Implementação eficiente de criptografia de curvas elípticas em sensores sem fio*

Diego Aranha, Danilo Câmara, Julio López, Leonardo Oliveira, Ricardo Dahab[1]

[1] Instituto de Computação – Universidade Estadual of Campinas (UNICAMP)
CEP 13083-970 – Campinas – SP - Brasil
{dfaranha,dfcamara,jlopez,leob,rdahab}@ic.unicamp.br

Abstract. *The deployment of cryptography on sensor networks is a challenging task, given the limited computational power and resource-constrained nature of the sensoring devices. This paper presents the implementation of binary elliptic curves on the MICAz Mote. Optimization techniques for arithmetic algorithms on binary fields, including squaring, multiplication and modular reduction are presented. Our implementation of field multiplication and modular reduction algorithms focus on the minimization of memory accesses and appear as the most efficient algorithms published for this platform. This leads to an improvement of 39% over the best implementation for computing a point multiplication on a Koblitz curve. The results also show that binary elliptic curves can perform on this platform as well or better than elliptic curves defined over prime fields.*

Resumo. *O emprego de criptografia em redes de sensores é desafiante, dado o baixo poder computacional e os recursos limitados dos dispositivos de sensoriamento. Este trabalho apresenta a implementação de curvas elípticas binárias em dispositivos MICAz. São apresentadas técnicas de otimização para algoritmos de aritmética em corpos binários, incluindo cálculo de quadrado, multiplicação e redução modular. Os algoritmos de multiplicação e redução modular resultantes minimizam a quantidade de acessos à memória e se mostram como os algoritmos mais eficientes para a plataforma escolhida já publicados. A eficiência destes algoritmos culmina em uma multiplicação de ponto com desempenho superior em até 39% à melhor implementação conhecida para corpos binários para o mesmo nível de segurança. Os resultados também mostram que curvas definidas sobre corpos binários podem ter desempenho idêntico ou superior nesta plataforma a curvas definidas sobre corpos primos.*

1. Introdução

Redes de Sensores Sem Fio (RSSFs) [Estrin et al. 1999] são redes *ad hoc* compostas por pequenos sensores de recursos limitados (pouca energia, largura de banda, capacidade computacional etc.) e uma ou mais estações rádio base (ERBs), as quais são mais poderosas e conectam os sensores com o ambiente externo. RSSFs são utilizadas com o objetivo de monitorar regiões, oferecendo dados sobre a área monitorada para o resto do sistema.

Além das vulnerabilidades já existentes em redes *ad hoc*, RSSFs enfrentam problemas adicionais. Elas comumente são dispostas em ambientes fisicamente acessíveis a adversários.

*Financiado por CAPES e FAPESP, processo 2007/06950-0.

Não obstante, nós sensores são mais escassos de recursos que nós de redes *ad hoc* e soluções convencionais não lhes são adequadas. Por exemplo, o fato de que nós sensores devem ser descartáveis e, por conseguinte, de baixo custo torna pouco viável equipá-los com dispositivos contra violação (*tampering*).

O baixo poder computacional dos sensores torna inviável a utilização de algoritmos convencionais de criptografia de chave pública (RSA/DSA, por exemplo) e, até recentemente, primitivas de segurança como sigilo, autenticação e integridade em RSSFs eram alcançadas apenas através de técnicas de criptografia simétrica [Perrig et al. 2002, Karlof et al. 2004]. Atualmente, criptografia de curvas elípticas [Miller 1986, Koblitz 1987] tem sido apontada como uma alternativa promissora aos métodos convencionais de criptografia assimétrica em redes de sensores [Gura et al. 2004], por exigir requisitos menores de processamento e armazenamento para o mesmo nível de segurança. Estas características estimulam a busca de algoritmos cada vez mais eficientes para implementação nestes dispositivos. A plataforma-alvo mais utilizada é o sensor MICAz Mote [Hill and Culler 2002], comum em instalações reais de redes de sensores. Características importantes desta plataforma são a baixa disponibilidade de memória RAM e o alto custo de instruções de acesso à memória.

Este trabalho propõe otimizações para aritmética de curvas elípticas sobre corpos binários, ampliando os seus limites de eficiência e viabilidade de aplicação. Particularmente, os resultados experimentais demonstram que o desempenho de curvas elípticas sobre corpos binários pode equiparar-se ao desempenho de curvas sobre corpos primos em implementações cuidadosas que consideram as características da plataforma. Este resultado contraria a observação de que dispositivos tão escassos em recursos não são suficientemente equipados para implementação de criptografia de curvas elípticas definidas sobre corpos binários [Gura et al. 2004, Eberle et al. 2005].

As contribuições principais deste trabalho consistem em:

- *Implementação eficiente de algoritmos para multiplicação, quadrado e redução modular em $\mathbb{F}_{2^{163}}$:* são apresentadas versões otimizadas de algoritmos conhecidos que minimizam os acessos à memória para ganho de desempenho. As novas otimizações produzem os algoritmos mais eficientes nesta plataforma para aritmética em $\mathbb{F}_{2^{163}}$ já publicados na literatura, sendo consideravelmente mais eficientes do que suas respectivas implementações padrão;
- *Implementação eficiente de criptografia de curvas elípticas:* são implementados algoritmos para calcular a multiplicação de ponto em curvas de Koblitz e curvas binárias aleatórias. O tempo de execução de uma multiplicação de ponto, em curvas de Koblitz, é 39% mais rápido do que a melhor implementação publicada [Seo et al. 2008] e 7% mais rápido do que a melhor implementação em uma curva definida sobre $\mathbb{F}_p$ para o nível de segurança de 160 *bits* [Großschädl 2006]. Para a multiplicação de ponto em uma curva binária aleatória, foi utilizado o algoritmo López-Dahab [López and Dahab 1999a].

