UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Dúnia Marchiori

DETERMINAÇÃO EM TEMPO CONSTANTE DE RAÍZES DE POLINÔMIOS SOBRE CORPOS FINITOS PARA CÓDIGOS DE CORREÇÃO DE ERROS

Florianópolis

2019

Dúnia Marchiori

DETERMINAÇÃO EM TEMPO CONSTANTE DE RAÍZES DE POLINÔMIOS SOBRE CORPOS FINITOS PARA CÓDIGOS DE CORREÇÃO DE ERROS

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciência da Computação para a obtenço do Grau de Bacharel em Ciência da Computação.
Orientador: Prof. Dr. Ricardo Felipe Custódio
Coorientador: Douglas Marcelino Beppler Martins

Florianópolis

2019

Dúnia Marchiori

DETERMINAÇÃO EM TEMPO CONSTANTE DE RAÍZES DE POLINÔMIOS SOBRE CORPOS FINITOS PARA CÓDIGOS DE CORREÇÃO DE ERROS

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciência da Computação", e aprovado em sua forma final pelo Programa de Graduação em Ciência da Computação.

Florianópolis, 5 de junho 2019.

Prof. Me. José Francisco Danilo de Guadalupe Correa Fletes
Coordenador do Curso

Banca Examinadora:

Prof. Dr. Ricardo Felipe Custódio
Orientador

Douglas Marcelino Beppler Martins
Coorientador

Me. Lucas Pandolfo Perin
Universidade Federal de Santa Catarina

À minha família

AGRADECIMENTOS

Agradeço a meus pais por sempre torcerem pelo meu sucesso e terem me ensinado a importância da educação. Tenho imensa gratidão pelo apoio e todo o incetivo que me foi dado para buscar alcançar sempre o meu melhor. Ambos me ensinaram o verdadeiro significado de amor, não medindo esforços para que eu chegasse até aqui.

Agradeço o professor Custódio, Douglas Martins e Gustavo Zambonin por terem despertado meu gosto pela pesquisa. Aos meus colegas no Laboratório de Segurança em Computadores agradeço por terem proporcionado um ambiente alegre e amistoso.

Agradeço os meus amigos por todo o carinho e companhia, especialmente a Vinicius Steffani Schweitzer e Fabio Moreira que foram meus companheiros desde o primeiro semestre até o último, e a todos os outros colegas que fizeram parte desta jornada. Agradeço as minhas amigas Isabela Dalla Brida e Carini Fontanelli por estarem sempre ao meu lado. Agradeço também a minha colega de quarto Eduarda Adami pela companhia durante esses anos e por estar sempre disposta a me ouvir.

'Well, I never heard it before,' said the Mock Turtle; 'but it sounds uncommon nonsense.'
- Lewis Carroll, Alice in Wonderland

RESUMO

Os sistemas de criptografia mais populares atualmente são Rivest-Shamir-Adleman (RSA) e criptografia de curvas elípticas, os quais se baseiam em problemas que algoritmos quânticos conseguem resolver em tempo polinomial. Por este motivo diferentes propostas de criptossistemas vêm sendo estudadas na área criptografia pós-quântica, para que as informações continuem seguras mesmo contra um adversário com um computador quântico. O esquema de McEliece é um esquema de criptografia de chave pública proposto em 1978. A sua segurança se baseia no fato de que a chave pública é indistinguível de uma matriz aleatória e no problema de decodificação geral, o que leva o esquema a resistir a ataques de computadores quânticos e que o categoriza como um criptossistema pós-quântico. O criptossistema de McEliece faz uso de código de Goppa em sua proposta original. Atualmente, diferentes estudos demonstraram a vulnerabilidade da decodificação dos códigos de Goppa em relação a ataques de temporização, onde a variabilidade do tempo de execução traz insegurança para uma implementação.
Este trabalho tem como objetivo apresentar os códigos de Goppa, seu uso no esquema de McEliece e analisar diferentes implementações para a determinação de raízes de polinômios, um passo crucial na decodificação dos códigos. Para isso, o esquema original de McEliece será apresentado juntamente com os códigos de Goppa. O principal algoritmo para decodificação dos códigos, o algoritmo de Patterson, será descrito e implementado fazendo o uso de diferentes métodos para a determinação de raízes de polinômios. Por fim, estas implementações serão comparadas em relação a sua vulnerabilidade a ataques de temporização.
Palavras-chave: criptografia, criptografia pós-quântica, McEliece, Goppa, ataques de temporização

LISTA DE FIGURAS

Figura 1 Ilustração de como uma mensagem com erros pode ser corrigida 37
Figura 2 Ilustração da execução do algoritmo Trace de Berlekamp recursivo 58

LISTA DE TABELAS

Tabela 1 Tempo de execução dos algoritmos para determinação de raízes de polinômios 61

LISTA DE ABREVIATURAS E SIGLAS

ECDSA Elliptic Curve Digital Signature Algorithm 25
ELP Error Locator Polynomial 37
RSA Rivest-Shamir-Adleman 43
BTA Berlekamp Trace Algorithm 53
MDC Máximo Divisor Comum 54

LISTA DE SÍMBOLOS

$\mathbb{F}_q$	Corpo de Galois de ordem q	31
$GF(q)$	Corpo de Galois de ordem q	31
$\langle v, u \rangle$	Produto interno dos vetores v e u	32
c^T	Matriz c transposta	32
$d(x, y)$	Distância de Hamming entre os vetores x e y	32
$w(x)$	Peso de Hamming do vetor x	32

LISTA DE ALGORITMOS

Algoritmo 1 Algoritmo de Patterson 38
Algoritmo 2 *Berlekamp Trace Algorithm* $BTA(p, i)$ 54
Algoritmo 3 Algoritmo de Fedorenko 55
Algoritmo 4 *Berlekamp Trace Algorithm* adaptado $BTA_adapt(p, t)$ 59

SUMÁRIO

1 INTRODUÇÃO 25
1.1 OBJETIVOS 26
1.1.1 Objetivo geral 26
1.1.2 Objetivos específicos 27
2 CÓDIGOS CORRETORES DE ERROS 29
2.1 TEORIA DE CÓDIGOS 29
2.2 CONCEITOS INICIAIS 29
2.2.1 Espaço nulo de matrizes 30
2.2.2 Espaço linha de matrizes 30
2.2.3 Código de Gray 31
2.3 CÓDIGOS LINEARES 31
2.3.1 Códigos de Hamming 32
2.4 CÓDIGOS DE GOPPA 33
2.4.1 Matriz de paridade de um código Goppa 34
2.4.2 Matriz geradora de um código Goppa 36
2.4.3 Codificação de um código Goppa 36
2.4.4 Decodificação de um código Goppa 36
2.4.4.1 Algoritmo de Patterson 37
2.4.5 Exemplo de um código Goppa binário 39
3 CRIPTOSSISTEMA DE MCELIECE 43
3.1 CRIPTOGRAFIA ASSIMÉTRICA 43
3.2 CRIPTOSSISTEMA DE MCELIECE 43
3.2.1 Geração do par de chaves 44
3.2.2 Cifragem de uma mensagem 44
3.2.3 Decifragem de uma mensagem 45
3.2.4 Exemplo de cifragem e decifragem de uma mensagem com McEliece 45
3.3 ATAQUES AO CRIPTOSSISTEMA DE MCELIECE 47
3.3.1 Ataques estruturais e de decodificação 48
3.3.2 Ataques de temporização 49
3.3.3 Variações do sistema 51
4 ALGORITMOS PARA DETERMINAÇÃO DE RAÍZES DE POLINÔMIOS 53
4.1 AVALIAÇÃO EXAUSTIVA 53
4.2 ALGORITMO TRACE DE BERLEKAMP 53
4.3 ALGORITMO DE FEDORENKO 54

5 COMPARAÇÃO ENTRE OS DIFERENTES MÉTODOS PARA DETERMINAÇÃO DE RAÍZES DE POLINÔMIOS 57
5.1 IMPLEMENTAÇÃO DOS ALGORITMOS 57
5.1.1 Algoritmo Trace de Berlekamp 57
5.1.1.1 Algoritmo de Berlekamp modificado 59
5.1.2 Algoritmo de Fedorenko 60
5.2 ANÁLISE DOS TEMPOS OBTIDOS 60
6 CONCLUSÃO 63
6.1 TRABALHOS FUTUROS 63
REFERÊNCIAS 65
ANEXO A – Implementação dos códigos de Goppa 71
ANEXO B – Implementação do Berlekamp Trace Algorithm 83
ANEXO C – Implementação do algoritmo de Fedorenko 87
ANEXO D – Artigo 93

1 INTRODUÇÃO

Um dos principais objetivos da segurança em computação é prover confidencialidade dos dados, que garante que dados sigilosos e privados não fiquem disponíveis para acesso de pessoas não autorizadas. Criptossistemas são um conjunto de algoritmos criptográficos que prezam pela confidencialidade, assegurando que um adversário que intercepte alguma mensagem cifrada não seja capaz de recuperar a mensagem original caso não tenha posse da chave necessária para a decifragem.

Criptossistemas podem ser classificados em dois tipos: simétricos e assimétricos. Em criptossistemas simétricos, ambas as partes da comunicação têm posse da mesma chave para a cifragem e decifragem das mensagens. A chave então é um segredo compartilhado apenas entre as partes que compõem a comunicação. Uma desvantagem desses sistemas é que, para a realização do acordo do segredo, que permitiria uma comunicação segura em um canal inseguro, há a necessidade de um canal seguro para o compartilhamento do mesmo.

Criptografia assimétrica, ou criptografia de chave pública, se caracteriza pela presença de um par de chaves, uma chave pública e outra privada, que são utilizadas para realizar operações complementares, como cifragem e decifragem ou assinar arquivos digitalmente e verificar estas assinaturas. A chave privada deve ser conhecida apenas pela entidade que a gerou mas a chave pública pode ser distribuída livremente, o que remove a necessidade de um canal seguro para o acordo de chaves.

Exemplos de algoritmos assimétricos comumente usados são o RSA, criado por Ron Rivest, Adi Shamir e Len Adleman (RIVEST; SHAMIR; ADLEMAN, 1978), e ECDSA . Estes e demais algoritmos baseiam sua segurança na dificuldade da fatoração de inteiros ou logaritmos discretos, ou seja, o fato destes problemas serem intratáveis garante a segurança para os esquemas. Em 1999, Peter Shor publicou um algoritmo capaz de calcular estes dois problemas em tempo polinomial em um computador quântico (SHOR, 1999). Isso significa que dados cifrados hoje não necessariamente estarão seguros no futuro.

Como forma de combater esta situação, na área da criptografia pós-quântica são estudados esquemas criptográficos baseados em problemas que não foram afetados pelo algoritmo de Shor para que a segurança das informações cifradas não seja ameaçada por um adversário com um computador quântico. Entre as propostas de criptossistemas pós-quânticos estão os sistemas baseados em funções de hash, em reti-

culados e em códigos de correção de erros.

Durante a transmissão de dados podem ocorrer erros que transformam a mensagem recebida numa mensagem diferente da que foi enviada. Códigos de correção de erros são capazes de detectar estes erros em uma mensagem e corrigi-los. Para isso fazem uso de redundância, adicionando informações à mensagem que podem auxiliar na sua correção após a transmissão por um canal com ruído. Em sistemas baseados em códigos deste tipo, os erros são adicionados propositalmente à mensagem, como forma de dificultar a leitura dos dados por partes indesejadas.

O esquema de McEliece (MCELIECE, 1978) foi o primeiro esquema de criptografia de chave pública baseado em códigos de correção de erros proposto e resistiu a diversos ataques realizados até os dias atuais, sendo assim, um ótimo candidato como criptossistema pós-quântico. O código corretor de erros em que McEliece se baseia originalmente são códigos de Goppa (GOPPA, 1970), que é utilizado para a geração das chaves pública e privada.

Nos últimos anos, diferentes estudos (BUCERZAN et al., 2017) (SHOUFAN et al., 2010) (STRENZKE et al., 2008) indicaram vulnerabilidades no algoritmo mais utilizado na etapa de decodificação dos códigos Goppa, o algoritmo de Patterson, em relação a ataques de temporização. Diferentes implementações do algoritmo de Patterson, fazendo uso de diferentes formas de determinação de raízes de polinômios, por exemplo, demonstraram variabilidade em seu tempo de execução quando a execução da mesma tarefa diversas vezes foi analisada. Isso cria brechas para ataques e compromete a segurança do esquema.

Este trabalho visa realizar um estudo sobre o esquema de McEliece original e comparar implementações do mesmo fazendo uso de diferentes métodos para a determinação de raízes de polinômios, analisando a variabilidade nos tempos de execução de cada método e, consequentemente, a vulnerabilidade a ataques de temporização.

1.1 OBJETIVOS

1.1.1 Objetivo geral

Este trabalho tem como objetivo apresentar um estudo sobre o esquema de criptografia de chave pública de McEliece utilizando códigos de Goppa e comparar diferentes estratégias de implementação do mesmo sob o ponto de vista de vulnerabilidade a ataques *side-channel*

na etapa de decodificação. Uma implementação para cada estratégia de decodificação dos códigos Goppa descrita no trabalho também será apresentada.

1.1.2 Objetivos específicos

i. Descrever o esquema de criptografia assimétrica de McEliece original, que utiliza códigos de Goppa;

ii. Descrever os códigos de Goppa, dando maior foco na etapa de decodificação do mesmo;

iii. Descrever o algoritmo de Patterson para a decodificação de códigos Goppa;

iv. Analisar o comportamento da implementação de diferentes métodos de determinação de raízes de polinômios no algoritmo de decodificação considerando o seu tempo de execução na avaliação de diferentes polinômios;

v. Comparar as implementações quanto à sua variabilidade de tempo de execução e sua vulnerabilidade a ataques de temporização segundo os valores resultantes da análise.

2 CÓDIGOS CORRETORES DE ERROS

Neste capítulo serão apresentados os principais conceitos sobre códigos lineares, necessários para um melhor entendimento sobre códigos de Goppa e o esquema de McEliece. O conceito de teoria de códigos é introduzido na Seção 2.1, seguido pela descrição de códigos lineares na Seção 2.3. Códigos de Hamming e códigos de Goppa são apresentados nas seções 2.3.1 e 2.4, respectivamente.

2.1 TEORIA DE CÓDIGOS

Em 1948, uma publicação de Claude Shannon deu início à uma nova área de pesquisa, chamada teoria da informação. Em (SHANNON, 1948), Shannon descreveu conceitos como bit, entropia da informação e capacidade de um canal com ruído e demonstrou um esquema de um sistema de comunicação, dividido em cinco partes.

Os resultados exibidos em relação à comunicação com canais com ruído demonstraram que seria possível codificar informação de forma que ela pudesse ser transmitida em taxas próximas à capacidade do canal com perda mínima de informação durante a comunicação. Isto teve influência em outra área, a de teoria de códigos.

A teoria de códigos lida com a detecção e correção de erros em transmissões. Como canais podem estar sujeitos à ruído durante a comunicação, erros podem ser introduzidos nas mensagens sendo enviadas. Fazendo uso de códigos de correção de erros, além de permitirem detectar tais erros, é possível recuperar a mensagem original. Para atingir este objetivo é feito uso do conceito de redundância, uma abordagem também explorada por Shannon.

Os códigos apresentados neste capítulo são lineares, a classe mais estudada entre todos os códigos pela sua simplicidade de codificação e decodificação quando comparados a códigos não-lineares.

2.2 CONCEITOS INICIAIS

Para um melhor entendimento do que será detalhado a seguir, é necessário que algumas definições sobre matrizes e códigos Gray sejam relembradas.

2.2.1 Espaço nulo de matrizes

Seja A uma matriz $m \times n$. Considerando $Ax = \overrightarrow{0}$, o conjunto de todos os vetores x que satisfazem a equação é chamado de espaço nulo da matriz A. Por exemplo, seja $A = \begin{pmatrix} 2 & 1 \\ -4 & -2 \end{pmatrix}$ e $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. Temos que $Ax = \overrightarrow{0}$ é

$$\begin{pmatrix} 2 & 1 \\ -4 & -2 \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{2.1}$$

Para resolvermos podemos reduzir a matriz A multiplicando a primeira linha por 2 e somando o resultado à segunda linha. Dessa forma, temos

$$A' = \begin{pmatrix} 2 & 1 \\ 0 & 0 \end{pmatrix} \tag{2.2}$$

. O sistema $A'x = \overrightarrow{0}$ é equivalente a $Ax = \overrightarrow{0}$. Podemos ver $A'x = \overrightarrow{0}$ como um sistema de equações tal que

$$\begin{cases} 2x_1 + x_2 = 0 & \text{(2.3a)} \\ 0x_1 + 0x_2 = 0 & \text{(2.3b)} \end{cases}$$

Pela equação 2.3b temos que x_2 pode ser considerada uma variável livre. Pela equação 2.3a temos que $x_1 = \frac{1}{2}x_2$. Logo, o espaço nulo da matriz A é constituído do conjunto $N(A) = \left\{ \begin{pmatrix} \frac{1}{2}x \\ x \end{pmatrix}, x \in \mathbb{R} \right\}$. O conceito de espaço nulo de uma matriz é necessário para a criação da matriz geradora de um código através da sua matriz de paridade.

2.2.2 Espaço linha de matrizes

Seja A uma matriz $m \times n$. O seu espaço linha é formado pelo conjunto de todos as possíveis combinações lineares de cada linha. Uma combinação linear de vetores $v_1, v_2, \ldots, v_m$ é qualquer vetor na forma $n_1 \cdot v_1 + n_2 \cdot v_2 + \cdots + n_m \cdot v_m$. Portanto, seja $A = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$. O seu espaço linha será o conjunto formado por todos os vetores na forma $n_1 \cdot \big(2, 0\big) + n_2 \cdot \big(0, 1\big) = \big(2n_1, n_2\big)$.

