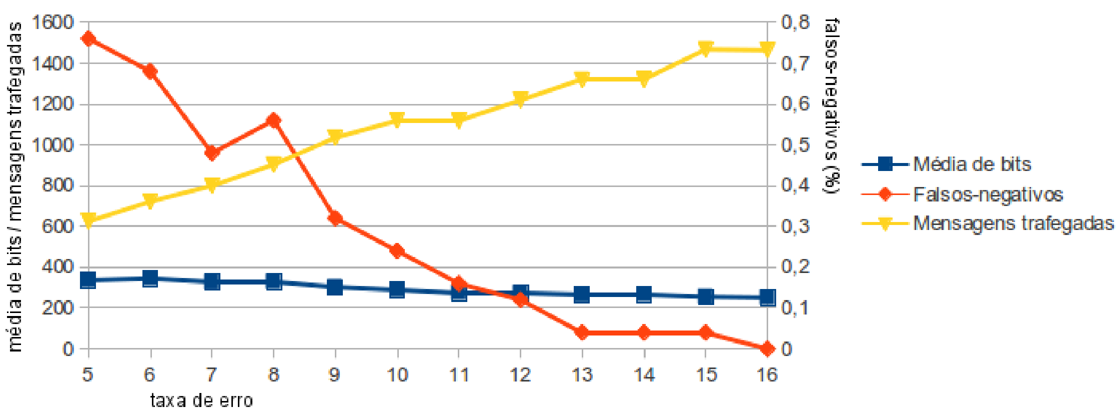

Figura 5.3: Gráfico: Cascade - Motorola

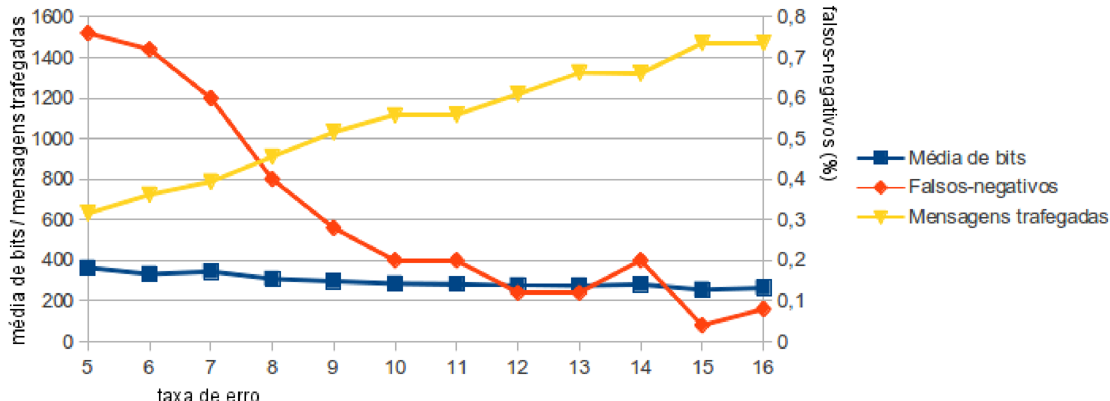

Figura 5.4: Gráfico: Cascade - Samsung

5.1.3 Análise de desempenho

A tabela 5.12 compara os resultados obtidos com os trabalhos citados anteriormente. Para esta comparação, foram tomadas a melhor instância do BBBSS e Cascade, 13% e 15% de taxa de erro respectivamente. Os testes dos dois dispositvos foram agrupados. Portanto, totalizam 1200 os testes correlatos e 50 os testes não correlatos.

Tabela 5.12: Análise de desempenho

	ShakeWell (ShaVe)	ShakeWell (ShaCK)	MartiniSync	BBBSS (13%)	Cascade (15%)
Falso-positivos	0.0	0.0	-	0.0	0.0
Falso-negativos	0.1024	0.12	0.02	0.04	0.04
Entropia (bits)/s	7 - 10	7 - 10	10 - 15	25.6 - 78	29.4 - 73.4
Tempo mínimo para chave de 128 bits	12 - 20 s	12 - 20 s	6 - 12 s	3 - 5 s	3 - 5 s

Pode-se observar que ambos tiveram melhor desempenho na quantidade de bits gerados por segundo (em destaque na tabela). Consequentemente, foi possível gerar uma chave confiável (mais de 128*bits*) em menos de 5*s* contra o mínimo de 6*s* do *MartiniSync*.

Com relação a confiabilidade do método, as instâncias escolhidas a execução demostraram boa segurança pois não apresentou nenhum falso-positivo, assim como o *MartiniSync* e o *ShakeWell*. No entanto teve 4% de falha quando os testes são correlatos, apresentando chaves diferentes, tendo desempenho inferior ao *MartiniSync*. A taxa de falso-positivos para o protocolo MartiniSync não foi informada pelo autor.

Capítulo 6

Considerações Finais

Este trabalho mostra que é possível aplicar os protocolos de reconciliação BBBSS e Cascade às chaves obtidas com acelerômetros com bastante precisão. O teste foi bastante satisfatório, conseguindo diferenciar entre amostras correlatas e não-correlatas. Além disso, produziu uma grande quantidade de bits em pouco tempo de interação com o usuário. Foi possível obter chaves de $128 bits$ com no máximo $5s$, contra os $6s$ e $12s$ mínimos exigidos pelos outros. Isso provê uma usabilidade confortável dos protocolos nos dispositivos, pois o usuário não precisa balançá-los por muito tempo.

Também pôde-se observar que não é necessário trabalhar com a magnitude dos 3 eixos desde que os aparelhos mantenham-se alinhados. Pôde-se notar a robustez dos métodos utilizados para extração e reconciliação, observando a diferença nos testes envolvendo amostragens não correspondentes. Estas possuem mais de 30% de erro em média.

Para sistemas que exigem maior segurança, é recomendado o Cascade com 15% de taxa de erro. Ele não permitiu que chaves de testes não correlatos produzam chaves iguais. E uma taxa de 4% de falhas nos testes correlatos não dificultam o procedimento que pode ser facilmente repetido pelo usuário.

O método se mostrou bastante flexível, permitindo aumentar a quantidade de bits gerada em detrimento da consistência (aumento de falsos-negativos). Isso pode ser aplicado em situações onde a taxa de aceitação é menos importante que o tamanho da chave gerada. Essa flexibilidade influencia diretamente o consumo de recursos pois quanto mais alto a taxa de erros utilizada, maior é a quantidade de mensagens enviadas pela rede para a execução dos protocolos.

Como trabalhos futuros pretendemos realizar testes com dispositivos diferentes (hardware, software e configurações) para analisar a melhor robustez do método. É interessante variar o número de passos do Cascade para obter resultados diferentes e o parâmetro k do BBBSS. Também é interessante avaliar outros dados de consumo de recursos nos dispositivos, como o processamento e a quantidade de memória envolvida. Outra avaliação importante é o consumo da bateria dos dispositivos. Por fim, calibrar os dispositivos antes do processo pode gerar resultados melhores.

Referências Bibliográficas

[1] C. Bennett, F. Bessette, G. Brassard, L. Salvail, and J. Smolin. Experimental quantum cryptography. *Journal of Cryptology*, 5:3–28, 1992.

[2] C. Bennett, G. Brassard, and U. M. Maurer. Generalized privacy amplification. *IEEE Transactions on Information Theory*, 41:1915–1923, 1995.

[3] C. H. Bennett, G. Brassard, and J.-M. Robert. Privacy amplification by public discussion. *SIAM J. Comput.*, 17:210–229, April 1988.

[4] G. Brassard and L. Salvail. Secret-key reconciliation by public discussion. In T. Helleseth, editor, *Advances in Cryptology - EUROCRYPT 93*, volume 765 of *Lecture Notes in Computer Science*, pages 410–423. Springer Berlin / Heidelberg, 1994. 10.1007/3-540-48285-735.

[5] J. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979.

[6] A. Developer. Api guides - sensors overview. http://developer.android.com/guide/topics/sensors/sensors_overview.html, 2011. [visitado em 26/11/2011].

[7] O. Goldreich. *Foundations of Cryptography: Volume I Basic Tools.* Cambridge University Press, Cambridge, 1st edition, 2001. ISBN 0–521–79172–3.

[8] T. Huynh and B. Schiele. Analyzing features for activity recognition. In *Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies*, sOc-EUSAI '05, pages 159–163, New York, NY, USA, 2005. ACM.

[9] J. Katz and Y. Lindell. *Introduction to Modern Cryptography.* Chapman & Hall/-CRC, Florida, USA, 1st edition, 2008. ISBN 978–1–58488–551–1.

[10] D. Kirovski, M. Sinclair, and D. Wilson. The martini synch. 2007.

[11] J. Lester, B. Hannaford, and G. Borriello. Are you with me? - Using accelerometers to determine if two devices are carried by the same person. In *Proceedings of Second International Conference on Pervasive Computing (Pervasive 2004*, pages 33–50, 2004.

[12] S. Liu. *Information-Theoretic Secret Key Agreement.* PhD thesis, Universiteitsdrukkerij Technische Universiteit Eindhoven, Holanda, 2002.

[13] D. MacKay. *Information Theory, Inference, and Learning Algorithms.* Cambridge University Press, Cambridge, UK, 1st edition, 2010. ISBN 978–0–521–64298–9.

[14] R. Mayrhofer and H. Gellersen. Shake well before use: authentication based on accelerometer data. In *Proceedings of the 5th international conference on Pervasive computing*, PERVASIVE'07, pages 144–161, Berlin, Heidelberg, 2007. Springer-Verlag.

[15] R. Mayrhofer and H. Gellersen. Shake well before use: two implementations for implicit context authentication. In *Adjunct Proc. Ubicomp 2007*, pages 72–75, September 2007.

[16] R. Mayrhofer and H. Gellersen. Shake well before use: Intuitive and secure pairing of mobile devices. *IEEE Trans. Mob. Comput.*, pages 792–806, 2009.

[17] A. Menezes, P. Van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography.* CRC Press Series on Discrete Mathematics and Its Applications. Crc Press, 1997.

[18] L. Peterson and B. Davie. *Computer Networks: a System Approach.* Morgan Kaufmann, San Francisco, CA, USA, 5th edition, 2011. ISBN 978–0–12–385059–1.

[19] B. Schneier. *Applied Cryptography.* Wiley, New York, 2nd edition, 1996. ISBN 0–471–59756–2.

[20] T. Shimizu, H. Iwai, and H. Sasaoka. Information reconciliation using reliability in secret key agreement scheme with espar antenna. In *Security and Privacy in Mobile Information and Communication Systems*, volume 17 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 148–159. Springer Berlin Heidelberg, 2009. 10.1007/978-3-642-04434-213.

[21] W. Stallings. *Cryptography and Network Security: Principles and Practice.* Prentice Hall, New York, 5th edition, 2010. ISBN 0–13–609704–9.

[22] K. Yamazaki, M. Osaki, and O. Hirota. On reconciliation of discrepant sequences shared through quantum mechanical channels. In E. Okamoto, G. I. Davida, and M. Mambo, editors, *ISW*, volume 1396 of *Lecture Notes in Computer Science*, pages 345–356. Springer, 1997.

[23] H. e. a. Yang. Security in mobile ad hoc networks: Challenges and solutions. *IEEE Wireless Communications*, 11(1):38–47, 2004.