O nível de segurança de 160 *bits* foi escolhido por ser compatível com o baixo tempo de vida das informações manipuladas pelos sensores e por oferecer um bom compromisso

entre segurança e desempenho para o ambiente considerado.

O restante deste documento está organizado da seguinte forma. Os trabalhos correlatos são apresentados na Seção 2 e conceitos de curvas elípticas são introduzidos na Seção 3. A Seção 4 investiga implementações eficientes para aritmética em corpos finitos na plataforma-alvo e a Seção 5 apresenta questões de implementação e comparação de resultados. A Seção 6 finaliza o trabalho com as conclusões obtidas.

2. Trabalhos Correlatos

Os estudos voltados para o emprego de criptografia de chave pública em redes de sensores mostram tentativas tanto de adequar algoritmos convencionais (RSA, por exemplo) para nós sensores [Watro et al. 2004], quanto o uso de técnicas mais eficientes (ECC).

[Gura et al. 2004] apresentaram os primeiros resultados sobre ECC e RSA em microcontroladores ATmega128 e mostraram a superioridade de desempenho do primeiro sobre o segundo. A implementação de ECC em C e *Assembly* utilizava corpos primos e executava uma multiplicação de ponto em 0.81 segundo em um dispositivo de 8MHz. Posteriormente, [Uhsadel et al. 2007] apresentaram o tempo esperado de 0.76 segundo em um dispositivo de 7.3828MHz. A implementação mais rápida já publicada para esta plataforma utiliza uma curva com endomorfismo eficiente e calcula uma multiplicação de ponto em 5.5 milhões de ciclos, ou 0.745 segundo [Großschädl 2006].

Para corpos binários, [Malan et al. 2004] implementaram ECC utilizando base polinomial e apresentaram resultados do protocolo Diffie-Hellman. O tempo de geração de uma chave pública, que envolve a multiplicação de um ponto, foi de 34 segundos. [Yan and Shi 2006] implementaram ECC sobre $\mathbb{F}_{2^{163}}$ e obtiveram uma multiplicação de ponto em 13.9 segundos, sugerindo que o custo de aritmética em corpos binários é alto demais para dispositivos tão restritos. [Eberle et al. 2005] implementaram ECC sobre $\mathbb{F}_{2^{163}}$ utilizando *Assembly* e obtiveram uma multiplicação de ponto em 4.14 segundos, recorrendo à introdução de suporte arquitetural para aceleração da implementação. NanoECC [Szczechowiak et al. 2008] optou pela especialização de porções da biblioteca MIRACL [Scott 2008] para execução eficiente em sensores sem fio, resultando em uma multiplicação de ponto em 2.16 segundos para corpos primos e 1.27 segundo para corpos binários, implementadas exclusivamente na linguagem C. Mais recentemente, TinyECCK [Seo et al. 2008] apresentou uma implementação de ECC sobre curvas binárias que considera as características da plataforma para otimizar a aritmética no corpo finito e obteve uma multiplicação de ponto em 1.14 segundo.

A Tabela 1 apresenta o aumento sucessivo de eficiência das implementações de criptografia de curvas elípticas em redes de sensores.

Tabela 1. Tempos publicados para a multiplicação de ponto em um MICAz Mote para o nível de segurança de 160 *bits*.

Corpo	Trabalho	Tempo de execução (s)
Binário	[Malan et al. 2004]	34
	[Yan and Shi 2006]	13.9
	[Eberle et al. 2005]	4.14
	[Szczechowiak et al. 2008]	2.16
	[Seo et al. 2008]	1.14
Primo	[Wang and Li 2006]	1.35
	[Szczechowiak et al. 2008]	1.27
	[Gura et al. 2004]	0.81
	[Uhsadel et al. 2007]	0.76
	[Großschädl 2006]	0.745

3. Criptografia de curvas elípticas

Uma *curva elíptica* E sobre o corpo $\mathbb{F}_{2^m}$ é o conjunto de soluções $(x, y) \in \mathbb{F}_{2^m} \times \mathbb{F}_{2^m}$ que satisfazem a equação

$$y^2 + xy = x^3 + ax^2 + b,$$

onde $a, b \in \mathbb{F}_{2^m}$ com $b \neq 0$, e um *ponto no infinito* denotado por ∞. O número de pontos da curva $E(\mathbb{F}_{2^m})$, denotado por $\#E(\mathbb{F}_{2^m})$, é chamado de *ordem* da curva sobre o corpo $\mathbb{F}_{2^m}$. O conjunto de pontos $\{(x, y) \in E(\mathbb{F}_{2^m})\} \cup \{\infty\}$ sob a operação $+$ (secante e tangente), comutativa e associativa, forma um grupo aditivo. Logo, $(E(\mathbb{F}_{2^m}), +)$ é um grupo abeliano com elemento de identidade ∞. Dado um ponto elíptico $P \in E(\mathbb{F}_{2^m})$ e um número inteiro k, a operação kP, chamada *multiplicação de ponto*, é definida pela relação de recorrência:

$$kP = \begin{cases} \infty, & \text{se } k = 0; \\ (-k)(-P) & \text{se } k \leq -1; \\ (k-1)P + P & \text{se } k \geq 1. \end{cases}$$

A multiplicação de ponto é a operação fundamental utilizada por protocolos baseados em curvas elípticas. Desta forma, é possível fundamentar sua segurança no problema do logaritmo discreto elíptico.

4. Algoritmos

Nesta seção, são considerados algoritmos para as operações aritméticas no corpo binário $\mathbb{F}_{2^m}$, cujos elementos são representados em base polinomial; dado $a \in \mathbb{F}_{2^m}$, $a(z) = \sum_{i=0}^{m-1} a_i z^i$. Em *software*, o elemento a é armazenado como um vetor com $n = \lceil \log_8(m) \rceil$ *bytes*. As operações aritméticas em $\mathbb{F}_{2^m}$ podem ser implementadas usando instruções comuns em processadores convencionais, tais como deslocamentos ($\gg$,$\ll$) e adição módulo 2 (XOR, $\oplus$).

