Carlos Filipe Lombizani De Bernardis

O Problema MQ aplicado a Identificação

Estudo de algoritmos de identificação baseados em sistemas de polinômios quadráticos.

Orientador:
Professor Routo Terada

UNIVERSIDADE DE SÃO PAULO

São Paulo

2013

Resumo

Conquanto problemas de fatoração de números inteiros e de logaritmos discretos possam ser tratados por computadores quânticos, sistemas hodiernos de criptografia ainda os empregam.

O problema MQ, que consiste em resolver um sistema de polinômios multivariados quadráticos sobre um corpo finito, é, até onde se sabe, intratável, até mesmo por computadores quânticos, podendo em teoria ser usado para construir sistemas criptográficos à sua prova.

Neste trabalho, estudamos o problema MQ propriamente e dois esquemas de identificação propostos em 2011 cuja segurança nele se baseia.

Incluimos exemplos numéricos e extensões desses algoritmos.

Sumário

1 Introdução

1.1 Protocolos de Identificação

Em segurança da informação, um objetivo frequente é capacitar uma entidade (o *verificador*) a corroborar que a identidade de outra entidade (o *provador*) corresponde ao declarado, de sorte a impedir personificação por terceiros. Denominam-se as diversas técnicas para tal de *protocolos de identificação* ou *autenticação*.

Normalmente isso se alcança reclamando ao provador que demonstre conhecer um segredo, tal como uma senha ou a solução de um problema. Diz-se que a autenticação é *fraca* quando o protocolo exige que o provador revele o segredo ao verificador, pois, uma vez que o segredo é revelado, o verificador pode daí em diante personificar o provador.

Os protocolos de identificação do tipo *desafio-resposta* aperfeiçoam a autenticação fazendo com que o provador responda a um desafio do verificador, a fim de demonstrar o conhecimento do segredo de maneira variável com o tempo, impedindo que o verificador reutilize diretamente a informação cedida pelo provador. Esse tipo de técnica é chamada *autenticação forte*, pois o segredo em si não é revelado ao longo do protocolo.

Um protocolo desafio-resposta que torna possível a um provador demonstrar conhecimento de um segredo sem revelar qualquer informação que possa ser usada pelo verificador para personificá-lo é dito *de conhecimento-zero*.

1.2 Os Protocolos MQID-3 e MQID-5

Chama-se de *problema MQ* o problema de encontrar soluções para um sistema de polinômios quadráticos multivariados sobre um corpo de Galois. Este problema é NP-completo sobre qualquer corpo, podendo portanto ser usado para se construir protocolos de identificação em que a solução para uma determinada instância do problema seja o segredo do provador.[1] Assim são construídos os protocolos de conhecimento-zero MQID-3 e MQID-5.

Nesses protocolos, um provador começa dividindo seu segredo em partes e depois demonstra a correção de algumas partes dependendo da escolha de um verificador sem revelar o segredo em si. Toda função formada de polinômios quadráticos multivariados, chamada de *função MQ*, possui uma forma polar bilinear que permite dividir o segredo em três partes, possibilitando essa abordagem.

Diversos protocolos baseados no problema MQ já foram propostos, e são conhecidos como *MPKC* (criptografia de chave-pública multivariável). Entretanto, sua segurança é baseada não apenas no problema MQ, mas também em algum problema de isomorfismo de polinômios, para os quais existe criptoanálise em andamento, tendo alguns desses esquemas já até se mostrado inseguros. Os protocolos MQID-3 e MQID-5, baseando-se somente no problema MQ, prometem maior segurança. Por não haver algoritmo quântico eficiente conhecido para resolver o problema MQ, estes protocolos são candidatos a proverem criptografia pós-quântica.

Uma função MQ pode ser usada como função de mão única de entrada e saída pequenas, permitindo que os protocolos MQID-3 e MQID-5 sejam usados com chaves menores do que outros protocolos.

[1] [Patarin e Goubin (1997)]

2 O Problema MQ

2.1 Preliminares

2.1.1 Corpos

Um conjunto $\mathbb{K}$ com ao menos dois elementos, fechado por duas operações denominadas *adição* e *multiplicação* (denotadas respectivamente $+$ e $\cdot$) satisfazendo os axiomas abaixo para quaisquer elementos x, y, e z em $\mathbb{K}$ é chamado de *corpo*.

A *Axiomas da adição*:

1 *Associatividade*: várias ocorrências seguidas da operação adição podem ser efetuadas em qualquer ordem contanto que não se altere a ordem dos termos, i.e. $(x+y)+z = x+(y+z)$

2 *Comutatividade*: a ordem dos termos não altera o resultado da operação, i.e. $x+y = y+x$

3 *Elemento neutro (zero)*: existe um elemento $0 \in \mathbb{K}$ tal que $x+0 = x$

4 *Elemento simétrico*: existe para cada x de $\mathbb{K}$ um elemento simétrico $-x$ tal que $x+(-x) = x-x = 0$

B *Axiomas da multiplicação*:

1 *Associatividade*: várias ocorrências seguidas da operação multiplicação podem ser efetuadas em qualquer ordem contanto que não se altere a ordem dos fatores, i.e. $(x \cdot y) \cdot z = x \cdot (y \cdot z)$

2 *Comutatividade*: a ordem dos fatores não altera o resultado da operação, i.e. $x \cdot y = y \cdot x$

3 *Elemento neutro (um)*: existe um elemento $1 \in \mathbb{K}$ tal que $1 \neq 0$ e $x \cdot 1 = x$

4 *Elemento inverso*: existe para cada $x \neq 0$ de $\mathbb{K}$ um elemento inverso $x^{-1} = \frac{1}{x}$ tal que $x \cdot x^{-1} = \frac{x}{x} = 1$

C *Axioma da distributividade*: as operações de adição e multiplicação relacionam-se de forma tal que $x \cdot (y+z) = x \cdot y + x \cdot z$

Desses axiomas, decorrem as propriedades da unicidade dos elementos neutros, do simétrico e do inverso e as leis de corte. Vejamos.

Suponhamos que haja $\overline{0} \in \mathbb{K}$ tal que $x + \overline{0} = x$. Então $x + \overline{0} - x = x - x \Rightarrow \overline{0} = 0$ (unicidade do zero). Suponhamos que $x \cdot y = x$ para todo $x \in \mathbb{K}$. Então, tomando $x = 1$, temos que $y = 1$ (unicidade do um). Além disso, se y é tal que $x + y = 0$, então $y = 0 - x = -x$ (unicidade do simétrico). Se tivermos a igualdade $x + z = y + z$, então, somando $-z$ a ambos os membros obtemos $x = y$ (lei do corte da adição). Se tivermos $x \cdot z = y \cdot z$ e $z \neq 0$, então, multiplicando z^{-1} a ambos os membros obtemos $x = y$ (lei do corte da multiplicação). Temos também que $x \cdot 0 = 0 \, \forall \, x \in \mathbb{K}$, pois $x \cdot 0 + x = x \cdot 0 + x \cdot 1 = x(0+1) = x \cdot 1 = x$. Disto segue-se que, se $x \cdot y = 0$ e $x \neq 0$, então $x \cdot y = x \cdot 0 \Rightarrow x \cdot y \cdot x^{-1} = x \cdot 0 \cdot x^{-1} \Rightarrow y = 0$, e, portanto, se $x \cdot y = 1$, então $x \neq 0$, $y \neq 0$ e, multiplicando por x^{-1}, $y = x^{-1}$ (unicidade do inverso).

A *característica* de um corpo é o menor número de vezes que é necessário somar o elemento um de modo a se obter o elemento zero. Se esta soma nunca alcançar o elemento neutro da adição, então diz-se que a característica do corpo é zero.

Um corpo é dito *de Galois* se o conjunto de seus elementos é finito. A característica de um corpo de Galois é sempre um número primo. Para verificá-lo, suponhemos que a característica m de um dado corpo $(\mathbb{K}, +, \cdot)$ seja um número composto, isto é, $m = st$ para algum s e algum t com $1 < s < m$ e $1 < t < m$. Como m é, por definição, o menor número de vezes que se deve somar um para obter zero e s e t são menores do que

m, então, se somarmos um s vezes ou t vezes, obteremos elementos de $\mathbb{K}$ diferentes de zero. Por distributividade, temos que $\underbrace{(1+\ldots+1)}_{s\ vezes} \cdot \underbrace{(1+\ldots+1)}_{t\ vezes} = \underbrace{(1+\ldots+1)}_{st=m\ vezes} = 0$. Encontramos pois dois elementos de $\mathbb{K}$ diferentes de zero cujo produto é igual a zero, o que é uma contradição, decorrente da suposição de que m seja composto. Portanto m deve ser obrigatoriamente primo.

O corpo dos números reais com as operações de soma e multiplicação típicas é um exemplo de corpo de característica zero. O conjunto $\mathbb{Z}_n$ é um corpo se e somente se n for primo, e neste caso tem característica n.

2.1.2 Espaços Vetoriais

Um *espaço vetorial sobre* $\mathbb{K}$ consiste num conjunto não-vazio V onde se definem uma operação chamada de *adição*, que associa a cada par de elementos u e v de V um elemento $u+v$ de V, e uma operação chamada de *multiplicação por escalar*, que associa a cada par de elementos k de $\mathbb{K}$ e u de V um elemento ku de V, de maneira que satisfaçam os seguintes axiomas para todo u, v e w em V e todo k e m em $\mathbb{K}$.

1. *Comutatividade da adição*: $u+v=v+u$
2. *Associatividade da adição*: $u+(v+w)=(u+v)+w$
3. *Vetor nulo*: existe um elemento 0 em V tal que $0+u=u+0=u$
4. *Vetor simétrico*: existe para cada u em V um elemento $-u$ em V tal que $u+(-u)=u-u=0$
5. *Distributividade da multiplicação por escalar em relação à adição*: $k(u+v)=ku+kv$
6. *Distributividade da adição em relação à multiplicação por escalar*: $(k+m)u=ku+mu$
7. *Associatividade da multiplicação por escalar*: $k(mu)=(km)(u)$

8. *Elemento neutro da multiplicação por escalar*: multiplicar um vetor u pelo elemento neutro multiplicativo do corpo não altera o vetor, isto é, $1u = u$

Dado um corpo $\mathbb{K}$, o conjunto de todas as ênuplas $(k_1, k_2, \ldots, k_n)$ de elementos de $\mathbb{K}$ com adição e multiplicação por escalar componente a componente forma um espaço vetorial sobre $\mathbb{K}$, que denotamos $\mathbb{K}^n$.

2.1.3 Bilinearidade

Sejam V e W espaços vetoriais sobre o corpo $\mathbb{K}$. Diz-se que uma função $f : V \to W$ é *linear* se, para quaisquer vetores u e v em V e qualquer escalar k em $\mathbb{K}$, as condições $f(u+v) = f(u) + f(v)$ e $f(ku) = kf(u)$ são satisfeitas.

Seja Y também um espaço vetorial sobre $\mathbb{K}$. Uma função $g : V \times W \to Y$ tal que, fixando w qualquer em W, a função $v \mapsto g(v,w)$ é linear de V em Y, e, fixando v qualquer em V, a função $w \mapsto g(v,w)$ é linear de W em Y é dita *bilinear*.