2.2.3 Código de Gray

Código de Gray é um sistema de código binário criado por Frank Gray (FRANK, 1947) em que apenas um bit muda de valor de um vetor para o seguinte como 00, 01, 11, 10. No algoritmo de Fedorenko (Seção 4.3) utilizamos códigos Gray para a ordenação de polinômios. Por exemplo, os polinômios $x^2 + x + 1$, x^2, $x^2 + 1$ podem ser representados pelos vetores 111, 100 e 101, respectivamente. Estes vetores ordenados em código Gray poderiam ter a sequência 100, 101, 111, ou seja, x^2, $x^2 + 1$ e $x^2 + x + 1$.

2.3 CÓDIGOS LINEARES

O alfabeto dos códigos lineares que serão descritos a seguir é um corpo finito denotado como $\mathbb{F}_q$ ou $GF(q)$. $\mathbb{F}_q$ é uma estrutura formada por um conjunto de q elementos e duas operações, soma e multiplicação, que possuem as propriedades de associatividade, fechamento, existência de um elemento neutro e um elemento inverso, comutatividade e distributividade da multiplicação(STALLINGS, 2014).

A quantidade de elementos de um corpo finito deve ser sempre uma potência de um número primo p^n, sendo n um número inteiro positivo, como provado em (NIEDERREITER; XING, 2009, pp. 1-2). O número primo p é chamado de característica do corpo. Corpos binários, ou seja, corpos com característica 2, são especiais para a computação pois seus elementos, por usarem apenas 0 e 1, podem ser facilmente relacionados a bits em um computador.

Seja $\mathbb{F}_q^n$ um espaço vetorial de todas as n-tuplas sobre $\mathbb{F}_q$. Um código linear C de dimensão k é um subespaço vetorial de dimensão k de $\mathbb{F}_q^n$, que pode ser denominado também como um código [n,k] sobre $\mathbb{F}_q$. Dessa forma, o código C possui q^k elementos, que são chamados de *codewords*, ou palavras.

Segundo (HUFFMAN; PLESS, 2010), podemos representar códigos lineares através de uma matriz geradora ou uma matriz de paridade.

- Uma matriz geradora G de um código [n,k] C é uma matriz de dimensão $k \times n$ em que as linhas são vetores que formam uma base de C. Portanto, para cada $c \in C$ há um vetor $x \in \mathbb{F}_q^k$ tal que $c = xG$.
- Uma matriz de teste de paridade H de um código C é a matriz geradora do código *dual* (ou ortogonal) $C^\perp$. $C^\perp$ é um conjunto

formado por vetores v_i cujo produto interno com vetores $u_i \in C$ resulta em 0, ou seja, $C^{\perp} = \{v \in \mathbb{F}_q^n \mid \langle v, u \rangle = 0, \forall u \in C\}$. Isto implica que uma palavra c pertence ao código C se, e somente se, $Hc^T = \overrightarrow{0}$.

Sendo H uma matriz de paridade de um código linear [n,k] C, o vetor resultante $S(y) = Hy^T$ chama-se *síndrome* de y.

Duas métricas usadas na teoria de códigos são a distância e o peso de Hamming. Seja $x, y \in \mathbb{F}_q^n$. A distância de Hamming $d(x, y)$ entre dois vetores x e y é o número de coordenadas que diferem entre x e y. Já o peso $w(x)$ do vetor w é o número de coordenadas diferentes de 0, ou seja, $w(x) = d(x, \overrightarrow{0})$.

Uma característica importante de um código C sobre $\mathbb{F}_q^k$ é a sua distância mínima, que indica o valor da menor medida de distância entre todas as palavras distintas de C. O código C tendo uma distância mínima d pode corrigir $t = \left\lfloor \frac{(d-1)}{2} \right\rfloor$ erros, como provado em (MACWILLIAMS; SLOANE, 1977, pp. 10).

Um exemplo simples de um código detector de erros é fazer o uso do bit de paridade. O bit de paridade *par* é um bit adicionado aos dados indicando se o número de bits do dado com valor 1 é par ou ímpar. Se for ímpar, o bit de validade tem valor 1, caso contrário, 0. Tornando o bit de paridade igual a 1 quando o dado tem um número ímpar de bits 1 resulta num valor binário (considerando o dado e o bit de paridade) com uma quantidade par de bits com valor 1. Por exemplo, para o valor binário 0011100 o bit de paridade par concatenado a ele será 1, pois há um número ímpar de bits 1.

O bit de paridade é chamado de *detector* pois pode indicar somente a presença de um erro, sem a possibilidade de corrigi-lo. Um código que não é capaz de corrigir erros gera a necessidade de descartar os dados recebidos e transmiti-los novamente. Para evitar essas situações de retransmissão, um código clássico criado é o código de Hamming. Ele é apresentado na Seção 2.3.1 para auxiliar no entendimento de códigos lineares, como forma de demonstrar conceitos importantes e necessários para a compreensão dos códigos de Goppa (Seção 2.4), que são mais complexos.

2.3.1 Códigos de Hamming

Propostos em 1950 por Richard Hamming, os códigos de Hamming (HAMMING, 1950) formam uma família de códigos fáceis de codi-

ficar e decodificar e que são capazes de corrigir no máximo um único erro em uma mensagem recebida.

A correção de erros dos códigos de Hamming fazem uso da propriedade de que, em um código linear [n,k] com uma matriz de paridade H, a síndrome é a soma das colunas de H que correspondem às posições onde ocorreram erros (MACWILLIAMS; SLOANE, 1977). Ou seja, na decodificação de um vetor $y \in \mathbb{F}_2^n$, a síndrome de y é calculada. Se ocorreu erro em apenas um bit, ele pode ser corrigido pois o valor da síndrome indica a coluna de H na qual o valor de y está incorreto. Se houver mais de um erro, o decodificador recebe a soma das colunas e não consegue corrigi-los.

Portanto, um código binário de Hamming é um código binário (ou seja, um código sobre $\mathbb{F}_2$) de comprimento $n = 2^m - 1$, $m \geq 2$, com uma matriz de paridade H de tamanho $m \times 2^m - 1$ cujas colunas são os inteiros no intervalo $[1, 2^m - 1]$ escritos em representação binária(HUFFMAN; PLESS, 2010).

Por exemplo, em um código de Hamming com $n = 7$ e $m = 3$, uma possível matriz de paridade H poderia ter valores como

$$H = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}. \quad (2.4)$$

Note que as colunas são formadas pelos números no intervalo [1,7] escritos em binário.

Considerando que se tenha recebido a mensagem $x = (1, 1, 0, 1, 1, 1, 1)$, temos que $S(x) = Hx^T = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$. Como $S(x)$ representa o número 3 em binário, temos que o erro na mensagem está na posição 3. Assim, corrigindo este erro, é possível obter a mensagem original correta, que é $y = (1, 1, 1, 1, 1, 1, 1)$.

2.4 CÓDIGOS DE GOPPA

Os códigos de Goppa formam uma classe de códigos corretores de erros e foram introduzidos por V. D. Goppa em 1970 (GOPPA, 1970). Um polinômio de Goppa $g(z)$ é um polinômio de grau t cujos coeficientes estão em $GF(q^m)$, sendo q um número primo e m um inteiro, isto é,

$$g(z) = g_0 + g_1 z + \cdots + g_t z^t = \sum_{i=0}^{t} g_i z^i. \tag{2.5}$$

Seja L um subconjunto $L = \{\alpha_1, \ldots, \alpha_n\} \in GF(q^m)$ tais que $g(\alpha_i) \neq 0$ para todo $\alpha_i \in L$. Um código de Goppa $\Gamma(L, g(z))$ consiste de todos vetores c_i sobre $GF(q)$ que satisfaçam a condição

$$R_c(z) = \sum_{i=1}^{n} \frac{c_i}{z - \alpha_i} \equiv 0 \ (mod\ g(z)) \tag{2.6}$$

onde $\frac{1}{z-\alpha_i}$ é o único polinômio que satisfaz $(z - \alpha_i) \cdot \frac{1}{z-\alpha_i} \equiv 1 (mod\ g(z))$.

Pelo fato do código de Goppa ser um código linear, seus parâmetros consistem no comprimento n, dimensão k e distância mínima d. O comprimento n é o tamanho das palavras do código, o que nos códigos de Goppa é fixado por L. A dimensão k satisfaz $k \geq n - m \cdot t$ e a distância mínima d satisfaz $d \geq t + 1$ (HUFFMAN; PLESS, 2010, pp. 523).

2.4.1 Matriz de paridade de um código Goppa

Para a decodificação, é necessária uma matriz de paridade H de um código de Goppa $\Gamma(L, g(x))$ tal que $c = c_0 c_1 \ldots c_{n-1} \in \Gamma(L, g(x))$ se, e somente se, $Hc^T = \overrightarrow{0}$. Seja

$$p_i(x) \equiv \frac{1}{x - \alpha_i} \equiv -g(\alpha_i)^{-1} \cdot \frac{g(x) - (\alpha_i)}{x - \alpha_i} (mod\ g(x)). \tag{2.7}$$

Pela definição do códigos de Goppa, temos

$$\sum_{i=1}^{n} \frac{c_i}{x - \alpha_i} \equiv 0 \ (mod\ g(x)) \equiv \sum_{i=1}^{n} c_i p_{ij} = 0, \text{para } 1 \leq j \leq t \tag{2.8}$$

ou seja,

$$\sum_{i=1}^{n} c_i \cdot g(\alpha_i)^{-1} \cdot \frac{g(x) - (\alpha_i)}{x - \alpha_i} \equiv 0 (mod\ g(x)). \tag{2.9}$$

Como $g(x) = g_0 + g_1 x + \cdots + g_t x^t$ e considerando $h_i = g(\alpha_i)^{-1}$ temos

$$\begin{aligned} p_i(x) &= h_i \cdot \frac{g_1 \cdot (x - \alpha_i) + \cdots + g_t \cdot (x^t - \alpha_i^t)}{x - \alpha_i} \\ &= h_i \cdot \frac{g_1 \cdot (x - \alpha_i)}{x - \alpha_i} + \cdots + \frac{g_t \cdot (x^t - \alpha_i^t)}{x - \alpha_i} \\ &= h_i \cdot (g_1 + g_2 \cdot (x + \alpha_i) + \dots \\ &\quad + g_t \cdot (x^{t-1} + x^{t-2}\alpha_i + \cdots + \alpha_i^{t-1})). \end{aligned} \tag{2.10}$$

Substituindo $p_i(x) = p_{i1} + p_{i2}x + p_{i3}x^2 + \cdots + p_{it}x^{t-1}$, se obtêm as seguintes expressões para p_{ij} (JOCHEMSZ, 2002):

$$\begin{cases} p_{i1} = h_i \cdot ((g_t \alpha_i^{t-1} + g_{t-1}\alpha_i^{t-2} + \cdots + g_2 \alpha_i + g_1); \\ p_{i2} = h_i \cdot (g_t \alpha_i^{t-2} + g_{t-1}\alpha_i^{t-3} + \cdots + g_2); \\ \quad \vdots \\ p_{i(t-1)} = h_i \cdot (g_t \alpha_i + g_{t-1}); \\ p_{it} = h_i \cdot g_t. \end{cases} \tag{2.11}$$

Considerando as equações em (2.11), podemos construir a matriz de paridade H de um código Goppa como

$$H = \begin{pmatrix} h_1 \cdot g_t & h_2 \cdot g_t & \dots & h_n \cdot g_t \\ h_1 \cdot (g_t \alpha_1 + g_{t-1}) & h_2 \cdot (g_t \alpha_2 + g_{t-1}) & \dots & h_n \cdot (g_t \alpha_n + g_{t-1}) \\ \vdots & \vdots & \vdots & \vdots \\ h_1 \cdot \sum_{j=1}^{t} g_j \alpha_1^{j-1} & h_2 \cdot \sum_{j=1}^{t} g_j \alpha_2^{j-1} & \dots & h_n \cdot \sum_{j=1}^{t} g_j \alpha_n^{j-1} \end{pmatrix}. \tag{2.12}$$

A matriz H em (2.12) pode ser reduzida na matriz H' em (2.13) por meio de operações por linhas (HUFFMAN; PLESS, 2010).

$$H' = \begin{pmatrix} h_1 & h_2 & \dots & h_n \\ h_1 \cdot \alpha_1 & h_2 \cdot \alpha_2 & \dots & h_n \cdot \alpha_n \\ \vdots & \vdots & \vdots & \vdots \\ h_1 \cdot \alpha_1^{t-1} & h_2 \cdot \alpha_2^{t-1} & \dots & h_n \cdot \alpha_n^{t-1} \end{pmatrix}. \tag{2.13}$$

2.4.2 Matriz geradora de um código Goppa

A matriz geradora G é utilizada para a etapa de codificação de uma mensagem. Uma palavra c é obtida através da mensagem m e da matriz geradora G como $c = mG$. Temos que para toda palavra $c \in \Gamma(L, g(z))$, a igualdade $Hc^T = 0$ é verdadeira. Por consequência, pode-se obter a matriz G a partir da matriz de paridade H através da equação $GH^T = 0$, de forma que o espaço nulo (Seção 2.2.1) da matriz H forme o espaço linha (Seção 2.2.2) de G.

2.4.3 Codificação de um código Goppa

Para codificar uma mensagem utilizando um código de Goppa $\Gamma(L, g(z))$ cuja matriz geradora é G, basta dividi-la em blocos de k símbolos e multiplicar cada bloco por G.

$$(m_0, \ldots, m_k) \cdot G = (c_1, \ldots, c_n). \tag{2.14}$$

O vetor c resultante é uma palavra do código de Goppa $\Gamma(L, g(z))$. Como os código de Goppa são códigos corretores de erros, mesmo que alguns erros sejam adicionados à palavra c, a mensagem original m ainda pode ser recuperada na etapa de decodificação.

2.4.4 Decodificação de um código Goppa

Seja y uma mensagem recebida que contém r erros, sendo $r \leq \left\lfloor \frac{(d-1)}{2} \right\rfloor$. Então y é a mensagem original com a adição de alguns erros, ou seja,

$$(y_1, \ldots, y_n) = (m_1, \ldots, m_n) + (e_1, \ldots, e_n) \tag{2.15}$$

com $e_i \neq 0$ em r posições, ou, o peso w do vetor de erros é $w((e_1, \ldots, e_n)) = r$. Para obter a mensagem original m a partir de y é preciso encontrar o vetor de erros e. Para obter este vetor em códigos de Goppa binários, o algoritmo mais utilizado é o algoritmo de Patterson, descrito em mais detalhes na Seção 2.4.4.1.

A correção de uma mensagem com erros é ilustrada pela Figura 1 (SAFIEDDINE; DESMARAIS, 2014). Cada círculo representa um conjunto de mensagens com erros que são corrigidos para determinada palavra do código, representada pelo X ao centro. O raio do círculo, rotulado t,

indica o máximo de erros que um código pode corrigir e d representa a distância mínima entre cada par de palavras que pertencem ao código. A área cinza representa elementos que o algoritmo de decodificação não consegue corrigir.

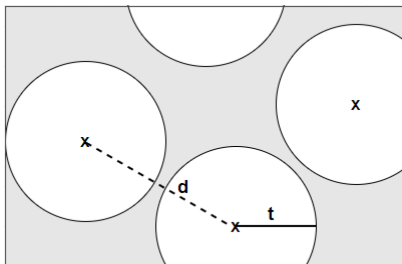

Figura 1 – Ilustração de como uma mensagem com erros pode ser corrigida

Qualquer mensagem com no máximo t erros dentro de um círculo será corrigido para a palavra mais próxima, ou seja, o centro do círculo.

2.4.4.1 Algoritmo de Patterson

O algoritmo de Patterson (PATTERSON, 1975) foi proposto por Nicholas Patterson em 1975 e é usado na etapa de decodificação de códigos Goppa binários, pois é capaz de corrigir t erros em uma dada palavra y.

Sendo $y = c + e$, para determinar o vetor de erros e adicionado a fim de corrigir a palavra e encontrar a mensagem original c, o algoritmo faz uso do cálculo de síndrome (Seção 2.3) e do polinômio localizador de erros (ou *error locator polynomial* (ELP)). O Algoritmo 1 descreve os passos que compõem o algoritmo de Patterson.

Algoritmo 1 Algoritmo de Patterson

Seja $y = (y_1, \ldots, y_n)$ uma *codeword* recebida que contém t erros e um código de Goppa $\Gamma(L, g(x))$.

1. Calcular o polinômio da síndrome $S(x)$ dado por

$$S(x) = \sum_{i=1}^{n} \frac{y_i}{x - L_i} \mod g(x)$$

2. Calcular o inverso $S(x)^{-1}$ do polinômio da síndrome $S(x)$ módulo $g(x)$. Para isto, pode-se usar o algoritmo de Euclides estendido(STALLINGS, 2014, pp.97).

3. Calcular a raiz quadrada do inverso do polinômio da síndrome $S(x)$ mais x no módulo $g(x)$ tal que

$$\tau(x) = \sqrt{S(x)^{-1} + x} \mod g(x)$$

O cálculo da raiz quadrada é detalhado em (RISSE, 2011) e em (SAFIEDDINE; DESMARAIS, 2014).

4. Determinar as partes par e ímpar do polinômio localizador de erros tal que

$$a(x) \equiv b(x)\tau(x) \mod g(x)$$

Para isto, pode-se usar o algoritmo de Euclides estendido.

5. Construir o polinômio localizador de erros $\sigma(x)$ tal que

$$\sigma(x) = a^2(x) + xb^2(x)$$

6. Determinar as raízes do polinômio localizador de erros $\sigma(x)$ e construir o vetor e. As raízes do polinômio $\sigma(x)$ correspondem às posições do erros (GOPPA, 1970).

$$e = (\sigma(\alpha_1), \ldots, \sigma(\alpha_n)) \oplus (1, \cdots, 1)$$

sendo que $\alpha_i \in L$.