4.1. Quadrado

O quadrado $a(z)^2$ de um elemento finito $a(z) \in \mathbb{F}_{2^m}$ é dado por $a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i} = a_{m-1}z^{2m-2}+\cdots+a_2z^4+a_1z^2+a_0$. A representação binária de $a(z)^2$ pode ser obtida inserindo um *bit* 0 entre *bits* consecutivos da representação binária de $a(z)$ [Hankerson et al. 2003]. Este método pode ser acelerado pela introdução de uma tabela de 16 *bytes*, como mostra o Algoritmo 1.

Algoritmo 1 Quadrado em $\mathbb{F}_{2^m}$.

Entrada: $a(z) = \sum_{i=0}^{m-1} a_i z^i$.
Saída: $c(z) = a(z)^2$.

1: Para cada conjunto u de 4 *bits*, calcular $T(u) = (0, u_3, 0, u_2, 0, u_1, 0, u_0)$.
2: **for** $i \leftarrow 0$ to $n-1$ **do**
3: $\quad c[2i] = T(a[i] \ \ \texttt{\&} \ \ \texttt{0x0F})$
4: $\quad c[2i+1] = T(a[i] \gg 4)$
5: **end for**
6: **return** c

4.2. Multiplicação

A operação de multiplicação de ponto é a operação com maior custo em criptografia de curvas elípticas e depende diretamente da aritmética em $\mathbb{F}_{2^m}$. A operação de soma de pontos em coordenadas projetivas é calculada em termos de multiplicações, quadrados e somas em $\mathbb{F}_{2^m}$. Em particular, a multiplicação em $\mathbb{F}_{2^m}$ é crucial para se obter bom desempenho na multiplicação de ponto.

Duas estratégias distintas são comumente consideradas para a implementação da multiplicação em $\mathbb{F}_{2^m}$. A primeira consiste na utilização do algoritmo de Karatsuba-Ofman em uma primeira instância para dividir a multiplicação em subproblemas e resolver cada subproblema utilizando um algoritmo de multiplicação. A segunda consiste em aplicar um algoritmo convencional para resolução direta do problema, como o algoritmo López-Dahab (LD) de multiplicação em $\mathbb{F}_{2^m}$ [López and Dahab 2000]. Naturalmente, o algoritmo de Karatsuba introduz uma sobrecarga que o inviabiliza para corpos de tamanho pequeno. Observando o fato de que as instruções mais caras na plataforma-alvo concentram-se em instruções de acesso à memória, decidiu-se por analisar o comportamento dos diferentes algoritmos em busca de estimativas de desempenho. Esta análise estima o custo de diferentes algoritmos em termos de acessos à memória (leituras e escritas) e instruções de aritmética (XOR).

O cálculo da multiplicação pelo algoritmo de Karatsuba é dado no Algoritmo 2 [Hankerson et al. 2003].

As multiplicações parciais C_0, C_1 e C_2 de polinômios de grau $m/2$ podem ser calculadas pela multiplicação LD, apresentada no Algoritmo 3. Quando n é ímpar, as multiplicações parciais são realizadas para polinômios de grau $8*\lfloor n/2 \rfloor+4$. O Algoritmo 3 utiliza uma tabela T de pré-computação de tamanho $2^t \times (n)$ para evitar multiplicações repetidas. Desta forma,

Algoritmo 2 Multiplicacão Karatsuba em $\mathbb{F}_{2^m}$.

Entrada: $a(z) = \sum_{i=0}^{m-1} a_i z^i$ e $b(z) = \sum_{i=0}^{m-1} b_i z^i$.
Saída: $c(z) = a(z)b(z)$.

1: Escrever $a(z)$ como $A_1(z)z^{m/2} + A_0(z)$ e $b(z)$ como $B_1(z)z^{m/2} + B_0(z)$
2: $C_0(z) \leftarrow A_0(z)B_0(z)$
3: $C_1(z) \leftarrow A_1(z)B_1(z)$
4: $C_2(z) \leftarrow (A_1 + A_0)(B_1 + B_0)$
5: $c(z) \leftarrow C_1(z)z^m + [C_2(z) + C_1(z) + C_0(z)]z^{m/2} + C_0(z)$
6: **return** c

cada elemento $T[i]$ requer $\lceil n/2 \rceil$ palavras de armazenamento, pois cada produto $u(z)b(z)$ tem grau $8 * \lceil n/2 \rceil$.

Algoritmo 3 Multiplicação López-Dahab em $\mathbb{F}_{2^m}$ [López and Dahab 2000].

Entrada: $a(z) = \sum_{i=0}^{m-1} a_i z^i$ e $b(z) = \sum_{i=0}^{m-1} b_i z^i$.
Saída: $c(z) = a(z)b(z)$.

1: Calcular $T(u) = u(z)b(z)$ para todos os polinômios $u(z)$ com grau até $t - 1$.
2: $c[0 \ldots 2n - 1] \leftarrow 0$
3: **for** $k \leftarrow 0$ to $n - 1$ **do**
4: $\quad u \leftarrow a[k] \gg t$
5: $\quad$ **for** $j \leftarrow 0$ to n **do**
6: $\quad\quad c[j + k] \leftarrow c[j + k] \oplus T(u)[j]$
7: $\quad$ **end for**
8: **end for**
9: $c(z) \leftarrow c(z)z^t$
10: **for** $k \leftarrow 0$ to $n - 1$ **do**
11: $\quad u \leftarrow a[k] \mod 2^t$
12: $\quad$ **for** $j \leftarrow 0$ to n **do**
13: $\quad\quad c[j + k] \leftarrow c[j + k] \oplus T(u)[j]$
14: $\quad$ **end for**
15: **end for**
16: **return** c

Para minimizar o número de operações de acesso à memória, é proposto o método LD com registradores, apresentado no Algoritmo 4. Neste algoritmo, todos os produtos parciais são mantidos em registradores e só são realizados acessos à memória para armazenar resultados definitivos. Desta forma, o número de escritas é reduzido, tanto nos laços desenrolados quanto no deslocamento intermediário. O tamanho da janela de pré-computação é $t = 4$.

