Universidade do Minho
Departamento de Informática

Criptografia e Segurança dos Sistemas de Informação

Projecto Integrador II
2012/2013

Milestone 1

Post quantum authentication

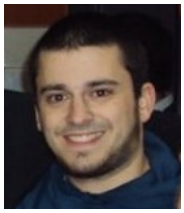
pg22797 Milton Nunes
pg22815 Nuno Carvalho

Supervisor:
Prof. Manuel Bernardo Barbosa

Abril de 2013

Milton Nunes

Nuno Carvalho



Resumo

Com o eventual aparecimento dos computadores quânticos, os esquemas de assinaturas baseados no problema da factorização ou do logaritmo discreto tornar-se-ão inúteis. Posto isto, tornou-se imperativo procurar esquemas de assinaturas digitais que se apresentassem como alternativas viáveis. Esta procura levou a que se chegassem a esquemas baseados em reticulados, que se revelaram uma alternativa válida ao problema que se pretendia resolver e que proporcionaram avanços significativos em várias áreas da criptografia.

Este Projecto Integrador pretende abordar e explorar a aplicabilidade de esquemas criptográficos de autenticação de mensagens baseados em reticulados. Neste relatório será apresentado todo o trabalho de análise e estudo efectuado sobre as duas publicações sugeridas [4, 5].

Do estudo efectuado para esta primeira *milestone*, resultou um pequeno protótipo em `Sage` do protocolo Lapin descrito em [4]. Além disso, são também apresentados os objectivos que se esperam alcançar no desenvolvimento deste projecto.

Conteúdo

Capítulo 1

Introdução

1.1 Objectivos

Nesta primeira fase pretende-se estudar o protocolo de autenticação Lapin. Para isso, vamos implementar um pequeno protótipo em `Sage`, visto que já utilizámos `Sage` para o primeiro Projecto Integrador. Depois de implementado o protótipo, é necessário escolher uma linguagem para a implementação do protocolo.
As opções consideradas para a implementação do protocolo eram C, C++ e Java. Estamos mais confortáveis em Java, mas visto que o Lapin é especialmente orientado para dispositivos low-cost, decidimos usar C.
Para esta primeira fase também efectuámos um estudo das operações aritmétricas mais eficientes, usando apenas operações binárias. Estas operações até poderão ser inicialmente criadas em `Sage`, com o objectivo de conhecer e perceber melhor os algoritmos das mesmas.
No resto do presente documento, apresenta-se todo o estudo e trabalho desenvolvido para esta primeira fase.

Capítulo 2

Lapin – Estudo

2.1 Descrição

O protocolo Lapin apresentado em [4] consiste num protocolo de autenticação baseado no problema Ring-LPN (*Ring variant of the Learning Parity with Noise*). Este protocolo, constituído por apenas dois *rounds* e seguro contra ataques activos, tem uma complexidade de comunicação bastante pequena, pelo que é indicado para dispositivos *low-cost* ou em cenários onde os recursos são limitados.
Em comparação com outros protocolos, que fazem uso de cifras por blocos como por exemplo o AES, o Lapin mostra-se como uma boa alternativa. Em casos onde se possuam algumas centenas de bytes de memória não-volátil, onde se poderão guardar alguns resultados pré-computados, o protocolo apenas é apenas duas vezes mais lento que o AES, mas em compensação, tem cerca de dez vezes menos código do que o AES.

2.2 Conceitos

2.2.1 Polinómios

O protocolo baseia-se essencialmente em operações sobre polinómios binários, ou seja, polinómios em $\mathbb{F}_2[x]$. Todos os polinómios do protocolo pertencem ao anel $\mathbb{F}_2[x]/f(x)$, sendo que um elemento deste anel tem grau máximo $deg(f) - 1$. No âmbito deste projecto, serão considerados dois anéis $\mathsf{R} = \mathbb{F}_2[x]/f(x)$: um anel em que $f(x)$ é irredutível e outro em que $f(x)$ é factorizável em m polinómios distintos. Denota-se por $\widehat{a}$ a representação CRT (Teorema Chinês dos Restos) do polinómio a em relação aos m factores do $f(x)$ factorizável, ou seja, $\widehat{a} = (a \bmod f_1, \ldots, a \bmod f_m)$

2.2.2 Distribuições

Uma distribuição D sobre determinado domínio representa-se sob a forma de $r \longleftarrow_{\$} \mathsf{D}$, sendo r o valor gerado de acordo com a distribuição D. Denomina-se uma distribuição uniforme sobre o domínio Y como $\mathsf{U(Y)}$. Seja a distribuição de Bernoulli, Ber_τ sobre $\mathbb{F}_2$ com o parâmetro $\tau \in]0, 1/2[$. Para um anel polinómio $\mathsf{R} = \mathbb{F}_2[x]/f(x)$, a distribuição $\mathsf{Ber}_\tau^{\mathsf{R}}$ denota a distribuição sobre polinómios de R, para os quais os coeficientes são determinados independentemente de Ber_τ. Para um anel R e um polinómio $s \in \mathsf{R}$, representa-se $\Lambda_\tau^{\mathsf{R},s}$ como uma distribuição sobre $\mathsf{R} \times \mathsf{R}$ sendo as amostragens obtidas através dos polinómios $r \longleftarrow_{\$} \mathsf{U(R)}$ e $e \longleftarrow_{\$} \mathsf{Ber}_\tau^{\mathsf{R}}$, cujo resultado é o seguinte *output*: $(r, rs + e)$.