7. A palavra original enviada c pode então ser obtida com $c = y - e$.

Para o passo 6, pode-se fazer uso de diferentes métodos para a

determinação de raízes em polinômios. Três métodos diferentes serão o foco deste trabalho e estão descritos em mais detalhe no Capítulo 4. São eles: análise exaustiva, algoritmo Trace de Berlekamp e o algoritmo de Fedorenko.

Estudos como (BUCERZAN et al., 2017) e (SHOUFAN et al., 2010) mostram como a etapa de determinação de raízes de polinômios pode ser vulnerável a ataques de temporização (Seção 3.3.2). Portanto, a análise dos três algoritmos supracitados será feita tendo como objetivo obter um método que possua um tempo de execução o mais constante possível, para que a segurança do criptossistema não seja prejudicada na implementação.

2.4.5 Exemplo de um código Goppa binário

Considerando $m = 3$ e $t = 2$, $g(X)$ deve ser um polinômio irredutível, então podemos ter $g(X) = Z^2 * X^2 + (Z^2 + 1) * X + Z^2 + 1$. Tendo $L = \{0, 1, Z, Z^2, Z + 1, Z^2 + Z, Z^2 + Z + 1, Z^2 + 1\}$, temos que a matriz de paridade deste código é

$$\mathrm{H} = \begin{pmatrix} Z & Z^2+Z+1 & Z^2+1 & Z^2+1 & Z^2+Z & Z & Z^2+Z+1 & Z^2+Z \\ 0 & Z^2+Z+1 & 1 & Z & 1 & Z^2+Z+1 & Z+1 & Z+1 \end{pmatrix}. \tag{2.16}$$

ou, em representação binária considerando, por exemplo, $Z + Z^2 = (0, 1, 1)^T$, $1 + Z^2 = (1, 0, 1)^T$, $1 + Z = (1, 1, 0)^T$.

$$H = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ -- & -- & -- & -- & -- & -- & -- & -- \\ 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}. \tag{2.17}$$

Portanto, o código Goppa $\Gamma(L, g(X))$ possui as palavras $C = \{00000000, 01011101$ que são as palavras que possuem síndrome 0. Se adicionarmos erros na palavra 11110110 na posição 4 e 7 teremos $11110110 \oplus 00010010 = 11100100$. Podemos encontrar a palavra original a partir da mensagem com erro 11100100 utilizando o algoritmo de Patterson.

Seguindo os passos do algoritmo de Patterson para $y = 11100100$ temos:

1. Calcular o polinômio $S(x) = \sum_{i=1}^{n} \frac{y_i}{x-L_i} \mod g(x)$.

$$\begin{aligned}
S(X) &= \sum_{i=1}^{n} \frac{y_i}{X - L_i} \mod g(X) \\
&= \frac{1}{X-0} + \frac{1}{X-1} + \frac{1}{X-Z} + \frac{1}{X-Z^2+Z} \\
&= ((Z+1)*X+1) \\
&+ (X+Z^2+Z+1) \\
&+ (Z*X+Z+1) \\
&+ ((Z+1)*X)
\end{aligned}$$

$$S(X) = (Z+1)*X+Z^2+1$$

2. Calcular $\tau(X) = \sqrt{S(X)^{-1}+X} \mod g(X)$.

$$\begin{aligned}
S(X)^{-1} &= (Z+1)*X+Z^2 \\
S(X)^{-1}+X &= (Z)*X+Z^2
\end{aligned}$$

$$\begin{aligned}
\tau(X) &= \sqrt{(Z)*X+Z^2} \mod g(X) \\
&= ((Z)*X+Z^2)^{2^{mt-1}} \mod g(X) \\
\tau(X) &= (Z^2)*X+Z^2
\end{aligned}$$

3. Encontrar $a(X)$ e $b(X)$ tal que $a(X) \equiv b(X)\tau(X) \mod g(X)$ sendo o grau do polinômio $a(X)$ menor ou igual a $\lfloor \frac{t}{2} \rfloor$ e o grau do polinômio $b(X)$ menor ou igual a $\lfloor \frac{t-1}{2} \rfloor$.

$$\begin{aligned}
a(X) &\equiv b(X)\tau(X) \mod g(X) \\
a(X) &\equiv b(X)((Z^2)*X+Z^2) \mod g(X) \\
(Z^2)*X+Z^2 &\equiv 1((Z^2)*X+Z^2) \mod g(X)
\end{aligned}$$

4. Calcular $\sigma(X) = a^2(X) + Xb^2(X)$.

$$\begin{aligned}\sigma(X) &= a^2(X) + Xb^2(X)\\ &= ((Z^2) * X + Z^2)^2 + X * 1^2\\ \sigma(X) &= (Z^2 + Z) * X^2 + X + Z^2 + Z\end{aligned}$$

5. Determinar as raízes de $\sigma(X)$ e construir o vetor de error e.

   Sendo $L = \{0, 1, Z, Z^2, Z+1, Z^2+Z, Z^2+Z+1, Z^2+1\}$, queremos verificar quais $L_i \in L$ resultam em zero quando aplicados em $\sigma(X)$, ou seja, $\sigma(L_i) = 0$. Temos que $\sigma(Z^2) = 0$ e $\sigma(Z^2+Z+1) = 0$, portanto, os erros estão nas posições 4 e 7 (da esquerda para a direita), o que resulta no vetor de erros $e = 00010010$.

6. Encontrar a palavra original $c = y - e$.

   Aplicando o vetor de erros e em y, temos

$$c = y - e = 11100100 \oplus 00010010$$

$$c = 11110110$$

3 CRIPTOSSISTEMA DE MCELIECE

Neste capítulo a definição de criptografia assimétrica é apresentada na Seção 3.1. A descrição do criptossistema de McEliece é feita na Seção 3.2 e em seguida são discutidos possíveis ataques ao criptossistema na Seção 3.3.1, incluindo ataques de canal lateral na Seção 3.3.2.

3.1 CRIPTOGRAFIA ASSIMÉTRICA

Criptografia assimétrica, ou criptografia de chave pública, se caracteriza pela presença de um par de chaves, uma chave pública e outra privada, que são utilizadas para realizar operações complementares, como cifragem e decifragem ou assinar arquivos digitalmente e verificar estas assinaturas. A chave pública pode ser distribuída livremente mas a chave privada deve ser conhecida apenas pela entidade que a gerou. Qualquer entidade pode utilizar a chave pública de outra para cifrar uma mensagem, que apenas poderá ser decifrada com a respectiva chave privada, ou seja, apenas a entidade que tem conhecimento da chave privada par da chave pública usada poderá ler a mensagem.

Este conceito surgiu em 1976 com a publicação de Whitfield Diffie e Martin Hellman (DIFFIE; HELLMAN, 1976) e era completamente diferente de tudo o que era utilizado em criptografia até o momento. Até então, fazia-se uso de apenas uma chave e os sistemas de criptografia se baseavam principalmente em substituição e permutação em contraste com os algoritmos de criptografia de chave pública, que se baseiam em funções matemáticas.

Um dos exemplos de algoritmos assimétricos mais conhecido é o RSA, criado por Ron Rivest, Adi Shamir e Len Adleman em 1978 (RIVEST; SHAMIR; ADLEMAN, 1978). O algoritmo de McEliece (MCELIECE, 1978) é também um exemplo e surgiu na mesma época.

3.2 CRIPTOSSISTEMA DE MCELIECE

Robert McEliece publicou em 1978 (MCELIECE, 1978) um criptossistema de chave pública que gera suas chaves pública e privada a partir de um código linear de correção de erros. O código proposto originalmente foi o código de Goppa. O sistema possui cifragem e deci-

fragem eficientes e se mantém resistente a ataques feitos nos últimos 40 anos como os estudados em (BERNSTEIN; LANGE; PETERS, 2008), onde parâmetros que provêm segurança contra os mais diversos ataques foram propostos. Um dos principais aspectos negativos do esquema é a grande quantidade de bytes que a chave pública ocupa.

O criptossistema proposto por McEliece pode ser visto como um conjunto de três etapas necessárias para a realização de troca de mensagens: geração das chaves, cifragem e decifragem. Na etapa de geração de chaves são geradas as chaves pública e privada que depois podem ser utilizadas para a cifragem e decifragem de mensagens.

3.2.1 Geração do par de chaves

No algoritmo original é utilizado código de Goppa para construir as chaves pública e privada. Inicialmente, é selecionado um polinômio de Goppa $g(z)$ de grau t cujos coeficientes estão em $GF(2^m)$. Depois é definida a matriz [k,n] geradora G que cria o código de Goppa C caracterizado por $g(z)$ e por L. Em seguida, são selecionas uma matriz $k \times k$ inversível S e uma matriz de permutação P de dimensões $n \times n$. Uma matriz de permutação é uma matriz formada apenas por zeros e uns, obtida através da permutação das linhas de uma matriz identidade, sendo que ao fim, há apenas um elemento 1 por coluna.

Tendo gerados as matrizes G, S e P, elas são multiplicadas de modo a gerar a matriz $G' = SGP$, que fará parte da chave pública $PU = (G', t)$. A chave privada consistirá em $PR = (g(z), S, G, P)$.

3.2.2 Cifragem de uma mensagem

A cifragem de mensagens no criptossistema de McEliece faz uso da ideia de adicionar erros propositalmente às mensagens. Consideremos que Alice e Bob desejam se comunicar utilizando o criptossistema de McEliece. Após a publicação da chave pública PU_A de Alice, Bob pode enviar uma mensagem m a ela. Para isso, Bob gera um vetor binário aleatório e de tamanho n com peso $w(e) \leq t$ e calcula $c = mG' + e$, que é o texto cifrado que será enviado à Alice.

3.2.3 Decifragem de uma mensagem

Ao receber a mensagem c que foi cifrada utilizando sua chave pública, Alice pode usar sua chave privada para decifrá-la da seguinte maneira: calcula-se $c' = cP^{-1}$, sendo P^{-1} a matriz inversa da matriz de permutação P. Sendo

$$c' = cP^{-1} = mG'P^{-1} + eP^{-1} = mSGPP^{-1} + eP^{-1} = (mS)G + e', \tag{3.1}$$

c' representa uma *codeword* do código de Goppa escolhido anteriormente, pois é uma mensagem mS multiplicada à matriz geradora G do código e com t erros adicionados, os quais a decodificação pode eliminar (pelo fato de o código de Goppa ser um código corretor de erros).

Portanto, Alice pode utilizar o algoritmo de decodificação do código sobre c' e obter $c' = mSG$. A partir daqui é possível obter a mensagem original m multiplicando c' por $(SG)^{-1}$ ou encontrando $m' = mS$ através de redução $[G^T|(mS)^T]$, como detalhado em (JOCHEMSZ, 2002), e em seguida encontrar $m = m'S^{-1}$, já que a matriz S é inversível.

3.2.4 Exemplo de cifragem e decifragem de uma mensagem com McEliece

Considere que o código Goppa utilizado seja o dado no exemplo da Seção 2.4.5. Para que uma entidade A envie uma mensagem m para B fazendo uso do sistema de McEliece são necessários três passos:

1. Geração do par de chaves

   Temos que

   $g(X) = Z^2 * X^2 + (Z^2 + 1) * X + Z^2 + 1$,

   $t = 2$ e

   $$G = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & Z & Z \\ 0 & 1 & 0 & 0 & 0 & 0 & Z & Z^2+Z \\ 0 & 0 & 1 & 0 & 0 & 0 & Z^2+Z+1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & Z^2+Z \\ 0 & 0 & 0 & 0 & 1 & 0 & Z^2 & Z \\ 0 & 0 & 0 & 0 & 0 & 1 & Z^2+Z+1 & Z+1 \end{pmatrix}.$$

   Depois da definição do código de Goppa, são criadas as matrizes S e P. S deve ser uma matriz inversível e P uma matriz de

permutação. Consideremos

$$S = \begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 \end{pmatrix}$$

e

$$P = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

.

Em seguida, é calculada a matriz $G' = SGP$.

$$G' = \begin{pmatrix} 1 & 0 & 1 & Z+1 & 1 & 0 & 1 & Z^2+Z \\ 0 & 0 & 0 & Z^2 & 1 & 1 & 1 & Z+1 \\ 1 & 1 & 1 & Z^2 & 0 & 1 & 1 & Z^2 \\ 1 & 1 & 1 & Z^2+1 & 1 & 1 & 1 & Z \\ 1 & 1 & 0 & Z+1 & 0 & 1 & 1 & Z^2+1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & Z+1 \end{pmatrix}$$

Desta forma, temos a chave pública $PU = (G', t)$ e chave privada $PR = (g(X), S, G, P)$.

2. Cifragem de uma mensagem

   Para uma entidade A enviar uma mensagem m para a entidade B, A pode fazer uso da chave pública de B $PU_B = (G', t)$.

   Consideremos que A deseja enviar a mensagem $m = 010100$. Basta que A crie um vetor de erros e, sendo $w(e) \leq t$, e calcule $c = mG' + e$.

   Como $t = 2$, consideremos o vetor de erros como $e = 00110000$. Calculando $c' = mG'$ temos $c' = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{pmatrix}$.

Para obter a mensagem cifrada c, temos

$$\begin{aligned} c &= c' + e \\ &= 11110001 \oplus 00110000 \\ c &= 11000001 \end{aligned}$$

3. Decifragem de uma mensagem

   Para que B decodifique a mensagem c ao recebê-la de A, é necessário que utilize sua chave privada $PR_B = (g(X), S, G, P)$.

   Primeiro, é calculado

$$\begin{aligned} c' &= cP^{-1} \\ &= 10001001 \end{aligned}$$

   Em seguida, é aplicado o algoritmo de Patterson (Seção 2.4.4.1) para decodificar c'. Como resultado, temos

$$\begin{aligned} c' &= mSG \\ &= 10101011 \end{aligned}$$

   Como a matriz S é inversível, podemos calcular $m' = mSG(SG)^{-1}$ e, assim, obter a mensagem original m.

$$\begin{aligned} m' &= c'S^{-1} \\ &= mSG(SG)^{-1} \\ m' &= m \\ &= 010100 \end{aligned}$$

   Desta maneira, B consegue identificar a mensagem original enviada por A.

3.3 ATAQUES AO CRIPTOSSISTEMA DE MCELIECE

Segundo Loidreau, a segurança do sistema depende da intratabilidade de decodificar um código aleatório até a sua capacidade de correção de erros (LOIDREAU, 2000). Isto se deve ao fato de que a chave pública gerada pelo esquema de McEliece é indistinguível de uma matriz aleatória (SENDRIER et al., 2009).

Para decodificar um código aleatório sem conhecimento da chave privada um atacante se depara com o problema de decodificação geral (*general decoding problem*) que consiste em, tendo uma matriz geradora G com elementos sobre $\mathbb{F}_q^{k \times n}$, um vetor c sobre $\mathbb{F}_q^n$ e um número inteiro $w < n$, determinar se existe um vetor m sobre $\mathbb{F}_q^k$ tal que $e = c - mG$ tenha peso $w(e) \leq w$. Isto é, o atacante deve encontrar a palavra mais próxima a um dato vetor em um código linear C, considerando que exista uma única palavra que satisfaça essa condição (BOLLAUF, 2015).

Outros algoritmos de criptografia assimétrica comumente se baseiam em dois problemas da teoria dos números, a dificuldade de fatorar um inteiro e a dificuldade de calcular um logaritmo discreto, ambos problemas que se tornam tratáveis com a computação quântica, como demonstrado pelo algoritmo de Shor (SHOR, 1999).

Considerando ataques feitos por um hipotético computador quântico, o problema da decodificação de um código continuaria não podendo ser resolvido em tempo polinomial (BERNSTEIN; BUCHMANN; DAHMEN, 2009), o que torna o esquema de McEliece um sistema seguro mesmo neste cenário. Entretanto, não são apenas ataques de computadores quânticos que um criptossistema seguro deve resistir. Nas seções 3.3.1 e 3.3.2 são apresentados outros ataques que podem ser realizados e como o esquema de McEliece pode ser resistente a eles.

3.3.1 Ataques estruturais e de decodificação

Há duas formas principais de ataque do sistema segundo Loidreau (LOIDREAU, 2000), que se baseiam em dois problemas diferentes:

i. Uma maneira de ataque consiste em reconstruir o decodificador para o código gerado pela chave pública G' analisando sua estrutura, o que é chamado de ataque estrutural. Se um ataque como este fosse bem sucedido, o atacante conseguiria revelar a chave privada G, o que faria todo o criptossistema quebrar.

   Códigos podem ser separados em classes de códigos equivalentes. Dois códigos de comprimento n equivalentes são códigos para os quais existem uma permutação das n posições que transforma um código no outro. O código C', gerado por G' é equivalente ao código gerado por G. O atacante teria que comparar um código de cada classe e C', de forma a tentar encontrar um código que pertença à mesma classe de equivalência que C'. Como as classes de equivalência têm uma cardinalidade baixa, este processo demanda muito

poder computacional, visto que, considerando o uso dos parâmetros originais, $n = 1024$ e $t = 50$, este ataque precisaria analisar certa de 2^{466} códigos (AU; EUBANKS-TURNER; EVERSON, 2003).

ii. Uma segunda maneira se consiste em decodificar textos cifrados m que foram interceptados, o que é chamado de ataque de decodificação. Como o código C' gerado pela chave pública e o código C gerado pela chave privada são equivalentes, ambos têm a capacidade de corrigir t erros. O custo desse ataque depende apenas dos parâmetros de C', ou seja, o seu comprimento, dimensão e capacidade de correção. Isto significa que os parâmetros do sistema devem ser escolhidos para que este tipo de ataque seja dificultado.

    Considerando o uso dos parâmetros originais, $n = 1024$, $k = 524$ e $t = 50$, este ataque demandaria 2^{64} operações binárias (LOIDREAU, 2000). A medida que o poder computacional aumenta, o sistema de McEliece como proposto originalmente se torna mais inseguro. Entretanto, aumentar os valores dos parâmetros não é necessariamente a melhor solução, pois o tamanho da chave se torna proibitivo. Por exemplo, para atingir a segurança de 128 bits (isto é, que ataques precisem de pelo menos $2^1$28 operações), o sistema precisaria dos parâmetros $n = 4096$, $k = 3376$ e $t = 60$, o que resultaria em uma chave pública de 303.840 bytes, se a chave estiver em forma sistemática, onde há uma matriz identidade $k \times k$ contida na matriz que forma a chave (SENDRIER, 2017).