Análise dos algoritmos de multiplicação

Sem considerar as multiplicações parciais, a implementação para o corpo $\mathbb{F}_{2^{163}}$ do Algoritmo 2 executa aproximadamente $11n$ leituras, $7n$ escritas e $4n$ instruções XOR.

Algoritmo 4 Otimização proposta para multiplicação em $\mathbb{F}_{2^m}$ com $n+1$ registradores.

Entrada: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Saída: $c(z) = c[0..2n-1]$.
Nota: v_i denota os registradores $(r_{i-1}, \ldots, r_0, r_n, \ldots, r_0, \ldots, r_i)$.

1: Calcular $T(u) = u(z)b(z)$ para todos os polinômios $u(z)$ com grau até 3.
2: Seja u_i os 4 *bits* mais significativos de $a[i]$.
3: $v_0 \leftarrow T(u_0)$, $c[0] \leftarrow r_0$
4: $v_1 \leftarrow v_1 \oplus T(u_1)$, $c[1] \leftarrow r_1$
5: $\cdots$
6: $v_{n-1} \leftarrow v_{n-1} \oplus T(u_{n-1})$, $c[n-1] \leftarrow r_{n-1}$
7: $c \leftarrow (v_{n-1} \,||\, (c[n-2], \ldots, c[0])) \ll 4$
8: Seja u_i os 4 *bits* menos significativos de $a[i]$.
9: $v_0 \leftarrow T(u_0)$, $c[0] \leftarrow c[0] \oplus r_0$
10: $\cdots$
11: $v_{n-1} \leftarrow v_{n-1} \oplus T(u_{n-1})$, $c[n-1] \leftarrow c[n-1] \oplus r_{n-1}$
12: $c[n \ldots 2n-1] \leftarrow c[n \ldots 2n-1] \oplus v_{n-1}$
13: **return** c

Para a multiplicação López-Dahab, a análise das operações executadas pelo algoritmo mostra que a construção da tabela requer n leituras para obter os valores $b[i]$ e $2^t n$ escritas e $11n$ instruções XOR para o preenchimento. Em cada laço interno do algoritmo, são executadas $2(n+1)$ leituras, $n+1$ escritas e $n+1$ instruções XOR. Em cada laço externo, são executadas n leituras para obter os valores $a[k]$ e n interações do laço interno, totalizando $n + 2n(n+1)$ leituras, $n(n+1)$ escritas e $n(n+1)$ instruções XOR. O deslocamento de $c(z)$ efetuado na etapa intermediária requer $2n-1$ leituras e escritas. Considerando a inicialização de c, temos $(3n-1) + 2(n + 2n(n+1))$ leituras, $2^t n + 2(2n-1) + 2n(n+1)$ escritas e $11n + 2n(n+1)$ instruções XOR.

Já para a otimização proposta (Algoritmo 4), a construção da tabela também requer n leituras para obter os valores $b[i]$, $2^t n$ escritas e $11n$ instruções XOR para o preenchimento. A linha 3 do algoritmo executa n leituras da tabela T e 1 escrita em $c[0]$. As linhas 4-6 executam n leituras da tabela, 1 escrita em $c[i]$ e n instruções XOR, por um total de $n-1$ vezes. O deslocamento intermediário exige $n-1$ leituras e $(2n-1)$ escritas. As linhas 9-11 executam n leituras da tabela, 1 leitura e escrita em $c[i]$ e n instruções XOR, por um total de n vezes. A operação final custa n leituras, escritas e instruções XOR. O algoritmo custa, portanto, $2n + n^2 + (n-1) + n(n+1)$ leituras, $2^t n + n + (2n-1) + 2n$ escritas e $11n + n(n-1) + n^2 + n$ instruções XOR.

A Tabela 2 apresenta os custos em operações de memória para os métodos López-Dahab original, Karatsuba+López-Dahab original, López-Dahab com registradores e Karatsuba+López-Dahab com registradores. A Tabela 3 apresenta o custo aproximado dos algoritmos em instruções para o caso $\mathbb{F}_{2^{163}}$, ou $n = 21$.

A alternativa Karatsuba+LD tem praticamente o mesmo custo do algoritmo LD ori-

Tabela 2. Custo em instruções dos algoritmos de multiplicação em $\mathbb{F}_{2^m}$.

	Instruções em função do número de palavras n		
Algoritmo	Leituras	Escritas	XOR
López-Dahab	$4n^2 + 9n - 1$	$2^t n + 2n^2 + 6n - 2$	$2n^2 + 13n$
LD com registradores	$2n^2 + 4n - 1$	$2^t n + 5n - 1$	$2n^2 + 11n$
Karatsuba+Mult. M	$11n + 3\mathrm{M}(\lceil n/2 \rceil)$	$7n + 3\mathrm{M}(\lceil n/2 \rceil)$	$4n + 3\mathrm{M}(\lceil n/2 \rceil)$

Tabela 3. Custo em instruções dos algoritmos de multiplicação em $\mathbb{F}_{2^{163}}$.