2.2 Função MQ

Definimos a família de funções

$$\mathscr{MQ}(n,m,\mathbb{F}_q) := \left\{ \begin{array}{c} F(x) = (f_1(x), \ldots, f_m(x)) \mid f_l(x) = \sum_{i,j} a_{lij} x_i x_j + \sum_i b_{li} x_i \, ; \\ a_{lij}, b_{li} \in \mathbb{F}_q \; \forall \, l \in \{1, \ldots, m\} \end{array} \right\} \tag{2.1}$$

onde $n, m \in \mathbb{N}$, $\mathbb{F}_q$ é um corpo de Galois de cardinalidade q e $x = (x_1, \ldots, x_n) \in \mathbb{F}_q^n$. Uma função qualquer $F : \mathbb{F}_q^n \to \mathbb{F}_q^m \in \mathscr{MQ}(n,m,\mathbb{F}_q)$ constitui um sistema de polinômios de grau 2, e recebe o nome de *função MQ*. Por simplicidade, os polinômios não possuem termos constantes, pois isto não afetará a complexidade do problema MQ.

Uma função $G : \mathbb{F}_q^n \times \mathbb{F}_q^n \to \mathbb{F}_q^m$ da forma $G(x,y) := F(x+y) - F(x) - F(y)$ é chamada *forma polar de* F. Podemos obter uma fórmula para as componentes g_l da função

G:

$$\forall\, l \in \{1,\ldots,m\},\ g_l(x,y) := f_l(x+y) - f_l(x) - f_l(y)$$

$$\begin{aligned}
f_l(x+y) &= \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_i+y_i)(x_j+y_j) + \sum_{i=1}^{n} b_{li}(x_i+y_i) = \\
&= \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_ix_j + x_iy_j + x_jy_i + y_iy_j) + \sum_{i=1}^{n} b_{li}x_i + \sum_{i=1}^{n} b_{li}y_i = \\
&= (\sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}x_ix_j + \sum_{i=1}^{n} b_{li}x_i) + (\sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}y_iy_j + \sum_{i=1}^{n} b_{li}y_i) + \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_iy_j + x_jy_i) = \\
&= f_l(x) + f_l(y) + \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_iy_j + x_jy_i)
\end{aligned}$$

$$\begin{aligned}
&\implies g_l(x,y) = \cancel{f_l(x)} + \cancel{f_l(y)} + \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_iy_j + x_jy_i) - \cancel{f_l(x)} - \cancel{f_l(y)} \\
&\implies g_l(x,y) = \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_iy_j + x_jy_i),\ \forall\, l \in \{1,\ldots,m\}
\end{aligned} \tag{2.2}$$

A função G é bilinear. A demonstração decorre da fórmula acima:

$$\begin{aligned}
g_l(x,y+z) &= \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_i(y_j+z_j) + x_j(y_i+z_i)) = \\
&= \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_iy_j + x_iz_j + x_jy_i + x_jz_i) = \\
&= \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_iy_j + x_jy_i) + \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_iz_j + x_jz_i) = \\
&= g_l(x,y) + g_l(x,z)
\end{aligned}$$

$$
\begin{aligned}
g_l(x,ky) &= \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_i k y_j + x_j k y_i) = \\
&= \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij} k (x_i y_j + x_j y_i) = \\
&= k \sum_{i=1}^{n}\sum_{j=1}^{i} a_{lij}(x_i y_j + x_j y_i) = \\
&= k g_l(x,y) \\
&\forall\, l \in \{1,...,m\};\ k \in \mathbb{F}_q;\ x,y,z \in \mathbb{F}_q^n \\
\implies & G(x,y+z) = G(x,y) + G(x,z);\ G(x,ky) = kG(x,y) \qquad \square
\end{aligned}
$$

Como G é simétrica, prova-se de forma análoga a linearidade na primeira variável.

2.3 Problema

Dados $\mathbb{F}_q$ um corpo de Galois de cardinalidade q, $n,m \in \mathbb{N}$, $F : \mathbb{F}_q^n \to \mathbb{F}_q^m \in \mathscr{MQ}(n,m,\mathbb{F}_q)$ uma função MQ e $y = (y_1,\ldots,y_m) \in \mathbb{F}_q^m$, o *problema MQ* consiste em encontrar $x = (x_1,\ldots,x_n) \in \mathbb{F}_q^n$ tal que $F(x) = y$, isto é, $f_i(x_1,\ldots,x_n) = y_i\ \forall\, i \in \{1,\ldots,m\}$.

Definindo m matrizes A_l triangulares $n \times n$ e m vetores b_l de dimensão n da forma

$$
A_l := \begin{bmatrix}
a_{l11} & 0 & 0 & \ldots & 0 \\
a_{l21} & a_{l22} & 0 & \ldots & 0 \\
a_{l31} & a_{l32} & a_{l33} & \ldots & 0 \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
a_{ln1} & a_{ln2} & a_{ln3} & \ldots & a_{lnn}
\end{bmatrix},\ b_l := \begin{bmatrix} b_{l1} \\ b_{l2} \\ b_{l3} \\ \vdots \\ b_{ln} \end{bmatrix}, \tag{2.3}
$$

sendo a_{lij} e b_{li} os coeficientes de F, podemos escrever F matricialmente, permitindo-

nos expressar o problema como encontrar $x \in \mathbb{F}_q^n$ tal que

$$\begin{bmatrix} x^T A_1 x + b_1^T x \\ \vdots \\ x^T A_m x + b_m^T x \end{bmatrix} = \begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}. \tag{2.4}$$

Esta representação permite-nos ver claramente o número de parâmetros necessários para caracterizar um problema MQ. Os coeficientes de F cabem nas m matrizes A_l e vetores b_l, que têm $\frac{n^2+n}{2}$ (por serem triangulares) e n elementos significativos respectivamente, totalizando $m(\frac{n^2+n}{2}+n)$ parâmetros para descrever o sistema.

Há várias técnicas para resolver este problema, entre as quais as melhores são baseadas em bases de Gröbner e busca por força bruta. Entretanto, mesmo os mais refinados algoritmos possuem complexidade exponencial. O melhor algoritmo conhecido para $m = n \leq 200$ foi concebido por Bouillaguet et al. e possibilita resolver o problema MQ para F em $\mathscr{MQ}(n,m,\mathbb{F}_2)$ com complexidade $2^{n+2} \cdot \log_2 n$. [1] Não se conhece algoritmo quântico eficiente para resolver o problema MQ.

Exemplo 2.1. Seja $\mathbb{Z}_2$ o nosso corpo de Galois, cuja cardinalidade é $q = 2$, e seja $F : \mathbb{Z}_2^2 \to \mathbb{Z}_2^2$ a seguinte

$$\begin{cases} f_1(x_1,x_2) = x_1^2 + x_2^2 \\ f_2(x_1,x_2) = x_1x_2 + x_1 + x_2 \end{cases},$$

com $n = 2$ e $m = 2$, isto é, o nosso sistema tem 2 incógnitas e 2 equações. Dado y da seguinte forma

$$y = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

há $x \in \mathbb{F}_q^n$ tal que $F(x) = y$? Afirmamos que $x = (1,0)$ satisfaz $F(x) = y$. Verifica-se

[1] [Bouillaguet *et al.* (2010)Bouillaguet, Chen, Cheng, Chou, Niederhagen, Shamir, e Yang]

facilmente:

$$\begin{aligned} f_1(1,0) &= 1^2+0^2 = 1 \\ f_2(1,0) &= 1\cdot 0+1+0 = 1 \end{aligned} \qquad \square$$

Exemplo 2.2. Seja $\mathbb{Z}_2$ o nosso corpo de Galois, cuja cardinalidade é $q = 2$, e seja $F : \mathbb{Z}_2^5 \to \mathbb{Z}_2^4$ a seguinte

$$\begin{cases} f_1(x_1,\ldots,x_5) = x_1^2+x_2^2+x_2x_3+x_1x_5+x_2x_5+x_3x_5+x_1 \\ f_2(x_1,\ldots,x_5) = x_1^2+x_1x_3+x_3x_5+x_2+x_3 \\ f_3(x_1,\ldots,x_5) = x_2^2+x_2x_3+x_1x_4+x_2x_4+x_3x_4+x_2+x_3+x_4 \\ f_4(x_1,\ldots,x_5) = x_1^2+x_1x_2+x_1x_3+x_3^2+x_1x_4+x_1x_5+x_2x_5+x_3x_5+x_4x_5+x_1+x_2+x_4 \end{cases},$$

com $n = 5$ e $m = 4$, isto é, o nosso sistema tem 5 incógnitas e 4 equações. Dado y da seguinte forma

$$y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix},$$

há $x \in \mathbb{F}_q^n$ tal que $F(x) = y$? Afirmamos que $x = (1,0,1,0,1)$ satisfaz $F(x) = y$. Verifica-se facilmente:

$$\begin{aligned} f_1(1,0,1,0,1) &= 1^2+0^2+0\cdot 1+1\cdot 1+0\cdot 1+1\cdot 1+1 = 0 \\ f_2(1,0,1,0,1) &= 1^2+1\cdot 1+1\cdot 1+0+1 = 0 \\ f_3(1,0,1,0,1) &= 0^2+0\cdot 1+1\cdot 0+0\cdot 0+1\cdot 0+0+1+0 = 1 \\ f_4(1,0,1,0,1) &= 1^2+1\cdot 0+1\cdot 1+1^2+1\cdot 0+1\cdot 1+0\cdot 1+1\cdot 1+0\cdot 1+1+0+0 = 0 \end{aligned} \qquad \square$$

3 *Identificação Conhecimento-Zero*

Protocolos de identificação de conhecimento-zero permitem demonstrar conhecimento de um segredo sem revelar informação alguma que possa ser usada pelo verificador para expressar essa mesma demonstração a outrem.

Eles são projetados para funcionar sem o uso de assinaturas digitais, criptografia de chave-pública e cifras de bloco. Eles são em muito semelhantes aos protocolos desafio-resposta ordinários, mas são baseados nos conceitos de sistemas de prova interativa e provas de conhecimento-zero, empregando números aleatórios como desafios e também como *compromissos* para prevenção de fraudes.

3.1 Esboço

Um *sistema de prova interativa* é um sistema no qual provador e verificador trocam várias mensagens (*desafios* e *respostas*), geralmente em função de números aleatórios. O objetivo do provador é demonstrar ao verificador a veracidade de uma afirmação, por exemplo o dito conhecimento de um segredo. O verificador pode aceitar ou rejeitar a demonstração. Entretanto, ao contrário das demonstrações matemáticas tradicionais, as demonstrações fornecidas pelo provador não são absolutas, mas probabilísticas. Isso significa que uma demonstração neste contexto necessita estar correta apenas com certa probabilidade, talvez arbitrariamente próxima de 1.

Provas interativas para fins de identificação são formuladas como provas de conhecimento. *Alice* tem posse de um segredo s, e pretende convencer *Beto* de que possui

conhecimento de s respondendo corretamente questões que envolvem informações públicas e funções padronizadas e requerem conhecimento de s para serem respondidas corretamente.

3.2 Definições

3.2.1 Probabilidade Desprezível

Quando a probabilidade ε de um evento é expressada em função de um parâmetro inteiro positivo l de segurança ($\varepsilon : \mathbb{Z}^+ \to [0,1]$), dizemos que a probabilidade é *desprezível* se, para todo $d \in \mathbb{N}$, existe $l_0 \in \mathbb{Z}^+$ tal que, se $l > l_0$, então $\varepsilon(l) \leq \frac{1}{l^d}$.

