UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação

departamento de sistemas de computação

SSC 747 - Engenharia de Segurança
Prof[A] Kalinka R. L. J. Castelo Branco

Projeto 1 - Smart Home do João

Elisa Jorge Marcatto	7961965
Victor Marcelino Nunes	8622381

São Carlos

2016

1 Introdução

Diante do cenário em que a Internet das Coisas (*Internet of Things, IoT*) vem ganhando um espaço cada vez maior no cotidiano das pessoas, a automatização das casas pode ser vista como mais um exemplo desse novo modo de vida instaurado pelo avanço dessa tecnologia. Uma *smart home* pode ser descrita como uma residência que possui sistemas de refrigeração, iluminação, entretenimento e segurança que se comunicam e que podem ser controlados de acordo com a preferência do morador, o que pode ser efetuado até remotamente. Entretanto, a construção de uma *smart home* exige, principalmente, a implementação de uma rede segura e robusta para que os dados trocados entre os sistemas e seu controle estejam disponíveis a apenas pessoas autorizadas. Caso a rede não seja implementada de forma adequada, a *smart home* estará vulnerável a diversos ataques, afetando diretamente a segurança e conforto de seus moradores.

Este projeto tem por objetivo propor um algoritmo de criptografia e de esteganografia que poderá ser usado em uma *smart home* para garantir a segurança das informações trocadas entre os sistemas que nela estão conectados. Além disso, são propostas medidas complementares ao uso do algoritmo construído para que sejam garantidas integridade, disponibilidade, confidencialidade e autenticidade na rede utilizada, proporcionando uma maior segurança para o morador da *smart home*.

2 Primeira Tarefa

2.1 Algoritmos

A implementação dos módulos de criptografia e esteganografia foi dividida em classes, e estas divididas em pacotes, com o intuito de se ter uma melhor organização do projeto.

Nas subseções a seguir, a metodologia de cada classe será explicada para que se possa demonstrar o processo utilizado para transformar uma mensagem plana em uma imagem e o processo de recuperação da mensagem plana a partir de uma imagem.

2.1.1 Pacote *toolbox*

O pacote *toolbox* contém quatro classes auxiliares que possuem funções estáticas, utilizadas frequentemente pelas outras classes.

A classe *Configurations* contém as configurações básicas que devem ser predeterminadas no início do programa. Uma instância de *Configurations* é criada na classe *Start.java*, em seu construtor, pois a mesma integra todas as outras classes com a interface gráfica. Dessa forma, é possível passar essa instância para todas as classes que necessitam desses parâmetros.

Essa classe foi criada para facilitar o processo de mudança de alguns parâmetros, como o tamanho da imagem a ser utilizada, o nome de saída da imagem gerada e o nome de saída do texto gerado. Como esses parâmetros são utilizados por mais de uma classe, a mudança manual deles poderia fazer com que um erro acidental acontecesse.

A classe *Converters* contém todos os conversores utilizados pelas outras classes. Ela foi criada pois alguns métodos de conversão dependia de muitas variáveis auxiliares, deixando o código "poluído".

A classe *Maths* contém todas as funções matemáticas utilizadas para gerar uma chave, como o algoritmo Euclides Estendido, o calculo do MDC de dois números e a verificação de um número para saber se o mesmo é um número primo.

Por fim, a classe *Operations* contém métodos que auxiliam em duas operações: a de obter os dois elementos de uma chave (será melhor explicado na subseção 2.1.2) e a operação de se adicionar num vetor de inteiros.

2.1.2 Geração de Chaves - *Keys.java*

A classe *Keys* contém como atributos os quatro elementos necessários para se ter um par de chave pública e privada: (E,N) e (D,N). O número N foi repetido para se ter uma melhor visualização das chaves.

Para obter esses elementos, foi utilizado o método RSA contido nos slides da quarta aula do *Crptus*, "Criptografia de chave pública".

Primeiramente, foram gerados, aleatoriamente, dois números primos, P e Q, entre 1073741824 e 2147483648. Poderia ser utilizado a classe *BigInteger* para gerar números de até 1024 bits, como indicado no slide, mas, por decisão de projeto, foi preferível criar as funções matemáticas (contidas em *Maths.java*) para se ter um maior contato e controle no processo de se criar as chaves, ao invés de usar as funções prontas da classe *BigInteger*. Como as operações com *BigInteger* não são tão triviais, foi decidido trabalhar com variáveis do tipo *long*. A perda de segurança gerada por essa diminuição da chave foi compensada pelo fato de se utilizar esteganografia logo após a criptografia.

Em seguida, foram gerados os números N e Z, tal que

N = P*Q

Z = (P-1)*(Q-1)

O próximo passo foi gerar um número E tal que E $<$ N e E e Z sejam primos entre si. Para saber se dois números são primos entre si basta calcular o MDC dos dois números e verificar se o resultado foi 1.

Com isso, a chave pública pode ser criada, sendo ela o par de números E e N.

Para se calcular o parâmetro D da chave privada, foi utilizado o resultado do algoritmo do Euclides Estendido entre os números E e Z.

Com isso, a chave privada pode ser criada, sendo ela o par de números D e N.

Por fim, para facilitar a utilização do módulo para o usuário, quando as chaves são geradas as mesmas tem seus parâmetros concatenados, com o caractere X dividindo-os. Por exemplo, a chave (5, 60) será mostrada para o usuário como 5X60. Essa concatenação é feita na classe *Start.java*, logo após o usuário apertar o botão "Gerar Chaves!". Como explicado na seção 2.1.1, um método da classe *Operations* realiza a obtenção dos parâmetros da chave quando a chave concatenada é enviada para ele.

2.1.3 Criptografia - *CriptIn.java*

Depois de se ter as chaves pública e privada em mãos, o usuário poderá ir para a aba Criptografar para criptografar e esconder um arquivo *.txt* numa imagem.