2.2.3 Ring Learning Parity with Noise (Ring-LPN) e Learning Parity with Noise (LPN)

A segurança do protocolo depende problema do Ring-LPN que se trata de uma expansão do problema LPN para anéis. Este problema pode também ser visto como uma instanciação particular de um outro problema com grande ligação aos reticulados, o Ring-LWE (*Learning with Errors over Rings*).
A diferença entre os dois problemas reside na diferença entre dois possíveis oráculos. O primeiro oráculo gera aleatoriamente um vector secreto $s \in \mathbb{F}_2^n$ que é usado produzir a resposta. No problema LPN, cada chamada do oráculo produz, de forma uniforme, uma matriz aleatória $A \in \mathbb{F}_2^{n \times n}$ e um vector $As + e = t \in \mathbb{F}_2^n$, onde e é um vector de $\mathbb{F}_2^n$ em que cada entrada é um valor aleatório gerado independentemente pela distribuição de *Bernoulli* com a probabilidade de 1 usando o parâmetro público τ entre 0 e $1 \setminus 2$. O segundo oráculo gera uma matriz $A \in \mathbb{F}_2^{n \times n}$ aleatória de forma uniforme e um vector aleatório $t \in \mathbb{F}_2^n$ de forma igualmente uniforme.
A diferença entre o LPN e o Ring-LPN está na geração da matriz A, em ambos os oráculos. Enquanto no problema LPN todas as entradas são geradas de forma uniforme e independente, no problema Ring-LPN apenas a primeira coluna é gerada dessa forma em $\mathbb{F}_2^n$, sendo que as restantes colunas dependem da primeira e do anel subjacente $\mathsf{R} = \mathbb{F}_2[x]/f(x)$. De assinalar ainda que a suposição Ring-LPN$^{\mathsf{R}}$ indica que é difícil distinguir entre os *outputs* dos dois oráculos.
O problema LPN é bastante usado em criptografia como uma suposição difícil, ao contrário do Ring-LPN. Contudo, uma publicação recente demonstra que o problema Ring-LWE é tão difícil quanto resolver quanticamente o pior caso de um pequeno vector de reticulados. Por sua vez, o Ring-LPN é bastante semelhante ao problema Ring-LWE.

2.3 Funcionamento

O protocolo é definido sobre o anel $\mathsf{R} = \mathbb{F}_2[x]/f(x)$ e envolve um *mapping* adequado $\pi : \{0,1\}^\lambda \to \mathsf{R}$. Este *mapping* deve ser definido de tal forma que $\forall\, c, c' \in \{0,1\}^\lambda$ tem-se que $\pi(c) - \pi(c') \in \mathsf{R} \setminus \mathsf{R}^\star$ sse $c = c'$.
Descreve-se de seguida o funcionamento do protocolo:

Parâmetros Públicos: $\mathsf{R}, \pi : \{0,1\}^\lambda \to \mathsf{R}, \tau, \tau'$
Chave Privada: $s, s' \in \mathsf{R}$

Tag $\mathcal{T}$		Leitor $\mathcal{R}$
	$\xleftarrow{c}$	$c \leftarrow_\$ \{0,1\}^\lambda$
$r \leftarrow_\$ \mathsf{R}^*$; $e \longleftarrow_\$ \mathsf{Ber}_\tau^R \in \mathsf{R}$		
$z := r \cdot (s \cdot \pi(c) + s') + e$	$\xrightarrow{(r,z)}$	
		se $r \notin \mathsf{R}^\star$ retorna reject
		$e' := z - r \cdot (s \cdot \pi(c) + s')$
		se $\mathsf{wt}(e') > n \cdot \tau'$ retorna reject
		retorna accept

Parâmetros públicos τ, τ' representam constantes (definidas mais adiante), n depende do parâmetro de segurança λ:

- Anel $\mathsf{R} = \mathbb{F}_2[x]/f(x)$ com $deg(f) = n$

- *Mapping* $\pi : \{0,1\}^\lambda \to \mathsf{R}$
- Parâmetro da distribuição de Bernoulli $\tau \in \mathbb{R}$ e *threshold* aceitação $\tau' \in \mathbb{R}$ tal que $0 < \tau < \tau' < 1/2$

Geração de chaves Algoritmo $\mathsf{KeyGen}(1^\lambda)$ amostra $s, s' \longleftarrow_\$ \mathsf{R}$ e retorna s, s' como a chave privada

Protocolo de autenticação O Leitor $\mathcal{R}$ e a *Tag* $\mathcal{T}$ partilham a chave secreta $s, s' \in \mathsf{R}$. Para que $\mathcal{T}$ seja autenticada por $\mathcal{R}$, ambos executam os seguintes passos:

1. $\mathcal{R}$ gera um *challenge* $c \longleftarrow_\$ \{0,1\}^\lambda$. $\mathcal{R}$ **envia c para** $\mathcal{T}$
2. $\mathcal{R}$ gera $r \longleftarrow_\$ \mathsf{R}$, $e \longleftarrow_\$ \mathsf{Ber}_\tau^\mathsf{R}$ e calcula $z = r \cdot (s \cdot \pi(c) + s') + e$. $\mathcal{T}$ **envia o par** $(\mathbf{r}, \mathbf{z})$ **para** $\mathcal{R}$
3. $\mathcal{T}$ recebe o par (r, z) e:
   - se $r \notin \mathsf{R}^\star$, retorna reject e protocolo termina;
   - calcula $e' = z - r \cdot (s \cdot \pi(c) + s')$;
   - se $\mathsf{wt}(e') > n \cdot \tau'$, retorna reject e protocolo termina[1];
   - retorna accept