3.2.2 Correção

Um protocolo de identificação é dito *correto* se funciona apropriadamente com participantes honestos, isto é, a probabilidade de que um provador honesto seja rejeitado por um verificador honesto é desprezível. Em particular, se essa probabilidade é nula então dizemos que a correção é *perfeita*.

3.2.3 Solidez

Dizemos que um algoritmo é *eficiente* se ele pode ser computado em tempo polinomial em função do tamanho n de sua entrada, isto é, o algoritmo é $O(f(n))$ e existe $k \in \mathbb{R}^+$ tal que $f(n) \leq n^k$ para n suficientemente grande.

Dizemos que um protocolo de identificação é *sólido* se funciona apropriadamente com um provador desonesto, isto é, existe um algoritmo eficiente tal que, se um determinado provador desonesto pode executar com sucesso o protocolo com um verificador honesto com probabilidade não-desprezível, então esse algoritmo pode ser usado para, à partir desse provador, extrair conhecimento suficiente para permitir execuções bem-sucedidas consecutivas com probabilidade não-desprezível do protocolo.

Em outras palavras, para ser capaz de personificar um provador honesto com probabilidade não-desprezível, faz-se necessário conhecer o seu segredo (ou algo equivalente). Dessa forma, a solidez garante que o protocolo de fato nos dá uma prova de conhecimento do segredo, pois ele é requerido para que o protocolo possa ser executado com sucesso. Por via de regra, estabelece-se a solidez de um protocolo supondo a existência de um provador desonesto capaz de executar o protocolo com sucesso e demonstrando que isso nos permite computar o segredo do provador verdadeiro.

3.2.4 Função de Hash

Uma função $h : \{0,1\}^* \to \{0,1\}^n$ é dita *de hash* se houver um algoritmo eficiente para computá-la e se levar cadeias de bits de tamanho arbitrário a cadeias de tamanho n fixo (denominadas *hashes*).

Pelo princípio da casa do pombo, haverá cadeias diferentes que serão levadas no mesmo hash. A esse evento damos o nome de *colisão*. Uma função de hash é dita *à prova de colisões* ou *resitente a colisão* quando não houver algoritmo eficiente que encontre colisões, isto é, que encontre entradas diferentes cujos hashes são idênticos.

3.2.5 Função de mão única

Uma função $f : A \to B$ é dita *de mão única* se houver um algoritmo eficiente para computá-la, porém nenhum algoritmo eficiente para invertê-la, isto é, dado $y \in Im(f)$, encontrar $x \in A$ tal que $f(x) = y$.

Como visto no capítulo anterior, a função MQ é uma função de mão única.

3.2.6 Indistinguibilidade

Sejam P_1 e P_2 duas distribuições de probabilidade em $\{0,1\}^n$, isto é, no espaço de cadeias de bits de tamanho n. Dizemos que P_1 e P_2 são *computacionalmente indistin-*

guíveis se, para todo algoritmo eficiente $A : \{0,1\}^n \to \{0,1\}$,

$$|\Pr_{x \leftarrow P_1}[A(x) = 1] - \Pr_{x \leftarrow P_2}[A(x) = 1]|$$

é desprezível.

Dizemos que P_1 e P_2 são *estatisticamente indistinguíveis* se

$$\Delta(P_1, P_2) = \frac{1}{2} \sum_{\alpha \in \{0,1\}^n} |Pr[P_1 = \alpha] - Pr[P_2 = \alpha]|$$

é desprezível.

Dizemos que P_1 e P_2 são *indistinguíveis* quando forem ou idênticas, ou computacionalmente indistinguíveis, ou estatisticamente indistinguíveis.

3.3 Protocolos de Conhecimento-zero

3.3.1 Propriedade de Conhecimento-zero

Uma prova interativa é dita ser *prova de conhecimento* se tiver as propriedades de correção e solidez.

Um protocolo que é prova de conhecimento tem a propriedade de *conhecimento-zero* se há um algoritmo eficiente (denominado *simulador*) capaz de produzir, à partir da entrada da afirmação a ser verificada e sem interação com o provador verdadeiro, transcrições indistinguíveis das resultantes da interação com o provador verdadeiro.

Esta propriedade implica que a execução do protocolo por um provador não revela nenhuma informação que não seja computável à partir de informações públicas, ainda que o provador esteja interagindo com um verificador desonesto. Entretanto, ela não garante a segurança do protocolo a não ser que seja difícil computacionalmente de se descobrir o segredo.

O conhecimento-zero é dito *perfeito* quando a distribuição de probabilidade do si-

mulador e da interação entre provador e verificador honestos são idênticas. Se forem computacionalmente indistinguíveis, então o conhecimento-zero é dito *computacional*. Finalmente, se forem estatisticamente indistinguíveis, então o conhecimento-zero é dito *estatístico*.

3.3.2 Esquemas de Comprometimento

Um *esquema de comprometimento* serve para ocultar temporariamente uma determinada informação, com a possibilidade de revelá-la posteriormente sem alterações, isto é, sem que se possa modificar a informação após ter-se comprometido com ela.

Esquemas de comprometimento são particularmente úteis em provas de conhecimento-zero. Eles permitem ao provador dividir seu segredo e apresentar ao verificador a opção de qual parte verificar, podendo o provador comunicar todo o segredo de antemão escondido na forma de um compromisso e depois revelar somente o que corresponde à escolha do verificador.

A definição formal de um esquema de comprometimento de cadeias C é a de um protocolo em duas fases que faz uso de um algoritmo de compromisso de cadeias Com, que recebe dois parâmetros s e ρ, onde s é a informação a ser ocultada e ρ é uma cadeia aleatória denominada *sal*. Na primeira fase, o provador utiliza Com para computar um valor de compromisso c, que é enviado ao verificador. Na segunda fase, o provador envia (s,ρ) ao verificador, que por sua vez verifica se $c = Com(s,\rho)$.

Um esquema de comprometimento seguro é *ocultante* e *computacionalmente vinculante*. Essas propriedades significam que dois valores de compromisso quaisquer são indistinguíveis (i.e. é difícil extrair quaisquer informações a respeito de s à partir de c) e que não há algoritmo eficiente capaz de encontrar $s' \neq s$ e ρ' não necessariamente igual a ρ tais que $Com(s',\rho') = Com(s,\rho)$, respectivamente.

É possível construir esquemas de comprometimento de cadeias seguros à partir de funções de hash à prova de colisões. [1]

[1] [Halevi e Micali (1996)]

3.3.3 Estrutura Geral dos Protocolos

Protocolos de identificação de conhecimento-zero canônicos são compostos de 3 passos.

$$\text{Alice} \longrightarrow \text{Beto: Comprometimento}$$
$$\text{Alice} \longleftarrow \text{Beto: Desafio}$$
$$\text{Alice} \longrightarrow \text{Beto: Resposta}$$

Inicialmente, o provador, que alega ser Alice, seleciona como compromisso secreto um elemento aleatório de um conjunto predefinido e à partir dele computa um compromisso público. Isso define um conjunto de perguntas que o provador alega ser capaz de responder corretamente, daí em diante restringindo a priori suas possíveis respostas. O desafio sucessivo de Beto seleciona uma dessas perguntas. Alice então fornece sua resposta, que Beto verifica se está correta. Se necessário, o protocolo pode ser reiterado a fim de minimizar a probabilidade de embuste.

Normalmente utiliza-se uma função de mão única para determinar os segredos dos provadores, sendo os elementos do domínio da função candidatos a segredo, e a sua imagem os valores que permitirão verificá-los, pois isso torna fácil verificar a veracidade dos segredos e ao mesmo tempo difícil de descobri-los. O segredo de um provador é chamado de *chave secreta*, e o valor que permite a um verificador verificá-lo é a sua respectiva *chave pública*.

3.4 Esquemas de Identificação

Um esquema de identificação é composto por um protocolo de identificação mais a infraestrutura de chaves e os algoritmos necessários para preparar o seu uso, que são os algoritmos de inicialização e de geração de chaves.

Formalmente, um esquema de identificação é uma tupla de algoritmos $(Inicia, Gera, P, V)$. *Inicia* recebe um parâmetro de segurança λ e devolve um conjunto de parâmetros do sistema. *Gera* é um algoritmo de geração de chaves que recebe os parâmetros devolvidos por *Inicia* e devolve um par de chaves pública e secreta. Finalmente, o par (P,V), de provador e verificador respectivamente, constitui o protocolo de identificação correspondente. Eles recebem os parâmetros de sistema e uma chave pública e P recebe também uma chave secreta. Após a interação protocolar entre ambos, V devolve um valor booleano representando o resultado da verificação.

4 *Protocolos de Identificação MQID*

Em 2011, os pesquisadores japoneses Koichi Sakumoto, Taizo Shirai, e Harunaga Hiwatari apresentaram uma proposta de protocolos de identificação de conhecimento-zero baseados somente no problema MQ. Até então, todos os sistemas criptográficos propostos em cima do problema MQ baseavam-se também em algum problema de isomorfismo de polinômios, isto é, utilizavam-se de uma função MQ não-aleatória e duas transformações afins para levá-la a uma instância aparentemente aleatória de uma função MQ.

Isso faz da proposta dos pesquisadores japoneses uma ótima candidata para criptografia pós-quântica, isto é, para substituir os atuais sistemas criptográficos vulneráveis aos vindouros computadores quânticos, pois, como vimos anteriormente, não se conhece algoritmo quântico eficiente para resolver o problema MQ, fazendo-o intratável até mesmo por computadores quânticos.

A proposta é composta de dois protocolos de identificação, um canônico de 3 passos a que nos referimos por *MQID-3* e um de 5 passos que chamamos *MQID-5*.

4.1 Princípios

Dada $F : \mathbb{F}_q^n \to \mathbb{F}_q^m \in \mathscr{MQ}(n,m,\mathbb{F}_q)$ uma função MQ, $v \in \mathbb{F}_q^m$ e $s \in \mathbb{F}_q^n$ tais que $F(s) = v$, temos, da equação 2.4 :

$$F(s) = \begin{bmatrix} s^T A_1 s + b_1^T s \\ \vdots \\ s^T A_m s + b_m^T s \end{bmatrix} = \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix} = v$$

A_l e b_l são parâmetros do sistema e portanto comuns a todos seus usuários, s é a chave secreta, isto é, o segredo sabido apenas pelo provador, e v é a chave pública, isto é, informação partilhada que permite a uma parte verificar a identidade do provador que alegar conhecer s.

A técnica usada para dividir o segredo s e demonstrar seu conhecimento consiste em utilizar a forma polar G de F. Sejam r_0 e r_1 em $\mathbb{F}_q^n$ quaisquer tais que $s = r_0 + r_1$. A chave pública $v = F(r_0 + r_1)$ pode ser representada da forma $v = F(r_0) + F(r_1) + G(r_0, r_1)$ usando a forma polar. Esta representação não obstante contém ainda um termo dependente de ambos r_0 e r_1.

Considere então a divisão adicional de r_0 em $r_0 = t_0 + t_1$ para algum t_0 e algum t_1 em $\mathbb{F}_q^n$, e de $F(r_0)$ em $F(r_0) = e_0 + e_1$ para algum e_0 e algum e_1 em $\mathbb{F}_q^m$. Usando a bilinearidade de G, a chave pública pode ser dividida em duas partes:

$$\begin{aligned} v &= e_0 + e_1 + F(r_1) + G(t_0 + t_1, r_1) \\ &= e_0 + e_1 + F(r_1) + G(t_0, r_1) + G(t_1, r_1) \\ \implies v &= (G(t_0, r_1) + e_0) + (F(r_1) + G(t_1, r_1) + e_1) \end{aligned} \tag{4.1}$$

Cada parte depende apenas ou da tupla (r_1, t_0, e_0) ou da tupla (r_1, t_1, e_1), e nenhuma informação sobre a chave secreta s pode ser obtida de apenas uma das duas tuplas.