Será utilizado o termo *texto plano* para indicar o arquivo texto de entrada e o termo *cifra* para indicar a saída deste primeiro módulo.

O primeiro passo é enviar o texto plano e a chave pública para a classe *CiptIn*, responsável por encriptar o texto plano com base no algoritmo RSA.

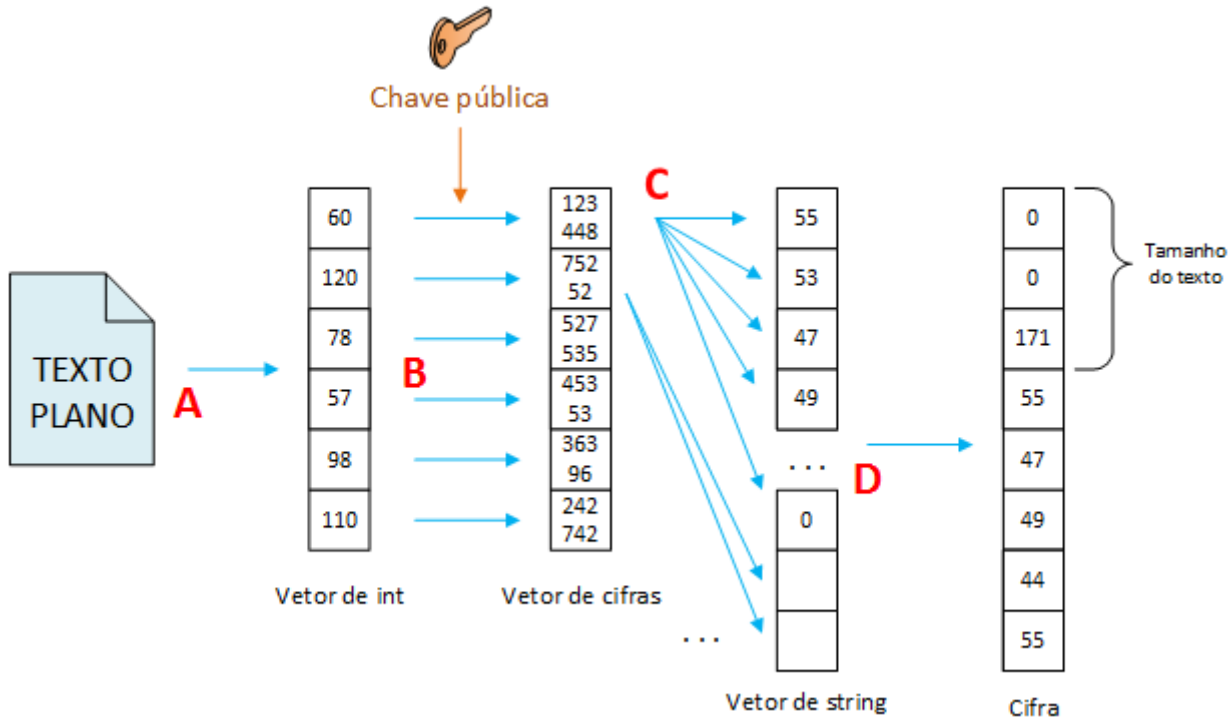

Figura 1: Processo utilizado para transformar o texto plano numa cifra

O ponto A presente na Figura 1 representa a primeira parte do módulo, onde o texto plano é convertido em um vetor de inteiros, onde cada elemento desse vetor representa o número (decimal) de um caractere do texto plano.

O ponto B presente na Figura 1 se refere à operação de cifragem de cada elemento do vetor de inteiros. A operação feita segue o algoritmo RSA, sendo ela:

$C = M^E mod N$

Onde C é o caractere cifrado, M é o caractere do vetor de inteiros e E e N são os parâmetros da chave pública.

Por decisão de projeto, esse cálculo foi feito utilizando a classe *BigInteger*, pois fazê-lo utilizando as variáveis como *long* não seria uma tarefa eficiente. Para tal, foi criado na classe *Converters* (subseção 2.1.1) métodos para converter *long*, *inteiro* e *string* para o tipo *BigInteger*. Após o cálculo, o número resultante é convertido para *string*.

A explicação a seguir se refere ao ponto C presente na Figura 1.

Como o número resultante é muito grande, e como ele deve ser escondido em uma imagem, o mesmo foi dividido em 19 números, cada um contendo 8 bits do número original. Os 19 números de cada caractere cifrado são armazenados num vetor de *string* (19 vezes maior que o vetor de inteiros), onde cada elemento desse vetor representa um número diferente.

Para manter um padrão de 19 números por caractere cifrado, quando um número pode ser divididos em N números, sendo N $<$ 19, as posições restantes são completadas com 0, até se ter as 19 posições por caractere cifrado.

O ponto D presente na Figura 1 representa o processo de armazenar o tamanho do vetor de *string* nas três primeiras posições do mesmo. Para isso é utilizado um método contido em *Converters.java*, que converte um inteiro em 3 outros inteiros, onde cada um possui no máximo 8 bits. Dessa forma, é possível trabalhar com um texto de até $2^{24}/19$ caracteres.

Portanto, o vetor de *string* representa a cifra, que será enviada para o módulo de esteganografia.

2.1.4 Esteganografia - *StegIn.java*

O módulo de esteganografia recebe como entrada somente o vetor de *string* gerado no módulo de criptografia.