[24] L. Zhou and J. Zygmunt. Securing ad hoc networks. *IEEE Network*, 16(6):24–30, 1999.

Apêndice A

Comparação visual de medições

Neste apêndice são exibidos gráficos de alguns testes correspondentes para mostrar a correlação entre eles de forma visual. A medição de cada eixo, variando de −15 À 15 é exibido individualmente ao longo do tempo (abscissa).

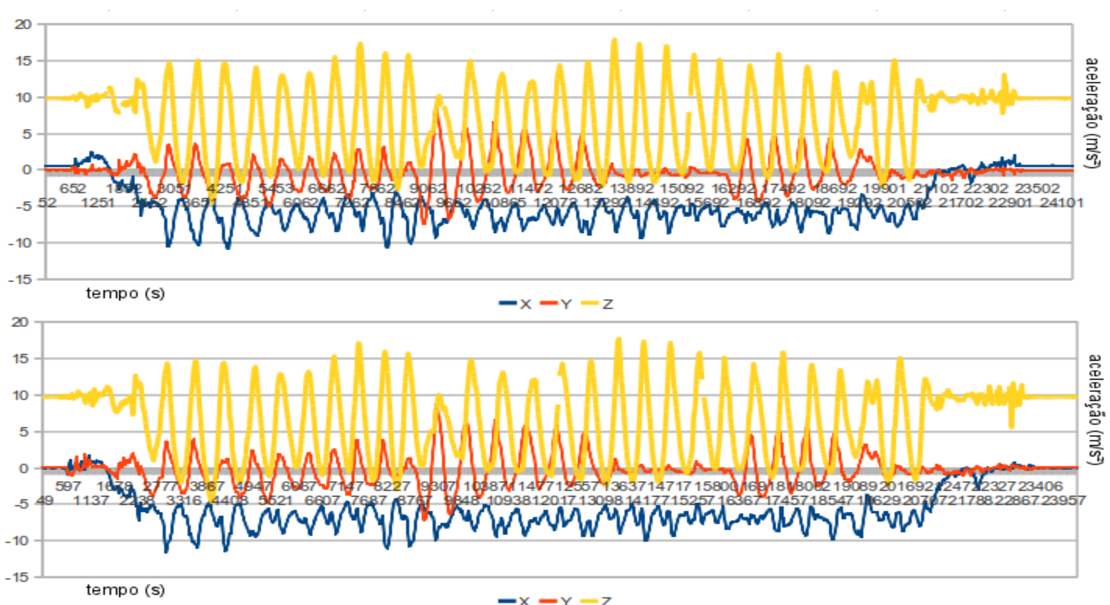

Figura A.1: Teste 1

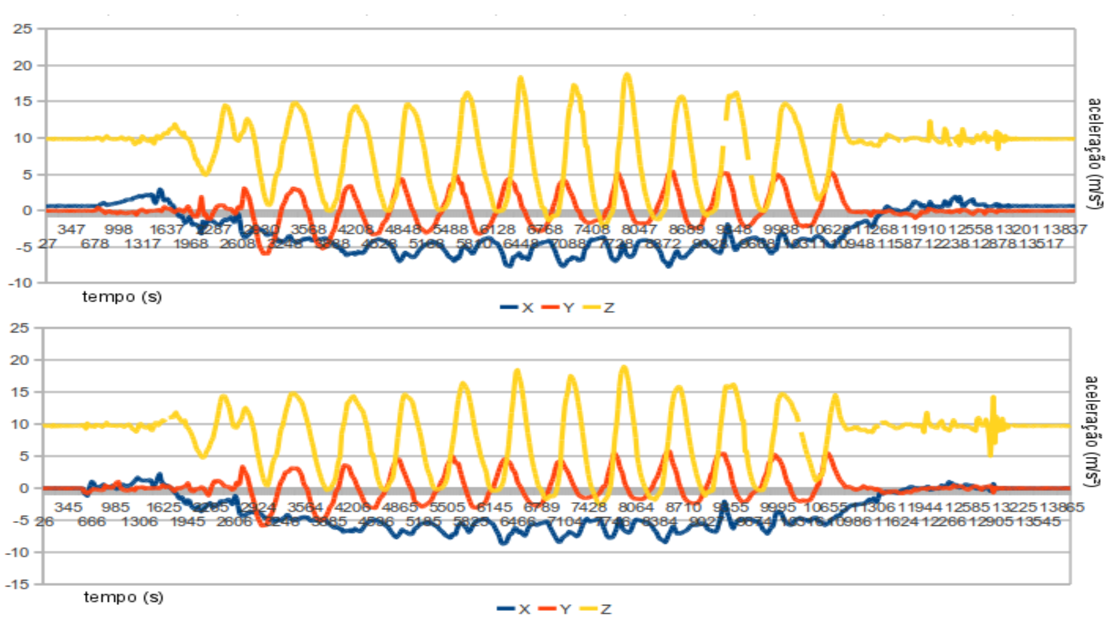

Figura A.2: Teste 2

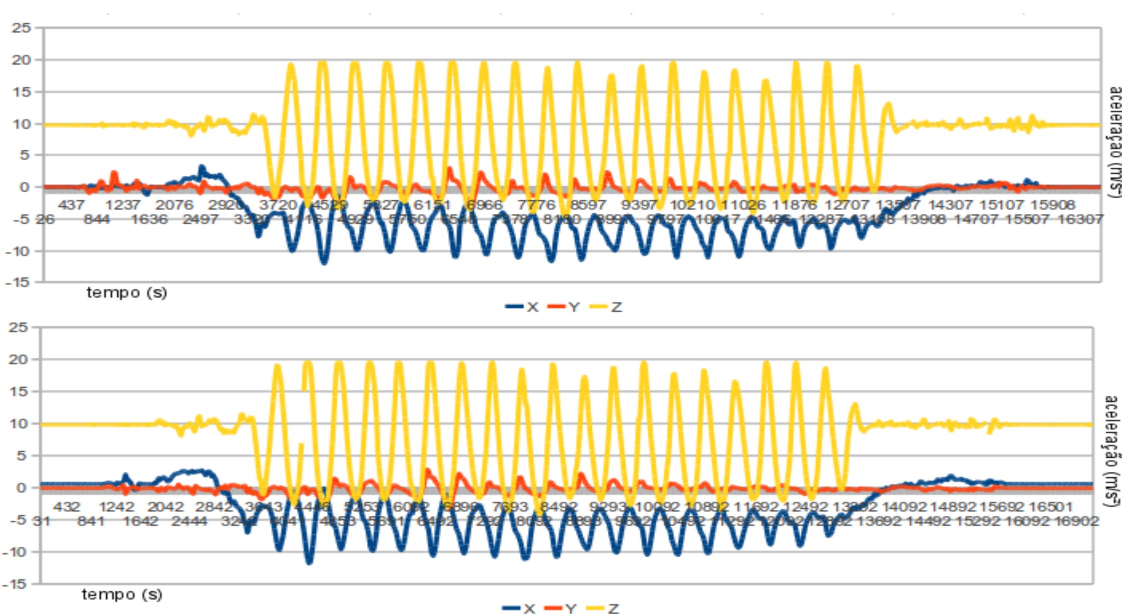

Figura A.3: Teste 3

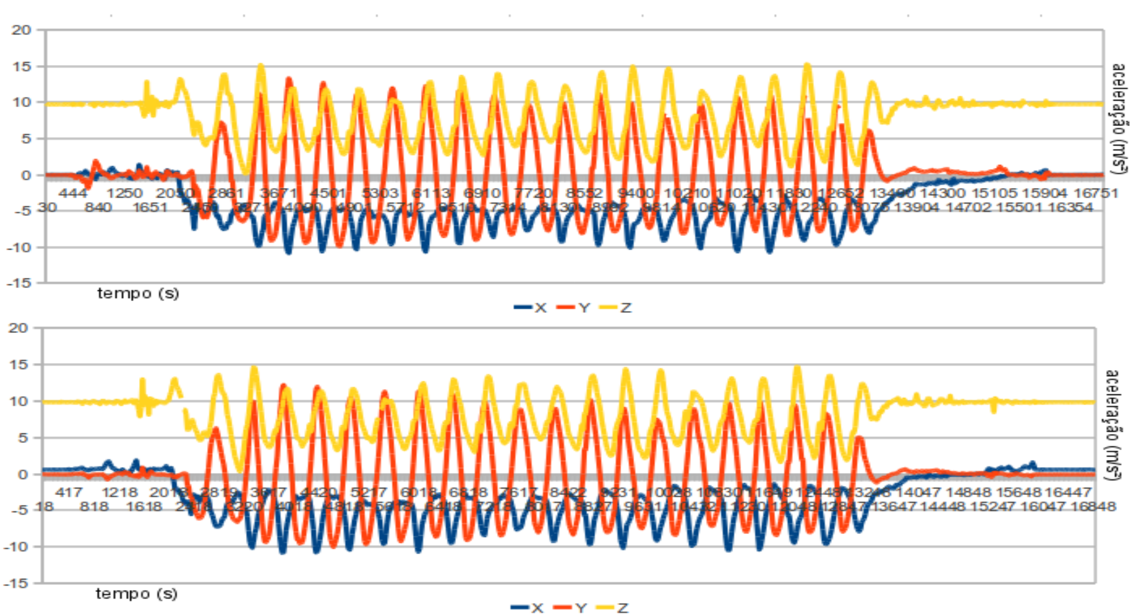

Figura A.4: Teste 4

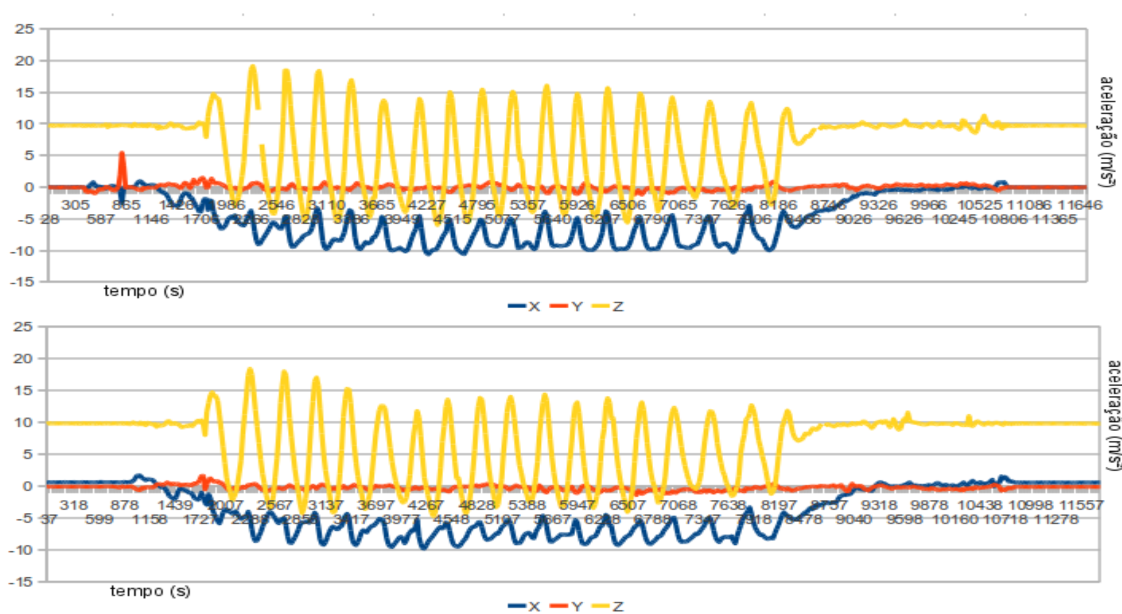

Figura A.5: Teste 5

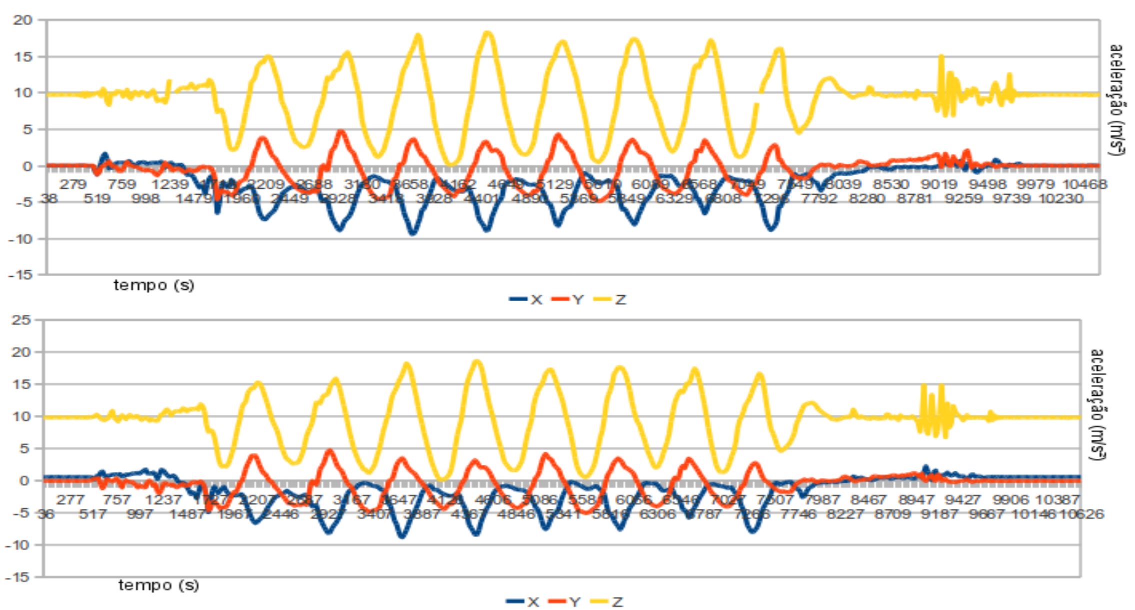

Figura A.6: Teste 6

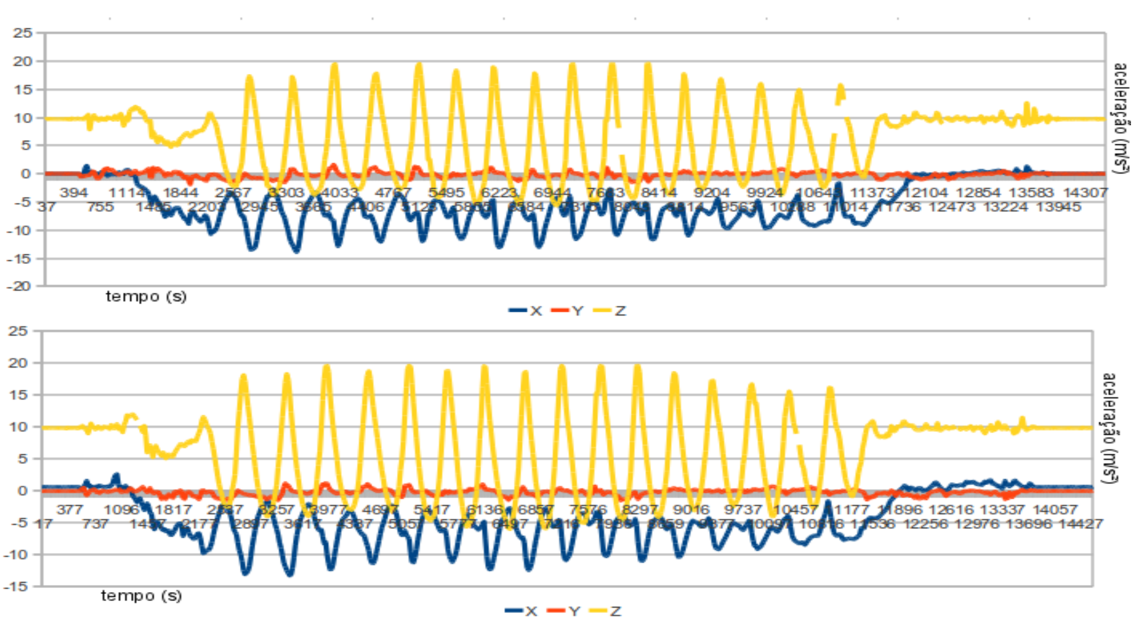

Figura A.7: Teste 7

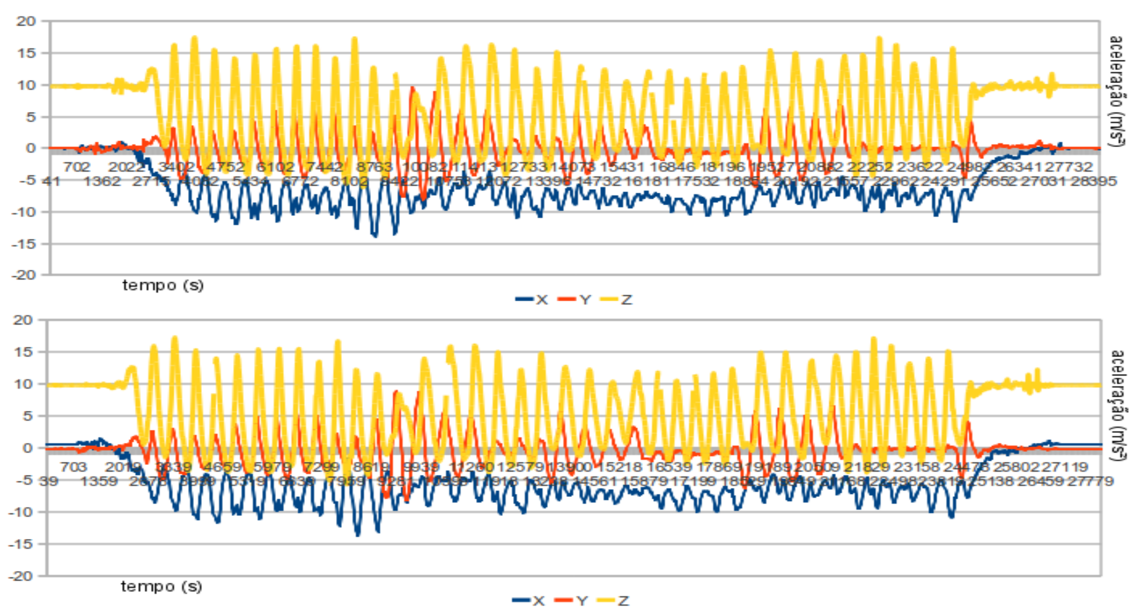

Figura A.8: Teste 8

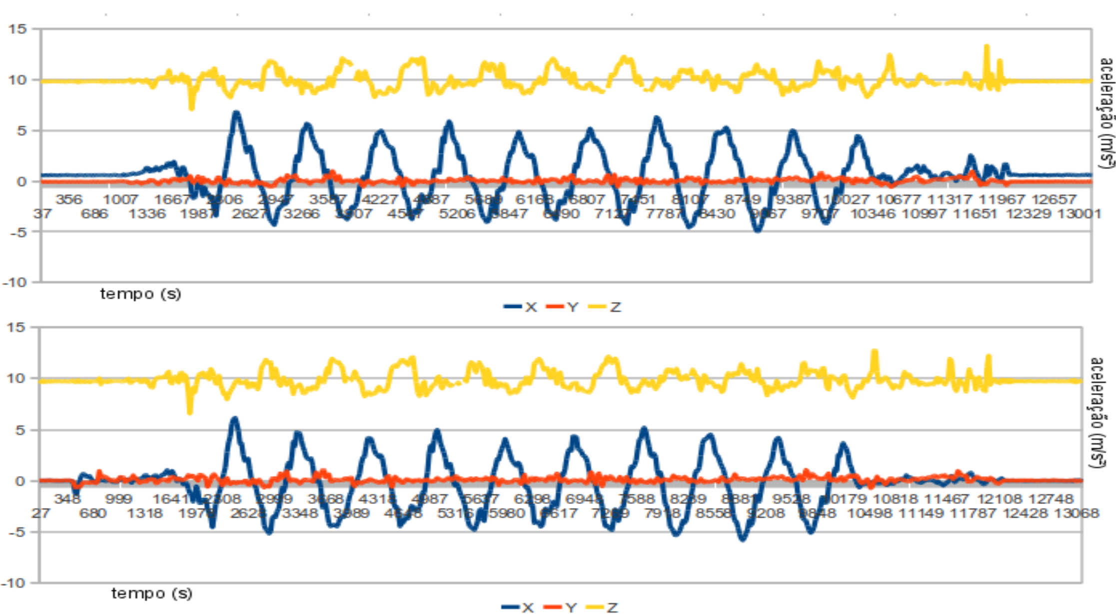

Figura A.9: Teste 9

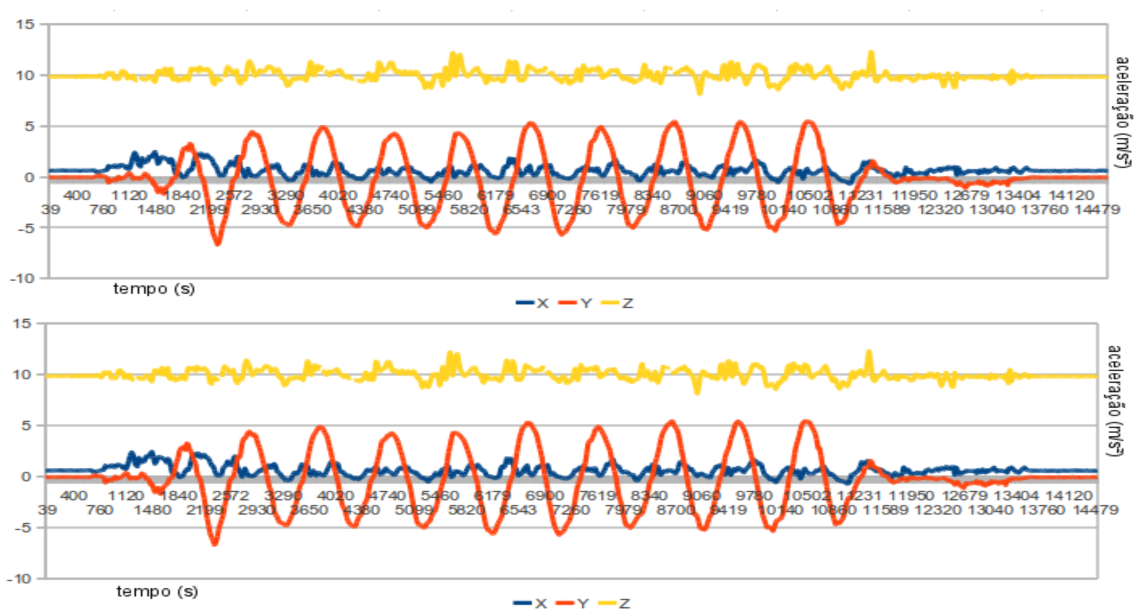

Figura A.10: Teste 11

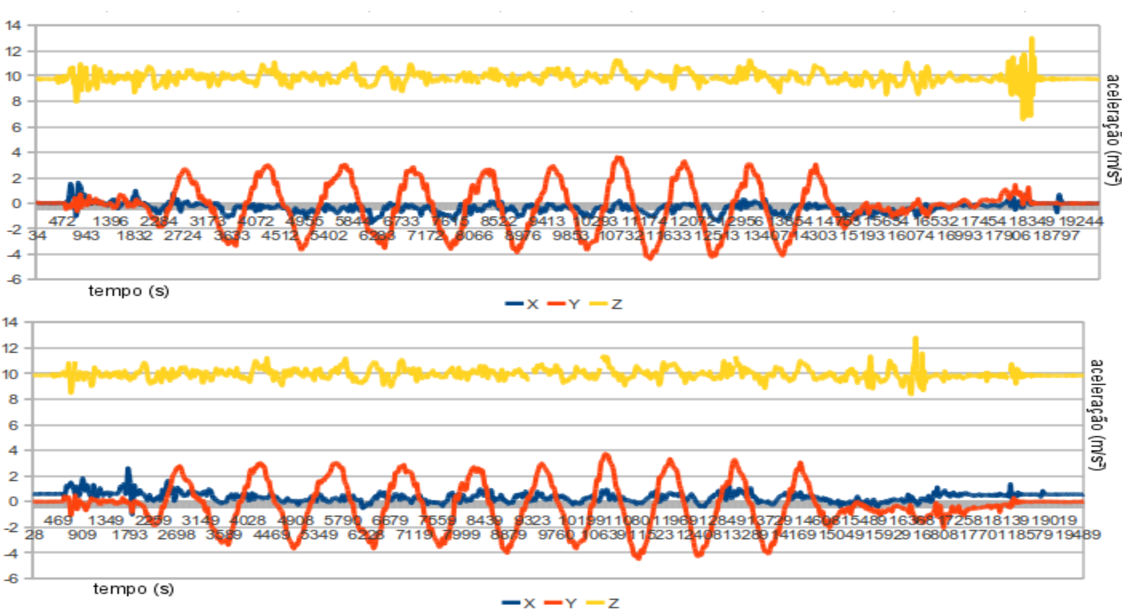

Figura A.11: Teste 12

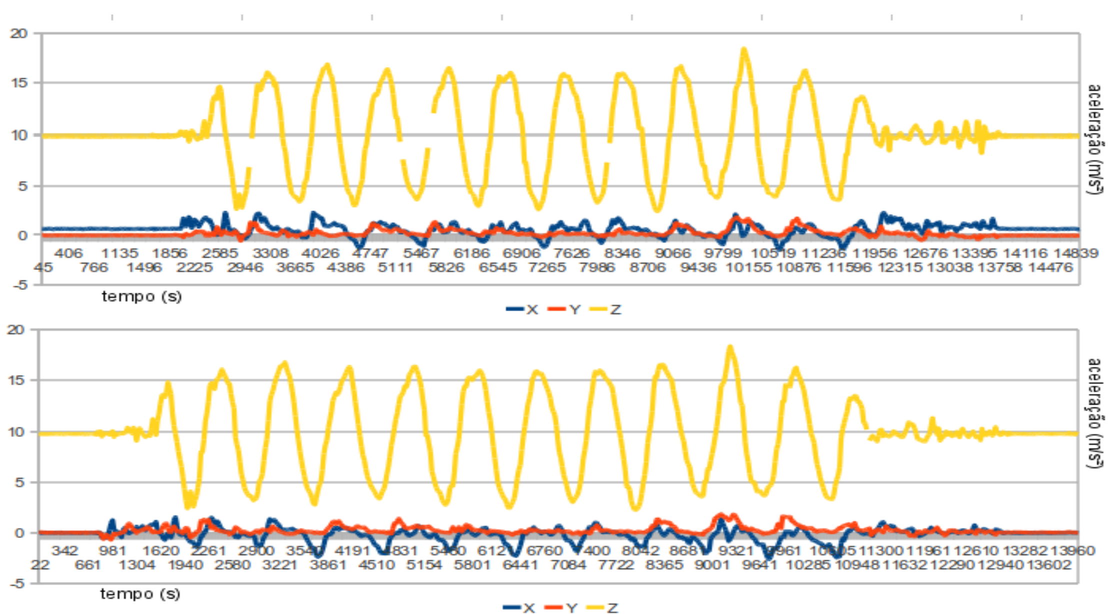

Figura A.12: Teste 13

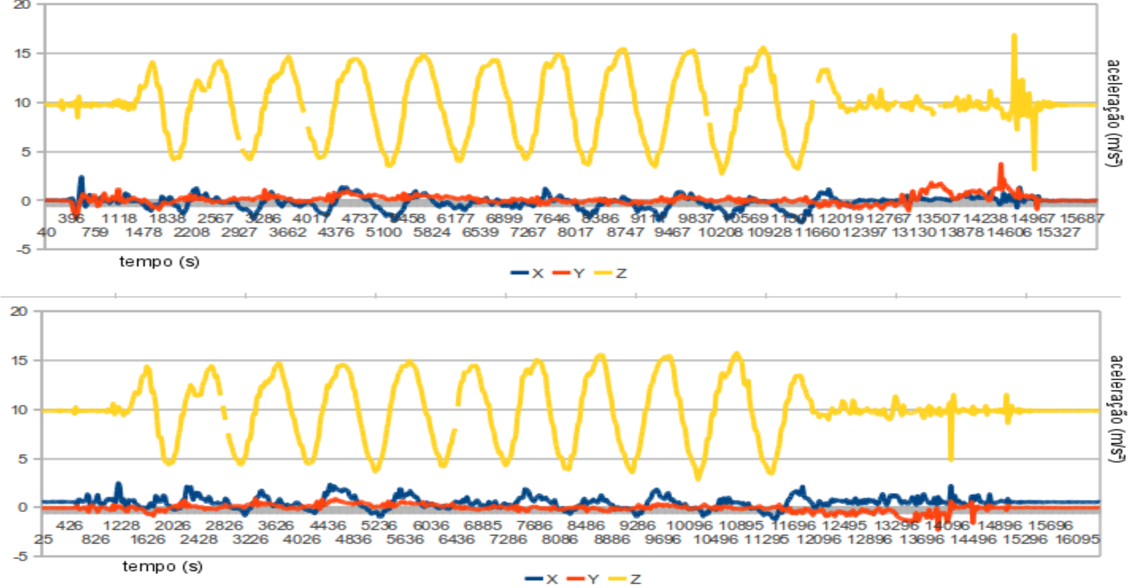

Figura A.13: Teste 14

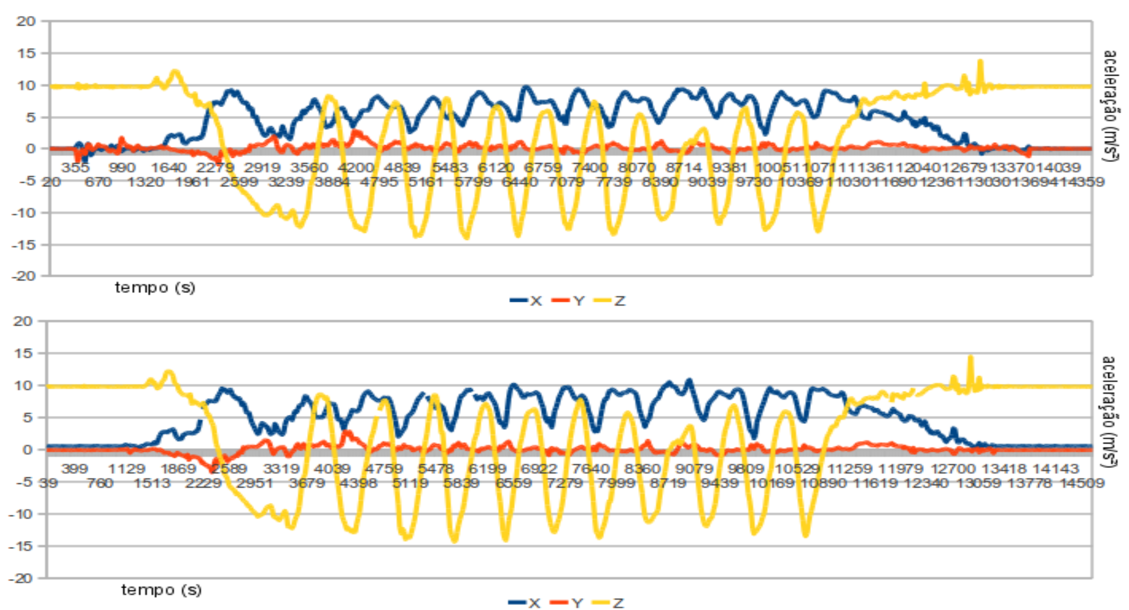

Figura A.14: Teste 15

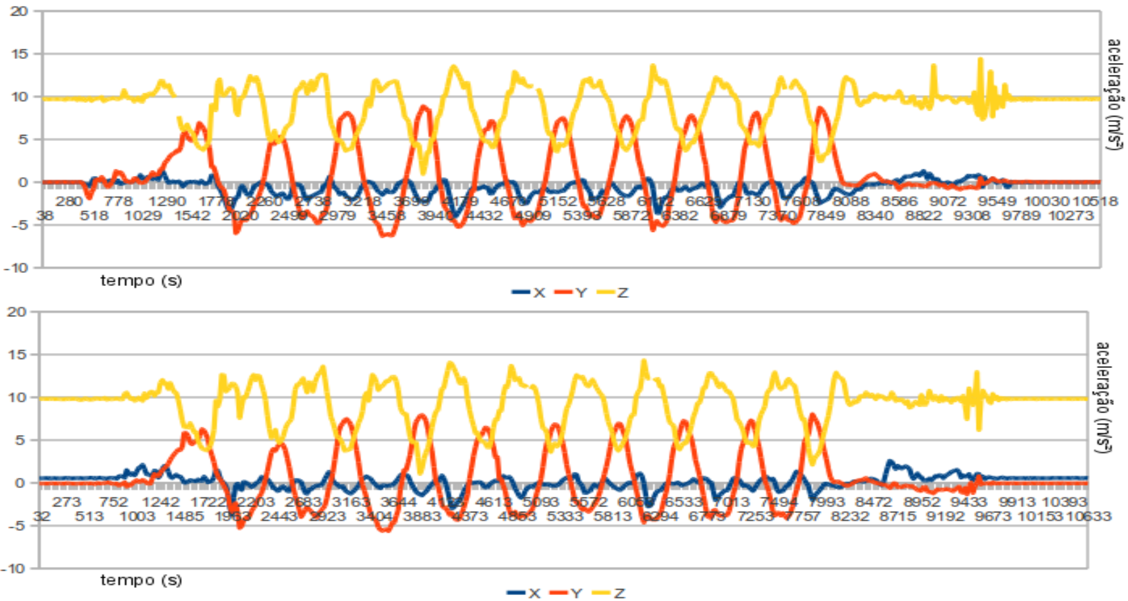

Figura A.15: Teste 16

Apêndice B

Código-fonte

```
package criptografia.keyreconciliation;

import criptografia.keyreconciliation.cascade.ClienteCascade;
import criptografia.keyreconciliation.cascade.ServidorCascade;
import util.Aux;
import util.BitArray;


public class Comum {

    protected BitArray chaveOriginal;
    protected BitArray chaves[];
    protected BitArray chaveReconciliada;
    protected Conexao conn;
    protected int tamanhoBlocoInicial = 10;
    protected int bitsRevelados = 0;
    protected int bitsCorrigidos = 0;
    protected float erroEstimado;

    public void aplicaPermutacao(int permutacao[]) {
        for (int i = 0; i < permutacao.length; i++) {
            for (int j = i + 1; j < permutacao.length; j++) {
                if (permutacao[j] < permutacao[i]) {
                    int aux = permutacao[i];
                    permutacao[i] = permutacao[j];
                    permutacao[j] = aux;
                    chaveOriginal.swap(i, j);
                }
            }
        }
    }

    public void aplicaPermutacao(int permutacao[], int passo) {
        for (int i = 0; i < permutacao.length; i++) {
            for (int j = i + 1; j < permutacao.length; j++) {
                if (permutacao[j] < permutacao[i]) {
                    int aux = permutacao[i];
```

```
                    permutacao[i] = permutacao[j];
                    permutacao[j] = aux;
                    chaves[passo].swap(i, j);
                }
            }
        }
    }

    public int[] geraPermutacao() {
        int permutacao[] = new int[chaveOriginal.size()];
        for (int i = 0; i < permutacao.length; i++) {
            permutacao[i] = Math.round(Math.round(Aux.getRandom().
                nextDouble() * (permutacao.length - 1) * 100));
        }
        return permutacao;
    }

    public String permutacao() {
        int permutacao[] = geraPermutacao();