	Instruções em função de $n = 21$		
Algoritmo	Leituras	Escritas	XOR
López-Dahab	1952	1342	1155
LD com registradores	965	440	1113
Karatsuba+LD	1977	1593	1239
Karatsuba+LD com registradores	1086	837	1173

ginal, ou seja, o tamanho do corpo n situa-se próximo ao ponto de corte onde aplicar o algoritmo de Karatsuba começa a trazer ganhos de desempenho. O número de operações de acesso à memória para o método LD com registradores (Algoritmo 4) é drasticamente menor em relação ao algoritmo original, reduzindo o número de leituras pela metade e o número de escritas por um fator quadrático. O número de instruções XOR é aproximadamente o mesmo em todos os algoritmos. A comparação entre o Algoritmo 4 e o mesmo algoritmo com a aplicação de Karatsuba no primeiro nível favorece o primeiro pelo baixo número de escritas. A decisão final dependerá da implementação dos dois algoritmos.

4.3. Redução modular

O polinômio irredutível para o corpo $\mathbb{F}_{2^{163}}$, $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$, permite um algoritmo de redução modular rápida que processa uma palavra por vez. O Algoritmo 5 [Seo et al. 2008] é a versão deste algoritmo para processadores de 8 *bits*.

A idéia de armazenar resultados intermediários em registradores também pode ser aplicada ao algoritmo de redução modular, gerando a otimização proposta no Algoritmo 6.

Análise dos algoritmos de redução modular

Como apontado também em [Seo et al. 2008], o Algoritmo 5 executa operações redundantes de acesso à memória. O algoritmo executa 4 leituras e 3 escritas em cada iteração do laço principal e 4 leituras e 3 escritas adicionais na etapa final. No total, são realizadas 88 leituras e 66 escritas. A otimização proposta minimiza o número de operações, executando apenas 42 leituras e 22 escritas. Apesar do Algoritmo 6 ser especializado para o polinômio escolhido, a mesma estratégia pode ser adaptada para corpos binários distintos.

4.4. Inversão

A inversão em $\mathbb{F}_{2^m}$ foi implementada a partir do Algoritmo 7 [Hankerson et al. 2000].

Algoritmo 5 Redução modular rápida para $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$.

Entrada: $a(z) = a[0..2n-1]$.
Saída: $c(z) = a(z) \mod f(z)$.

1: **for** $i \leftarrow 41$ to 21 **do**
2: $\quad t \leftarrow c[i]$
3: $\quad c[i-21] \leftarrow c[i-21] \oplus (t \ll 5)$
4: $\quad c[i-20] \leftarrow c[i-20] \oplus (t \ll 4) \oplus (t \gg 3) \oplus t \oplus (t \gg 3)$
5: $\quad c[i-19] \leftarrow c[i-19] \oplus (t \gg 4) \oplus (t \gg 5)$
6: **end for**
7: $t \leftarrow c[20] \gg 3$
8: $c[0] \leftarrow c[0] \oplus (t \ll 7) \oplus (t \ll 6) \oplus (t \ll 3) \oplus t$
9: $c[1] \leftarrow c[1] \oplus (t \gg 1) \oplus (\gg 2)$
10: $c[20] \leftarrow c[20]$ `&` `0x07`
11: **return** c

4.5. Aritmética na curva elíptica

A operação de multiplicação de ponto é fundamental para a implementação de protocolos baseados em curvas elípticas. A relação de recorrência fornecida na Seção 3 motiva um método binário para multiplicação de ponto [Hankerson et al. 2003]. Para eliminar inversões durante o cálculo da adição de pontos na curva elíptica, coordenadas projetivas são particularmente úteis em curvas binárias [López and Dahab 1999b].

As curvas de Koblitz permitem otimizações. Estas curvas binárias possuem a forma $y^2 + xy = x^3 + 1$ ou $y^2 + xy = x^3 + x^2 + 1$ e fornecem um endomorfismo eficiente calculado a partir do mapa de Frobenius $\tau(x, y) = (x^2, y^2)$ que permite substituir a duplicação de ponto por uma aplicação deste endomorfismo [Hankerson et al. 2003]. O inverso de um ponto $P = (x, y) \in E(\mathbb{F}_{2^m})$ é dado por $-P = (x, x + y)$. Desta forma, é possível acelerar o cálculo de kP convertendo algumas das adições efetuadas no método binário em subtrações. Para isso, representa-se o inteiro k como o somatório $\sum_{i=0}^{t-1} u_i \tau^i$, com $u_i \in \{-1, 0, +1\}$. Uma representação muito utilizada é a *forma não-adjacente de comprimento* w (w-TNAF). A multiplicação de um ponto P por um inteiro k representado na forma w-TNAF é apresentada no Algoritmo 8 [Solinas 2000].

Algoritmo 6 Otimização proposta para redução modular rápida.

Entrada: $a(z) = a[0..2n-1]$.
Saída: $c(z) = a(z) \mod f(z)$.
Nota: A função auxiliar de acumulação $R(r_0, r_1, r_2, t)$ executa:

$s_0 \leftarrow t \ll 4$
$r_0 \leftarrow (r_0 \oplus t \oplus (t \gg 1)) \gg 4$
$r_1 \leftarrow r_1 \oplus t \oplus (t \ll 3) \oplus s_0 \oplus (t \gg 3)$
$r_2 \leftarrow s_0 \ll 1$

1: $r_b \leftarrow 0$
2: $r_c \leftarrow 0$
3: $i \leftarrow 21, j \leftarrow 40$
4: **while** $i > 3$ **do**
5: $R(r_b, r_c, r_a, c[j])$, $c[i] \leftarrow c[i] \oplus r_b$
6: $R(r_c, r_a, r_b, c[j-1])$, $c[i-1] \leftarrow c[i-1] \oplus r_c$
7: $R(r_a, r_b, r_c, c[j-2])$, $c[i-2] \leftarrow c[i-2] \oplus r_a$
8: $i \leftarrow i-3, j \leftarrow j-3$
9: **end while**
10: $R(r_b, r_c, r_a, c[22])$, $c[3] \leftarrow c[3] \oplus r_b$
11: $R(r_c, r_a, r_b, c[21])$, $c[2] \leftarrow c[2] \oplus r_c$
12: $c[1] \leftarrow c[1] \oplus r_a$
13: $c[0] \leftarrow c[0] \oplus r_b$
14: **return** c

Algoritmo 7 Inversão em $\mathbb{F}_{2^m}$ [Hankerson et al. 2000].