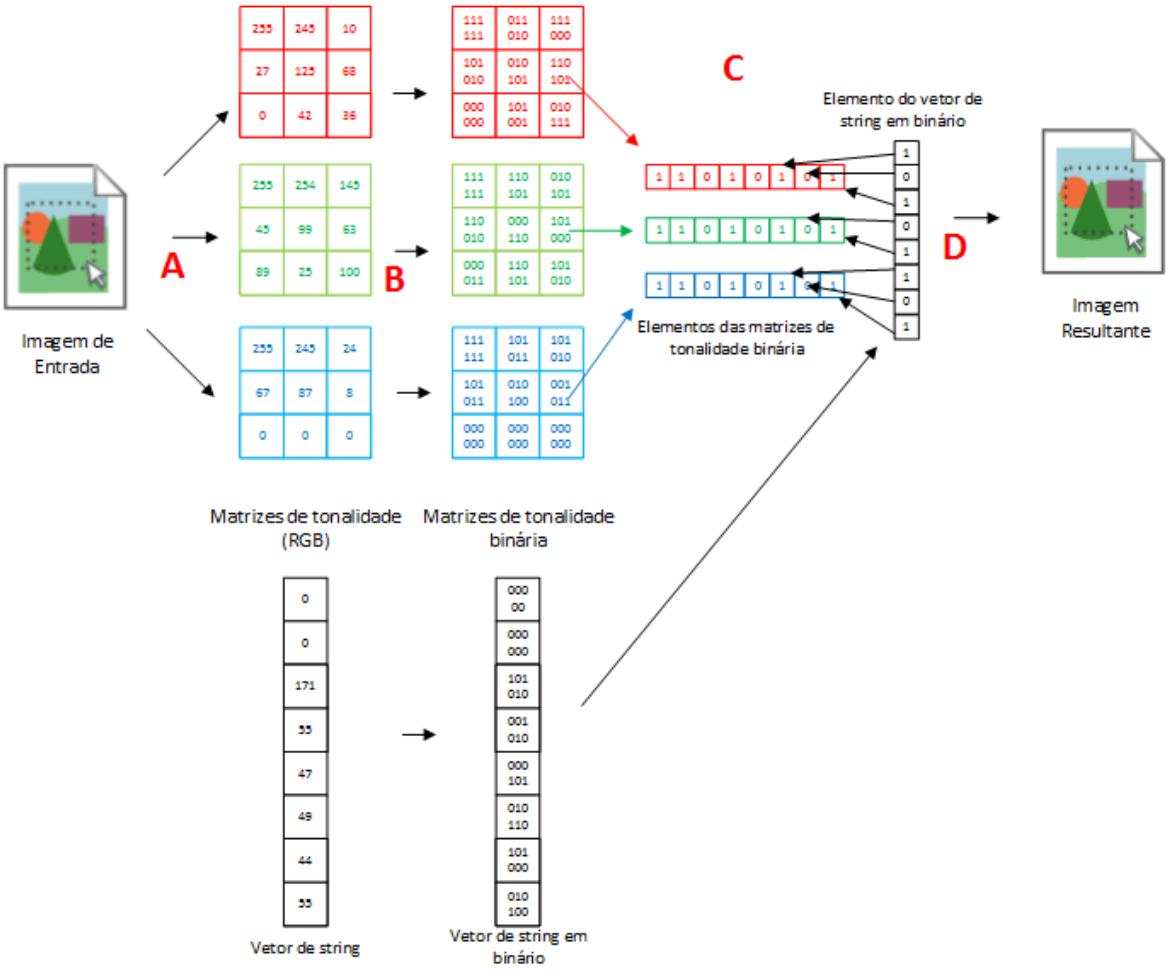

Figura 2: Processo utilizado para esconder a cifra na imagem de entrada, gerando a imagem resultante

O ponto A presente na Figura 2 representa a obtenção dos *pixel* da imagem de entrada. Essa imagem é uma imagem padrão e não é dada como entrada do programa. O motivo disso é que a imagem deve ter um certo formado e um certo tamanho (PNG, 1000x1000). Portanto, para o usuário não precisar procurar por uma imagem que satisfaça essas propriedades, foi decidido deixar uma imagem padrão.

Para começar a trabalhar com a imagem de entrada, primeiramente foram construídas três matrizes, que representam, seguindo a ordem que aparecem na imagem, os *pixel* que a formam. Cada matriz armazena o valor em binário de uma das três tonalidades do sistema RGB (vermelho, verde e azul). Sendo assim, tem-se uma matriz com as tonalidades de vermelho de cada pixel, uma com as tonalidades de verde e uma com as tonalidades de azul.

O ponto B presente na Figura 2 ilustra que cada elemento do vetor de *string* é convertido para binário, assim como cada elemento das matrizes de tonalidade.

O ponto C presente na Figura 2 indica o processo de esconder cada elemento convertido do vetor de *string* em cada elemento convertido das três matrizes de tonalidade. O processo segue a seguinte ordem:

- Os 3 bits mais significativos do elemento convertido do vetor de *string* substituem os 3 bits menos significativos do elemento convertido da matriz de tonalidade vermelha.
- Os 2 bits seguintes do elemento convertido do vetor de *string* substituem os 2 bits menos significativos do elemento convertido da matriz de tonalidade verde.
- Os 3 bits menos significativos do elemento convertido do vetor de *string* substituem os 3 bits menos significativos do elemento convertido da matriz de tonalidade azul.

As matrizes de tonalidade resultante são utilizadas para gerar a imagem resultante, como indicado no ponto D presente na Figura 2. Pelo fato de só ter mudado os bits menos significativos, o *pixel* resultante de cada processo é visualmente igual ao *pixel* original, fazendo com que não se possa observar que há uma mensagem escondida na imagem.

Porém, como será utilizado uma imagem padrão, é possível que se possa compará-la com a imagem resultante e, com isso, descobrir que há algo escondido na segunda. Além disso, ainda tem como descobrir até onde a imagem resultante difere da original, podendo limitar a área onde a mensagem está escondida. Para isso, foi decidido esconder números aleatórios nos *pixel* restantes, fazendo com que não seja trivial descobrir que parte da imagem resultante é a cifra e que parte é informação aleatória. O único método de se obter a mensagem é pegando o tamanho da mesma, contido nos três primeiros *pixel* da imagem. Entretanto, essa tarefa não é trivial para quem não conhece o algoritmo, já que a ordem que a informação é escondida nos elementos das matrizes de tonalidade importa quando se quer recuperar alguma informação delas.