        aplicaPermutacao(permutacao.clone());
        String permutacaoS = "";
        for (int p : permutacao) {
            permutacaoS += ":" + p;
        }
        return getConn().enviaRecebeResposta("permutacao" + permutacaoS);
    }

    public String permutacao(int passo) {
        int permutacao[] = geraPermutacao();
        aplicaPermutacao(permutacao.clone(), passo);
        String permutacaoS = "";
        for (int p : permutacao) {
            permutacaoS += ":" + p;
        }
        return getConn().enviaRecebeResposta("permutacao:" + passo +
            permutacaoS);
    }

    public String _permutacao(String msg) {
        String msgA[] = msg.split(":");
        int permutacao[] = new int[msgA.length - 1];
        for (int i = 1; i < msgA.length; i++) {
            permutacao[i - 1] = Integer.parseInt(msgA[i]);
        }
        aplicaPermutacao(permutacao);
        getConn().envia("_permutacao");
        return msg;
    }

    public String _permutacao(String msg, int passo) {
        String msgA[] = msg.split(":");
        int permutacao[] = new int[msgA.length - 2];
        for (int i = 2; i < msgA.length; i++) {
```

```
            permutacao[i - 2] = Integer.parseInt(msgA[i]);
        }
        aplicaPermutacao(permutacao, passo);
        getConn().envia("_permutacao");
        return msg;
    }

    public int binary(BitArray chave) {
        if (chave.size() == 1) {
            return chave.getPosicao(0);
        } else {
            int meio = chave.size() / 2;
            // Calcula e envia paridade da parte esquerda
            boolean paridadeCliente = chave.subList(0, meio).xor();
            getConn().envia("binaryparidade:" + paridadeCliente);
            bitsRevelados++;
            String msg = getConn().recebe("_binaryparidade");
            // Recebe resposta sobre a paridade esquerda
            boolean saoIguais = Boolean.valueOf(msg.split(":")[1]);
            if (saoIguais) { // O bit invertido esta na segunda metade
                return binary(chave.subList(meio, chave.size()));
            } else { // O bit invertido esta na primeira metade
                return binary(chave.subList(0, meio));
            }
        }
    }

    public String _binary(BitArray chave) {
        String msg = getConn().recebe();
        while (msg.split(":")[0].equals("binaryparidade")) {
            int meio = chave.size() / 2;
            BitArray chaveEsquerda = chave.subList(0, meio);
            BitArray chaveDireita = chave.subList(meio, chave.size());
            // Sempre recebe a paridade da esquerda
            boolean paridadeCliente = Boolean.valueOf(msg.split(":")[1]);
            if (chaveEsquerda.xor() == paridadeCliente) { // O bit
                invertido esta na segunda metade
                chave = chaveDireita;
                getConn().envia("_binaryparidade:" + "true");
            } else { // O bit invertido esta na primeira metade
                chave = chaveEsquerda;
                getConn().envia("_binaryparidade:" + "false");
            }
            if (chave.size() == 1) {
                /*
                 * O bit errado foi encontrado e sera invertido.
                 */