Faz-se uso de um esquema de comprometimento de cadeias C estatisticamente ocultante e computacionalmente vinculante arbitrário.

4.2 Inicialização e Geração de Chaves

Os algoritmos de inicialização e geração de chaves que compõem os esquemas completos são idênticos para as duas versões do protocolo MQID.

Sejam $n = n(\lambda)$, $m = m(\lambda)$ e $q = q(\lambda)$ funções polinomialmente limitadas de um parâmetro de segurança λ. O algoritmo de inicialização *Inicia* escolhe como parâmetro de sistema F aleatoriamente em $\mathscr{MQ}(n,m,\mathbb{F}_q)$. O algoritmo gerador de chaves *Gera* recebe F, escolhe um vetor aleatório $s \in \mathbb{F}_q^n$, computa $v = F(s)$ e devolve (v,s).

4.3 Protocolo MQID-3 de 3 passos

Dado que, se $(r_0, r_1, t_0, t_1, e_0, e_1)$ são tais que

$$G(t_0, r_1) + e_0 = v - F(r_1) - G(t_1, r_1) - e_1 \tag{4.2}$$

e

$$(t_0, e_0) = (r_0 - t_1, F(r_0) - e_1) \tag{4.3}$$

então $v = F(r_0 + r_1)$, basta logo o provador demonstrar que possui uma tupla $(r_0, r_1, t_0, t_1, e_0, e_1)$ que satisfaz as igualdades 4.2 e 4.3 para provar conhecimento do segredo.

No entanto, revelar esta tupla seria equivalente a revelar o segredo completo. Portanto, no protocolo MQID-3 o provador revela, de acordo com um desafio $Ch \in \{0,1,2\}$ do verificador, ou a tupla (r_0, t_1, e_1), ou a tupla (r_1, t_1, e_1), ou a tupla (r_1, t_0, e_0). O verificador pode então conferir cada membro das equações 4.2 e 4.3 usando uma das três tuplas. Como r_0, t_0 e e_0 são escolhidos aleatoriamente, o verificador não obtém nenhuma informação a respeito da chave secreta s à partir dela.

O protocolo MQID-3 é descrito na figura 4.1, onde *Com* é o algoritmo de compromisso de cadeias de C e "$\in_R$" denota uma escolha de um elemento aleatório pertencente a um conjunto finito. Por simplicidade, a cadeia aleatória ρ é omitida.

O verificador aceita o provador se ambas as verificações de "$\stackrel{?}{=}$" resultarem em

Provador, entrada: $((F,v),s)$		**Verificador**, entrada: (F,v)
Escolha $r_0,t_0 \in_R \mathbb{F}_q^n$, $e_0 \in_R \mathbb{F}_q^m$		
$r_1 \leftarrow s - r_0$, $t_1 \leftarrow r_0 - t_0$		
$e_1 \leftarrow F(r_0) - e_0$		
$c_0 \leftarrow Com(r_1, G(t_0,r_1) + e_0)$		
$c_1 \leftarrow Com(t_0,e_0)$		
$c_2 \leftarrow Com(t_1,e_1)$		
	$\xrightarrow{(c_0,c_1,c_2)}$	
		Escolha $Ch \in_R \{0,1,2\}$
	$\xleftarrow{Ch}$	
Se $Ch = 0$, $Rsp \leftarrow (r_0,t_1,e_1)$		
Se $Ch = 1$, $Rsp \leftarrow (r_1,t_1,e_1)$		
Se $Ch = 2$, $Rsp \leftarrow (r_1,t_0,e_0)$		
	$\xrightarrow{Rsp}$	
		Se $Ch = 0$, leia $Rsp = (r_0,t_1,e_1)$, verifique
		$c_1 \stackrel{?}{=} Com(r_0 - t_1, F(r_0) - e_1)$
		$c_2 \stackrel{?}{=} Com(t_1,e_1)$
		Se $Ch = 1$, leia $Rsp = (r_1,t_1,e_1)$, verifique
		$c_0 \stackrel{?}{=} Com(r_1, v - F(r_1) - G(t_1,r_1) - e_1)$
		$c_2 \stackrel{?}{=} Com(t_1,e_1)$
		Se $Ch = 2$, leia $Rsp = (r_1,t_0,e_0)$, verifique
		$c_0 \stackrel{?}{=} Com(r_1, G(t_0,r_1) + e_0)$
		$c_1 \stackrel{?}{=} Com(t_0,e_0)$

Figura 4.1: Descrição do protocolo MQID-3

igualdades. Se não, ele o rejeita. Como o verificador sempre aceita um provador honesto, então o protocolo é correto. Ademais, se C é computacionalmente vinculante, verifica-se que um provador desonesto tem $2/3$ de probabilidade de personificar com êxito o provador honesto em cada execução do protocolo. Isso significa que, dadas suficientes iterações do protocolo, a probabilidade de personificação bem sucedida em todas as iterações pode ser tão pequena quanto se queira, e portanto desprezível. Isso implica que o protocolo é sólido. O protocolo MQID-3 é portanto prova de conhecimento, pois possui correção e solidez. Se o esquema de comprometimento de cadeias C é estatisticamente ocultante, então verifica-se que o protocolo MQID-3 possui a propriedade de conhecimento-zero estatístico. Como além disso o problema MQ é computacionalmente difícil, então o protocolo é considerado seguro.

Exemplo 4.1. (*Execução do protocolo MQID-3 com parâmetros artificialmente pequenos*) Seja o parâmetro de sistema F do nosso esquema MQID-3 de exemplo a função MQ descrita no exemplo 2.1, a saber:

$$A_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, A_2 = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, b_1 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \text{ e } b_2 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

Suponhamos, para simplificar, que o algoritmo de compromisso de cadeias *Com* utilizado em nosso esquema seja somente a soma dos dois parâmetros s e ρ (i.e. s é concatenado e interpretado como uma representação de um número inteiro que é então somado ao sal ρ). Observe que o esquema de comprometimento resultante não é nem ocultante nem computacionalmente vinculante, servindo portanto apenas de exemplo.

Alice utiliza *Gera* e recebe como chaves secreta e pública aleatórias os vetores x e y do exemplo 2.1, respectivamente. A chave secreta de Alice é então $s = (1,0)$ e sua chave pública é $v = (1,1)$.

Alice deseja então provar sua identidade para Beto. O seguinte se sucede:

1. Alice escolhe aleatoriamente $r_0 = (1,1)$, $t_0 = (0,0)$, $e_0 = (1,1)$ e $\rho = 15$.
2. Alice computa $r_1 = (1,0) - (1,1) = (0,1)$, $t_1 = (1,1) - (0,0) = (1,1)$ e $e_1 =$

$F(1,1)-(1,1)=(1,0)$.

3. Alice computa os valores de compromisso $c_0 = 22$, $c_1 = 18$ e $c_2 = 29$ e os envia a Beto.

4. Beto escolhe aleatoriamente $Ch = 1$ e o envia a Alice.

5. Alice responde enviando a Beto o vetor (r_1, t_1, e_1, ρ).

6. Beto computa, com os dados fornecidos por Alice:

   (a) $v - F(r_1) - G(t_1, r_1) - e_1 = (1,1) - (1,1) - (0,1) - (1,0) = (1,1)$;

   (b) $Com(r_1, (1,1), \rho) = 22$ e $Com(t_1, e_1, \rho) = 29$.

7. Beto aceita a identidade de Alice pois $c_0 = 22$ e $c_2 = 29$.

□

4.4 Protocolo MQID-5 de 5 passos

No protocolo de 5 passos, o provador divide sua chave secreta s e sua chave pública $F(s)$ em $s = r_0 + r_1$ e $F(s) = F(r_0 + r_1) = F(r_0) + F(r_1) + G(r_0, r_1)$, respectivamente. A diferença do protocolo de 3 passos é que r_0 e $F(r_0)$ são divididos em $\alpha r_0 = t_0 + t_1$ e $\alpha F(r_0) = e_0 + e_1$, onde $\alpha \in \mathbb{F}_q$ é escolhido pelo verificador. Após enviar (t_1, e_1) ao verificador, segundo um desafio $Ch \in \{0,1\}$, o provador revela apenas um dentre os dois vetores r_0 e r_1. Quando r_0, t_0 e e_0 são escolhidos aleatoriamente, o verificador não pode obter informação alguma sobre a chave secreta s à partir de apenas um dos dois vetores. Por outro lado, a prova é prova de conhecimento pois, para mais de uma escolha de $\alpha \in \mathbb{F}_q$, um personificador é incapaz de responder a ambos os desafios possíveis do verificador a não ser que conheça uma solução s para o problema MQ em v.

O protocolo MQID-5 é descrito na figura 4.2, onde *Com* é o algoritmo de compromisso de cadeias de C e "$\in_R$" denota uma escolha de um elemento aleatório pertencente a um conjunto finito. Por simplicidade, a cadeia aleatória ρ é omitida.

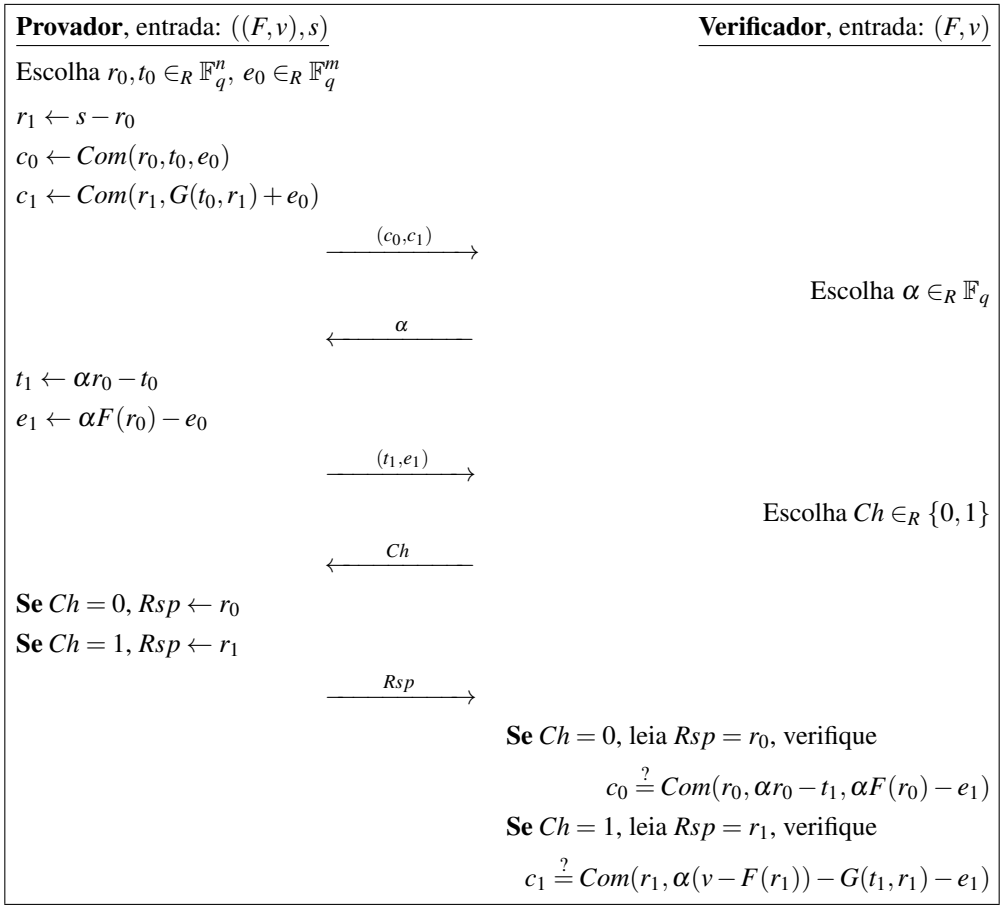

Figura 4.2: Descrição do protocolo MQID-5

O verificador aceita o provador se a verificação de "$\stackrel{?}{=}$" resultar em igualdade. Se não, ele o rejeita. Como anteriormente, o verificador sempre aceita um provador honesto, sendo o protocolo portanto correto. Se C é computacionalmente vinculante, verifica-se que um provador desonesto tem $1/2 + 1/2q$ de probabilidade de executar o protocolo com êxito com um verificador honesto. Isso significa que, dadas suficientes iterações do protocolo, a probabilidade de personificação bem sucedida em todas as iterações pode ser tão pequena quanto se queira, e portanto desprezível. O protocolo é portanto sólido. Logo, o protocolo MQID-5 é prova de conhecimento. Se C é estatisticamente ocultante, então verifica-se ademais que o protocolo é de conhecimento-zero estatístico. Como além disso o problema MQ é computacionalmente difícil, então o protocolo é considerado seguro.