Estes ataques tentam se aproveitar de falhas teóricas e são feitos sobre a estrutura do criptossistema, mas outros ataques podem explorar vulnerabilidades que só se tornam aparentes em implementações, como, por exemplo, ataques de canal lateral.

3.3.2 Ataques de temporização

Outros ataques explorados no criptossistema de McEliece são os chamados ataques de canal lateral (*side-channel attacks*), que se baseiam em informações além da mensagem original e da mensagem cifrada no processo de cifragem ou decifragem. Estas informações adicionais podem ser, por exemplo, estatísticas de consumo de energia ou informações em relação ao tempo que as operações duram.

Os ataques propostos em (STRENZKE et al., 2008) e (STRENZKE, 2011) são ataques *side-channel* que utilizam informação de tempo, conhecidos como ataques de temporização (*timing attacks*). Eles demons-

tram como a variação do tempo entre diferentes execuções de uma mesma tarefa podem revelar informações importantes sobre a chave privada utilizada.

Em (BUCERZAN et al., 2017) e (SHOUFAN et al., 2010) são descritos ataques de temporização sobre a etapa de determinação das raízes do polinômio localizador de erros na decodificação de um código Goppa (Seção 2.4.4.1). O propósito destes ataques é o de recuperar a mensagem original através da obtenção do polinômio localizador de erros $\sigma(x)$. Isto somente é possível se o tempo de execução da decodificação variar de acordo com a quantidade de raízes do polinômio.

Um ataque possível explora a diferença no tempo de execução da determinação de raízes entre a decodificação de uma mensagem que contém t erros e outra que contém $w > t$ erros, sendo t a quantidade de erros que o código Goppa utilizado pode corrigir. Quando uma mensagem contém uma quantidade de erros w menor ou igual a t, o polinômio $\sigma(x)$ contém w raízes, mas quando a quantidade de erros w é maior que t, $\sigma(x)$ contém apenas uma fração de w raízes (STRENZKE, 2012).

Com esta informação o ataque pode ser feito da seguinte maneira: tendo em mãos um texto cifrado c, o atacante pode inverter um bit de c e obter c'. Ao observar o tempo de execução de c' é possível inferir se c' possui $t-1$ ou $t+1$ erros, já que uma mensagem com $t+1$ erros possui um $\sigma(x)$ com menos raízes, o que poderia acarretar em uma execução mais rápida. Realizando isto para todos os bits de c, o atacante pode descobrir quais as posições que contém erros.

Em uma implementação que realize a avaliação do polinômio localizador de erros em tempo constante independentemente da quantidade de raízes do polinômio, os ataques de temporização supracitados não têm chance de suceder. Portanto, no Capítulo 4 serão apresentados diferentes algoritmos que podem ser utilizados na análise do polinômio localizador de erros que posteriormente serão avaliados em relação à sua variabilidade de tempo de execução, analisando se os mesmos podem trazer alguma vulnerabilidade ao criptossistema levando em consideração ataques de temporização, ou seja, analisar se a implementação do algoritmo possui tempo de execução constante para a decodificação de diferentes mensagens.

3.3.3 Variações do sistema

Como forma de manter a segurança do sistema considerando o ataque de decodificação citado anteriormente, no estado da arte é possível encontrar variações que substituem os códigos de Goppa utilizados originalmente por outras famílias de código. Em (GABORIT, 2005), o uso de códigos *quasi-cyclic* é proposto como forma de diminuir o tamanho da chave. Em (BALDI; CHIARALUCE, 2007), é proposto o uso de códigos *quasi-cyclic* com matriz matriz de paridade de baixa densidade (QC-LDPC). Já em (MISOCZKI; BARRETO, 2009), a classe de códigos indicada é a de códigos Goppa *quasi-dyadic*. Outra proposta consiste na apresentação de variantes do criptossistema de McEliece que fazem uso de códigos com matrizes de paridade de densidade moderada (MDPC)(MISOCZKI et al., 2013).

4 ALGORITMOS PARA DETERMINAÇÃO DE RAÍZES DE POLINÔMIOS

Neste capítulo serão descritos os três algoritmos usados para determinação de raízes de polinômios em corpos finitos que são foco deste trabalho. Estes algoritmos são utilizados no sexto passo do algoritmo de Patterson, detalhado na Seção 2.4.4.1. Na Seção 4.1, a avaliação exaustiva é descrita, seguida pela descrição do algoritmo Berlekamp Trace na Seção 4.2 e, por fim, o algoritmo descrito por Sergei Fedorenko é detalhado na Seção 4.3.

4.1 AVALIAÇÃO EXAUSTIVA

A maneira mais simples para encontrar as raízes do polinômio localizador de erros $\sigma(x)$ é testando todos os elementos do código. Todos os elementos X_i (exceto zero) de um corpo finito $GF(q)$ pode ser descrito como α^i para $0 \leq i < q$, onde α é um elemento primitivo de $GF(q)$ (HUFFMAN; PLESS, 2010).

Desta maneira, a busca pelas raízes de $\sigma(x)$ pode ser feita avaliando o resultado do polinômio localizador de erros para cada X_i. Todos os X_i tal que $\sigma(X_i) = 0$ são raízes do polinômio e indicam as posições dos erros da palavra sendo analisada.

4.2 ALGORITMO TRACE DE BERLEKAMP

O algoritmo Trace de Berlekamp (*Berlekamp Trace Algorithm* (BTA), em inglês) é um algoritmo recursivo proposto por Elwyn Berlekamp em 1970 (BERLEKAMP, 1970) que determina as raízes de um polinômio através da sua fatoração. Por exemplo, fatorando o polinômio $p(x) = x^2 - 4$ temos $p(x) = (x - 2)(x + 2)$, onde podemos ver facilmente que a raiz de $(x - 2)$ é 2 e a raiz de $(x + 2)$ é -2. Portanto, fatorando o polinômio $p(x)$ conseguimos encontrar suas raízes 2 e -2.

O algoritmo se baseia nas propriedades da função *trace*, definida como $Tr(x) = x^{2^0} + x^{2^1} + x^{2^2} + \cdots + x^{2^{m-1}}$. Uma propriedade importante da função *trace* é que , sendo $B = \{\beta_1, \ldots, \beta_m\}$ uma base de $\mathbb{F}_{2^m}$, todo elemento $\alpha \in \mathbb{F}_{2^m}$ é representado unicamente pela m-tupla $(Tr(\beta_1 \cdot \alpha), \ldots, Tr(\beta_m \cdot \alpha))$. Com isso, qualquer polinômio $\sigma(z)$ sobre $\mathbb{F}_{2^m}$ que divide $z^{2^m} - z$ é fatorado em dois outros polinômios

$g(z) = mdc(\sigma(z), Tr(\beta \cdot \alpha))$ e $h(z) = mdc(\sigma(z), 1 + Tr(\beta \cdot \alpha))$, onde $mdc(x, y)$ indica o máximo divisor comum entre x e y. Se β iterar por B e a fatoração for aplicada recursivamente para $g(z)$ e $h(z)$, o polinômio $\sigma(z)$ pode ser dividido em fatores lineares, o que pode ser utilizado para encontrar suas raízes.

O Algoritmo 2 descreve o algoritmo Trace de Berlekamp recursivo.

Algoritmo 2 *Berlekamp Trace Algorithm* $BTA(p, i)$

Seja $p(x)$ o polinômio que se deseja encontrar as raízes e i um inteiro.

1. Se $p(x)$ tem grau menor ou igual a 1, retorne a raiz de $p(x)$.
2. Calcule $p_0(x) = \text{mdc}(p(x), \text{Tr}(\beta_i, x))$.
3. Calcule $p_1(x) = \text{mdc}(p(x), 1 + \text{Tr}(\beta_i, x))$.
4. Retorne $BTA(p_0(x), i + 1) \cup BTA(p_1(x), i + 1)$.

4.3 ALGORITMO DE FEDORENKO

Outra maneira de determinar as raízes de um polinômio é decompondo-o em polinômios lineares. Esta técnica é interessante pois a análise exaustiva de um polinômio linearizado (*linearized polynomial*) tem menor custo quanto comparado com a análise exaustiva citada na Seção 4.1 (FEDORENKO; TRIFONOV, 2002).

Um polinômio p na forma

$$p(y) = \sum_i p_i y^{q^i}, \; p_i \in \mathbb{F}_{q^m}.$$

é chamado de polinômio linearizado (*linearised polynomial*) ou q-polinômio sobre $\mathbb{F}_{q^m}$.

Segundo as propriedades de polinômios linearizados e polinômios afim (*affine polynomial*), descritas em (FEDORENKO; TRIFONOV, 2002), um polinômio afim $A(y) = p(y) + \beta, \beta \in \mathbb{F}_{2^m}$ pode ser avaliado como $A(y_i) = A(y_{i-1}) + p(\alpha^{\delta(y_i, y^{i-1})})$ para todos os pontos $y_i \in \mathbb{F}_{2^m}$, onde $\delta(y_i, y^{i-1})$ indica a posição na qual y_i e y_{i-1} têm valores diferentes quando todos os elementos estão ordenados em código Gray e $\{\alpha^0, \alpha^1, \ldots, \alpha^{m-1}\}$ é uma uma base de $\mathbb{F}_{2^m}$. Esta propriedade torna possível que a avaliação de um polinômio afim seja feita com apenas

uma adição a mais a cada $y_i \in \mathbb{F}_{2^m}$.

O Algoritmo 3 descreve o algoritmo proposto em (FEDORENKO; TRIFONOV, 2002) para a determinação de raízes.

Algoritmo 3 Algoritmo de Fedorenko

Seja $P(x) = \sum_{j=0}^{t} f_j x^j$ com $f_j \in \mathbb{F}_{2^m}$ o polinômio que se deseja encontrar as raízes, $\{\alpha^0, \alpha^1, \ldots, \alpha^{m-1}\}$ uma base de $\mathbb{F}_{2^m}$ e $\delta(y_i, y^{i-1})$ determina em qual posição y_i tem um bit de valor diferente de y_{i-1}.

1. Calcular $p_i^k = L_i(\alpha^k)$, $k = [0; m-1]$, $i = [0; \lceil \frac{t-4}{5} \rceil]$, onde L_i é um polinômio linearizado decomposto como $L_i(x) = \sum_{j=0}^{t} f_{5i+2^j} x^{2^j}$.
2. Inicializar $A_i^0 = f_{5i}$.
3. Representar cada $x_j \in \mathbb{F}_{2^m}$, $j = [0; 2^m - 1]$ como um elemento de código Gray, com $x_0 = 0$.
4. Calcular $A_i^j = A_i^{j-1} + L_i^{\delta(x_i, x^{i-1})}$, $j = [1; 2^m - 1]$.
5. Calcular $F(x_j) = f_3 x_j^3 + \sum_{i=0}^{\frac{t-4}{5}} x_j^{5i} + A_i^j$, com $j = [1; 2^m - 1]$ e $F(0) = f_0$.

   Se $F(x_i) = 0$, então x_i é uma raiz do polinômio $P(x)$.

---

5 COMPARAÇÃO ENTRE OS DIFERENTES MÉTODOS PARA DETERMINAÇÃO DE RAÍZES DE POLINÔMIOS

Neste capítulo serão comparadas as variações entre os tempos de execução da implementação de cada algoritmo descrito no Capítulo 4. A Seção 5.1 apresenta os detalhes das implementações feitas e na Seção 5.2 serão feitas as comparações e observações sobre os tempos de execução apresentados por cada algoritmo.

5.1 IMPLEMENTAÇÃO DOS ALGORITMOS

Todas as implementações realizadas para este trabalho foram feitas com SageMath[1], um software matemático livre e open-source que utiliza a linguagem Python e permite a programação com corpos finitos, anéis e polinômios.

Foram produzidos código-fonte para a codificação e decodificação de códigos de Goppa (Anexo A) baseado nos trabalhos de (RISSE, 2011) (ROERING, 2013) (CODE..., 2014). A decodificação faz uso do algoritmo de Patterson, cujo passo de determinação das raízes do polinômio localizador de erros foi implementado com três algoritmos diferentes, de acordo com o que foi detalhado no Capítulo 4. O código-fonte feito para a avaliação exaustiva (Seção 4.1) se encontra no Anexo A, para o algoritmo Trace de Berlekamp (Seção 4.2) se encontra no Anexo B e para o algoritmo proposto por Fedorenko (Seção 4.3), no Anexo C.

5.1.1 Algoritmo Trace de Berlekamp

O algoritmo Trace de Berlekamp se encontra em duas maneiras: uma implementação recursiva e uma iterativa desenvolvida durante este trabalho. A versão recursiva foi implementada de acordo com o algoritmo original descrito na Seção 4.2 e tem um tempo de execução variável pois a quantidade de chamadas recursivas depende do maior divisor comum e da quantidade de raízes que o polinômio possui.

A execução do BTA pode ser vista como uma árvore, como mostra a Figura 2. As folhas são as chamadas da função para polinômios de grau 1, cujas raízes são trivialmente calculadas e retornadas para a

[1] http://www.sagemath.org/

chamada um nível acima do seu. Isto ocorre até que todas as raízes sejam retornadas para a raiz da árvore, ou seja, a chamada inicial do algoritmo. Na decodificação de um código Goppa, a altura da árvore que representa sua execução está ligada ao maior divisor comum entre o polinômio p (inicialmente o polinômio localizador de erros) e a função Trace.

$BTA(x^n + \ldots, 1)$

$BTA(x^{n-k} + \ldots, 2)$ $BTA(x^k + \ldots, 2)$

$BTA(x + \ldots, i_1)$ $BTA(x + \ldots, i_3)$

$BTA(x + \ldots, i_2)$

$BTA(x + \ldots, i_4)$

Figura 2 – Ilustração da execução do algoritmo Trace de Berlekamp recursivo

É importante notar na execução ilustrada na Figura 2 que cada ramificação da árvore possui uma quantidade de chamadas recursivas diferente. Isto se deve ao fato de que o maior divisor comum é feito entre o polinômio recebido por parâmetro e o polinômio resultante da função Trace aplicada em um elemento β_i da base de $\mathbb{F}_{2^m}$. Logo, conforme o β_i definido em uma chamada recursiva $r_0 = BTA(x^a + \ldots, i)$, por exemplo, pode ocorrer de esta chamada gerar outras duas chamadas $r_1 = BTA(x^{a-1} + \ldots, i + 1)$ e $r_2 = BTA(x + \ldots, i + 1)$, onde o ramo gerado por r_1 será mais longo que o de r_2.

Desta forma, a altura da árvore de execução da decodificação de uma mensagem de um código Goppa que corrige t erros pode variar de $\log_2 n$ no melhor caso, caso o polinômio p seja sempre divido ao meio, até n no pior caso, caso p seja sempre diminuído em 1.

Por causa disto foi desenvolvida uma versão do algoritmo de Berlekamp modificada. Esta versão é iterativa, criada com o objetivo de remover as recursões e, juntamente com outras modificações, conseguir um tempo de execução mais constante e independente da quantidade

de raízes do polinômio.

5.1.1.1 Algoritmo de Berlekamp modificado

A maior fonte de variação de tempo no código iterativo é a iteração pela lista. Somente transformando a execução recursiva em iterativa mantém o mesmo problema com a árvore gerada pela execução. Para remover este problema e executar o laço uma quantidade de vezes constante, decidiu-se por tentar encontrar a melhor divisão possível para o polinômio p, ou seja, procurar dividir o polinômio sempre ao meio. Para tal, é calculado o maior divisor comum p_0 entre p e a função Trace para todos os β_i da base e a próxima iteração será feita utilizando o p_0 que tenha o grau mais próximo da metade do grau do polinômio p. Como o polinômio é dividido sempre ao meio, é possível fixar a quantidade de iterações do algoritmo para $2^{\lceil \log_2 t \rceil + 1} - 1$. O Algoritmo 4 descreve o procedimento.

Algoritmo 4 *Berlekamp Trace Algorithm* adaptado $BTA_adapt(p, t)$

Seja $p(x)$ o polinômio que se deseja encontrar as raízes e t a quantidade de erros que é possível corrigir.

1. Insira $p(x)$ na lista L
2. Enquanto $i < 2^{\lceil \log_2 t \rceil + 1} - 1$ faça
3. f recebe a cabeça da lista L
4. Se f tem grau menor ou igual a 1, retorne a raiz de f.
5. Calcule $\text{mdc}(f, \text{Tr}(\beta_i, x))$ para cada $\beta_i \in B$. Seja $p_0(x) = \text{mdc}(f, \text{Tr}(\beta_k, x))$ o polinômio com grau mais próximo da metade do grau de f.
6. Calcule $p_1(x) = \text{mdc}(f, 1 + \text{Tr}(\beta_k, x))$.
7. Insira $p_0(x)$ e $p_1(x)$ na lista L.

5.1.2 Algoritmo de Fedorenko

O código-fonte escrito para o algoritmo de Fedorenko possui pequenos detalhes feitos para que seu tempo de execução fosse mais constante, como a adição de computações de peso equivalente em cada avaliação condicional, seja ela verdadeira ou falsa. Por exemplo, caso o resultado dos cálculos feitos no passo 4 do Algoritmo 3 seja igual a 0, uma raiz do polinômio foi encontrada e ela é adicionada à lista de raízes. Na implementação feita, quando o resultado não é igual a 0, este valor é adicionado à uma lista auxiliar para que todas as iterações do laço tenham custo computacional similar e uma grande variação no tempo de execução do algoritmo seja evitada.

O algoritmo de Fedorenko não possui tempo de execução diretamente ligado a quantidade de raízes do polinômio sendo avaliado. Sendo assim, nenhuma modificação visando melhoria neste aspecto foi feita.