                return getConn().recebe();
            }
            msg = getConn().recebe();
        }
        return msg;
    }
```

```
    public boolean confirm(BitArray subChave, int idxInicial, int
        tamanhoBloco, int passo) {
        boolean xor = subChave.xor();
        String msg = getConn().enviaRecebeResposta("confirm:" + idxInicial
             + ":" + tamanhoBloco + ":" + xor + ":" + passo);
        bitsRevelados++;
        return Boolean.valueOf(msg.split(":")[1]);
    }

    public void _confirm(String msg) {
        String msgArray[] = msg.split(":");
        int idxInicial = Integer.parseInt(msgArray[1]);
        int tamanhoBloco = Integer.parseInt(msgArray[2]);
        boolean xor = Boolean.parseBoolean(msgArray[3]);
        int passo = Integer.parseInt(msgArray[4]);
        Boolean confirm;
        if (passo == 0) {
            confirm = (chaveOriginal.xor(idxInicial, tamanhoBloco) == xor)
                ;
        } else {
            confirm = (chaves[passo].xor(idxInicial, tamanhoBloco) == xor)
                ;
        }
        getConn().envia("_confirm:" + confirm);
    }
}
```

Code B.1: Comum

```
package criptografia.keyreconciliation.bbbss;

import criptografia.keyreconciliation.Comum;
import criptografia.keyreconciliation.Conexao;
import util.BitArray;

public class ClienteBBBSS extends Comum {

    private int tamanhoBlocoAtual;
    private int tamanhoChaveInicial;
    private int k = 10;

    public ClienteBBBSS(BitArray chaveOriginal, float erroEstimado, int
        porta) {
        this.chaveOriginal = chaveOriginal;
        this.erroEstimado = erroEstimado;
        conn = new Conexao(Conexao.TIPO_CLIENTE, porta);
    }

    public void reconcilia() {
        tamanhoBlocoInicial = (int) Math.ceil((float) 1 / erroEstimado);
        tamanhoBlocoAtual = tamanhoBlocoInicial;
        tamanhoChaveInicial = chaveOriginal.size();
        getConn().enviaRecebeResposta("inicio");
```

```
        for (int i = 0; i < k && chaveOriginal.size() >= tamanhoBlocoAtual
            ; i++) {
25          permutacao();
            int errosEncontrados = biconf();
            eliminaBitsRevelados();
            if (errosEncontrados > 0) {
                bitsCorrigidos += errosEncontrados;
30              float erroEstimadoAtual = ((float) tamanhoChaveInicial *
                    erroEstimado - (float) bitsCorrigidos) / (float)
                    chaveOriginal.size();
                tamanhoBlocoAtual = (int) Math.ceil((float) 1 /
                    erroEstimadoAtual);
                i = -1;
            }
        }
35      getConn().envia("fim");
        getConn().stop();
    }

    public int biconf() {
40      int errosEncontrados = 0;
        for (int b = 0; b < chaveOriginal.size() / tamanhoBlocoAtual; b++)
            {
            int idxInicial = b * tamanhoBlocoAtual;
            int idxFinal = idxInicial + tamanhoBlocoAtual > chaveOriginal.
                size() ? chaveOriginal.size() : idxInicial +
                tamanhoBlocoAtual;
            BitArray subChave = chaveOriginal.subList(idxInicial, idxFinal
                );
45          boolean result = confirm(subChave, idxInicial,
                tamanhoBlocoAtual, 0);