Exemplo 4.2. (*Execução do protocolo MQID-5 com parâmetros artificialmente pequenos*) Suponhamos que Alice e Beto sejam os mesmos e usem o mesmo esquema do exemplo 4.1, exceto que desta vez eles querem utilizar o protocolo de 5 passos para efetuar a identificação. Suponhamos também que Alice sorteia os mesmos valores para ρ e os vetores r_0, t_0 e e_0, por simplicidade. O seguinte se sucede:

1. Alice escolhe aleatoriamente $r_0 = (1,1)$, $t_0 = (0,0)$, $e_0 = (1,1)$ e $\rho = 15$.
2. Alice computa $r_1 = (1,0) - (1,1) = (0,1)$.
3. Alice computa os valores de compromisso $c_0 = 66$ e $c_1 = 22$ e os envia a Beto.
4. Beto escolhe aleatoriamente $\alpha = 0$ e o envia a Alice.
5. Alice computa $t_1 = 0 \cdot (1,1) - (0,0) = (0,0)$ e $e_1 = 0 \cdot (0,l) - (1,1) = (1,1)$ e os envia a Beto.
6. Beto escolhe aleatoriamente $Ch = 1$ e o envia a Alice.
7. Alice envia r_1 para Beto.
8. Beto computa, com os dados fornecidos por Alice:

(a) $\alpha(v - F(r_1)) - G(t_1, r_1) - e_1 = (0,0) - (0,0) - (1,1) = (1,1)$;

(b) $Com(r_1, (1,1), \rho) = 22$.

9. Beto aceita a identidade de Alice pois $c_1 = 22$.

□

5 *Aplicações*

Dentre os principais usos de identificação está simplificar o controle de acesso a um recurso quando o direito de acesso está ligado a uma identidade em particular. Ela pode ser usada também para rastrear o uso de recursos a fim de limitá-lo ou cobrá-lo.

Identificação é geralmente um requisito em protocolos de troca de chaves e está particularmente relacionada a esquemas de assinatura digital. Em alguns casos, esquemas de identificação podem ser convertidos em esquemas de assinatura digital usando técnicas padronizadas. Neste capítulo vamos explorar essa aplicação e uma melhoria do protocolo MQID-3 indicada para aplicações em assinaturas digitais e em dispositivos embarcados com limitações de energia.

5.1 Assinaturas Digitais

Assinaturas digitais são como as correspondentes digitais das assinaturas manuscritas. Elas permitem verificar se uma parte de fato assinou determinada mensagem. A *assinatura digital* de uma mensagem é um número dependente da mensagem em si e de algum segredo conhecido apenas pelo assinante. Ela deve ser verificável sem que seja necessário o conhecimento da informação secreta do assinante.

Um *esquema de assinatura digital* é uma tupla de algoritmos $(Inicia, Gera, Assina, Verifica)$. *Inicia* recebe um parâmetro de segurança κ e devolve um conjunto de parâmetros do sistema. *Gera* é um algoritmo de geração de chaves que recebe os parâmetros devolvidos por *Inicia* e devolve um par de chaves pública e secreta. *Assina* é um algoritmo

probabilístico que emite uma assinatura σ à partir de uma chave secreta e uma certa mensagem. *Verifica* é um algoritmo determinístico que recebe uma assinatura σ e uma chave pública e devolve um valor booleano representando o resultado da verificação.

5.2 Heurística de Fiat-Shamir

Em 1986, os pesquisadores israelenses Amos Fiat e Adi Shamir criaram um método para se obter provas de conhecimento-zero não-interativas à partir de protocolos de identificação de conhecimento-zero canônicos. Isso permite a construção de um esquema de assinatura a partir de esquemas de identificação. O método ficou conhecido como transformação Fiat-Shamir.

Assina (s, msg)	**Verifica** (v, msg, σ)
{	{
$Com \leftarrow P(s)$;	$Com, Rsp \leftarrow \sigma$;
$Ch \leftarrow h(Com, msg)$;	$Ch \leftarrow h(Com, msg)$;
$Rsp \leftarrow P(s, Com, Ch)$;	$Bool \leftarrow V(v, Com, Ch, Rsp)$;
Devolver Com, Rsp;	**Devolver** $Bool$;
}	}

Figura 5.1: Pseudocódigo dos algoritmos do esquema de assinatura

A transformação consiste em substituir o desafio aleatório do verificador por um hash da concatenação do compromisso do provador com a mensagem a ser assinada. Usa-se os algoritmos *Inicia* e *Gera* do esquema de identificação a ser transformado sem modificação como os respectivos algoritmos *Inicia* e *Gera* do novo esquema de assinatura. O *Assina* computa à partir de uma chave secreta um compromisso usando P, concatena-o com a mensagem e alimenta-o a uma função de hash h, cujo resultado é então devolvido a P como desafio. A resposta de P é então concatenada com o com-

promisso e devolvida como assinatura σ. O algoritmo $Verifica$ utiliza a metade de σ correspondente ao compromisso, concatenando-a com a mensagem a ser verificada e alimentando-o a h. O resultado é alimentado a V juntamente com o próprio σ e, claro, uma chave pública. A resposta de V é usada então como a resposta de $Verifica$.

5.3 Protocolo de Identificação MQID-3A

Em 2012, o pesquisador brasileiro Fábio S. Monteiro apresentou um aprimoramento do algoritmo MQID-3 que reduz o trâmite de dados entre as partes provadora e verificadora.

A melhoria consiste em dividir adicionalmente parte do segredo em $r_1 = d_0 + d_1$ e de $F(r_1)$ em $F(r_1) = u_0 + u_1$. A equação 4.2 pode ser então aplicada a r_1:

$$G(r_0, d_1) + u_1 = v - F(r_0) - G(r_0, d_0) - u_0 \tag{5.1}$$

Dessa forma, basta o provador demonstrar que conhece uma tupla $(r_0, r_1, t_0, t_1, e_0, e_1, d_0, d_1, u_0, u_1)$ que satisfaz 4.2, 4.3, 5.1 e 5.2.

$$(d_0, u_0) = (r_1 - d_1, F(r_1) - u_1) \tag{5.2}$$

Como revelar esta tupla seria equivalente a revelar o segredo completo, o provador no protocolo MQID-3A revela, de acordo com um desafio $Ch \in \{0,1,2,3\}$ do verificador, ou a tupla $(r_1, t_1, e_1, d_0, u_0)$, ou a tupla $(r_1, t_0, e_0, d_1, u_1)$, ou a tupla $(r_0, t_0, e_0, d_1, u_1)$, ou a tupla $(r_0, t_1, e_1, d_0, u_0)$. O verificador pode então conferir cada membro das equações acima sem obter nenhuma informação a respeito da chave secreta s, conforme esquematizado na figura 5.2.

O protocolo MQID-3A é prova de conhecimento-zero estatístico como o MQID-3, porém com probabilidade de erro $1/2$ ao invés de $2/3$. Isso significa que menos iterações são necessárias para se atingir o nível de segurança desejado. Assim a comunicação

Provador, entrada: $((F,v),s)$	**Verificador**, entrada: (F,v)
Escolha $r_0,t_0,d_0 \in_R \mathbb{F}_q^n,\ e_0,u_0 \in_R \mathbb{F}_q^m$	
$r_1 \leftarrow s-r_0,\ t_1 \leftarrow r_0-t_0,\ d_1 \leftarrow r_1-d_0$	
$e_1 \leftarrow F(r_0)-e_0,\ u_1 \leftarrow F(r_1)-u_0$	
$c_0 \leftarrow Com(r_0, G(r_0,d_1)+u_1)$	
$c_1 \leftarrow Com(r_1, G(t_0,r_1)+e_0)$	
$c_2 \leftarrow Com(t_0,e_0),\ c_3 \leftarrow Com(t_1,e_1)$	
$c_4 \leftarrow Com(d_0,u_0),\ c_5 \leftarrow Com(d_1,u_1)$	
$\xrightarrow{(c_0,c_1,c_2,c_3,c_4,c_5)}$	
	Escolha $Ch \in_R \{0,1,2,3\}$
$\xleftarrow{Ch}$	
Se $Ch=0$, $Rsp \leftarrow (r_1,t_1,e_1,d_0,u_0)$	
Se $Ch=1$, $Rsp \leftarrow (r_1,t_0,e_0,d_1,u_1)$	
Se $Ch=2$, $Rsp \leftarrow (r_0,t_0,e_0,d_1,u_1)$	
Se $Ch=3$, $Rsp \leftarrow (r_0,t_1,e_1,d_0,u_0)$	
$\xrightarrow{Rsp}$	
	Se $Ch=0$, leia $Rsp=(r_1,t_1,e_1,d_0,u_0)$, verifique
	$c_1 \stackrel{?}{=} Com(r_1, v-F(r_1)-G(t_1,r_1)-e_1)$
	$c_3 \stackrel{?}{=} Com(t_1,e_1)$
	$c_4 \stackrel{?}{=} Com(d_0,u_0)$
	$c_5 \stackrel{?}{=} Com(r_1-d_0, F(r_1)-u_0)$
	Se $Ch=1$, leia $Rsp=(r_1,t_0,e_0,d_1,u_1)$, verifique
	$c_1 \stackrel{?}{=} Com(r_1, G(t_0,r_0)+e_0)$
	$c_2 \stackrel{?}{=} Com(t_0,e_0)$
	$c_4 \stackrel{?}{=} Com(r_1-d_1, F(r_1)-u_1)$
	$c_5 \stackrel{?}{=} Com(d_1,u_1)$
	Se $Ch=2$, leia $Rsp=(r_0,t_0,e_0,d_1,u_1)$, verifique
	$c_0 \stackrel{?}{=} Com(r_0, G(r_0,d_1)+u_1)$
	$c_2 \stackrel{?}{=} Com(t_0,e_0)$
	$c_3 \stackrel{?}{=} Com(r_0-t_0, F(r_0)-e_0)$
	$c_5 \stackrel{?}{=} Com(d_1,u_1)$
	Se $Ch=3$, leia $Rsp=(r_0,t_1,e_1,d_0,u_0)$, verifique
	$c_0 \stackrel{?}{=} Com(r_0, v-F(r_0)-G(r_0,d_0)-u_0)$
	$c_2 \stackrel{?}{=} Com(r_0-t_1, P(r_0)-e_1)$
	$c_3 \stackrel{?}{=} Com(t_1,e_1)$
	$c_4 \stackrel{?}{=} Com(d_0,u_0)$

Figura 5.2: Descrição do protocolo MQID-3A

necessária entre provador e verificador é menor, resultando em economia de energia em dispositivos embarcados onde a transmissão de dados é dispendiosa. Além disso, o desafio do verificador nesta versão do protocolo cabe em 2 bits sem sobras, facilitando a implementação de uma função de hash para transformá-lo por Fiat-Shamir.