2.1.5 Esteganografia - *StegOut.java*

Para poder recuperar o texto plano a partir da imagem resultante, o usuário deverá ir na aba Decriptografar para poder recuperar a cifra da imagem resultante e depois decriptá-la.

Esse módulo possui como entrada somente uma imagem.

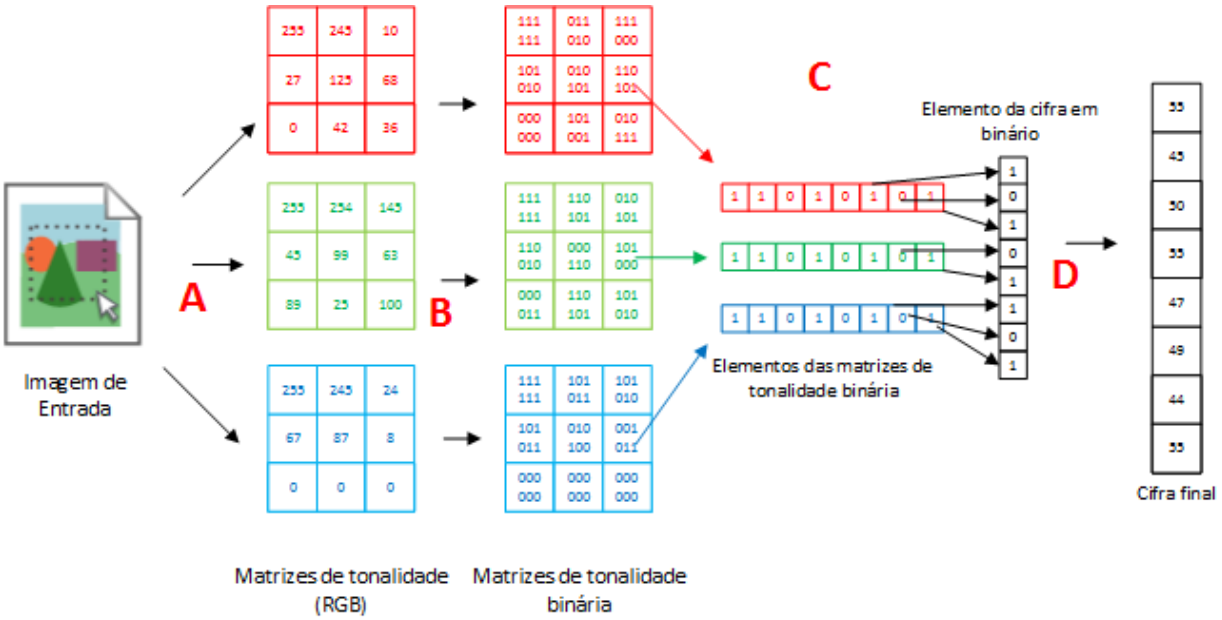

Figura 3: Processo utilizado para recuperar a cifra da imagem resultante

A Figura 3 ilustra o processo inverso mostrado na subseção 2.1.4.

O ponto A presente na Figura 3 representa a obtenção das matrizes de tonalidade da imagem de entrada. O método utilizado é o mesmo que o da subseção 2.1.4.

O ponto B presente na Figura 3 ilustra que cada elemento das matrizes de tonalidade é convertido para binário.

O ponto C presente na Figura 3 representa a obtenção do tamanho da mensagem, presente nos três primeiros *pixel* da imagem de entrada. Logo após, tendo como critério de parada o tamanho da mensagem, é obtido a cifra em binário, representada como um vetor de *string* onde cada elemento representa um dos 19 números (em binário) que representam um dos caracteres do texto plano.

O modo de recuperação das informações da imagem de entrada é o processo inverso do citado na subseção 2.1.4, ou seja:

- Os 3 bits mais significativos do elemento da cifra em binário é constituído pelos 3 bits menos significativos do elemento convertido da matriz de tonalidade vermelha.
- Os 2 bits seguintes do elemento da cifra em binário é constituído pelos 2 bits menos significativos do elemento convertido da matriz de tonalidade verde.
- Os 3 bits menos significativos do elemento da cifra em binário é constituído pelos 3 bits menos significativos do elemento convertido da matriz de tonalidade azul.

O ponto D presente na Figura 3 representa a conversão da cifra para decimal, gerando a cifra final. Essa conversão utiliza um método contido da classe *Maths.java*, explicado na subseção 2.1.1.

A cifra final é enviada para o módulo de decriptografia, que será explicado na próxima subseção.

2.1.6 Deriptografia - *CriptOut.java*

Para decriptografar uma mensagem foi utilizado o método inverso do módulo de criptografar, explicado na subseção 2.1.3. Este módulo tem como entrada a cifra resultante da subseção anterior e uma chave privada, referente a chave pública utilizada para gerar a cifra de entrada.