2.3.1 Polinómio redutível

Por questões de eficiência, por vezes é melhor utilizar um polinómio $f(x)$ que seja redutível sobre $\mathbb{F}_2$. Isto permite-nos utilizar a representação CRT dos elementos de $\mathbb{F}_2[x]/f(x)$ para efectuar multiplicações que se tornam muitos mais eficientes quando em CRT.
Se o polinómio é factorizável em $\prod_{i=1}^m f_i$, então é possível tentar resolver o problema do Ring-LPN módulo um qualquer f_i, em vez de f. Sendo que o $deg(f_i) < deg(f)$, resolver o Ring-LPN torna-se mais fácil.
A implementação do protocolo através da utilização de um polinómio redutível permite-nos tirar vantagens da aritmética baseada no Teorema Chinês dos Restos.
Na implementação é definido o anel $\mathsf{R} = \mathbb{F}_2[x]/f(x)$, e escolhido o polinómio redutível f como o produto de $m = 5$ polinómios irredutíveis:

$$\begin{array}{rcccccccccc}
f_1 & = & x^{127} & + & x^8 & + & x^7 & + & x^3 & + & 1 \\
f_2 & = & x^{126} & + & x^9 & + & x^6 & + & x^5 & + & 1 \\
f_3 & = & x^{125} & + & x^9 & + & x^7 & + & x^4 & + & 1 \\
f_4 & = & x^{122} & + & x^7 & + & x^4 & + & x^3 & + & 1 \\
f_5 & = & x^{121} & + & x^8 & + & x^6 & + & x & + & 1
\end{array}$$

Por isso, $deg(f) = n = 621$. É definido $\tau = 1/6$ e $\tau' = 0.29$ de forma a obter um erro de solidez mínimo $\varepsilon_{ms} \approx c(\tau', 1/2)^{-n} \leq 2^{-82}$ e um erro de integridade $\varepsilon_c \leq 2^{-42}$. O melhor ataque conhecido sobre $\mathsf{Ring\text{-}LPN}_\tau^\mathsf{R}$ tem como parâmetro a complexidade $> 2^{80}$.
O *mapping* $\pi : \{0,1\}^{80} \to \mathsf{R}$ é definido no Algoritmo 1. Cada v_i pode ser visto como a representação dos coeficientes de um polinómio em $\mathbb{F}_2[x]/f_i(x)$. O *mapping* π está representado na forma CRT. De referir ainda que, para $c, c^\star \in \{0,1\}^{80}$, temos que $\pi(c) - \pi(c^\star) \in \mathsf{R} \setminus \mathsf{R}^\star$ sse $c = c^\star$ e por isso π é adequado para R.

[1] $\mathsf{wt}(e)$ representa o *hamming weight* de uma *string* binária e, ou seja, o número de bits 1 em e

Algoritmo 1 *Mapping* π para o anel $\mathsf{R} = \mathbb{F}_2[x]/f(x)$, no caso em que $f(x)$ é irredutível

Input: $c \in \{0,1\}^{80}$
Output: $\pi = (v_1, \ldots, v_5)$
  for $i = 1$ to 5 **do**
    $toPad = deg(f_i) - 80$
    $v_i = padding(c, toPad)$
  end for
  return v

Algoritmo 2 *Mapping* π para o corpo $\mathsf{F} = \mathbb{F}_2[x]/f(x)$, no caso em que $f(x)$ é redutível

Input: $c \in \{0,1\}^{80} = (c_1, \ldots, c_16)$, com cada c_j um número de 5 bits entre 1 e 31
Output: $(v_1, \ldots, v_{16})$ os 16 coeficientes não nulos.
  for $j = 1$ to 16 **do**
    $v_i = 16 \cdot (j-1) + c_j$
  end for
  return v

2.3.2 Polinómio irredutível

Quando $f(x)$ é irredutível em $\mathbb{F}_2$, o anel $\mathbb{F}_2[x]/f(x)$ é um corpo. Para estes anéis, não são conhecidos algoritmos capazes de tirar partido da estrutura algébrica adicionada a esta instância particular do Ring-LPN. Desta forma, o algoritmo conhecido mais eficiente para resolver este problema é usado para resolver o LPN.
A complexidade computacional do problema LPN baseia-se no tamanho de n e na distribuição de ruído Ber_τ. Intuitivamente, quanto maior for n e τ mais próximo de $1/2$, mais difícil se torna o problema.
Habitualmente, são considerados como valores de τ valores constantes entre 0.05 e 0.25. Para tais valores de τ, o algoritmo assimptótico mais rápido para a resolução do problema LPN demora $2^{\Omega(n/log\ n)}$ e requer aproxidamente $2^{\Omega(n/log\ n)}$ amostragens do oráculo LPN. No caso do número de amostragens ser menor, o algoritmo, naturalmente, tem uma performance inferior.
No protocolo o número de amostragens disponíveis para o adversário é limitado a n vezes o número de execuções. Então, assumindo que o adversário tem acesso a um número limitado de amostragens, é mais difícil quebrar o protocolo de autenticação do que resolver o problema Ring-LPN.
Para definir o corpo $\mathsf{F} = \mathbb{F}_2[x]/f(x)$ foi escolhido o polinómio irredutível:

$$f(x) = x^{532} + x + 1$$

Por isso com grau $n = 532$, e definidos $\tau = 1/8$ e $\tau' = 0.27$ que permite obter um erro de solidez mínimo $\varepsilon_{ms} \approx c(\tau', 1/2)^{-n} \leq 2^{-80}$ e um erro de integridade $\varepsilon_c \leq 2^{-55}$.
De acordo com análises citadas na publicação, o algoritmo mais rápido na resolução do problema LPN de dimensão 532 com $\tau = 1/8$ requer 2^{77} de memória para o resolver quando é dado aproxidamente o mesmo número de amostras. Dado que a dimensão é pouco maior e o número de amostras limitado, é razoável assumir que esta instância tem segurança de 80 bits.
O *mapping* $\pi : \{0,1\}^{80} \to \mathsf{F}$ é definido no Algoritmo 1. Dado que π descrito é injectivo e F é um corpo, o *mapping* π é adequado para F.