APÊNDICE A – Implementação de exemplo

Disponibilizamos abaixo o código fonte em C de uma implementação ingênua dos protocolos MQID-3 e MQID-5 criada para estudo. O programa abaixo não deve ser utilizado em qualquer aplicação prática, pois o gerador de números pseudo-aleatórios empregado não é seguro. Foi utilizada a função de hash Whirlpool do professor Paulo S. L. M. Barreto da Escola Politécnica da Universidade de São Paulo, cujo código fonte não está incluso. O código também pode ser obtido com maior facilidade em `http://www.linux.ime.usp.br/~cfilipe/tcc/mqid.tar.gz`.

A.1 field.h

```
1  /* generic interface for finite-field operations */
2
3  typedef int Field;
4
5  /* returns the order in use */
6  int getOrder();
7
8  /* adds two field elements (modulo order addition) */
9  Field add(Field a, Field b);
10
11 /* multiplies two field elements (modulo order multiplication) */
12 Field mult(Field a, Field b);
13
```

```
14 /* returns the zero element of the field (s.t. x+0 = x, x*0 = 0)
       */
15 Field zero();
16
17 /* returns -a (s.t. a + (-a) = 0) */
18 Field neg(Field a);
19
20 /* whether a is zero or not */
21 int isZero(Field a);
22
23 /* adds two n-dimentional vectors on the field */
24 Field * vadd(Field *a, Field *b, int n);
25
26 /* subtracts two n-dimentional vectors on the field */
27 Field * vsub(Field *a, Field *b, int n);
28
29 /* performs scalar multiplication on a vector */
30 void vmult(Field *a, Field x, int n);
31
32 /* returns a string on the heap containing a hex representation of
       a vector */
33 char * tohexstr(Field *a, int n);
```

MQID/field.h

A.2 field2.c

```
1 /* order 2 implementation of field.h interface */
2
3 #include <stdlib.h>
4 #include "field.h"
5
6 static int order = 2;
7
8 int getOrder()
```

```
9  {
10     return order;
11 }
12
13 Field add(Field a, Field b)
14 {
15     return a ^ b;
16 }
17
18 Field mult(Field a, Field b)
19 {
20     return a & b;
21 }
22
23 Field zero()
24 {
25     return 0;
26 }
27
28 Field neg(Field a)
29 {
30     return a;
31 }
32
33 int isZero(Field a)
34 {
35     return !a;
36 }
37
38 Field * vadd(Field *a, Field *b, int n)
39 {
40     int i;
41
42     for (i = 0; i < n; i++)
43         a[i] = add(a[i], b[i]);
44
```

```
45      return a;
46  }
47
48  Field * vsub(Field *a, Field *b, int n)
49  {
50      return vadd(a, b, n); /* addition and subtraction are
            equivalent in GF(2) */
51  }
52
53  void vmult(Field *a, Field x, int n)
54  {
55      int i;
56
57      for(i = 0; i < n; i++)
58          a[i] = mult(a[i], x);
59  }
60
61  char * tohexstr(Field *a, int n)
62  {
63      int i, j, m, p;
64      char s;
65      char *str;
66
67      p = m = (n + 3) / 4;
68
69      if(m % 2 == 0)
70          str = malloc((m + 1) * sizeof(char));
71      else { /* this is because we want an even number of
            hexadecimal digits */
72          m++;
73          str = malloc((m + 1) * sizeof(char));
74          str[0] = '0';
75      }
76      str[m] = '\0';
77
78      for (i = 0; i < p; ++i) {
```

```
79          int q = 4*i, t;
80
81          s = 0;
82          t = 1;
83          for (j = 0; j < 4 && (q + j) < n; ++j) {
84              s += a[q + j]*t;
85              t *= 2;
86          }
87
88          if (s < 10)
89              str[m-(i+1)] = '0' +  s;
90          else
91              str[m-(i+1)] = 'A' +  s - 10;
92      }
93
94      return str;
95  }
```

MQID/field2.c

A.3 field2_4.c

```
1  /* order 16 implementation of field.h interface */
2  /* GF(2^4) with decimal 19 as primitive */
3
4  #include <stdlib.h>
5  #include "field.h"
6
7  static int order = 16;
8
9  int getOrder()
10 {
11     return order;
12 }
13
```

```
14 Field add(Field a, Field b)
15 {
16     return (a ^ b) % order;
17 }
18
19 Field mult(Field a, Field b)
20 {
21     int p = 0;
22     int i;
23     int carry;
24     for (i = 0; i < 4; i++) {
25         if (b & 1)
26             p ^= a;
27         carry = (a & 8);
28         a <<= 1;
29         if (carry)
30             a ^= 19; /* x^4 + x + 1 */
31         b >>= 1;
32     }
33     return p % order;
34 }
35
36 Field zero()
37 {
38     return 0;
39 }
40
41 Field neg(Field a)
42 {
43     return a;
44 }
45
46 int isZero(Field a)
47 {
48     return (a == 0);
49 }
```

```
50
51 Field * vadd(Field *a, Field *b, int n)
52 {
53     int i;
54
55     for (i = 0; i < n; i++)
56         a[i] = add(a[i], b[i]);
57
58     return a;
59 }
60
61 Field * vsub(Field *a, Field *b, int n)
62 {
63     int i;
64
65     for (i = 0; i < n; i++)
66         a[i] = add(a[i], neg(b[i]));
67
68     return a;
69 }
70
71 void vmult(Field *a, Field x, int n)
72 {
73     int i;
74
75     for(i = 0; i < n; i++)
76         a[i] = mult(a[i], x);
77 }
78
79 char * tohexstr(Field *a, int n)
80 {
81     int i, m;
82     char *str;
83
84     if(n % 2 == 0) {
85         m = n + 1;
```

```
86          str = malloc(m * sizeof(char));
87          str[n] = '\0';
88
89      } else {  /* this is because we want an even number of
            hexadecimal digits */
90          m = n + 2;
91          str = malloc(m * sizeof(char));
92          str[0] = '0';
93          str[n+1] = '\0';
94      }
95
96      for (i = 0; i < n; ++i) {
97          if (a[i] < 10)
98              str[m-(i+2)] = '0' +  (char)(a[i]);
99          else
100             str[m-(i+2)] = 'A' +  (char)(a[i]) - 10;
101     }
102
103     return str;
104 }
```

MQID/field2_4.c

A.4 mqid.h

```
1 /* MQID-3 and MQID-5 generic interfaces */
2
3 #include "field.h"
4 #include "Whirlpool/Whirlpool.h"
5
6 typedef struct mqid *MQIDScheme; /* stores a reference for the
      scheme and its data */
7
8 typedef Field *MQID_key; /* a key is a vector of field elements */
9
```

```
10
11 /* returns the size of all commitment strings */
12 int getComStrSize();
13
14 /* initializes a mxn sized scheme with random parameters */
15 MQIDScheme setupMQIDScheme (int m, int n);
16
17 /* returns the number of equations of a scheme */
18 int MQID_getm(MQIDScheme s);
19
20 /* returns the number of unknowns of a scheme */
21 int MQID_getn(MQIDScheme s);
22
23 /* returns the quadratic coefficients */
24 Field MQID_geta(MQIDScheme s, int k, int i, int j);
25
26 /* returns the linear coefficients */
27 Field MQID_getb(MQIDScheme s, int k, int i);
28
29 /* F(x) */
30 MQID_key MQID_F(MQIDScheme s, MQID_key x);
31
32 /* G(x,y) */
33 MQID_key MQID_G(MQIDScheme s, MQID_key x, MQID_key y);
34
35 /* key-generation algorithm */
36 void MQID_Gen(MQIDScheme s, MQID_key *pk, MQID_key *sk);
37
38 /* sorts a uniformly distributed random m-dimentional Field array
      */
39 MQID_key MQID_unidrndkey(int m);
40
41 // takes as parameters n keys and its respective sizes, computes
      the commitment scheme function on them
42 // puts it on comstr and returns it
```

```
43 // sample usage: comstr = MQID_com(comstr, 3, key1, n1, key2, n2,
       key3, n3)
44 char * MQID_Com(char * comstr, int n, ...);
45
46 // clears memory
47 void clearMQIDScheme(MQIDScheme s);
```