5.2 ANÁLISE DOS TEMPOS OBTIDOS

A Tabela 1 mostra a média e o desvio padrão dos tempos de execução de cada algoritmo (em minutos) para um código Goppa $\Gamma(L, g(z))$ onde os elementos de L e os coeficientes de $g(z)$ estão em $GF(2^{12})$ e $g(z)$ possui grau $t = 32$.

Foram feitas medições para 10 mensagens diferentes, 5 mensagens com t erros e 5 com $t + 1$ erros. Para cada mensagem foram medidos os tempos de 10 repetições da execução de cada algoritmo. Os valores foram coletados em um computador com processador Intel® Core i5-8400 com 2.80 GHz de frequência e 8GB de memória RAM e SageMath na versão 8.6. Para tal, foi feito uso da função *sage_timeit()*[2] e por ser uma medição em software, não muito precisa se comparada a uma medição em ciclos, por exemplo, foi considerado para a análise do resultado final apenas o valor mínimo de cada 10 repetições. Isto se deve ao fato de que o valor mínimo indica um limite inferior do quão rápido a máquina pode executar o algoritmo e valores acima disto são geralmente causados por outros processos interferindo na execução. Portanto, a média e o desvio padrão foram feitos a partir dos tempos mínimos da decodificação de cada mensagem.

[2] `http://doc.sagemath.org/html/en/reference/misc/sage/misc/sage_timeit.html`

	Tempo médio para t erros (em segundos)	**Desvio padrão para t erros**	**Tempo médio para $t+1$ erros (em segundos)**	**Desvio padrão para $t+1$ erros**
Análise exaustiva	1.013617	0.017623	0.997102	0.005291
BTA recursivo	3.275600	0.256375	0.149524	0.014667
BTA adaptado	26.972274	0.189169	36.443187	0.420316
Algoritmo de Fedorenko	3.737185	0.023487	3.674740	0.035016

Tabela 1 – Tempo de execução dos algoritmos para determinação de raízes de polinômios

Como dito na Seção 3.3.2, um bom algoritmo contra ataques de temporização deve executar em tempo constante independentemente da quantas raízes o polinômio sendo avaliado possui. Através dos resultados temos que a análise exaustiva foi uma boa alternativa em questão de tempo médio e desvio padrão, mas em parte isto se deve aos parâmetros pequenos utilizados para o teste. Como detalhado em (STRENZKE, 2012), a abordagem exaustiva não é uma boa alternativa para algoritmo de determinação de raízes em criptografia baseada em códigos, pois apresenta outros tipos de vulnerabilidades.

Os resultados para a versão original recursiva do algoritmos Trace de Berlekamp comprovam o problema de variabilidade no seu tempo de execução, por ser influenciado pela quantidade de raízes do polinômio. Pela diferença entre os tempos obtidos para mensagens com t e $t+1$, um atacante teria sucesso em um ataque de temporização e conseguiria obter as posições dos erros apenas pela avaliação da variabilidade do tempo de execução.

Observando os valores obtidos, vemos que a nova versão criada para o BTA (chamada de BTA adaptado) não se comportou como previsto e não atingiu a expectativa de ser um algoritmo com execução em tempo constante. Os resultados mostram uma diferença de tempo significativa para decodificações de mensagens com quantidade de erros maior que t, o que torna a implementação vulnerável a ataques

side-channel de temporização.

Um outro ponto negativo da nova versão do BTA foi o seu tempo de execução mais lento quando comparado com o original, o que se deve pelo fato de que as modificações implicam em manter mais variáveis de controle e mais operações em listas, que acaba sendo custoso. Um outro fator que pode ter influenciado no tempo de execução foi que nas situações em que a mensagem continha mais do que o t máximo de erros que podiam ser corrigidos, a fatoração era feita em menos passos do que o imposto pelo laço criado. Para continuar a execução do laço com o intuito de que, independente da quantidade de raízes, a mesma quantidade de iterações fosse feita, foi decidido por seguir a execução começando a fatoração novamente, ou seja, o polinômio localizador de erros é fatorado mais uma vez e os cálculos são realizados até que as iterações acabem.

Uma outra variação que pode ser feita no algoritmos Trace de Berlekamp é utilizar o algoritmo proposto por Zinoviev (ZINOVIEV, 1996) para determinar as raízes de polinômios com grau $d \leq 10$, que pode reduzir o tempo de execução, como analisado em (STRENZKE, 2012).

Por fim, o algoritmo de Fedorenko se mostra como uma boa alternativa entre as quatro analisadas, pois apresentou tempos pequenos e mais constantes, que não foram influenciados pela quantidade de raízes do polinômio. Apesar de nestes resultados a análise exaustiva tenha exibido tempos menores, segundo (FEDORENKO; TRIFONOV, 2002), dependendo da forma como as operações sobre corpos finitos são implementadas, o algoritmo de Fedorenko pode ser mais rápido que a busca de Chien (CHIEN, 1964), uma adaptação da análise exaustiva.

6 CONCLUSÃO

Neste trabalho, foram discutidos algoritmos para determinação de raízes de polinômios com o intuito de determinar sua vulnerabilidade em relação a ataques de temporização. Entre conceitos apresentados temos o criptossistema de McEliece, códigos de Goppa e seu principal método para realizar a decodificação de mensagens, o algoritmo de Patterson.

Entre os passos do algoritmo de Patterson temos a avaliação de um polinômio, a fim de encontrar suas raízes, que é o propósito dos algoritmos estudados no Capítulo 4. Pelo fato destes algoritmos serem utilizados no criptossistema de McEliece, considerado uma alternativa para ser utilizado em ambientes de alta segurança, há a demanda de que diversos aspectos de segurança sejam cumpridos, entre eles a sua invulnerabilidade a ataques de temporização. Para tal, é necessário que a implementação dos algoritmos tenha tempo de execução constante, o que motivou a adaptação do algoritmo Trace de Berlekamp e pequenas modificações no algoritmo de Fedorenko.

O principal resultado consiste na avaliação da implementação de cada algoritmo de determinação de raízes de polinômios em relação à sua variabilidade de tempo de execução e na comparação de quais poderiam ser as melhores alternativas para serem utilizadas na decodificação de códigos Goppa provendo um tempo de execução constante.

6.1 TRABALHOS FUTUROS

A modificação do algoritmo Trace de Berlekamp para utilizar o algoritmo de Zinoviev citada na Seção 5.2 poderia ser implementada e comparada com a implementação dos outros algoritmos vistos neste trabalho. As implementações podem ser refeitas em outra linguagem que permita operações sobre corpos finitos mais eficientes e que permita medições mais precisas, como em ciclos, para que uma comparação mais precisa entre os algoritmos seja feita.

Outros algoritmos de fatoração de polinômios podem ser utilizados para encontrar as raízes de um polinômio, como os algoritmos propostos por Rabin (RABIN, 1980), Ben-Or (BEN-OR, 1981) e van Oorschot and Vanstone (OORSCHOT; VANSTONE, 1989). A implementação destes algoritmos poderia ser feita e seu tempo de execução comparado aos obtidos pelos outros métodos apresentados neste trabalho.

O algoritmo de Patterson, fazendo uso de uma boa estratégia para determinação de raízes de polinômios, pode ser comparado a outros algoritmos para decodificação de códigos de Goppa, como o algoritmo de Berlekamp-Massey (MASSEY, 1969), sob o ponto de vista da sua vulnerabilidade em relação a ataques de temporização e outros ataques *side-channel*.

REFERÊNCIAS

AU, S.; EUBANKS-TURNER, C.; EVERSON, J. *The McEliece Cryptosystem.* 2003. Disponível em: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.452.3025&rep=rep1&type=pdf>. Acesso em: 15 de outubro de 2018.

BALDI, M.; CHIARALUCE, F. Cryptanalysis of a new instance of mceliece cryptosystem based on qc-ldpc codes. In: IEEE. *Information Theory, 2007. ISIT 2007. IEEE International Symposium.* França, 2007. p. 2591–2595.

BEN-OR, M. Probabilistic algorithms in finite fields. In: IEEE. *22nd Annual Symposium on Foundations of Computer Science.* Estados Unidos, 1981. p. 394–398.

BERLEKAMP, E. R. Factoring polynomials over large finite fields. *Mathematics of computation*, v. 24, n. 111, p. 713–735, 1970.

BERNSTEIN, D. J.; BUCHMANN, J.; DAHMEN, E. *Post-Quantum Cryptography.* 1. ed. Alemanha: [s.n.], 2009.

BERNSTEIN, D. J.; LANGE, T.; PETERS, C. Attacking and defending the mceliece cryptosystem. In: SPRINGER. *International Workshop on Post-Quantum Cryptography.* [S.l.], 2008. p. 31–46.

BOLLAUF, M. F. Códigos, reticulados e aplicações em criptografia. 2015.

BUCERZAN, D. et al. Improved timing attacks against the secret permutation in the mceliece pkc. *International Journal of Computers Communications & Control*, v. 12, n. 1, p. 7–25, 2017.

CHIEN, R. Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes. *IEEE Transactions on information theory*, IEEE, v. 10, n. 4, p. 357–363, 1964.

CODE Based Cryptography Python. 2014. Disponível em: <https://github.com/davidhoo1988/Code_Based_Cryptography_Python>. Acesso em: 03 de março de 2019.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. *IEEE transactions on Information Theory*, IEEE, v. 22, n. 6, p. 644–654, 1976.

FEDORENKO, S. V.; TRIFONOV, P. V. Finding roots of polynomials over finite fields. *IEEE Transactions on communications*, IEEE, v. 50, n. 11, p. 1709–1711, 2002.

FRANK, G. *Pulse code communication.* 1947. Disponível em: <https://patents.google.com/patent/US2632058>.

GABORIT, P. Shorter keys for code based cryptography. *Workshop Coding and Cryptography (WCC 05)*, p. 81–90, 2005.

GOPPA, V. D. A new class of linear correcting codes. *Problemy Peredachi Informatsii*, Russian Academy of Sciences, Branch of Informatics, Computer Equipment and Automatization, v. 6, n. 3, p. 24–30, 1970.

HAMMING, R. W. Error detecting and error correcting codes. *Bell System technical journal*, Wiley Online Library, v. 29, n. 2, p. 147–160, 1950.

HUFFMAN, W. C.; PLESS, V. *Fundamentals of Error-Correcting Codes*. EUA: Cambridge University Press, 2010.

JOCHEMSZ, E. Goppa codes & the mceliece cryptosystem. *Doktorarbeit, Universiteit van Amsterdam*, 2002.

LOIDREAU, P. Strengthening mceliece cryptosystem. In: SPRINGER. *International Conference on the Theory and Application of Cryptology and Information Security.* Berlim, 2000. p. 585–598.

MACWILLIAMS, F.; SLOANE, N. *The Theory of Error-Correcting Codes*. 3. ed. Países Baixos: North-Holland, 1977.

MASSEY, J. Shift-register synthesis and bch decoding. *IEEE transactions on Information Theory*, IEEE, v. 15, n. 1, p. 122–127, 1969.

MCELIECE, R. J. A public-key cryptosystem based on algebraic. *Coding Thv*, v. 4244, p. 114–116, 1978.

MISOCZKI, R.; BARRETO, P. S. Compact mceliece keys from goppa codes. In: SPRINGER. *International Workshop on Selected Areas in Cryptography.* Berlim, 2009. p. 376–392.

MISOCZKI, R. et al. Mdpc-mceliece: New mceliece variants from moderate density parity-check codes. In: IEEE. *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium.* Turquia, 2013. p. 2069–2073.

NIEDERREITER, H.; XING, C. *Algebraic geometry in coding theory and cryptography*. 1. ed. EUA: Princeton University Press, 2009.

OORSCHOT, P. C. V.; VANSTONE, S. A. A geometric approach to root finding in gf(q^m). *IEEE Transactions on Information Theory*, IEEE, v. 35, n. 2, p. 444–453, 1989.

PATTERSON, N. The algebraic decoding of goppa codes. *IEEE Transactions on Information Theory*, IEEE, v. 21, n. 2, p. 203–207, 1975.

RABIN, M. O. Probabilistic algorithms in finite fields. *SIAM Journal on computing*, SIAM, v. 9, n. 2, p. 273–280, 1980.

RISSE, T. *How SAGE helps to implement Goppa codes and McEliece PKCSs*. 2011. Disponível em: <http://www.weblearn.hs-bremen.de/risse/papers/ICIT11/526_ICIT11_Risse.pdf>. Acessado em 24 de março de 2019.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, ACM, v. 21, n. 2, p. 120–126, 1978.

ROERING, C. Coding theory-based cryptopraphy: Mceliece cryptosystems in sage. 2013.

SAFIEDDINE, R.; DESMARAIS, A. Comparison of different decoding algorithms for binary goppa codes. 2014.

SENDRIER, N. Code-based cryptography: State of the art and perspectives. *IEEE Security & Privacy*, IEEE, v. 15, n. 4, p. 44–50, 2017.

SENDRIER, N. et al. On the use of structured codes in code based cryptography. *Coding Theory and Cryptography III*, p. 59–68, 2009.

SHANNON, C. E. A mathematical theory of communication. *Bell system technical journal*, Wiley Online Library, v. 27, n. 3, p. 379–423, 1948.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, SIAM, v. 41, n. 2, p. 303–332, 1999.

SHOUFAN, A. et al. A timing attack against patterson algorithm in the mceliece pkc. p. 161–175, 2010.

STALLINGS, W. *Cryptography and network security*. 6. ed. Estados Unidos: Pearson Education, 2014.

STRENZKE, F. Timing attacks against the syndrome inversion in code-based cryptosystems. p. 95–107, 2011.

STRENZKE, F. Fast and secure root finding for code-based cryptosystems. In: SPRINGER. *International Conference on Cryptology and Network Security*. [S.l.], 2012. p. 232–246.

STRENZKE, F. et al. Side channels in the mceliece pkc. p. 216–229, 2008.

ZINOVIEV, V. *On the Solution of Equations of Degree*. Tese (Doutorado) — INRIA, 1996.