            if (!result) {
                getConn().envia("binary:" + idxInicial + ":" +
                    tamanhoBlocoAtual);
                int posicaoBitInvertido = binary(subChave);
                /*
50               * O bit errado foi encontrado e sera invertido.
                 */
                chaveOriginal.inverteBit(chaveOriginal.getPosicaoIndex(
                    posicaoBitInvertido));
                getConn().envia("fimbinary");
                errosEncontrados++;
55          }
        }
        return errosEncontrados;
    }

60  public void eliminaBitsRevelados() {
        int nroBlocos = (int) Math.ceil((float) chaveOriginal.size() / (
            float) tamanhoBlocoAtual);
        for (int b = nroBlocos; b > 0; b--) {
            int idxRemocao = (b * tamanhoBlocoAtual - 1) >= chaveOriginal.
                size() ? chaveOriginal.size() - 1 : (b * tamanhoBlocoAtual
```

```
                    - 1);
                chaveOriginal.remove(idxRemocao);
            }
            getConn().enviaRecebeResposta("remove:" + tamanhoBlocoAtual);
        }
}
```

Code B.2: ClienteBBBSS

```
package criptografia.keyreconciliation.bbbss;

import criptografia.keyreconciliation.Comum;
import criptografia.keyreconciliation.Conexao;
import util.BitArray;

public class ServidorBBBSS extends Comum {

    public ServidorBBBSS(BitArray chaveOriginal, int porta) {
        this.chaveOriginal = chaveOriginal;
        conn = new Conexao(Conexao.TIPO_SERVIDOR, porta);
    }

    public void _reconcilia() {
        getConn().recebeResponde("inicio");
        String msg = getConn().recebe();
        while (!msg.split(":")[0].equals("fim")) {
            if (msg.split(":")[0].equals("confirm")) {
                _confirm(msg);
                msg = getConn().recebe();
            } else if (msg.split(":")[0].equals("binary")) {
                String msgArray[] = msg.split(":");
                int idxInicial = Integer.parseInt(msgArray[1]);
                int tamanhoBloco = Integer.parseInt(msgArray[2]);
                BitArray chave = chaveOriginal.subList(idxInicial,
                    idxInicial + tamanhoBloco);
                msg = _binary(chave);
            } else if (msg.split(":")[0].equals("fimbinary")) {
                msg = getConn().recebe();
            } else if (msg.split(":")[0].equals("permutacao")) {
                _permutacao(msg);
                msg = getConn().recebe();
            } else if (msg.split(":")[0].equals("remove")) {
                _eliminaBitsRevelados(msg);
                msg = getConn().recebe();
            }
        }
        getConn().stop();
    }

    public void _eliminaBitsRevelados(String msg) {
        int tamanhoBloco = Integer.parseInt(msg.split(":")[1]);
        int nroBlocos = (int) Math.ceil((float) chaveOriginal.size() / (
            float) tamanhoBloco);
        for (int b = nroBlocos; b > 0; b--) {
```

```
                int idxRemocao = (b * tamanhoBloco - 1) >= chaveOriginal.size
                    () ? chaveOriginal.size() - 1 : (b * tamanhoBloco - 1);
45              chaveOriginal.remove(idxRemocao);
            }
            getConn().envia("_remove");
        }
    }
```