MQID/mqid.h

A.5 mqid.c

```
1  /* sample naive implementation of Sakumoto, Shirai and Hiwatari's
2   * public-key id. scheme based on the MQ problem
3   *
4   * WARNING: this is a naive implementation made for academic
        purposes and
5   * shouldn't be put to security sensitive use
6   *
7   * NOTE: this implementation does not provide a mechanism to store
        scheme system
8   * parameter data
9   *
10  * by: Carlos Filipe Lombizani
11  *     cfilipe@linux.ime.usp.br
12  */
13
14 #include <stdlib.h>
15 #include <stdarg.h>
16 #include <string.h>
17 #include "mqid.h"
18
19 struct mqid {
20     /* system parameters */
21     int m; /* equations */
22     int n; /* unknowns */
```

```
23
24      /* coefficient matrixes */
25      Field ***a; /* quadratic */
26      Field **b; /* linear */
27  };
28
29  int getComStrSize(){
30      return HEXHASHSIZE;
31  }
32
33  /* initialization */
34  MQIDScheme setupMQIDScheme (int m, int n)
35  {
36      int i, j, l, row;
37      int sizeA, sizeB;
38      MQIDScheme s;
39
40      sizeA = m * (n * n + n)/2;
41      sizeB = n * m;
42
43      /* alocacao de espaco e configuracao de ponteiros */
44      s = malloc(sizeof(struct mqid));
45      s->n = n;
46      s->m = m;
47
48      s->a = malloc(m * sizeof(Field **));
49      s->a[0] = malloc(sizeB * sizeof(Field *));
50
51      for (i = 1; i < m; i++) {
52          s->a[i] = s->a[i-1] + n;
53      }
54
55      s->a[0][0] = malloc(sizeA * sizeof(Field));
56
57      j = 1;
58      row = 0;
```

```
59      for (i = 0; i < m; i++) {
60          while (j < n) {
61              int set = i;
62              if (row == n - 1)
63                  set--;
64              s->a[i][j] = s->a[set][row] + (row + 1);
65              if (row == n - 1)
66                  set++;
67              j++; row++;
68              row = row % n;
69          }
70          j = 0;
71      }
72
73      s->b = malloc(m * sizeof(Field *));
74      s->b[0] = malloc(sizeB * sizeof(Field));
75
76      for (i = 1; i < m; i++) {
77          s->b[i] = s->b[i-1] + n;
78      }
79
80      for (l = 0; l < m; l++)
81          for (i = 0; i < n; i++) {
82              for (j = 0; j <= i; j++) {
83                  s->a[l][i][j] = rand() % getOrder();
84              }
85              s->b[l][i] = rand() % getOrder();
86          }
87
88      return s;
89  }
90
91  int MQID_getm(MQIDScheme s)
92  {
93      return s->m;
94  }
```

```
95
96  int MQID_getn(MQIDScheme s)
97  {
98      return s->n;
99  }
100
101 Field MQID_geta(MQIDScheme s, int l, int i, int j)
102 {
103     /* boundcheck */
104     if (l >= s->m || i >= s-> n || j > i)
105         return -1;
106
107     return s->a[l][i][j];
108 }
109
110 Field MQID_getb(MQIDScheme s, int l, int i)
111 {
112     /* boundcheck */
113     if (l >= s->m || i >= s->n)
114         return -1;
115
116     return s->b[l][i];
117 }
118
119 /* calcula y = F(x) */
120 MQID_key MQID_F(MQIDScheme s, MQID_key x)
121 {
122     int l, i, j;
123     MQID_key y;
124
125     y = malloc(s->m * sizeof(Field));
126
127     for (l = 0; l < s->m; l++) {
128         y[l] = zero();
129         for (i = 0; i < s->n; i++) {
130             y[l] = add(y[l], mult(s->b[l][i], x[i]));
```

```
131             for (j = 0; j <= i; j++)
132                 y[l] = add(y[l], mult(s->a[l][i][j], mult(x[i], x[
                        j])));
133         }
134     }
135
136     return y;
137 }
138
139 /* computes the polar form F(x+y)-F(x)-F(y) = G(x,y) */
140 MQID_key MQID_G(MQIDScheme s, MQID_key x, MQID_key y)
141 {
142     int l, i, j;
143     MQID_key g;
144
145     g = malloc(s->m * sizeof(Field));
146
147     for (l = 0; l < s->m; l++) {
148         g[l] = zero();
149         for (i = 0; i < s->n; i++)
150             for (j = 0; j <= i; j++)
151                 g[l] = add(g[l], mult(s->a[l][i][j], add(mult(x[j
                        ], y[i]), mult(x[i], y[j]))));
152     }
153
154     return g;
155 }
156
157 void MQID_Gen(MQIDScheme s, MQID_key *pk, MQID_key *sk)
158 {
159     *sk = MQID_unidrndkey(s->n);
160     *pk = MQID_F(s, *sk);
161 }
162
163 MQID_key MQID_unidrndkey(int m)
164 {
```

```
165     MQID_key key;
166     int l;
167
168     key = malloc(m * sizeof(Field));
169
170     for (l = 0; l < m; l++)
171         key[l] = rand() % getOrder();
172
173     return key;
174 }
175
176 /* String commitment scheme */
177 char * MQID_Com(char * comstr, int n, ...)
178 {
179     va_list ap;
180     int i, m;
181     MQID_key k;
182     char * auxstr, * tmpstr, * hex;
183
184     auxstr = malloc(sizeof(char));
185     auxstr[0] = '\0';
186
187     va_start(ap, n);
188     for (i = 0; i < n; i++) {
189         k = va_arg(ap, MQID_key);
190         m = va_arg(ap, int);
191
192         hex = tohexstr(k, m);
193         tmpstr = malloc((strlen(auxstr) + strlen(hex) + 1) *
                sizeof(char));
194         tmpstr[0] = '\0';
195         tmpstr = strcat(tmpstr, auxstr);
196         tmpstr = strcat(tmpstr, hex);
197
198         free(auxstr);
199         auxstr = tmpstr;
```

```
200     }
201
202     comstr = whirlpoolsum(comstr, tmpstr);
203
204     free(tmpstr);
205
206     return comstr;
207 }
208
209 void clearMQIDScheme(MQIDScheme s)
210 {
211     free(s->b[0]);
212     free(s->b);
213     free(s->a[0][0]);
214     free(s->a[0]);
215     free(s->a);
216     free(s);
217 }
```

MQID/mqid.c

A.6 mqid3.h

```
1  /* MQID-3 specific interface */
2
3  #include "mqid.h"
4
5  /* store prover and verifier data */
6  typedef struct mqid3_prover *MQID3_prover;
7  typedef struct mqid3_verifier *MQID3_verifier;
8
9  /* step-by-step protocol algorithm implementation */
10
11 MQID3_prover prover1stStep(MQIDScheme s, MQID_key sk, char * c0,
       char * c1, char * c2);
```

```
12
13 MQID_key * prover2ndStep(MQID3_prover p, char ch);
14
15 MQID3_verifier verifier1stStep(MQIDScheme s, MQID_key pk, char *
       c0, char * c1, char * c2, char * ch);
16
17 int verifier2ndStep(MQID3_verifier v, MQID_key * rsp);
18
19 /* clears everything up */
20 void clearMQID3(MQID3_prover p, MQID3_verifier v);
```

MQID/mqid3.h

A.7 mqid3.c

```
1 /* sample naive implementation of Sakumoto, Shirai and Hiwatari's
2  * 3-pass public-key id. scheme based on the MQ problem
3  * by: Carlos Filipe Lombizani
4  *     cfilipe@linux.ime.usp.br
5  */
6
7 #include <stdlib.h>
8 #include <string.h>
9 #include "mqid3.h"
10
11 struct mqid3_prover {
12     MQID_key r0, t0, e0, r1, t1, e1;
13 };
14
15 struct mqid3_verifier {
16     char ch;
17     char * com1;
18     char * com2;
19     MQIDScheme s;
20     MQID_key pk;
```

```
21 };
22
23 MQID3_prover prover1stStep(MQIDScheme s, MQID_key sk, char * c0,
       char * c1, char * c2)
24 {
25     MQID3_prover p;
26     size_t unknowns;
27     MQID_key fr0, gt0r1_e0;
28
29     p = malloc(sizeof(struct mqid3_prover));
30
31     p->r0 = MQID_unidrndkey(MQID_getn(s));
32     p->t0 = MQID_unidrndkey(MQID_getn(s));
33     p->e0 = MQID_unidrndkey(MQID_getm(s));
34
35     unknowns = MQID_getn(s) * sizeof(Field);
36     fr0 = MQID_F(s, p->r0);
37
38     p->r1 = malloc(MQID_getn(s) * sizeof(Field));
39     p->t1 = malloc(MQID_getn(s) * sizeof(Field));
40     p->e1 = malloc(MQID_getm(s) * sizeof(Field));
41
42     p->r1 = memcpy(p->r1, sk, unknowns);
43     p->t1 = memcpy(p->t1, p->r0, unknowns);
44     p->e1 = memcpy(p->e1, fr0, MQID_getm(s) * sizeof(Field));
45
46     p->r1 = vsub(p->r1, p->r0, MQID_getn(s));
47     p->t1 = vsub(p->t1, p->t0, MQID_getn(s));
48     p->e1 = vsub(p->e1, p->e0, MQID_getm(s));
49
50     gt0r1_e0 = MQID_G(s, p->t0, p->r1);
51     gt0r1_e0 = vadd(gt0r1_e0, p->e0, MQID_getm(s));
52
53     c0 = MQID_Com(c0, 2, p->r1, MQID_getn(s), gt0r1_e0, MQID_getm(
           s));
```

```
54      c1 = MQID_Com(c1, 2, p->t0, MQID_getn(s), p->e0, MQID_getm(s))
            ;
55      c2 = MQID_Com(c2, 2, p->t1, MQID_getn(s), p->e1, MQID_getm(s))
            ;
56
57      free(fr0);
58      free(gt0r1_e0);
59
60      return p;
61  }
62
63  MQID_key * prover2ndStep(MQID3_prover p, char ch)
64  {
65      MQID_key * rsp;
66
67      rsp = malloc(3 * sizeof(MQID_key));
68
69      switch (ch) {
70          case 0:
71              rsp[0] = p->r0;
72              rsp[1] = p->t1;
73              rsp[2] = p->e1;
74              free(p->t0);
75              p->t0 = NULL;
76              free(p->e0);
77              p->e0 = NULL;
78              free(p->r1);
79              p->r1 = NULL;
80              break;
81          case 1:
82              rsp[0] = p->r1;
83              rsp[1] = p->t1;
84              rsp[2] = p->e1;
85              free(p->r0);
86              p->r0 = NULL;
87              free(p->t0);
```

```
88              p->t0 = NULL;
89              free(p->e0);
90              p->e0 = NULL;
91              break;
92          case 2:
93              rsp[0] = p->r1;
94              rsp[1] = p->t0;
95              rsp[2] = p->e0;
96              free(p->r0);
97              p->r0 = NULL;
98              free(p->t1);
99              p->t1 = NULL;
100             free(p->e1);
101             p->e1 = NULL;
102             break;
103         default:
104             free(rsp);
105             rsp = NULL;
106     }
107
108     return rsp;
109 }
110
111 MQID3_verifier verifier1stStep(MQIDScheme s, MQID_key pk, char *
        c0, char * c1, char * c2, char * ch)
112 {
113     MQID3_verifier v;
114
115     v = malloc(sizeof(struct mqid3_verifier));
116
117     v->ch = (*ch) = (char)(rand()%3);
118
119     v->s = s;
120     v->pk = pk;
121
122     switch(v->ch) {
```

```
123         case 0:
124             v->com1 = c1;
125             v->com2 = c2;
126             break;
127         case 1:
128             v->com1 = c0;
129             v->com2 = c2;
130             break;
131         case 2:
132             v->com1 = c0;
133             v->com2 = c1;
134             break;
135     }
136 
137     return v;
138 }
139 
140 int verifier2ndStep(MQID3_verifier v, MQID_key * rsp)
141 {
142     int verified;
143     char * c1;
144     char * c2;
145     MQID_key f, g, second;
146 
147     f = NULL;
148     g = NULL;
149 
150     switch(v->ch) {
151         case 0:
152             f = MQID_F(v->s, rsp[0]);
153             vsub(f, rsp[2], MQID_getm(v->s));
154             vsub(rsp[0], rsp[1], MQID_getn(v->s));
155             second = f;
156             break;
157         case 1:
158             f = MQID_F(v->s, rsp[0]);
```

```
159             g = MQID_G(v->s, rsp[1], rsp[0]);
160             vadd(g, rsp[2], MQID_getm(v->s));
161             vadd(f, g, MQID_getm(v->s));
162             vsub(v->pk, f, MQID_getm(v->s));
163             second = v->pk;
164             break;
165         case 2:
166             g = MQID_G(v->s, rsp[1], rsp[0]);
167             vadd(g, rsp[2], MQID_getm(v->s));
168             second = g;
169             break;
170     }
171
172     c1 = malloc(sizeof(unsigned char) * (getComStrSize() + 1));
173     c2 = malloc(sizeof(unsigned char) * (getComStrSize() + 1));
174     c1 = MQID_Com(c1, 2, rsp[0], MQID_getn(v->s), second,
            MQID_getm(v->s));
175     c2 = MQID_Com(c2, 2, rsp[1], MQID_getn(v->s), rsp[2],
            MQID_getm(v->s));
176
177     verified = (strcmp(c1, v->com1) == 0) && (strcmp(c2, v->com2)
            == 0);
178
179     free(c1);
180     free(c2);
181
182     free(f);
183     free(g);
184
185     return verified;
186 }
187
188 void clearMQID3(MQID3_prover p, MQID3_verifier v)
189 {
190     free(v);
191     free(p->r0);
```

```
192     free(p->t0);
193     free(p->e0);
194     free(p->r1);
195     free(p->t1);
196     free(p->e1);
197     free(p);
198 }
```