Entrada: $a(z) \in \mathbb{F}_{2^m}, a \neq 0$.
Saída: $b(z) = a(z)^{-1} \mod f(z)$.

1: $b \leftarrow 1, c \leftarrow 0, u \leftarrow a, v \leftarrow f$.
2: **while** $u \neq 1$ **do**
3: **while** z divide u **do**
4: $u \leftarrow u/z$
5: **if** z divide b **then** $b \leftarrow b/z$; **else** $b \leftarrow (b+f)/z$
6: **end while**
7: **if** $u = 1$ **then return** b
8: **if** $deg(u) < deg(v)$ **then** $u \leftrightarrow v, b \leftrightarrow c$
9: $u \leftarrow u + v, b \leftarrow b + c$
10: **end while**

Para calcular uma multiplicação de ponto em curvas binárias, foi utilizado o Algoritmo 9 [López and Dahab 1999a]. Este algoritmo não utiliza pré-computação, seu tempo de execução é constante e cada iteração executa o mesmo número de instruções, independente do i-ésimo *bit* de k.

Algoritmo 8 Método w-TNAF para multiplicação de ponto [Solinas 2000].

Entrada: $k \in \mathbb{Z}, P \in E(\mathbb{F}_{2^m})$.
Saída: $kP \in E(\mathbb{F}_{2^m})$.

1: Calcular a representação $TNAF_w(k) = \sum_{i=0}^{t-1} u_i \tau^i$
2: Calcular $P_u = \alpha_u P$, para $u \in \{1, 3, 5, \ldots, 2^{w-1} - 1\}$
3: $Q \leftarrow \infty$
4: **for** $i \leftarrow t - 1$ to 0 **do**
5: $\quad Q \leftarrow \tau Q$
6: $\quad$ **if** $u_i \neq 0$ **then**
7: $\quad\quad$ Seja u_i tal que $\alpha_u = u_i$ ou $\alpha_{-u} = -u_i$
8: $\quad\quad$ **if** $u_i > 0$ **then** $Q \leftarrow Q + P_u$; **else** $Q \leftarrow Q - P_u$
9: $\quad$ **end if**
10: **end for**
11: **return** Q

Algoritmo 9 Método LD para multiplicação de ponto [López and Dahab 1999a].

Entrada: $k = \sum_{i=0}^{t-1} k_i \in \mathbb{Z}, P = (x, y) \in E(\mathbb{F}_{2^m})$, coeficiente b da curva.
Saída: $kP \in E(\mathbb{F}_{2^m})$.

1: $x_1 \leftarrow x$, $z_1 \leftarrow 1$, $z_2 \leftarrow x^2$, $x_2 \leftarrow z_2^2 + b$,
2: **for** $i \leftarrow t - 2$ to 0 **do**
3: $\quad r_1 \leftarrow x_1 z_2, r_2 \leftarrow x_2 z_1, r_3 \leftarrow r_1 + r_2, r_4 \leftarrow r_1 r_2$
4: $\quad$ **if** $k_i \neq 0$ **then**
5: $\quad\quad z_1 \leftarrow r_3^2, r_1 \leftarrow x z_1, x_1 \leftarrow r_1 + r_4, r_1 \leftarrow z_2^2, r_2 \leftarrow x_2^2$
6: $\quad\quad z_2 \leftarrow r_1 r_2, x_2 \leftarrow r_1^2, r_1 \leftarrow r_2^2, r_2 \leftarrow b r_1, x_2 \leftarrow x_2 + r_2$
7: $\quad$ **else**
8: $\quad\quad z_2 \leftarrow r_3^2, r_1 \leftarrow x z_2, x_2 \leftarrow r_1 + r_4, r_1 \leftarrow z_1^2, r_2 \leftarrow x_1^2$
9: $\quad\quad z_1 \leftarrow r_1 r_2, x_1 \leftarrow r_1^2, r_2 \leftarrow r_2^2, r_2 \leftarrow b r_1, x_1 \leftarrow x_1 + r_2$
10: $\quad$ **end if**
11: **end for**
12: **return** $Q = (x_3, y_3)$ a partir de $(x_1/z_1, x_2/z_2)$;

5. Implementação e resultados

O MICAz Mote possui um processador ATmega128 com palavra de 8 *bits*, freqüência de 7.3828 MHz e 4KB de memória RAM. O código de programa é carregado a partir de uma memória ROM de 128 KB. O processador é um RISC típico com 32 registradores, onde instruções de acesso a registradores consomem 1 ciclo de processamento e instruções de leitura/escrita para a memória consomem 2 ciclos de processamento. O *pipeline* do processador tem 2 estágios e instruções de acesso à memória sempre provocam *stalls* no *pipeline* [Hill and Culler 2002].

O compilador e montador utilizado é o GCC 4.1.2 para ATmega128. As tomadas de tempo foram realizadas com o AVR Studio, versão 4.14 [Atmel Corporation 2005]. Esta fer-

ramenta é um simulador com acurácia de ciclo muito utilizado na prototipação de programas para execução na plataforma-alvo. As implementações foram realizadas com modificações da biblioteca MIRACL [Scott 2008], versão 5.32. Esta decisão agiliza o desenvolvimento, mas adiciona sobrecargas decorrentes da complexidade da biblioteca, especialmente no tamanho do código.