Capítulo 3

Lapin – Implementação

3.1 Representações e operações em Sage

Antes de partir para a implementação do protocolo em C, decidimos criar um pequeno protótipo em Sage. A grande vantagem é o facto de o Sage ter diversas classes definidas para a representação de polinómios, sendo que as operações usuais de soma, multiplicação ou módulo estão já implementadas, poupando-nos algum tempo que pode ser utilizado para melhor perceber o funcionamento do protocolo.
Apresentamos neste secção algumas das classes usadas, e como implementar algumas das operações do protocolo.
Para se representar $\mathbb{F}_2[x]$ usa-se:

```
F = PolynomialRing(GF(2), 'x')
```

Para declarar um polinómio não basta declará-lo com, por exemplo, `x^2 + x`, pois o tipo da variável `x` é uma `Expression` do Sage. Posto isto, é preciso declarar a variável `x` para que o Sage saiba que está a lidar com polinómios e não com simples `Expressions` Sage. Sendo assim, para declarar $f(x) = x^7 + x + 1$ faz-se:

```
sage: x = F.gen()
sage: f = x**7 + x + 1
```

É desta forma que vamos definir todos os polinómios em Sage. Agora é possível efectuar operações sobre polinómios. Seguem alguns exemplos, em que todas as operações são efectuadas em $\mathbb{F}_2[x]$:

```
sage: g = x**5 + x**2 + x + 1
sage: f+g
x^7 + x^5 + x^2
sage: f*g
x^12 + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + 1
sage: f.mod(g)
x^4 + x^3 + x^2 + x + 1
sage:
```

Tendo bem definido como se tratariam dos polinómios, é necessário representar o anel R. Usando como exemplo o anel $\mathbb{R}_2[x]/f(x)$ tem-se:

```
sage: R = R.quotient_ring(f, 'x')
sage: R.random_element()
x^6 + x^5 + x^4 + x
# CAREFUL: type of this R.random_element() doesn't support mod operations,
    so we have to convert it to type Polynomial_GF2X
```

```
sage: type(R.random_element())
sage.rings.polynomial.polynomial_quotient_ring_element.Polynomial
QuotientRing_generic_with_category.element_class
sage: type(binToPoly(polyToBin(R.random_element(), x), x)
sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X
```

Note-se que as funções `binToPoly` e `polyToBin` estão definias no Anexo **??** Como também vamos ter de trabalhar com polinómios em CRT, vamos criar um exemplo de um anel R com um $f(x) = x^8 + x^6 + x^4 + x^3 + x$ factorizável em três polinómios x, $x^3 + x + 1$ e $x^4 + x + 1$ e com algumas operações em CRT:

```
sage: fi = [x, x**3 + x + 1, x**4 + x + 1]; f = reduce(operator.mul, fi, 1)
sage: R = F.quotient_ring(f, 'x')
sage: z = R.random_element();
# convert z to Polynomial_GF2X
...
sage: zCRT = map(lambda m : z.mod(m), fi) ; zCRT
[1, x^2, x^2 + 1]
sage: CRT_list(zCRT, fi) == z
True
sage: w = R.random_element()
# convert w to Polynomial_GF2X
...
sage: wCRT = map(lambda m : w.mod(m), fi) ; wCRT
[1, x + 1, x^3 + x + 1]
# perform w*z (mod f)
sage: tmp1 = zip(wCRT, fi); tmp2 = zip(zCRT, fi)
sage: wzCRT=map(lambda ((x1, y1), (x2, y2)) : (x1*x2).mod(y1),
    zip(tmp1,tmp2))
sage: CRT_list(wzCRT, fi) == (w*z).mod(f)
True
```

Como também vamos precisar de trabalhar com a string de bits $c \in \{0,1\}^\lambda$, seguem-se os comandos para a criar, converter para representação inteira e para fazer *padding*:

```
sage: c = bin(getrandbits(80))[2:].zfill(80) ; Integer(c,2)
64597992717886966981887
# type of c is str and len(c) = 80
sage: ''.join(['0' * 5], c) # pad c with 5 bits
```

3.2 Algumas operações em Sage

Tendo já conhecimento dos métodos apresentados anteriormente, é relativamente fácil concluir a implementação do Lapin no Sage. Apresentámos agora alguns dos métodos e funções com maior relevância. Todo o código pode ser consultado no Anexo A.
Decidimos dividir a implementação do Lapin em três classes:

Lapin contém a definição dos parâmetros do protocolo, conforme os recomendados em [4]. Contém todas as operações do protocolo apresentadas em 2.3, à excepção do *mapping* π;

Reducible contém todas as operações comuns ao protocolo no caso em que f é redutível. É aqui definido o *mapping* π e o anel R

Irreducible contém todas as operações comuns ao protocolo no caso em que f é irredutível. É aqui definido o *mapping* π e o corpo F