Code B.3: ServidorBBBSS

```
    package criptografia.keyreconciliation.cascade;

    import criptografia.keyreconciliation.Comum;
    import criptografia.keyreconciliation.Conexao;
5   import criptografia.keyreconciliation.PrivacyAmplification;
    import java.util.Random;
    import util.BitArray;

    public class ClienteCascade extends Comum {
10
        private int nroPassos = 4;

        public ClienteCascade(BitArray chaveOriginal, float erroEstimado, int
            porta) {
            this.chaveOriginal = chaveOriginal;
15          this.erroEstimado = erroEstimado;
            conn = new Conexao(Conexao.TIPO_CLIENTE, porta);
        }

        public void reconcilia() {
20          tamanhoBlocoInicial = (int) Math.ceil((float) 1 / erroEstimado);
            getConn().enviaRecebeResposta("inicio:" + nroPassos + ":" +
                tamanhoBlocoInicial);
            chaves = new BitArray[nroPassos + 1];
            chaves[0] = chaveOriginal;
            int passo = 1;
25          while (passo <= nroPassos) {
                chaves[passo] = chaves[passo - 1].clone();
                permutacao(passo);
                biconf(passo);
                passo++;
30          }
            getConn().envia("fim");
            chaveOriginal = chaves[1];
            privacyAmplification();
            getConn().stop();
35      }

        public void biconf(int passo) {
            int tamanhoBloco = tamanhoBlocoInicial * (int) Math.pow(2, passo -
                1);
            for (int i = 0; i < chaves[passo].size() / tamanhoBloco; i++) {
40              int idxInicial = i * tamanhoBloco;
```

```
                int idxFinal = idxInicial + tamanhoBloco > chaveOriginal.size
                    () ? chaveOriginal.size() : idxInicial + tamanhoBloco;
                BitArray subChave = chaves[passo].subList(idxInicial, idxFinal
                    );
                if (!confirm(subChave, idxInicial, tamanhoBloco, passo)) {
                    getConn().envia("binary:" + passo + ":" + idxInicial);
45                  int posicaoBitInvertido = binary(subChave);
                    for (int p = 1; p <= passo; p++) {
                        chaves[p].inverteBit(chaves[p].getPosicaoIndex(
                            posicaoBitInvertido));
                    }
                    bitsCorrigidos++;
50                  getConn().envia("fimbinary");
                    cascade(posicaoBitInvertido, passo - 1, passo);
                }
            }
        }
55
        public void cascade(int posicaoBitInvertido, int p, int nroPassos) {
            if (p < 1) {
                return;
            }
60          int idxBitInvertido = chaves[p].getPosicaoIndex(
                posicaoBitInvertido);
            int tamanhoBloco = tamanhoBlocoInicial * (int) Math.pow(2, p - 1);
            int idxInicial = (int) (Math.floor(idxBitInvertido / tamanhoBloco)
                ) * tamanhoBloco;
            int idxFinal = idxInicial + tamanhoBloco > chaves[p].size() ?
                chaves[p].size() : idxInicial + tamanhoBloco;
            BitArray subChave = chaves[p].subList(idxInicial, idxFinal);
65          if (!confirm(subChave, idxInicial, tamanhoBloco, p)) {
                getConn().envia("binary:" + p + ":" + idxInicial);
                int posicaoBitInvertido2 = binary(subChave);
                for (int i = 1; i <= nroPassos; i++) {
                    chaves[i].inverteBit(chaves[i].getPosicaoIndex(
                        posicaoBitInvertido2));
70              }
                getConn().envia("fimbinary");
                cascade(posicaoBitInvertido2, p - 1, nroPassos);
            }
        }
75
        public void privacyAmplification() {
            Random r = new Random();
            long seed = r.nextLong();
            if (bitsRevelados < chaveOriginal.size()) {
80              chaveOriginal = PrivacyAmplification.run(chaveOriginal,
                    chaveOriginal.size() - bitsRevelados, seed);
            }
            getConn().envia("privacy:" + seed + ":" + bitsRevelados);
        }
    }
```

Code B.4: ClienteCascade

```java
package criptografia.keyreconciliation.cascade;

import criptografia.keyreconciliation.Comum;
import criptografia.keyreconciliation.Conexao;
import criptografia.keyreconciliation.PrivacyAmplification;
import util.BitArray;

public class ServidorCascade extends Comum {
    private int nroPassos = 4;

    public ServidorCascade(BitArray chaveOriginal, int porta) {
        this.chaveOriginal = chaveOriginal;
        conn = new Conexao(Conexao.TIPO_SERVIDOR, porta);
    }

    public void _reconcilia() {
        String msg = getConn().recebeResponde("inicio");
        nroPassos = Integer.parseInt(msg.split(":")[1]);
        tamanhoBlocoInicial = Integer.parseInt(msg.split(":")[2]);
        chaves = new BitArray[nroPassos + 1];
        chaves[0] = chaveOriginal;

        msg = getConn().recebe();
        while (!msg.split(":")[0].equals("fim")) {
            if (msg.split(":")[0].equals("confirm")) {
                _confirm(msg);
            } else if (msg.split(":")[0].equals("permutacao")) {
                int passo = Integer.parseInt(msg.split(":")[1]);
                chaves[passo] = chaves[passo - 1].clone();
                _permutacao(msg, passo);
            } else if (msg.split(":")[0].equals("binary")) {
                String msgA[] = msg.split(":");
                int passo = Integer.parseInt(msgA[1]);
                int idxInicial = Integer.parseInt(msgA[2]);
                int tamanhoBloco = tamanhoBlocoInicial * (int) Math.pow(2,
                     passo - 1);
                int idxFinal = idxInicial + tamanhoBloco > chaves[passo].
                    size() ? chaves[passo].size() : idxInicial +
                    tamanhoBloco;
                BitArray subChave = chaves[passo].subList(idxInicial,
                    idxFinal);
                _binary(subChave);
            }
            msg = getConn().recebe();
        }
        while (!msg.equals("fim")) {
            msg = getConn().recebe();
        }
        chaveOriginal = chaves[1];

        _privacyAmplification();
        getConn().stop();
    }
```

```
    public String _privacyAmplification() {
        String msg = getConn().recebe();
        while (!msg.split(":")[0].equals("privacy")) {
            msg = getConn().recebe();
        }
        long seed = Long.parseLong(msg.split(":")[1]);
        int parametroBitsRevelados = Integer.parseInt(msg.split(":")[2]);
        if (parametroBitsRevelados < chaveOriginal.size()) {
            chaveOriginal = PrivacyAmplification.run(chaveOriginal,
                chaveOriginal.size() - parametroBitsRevelados, seed);
        }
        return msg;
    }

}
```

Code B.5: ServidorCascade

```
package criptografia.keyreconciliation;

import java.util.Random;
import util.BitArray;

public class PrivacyAmplification {

    public static BitArray run(BitArray key, int newKeySize, long seed) {
        Random r = new Random(seed);
        BitArray m[] = new BitArray[newKeySize];
        for (int i = 0; i < m.length; i++) {
            m[i] = BitArray.getBitArray();
            for (int j = 0; j < key.size(); j++) {
                m[i].add(r.nextBoolean());
            }
        }
        BitArray paKey = BitArray.getBitArray();
        for (int i = 0; i < m.length; i++) {
            paKey.add(m[i].and(key).xor());
        }
        return paKey;
    }
}
```

Code B.6: Privacy Amplification

Apêndice C

Taxa de erros

Tabela C.1: Taxa de erros (4 bits) - Motorola Xoom

	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	b19	b20	b21	b22	b23	b24	b25
a1	0.042	0.21	0.227	0.255	0.224	0.316	0.314	0.34	0.38	0.273	0.312	0.302	0.263	0.328	0.27	0.345	0.304	0.323	0.306	0.306	0.307	0.365	0.351	0.307	0.328
a2	0.217	0.039	0.243	0.25	0.299	0.443	0.326	0.36	0.323	0.356	0.416	0.305	0.351	0.309	0.261	0.388	0.325	0.371	0.342	0.318	0.321	0.345	0.401	0.406	0.335
a3	0.226	0.22	0.068	0.214	0.259	0.345	0.296	0.313	0.331	0.289	0.427	0.327	0.26	0.345	0.325	0.352	0.289	0.37	0.275	0.351	0.267	0.284	0.339	0.401	0.277
a4	0.256	0.224	0.332	0.078	0.21	0.358	0.341	0.344	0.339	0.384	0.404	0.343	0.311	0.314	0.304	0.347	0.422	0.337	0.309	0.34	0.357	0.329	0.341	0.422	0.345
a5	0.226	0.213	0.204	0.281	0.064	0.268	0.243	0.302	0.263	0.246	0.353	0.258	0.231	0.297	0.307	0.307	0.267	0.255	0.238	0.284	0.243	0.244	0.302	0.361	0.239
a6	0.299	0.324	0.295	0.359	0.253	0.079	0.376	0.284	0.288	0.285	0.27	0.253	0.289	0.353	0.362	0.302	0.268	0.304	0.26	0.352	0.265	0.291	0.312	0.302	0.251
a7	0.329	0.324	0.294	0.335	0.243	0.275	0.081	0.27	0.258	0.268	0.266	0.275	0.287	0.312	0.403	0.312	0.275	0.339	0.226	0.268	0.231	0.222	0.214	0.388	0.364
a8	0.325	0.395	0.325	0.309	0.302	0.299	0.277	0.068	0.253	0.31	0.278	0.285	0.285	0.338	0.328	0.331	0.297	0.309	0.289	0.315	0.301	0.306	0.301	0.304	0.354
a9	0.296	0.322	0.336	0.352	0.334	0.285	0.243	0.268	0.127	0.3	0.284	0.244	0.258	0.301	0.31	0.294	0.284	0.27	0.231	0.314	0.243	0.261	0.304	0.323	0.275
a10	0.26	0.28	0.292	0.321	0.217	0.275	0.253	0.248	0.27	0.062	0.234	0.258	0.236	0.301	0.258	0.363	0.243	0.35	0.227	0.291	0.277	0.251	0.321	0.226	0.214
a11	0.298	0.378	0.337	0.368	0.248	0.282	0.259	0.336	0.369	0.349	0.088	0.373	0.304	0.36	0.372	0.389	0.298	0.284	0.237	0.289	0.338	0.362	0.256	0.301	0.246
a12	0.297	0.316	0.317	0.399	0.275	0.272	0.38	0.331	0.332	0.324	0.327	0.098	0.253	0.338	0.364	0.394	0.25	0.282	0.296	0.299	0.287	0.351	0.346	0.34	0.29
a13	0.28	0.295	0.282	0.336	0.221	0.282	0.277	0.312	0.307	0.253	0.31	0.248	0.134	0.318	0.34	0.367	0.27	0.291	0.272	0.282	0.284	0.348	0.324	0.318	0.277
a14	0.295	0.311	0.294	0.297	0.29	0.375	0.292	0.355	0.319	0.312	0.33	0.307	0.306	0.119	0.272	0.326	0.326	0.279	0.284	0.346	0.306	0.314	0.337	0.392	0.306
a15	0.311	0.291	0.385	0.294	0.326	0.38	0.356	0.371	0.355	0.318	0.379	0.388	0.246	0.306	0.125	0.425	0.331	0.356	0.337	0.324	0.362	0.38	0.388	0.355	0.389
a16	0.349	0.364	0.324	0.37	0.294	0.297	0.323	0.294	0.306	0.323	0.328	0.394	0.342	0.346	0.315	0.122	0.278	0.282	0.243	0.348	0.268	0.26	0.329	0.317	0.25
a17	0.29	0.319	0.302	0.343	0.265	0.244	0.273	0.282	0.304	0.256	0.302	0.214	0.277	0.318	0.369	0.287	0.13	0.371	0.258	0.295	0.273	0.284	0.288	0.26	0.243
a18	0.288	0.312	0.288	0.345	0.248	0.308	0.348	0.339	0.26	0.286	0.363	0.261	0.267	0.258	0.352	0.358	0.268	0.112	0.236	0.304	0.214	0.346	0.316	0.329	0.229
a19	0.304	0.4	0.3	0.342	0.268	0.306	0.272	0.294	0.365	0.251	0.27	0.303	0.282	0.302	0.341	0.241	0.292	0.243	0.103	0.304	0.238	0.352	0.306	0.292	0.215
a20	0.278	0.28	0.292	0.349	0.27	0.314	0.333	0.347	0.352	0.275	0.29	0.323	0.275	0.329	0.372	0.378	0.295	0.316	0.303	0.076	0.287	0.324	0.328	0.289	0.323
a21	0.308	0.314	0.297	0.344	0.244	0.306	0.232	0.289	0.246	0.28	0.234	0.267	0.273	0.297	0.285	0.287	0.285	0.21	0.21	0.319	0.073	0.255	0.287	0.309	0.226
a22	0.292	0.382	0.413	0.331	0.255	0.303	0.239	0.299	0.267	0.26	0.354	0.273	0.303	0.318	0.356	0.284	0.292	0.34	0.248	0.27	0.26	0.061	0.294	0.392	0.383
a23	0.324	0.322	0.321	0.316	0.294	0.285	0.219	0.275	0.304	0.317	0.308	0.337	0.308	0.323	0.351	0.318	0.282	0.349	0.292	0.293	0.26	0.289	0.079	0.321	0.308
a24	0.306	0.338	0.33	0.397	0.282	0.301	0.398	0.35	0.365	0.227	0.304	0.327	0.292	0.359	0.284	0.343	0.248	0.313	0.27	0.301	0.326	0.413	0.336	0.112	0.234
a25	0.29	0.403	0.289	0.366	0.238	0.248	0.275	0.274	0.339	0.246	0.352	0.248	0.27	0.317	0.341	0.271	0.229	0.217	0.207	0.328	0.225	0.362	0.297	0.363	0.117