Aritmética no corpo finito

Foram implementados os algoritmos de cálculo de quadrado, multiplicação, redução modular e inversão apresentados, nas linguagens C e *Assembly*. A Tabela 4 apresenta os custos em ciclos e tempo absoluto de cada operação implementada. Como a plataforma não tem memória *cache* ou execução fora de ordem, as operações no corpo sempre custam o mesmo número de ciclos e as tomadas de tempo foram efetuadas uma única vez.

Tabela 4. Custo dos algoritmos de aritmética em $\mathbb{F}_{2^{163}}$.

	Linguagem C		*Assembly*	
Algoritmo	Ciclos	Tempo (μs)	Ciclos	Tempo(μs)
Quadrado	725	98	456	62
Mult. LD com registradores	13838	1874	5433	735
Mult. LD (variante otimizada)	9752	1320	9120	1273
Mult. Karatsuba+LD com registradores	12246	1659	6968	943
Redução Modular	621	84	609	83
Inversão	365658	49528	231258	31323

É possível observar que na implementação realizada em linguagem C, a multiplicação Karatsuba+LD com registradores é mais eficiente do que a aplicação direta do método LD com registradores, contrariando a análise preliminar baseada no número de operações de acesso à memória. Isto é explicado pelo fato de que o multiplicador LD com registradores utiliza 21 dos 32 registradores disponíveis apenas para manter os valores intermediários da multiplicação. Vários dos registradores restantes também são necessários para armazenar endereços e servir como variáveis temporárias para operações aritméticas. A ineficiência encontrada é decorrente, portanto, da dificuldade do compilador C em manter todos os valores intermediários em registradores. Para confirmar essa limitação, uma variante do algoritmo que minimiza o número de variáveis temporárias foi também implementada e apresentou o desempenho esperado. As implementações em *Assembly* demonstram a ineficiência do compilador na geração de código otimizado para a plataforma e na alocação de recursos em registradores, sendo bem mais rápidas.

Aritmética na curva elíptica

Foram implementadas a multiplicação de ponto utilizando o método 4-TNAF (Algoritmo 8) com coordenadas projetivas na curva `sect163k1` e a multiplicação de ponto utilizando o método López-Dahab (Algoritmo 9) nas curvas `sect163k1` e `sect163r2` [SECG 2000]. A adição e subtração de pontos na curva `sect163k1` é realizada com coordenadas misturadas [Hankerson et al. 2003]. A Tabela 5 apresenta os custos em ciclos e tempo absoluto

da multiplicação de um ponto aleatório, utilizando a aritmética subjacente em C ou *Assembly*. Em cada uma das linguagens, a multiplicação no corpo finito mais eficiente naquela linguagem é utilizada. Os tempos apresentados foram calculados pela média aritmética de 20 execuções sucessivas do algoritmo.

Tabela 5. Custo da multiplicação de ponto.

	Linguagem C		*Assembly*	
Algoritmo	Ciclos	Tempo (s)	Ciclos	Tempo (s)
4-TNAF na curva `sect163k1`	7022289	0.95	5100400	0.69
LD na curva `sect163k1`	9979042	1.30	6135263	0.83
LD na curva `sect163r2`	11872320	1.60	7244045	0.98

Comparação

A implementação das otimizações propostas foi comparada com TinyECCK, a implementação até então mais rápida para corpos binários publicada na literatura [Seo et al. 2008]. A coluna de ganho refere-se à melhoria de desempenho obtida comparando a versão em C ou *Assembly* do algoritmo proposto com a versão em C presente em TinyECCK. Apesar de vários dos algoritmos implementados em C serem substancialmente mais eficientes do que as contrapartes em TinyECCK, o tempo da multiplicação de ponto em C é apenas 17% mais rápido. Isto se deve às sobrecargas trazidas pela configuração de biblioteca utilizada na presente implementação, que produzem perdas de desempenho. Comparando as versões em *Assembly* com as operações em TinyECCK, pode-se observar ganhos expressivos de desempenho tanto na aritmética do corpo finito quanto na aritmética da curva elíptica, especialmente no algoritmo de multiplicação.

Tabela 6. Comparação entre implementações distintas. Os tempos são fornecidos em quantidades de ciclos (c) ou segundos (s). Os tempos do quadrado e multiplicação para TinyECCK foram obtidos pela subtração dos tempos publicados pelo tempo da redução modular [Seo et al. 2008].

	Proposta				**TinyECCK**
	Linguagem C		*Assembly*		Linguagem C
Algoritmo	Tempo	Ganho	Tempo	Ganho	Tempo
Quadrado	725 c	12%	456 c	45%	825 c
Multiplicação	9752 c	51%	5433 c	72%	19670 c
Redução Modular	621 c	67%	609 c	68%	1904 c
Inversão	365658 c	32%	231258 c	57%	539132 c
4-TNAF `sect163k1`	0.95 s	17%	0.69 s	39%	1.14 s
LD `sect163k1`	1.30 s	-12%	0.83 s	27%	–
LD `sect163r2`	1.60 s	-29%	0.98 s	14%	–

A multiplicação de ponto mais rápida já realizada nesta plataforma para o nível de segurança escolhido custa 0.745 segundo [Großschädl 2006]. Em relação à esta

implementação, que utiliza corpos primos, as otimizações propostas resultam em uma multiplicação de ponto com tempo de execução 7% mais rápido.

As otimizações implementadas provocam tanto ganho de desempenho quanto impacto no consumo de memória. Na Tabela 7, pode-se observar o consumo de memória ROM para armazenamento de programa e de memória RAM para execução das diferentes implementações. Parte do consumo adicional de memória é ocasionado pela decisão de se utilizar uma biblioteca no lugar de uma implementação mais especializada. Em particular, as otimizações em *Assembly* ocasionam uma expansão de código significativa.

Tabela 7. Custo em *bytes* de memória para implementações do cálculo de kP.