As operações auxiliares comuns não estão em nenhuma classe de forma a possibilitar a sua utilização por todas as classes/funções do protocolo. Exemplo de tais funções são as que implementam somas e multiplicações em CRT ou operações binárias.
De seguida apresenta-se o passo 2 do protocolo no caso em que f é redutível:

```
# method of Reducible class
def tag_step2(self, c):
  # ....

  # list of factors of f
  fi = self.protocol.fi

  r = reduceToCRT(self.protocol.genR(), fi)
  e = reduceToCRT(self.protocol.genE(self.tau), fi)
  # keys in crt form
  (s1, s2) = (reduceToCRT(self.key1, fi), reduceToCRT(self.key2, fi))
  pi = self.protocol.pimapping(c)

  z = addCRT(multCRT(r, addCRT(multCRT(s1, pi, fi), s2, fi), fi), e, fi)
  return (r, z)

# ...
# function, not a method of any class
def reduceToCRT(a, fi):
  return map(lambda f : a.mod(f), fi)
```

O método `pimapping` é a implementação do Algoritmo 1.
O passo 3 consiste na verificação. Eis a sua implementação para o caso em que f é irredutível:

```
# method of Irreducible class
def reader_step3(self, c, r, z):
  # ....

  # list of factors of f
  fi = self.protocol.fi

  (r1, z1) = (CRT_list(r, fi), CRT_list(z, fi))
  if r1.gcd(self.protocol.f) != 1:
    print "reject R*"
    return False
  pi = CRT_list(self.protocol.pimapping(c), fi)

  e2 = (z1 - r1 * (self.key1 * pi + self.key2)).mod(self.protocol.f)

  if e2.hamming_weight() > (self.n * self.tau2):
    print "reject wt"
  return False

  print "accept"
  return True
```

3.3 Polinómios binários

Como pretendermos implementar de raíz o protocolo em C, é necessário perceber como funcionam as operações sobre polinómios ao mais baixo nível, ou seja, apenas com a utilização de operações sobre bits.
Neste capítulo pretende-se explicar como implementar as operações aritmétricas mais comuns sobre polinómios de forma eficiente. Até ao momento, só efectuámos a análise de duas opera-

Algoritmo 3 Soma de palavras binárias de W bits

Input: Polinómios $a(x)$ e $b(x)$ de grau menor ou igual a $m-1$
Output: Polinómio $c(x) = a(x) + b(x)$
  for $i = 0$ to $t-1$ **do**
    $C[i] = A[i] \oplus B[i]$
  end for
  return c

ções: soma e multiplicação. A soma é de relativamente fácil percepção e implementação, mas na multiplicação demorámos mais tempo a perceber como realmente funciona. Devido a este facto, não nos é possível apresentar ainda a implementação das outras operações aritmétricas necessárias, tais como o módulo e máximo divisor comum (gcd).

3.3.1 Representação binária

De forma a que as operações com polinómios sejam o mais eficiente possível, é necessário trabalhar sobre a sua representação binária, em vez de os representar com simples inteiros. Neste capítulo descreve-se a representação usada na implementação do Lapin.
Uma forma bastante simples de representar polinómios é colocar o bit menos significativo do lado direito, sendo a tradução de um polinómio para binário a seguinte:

$$x^7 + x^4 + x^2 + x + 1 \Leftrightarrow 10010111$$

O bit mais significativo representa o monómio x^7, enquanto que o menos significativo representa o 1.
Mas esta forma, por si só, não é a mais apropriada para implementar as operações aritmétricas em *software*. Em [4], o autor refere-se a um método de multiplicação de polinómios mais eficiente, descrito em [2]. Antes de descrever esses métodos, é necessário introduzir uma nova forma de representar polinómios binários.
Nesta nova forma, assume-se que a máquina tem um arquitectura de W-bits, em que W é um múltiplo de 8. Um polinómio passa a ser visto como um conjunto de palavras de W bits, em que o bit menos significativo continua a estar do lado direito.
Assume-se que existe o polinómio irredutível $f(x)$ com $deg(f) = m$. Todos os elementos $a(x)$ de $\mathbb{F}_2[x]/f(x)$ tem grau menor ou igual a $m-1$. Cada elemento $a(x)$ tem associada a representação descrita anteriormente, ou seja, é o vector binário $a = (a_{m-1}, a_m, \ldots, a_1, a_0)$ de tamanho m. Considere-se $t = \lceil m/W \rceil$ e $s = Wt - m$. O vector binário a pode ser guardado num *array* de t palavras de W bits: $A = A[t-1], A[t], \ldots, A[1], A[0]$. Os s bits mais à direita de $A[t-1]$ nunca são usados (estão sempre a 0).
A partir de agora vamos usar sempre esta última representação para tratar de polinómios.

3.3.2 Soma

A soma de dois polinómios pode ser vista como um simples XOR entre as palavras dos dois polinómios, sendo efectuado o XOR palavra a palavra, bit a bit. No Algoritmo 3 descreve-se este processo.