ANEXO A – Implementação dos códigos de Goppa

```
#!/usr/bin/env sage

from sage.misc.sage_timeit import sage_timeit
'''
  Based in https://digitalcommons.csbsju.edu/cgi/
      viewcontent.cgi?referer=https://www.google.com/
      &httpsredir=1&article=1019&context=honors_theses
  http://www.weblearn.hs-bremen.de/risse/papers/
      ICIT11/526_ICIT11_Risse.pdf
  https://github.com/davidhoo1988/
      Code_Based_Cryptography_Python
'''

def GetRandomMessage(message_length):
  message = matrix(GF(2), 1, message_length);
  for i in range(message_length):
    message[0,i] = randint(0,1);
  return message;


def GetRandomMessageWithWeight(message_length,
    message_weight):
  message = matrix(GF(2), 1, message_length);
  rng = range(message_length);
  for i in range(message_weight):
    p = floor(len(rng)*random());
    message[0,rng[p]] = 1; rng=rng[:p]+rng[p+1:];
  return message;


def split(p):
  # split polynomial p over F into even part po
  # and odd part p1 such that
  # p(z) = p^{2}_0(z) + z p{^2}_1(z)
  Phi = p.parent()
  p0 = Phi([sqrt(c) for c in p.list()[0::2]])
  p1 = Phi([sqrt(c) for c in p.list()[1::2]])
  return (p0,p1)


class goppa_code:
```

```
def __init__(self, m, t, debug = True):

    # Initialization and creation of fields
    n = pow(2, m)
    F2 = GF(2)
    F_2m = GF(n, 'Z')
    PR_F_2m = PolynomialRing(F_2m, 'X')
    Z = F_2m.gen()
    X = PR_F_2m.gen()
    k = n - m*t

    print('m = {}, n = {}, t = {} and k =
        {}\n'.format(m, n, t, k))

    if True:
        print('F_2m = {}'.format(F_2m))
        print('PR_F_2m = {}\n'.format(PR_F_2m))


    # find goppa polynomial
    while 1:
        irr_poly = PR_F_2m.random_element(t)
        if irr_poly.is_irreducible():
            break
    g = irr_poly


    print('Goppa Polynomial = {}\n'.format(g))


    factor_list = list(factor(2^m-1))
    final_factor_list = []
    for i in range(len(factor_list)):
        for j in range(factor_list[i][1]):
            final_factor_list.append(factor_list[i][0])

    if debug:
        print('factor_list = {}'.format(factor_list))
        print('final_factor_list =
            {}\n'.format(final_factor_list))
```

```
# find the code locators
codelocators = []
for i in range(n-1):
  codelocators.append(Z^(i))
codelocators.append(F_2m(0))


if debug and n < 100:
  print('codelocators_size =
     {}'.format(len(codelocators)))
  for cl in codelocators:
    print('codelocator {} =
       {}'.format(codelocators.index(cl), cl))
  print('')

h = PR_F_2m(1)

# GAMMA is a list determine membership in the
   code
gamma = []
for a_i in codelocators:
  gamma.append((h*((X-a_i).inverse_mod(g))).mod(g))

# compute the canonical parity matrix of the
   generalized Reed-Solomon
H_gRS = matrix(F_2m, t, n)
for i in range(n):
  coeffs = list(gamma[i])
  for j in range(t):
  # check to make sure the coefficient exists
    H_gRS[j,i] = coeffs[j] if j < len(coeffs)
       else F_2m(0)

# construct the binary representation of H
H = matrix(F2,m*H_gRS.nrows(),H_gRS.ncols())
for i in range(H_gRS.nrows()):
  for j in range(H_gRS.ncols()):
    be =
       bin(H_gRS[i,j].integer_representation())[2:]
    be = be[::-1]
```

```
      be = be+'0'*(m-len(be))
      be = list(be)
      H[m*i:m*(i+1),j] = vector(map(int,be))

  # let G by basis for the null-space of H.
  G = H.transpose().kernel().basis_matrix()
  if debug and n < 100:
    print('H = \n{}\n'.format(H))
    print('G = \n{}\n'.format(G))
    print('H*G = \n{}\n'.format(H*G.transpose()))


  if debug:
    print('H.nrows = {}'.format(H.nrows()))
    print('H.ncols = {}'.format(H.ncols()))
    print('G.nrows = {}'.format(G.nrows()))
    print('G.ncols = {}\n'.format(G.ncols()))



  #Construct the syndrome calculator. This will
      be used
  #to simplify the calculation of syndromes for
      decoding.
  SyndromeCalculator = matrix(PR_F_2m, 1,
      len(codelocators))
  for i in range(len(codelocators)):
    SyndromeCalculator[0,i] = (X -
        codelocators[i]).inverse_mod(g)

  if debug and n < 100:
    for i in SyndromeCalculator:
      print('SyndromeCalculator = {}'.format(i))

  self._n = n
  self._m = m
  self._g = g
  self._t = t
  self._k = k
  self._codelocators = codelocators
  self._H = H
```

```
        self._H_gRS = H_gRS
        self._G = G
        self._SyndromeCalculator = SyndromeCalculator
        self._PR_F_2m = PR_F_2m


        # patterson algorithm variables
        (g0, g1) = split(self._g)
        (d,u,v) = xgcd(g1, self._g);
        g_inv = u.mod(self._g)
        self.w = g0 * g_inv
        self.i_subtracting = []

        print('Goppa Code generated, starting
            tests...\n')


    def statistics(self, name, time_list):
        print('{} time average = {}'.format(name,
            numpy.average(time_list)))
        print('standard deviation =
            {}'.format(numpy.std(time_list)))
        print('max = {}'.format(max(time_list)))
        print('min = {}'.format(min(time_list)))
        print('dif = {}\n'.format(max(time_list) -
            min(time_list)))


    def G(self):
        return self._G

    def H(self):
        return self._H

    def goppa_polynomial(self):
        return self._g

    def n(self):
        return self._n

    def t(self):
```

```
    return self._t

  def k(self):
    return self._k

  def encode(self, message):
    return message * self._G

  def syndrome(self, message):
    return self._SyndromeCalculator *
        message.transpose()

  def syndrome_H_gRS(self, message):
    return
        self._SyndromeCalculator*message.transpose()

  def decode(self, message):
    '''
    Decoding method
    '''
    # Syndrome computation
    X = self.goppa_polynomial().parent().gen()
    syndrome =
        self._SyndromeCalculator*message.transpose()

    syndrome_poly = 0;
    for i in range (syndrome.nrows()):
      syndrome_poly += syndrome[i,0]*X^i

    error = matrix(GF(2), 1, self._n)

    # Patterson algorithm initialization
    T = syndrome_poly.inverse_mod(self._g)
    (T0, T1) = split(T + X)
    R = T0 + self.w * T1


    #################### lattice basis reduction
        ####################
    a = []
    a.append(0)
```

```
b = []
b.append(0)
(q,r) = self._g.quo_rem(R)
(a[0],b[0]) = simplify((self._g - q*R, 0 - q))


#If the norm is already small enough, we
#are done. Otherwise, intialize the base
#case of the recursive process.
lbr_done = True

condition_if = ((a[0]^2+X*b[0]^2).degree()) >
    self._t
if condition_if:
  a.append(0)
  b.append(0)
  (q,r) = R.quo_rem(a[0])
  (a[1],b[1]) = (r, 1 - q*b[0])

  if a[1] == 0:
    lbr_done = False
    (alpha, beta) = (R,1)

else:
  lbr_done = False
  (alpha, beta) = (a[0], b[0])


#Continue subtracting integer multiples of
#the shorter vector from the longer until
#the produced vector has a small enough norm.

# in the average i is t / 2 and few cases with
    t / 2-1
i = 1;
while lbr_done and
    ((a[i]^2+X*b[i]^2).degree()) > self._t:
  a.append(0)
  b.append(0)
  (q,r) = a[i-1].quo_rem(a[i])
  (a[i+1],b[i+1]) = (r, b[i-1] - q*b[i])
```

```
            i+=1

        self.i_subtracting.append(i)

        if lbr_done:
            (alpha, beta) = (a[i],b[i])

        # end lattice basis reduction
        #################### lattice basis reduction
            ####################
        #error-locator polynomial.
        elp = (alpha*alpha) + (beta*beta)*X
        #Pre-test elp
        if (X^(2^self._m)).mod(elp) != X:
            return codeword; # return without erros

        error_all = self.elp_exaust_evaluate(elp)
        codeword = message + error_all
        return codeword


    def elp_exaust_evaluate(self, elp):
        for i in range(len(self._codelocators)):
            if elp(self._codelocators[i]) == 0:
                error[0,i] = 1;

        return error



###################### Tests
    #########################

gp = goppa_code(12, 32, False)

for ind in range(20):

    message = GetRandomMessage(gp.k())
    encoded_message = gp.encode(message)
    error = GetRandomMessageWithWeight(gp.n(),
        gp.t())
```

```
transmited = encoded_message + error
decoded_message = gp.decode(transmited)
assert encoded_message == decoded_message,
    'Message decoding fail'
print('message = {}'.format(message))
print('encoded_message =
    {}'.format(encoded_message))
print('error = {}\n\n'.format(error))
```

ANEXO B – Implementação do Berlekamp Trace Algorithm

```
def bta_rec(self, f, i):
  if f.degree() == 0: return []
  if f.degree() == 1: return [-f[0]]
  Z = f.base_ring().gen()
  X = f.parent().gen()
  tr = lambda x: sum(x**(2**j) for j in
      range(self._m))
  beta = [Z**j for j in range(self._m)]
  t = tr(beta[i%self._m]*X)
  g = gcd(f, t)
  aux = gcd(f, 1+t)
  return self.bta_rec(g, i+1) + self.bta_rec(aux,
      i+1)

def bta_iterav(self, f):
  X = f.parent().gen()
  Z = f.base_ring().gen()
  beta = [Z**i for i in range(self._m)]
  tr = lambda x: sum(x**(2**i) for i in
      range(self._m))
  result = []
  dummy_list = []
  next_fs = [f]

  quantity = 2**(ceil(log(self._t,2))+1)-1
  for index in range(quantity):
    leng_next = len(next_fs)
    half_len = floor(leng_next/2)
    p = next_fs[0]
    if index < leng_next:
      p = next_fs[index]
    else:
      p = next_fs[0]

    if p.degree() == 1:
      result.append(-p[0])
    else:
      dummy_list.append(-p[0])

    half = floor(p.degree()/2)
    ct = tr(beta[0]*X)
```

```
        best = gcd(p, ct)
        bt = ct
        for i in range(1, self._m):
            ct = tr(beta[i]*X)
            current = gcd(p, ct)
            diff_best = abs(half - best.degree())
            diff_current = abs(half - current.degree())
            if diff_current < diff_best:
                best = current
                bt = ct
            else:
                dummy = current
                dt = ct

        if p.degree() > 1:
            next_fs.append(best)
            next_fs.append(gcd(p, bt+1))

        else:
            dummy_list.append(best)
            dummy_list.append(gcd(p, bt+1))

    return result
```

ANEXO C – Implementação do algoritmo de Fedorenko

```
def gray(self, i):
  gray_n = i^^(i>>1)
  return "{0:0{1}b}".format(gray_n, self._m)


def fedorenko(self, f):
  t = f.degree()
  max_i = ceil((t-4)/5)
  max_i_include = max_i + 1
  # step 1
  L = matrix([[self.l(self._codelocators[k], i, f)
      for k in range(self._m)] for i in
      range(max_i_include)])

  # step 2
  A =  matrix(self._PR_F_2m, max_i_include,
      self._n)
  for i in range(max_i_include):
    A[i,0] = f[5*i]

  # step 3
  for i in range(max_i_include):
    for j in range(1, self._n):
      pos = self.sigma(j)
      if pos != -1:
        A[i,j] = A[i,(j-1)] + L[i,pos]
      else:
        print 'error sigma'

  # step 4
  roots = []
  dummy_list = []
  Z = f.base_ring().gen()
  for j in range(1, self._n):
    next_gray = self.gray(j)
    xj = self.get_element(next_gray, Z)

    if (xj == 0):
      result = f[0]
      #dummy
      for i in range(max_i_include):
```

```
                dummy += (xj**(5*i)) * A[i,j]
        else:
            result = f[3] * (xj**3)
            for i in range(max_i_include):
                result += (xj**(5*i)) * A[i,j]

        if (result == 0):
            roots.append(xj)
        else:
            dummy_list.append(xj)

    return roots


def l(self, x, i, f):
    result = 0
    for j in range(4):
        j2 = 2**j
        result += f[(5*i)+j2] * (x**j2)
    return result


'''
    position in which gray_vector[i]
    differs from gray_vector[i-1]
'''
def sigma(self, i):
    pos = -1
    a = self.gray(i)
    b = self.gray(i-1)
    a_length = len(a)
    for j in range(a_length):
        if a[j] != b[j]:
            if pos == -1:
                pos = j
        else: #dummy
            if pos == -1:
                dummy = j
    return (a_length - 1) - pos
```

```
def get_element(self, gray, p):
  gray_len = len(gray)
  element = 0
  dummy = 0
  for index in range(gray_len):
    if gray[index] == '1':
      element += p^((gray_len - 1) - index)
    else:
      dummy += p^((gray_len - 1) - index)
  return element
```

ANEXO D – Artigo

Determinação em tempo constante de raízes de polinômios sobre corpos finitos para códigos de correção de erros

Dúnia Marchiori[1]

[1] Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis, SC – Brazil

dunia.marchiori@grad.ufsc.br

Resumo. *Os sistemas de criptografia mais populares atualmente são Rivest-Shamir-Adleman (RSA) e criptografia de curvas elípticas, os quais se baseiam em problemas que algoritmos quânticos conseguem resolver em tempo polinomial. Por este motivo diferentes propostas de criptossistemas vêm sendo estudadas na área criptografia pós-quântica, para que as informações continuem seguras mesmo contra um adversário com um computador quântico.*
O esquema de McEliece é um esquema de criptografia de chave pública proposto em 1978. A sua segurança se baseia no fato de que a chave pública é indistinguível de uma matriz aleatória e no problema de decodificação geral, o que leva o esquema a resistir a ataques de computadores quânticos e que o categoriza como um criptossistema pós-quântico. O criptossistema de McEliece faz uso de código de Goppa em sua proposta original. Atualmente, diferentes estudos demonstraram a vulnerabilidade da decodificação dos códigos de Goppa em relação a ataques de temporização, onde a variabilidade do tempo de execução traz insegurança para uma implementação.
Este trabalho tem como objetivo apresentar os códigos de Goppa, seu uso no esquema de McEliece e analisar diferentes implementações para a determinação de raízes de polinômios, um passo crucial na decodificação dos códigos. Para isso, o esquema original de McEliece será apresentado juntamente com os códigos de Goppa. O principal algoritmo para decodificação dos códigos, o algoritmo de Patterson, será descrito e implementado fazendo o uso de diferentes métodos para a determinação de raízes de polinômios. Por fim, estas implementações serão comparadas em relação a sua vulnerabilidade a ataques de temporização.

1. Introdução

Um dos principais objetivos da segurança em computação é prover confidencialidade dos dados, que garante que dados sigilosos e privados não fiquem disponíveis para acesso de pessoas não autorizadas. Criptossistemas são um conjunto de algoritmos criptográficos que prezam pela confidencialidade, assegurando que um adversário que intercepte alguma mensagem cifrada não seja capaz de recuperar a mensagem original caso não tenha posse da chave necessária para a decifragem.

Criptossistemas podem ser classificados em dois tipos: simétricos e assimétricos. Em criptossistemas simétricos, ambas as partes da comunicação têm posse da mesma chave para a cifragem e decifragem das mensagens. A chave então é um segredo compartilhado apenas entre as partes que compõem a comunicação. Uma desvantagem desses

sistemas é que, para a realização do acordo do segredo, que permitiria uma comunicação segura em um canal inseguro, há a necessidade de um canal seguro para o compartilhamento do mesmo.

Criptografia assimétrica, ou criptografia de chave pública, se caracteriza pela presença de um par de chaves, uma chave pública e outra privada, que são utilizadas para realizar operações complementares, como cifragem e decifragem ou assinar arquivos digitalmente e verificar estas assinaturas. A chave privada deve ser conhecida apenas pela entidade que a gerou mas a chave pública pode ser distribuída livremente, o que remove a necessidade de um canal seguro para o acordo de chaves.

Exemplos de algoritmos assimétricos comumente usados são o RSA, criado por Ron Rivest, Adi Shamir e Len Adleman [Rivest et al. 1978], e ECDSA ECDSA Elliptic Curve Digital Signature Algorithm. Estes e demais algoritmos baseiam sua segurança na dificuldade da fatoração de inteiros ou logaritmos discretos, ou seja, o fato destes problemas serem intratáveis garante a segurança para os esquemas. Em 1999, Peter Shor publicou um algoritmo capaz de calcular estes dois problemas em tempo polinomial em um computador quântico [Shor 1999]. Isso significa que dados cifrados hoje não necessariamente estarão seguros no futuro.

Como forma de combater esta situação, na área da criptografia pós-quântica são estudados esquemas criptográficos baseados em problemas que não foram afetados pelo algoritmo de Shor para que a segurança das informações cifradas não seja ameaçada por um adversário com um computador quântico. Entre as propostas de criptossistemas pós-quânticos estão os sistemas baseados em funções de hash, em reticulados e em códigos de correção de erros.

Durante a transmissão de dados podem ocorrer erros que transformam a mensagem recebida numa mensagem diferente da que foi enviada. Códigos de correção de erros são capazes de detectar estes erros em uma mensagem e corrigi-los. Para isso fazem uso de redundância, adicionando informações à mensagem que podem auxiliar na sua correção após a transmissão por um canal com ruído. Em sistemas baseados em códigos deste tipo, os erros são adicionados propositalmente à mensagem, como forma de dificultar a leitura dos dados por partes indesejadas.

O esquema de McEliece [Mceliece 1978] foi o primeiro esquema de criptografia de chave pública baseado em códigos de correção de erros proposto e resistiu a diversos ataques realizados até os dias atuais, sendo assim, um ótimo candidato como criptossistema pós-quântico. O código corretor de erros em que McEliece se baseia originalmente são códigos de Goppa [Goppa 1970], que é utilizado para a geração das chaves pública e privada.

Nos últimos anos, diferentes estudos [Bucerzan et al. 2017] [Shoufan et al. 2010] [Strenzke et al. 2008] indicaram vulnerabilidades no algoritmo mais utilizado na etapa de decodificação dos códigos Goppa, o algoritmo de Patterson, em relação a ataques de temporização. Diferentes implementações do algoritmo de Patterson, fazendo uso de diferentes formas de determinação de raízes de polinômios, por exemplo, demonstraram variabilidade em seu tempo de execução quando a execução da mesma tarefa diversas vezes foi analisada. Isso cria brechas para ataques e compromete a segurança do esquema.

Este trabalho visa realizar um estudo sobre o esquema de McEliece original e com-

parar implementações do mesmo fazendo uso de diferentes métodos para a determinação de raízes de polinômios, analisando a variabilidade nos tempos de execução de cada método e, consequentemente, a vulnerabilidade a ataques de temporização.

2. Códigos lineares

O alfabeto dos códigos lineares que serão descritos a seguir é um corpo finito denotado como $\mathbb{F}_q$ ou $GF(q)$, que é uma estrutura formada por um conjunto de q elementos e duas operações, soma e multiplicação, que possuem as propriedades de associatividade, fechamento, existência de um elemento neutro e um elemento inverso, comutatividade e distributividade da multiplicação [Stallings 2014].

A quantidade de elementos de um corpo finito deve ser sempre uma potência de um número primo p^n, sendo n um número inteiro positivo, como provado em [Niederreiter and Xing 2009, pp. 1-2]. O número primo p é chamado de característica do corpo. Corpos binários, ou seja, corpos com característica 2, são especiais para a computação pois seus elementos, por usarem apenas 0 e 1, podem ser facilmente relacionados a bits em um computador.

Seja $\mathbb{F}_q^n$ um espaço vetorial de todas as n-tuplas sobre $\mathbb{F}_q$. Um código linear C de dimensão k é um subespaço vetorial de dimensão k de $\mathbb{F}_q^n$, que pode ser denominado também como um código [n,k] sobre $\mathbb{F}_q$. Dessa forma, o código C possui q^k elementos, que são chamados de *codewords*, ou palavras.

Segundo [Huffman and Pless 2010], podemos representar códigos lineares através de uma matriz geradora ou uma matriz de paridade.

- Uma matriz geradora G de um código [n,k] C é uma matriz de dimensão $k \times n$ em que as linhas são vetores que formam uma base de C. Portanto, para cada $c \in C$ há um vetor $x \in \mathbb{F}_q^k$ tal que $c = xG$.
- Uma matriz de teste de paridade H de um código C é a matriz geradora do código *dual* (ou ortogonal) $C^\perp$. $C^\perp$ é um conjunto formado por vetores v_i cujo produto interno com vetores $u_i \in C$ resulta em 0, ou seja, $C^\perp = \{v \in \mathbb{F}_q^n \mid \langle v, u \rangle = 0, \forall u \in C\}$. Isto implica que uma palavra c pertence ao código C se, e somente se, $Hc^T = \overrightarrow{0}$.

Sendo H uma matriz de paridade de um código linear [n,k] C, o vetor resultante $S(y) = Hy^T$ chama-se *síndrome* de y.

Duas métricas usadas na teoria de códigos são a distância e o peso de Hamming. Seja $x, y \in \mathbb{F}_q^n$. A distância de Hamming $d(x, y)$ entre dois vetores x e y é o número de coordenadas que diferem entre x e y. Já o peso $w(x)$ do vetor w é o número de coordenadas diferentes de 0, ou seja, $w(x) = d(x, \overrightarrow{0})$.