	Memória ROM	Memória RAM
Método 4-TNAF - versão C	26668	48
Método 4-TNAF - versão C+*Assembly*	32392	624
Método LD - versão C	10714	16
Método LD - versão C+*Assembly*	16438	528
TinyECCK	5592	618

6. Conclusão

Apesar dos anos de intensa pesquisa, as áreas de segurança e criptografia em RSSFs ainda possuem vários problemas em aberto. Neste trabalho, apresentamos implementações eficientes dos algoritmos de quadrado, multiplicação, redução modular e inversão em corpos finitos que consideram as características da plataforma de sensores sem fio. Em particular, os algoritmos de redução modular e multiplicação são os mais eficientes para aritmética em corpos binários já publicados para esta plataforma. Estas otimizações resultaram em um ganho de 39% de desempenho em relação ao melhor tempo para uma multiplicação de ponto em curva de Koblitz publicado anteriormente. Também foram apresentados tempos de execução para a multiplicação de ponto em curvas binárias aleatórias. Espera-se que estes resultados possam ampliar ainda mais a eficiência e a viabilidade de criptografia de curvas elípticas e que essa ampliação resulte em maior segurança para redes de sensores.

Referências

Atmel Corporation (2005). AVR Studio 4.14. `http://www.atmel.com/`.

Eberle, H., Wander, A., Gura, N., Chang-Shantz, S., and Gupta, V. (2005). Architectural Extensions for Elliptic Curve Cryptography over GF(2^m) on 8-bit Microprocessors. In *Proceedings of ASAP '05*, pages 343–349, Washington, DC, USA. IEEE.

Estrin, D., Govindan, R., Heidemann, J. S., and Kumar, S. (1999). Next century challenges: Scalable coordination in sensor networks. In *Mobile Computing and Networking (MobiCom'99)*, pages 263–270, Seattle, WA USA.

Großschädl, J. (2006). TinySA: a security architecture for wireless sensor networks. In *Proceedings of CoNEXT '06*, New York, NY, USA. ACM.

Gura, N., Patel, A., Wander, A., Eberle, H., and Shantz, S. C. (2004). Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In *Proc. of CHES'04*, pages 119–132.

Hankerson, D., López, J., and Menezes, A. (2000). Software Implementation of Elliptic Curve Cryptography over Binary Fields. In *Proceedings of CHES '00*, pages 1–24. Springer-Verlag.

Hankerson, D., Menezes, A. J., and Vanstone, S. (2003). *Guide to Elliptic Curve Cryptography*. Springer-Verlag, Secaucus, NJ, USA.

Hill, J. L. and Culler, D. E. (2002). MICA: A Wireless Platform for Deeply Embedded Networks. *IEEE Micro*, 22(6):12–24.

Karlof, C., Sastry, N., and Wagner, D. (2004). TinySec: A link layer security architecture for wireless sensor networks. In *2nd ACM SenSys*, pages 162–175.

Koblitz, N. (1987). Elliptic curve cryptosystems. *Mathematics of computation*, 48:203–9.

López, J. and Dahab, R. (2000). High-speed software multiplication in GF(2^m). In *INDOCRYPT '00*, pages 203–212.

López, J. and Dahab, R. (1999a). Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation. In *Proceedings of CHES '99*, pages 316–327, London, UK. Springer-Verlag.

López, J. and Dahab, R. (1999b). Improved algorithms for elliptic curve arithmetic in GF(2^n). In *SAC '98*, pages 201–212, London, UK. Springer-Verlag.

Malan, D. J., Welsh, M., and Smith, M. D. (2004). A public-key infrastructure for key distribution in TinyOS based on elliptic curve cryptography. In *Proceedings of SECON'04*, Santa Clara, California.

Miller, V. (1986). Uses of elliptic curves in cryptography, Advances in Cryptology. In *Crypto'85, Lecture Notes in Computer Science*, volume 218, pages 417–426. Springer.

Perrig, A., Szewczyk, R., Wen, V., Culler, D., and Tygar, J. D. (2002). SPINS: Security protocols for sensor networks. *Wireless Networks*, 8(5):521–534.

Scott, M. (2008). MIRACL – Multiprecision Integer and Rational Arithmetic C/C++ Library. `http://www.shamus.ie/`.

SECG (2000). SEC 2: Recommended Elliptic Curve Domain Parameters. `http://www.secg.org`.

Seo, S. C., Han, D.-G., and Hong, S. (2008). TinyECCK: Efficient Elliptic Curve Cryptography Implementation over GF(2^m) on 8-bit MICAz Mote. Cryptology ePrint Archive, Report 2008/122. `http://eprint.iacr.org/`.

Solinas, J. A. (2000). Efficient Arithmetic on Koblitz Curves. *Designs, Codes and Cryptography*, 19(2-3):195–249.

Szczechowiak, P., Oliveira, L. B., Scott, M., Collier, M., and Dahab, R. (2008). NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In Verdone, R., editor, *EWSN*, volume 4913 of *LNCS*, pages 305–320. Springer.

Uhsadel, L., Poschmann, A., and Paar, C. (2007). Enabling Full-Size Public-Key Algorithms on 8-Bit Sensor Nodes. In *Proceedings of ESAS '07*, pages 73–86.

Wang, H. and Li, Q. (2006). Efficient Implementation of Public Key Cryptosystems on Mote Sensors. In *Proceedings of ICICS'06, LNCS 4307*, pages 519–528, Raleigh, NC.

Watro, R. J., Kong, D., fen Cuti, S., Gardiner, C., Lynn, C., and Kruus, P. (2004). TinyPK: securing sensor networks with public key technology. In *Proceedings of SASN'04*, pages 59–64, Washington, DC.

Yan, H. and Shi, Z. J. (2006). Studying Software Implementations of Elliptic Curve Cryptography. In *Proceedings of ITNG '06*, pages 78–83, Washington, USA. IEEE.