Tabela C.2: Taxa de erros (4 bits) - Samsung Galaxy

	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	b19	b20	b21	b22	b23	b24	b25
a1	0.054	0.299	0.278	0.275	0.304	0.278	0.262	0.285	0.28	0.284	0.263	0.398	0.321	0.369	0.344	0.282	0.357	0.34	0.354	0.314	0.284	0.275	0.368	0.316	0.295
a2	0.331	0.113	0.307	0.297	0.261	0.324	0.349	0.345	0.329	0.302	0.357	0.343	0.309	0.367	0.351	0.325	0.304	0.385	0.331	0.35	0.346	0.354	0.377	0.389	0.379
a3	0.294	0.328	0.095	0.313	0.282	0.29	0.324	0.338	0.289	0.306	0.312	0.345	0.351	0.332	0.358	0.321	0.287	0.312	0.316	0.326	0.323	0.322	0.389	0.338	0.324
a4	0.289	0.335	0.316	0.105	0.28	0.309	0.364	0.333	0.321	0.381	0.324	0.38	0.358	0.383	0.389	0.323	0.323	0.362	0.391	0.36	0.352	0.347	0.43	0.343	0.363
a5	0.302	0.251	0.292	0.275	0.035	0.292	0.326	0.336	0.29	0.284	0.311	0.35	0.313	0.361	0.331	0.323	0.258	0.338	0.367	0.33	0.314	0.333	0.389	0.346	0.323
a6	0.267	0.307	0.297	0.28	0.306	0.079	0.285	0.314	0.272	0.282	0.275	0.362	0.333	0.408	0.383	0.305	0.328	0.296	0.373	0.318	0.357	0.333	0.373	0.337	0.348
a7	0.325	0.361	0.347	0.295	0.326	0.308	0.115	0.349	0.303	0.323	0.298	0.407	0.339	0.488	0.394	0.345	0.356	0.413	0.378	0.358	0.342	0.312	0.425	0.319	0.321
a8	0.289	0.34	0.312	0.295	0.331	0.292	0.328	0.047	0.27	0.297	0.256	0.306	0.338	0.395	0.372	0.309	0.346	0.359	0.306	0.318	0.287	0.238	0.375	0.299	0.292
a9	0.388	0.323	0.284	0.318	0.292	0.344	0.324	0.272	0.071	0.328	0.323	0.312	0.318	0.342	0.371	0.29	0.322	0.364	0.36	0.312	0.309	0.273	0.362	0.314	0.272
a10	0.287	0.277	0.328	0.351	0.268	0.267	0.331	0.314	0.336	0.047	0.306	0.308	0.319	0.369	0.346	0.324	0.29	0.332	0.366	0.312	0.319	0.292	0.36	0.324	0.324
a11	0.29	0.362	0.363	0.297	0.326	0.27	0.357	0.284	0.329	0.309	0.085	0.324	0.35	0.336	0.373	0.318	0.355	0.289	0.353	0.323	0.309	0.29	0.366	0.307	0.311
a12	0.315	0.323	0.311	0.306	0.292	0.311	0.47	0.318	0.333	0.309	0.311	0.09	0.338	0.331	0.312	0.322	0.301	0.364	0.385	0.338	0.337	0.264	0.351	0.351	0.294
a13	0.322	0.301	0.371	0.332	0.308	0.341	0.426	0.323	0.309	0.299	0.325	0.302	0.139	0.365	0.338	0.294	0.324	0.332	0.316	0.377	0.335	0.318	0.392	0.352	0.318
a14	0.358	0.347	0.367	0.38	0.368	0.326	0.352	0.373	0.373	0.344	0.338	0.294	0.381	0.066	0.304	0.351	0.342	0.386	0.301	0.352	0.32	0.357	0.367	0.334	0.312
a15	0.325	0.326	0.362	0.33	0.309	0.345	0.333	0.365	0.332	0.311	0.387	0.37	0.37	0.328	0.102	0.309	0.325	0.301	0.355	0.36	0.34	0.329	0.367	0.326	0.392
a16	0.315	0.319	0.278	0.392	0.314	0.309	0.34	0.299	0.294	0.304	0.311	0.328	0.28	0.358	0.353	0.136	0.313	0.27	0.365	0.287	0.307	0.316	0.39	0.312	0.323
a17	0.328	0.277	0.336	0.341	0.272	0.307	0.352	0.341	0.319	0.275	0.35	0.322	0.346	0.373	0.345	0.355	0.1	0.372	0.345	0.352	0.329	0.368	0.398	0.341	0.362
a18	0.289	0.401	0.414	0.382	0.339	0.312	0.374	0.306	0.286	0.349	0.332	0.289	0.362	0.423	0.297	0.322	0.377	0.112	0.336	0.291	0.366	0.287	0.277	0.379	0.301
a19	0.339	0.345	0.344	0.346	0.329	0.335	0.343	0.312	0.314	0.357	0.347	0.301	0.324	0.349	0.357	0.293	0.375	0.348	0.069	0.339	0.268	0.282	0.333	0.293	0.272
a20	0.323	0.363	0.344	0.326	0.324	0.295	0.364	0.309	0.306	0.331	0.304	0.32	0.366	0.358	0.368	0.316	0.351	0.315	0.373	0.107	0.331	0.312	0.344	0.302	0.367
a21	0.287	0.369	0.343	0.352	0.319	0.341	0.307	0.287	0.312	0.336	0.295	0.318	0.346	0.292	0.366	0.29	0.345	0.316	0.275	0.357	0.061	0.28	0.366	0.311	0.273
a22	0.278	0.347	0.336	0.34	0.344	0.4	0.366	0.269	0.29	0.31	0.301	0.344	0.341	0.394	0.388	0.306	0.378	0.263	0.273	0.284	0.308	0.112	0.278	0.267	0.21
a23	0.402	0.4	0.395	0.406	0.376	0.383	0.434	0.316	0.37	0.383	0.328	0.302	0.386	0.363	0.358	0.334	0.405	0.272	0.333	0.359	0.382	0.281	0.096	0.293	0.313
a24	0.316	0.35	0.35	0.36	0.328	0.295	0.339	0.285	0.304	0.321	0.29	0.298	0.352	0.393	0.375	0.285	0.333	0.294	0.381	0.309	0.297	0.26	0.362	0.146	0.289
a25	0.307	0.35	0.327	0.36	0.324	0.336	0.317	0.275	0.304	0.314	0.301	0.301	0.328	0.311	0.389	0.309	0.357	0.29	0.27	0.311	0.267	0.193	0.351	0.299	0.061

Tabela C.3: Quantidade de bits errados após BBBSS (13%) - Motorola Xoom

	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	b19	b20	b21	b22	b23	b24	b25
a1	0	25	40	48	31	75	73	91	103	54	78	78	50	74	55	87	77	75	68	75	68	106	81	73	77
a2	38	0	45	40	74	128	81	94	74	89	109	75	96	67	48	100	77	109	89	67	78	83	107	118	80
a3	40	32	0	27	48	79	66	77	86	64	124	81	53	81	64	85	57	106	56	78	45	55	71	108	54
a4	48	25	80	0	24	83	91	89	78	107	99	83	69	76	71	83	125	88	68	90	87	80	90	117	89
a5	43	30	23	55	0	61	35	78	52	42	100	49	38	63	80	76	50	49	38	58	43	43	50	94	39
a6	66	76	59	86	51	0	100	56	60	72	56	36	59	96	94	75	61	59	45	81	45	64	76	64	50
a7	86	73	54	91	46	60	0	54	53	54	52	55	59	74	106	67	58	80	33	48	36	43	35	101	94
a8	76	95	75	73	68	58	54	0	44	68	48	58	55	76	85	82	71	65	56	75	70	73	78	56	96
a9	64	87	89	90	81	61	43	50	0	78	58	43	54	82	60	58	55	51	37	74	41	53	63	75	47
a10	55	61	56	76	31	67	46	39	50	0	45	46	42	55	63	90	39	95	27	58	62	50	75	34	34
a11	57	111	81	88	43	66	54	84	98	86	0	95	63	97	105	104	51	66	45	67	89	99	46	69	24
a12	76	73	80	97	59	52	98	86	83	79	80	0	40	79	88	97	47	53	66	67	57	86	89	87	64
a13	51	57	64	72	35	66	56	67	71	30	66	53	0	82	83	105	57	65	57	52	66	85	73	71	65
a14	65	84	68	65	70	102	64	97	83	69	85	71	72	0	59	79	73	61	55	80	71	70	82	109	70
a15	66	55	111	54	70	103	88	98	79	75	103	104	42	77	0	108	80	95	84	79	98	106	99	95	107
a16	93	99	73	94	65	64	80	66	69	75	80	109	85	86	77	0	55	65	38	79	51	50	70	76	40
a17	63	69	61	84	55	48	61	61	59	46	66	34	47	77	85	49	0	100	44	73	58	58	54	57	36
a18	52	81	60	78	50	71	91	74	52	60	96	46	55	46	87	81	53	0	40	67	30	94	63	89	29
a19	71	105	65	79	64	76	60	68	93	43	55	64	59	77	82	41	64	36	0	63	33	86	71	70	32
a20	55	49	67	94	57	83	83	75	103	56	72	76	60	78	95	104	56	73	77	0	60	76	76	62	76
a21	75	76	72	89	44	59	45	56	43	57	46	48	56	66	67	53	55	29	28	74	0	49	52	73	36
a22	58	96	115	81	45	61	32	68	50	46	93	64	66	63	89	56	72	73	40	51	51	0	68	114	107
a23	74	72	78	65	56	68	32	53	71	74	70	78	70	72	89	77	48	91	60	59	46	60	0	80	66
a24	67	77	83	100	57	66	119	92	98	36	78	72	68	92	68	76	42	75	50	63	84	113	77	0	38
a25	63	98	57	98	38	44	58	57	84	42	93	43	59	78	90	51	32	35	25	83	39	88	71	90	0