3.3.3 Multiplicação de dois polinómios binários

Ao contrário da soma descrita no Algoritmo 3, a multiplicação de polinómios já não é tão simples. Em [4], sugere-se a utilização do o método *right-to-left comb* descrito em [2] por ser

Algoritmo 4 Método *Right-to-left comb* para a multiplicação de dois polinómios

Input: Polinómios $a(x)$ e $b(x)$ de grau menos ou igual a $m-1$
Output: Polinómio $c(x) = a(x) \cdot b(x)$

```
C = 0
for k = 0 to W − 1 do
    for j = 0 to t − 1 do
        if k-ésimo bit de A[j] é 1 then
            Adicionar B a C{j}
            c(x) = c(x) + b(x) · x^(Wj+k)
            C = C{j} + B
        end if
    end for
    if k ≠ (W − 1) then
        b(x) = b(x) · x
        B = B · x
    end if
end for
return C
```

o mais eficiente.
Tal como para a soma, um polinómio é visto como t palavras de W bits. Em cada iteração conhece-se o resultado de $b(x) \cdot x^k$ para $k \in [0, W-1]$, logo, $b(x) \cdot x^{Wj+k}$ pode facilmente ser obtido adicionando j palavras nulas (a zero) à direita do vector que representa $b(x) \cdot x^k$. Tal como o próprio nome indica, as palavras de A são percorridas da direita para a esquerda. Quando se tem um array $C = C[n], \ldots, C[1], C[0]$, denota-se por $C\{j\} = C[n], \ldots, C[j+1], C[j]$ como o array truncado.

Capítulo 4

Conclusão

Baseado no estudo efectuado sobre o protocolo de autenticação Lapin, pode-se afirmar que o mesmo é uma opção viável e segura para implementar em sistemas *low-cost* e com recursos limitados, em que se pretenda um protocolo com pequena complexidade comunicacional. A análise de [4] permitiu também verificar a utilidade de algumas operações realizadas sobre polinómios que permitem alcançar resultados muito mais eficientes comparativamente às operações realizadas de forma habitual.
De assinalar ainda que a utilização da suposição Ring-LPN, apesar de ainda não ser um problema muito utilizado em criptografia, vem demonstrar que a sua utilização pode vir a aumentar, especialmente no âmbito das construções de esquemas *low-cost*.

4.1 Trabalho futuro

Depois de, nesta primeira fase, ter sido estudada a publicação referente ao protocolo Lapin e, inclusivamente, este ter sido implementado em `Sage`, pretende-se que na fase seguinte seja implementado em C/C++ (de momento, referimos C/C++, apesar de estarmos mais inclinados para C e não C++). Como foi sendo dito ao longo deste documento, o protocolo Lapin é extremamente eficiente, pelo que nos parece óbvio usar C/C++ para se obter uma aplicação igualmente eficiente.
Mas ainda antes de partir para a implementação em C/C++, é necessário acabar a implementação do método *right-to-left comb* (Algoritmo 4) e as restantes operações aritmétricas sobre polinómios, utilizando apenas operações binárias eficientes.
Relativamente a [5], a análise feita até ao momento não foi tão aprofundada quanto à feita a [4], portanto optámos por não incluir nada sobre o mesmo neste relatório. Contudo, o nosso propósito é o de também implementar o esquema de assinaturas baseado em reticulados em `Sage`, e se possível ainda, tentar uma implementação em C/C++. Dado que o estudo da primeira publicação nos permitiu ter uma noção exacta da complexidade da implementação da mesma, consideramos os objectivos descritos para o conjunto deste Projecto Integrador II atingíveis e razoáveis.

Bibliografia

[1] *Sage's Reference Manual*, April 2013. http://www.sagemath.org/doc/reference/.

[2] Hankerson, Darrel and Menezes, Alfred J. and Vanstone, Scott. *Guide to Elliptic Curve Cryptography*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.

[3] Menezes, Alfred J. and Vanstone, Scott A. and Oorschot, Paul C. Van. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 1996.

[4] Stefan Heyse and Eike Kiltz and Vadim Lyubashevsky and Christof Paar and Krzysztof Pietrzak. Lapin: An Efficient Authentication Protocol Based on Ring-LPN. In *Fast Software Encryption*, pages 346–365, 2012. http://dx.doi.org/10.1007/978-3-642-34047-5_20.

[5] Vadim Lyubashevsky. Lattice Signatures Without Trapdoors. Cryptology ePrint Archive, Report 2011/537, 2011. http://eprint.iacr.org/2011/537.