Uma característica importante de um código C sobre $\mathbb{F}_q^k$ é a sua distância mínima, que indica o valor da menor medida de distância entre todas as palavras distintas de C. O código C tendo uma distância mínima d pode corrigir $t = \left\lfloor \frac{(d-1)}{2} \right\rfloor$ erros, como provado em [MacWilliams and Sloane 1977, pp. 10].

Um código que é capaz de detectar erros mas que não é capaz de corrigi-los gera a necessidade de descartar os dados recebidos e transmiti-los novamente. Para evitar essas

situações de retransmissão, pode-se fazer uso de códigos corretores de erros. Os códigos de Goppa (Seção 3) pertencem a essa categoria de códigos.

3. Códigos de Goppa

Os códigos de Goppa formam uma classe de códigos corretores de erros e foram introduzidos por V. D. Goppa em 1970 [Goppa 1970]. Um polinômio de Goppa $g(z)$ é um polinômio de grau t cujos coeficientes estão em $GF(q^m)$, sendo q um número primo e m um inteiro, isto é,

$$g(z) = g_0 + g_1 z + \cdots + g_t z^t = \sum_{i=0}^{t} g_i z^i. \tag{1}$$

Seja L um subconjunto $L = \{\alpha_1, \ldots, \alpha_n\} \in GF(q^m)$ tais que $g(\alpha_i) \neq 0$ para todo $\alpha_i \in L$. Um código de Goppa $\Gamma(L, g(z))$ consiste de todos vetores c_i sobre $GF(q)$ que satisfaçam a condição

$$R_c(z) = \sum_{i=1}^{n} \frac{c_i}{z - \alpha_i} \equiv 0 \ (mod\ g(z)) \tag{2}$$

onde $\frac{1}{z-\alpha_i}$ é o único polinômio que satisfaz $(z - \alpha_i) \cdot \frac{1}{z-\alpha_i} \equiv 1(mod\ g(z))$.

Pelo fato do código de Goppa ser um código linear, seus parâmetros consistem no comprimento n, dimensão k e distância mínima d. O comprimento n é o tamanho das palavras do código, o que nos códigos de Goppa é fixado por L. A dimensão k satisfaz $k \geq n - m \cdot t$ e a distância mínima d satisfaz $d \geq t + 1$ [Huffman and Pless 2010, pp. 523].

3.1. Matriz de paridade de um código Goppa

Para a decodificação, é necessária uma matriz de paridade H de um código de Goppa $\Gamma(L, g(x))$ tal que $c = c_0 c_1 \ldots c_{n-1} \in \Gamma(L, g(x))$ se, e somente se, $Hc^T = \overrightarrow{0}$. A matriz de paridade H de um código de Goppa $\Gamma(L, g(x))$ é definida como

$$H = \begin{pmatrix} h_1 & h_2 & \ldots & h_n \\ h_1 \cdot \alpha_1 & h_2 \cdot \alpha_2 & \ldots & h_n \cdot \alpha_n \\ \vdots & \vdots & \vdots & \vdots \\ h_1 \cdot \alpha_1^{t-1} & h_2 \cdot \alpha_2^{t-1} & \ldots & h_n \cdot \alpha_n^{t-1} \end{pmatrix}. \tag{3}$$

3.2. Matriz geradora de um código Goppa

A matriz geradora G é utilizada para a etapa de codificação de uma mensagem. Uma palavra c é obtida através da mensagem m e da matriz geradora G como $c = mG$. Temos que para toda palavra $c \in \Gamma(L, g(z))$, a igualdade $Hc^T = 0$ é verdadeira. Por consequência, pode-se obter a matriz G a partir da matriz de paridade H através da equação $GH^T = 0$, de forma que o espaço nulo da matriz H forme o espaço linha de G.

3.3. Codificação de um código Goppa

Para codificar uma mensagem utilizando um código de Goppa $\Gamma(L, g(z))$ cuja matriz geradora é G, basta dividi-la em blocos de k símbolos e multiplicar cada bloco por G.

$$(m_0, \ldots, m_k) \cdot G = (c_1, \ldots, c_n). \tag{4}$$

O vetor c resultante é uma palavra do código de Goppa $\Gamma(L, g(z))$. Como os código de Goppa são códigos corretores de erros, mesmo que alguns erros sejam adicionados à palavra c, a mensagem original m ainda pode ser recuperada na etapa de decodificação.

3.4. Decodificação de um código Goppa

Seja y uma mensagem recebida que contém r erros, sendo $r \leq \left\lfloor \frac{(d-1)}{2} \right\rfloor$. Então y é a mensagem original com a adição de alguns erros, ou seja,

$$(y_1, \ldots, y_n) = (m_1, \ldots, m_n) + (e_1, \ldots, e_n) \tag{5}$$

com $e_i \neq 0$ em r posições, ou, o peso w do vetor de erros é $w((e_1, \ldots, e_n)) = r$. Para obter a mensagem original m a partir de y é preciso encontrar o vetor de erros e. Para obter este vetor em códigos de Goppa binários, o algoritmo mais utilizado é o algoritmo de Patterson.

3.4.1. Algoritmo de Patterson

O algoritmo de Patterson [Patterson 1975] foi proposto por Nicholas Patterson em 1975 e é usado na etapa de decodificação de códigos Goppa binários, pois é capaz de corrigir t erros em uma dada palavra y.

Sendo $y = c + e$, para determinar o vetor de erros e adicionado a fim de corrigir a palavra e encontrar a mensagem original c, o algoritmo faz uso do cálculo de síndrome e do polinômio localizador de erros (ou *error locator polynomial* (ELPELP Error Locator Polynomial)). O Algoritmo 1 descreve os passos que compõem o algoritmo de Patterson.

Para o passo 6, pode-se fazer uso de diferentes métodos para a determinação de raízes em polinômios. Três métodos diferentes são o foco deste trabalho e estão descritos em mais detalhe no Capítulo 5. São eles: análise exaustiva, algoritmo Trace de Berlekamp e o algoritmo de Fedorenko.

Estudos como [Bucerzan et al. 2017] e [Shoufan et al. 2010] mostram como a etapa de determinação de raízes de polinômios pode ser vulnerável a ataques de temporização (Seção 4.1.4). Portanto, a análise dos três algoritmos supracitados será feita tendo como objetivo obter um método que possua um tempo de execução o mais constante possível, para que a segurança do criptossistema não seja prejudicada na implementação.

4. Criptografia assimétrica

Criptografia assimétrica, ou criptografia de chave pública, se caracteriza pela presença de um par de chaves, uma chave pública e outra privada, que são utilizadas para realizar operações complementares, como cifragem e decifragem ou assinar arquivos digitalmente

Algoritmo 1 Algoritmo de Patterson

Seja $y = (y_1, \ldots, y_n)$ uma *codeword* recebida que contém t erros e um código de Goppa $\Gamma(L, g(x))$.

1. Calcular o polinômio da síndrome $S(x)$ dado por

$$S(x) = \sum_{i=1}^{n} \frac{y_i}{x - L_i} \mod g(x)$$

2. Calcular o inverso $S(x)^{-1}$ do polinômio da síndrome $S(x)$ módulo $g(x)$. Para isto, pode-se usar o algoritmo de Euclides estendido [Stallings 2014, pp.97].
3. Calcular a raiz quadrada do inverso do polinômio da síndrome $S(x)$ mais x no módulo $g(x)$ tal que

$$\tau(x) = \sqrt{S(x)^{-1} + x} \mod g(x)$$

O cálculo da raiz quadrada é detalhado em [Risse 2011] e em [Safieddine and Desmarais 2014].
4. Determinar as partes par e ímpar do polinômio localizador de erros tal que

$$a(x) \equiv b(x)\tau(x) \mod g(x)$$

Para isto, pode-se usar o algoritmo de Euclides estendido.
5. Construir o polinômio localizador de erros $\sigma(x)$ tal que

$$\sigma(x) = a^2(x) + xb^2(x)$$

6. Determinar as raízes do polinômio localizador de erros $\sigma(x)$ e construir o vetor e. As raízes do polinômio $\sigma(x)$ correspondem às posições do erros [Goppa 1970].

$$e = (\sigma(\alpha_1), \ldots, \sigma(\alpha_n)) \oplus (1, \cdots, 1)$$

sendo que $\alpha_i \in L$.
7. A palavra original enviada c pode então ser obtida com $c = y - e$.

e verificar estas assinaturas. A chave pública pode ser distribuída livremente mas a chave privada deve ser conhecida apenas pela entidade que a gerou. Qualquer entidade pode utilizar a chave pública de outra para cifrar uma mensagem, que apenas poderá ser decifrada com a respectiva chave privada, ou seja, apenas a entidade que tem conhecimento da chave privada par da chave pública usada poderá ler a mensagem.

Este conceito surgiu em 1976 com a publicação de Whitfield Diffie e Martin Hellman [Diffie and Hellman 1976] e era completamente diferente de tudo o que era utilizado em criptografia até o momento. Até então, fazia-se uso de apenas uma chave e os sistemas de criptografia se baseavam principalmente em substituição e permutação em contraste com os algoritmos de criptografia de chave pública, que se baseiam em funções matemáticas.

Um dos exemplos de algoritmos assimétricos mais conhecido é o RSA, criado por Ron Rivest, Adi Shamir e Len Adleman em 1978 [Rivest et al. 1978]. O algoritmo de McEliece [Mceliece 1978] é também um exemplo e surgiu na mesma época.

4.1. Criptossistema de McEliece

Robert McEliece publicou em 1978 [Mceliece 1978] um criptossistema de chave pública que gera suas chaves pública e privada a partir de um código linear de correção de erros. O código proposto originalmente foi o código de Goppa. O sistema possui cifragem e decifragem eficientes e se mantém resistente a ataques feitos nos últimos 40 anos como os estudados em [Bernstein et al. 2008], onde parâmetros que provêm segurança contra os mais diversos ataques foram propostos. Um dos principais aspectos negativos do esquema é a grande quantidade de bytes que a chave pública ocupa.

O criptossistema proposto por McEliece pode ser visto como um conjunto de três etapas necessárias para a realização de troca de mensagens: geração das chaves, cifragem e decifragem. Na etapa de geração de chaves são geradas as chaves pública e privada que depois podem ser utilizadas para a cifragem e decifragem de mensagens.

4.1.1. Geração do par de chaves

No algoritmo original é utilizado código de Goppa para construir as chaves pública e privada. Inicialmente, é selecionado um polinômio de Goppa $g(z)$ de grau t cujos coeficientes estão em $GF(2^m)$. Depois é definida a matriz [k,n] geradora G que cria o código de Goppa C caracterizado por $g(z)$ e por L. Em seguida, são selecionas uma matriz $k \times k$ inversível S e uma matriz de permutação P de dimensões $n \times n$. Uma matriz de permutação é uma matriz formada apenas por zeros e uns, obtida através da permutação das linhas de uma matriz identidade, sendo que ao fim, há apenas um elemento 1 por coluna.

Tendo gerados as matrizes G, S e P, elas são multiplicadas de modo a gerar a matriz $G' = SGP$, que fará parte da chave pública $PU = (G', t)$. A chave privada consistirá em $PR = (g(z), S, G, P)$.

4.1.2. Cifragem de uma mensagem

A cifragem de mensagens no criptossistema de McEliece faz uso da ideia de adicionar erros propositalmente às mensagens. Consideremos que Alice e Bob desejam se comunicar utilizando o criptossistema de McEliece. Após a publicação da chave pública PU_A de Alice, Bob pode enviar uma mensagem m a ela. Para isso, Bob gera um vetor binário aleatório e de tamanho n com peso $w(e) \leq t$ e calcula $c = mG' + e$, que é o texto cifrado que será enviado à Alice.

4.1.3. Decifragem de uma mensagem

Ao receber a mensagem c que foi cifrada utilizando sua chave pública, Alice pode usar sua chave privada para decifrá-la da seguinte maneira: calcula-se $c' = cP^{-1}$, sendo P^{-1} a matriz inversa da matriz de permutação P. Sendo

$$c' = cP^{-1} = mG'P^{-1} + eP^{-1} = mSGPP^{-1} + eP^{-1} = (mS)G + e', \tag{6}$$

c' representa uma *codeword* do código de Goppa escolhido anteriormente, pois é uma mensagem mS multiplicada à matriz geradora G do código e com t erros adicionados, os quais a decodificação pode eliminar (pelo fato de o código de Goppa ser um código corretor de erros).

Portanto, Alice pode utilizar o algoritmo de decodificação do código sobre c' e obter $c' = mSG$. A partir daqui é possível obter a mensagem original m multiplicando c' por $(SG)^{-1}$ ou encontrando $m' = mS$ através de redução $[G^T|(mS)^T]$, como detalhado em [Jochemsz 2002], e em seguida encontrar $m = m'S^{-1}$, já que a matriz S é inversível.

4.1.4. Ataques de temporização

Uma categoria de ataques explorados no criptossistema de McEliece são os chamados ataques de canal lateral (*side-channel attacks*), que se baseiam em informações além da mensagem original e da mensagem cifrada no processo de cifragem ou decifragem. Estas informações adicionais podem ser, por exemplo, estatísticas de consumo de energia ou informações em relação ao tempo que as operações duram.

Os ataques propostos em [Strenzke et al. 2008] e [Strenzke 2011] são ataques *side-channel* que utilizam informação de tempo, conhecidos como ataques de temporização (*timing attacks*). Eles demonstram como a variação do tempo entre diferentes execuções de uma mesma tarefa podem revelar informações importantes sobre a chave privada utilizada.

Em [Bucerzan et al. 2017] e [Shoufan et al. 2010] são descritos ataques de temporização sobre a etapa de determinação das raízes do polinômio localizador de erros na decodificação de um código Goppa (Seção 3.4.1). O propósito destes ataques é o de recuperar a mensagem original através da obtenção do polinômio localizador de erros $\sigma(x)$. Isto somente é possível se o tempo de execução da decodificação variar de acordo com a quantidade de raízes do polinômio.

Um ataque possível explora a diferença no tempo de execução da determinação de raízes entre a decodificação de uma mensagem que contém t erros e outra que contém $w > t$ erros, sendo t a quantidade de erros que o código Goppa utilizado pode corrigir. Quando uma mensagem contém uma quantidade de erros w menor ou igual a t, o polinômio $\sigma(x)$ contém w raízes, mas quando a quantidade de erros w é maior que t, $\sigma(x)$ contém apenas uma fração de w raízes [Strenzke 2012].

Com esta informação o ataque pode ser feito da seguinte maneira: tendo em mãos um texto cifrado c, o atacante pode inverter um bit de c e obter c'. Ao observar o tempo de execução de c' é possível inferir se c' possui $t-1$ ou $t+1$ erros, já que uma mensagem com $t+1$ erros possui um $\sigma(x)$ com menos raízes, o que poderia acarretar em uma execução mais rápida. Realizando isto para todos os bits de c, o atacante pode descobrir quais as posições que contém erros.

Em uma implementação que realize a avaliação do polinômio localizador de erros em tempo constante independentemente da quantidade de raízes do polinômio, os ataques de temporização supracitados não têm chance de suceder. Portanto, no Capítulo 5 serão apresentados diferentes algoritmos que podem ser utilizados na análise do polinômio localizador de erros que posteriormente serão avaliados em relação à sua variabilidade de tempo de execução, analisando se os mesmos podem trazer alguma vulnerabilidade ao criptossistema levando em consideração ataques de temporização, ou seja, analisar se a implementação do algoritmo possui tempo de execução constante para a decodificação de diferentes mensagens.

5. Algoritmos para determinação de raízes de polinômios

Três algoritmos usados para determinação de raízes de polinômios em corpos finitos foram foco deste trabalho. Estes algoritmos podem ser utilizados no sexto passo do algoritmo de Patterson, detalhado na Seção 3.4.1.

5.1. Avaliação exaustiva

A maneira mais simples para encontrar as raízes do polinômio localizador de erros $\sigma(x)$ é testando todos os elementos do código. Todos os elementos X_i (exceto zero) de um corpo finito $GF(q)$ pode ser descrito como α^i para $0 \leq i < q$, onde α é um elemento primitivo de $GF(q)$ [Huffman and Pless 2010].

Desta maneira, a busca pelas raízes de $\sigma(x)$ pode ser feita avaliando o resultado do polinômio localizador de erros para cada X_i. Todos os X_i tal que $\sigma(X_i) = 0$ são raízes do polinômio e indicam as posições dos erros da palavra sendo analisada.

5.2. Algoritmo Trace de Berlekamp

O algoritmo Trace de Berlekamp (*Berlekamp Trace Algorithm* (BTA), em inglês) é um algoritmo recursivo proposto por Elwyn Berlekamp em 1970 [Berlekamp 1970] que determina as raízes de um polinômio através da sua fatoração. Por exemplo, fatorando o polinômio $p(x) = x^2 - 4$ temos $p(x) = (x-2)(x+2)$, onde podemos ver facilmente que a raiz de $(x-2)$ é 2 e a raiz de $(x+2)$ é -2. Portanto, fatorando o polinômio $p(x)$ conseguimos encontrar suas raízes 2 e -2.

O algoritmo se baseia nas propriedades da função *trace*, definida como $Tr(x) = x^{2^0} + x^{2^1} + x^{2^2} + \cdots + x^{2^{m-1}}$. Uma propriedade importante da função *trace* é que , sendo

$B = \{\beta_1, \ldots, \beta_m\}$ uma base de $\mathbb{F}_{2^m}$, todo elemento $\alpha \in \mathbb{F}_{2^m}$ é representado unicamente pela m-tupla $(Tr(\beta_1 \cdot \alpha), \ldots, Tr(\beta_m \cdot \alpha))$. Com isso, qualquer polinômio $\sigma(z)$ sobre $\mathbb{F}_{2^m}$ que divide $z^{2^m} - z$ é fatorado em dois outros polinômios $g(z) = mdc(\sigma(z), Tr(\beta \cdot \alpha))$ e $h(z) = mdc(\sigma(z), 1 + Tr(\beta \cdot \alpha))$, onde $mdc(x, y)$ indica o máximo divisor comum entre x e y. Se β iterar por B e a fatoração for aplicada recursivamente para $g(z)$ e $h(z)$, o polinômio $\sigma(z)$ pode ser dividido em fatores lineares, o que pode ser utilizado para encontrar suas raízes.