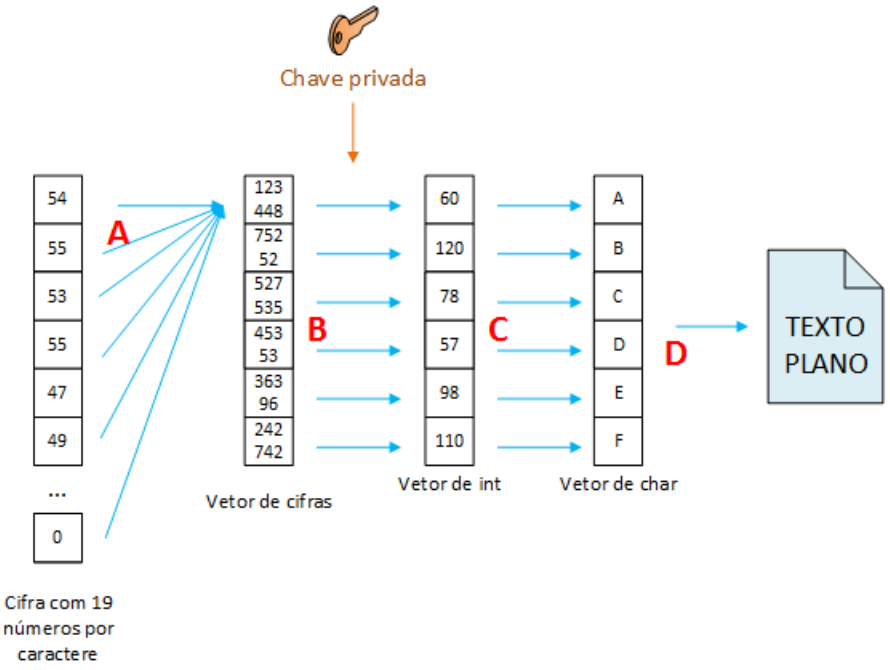

Figura 4: Processo utilizado para decriptar a cifra, gerando o texto plano

O ponto A presente na Figura 4 representa a obtenção no número cifrado a partir dos 19 números que ele foi dividido. Para isso é utilizado um método presente na classe *Converters* (subseção 2.1.1), que a cada 19 elementos da cifra, ignora-se os "0" e recupera o número cifrado original, salvando-o num vetor de cifras.

O ponto B presente na Figura 4 se refere à operação de decriptografar cada elemento do vetor de cifras. A operação feita segue o algoritmo RSA, sendo ela:

$M = C^D mod N$

Onde M é um caractere (decimal) do texto plano, C é o caractere cifrado do vetor de cifras e D e N são os parâmetros da chave privada. Todos os caracteres M são salvos num vetor de inteiros.

O ponto C presente na Figura 4 se refere ao processo de converter todos os elementos do vetor de inteiros para suas respectivas representações em ASCII, armazenando-as num vetor de *char*.

Por fim, o ponto D presente na Figura 4 representa a criação de um arquivo e escrevendo o vetor de *char* no mesmo.

O arquivo criado é idêntico ao texto plano original, provando a validade deste projeto.

2.2 Restrições do algoritmo

As análises a seguir se referem as restrições baseadas nos testes deste projeto, podendo sofrer alterações na aplicação do mesmo na *smart home*.

2.2.1 Tamanho da imagem

O tamanho da imagem de entrada influencia no tamanho máximo que o arquivo a ser escondido na mesma pode ter e no tempo de execução do projeto. Por isso, foi realizada uma análise para se decidir qual seria o tamanho ideal para este projeto, dado os testes que poderiam ser realizados.

Foi decidido, por fim, que a imagem de entrada deve medir 1000x1000 *pixel*.

Como dito e explicado na subseção 2.1.4, a imagem de entrada foi deixada como padrão para todas as esteganografias.

2.2.2 Tamanho do arquivo

O tamanho de um arquivo de texto (.txt) é simplesmente a quantidade de caracteres que o mesmo possui, sendo cada caractere equivalente a um byte. Como cada caractere é dividido em 19 números e cada um desses 19 números são armazenados em um pixel da imagem de entrada, então o tamanho máximo do arquivo pode ser calculado da seguinte forma:

Tamanho máximo = (Tamanho da imagem)^2/ 19

Sendo o tamanho da imagem 1000, temos que que o tamanho máximo do texto de entrada é 52631 bytes. Esse tamanho é suficiente para os testes que serão realizados para validar o projeto.

Uma mensagem de erro aparece quando o usuário escolhe um arquivo que ultrapassa esse tamanho.

2.2.3 Tempo de execução

O tempo de execução é dado pelo tempo em que a imagem demora para ser processada ou criada posteriormente. Apesar de ser um tempo curto (cerca de 5 segundos em computadores domésticos), o tempo pode ser otimizado se for utilizado um processador com maior poder computacional.

2.2.4 Nome dos arquivos de saída

Quando é executada a funcionalidade de criptografar um arquivo no formato *.txt*, gera-se como saída uma imagem no formato *.bmp* denominada *imagem* no diretório *out*. Já quando é realizada a funcionalidade de decriptografar a imagem produzida anteriormente, gera-se como saída um arquivo no formato *.txt* denominado *mensagem*.

O nome original do texto poderia ser armazenado, juntamente com seu conteúdo, na imagem. Porém, para deixar o código mais simples, foi utilizado o nome *mensagem.txt* como o texto plano de saída. Além disso, os nomes dos arquivos de saída não são escolhidos pelo usuário com o intuito de deixar o processo mais rápido.

2.2.5 Considerações importantes

Como dito anteriormente, essas análises foram feitas com base nos possíveis testes deste projeto. Porém, se fosse feita uma análise no ambiente da *smart home*, analisando o tamanho dos dados transferidos, essas restrições podem sofrer variações.

Essa análise pode ser feita através do limite de cada dispositivo, verificando qual seria o tamanho máximo de dado que cada um pode enviar (se for possível tal verificação). Outra análise que pode ser feita é verificar durante um determinado tempo o tamanho de todos os dados transferidos. Assim, pode ser considerado como tamanho máximo o maior valor encontrado, somado com algum fator de prevenção.

Tendo o tamanho máximo, o tamanho da imagem pode ser calculado da seguinte forma:

Tamanho da imagem = $\sqrt[2]{TamanhoMaximo * 19}$

Considerando o cenário em que o tamanho máximo foi definido como 100 bytes, o tamanho da imagem passaria a ser 44x44 *pixel*. Tendo somente 1936 *pixel* para serem processados, as matrizes de tonalidade da imagem poderiam estar predefinidas no algoritmo, fazendo com que ocorra uma economia de tempo, pois evitaria as conversões para o formato correto e as conversões para binário.

Dessa forma, fazendo uma boa análise do ambiente, o algoritmo ganharia eficiência, fator importante no processo de criptografia e troca de mensagens.