Apêndices

Apêndice A

Lapin em Sage

A.1 Protocolo Lapin, classe principal

Código A.1: Implementação do protocolo Lapin em Sage

```
class Lapin:
  """ Lapin superclass """
  def __init__(self, reducible=False):
    self.reducible = reducible
    if reducible == False:
      self.sec_param = 80
      self.tau = 1/8
      self.tau2 = 0.27
      self.n = 532
      self.protocol = Irreducible()
    else:
      self.sec_param = 80
      self.tau = 1/6
      self.tau2 = 0.29
      self.n = 621
      self.protocol = Reducible()

  def genKey(self):
    self.key1 = binToPoly(polyToBin(self.protocol.R.random_element(),
        self.protocol.x), self.protocol.x)
    self.key2 = binToPoly(polyToBin(self.protocol.R.random_element(),
        self.protocol.x), self.protocol.x)

  # generate c
  def reader_step1(self):
    # generate binary string with len(c) = self.sec_param
    return bin(getrandbits(self.sec_param))[2:].zfill(self.sec_param)

  # generate (r,z)
  def tag_step2(self,c):
    if self.reducible == False:
      """
      " Irreducible
      """
      r = protocol.genR()
      e = protocol.genE(self.tau)
      pi = protocol.pimapping(c)
      z = r * (self.key1 * pi + self.key2) + e
      return (r,z)
    elif self.reducible == True:
      """
```

```
      " Reducible
      """
      fi = self.protocol.fi

      r = reduceToCRT(self.protocol.genR(), fi)
      e = reduceToCRT(self.protocol.genE(self.tau), fi)
      # keys in crt form
      (s1, s2) = (reduceToCRT(self.key1, fi), reduceToCRT(self.key2, fi))
      pi = self.protocol.pimapping(c)

      z = addCRT(multCRT(r, addCRT(multCRT(s1, pi, fi), s2, fi), fi), e, fi)
      return(r,z)
    else:
      print "irreducible parameter must be either 0 or 1"

  # verification
  def reader_step3(self,c,r,z):
    if self.reducible == False:
      """
      " Irreducible
      """
      if r.gcd(self.protocol.f) != 1:
        print "reject"
        return False
      e2 = (z - r * (self.key1 * protocol.pimapping(c) +
          self.key2)).mod(self.protocol.f)
      if e2.hamming_weight() > (self.n * self.tau2):
        print "reject"
        return False
      print "accept"
      return True
    elif self.reducible == True:
      """
      " Reducible
      """
      fi = self.protocol.fi
      (r1, z1) = (CRT_list(r, fi), CRT_list(z, fi))
      if r1.gcd(self.protocol.f) != 1:
        print "reject"
        return False
      pi = CRT_list(self.protocol.pimapping(c), fi)

      e2 = (z1 - r1 * (self.key1 * pi + self.key2)).mod(self.protocol.f)

      if e2.hamming_weight() > (self.n * self.tau2):
        print "reject"
        return False

      print "accept"
      return True
    else:
      print "irreducible parameter must be either 0 or 1"
      return False
```

A.2 Operações do Lapin, com o polinómio redutível

Código A.2: Operações do protocolo, com o polinómio redutível, em Sage

```
class Reducible:
```

```
  """ Reducible protocol """
  def __init__(self):
    (self.R, self.f, self.fi, self.x) = self.__initRing()

  def __initRing(self):
    F = PolynomialRing(GF(2), 'x')
    x = F.gen()
    # irreducible pentanomials
    f1 = x^127 + x^8 + x^7 + x^3 + 1
    f2 = x^126 + x^9 + x^6 + x^5 + 1
    f3 = x^125 + x^9 + x^7 + x^4 + 1
    f4 = x^122 + x^7 + x^4 + x^3 + 1
    f5 = x^121 + x^8 + x^5 + x + 1
    # calculate polynomial f
    f = f1 * f2 * f3 * f4 * f5
    # create the ring
    R = F.quotient_ring(f, 'x')
    return (R, f, [f1,f2,f3,f4,f5], x)

  # c must be a binary
  # returns a list of polynomials in CRT
  def pimapping(self, c):
    v = [] # list of v_i
    for p in self.fi:
      toPad = p.degree() - 80
      # pad to the left
      poly = (''.join(['0' * toPad, c]))
      v.append(binToPoly(poly, self.x))
    return v

  # generate r
  def genR(self):
    # workaround to convert to
      sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X
    r = binToPoly(polyToBin(self.R.random_element(), self.x), self.x)
    return r

  # generate e
  def genE(self, tau):
    l = []
    e = 0*self.x
    for i in xrange(0, self.f.degree()):
      ci = bernoulli(tau)
      if ci == 1:
        e += self.x**i
    return e
```

A.3 Operações do Lapin, com o polinómio irredutível

Código A.3: Operações do protocolo, com o polinómio redutível, em Sage

```
class Irreducible:
  """ Irreducible protocol """
  def __init__(self):
    (self.R, self.f, self.x) = self.__initRing()

  def __initRing(self):
    F = PolynomialRing(GF(2), 'x')
```

```
    x = F.gen()
    f = x^532 + x + 1
    R = F.quotient(f, 'x')
    return (R, f, x)

  # returns the list of coefficients of the new polynomial v
  # c must be in binary format
  def pimapping(self, c):
    v = 0*self.x
    for j in xrange(0, 16):
      # 32 = 5 bits
      cj = c[j*5 : (j*5) + 5]
      # here we use x instead of the (x-1) indicated in the paper
      # because we're in 0-index mode
      i = (16 * j) + binToInt(cj) # TODO: use binary operations here as well
      v += self.x**i
    return v

  # generate r
  def genR(self):
    # workaround to convert to
        sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X
    r = binToPoly(polyToBin(self.R.random_element(), self.x), self.x)
    while r.gcd(self.f) != 1:
      # workaround to convert to
          sage.rings.polynomial.polynomial_gf2x.Polynomial_GF2X
      r = binToPoly(polyToBin(self.R.random_element(), self.x), self.x)
    return r

  # generate e
  def genE(self, tau):
    l = []
    e = 0*self.x
    for i in xrange(0, self.f.degree()):
      ci = bernoulli(tau)
      if ci == 1:
        e += self.x**i
    return e
```