O Algoritmo 2 descreve o algoritmo Trace de Berlekamp recursivo.

Algoritmo 2 *Berlekamp Trace Algorithm* $BTA(p, i)$

Seja $p(x)$ o polinômio que se deseja encontrar as raízes e i um inteiro.

1. Se $p(x)$ tem grau menor ou igual a 1, retorne a raiz de $p(x)$.
2. Calcule $p_0(x) = \text{mdc}(p(x), \text{Tr}(\beta_i, x))$.
3. Calcule $p_1(x) = \text{mdc}(p(x), 1 + \text{Tr}(\beta_i, x))$.
4. Retorne $BTA(p_0(x), i + 1) \cup BTA(p_1(x), i + 1)$.

5.3. Algoritmo de Fedorenko

Outra maneira de determinar as raízes de um polinômio é decompondo-o em polinômios lineares. Esta técnica é interessante pois a análise exaustiva de um polinômio linearizado (*linearized polynomial*) tem menor custo quanto comparado com a análise exaustiva citada na Seção 5.1 [Fedorenko and Trifonov 2002].

Um polinômio p na forma

$$p(y) = \sum_i p_i y^{q^i},\ p_i \in \mathbb{F}_{q^m}.$$

é chamado de polinômio linearizado (*linearised polynomial*) ou q-polinômio sobre $\mathbb{F}_{q^m}$.

Segundo as propriedades de polinômios linearizados e polinômios afim (*affine polynomial*), descritas em [Fedorenko and Trifonov 2002], um polinômio afim $A(y) = p(y) + \beta, \beta \in \mathbb{F}_{2^m}$ pode ser avaliado como $A(y_i) = A(y_{i-1}) + p(\alpha^{\delta(y_i, y^{i-1})})$ para todos os pontos $y_i \in \mathbb{F}_{2^m}$, onde $\delta(y_i, y^{i-1})$ indica a posição na qual y_i e y_{i-1} têm valores diferentes quando todos os elementos estão ordenados em código Gray e $\{\alpha^0, \alpha^1, \ldots, \alpha^{m-1}\}$ é uma uma base de $\mathbb{F}_{2^m}$. Esta propriedade torna possível que a avaliação de um polinômio afim seja feita com apenas uma adição a mais a cada $y_i \in \mathbb{F}_{2^m}$.

6. Análise dos diferentes métodos para determinação de raízes de polinômios

Todas as implementações realizadas para este trabalho foram feitas com SageMath[1], um software matemático livre e open-source que utiliza a linguagem Python e permite a programação com corpos finitos, anéis e polinômios.

Foram produzidos código-fonte para a codificação e decodificação de códigos de Goppa baseado nos trabalhos de [Risse 2011] [Roering 2013] [git 2014]. A decodificação faz uso do algoritmo de Patterson, cujo passo de determinação das raízes do polinômio localizador de erros foi implementado com três algoritmos diferentes: a avaliação exaustiva, o algoritmo Trace de Berlekamp e o algoritmo proposto por Fedorenko.

[1] http://www.sagemath.org/

6.0.1. Algoritmo Trace de Berlekamp

O algoritmo Trace de Berlekamp foi feito de duas maneiras: uma implementação recursiva e uma iterativa desenvolvida durante este trabalho. A versão recursiva foi implementada de acordo com o algoritmo original descrito na Seção 5.2 e tem um tempo de execução variável pois a quantidade de chamadas recursivas depende do maior divisor comum e da quantidade de raízes que o polinômio possui.

A execução do BTA pode ser vista como uma árvore, como mostra a Figura 1. As folhas são as chamadas da função para polinômios de grau 1, cujas raízes são trivialmente calculadas e retornadas para a chamada um nível acima do seu. Isto ocorre até que todas as raízes sejam retornadas para a raiz da árvore, ou seja, a chamada inicial do algoritmo. Na decodificação de um código Goppa, a altura da árvore que representa sua execução está ligada ao maior divisor comum entre o polinômio p (inicialmente o polinômio localizador de erros) e a função Trace.

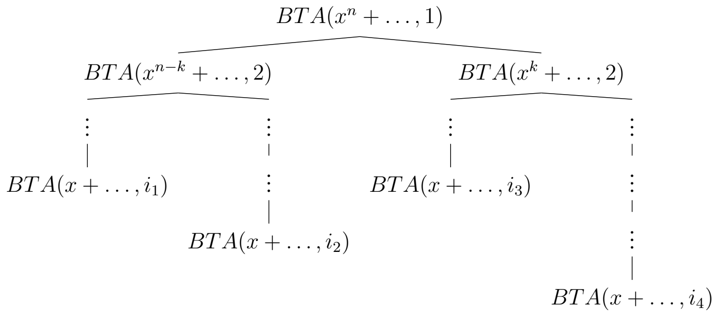

Figura 1. Ilustração da execução do algoritmo Trace de Berlekamp recursivo

É importante notar na execução ilustrada na Figura 1 que cada ramificação da árvore possui uma quantidade de chamadas recursivas diferente. Isto se deve ao fato de que o maior divisor comum é feito entre o polinômio recebido por parâmetro e o polinômio resultante da função Trace aplicada em um elemento β_i da base de $\mathbb{F}_{2^m}$. Logo, conforme o β_i definido em uma chamada recursiva $r_0 = BTA(x^a + \dots, i)$, por exemplo, pode ocorrer de esta chamada gerar outras duas chamadas $r_1 = BTA(x^{a-1} + \dots, i + 1)$ e $r_2 = BTA(x + \dots, i + 1)$, onde o ramo gerado por r_1 será mais longo que o de r_2.

Desta forma, a altura da árvore de execução da decodificação de uma mensagem de um código Goppa que corrige t erros pode variar de $\log_2 n$ no melhor caso, caso o polinômio p seja sempre divido ao meio, até n no pior caso, caso p seja sempre diminuído em 1.

Por causa disto foi desenvolvida uma versão do algoritmo de Berlekamp modificada. Esta versão é iterativa, criada com o objetivo de remover as recursões e, juntamente com outras modificações, conseguir um tempo de execução mais constante e independente da quantidade de raízes do polinômio.

Algoritmo de Berlekamp modificado A maior fonte de variação de tempo no código iterativo é a iteração pela lista. Somente transformando a execução recursiva em iterativa mantém o mesmo problema com a árvore gerada pela execução. Para remover este problema e executar o laço uma quantidade de vezes constante, decidiu-se por tentar encontrar a melhor divisão possível para o polinômio p, ou seja, procurar dividir o polinômio sempre ao meio. Para tal, é calculado o maior divisor comum p_0 entre p e a função Trace para todos os β_i da base e a próxima iteração será feita utilizando o p_0 que tenha o grau mais próximo da metade do grau do polinômio p. Como o polinômio é dividido sempre ao meio, é possível fixar a quantidade de iterações do algoritmo para $2^{\lceil \log_2 t \rceil + 1} - 1$. O Algoritmo 3 descreve o procedimento.

Algoritmo 3 *Berlekamp Trace Algorithm* adaptado $BTA_adapt(p, t)$

Seja $p(x)$ o polinômio que se deseja encontrar as raízes e t a quantidade de erros que é possível corrigir.

1. Insira $p(x)$ na lista L
2. Enquanto $i < 2^{\lceil \log_2 t \rceil + 1} - 1$ faça
3. f recebe a cabeça da lista L
4. Se f tem grau menor ou igual a 1, retorne a raiz de f.
5. Calcule $\text{mdc}(f, \text{Tr}(\beta_i, x))$ para cada $\beta_i \in B$. Seja $p_0(x) = \text{mdc}(f, \text{Tr}(\beta_k, x))$ o polinômio com grau mais próximo da metade do grau de f.
6. Calcule $p_1(x) = \text{mdc}(f, 1 + \text{Tr}(\beta_k, x))$.
7. Insira $p_0(x)$ e $p_1(x)$ na lista L.

6.0.2. Algoritmo de Fedorenko

O código-fonte escrito para o algoritmo de Fedorenko possui pequenos detalhes feitos para que seu tempo de execução fosse mais constante, como a adição de computações de peso equivalente em cada avaliação condicional, seja ela verdadeira ou falsa. O algoritmo de Fedorenko não possui tempo de execução diretamente ligado a quantidade de raízes do polinômio sendo avaliado. Sendo assim, nenhuma modificação visando melhoria neste aspecto foi feita.

6.1. Análise dos tempos obtidos

A Tabela 1 mostra a média e o desvio padrão dos tempos de execução de cada algoritmo (em minutos) para um código Goppa $\Gamma(L, g(z))$ onde os elementos de L e os coeficientes de $g(z)$ estão em $GF(2^{12})$ e $g(z)$ possui grau $t = 32$.

Foram feitas medições para 10 mensagens diferentes, 5 mensagens com t erros e 5 com $t + 1$ erros. Para cada mensagem foram medidos os tempos de 10 repetições da execução de cada algoritmo. Os valores foram coletados em um computador com processador Intel® Core™ i5-8400 com 2.80 GHz de frequência e 8GB de memória RAM e SageMath na versão 8.6. Para tal, foi feito uso da função *sage_timeit()*[2] e por ser uma medição em software, não muito precisa se comparada a uma medição em ciclos, por

[2] http://doc.sagemath.org/html/en/reference/misc/sage/misc/sage_timeit.html

exemplo, foi considerado para a análise do resultado final apenas o valor mínimo de cada 10 repetições. Isto se deve ao fato de que o valor mínimo indica um limite inferior do quão rápido a máquina pode executar o algoritmo e valores acima disto são geralmente causados por outros processos interferindo na execução. Portanto, a média e o desvio padrão foram feitos a partir dos tempos mínimos da decodificação de cada mensagem.

	Tempo médio para t erros (em segundos)	**Desvio padrão para t erros**	**Tempo médio para $t+1$ erros (em segundos)**	**Desvio padrão para $t+1$ erros**
Análise exaustiva	1.013617	0.017623	0.997102	0.005291
BTA recursivo	3.275600	0.256375	0.149524	0.014667
BTA adaptado	26.972274	0.189169	36.443187	0.420316
Algoritmo de Fedorenko	3.737185	0.023487	3.674740	0.035016

Tabela 1. Tempo de execução dos algoritmos para determinação de raízes de polinômios

Como dito na Seção 4.1.4, um bom algoritmo contra ataques de temporização deve executar em tempo constante independentemente da quantas raízes o polinômio sendo avaliado possui. Através dos resultados temos que a análise exaustiva foi uma boa alternativa em questão de tempo médio e desvio padrão, mas em parte isto se deve aos parâmetros pequenos utilizados para o teste. Como detalhado em [Strenzke 2012], a abordagem exaustiva não é uma boa alternativa para algoritmo de determinação de raízes em criptografia baseada em códigos, pois apresenta outros tipos de vulnerabilidades.

Os resultados para a versão original recursiva do algoritmos Trace de Berlekamp comprovam o problema de variabilidade no seu tempo de execução, por ser influenciado pela quantidade de raízes do polinômio. Pela diferença entre os tempos obtidos para mensagens com t e $t + 1$, um atacante teria sucesso em um ataque de temporização e conseguiria obter as posições dos erros apenas pela avaliação da variabilidade do tempo de execução.

Observando os valores obtidos, vemos que a nova versão criada para o BTA (chamada de BTA adaptado) não se comportou como previsto e não atingiu a expectativa de ser um algoritmo com execução em tempo constante. Os resultados mostram uma diferença de tempo significativa para decodificações de mensagens com quantidade de erros maior que t, o que torna a implementação vulnerável a ataques *side-channel* de temporização.

Um outro ponto negativo da nova versão do BTA foi o seu tempo de execução mais lento quando comparado com o original, o que se deve pelo fato de que as modificações implicam em manter mais variáveis de controle e mais operações em listas, que acaba

sendo custoso. Um outro fator que pode ter influenciado no tempo de execução foi que nas situações em que a mensagem continha mais do que o t máximo de erros que podiam ser corrigidos, a fatoração era feita em menos passos do que o imposto pelo laço criado. Para continuar a execução do laço com o intuito de que, independente da quantidade de raízes, a mesma quantidade de iterações fosse feita, foi decidido por seguir a execução começando a fatoração novamente, ou seja, o polinômio localizador de erros é fatorado mais uma vez e os cálculos são realizados até que as iterações acabem.

Uma outra variação que pode ser feita no algoritmos Trace de Berlekamp é utilizar o algoritmo proposto por Zinoviev [Zinoviev 1996] para determinar as raízes de polinômios com grau $d \leq 10$, que pode reduzir o tempo de execução, como analisado em [Strenzke 2012].

Por fim, o algoritmo de Fedorenko se mostra como uma boa alternativa entre as quatro analisadas, pois apresentou tempos pequenos e mais constantes, que não foram influenciados pela quantidade de raízes do polinômio. Apesar de nestes resultados a análise exaustiva tenha exibido tempos menores, segundo [Fedorenko and Trifonov 2002], dependendo da forma como as operações sobre corpos finitos são implementadas, o algoritmo de Fedorenko pode ser mais rápido que a busca de Chien [Chien 1964], uma adaptação da análise exaustiva.

7. Considerações finais

Neste trabalho, foram discutidos algoritmos para determinação de raízes de polinômios com o intuito de determinar sua vulnerabilidade em relação a ataques de temporização. Entre conceitos apresentados temos o criptossistema de McEliece, códigos de Goppa e seu principal método para realizar a decodificação de mensagens, o algoritmo de Patterson.

Entre os passos do algoritmo de Patterson temos a avaliação de um polinômio, a fim de encontrar suas raízes, que é o propósito dos algoritmos estudados no Capítulo 5. Pelo fato destes algoritmos serem utilizados no criptossistema de McEliece, considerado uma alternativa para ser utilizado em ambientes de alta segurança, há a demanda de que diversos aspectos de segurança sejam cumpridos, entre eles a sua invulnerabilidade a ataques de temporização. Para tal, é necessário que a implementação dos algoritmos tenha tempo de execução constante, o que motivou a adaptação do algoritmo Trace de Berlekamp e pequenas modificações no algoritmo de Fedorenko.

O principal resultado consiste na avaliação da implementação de cada algoritmo de determinação de raízes de polinômios em relação à sua variabilidade de tempo de execução e na comparação de quais poderiam ser as melhores alternativas para serem utilizadas na decodificação de códigos Goppa provendo um tempo de execução constante.

Referências

(2014). Code based cryptography python. Disponível em: <`https://github.com/davidhoo1988/Code_Based_Cryptography_Python`>. Acesso em: 03 de março de 2019.

Berlekamp, E. R. (1970). Factoring polynomials over large finite fields. *Mathematics of computation*, 24(111):713–735.

Bernstein, D. J., Lange, T., and Peters, C. (2008). Attacking and defending the mceliece cryptosystem. In *International Workshop on Post-Quantum Cryptography*, pages 31–46. Springer.

Bucerzan, D., Cayrel, P.-L., Dragoi, V., and Richmond, T. (2017). Improved timing attacks against the secret permutation in the mceliece pkc. *International Journal of Computers Communications & Control*, 12(1):7–25.

Chien, R. (1964). Cyclic decoding procedures for bose-chaudhuri-hocquenghem codes. *IEEE Transactions on information theory*, 10(4):357–363.

Diffie, W. and Hellman, M. (1976). New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654.

Fedorenko, S. V. and Trifonov, P. V. (2002). Finding roots of polynomials over finite fields. *IEEE Transactions on communications*, 50(11):1709–1711.

Goppa, V. D. (1970). A new class of linear correcting codes. *Problemy Peredachi Informatsii*, 6(3):24–30.

Huffman, W. C. and Pless, V. (2010). *Fundamentals of Error-Correcting Codes*. Cambridge University Press, EUA.

Jochemsz, E. (2002). Goppa codes & the mceliece cryptosystem. *Doktorarbeit, Universiteit van Amsterdam*.

MacWilliams, F. and Sloane, N. (1977). *The Theory of Error-Correcting Codes*, volume 16. North-Holland, Países Baixos, 3 edition.

Mceliece, R. J. (1978). A public-key cryptosystem based on algebraic. *Coding Thv*, 4244:114–116.

Niederreiter, H. and Xing, C. (2009). *Algebraic geometry in coding theory and cryptography*. Princeton University Press, EUA, 1 edition.

Patterson, N. (1975). The algebraic decoding of goppa codes. *IEEE Transactions on Information Theory*, 21(2):203–207.

Risse, T. (2011). How sage helps to implement goppa codes and mceliece pkcss. Disponível em: <http://www.weblearn.hs-bremen.de/risse/papers/ICIT11/526_ICIT11_Risse.pdf>. Acessado em 24 de março de 2019.

Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Roering, C. (2013). Coding theory-based cryptopraphy: Mceliece cryptosystems in sage.

Safieddine, R. and Desmarais, A. (2014). Comparison of different decoding algorithms for binary goppa codes.

Shor, P. W. (1999). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332.

Shoufan, A., Strenzke, F., Molter, H. G., and Stöttinger, M. (2010). A timing attack against patterson algorithm in the mceliece pkc. pages 161–175.

Stallings, W. (2014). *Cryptography and network security*. Pearson Education, Estados Unidos, 6 edition.

Strenzke, F. (2011). Timing attacks against the syndrome inversion in code-based cryptosystems. pages 95–107.

Strenzke, F. (2012). Fast and secure root finding for code-based cryptosystems. In *International Conference on Cryptology and Network Security*, pages 232–246. Springer.

Strenzke, F., Tews, E., Molter, H. G., Overbeck, R., and Shoufan, A. (2008). Side channels in the mceliece pkc. pages 216–229.

Zinoviev, V. (1996). *On the Solution of Equations of Degree*. PhD thesis, INRIA.