2.3 Estratégia para troca de chaves

Como será abordado na segunda parte deste projeto, uma forma para se melhorar a segurança da *smart home* é modularizar os sistemas existentes na casa, como, por exemplo, o de iluminação e o de entretenimento. Assim, uma estratégia interessante para que cada dispositivo dispusesse de autenticidade e de confidencialidade na comunicação com os demais é a atribuição de uma chave pública e outra privada. O uso dessas duas chaves permitirá que os dispositivos recebam e enviem informações de forma segura, como é previsto pelo uso de um método de criptografia assimétrico, já que a chave de encriptar é pública (permitindo que todos os dispositivos enviem dados a todos) e a chave de decriptar é privada (permitindo que apenas o dispositivo destino interprete os dados enviados).

Além dessas características intrínsecas à criptografia assimétrica, uma estratégia de troca das chaves pode ser implementada para que o morador (ou até mesmo automaticamente) altere as chaves pública e privada dos dispositivos periodicamente, o que melhoria ainda mais a segurança dos dados que circulam na rede da casa, já que esta medida reduziria a vulnerabilidade do sistema caso alguma das chaves privadas fosse descoberta.

2.4 Instruções Gerais

O algoritmo construído neste trabalho foi desenvolvido na IDE *Netbeans* no sistema operacional *Windows* principalmente para facilitar o desenvolvimento da interface gráfica. Assim o arquivo enviado estará disponível para importação como um projeto do *Netbeans* e, além dele, será disponibilizado também o próprio arquivo executável no formato *.jar*.

2.4.1 Como compilar e executar

Como mencionado na seção anterior, o código do algoritmo deverá ser importado para a IDE *Netbeans* para que possa ser compilado, já que os detalhes de implementação utilizados por esta ferramenta na geração de interfaces gráficas são muito peculiares. Dessa forma, o primeiro passo é extrair o arquivo disponibilizado no formato *.rar* e abri-lo através do *Netbeans*. Uma vez que o projeto estiver aberto, será possível encontrar todos os pacotes e classes detalhados em seções anteriores, sendo possível executá-lo através do botão 'Executar projeto' (ou tecla F6). Um método alternativo em que não é necessário compilar o projeto é simplesmente executar o arquivo *.jar* enviado, bastando abri-lo.

2.4.2 Tutorial de uso

Assim que o projeto estiver em execução, a interface encontrada será a seguinte:

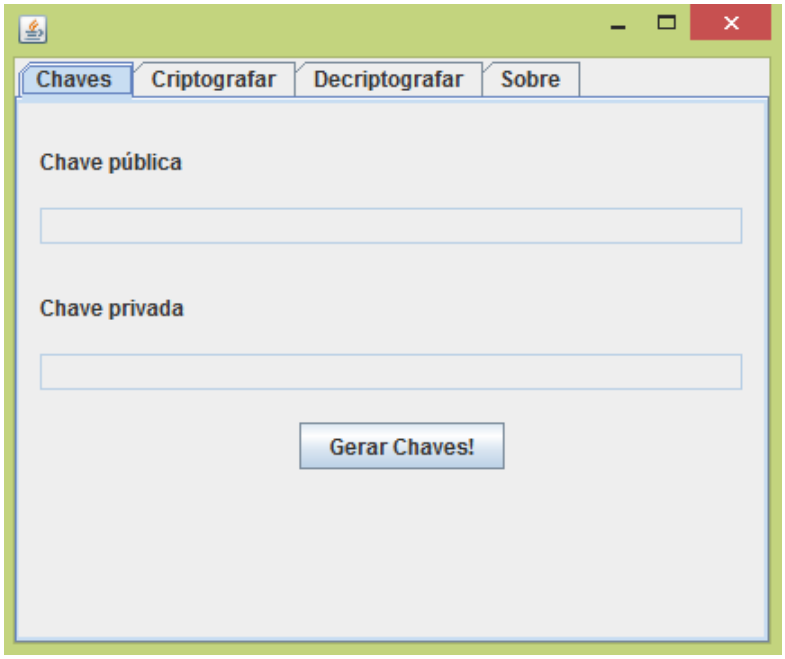

Figura 5: Tela gerada ao executar o projeto

Como pode ser observado na Figura 5, o primeiro passo se resume à geração das chaves pública e privada para que possam ser utilizados o módulo de criptografia e de decriptografia. Basta clicar no botão 'Gerar chaves!' para que esta ação seja realizada e as chaves estejam disponíveis para o usuário. O próximo passo é selecionar a aba 'Criptografar' e escolher o arquivo que será criptografado através do botão 'Pesquisar', sendo possível acessar qualquer diretório e qualquer arquivo (.txt) existente na máquina. Uma vez que este foi escolhido, é necessário copiar para o campo 'Chave pública' o valor da chave pública disponibilizado na aba 'Chaves'. Quando estas ações tiverem sido realizadas, basta clicar no botão 'Criptografar!', como mostrado na Figura 6, para que a mensagem do arquivo selecionado seja criptografada e acessível através da imagem de saída *imagem.bmp*, a qual se encontra no diretório 'out' do arquivo *.rar* enviado. Para especificar este detalhe ao usuário, é exibida a mensagem ilustrada na Figura 7.