Tabela C.4: Quantidade de bits errados após BBBSS (13%) - Samsung Galaxy

	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	b19	b20	b21	b22	b23	b24	b25
a1	0	70	53	52	66	52	44	50	63	61	50	116	84	92	70	56	91	83	84	72	66	53	97	74	68
a2	77	0	69	58	54	79	81	82	79	66	94	88	73	93	83	72	69	96	87	92	87	90	97	109	95
a3	69	74	0	70	61	62	69	73	61	71	66	91	86	79	100	81	65	69	74	80	81	73	100	84	82
a4	66	78	77	0	57	72	84	83	69	97	89	105	99	94	95	79	79	99	99	85	84	86	131	78	102
a5	69	44	62	61	0	65	81	77	61	60	66	89	76	86	81	83	50	81	91	82	76	83	111	91	74
a6	55	61	58	63	69	0	61	77	52	49	50	87	75	109	100	60	73	65	86	86	90	76	91	88	87
a7	87	96	90	56	77	66	0	84	59	69	60	108	89	144	102	89	95	106	101	90	87	77	119	74	77
a8	68	82	78	59	81	65	74	0	54	66	53	73	88	107	92	69	84	94	61	72	65	34	99	72	73
a9	116	77	60	65	58	81	73	57	0	85	76	69	68	84	88	56	77	85	83	70	77	50	93	78	58
a10	61	63	77	86	39	54	83	76	81	0	63	75	72	101	89	72	63	81	92	73	70	70	87	82	74
a11	60	93	82	68	82	62	89	53	85	73	0	89	97	84	93	77	94	58	81	79	65	60	100	77	75
a12	74	80	64	64	64	69	135	70	83	64	69	0	86	84	80	72	62	87	108	82	78	45	92	89	62
a13	71	77	95	73	81	88	117	69	72	59	77	71	0	100	86	63	73	74	76	109	90	72	105	90	71
a14	83	81	109	94	84	86	84	104	91	85	84	63	103	0	62	89	89	99	71	87	81	90	95	72	73
a15	78	72	99	84	75	83	80	92	81	64	85	82	92	75	0	73	80	57	93	91	78	71	103	83	94
a16	70	86	55	109	71	66	87	68	69	63	70	73	60	96	91	0	81	47	88	61	78	74	108	77	71
a17	77	61	84	80	60	76	92	81	67	57	88	78	82	92	91	91	0	100	79	83	82	103	105	85	89
a18	63	103	118	91	81	76	98	62	63	89	70	62	89	121	72	69	100	0	83	62	86	66	50	96	64
a19	76	88	77	88	75	87	78	72	79	90	92	71	85	84	89	56	96	88	0	79	53	61	81	63	54
a20	66	99	80	80	79	64	90	73	70	90	70	74	92	90	97	74	99	79	101	0	78	84	83	68	99
a21	60	93	83	85	71	74	70	57	69	78	55	71	82	74	99	65	82	69	56	102	0	63	98	77	61
a22	59	85	78	83	84	104	98	53	64	70	69	84	93	102	116	74	90	53	68	56	71	0	53	51	29
a23	106	104	107	109	103	99	114	71	103	103	78	65	92	92	93	76	95	56	83	99	103	59	0	66	64
a24	72	95	83	88	79	60	76	63	73	72	64	66	91	107	98	59	74	57	109	68	57	51	90	0	66
a25	64	88	84	91	73	90	71	46	68	78	71	62	77	65	100	71	88	63	47	66	48	26	91	72	0

Tabela C.5: Quantidade de bits errados após Cascade (15%) - Motorola Xoom

	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	b19	b20	b21	b22	b23	b24	b25
a1	0	4	6	27	18	54	56	21	98	38	60	41	32	51	30	67	46	51	67	54	67	82	83	62	73
a2	5	0	29	9	43	123	73	70	71	72	121	53	67	55	34	98	56	100	81	50	1	74	118	105	84
a3	14	8	0	8	5	89	51	60	72	29	4	70	29	79	65	91	13	16	26	65	21	50	69	116	47
a4	27	19	73	0	6	74	55	81	78	10	3	82	57	47	62	71	134	69	54	80	15	79	63	124	75
a5	20	4	9	14	0	40	48	58	31	5	5	15	5	50	55	55	40	6	9	53	13	31	37	9	15
a6	56	58	37	88	25	0	86	50	41	49	41	17	5	76	94	49	38	2	26	92	23	46	53	29	25
a7	80	2	55	74	27	37	0	2	0	2	38	20	43	53	87	62	51	74	5	29	25	20	25	100	95
a8	61	103	65	57	57	54	46	0	12	66	14	9	0	58	69	57	27	65	37	60	53	2	53	50	81
a9	42	56	8	17	10	7	9	42	0	42	36	7	24	37	9	55	41	41	7	45	28	18	42	65	47
a10	16	29	52	43	13	32	34	34	43	0	21	38	21	48	23	94	24	13	10	40	46	8	78	6	8
a11	56	104	72	93	29	32	40	77	89	70	0	85	37	97	101	17	49	0	12	43	16	90	31	54	6
a12	42	11	50	98	40	28	16	1	63	25	68	0	2	86	81	5	31	19	46	39	30	9	80	3	49
a13	44	47	40	71	15	34	40	46	44	27	66	28	0	66	67	93	16	39	28	47	38	90	77	54	36
a14	8	62	54	47	54	98	39	85	50	66	64	53	37	0	30	19	76	13	37	70	52	65	72	111	51
a15	54	12	99	47	77	100	14	100	92	20	93	99	25	47	0	121	70	93	5	60	93	95	94	91	107
a16	56	82	70	91	56	57	51	16	49	72	55	84	88	88	60	0	9	38	10	64	43	32	73	63	17
a17	8	43	50	67	28	12	41	34	34	34	51	8	19	65	11	31	78	85	22	51	39	38	48	35	27
a18	38	50	4	75	16	53	90	82	29	45	86	32	45	18	11	4	3	0	12	60	9	88	59	64	16
a19	42	105	58	65	18	53	39	48	81	14	15	7	42	23	79	3	45	27	0	14	19	84	57	53	8
a20	44	39	14	74	5	62	48	89	79	37	47	52	36	19	14	85	44	60	45	0	8	72	2	60	71
a21	1	55	40	89	10	42	8	46	18	45	13	33	49	61	44	44	54	11	6	70	0	30	53	58	18
a22	36	17	128	68	27	36	14	46	13	30	3	28	54	50	87	38	11	76	24	33	33	0	54	111	104
a23	2	37	57	66	53	49	9	6	56	47	48	72	43	4	89	74	44	84	56	44	29	36	0	62	49
a24	54	53	73	101	33	7	109	63	73	13	52	76	59	80	51	9	25	67	48	44	64	14	13	0	15
a25	51	107	40	79	17	16	33	32	76	14	80	26	40	70	84	23	18	12	4	53	6	85	59	19	0

Tabela C.6: Quantidade de bits errados após Cascade (15%) - Samsung Galaxy

	b1	b2	b3	b4	b5	b6	b7	b8	b9	b10	b11	b12	b13	b14	b15	b16	b17	b18	b19	b20	b21	b22	b23	b24	b25
a1	0	60	36	39	50	40	39	36	43	52	40	107	0	78	46	37	82	78	93	68	49	36	51	53	56
a2	63	0	58	45	32	1	74	2	66	48	84	60	18	13	89	48	47	95	82	16	84	86	37	104	6
a3	43	19	0	10	42	37	11	73	55	13	25	77	76	56	87	61	7	71	66	42	59	65	106	84	58
a4	46	83	56	0	47	67	5	76	66	78	68	6	97	94	112	9	69	33	106	92	83	3	125	83	87
a5	56	5	43	50	0	52	67	59	51	45	37	10	59	96	75	72	19	58	82	57	0	76	0	90	74
a6	29	44	48	30	7	0	9	63	30	34	27	18	52	114	114	57	71	40	83	70	87	63	93	78	71
a7	66	69	67	41	65	47	0	89	55	62	37	117	1	148	2	8	17	116	100	63	86	65	123	58	11
a8	52	27	61	59	71	59	54	0	31	39	28	59	66	114	84	42	78	76	49	62	42	38	98	56	42
a9	103	65	53	51	53	76	55	43	0	63	75	56	43	31	82	37	51	102	70	20	35	31	94	56	29
a10	2	40	59	89	4	39	75	69	75	0	62	52	71	86	83	74	61	62	91	43	63	36	86	80	65
a11	54	90	5	64	66	42	81	39	71	10	0	2	66	63	97	57	81	47	70	68	54	33	93	56	7
a12	53	65	69	59	43	3	141	69	61	17	47	0	55	67	62	45	48	90	105	82	68	34	4	12	14
a13	8	45	90	13	57	67	7	8	44	57	62	63	0	93	57	47	63	56	70	4	70	11	84	82	74
a14	34	14	86	95	95	1	88	92	9	74	55	56	108	0	58	71	66	106	52	12	70	5	20	78	62
a15	67	72	83	8	5	61	64	89	81	10	8	9	13	66	0	54	77	2	76	82	82	78	7	65	10
a16	53	66	43	89	63	22	70	15	33	54	7	64	44	78	90	0	63	35	70	16	57	70	115	7	48
a17	74	16	2	78	35	57	93	79	49	50	92	62	61	82	72	77	0	99	81	80	54	99	116	16	77
a18	50	117	125	25	6	57	100	15	27	84	20	41	4	123	52	67	99	0	70	40	91	53	35	107	38
a19	63	83	80	52	60	13	58	71	53	87	82	56	65	85	93	47	88	84	0	81	38	51	69	46	34
a20	54	92	75	71	72	12	77	62	58	68	64	24	93	9	87	63	79	68	98	0	53	63	61	56	90
a21	49	84	1	75	53	89	11	55	27	67	47	66	73	6	8	33	11	53	27	76	0	34	94	13	31
a22	30	71	82	78	12	112	83	28	45	61	55	25	85	105	114	65	101	31	14	37	51	117	49	41	7
a23	2	107	112	118	100	93	10	48	83	38	70	45	99	7	77	62	120	29	72	6	108	51	0	51	59
a24	61	67	80	1	70	43	70	41	62	69	31	54	64	108	107	50	62	57	82	54	57	26	96	0	38
a25	41	75	75	87	61	13	12	46	55	64	57	62	16	50	111	58	3	5	36	50	47	2	70	55	0