MQID/mqid3.c

A.8 mqid5.h

```
1  /* MQID-3 specific interface */
2
3  #include "mqid.h"
4
5  /* store prover and verifier data */
6  typedef struct mqid5_prover *MQID5_prover;
7  typedef struct mqid5_verifier *MQID5_verifier;
8
9  /* step-by-step protocol algorithm implementation */
10
11 MQID5_prover prover1stStep(MQIDScheme s, MQID_key sk, char * c0,
       char * c1);
12
13 void prover2ndStep(MQID5_prover p, Field alpha, MQID_key * t1,
       MQID_key * e1);
14
15 MQID_key prover3rdStep(MQID5_prover p, char ch);
16
17 MQID5_verifier verifier1stStep(MQIDScheme s, MQID_key pk, char *
       c0, char * c1, Field * alpha);
18
19 char verifier2ndStep(MQID5_verifier v, MQID_key t1, MQID_key e1);
20
```

```
21 int verifier3rdStep(MQID5_verifier v, MQID_key rsp);
22
23 /* clears everything up */
24 void clearMQID5(MQID5_prover p, MQID5_verifier v);
```

MQID/mqid5.h

A.9 mqid5.c

```
1  /* sample naive implementation of Sakumoto, Shirai and Hiwatari's
2   * 5-pass public-key id. scheme based on the MQ problem
3   * by: Carlos Filipe Lombizani
4   *     cfilipe@linux.ime.usp.br
5   */
6
7  #include <stdlib.h>
8  #include <string.h>
9  #include "mqid5.h"
10
11 struct mqid5_prover {
12     MQIDScheme s;
13     MQID_key r0, t0, e0, r1, t1, e1;
14 };
15
16 struct mqid5_verifier {
17     Field alpha;
18     char ch;
19     char * com0;
20     char * com1;
21     MQIDScheme s;
22     MQID_key pk, t1, e1;
23 };
24
25 MQID5_prover prover1stStep(MQIDScheme s, MQID_key sk, char * c0,
       char * c1)
```

```
26 {
27     MQID5_prover p;
28     MQID_key gt0r1_e0;
29 
30     p = malloc(sizeof(struct mqid5_prover));
31 
32     p->s = s;
33 
34     p->r0 = MQID_unidrndkey(MQID_getn(s));
35     p->t0 = MQID_unidrndkey(MQID_getn(s));
36     p->e0 = MQID_unidrndkey(MQID_getm(s));
37 
38     p->r1 = malloc(MQID_getn(s) * sizeof(Field));
39     p->r1 = memcpy(p->r1, sk, MQID_getn(s) * sizeof(Field));
40     p->r1 = vsub(p->r1, p->r0, MQID_getn(s));
41 
42     gt0r1_e0 = MQID_G(s, p->t0, p->r1);
43     gt0r1_e0 = vadd(gt0r1_e0, p->e0, MQID_getm(s));
44 
45     c0 = MQID_Com(c0, 3, p->r0, MQID_getn(s), p->t0, MQID_getn(s),
          p->e0, MQID_getm(s));
46     c1 = MQID_Com(c1, 2, p->r1, MQID_getn(s), gt0r1_e0, MQID_getm(
         s));
47 
48     return p;
49 }
50 
51 void prover2ndStep(MQID5_prover p, Field alpha, MQID_key * t1,
     MQID_key * e1)
52 {
53     MQID_key f;
54 
55     p->t1 = malloc(MQID_getn(p->s) * sizeof(Field));
56     p->t1 = memcpy(p->t1, p->r0, MQID_getn(p->s) * sizeof(Field));
57     vmult(p->t1, alpha, MQID_getn(p->s));
58     p->t1 = vsub(p->t1, p->t0, MQID_getn(p->s));
```

```
59
60      f = MQID_F(p->s, p->r0);
61      p->e1 = malloc(MQID_getm(p->s) * sizeof(Field));
62      p->e1 = memcpy(p->e1, f, MQID_getm(p->s) * sizeof(Field));
63      vmult(p->e1, alpha, MQID_getm(p->s));
64      p->e1 = vsub(p->e1, p->e0, MQID_getm(p->s));
65
66      *t1 = p->t1;
67      *e1 = p->e1;
68
69      free(f);
70  }
71
72  MQID_key prover3rdStep(MQID5_prover p, char ch)
73  {
74      MQID_key rsp;
75
76      switch (ch) {
77          case 0:
78              rsp = p->r0;
79              p->r0 = NULL;
80              break;
81          case 1:
82              rsp = p->r1;
83              p->r1 = NULL;
84              break;
85          default:
86              rsp = NULL;
87      }
88
89      return rsp;
90  }
91
92  MQID5_verifier verifier1stStep(MQIDScheme s, MQID_key pk, char *
        c0, char * c1, Field * alpha)
93  {
```

```
94      MQID5_verifier v;
95
96      v = malloc(sizeof(struct mqid5_verifier));
97
98      *alpha = v->alpha = rand() % getOrder();
99
100     v->s = s;
101     v->pk = pk;
102
103     v->com0 = c0;
104     v->com1 = c1;
105
106     return v;
107 }
108
109 char verifier2ndStep(MQID5_verifier v, MQID_key t1, MQID_key e1)
110 {
111     v->t1 = t1;
112     v->e1 = e1;
113     v->ch = rand() % 2;
114     return v->ch;
115 }
116
117 int verifier3rdStep(MQID5_verifier v, MQID_key rsp)
118 {
119     int verified;
120     char * c0;
121     MQID_key f, g, second;
122
123     f = NULL;
124     g = NULL;
125     second = NULL;
126
127     switch(v->ch) {
128         case 0:
129             second = malloc(MQID_getn(v->s) * sizeof(Field));
```

```
130             second = memcpy(second, rsp, MQID_getn(v->s) * sizeof(
                    Field));
131             vmult(second, v->alpha, MQID_getn(v->s));
132             second = vsub(second, v->t1, MQID_getn(v->s));
133 
134             f = MQID_F(v->s, rsp);
135             vmult(f, v->alpha, MQID_getm(v->s));
136             f = vsub(f, v->e1, MQID_getm(v->s));
137 
138             c0 = malloc(sizeof(char) * (getComStrSize() + 1));
139             c0 = MQID_Com(c0, 3, rsp, MQID_getn(v->s), second,
                    MQID_getn(v->s), f, MQID_getm(v->s));
140 
141             verified = (strcmp(c0, v->com0) == 0);
142             break;
143         case 1:
144             f = MQID_F(v->s, rsp);
145             g = MQID_G(v->s, v->t1, rsp);
146             vadd(g, v->e1, MQID_getm(v->s));
147             vsub(v->pk, f, MQID_getm(v->s));
148             vmult(v->pk, v->alpha, MQID_getm(v->s));
149             vsub(v->pk, g, MQID_getm(v->s));
150 
151             c0 = malloc(sizeof(char) * (getComStrSize() + 1));
152             c0 = MQID_Com(c0, 2, rsp, MQID_getn(v->s), v->pk,
                    MQID_getm(v->s));
153 
154             verified = (strcmp(c0, v->com1) == 0);
155             break;
156     }
157 
158     free(f);
159     free(g);
160     free(second);
161     free(c0);
162 
```

```
163     return verified;
164 }
165
166 void clearMQID5(MQID5_prover p, MQID5_verifier v)
167 {
168     free(v);
169     free(p->r0);
170     free(p->t0);
171     free(p->e0);
172     free(p->r1);
173     free(p->t1);
174     free(p->e1);
175     free(p);
176 }
```

MQID/mqid5.c

A.10 Whirlpool/Whirlpool.h

```
1 #define HEXHASHSIZE 128
2
3 /* computes the hex Whirlpool hash checksum of str and copies it
      to *sum, returns sum */
4 char * whirlpoolsum(char * sum, char * const str);
```

MQID/Whirlpool/Whirlpool.h

Referências Bibliográficas

[Anton e Rorres (2005)] **Anton e Rorres (2005)** Howard Anton e Chris Rorres. *Elementary Linear Algebra, Applications Version*. John Wiley and Sons, Inc., ninth edition ed. ISBN 978-04-716-6959-8.

[Bouillaguet *et al.* (2010)Bouillaguet, Chen, Cheng, Chou, Niederhagen, Shamir, e Yang] **Bouillaguet** *et al.* **(2010)** Charles Bouillaguet, Hsieh-Chung Chen, Chen-Mou Cheng, Tung Chou, Ruben Niederhagen, Adi Shamir e Bo-Yin Yang. Fast Exhaustive Search for Polynomial Systems in $\mathbb{F}_2$. *Cryptographic Hardware and Embedded Systems*, 6225:203–218.

[Fiat e Shamir (1986)] **Fiat e Shamir (1986)** Amos Fiat e Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. páginas 186–194. Springer-Verlag London. ISBN 0-387-18047-8.

[Goldreich *et al.* (1991)Goldreich, Micali, e Widgerson] **Goldreich** *et al.* **(1991)** Oded Goldreich, Silvio Micali e Avi Widgerson. Proofs that yield nothing but their validity, or all languages in NP have zero-knowledge proof systems. *Journal of the ACM*, 38:690–728.

[Halevi e Micali (1996)] **Halevi e Micali (1996)** Shai Halevi e Silvio Micali. Practical and Provably-Secure Commitment Schemes from Collision-Free Hashing. páginas 201–215. Springer-Verlag London. ISBN 3-540-61512-1.

[Lima (2009)] **Lima (2009)** Elon Lages Lima. *Curso de Análise*, volume 1 of *Projeto Euclides*. IMPA - Instituto Nacional de Matemática Pura e Aplicada, 4ª imp. da 12ª ed. ISBN 978-85-244-0118-3.

[Matsumoto e Imai (1988)] **Matsumoto e Imai (1988)** Tsutomu Matsumoto e Hideki Imai. Public quadratic polynomial-tuples for efficient signature-verification and message-encryption. páginas 419–453. Springer-Verlag New York, Inc. ISBN 0-387-50251-3.

[Menezes *et al.* (1996)Menezes, Vanstone, e Oorschot] **Menezes** *et al.* **(1996)** Alfred J. Menezes, Scott A. Vanstone e Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc. ISBN 0-849-38523-7.

[Monteiro (2012)] **Monteiro (2012)** Fábio S. Monteiro. Protocolo de Identificação baseado em Polinômios Multivariáveis Quadráticos. Dissertação de Mestrado, Universidade de São Paulo.

[Patarin e Goubin (1997)] **Patarin e Goubin (1997)** Jacques Patarin e Louis Goubin. Trapdoor One-Way Permutations and Multivariate Polynomials. *Information and Communications Security*, 1334:356–368.

[Sakumoto *et al.* (2011)Sakumoto, Shirai, e Hiwatari] **Sakumoto** *et al.* **(2011)** Koichi Sakumoto, Taizo Shirai e Harunaga Hiwatari. Public-Key Identification Schemes Based on Multivariate Quadratic Polynomials. *Advances in Cryptology - CRYPTO'2011*, 6841:706–723.

[Shoup (2008)] **Shoup (2008)** Victor Shoup. *A Computational Introduction to Number Theory and Algebra*. Cambridge University Press. ISBN 978-05-215-1644-0.

[Terada (2011)] **Terada (2011)** Routo Terada. *Segurança de dados: Criptografia em redes de computador*. Editora Blucher, 2ª ed. ISBN 978-85-212-0439-8.