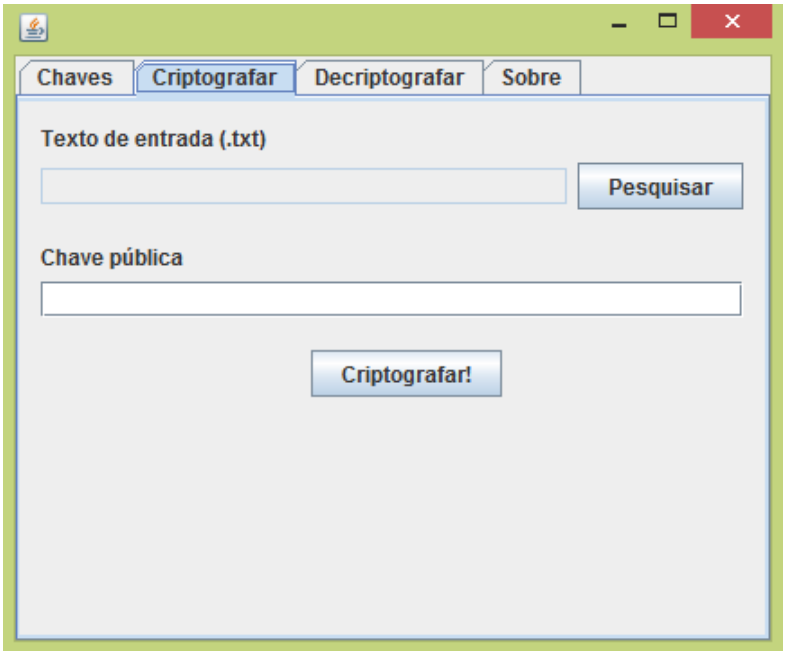

Figura 6: Tela referente à aba Criptografar

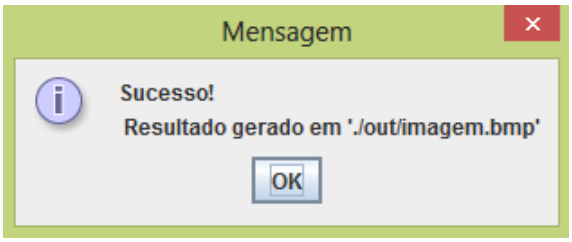

Figura 7: Tela gerada ao criptografar um arquivo

Após clicar clicar em 'Ok' na tela anterior, as funcionalidades da ferramenta estarão disponíveis novamente, sendo possível realizar o próximo passo, referente à decriptografia da mensagem escondida na imagem de saída gerada. Para isso, é necessário selecionar a aba 'Decriptografar' e selecionar através do botão 'Pesquisar' o diretório 'out' e escolher a imagem designada *imagem.bmp*. Após escolher a imagem de entrada, é preciso copiar para o campo 'Chave privada' o valor da chave privada disponibilizado na aba 'Chaves'. Quando estas ações tiverem sido realizadas, basta clicar no botão 'Decriptografar', como mostrado na Figura 8, para que a mensagem escondida na imagem de entrada seja descriptografada. A mensagem decriptografada é gerada no diretório 'out' no arquivo denominado *mensagem.txt*, como especificado pela tela mostrada na Figura 9.

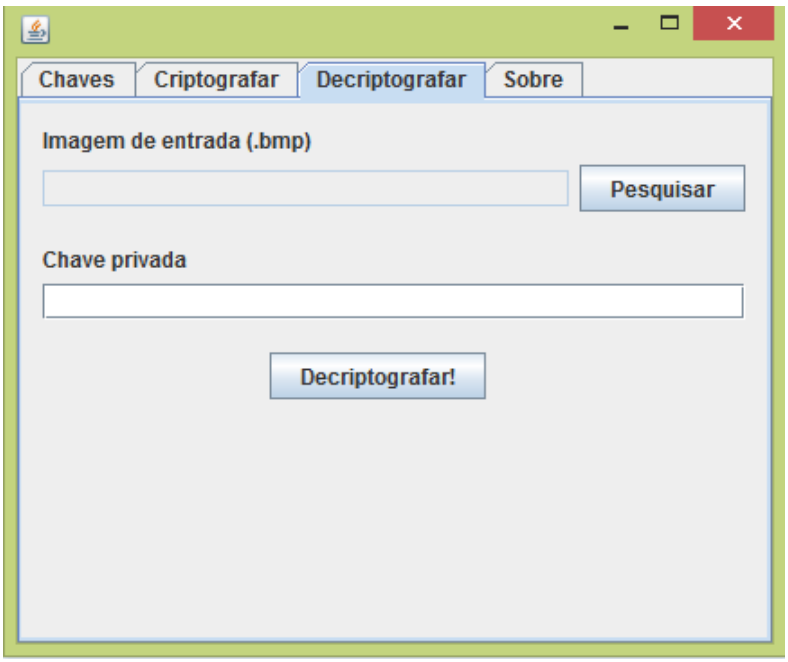

Figura 8: Tela referente à aba Decriptografar

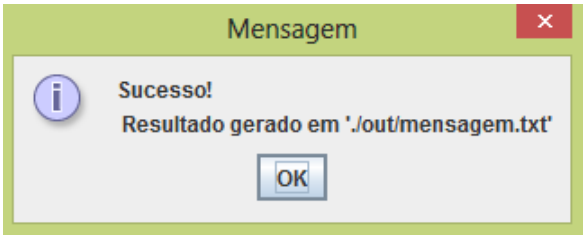

Figura 9: Tela referente gerada ao decriptografar um arquivo

3 Segunda Tarefa

3.1 Segurança das informações trocadas

Uma das medidas necessárias para se garantir a segurança de uma *smart home* é a implementação do controle da troca de informações entre os dispositivos, visto que os dados transmitidos pela rede da residência, uma vez que obtidos e analisados por um atacante, podem se tornar uma ameaça para a segurança de todos os dispositivos conectados a ela.

Para que haja um controle eficiente e seguro da troca de informações, primeiramente sugere-se que cada dispositivo tenha seu par de chaves pública e privada para que todo o tráfego de dados seja realizado com o uso do algoritmo construído, conferindo confidencialidade e autenticidade das mensagens transmitidas a todos os dispositivos.

Aliada a esta medida, é proposta a modularização do sistema de automatização da *smart home*, a qual consiste em dividir o sistema principal em subsistemas (módulos) de acordo com a funcionalidade disponibilizada à casa. Diferentes subsistemas que poderiam ser implementados são os de iluminação, de refrigeração e de

monitoramento de câmeras.