A.4 Funções comuns usadas pelo Lapin

Código A.4: Funções usadas pelo protocolo Lapin em Sage

```
""" Aux methods """
def bernoulli(tau):
  return int(random() < tau)

# p: poly, var=x, fill=length in binary
def polyToBin(p, var, fill=0):
  l = p.list()
  # traverse in reversed order
  return (''.join(str(l[bit]) for bit in xrange(len(l)-1, -1,
      -1))).zfill(fill)

# convert a binary string to a polynomial
# [1,0,1,1,1] = x^4 + x^2 + x + 1
def binToPoly(b, var):
  v = 0*var
  n = len(b)
```

```
  for i in xrange(0, n):
    if int(b[i]) == 1:
      v += var**(n - i - 1)
  return v

def intToBin(i, fill=0):
  return ''.join(bin(i)[2:]).zfill(fill)

# converts a binary string to int
def binToInt(i):
  return Integer(i, 2)

# perform bitwise xor on two bins
# lists can have different length
def bitwiseXor(a, b):
  lenA = len(a)
  lenB = len(b)
  if len(a) >= len(b):
    diff = lenA - lenB
    c = a[:diff]
    c += ''.join(str(e) for e in map(lambda x,y : int(x).__xor__(int(y)),
        a[diff:], b))
  else:
    diff = lenB - lenA
    c = b[:diff]
    c += ''.join(str(e) for e in map(lambda x,y : int(x).__xor__(int(y)), a,
        b[diff:]))
  return c

# reduces a polynomial to its CRT form
# a: polynomial to reduce
# fi: list of modulus
def reduceToCRT(a, fi):
  return map(lambda f : a.mod(f), fi)

# f1 and f2 must be in CRT form
# result in CRT form
def multCRT(p1, p2, fi):
  p1zip = zip(p1, fi)
  p2zip = zip(p2, fi)

  # returns a list with all multiplications done modulo each fi
  # CRT_list(return value, fi) == (p1*p2).mod(fi)
  return map(lambda ((x1, y1), (x2, y2)) : (x1 * x2).mod(y1), zip(p1zip,
      p2zip))

# p1 and p2 must be in CRT form
# result in CRT form
def addCRT(p1, p2, fi):
  pAddZip = zip(p1,p2)

  return map(lambda (x,y,z) : (x+y).mod(z), zip(p1, p2, fi))
```

Apêndice B

Operações binárias em Sage

B.1 Operações com polinómios binários

De seguida apresenta-se a soma de polinómios binários. Infelizemente, não conseguimos terminar a tempo a implementação da multiplicação de polinómios, mas neste apêndice encontram-se a maior parte das funções necessárias à sua implementação.

Código B.1: Implementação da soma de polinómios binários. Contém também algumas funções auxiliares para conversão de polinómios para binário e vice-versa.

```
class BinaryPolynomial:
  def __init__(self, f, W=8):
    self.var = PolynomialRing(GF(2),'x').gen()
    if type(f) != type(self.var):
      # convert expression f to polynomial
      f = expressionToPoly(f, self.var)

    self.f = f
    self.W = W
    self.m = self.f.degree()
    self.t = ceil(self.m / self.W)
    self.s = (self.W * self.t) - self.m

  # convert polynomial to its representation in binary, with W-bit words
  def polyToBin(self, a):
    if type(a) != type(self.var):
      # convert a to poly type
      a = expressionToPoly(a, self.var)
    if a.degree() >= self.m:
      print "degree a >= degree m"
      return
    aList = a.list()
    A = [0] * (self.t * self.W)

    lenA = len(A)
    for i in xrange(0, len(aList)):
      A[lenA - i - 1] = int(aList[i])
    return A

  # convert an array of W-bit words to a polynomial
  def binToPoly(self, A):
    a = self.var*0
    lenA = len(A)
    for i in xrange(0, self.m):
      # len(A) - i - 1 == last position of A == a_0
      pos = lenA - i - 1
```

```
      #print "x^{0} = {1}".format(i, pos)
      if A[pos] == 1:
        a += self.var**i
    return a

  # get the i-th W-bit word from an array
  def getWord(self, A, i):
    if i >= (len(A) // self.W) or i < 0:
      print "Error: cant found {0}-th word. Allowed: 0 .. {1}".format(i,
          len(A)-1)
      return
    start = len(A) - (self.W * i)
    end = start - self.W
    return A[end : start]

  # get the i-th bit of the W-bit word a
  def getBit(self, a, i):
    if len(a) != self.W:
      print "Word a is not a {0}-bit word!. Found W={1}
          instead.".format(self.W, len(a))
    return a[len(a) - i - 1]

  # Given an array in the form C = (C[n],..,C[1],C[0]),
  # this returns the truncated array (C[n],..,C[j+1],C[j])
  def truncate(self, A, j):
    t = len(A) / self.W
    C = []
    for i in xrange(t - 1, j - 1, -1):
      C.extend(self.getWord(A, i))
    return C

  # A and B are already in binary form
  # C is returned in binary form
  def polyAddition(self, a, b):
    A = self.polyToBin(a)
    B = self.polyToBin(b)
    return bitwiseXor(A, B)

""" Auxiliary functions """

# convert a sage expression to a polynomial
def expressionToPoly(e, var):
  p = 0*var
  for op in e.coeffs():
    p += var**op[1]
  return p

# perform bitwise xor on two lists
# lists can have different length
def bitwiseXor(a, b):
  lenA = len(a)
  lenB = len(b)
  if len(a) >= len(b):
    diff = lenA - lenB
    c = a[:diff]
    c.extend(map(lambda x,y : int(x).__xor__(int(y)), a[diff:], b))
  else:
    diff = lenB - lenA
    c = b[:diff]
    c.extend(map(lambda x,y : int(x).__xor__(int(y)), a, b[diff:]))
  return c
```

```
# leftmost s bits are not considered in the sift operation
# this is equivalent to b = a*x
def shiftRight(A, s=0):
  toShift = A[s:]
  b = toShift.pop(0)
  toShift.append(b)
  B = [0] * s
  B.extend(toShift)
  return B

def shiftLeft(A, s=0):
  toShift = A[s:]
  b = toShift.pop(len(toShift)-1)
  B = [0] * s
  B.append(b)
  B.extend(toShift)
  return B
```