Dessa forma, a responsabilidade de cada módulo baseia-se no agrupamento de dispositivos de funcionalidade semelhante, como câmeras e alarmes no subsistema de segurança e luzes e cortinas no subsistema de iluminação. A partir desse agrupamento, é possível atribuir para cada subsistema uma subrede, o que permitirá uma gerência mais robusta em relação aos IPs de cada dispositivos.

Reunindo as propostas anteriores, sugere-se que apenas os dispositivos do mesmo subsistema possuam as chaves públicas uns dos outros, já que a troca de informações é mais intensa em aparelhos que atuam na mesma facilidade. Caso dispositivos de diferentes subsistemas queiram se comunicar, será necessária a comunicação entre os nós responsáveis pela gerência de cada subrede. Assim, por exemplo, quando algum dispositivo da iluminação quiser enviar alguma informação para outro dispositivo do monitoramento externo, será preciso que os nós de cada um desses dois subsistemas se comuniquem de forma que o dispositivo de iluminação receba a chave pública do dispositivo de monitoramento externo.

Uma última medida para se melhorar a segurança das informações trocadas é alterar periodicamente as chaves pública e privada dos dispositivos, diminuindo as chances de acesso a elas. Este processo seria controlado por cada nó de cada subsistema, o qual teria a função de enviar dados a cada dispositivo de seu grupo informando a nova chave privada e todas as novas chaves públicas do módulo a qual pertence. Esta atualização poderia ser programada para ser realizada em algum dia e horário em que as atividades dos moradores sejam mínimas, como o período em que eles estão dormindo por exemplo.

3.2 Segurança das informações armazenadas

Da mesma forma como os nós de cada subsistema são responsáveis pela atualização dos pares de chave e pela comunicação entre dispositivos de diferentes módulos, eles tem a função de armazenar as informações necessárias, incluindo as chaves dos dispositivos. Entretanto, é arriscado armazenar esses dados sem nenhuma criptografia, pois os nós podem sofrer algum tipo de ataque e o sigilo das chaves pode ser quebrado. Dessa forma, sugere-se que cada nó também tenha seu par de chaves pública e privada e execute o algoritmo elaborado, assim todas as informações armazenadas terão sido criptografadas em formato *.bmp*.

Além dessa medida, seria adequada a implementação de *firewalls* em cada um dos nós, fazendo com que sejam restringidos qualquer tipo de acesso externo através da proteção de intervalos de endereço IP contra rastreamento e ataques em geral.

3.3 Disponibilização de acesso remoto à *smart home*

Analisando os métodos disponíveis para a implementação do acesso remoto à *smart home*, verifica-se viabilidade tanto através do uso de serviços de nuvem, quanto de um servidor local. Entretanto, caso fosse escolhido utilizar os serviços de nuvem, seria necessário armazenar todas as chaves e informações adicionais do sistema na nuvem, o que submeteria estes dados às políticas de segurança da empresa de *Cloud Service*. Como um projeto de uma *smart home* possui um nível crítico relacionado à segurança, é imperativo o sigilo dos dados compar-

tilhados, algo que é difícil de assegurar ou detectar quando se faz uso da infraestrutura de nuvem, interferindo na confidencialidade do projeto desenvolvido.

Além dessa questão, é preciso analisar que haverá um fluxo constante aos nós de cada subsistema, o que pode representar um gargalo ao acessar à rede se escolhida a implementação através de servidores locais. Porém, ao se realizar uma atividade intensa em serviços de nuvem (como a que ocorreria caso os nós fossem implementados através deste paradigma), pode ocorrer uma lentidão na resposta gerada para cada dispositivo, o que não é atrativo em um projeto de uma residência. É valido também pensar em cenários em que a infraestrutura da nuvem passe por algum tipo de atualização, o que talvez impedisse a utilização de todas as funcionalidades da casa, afetando a disponibilidade do sistema.

Os custos para se manter uma rede eficiente e para se manter serviços executando na nuvem podem ser considerados equivalente, já que ambos exigem um alto investimento para que o desempenho do sistema de automatização seja implementado com excelência. Dessa forma, a implementação de cada nó dos subsistemas baseada no uso de servidores locais é ainda a melhor solução tendo em vista todas as medidas descritas nas seções anteriores, envolvendo o uso do algoritmo criptográfico e de *firewalls* nos nós responsáveis por cada subsistema.

4 Conclusão

O projeto de uma *smart home* exige a implementação de medidas que reforcem a segurança do sistema de automatização da casa, para isso foram propostos o uso de um algoritmo que apresenta módulos de criptografia e esteganografia aliados a uma organização da rede da residência em subredes, constituindo a modularização do sistema da casa de acordo com o agrupamento de dispositivos que atuam na mesma funcionalidade da residência.

Cada dispositivo terá um par de chaves pública e privada, o que possibilita a execução do algoritmo assimétrico em toda a troca de informações na rede, garantindo a existência de autenticidade e de confidencialidade na comunicação entre os dispositivos. Uma característica referente da modularização do sistema é a presença de um nó para cada subsistema, o qual é responsável pela comunicação entre subsistemas e pela atualização dos pares de chaves para cada dispositivo pertencente a seu módulo. Ainda sobre os subsistemas, foi proposta a instalação de *firewalls* para que acessos externos fossem restringidos, fortalecendo a segurança da rede da *smart home*.

Diante das características citadas anteriormente sobre o planejamento desenvolvido, foi possível definir que a melhor forma de se implementar o acesso remoto à *smart home* é através do uso de servidores locais, já que a intensa atividade de comunicação entre os dispositivos exige uma infraestrutura que permita um controle robusto da segurança e do sigilo dos dados transmitidos e armazenados. Além disso, é necessário que o método escolhido forneça uma alta disponibilidade, o que, muitas vezes, pode ser algo crítico em serviços de nuvem, visto que podem ocorrer atualizações na nuvem que limitem o desempenho ótimo das funcionalidades da *smart home*.