Instituto de Computação
Universidade Estadual de Campinas

Implementação eficiente em *software* de curvas elípticas e emparelhamentos bilineares

Diego de Freitas Aranha

Este exemplar corresponde à redação final da Tese devidamente corrigida e defendida por Diego de Freitas Aranha e aprovada pela Banca Examinadora.

Campinas, 26 de Agosto de 2011.

[illegible]

Prof. Dr. Julio César López Hernández
(Orientador)

Tese apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Doutor em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA POR ANA REGINA MACHADO – CRB8/5467
BIBLIOTECA DO INSTITUTO DE MATEMÁTICA, ESTATÍSTICA E
COMPUTAÇÃO CIENTÍFICA – UNICAMP

Aranha, Diego de Freitas, 1982-
Ar14i Implementação eficiente em software de curvas elípticas e emparelhamentos bilineares / Diego de Freitas Aranha. - Campinas, SP: [s.n.], 2011.

Orientador: Júlio César Lopez Hernández.
Tese (doutorado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Curvas elípticas. 2. Emparelhamentos bilineares. 3. Criptografia de chave-pública. 4. Algoritmos paralelos. 5. Aritmética de computador. I. Lopez Hernández, Júlio César, 1961-. II. Universidade Estadual de Campinas, Instituto de Computação. III. Título.

Informações para Biblioteca Digital

Título em Inglês: Efficient software implementation of elliptic curves and bilinear pairings
Palavras-chave em Inglês:
Elliptic curves
Bilinear pairings
Public key cryptography
Parallel algorithms
Computer arithmetic
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Júlio César Lopez Hernández [Orientador]
Paulo Sérgio Licciardi Messeder Barreto
Anderson Clayton Alves Nascimento
Ricardo Dahab
Marco Aurélio Amaral Henriques
Data da defesa: 26-08-2011
Programa de Pós-Graduação: Ciência da Computação

TERMO DE APROVAÇÃO

Tese Defendida e Aprovada em 26 de agosto de 2011, pela Banca examinadora composta pelos Professores Doutores:

Anderson C. A. Nascimento

Prof. Dr. Anderson Clayton Alves Nascimento
ENE / UnB

[illegible]

Prof. Dr. Paulo Sérgio Licciardi Messeder Barreto
LARC / USP

[illegible]

Prof. Dr. Marco Aurélio Amaral Henriques
FEEC / UNICAMP

[illegible]

Prof. Dr. Ricardo Dahab
IC / UNICAMP

[illegible]

Prof. Dr. Julio César López Hernández
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Implementação eficiente em *software* de curvas elípticas e emparelhamentos bilineares

Diego de Freitas Aranha[1]

26 de Agosto de 2011

[1]Financiado por Bolsa CAPES em 2007, Bolsa FAPESP (processo 2007/06950-0) em 2008–2010, Bolsa Sanduíche CAPES PDEE (processo 5551-09-9) em 2010–2011.

Resumo

O advento da criptografia assimétrica ou de chave pública possibilitou a aplicação de criptografia em novos cenários, como assinaturas digitais e comércio eletrônico, tornando-a componente vital para o fornecimento de confidencialidade e autenticação em meios de comunicação. Dentre os métodos mais eficientes de criptografia assimétrica, a criptografia de curvas elípticas destaca-se pelos baixos requisitos de armazenamento para chaves e custo computacional para execução. A descoberta relativamente recente da criptografia baseada em emparelhamentos bilineares sobre curvas elípticas permitiu ainda sua flexibilização e a construção de sistemas criptográficos com propriedades inovadoras, como sistemas baseados em identidades e suas variantes. Porém, o custo computacional de criptossistemas baseados em emparelhamentos ainda permanece significativamente maior do que os assimétricos tradicionais, representando um obstáculo para sua adoção, especialmente em dispositivos com recursos limitados.

As contribuições deste trabalho objetivam aprimorar o desempenho de criptossistemas baseados em curvas elípticas e emparelhamentos bilineares e consistem em: (i) implementação eficiente de corpos binários em arquiteturas embutidas de 8 *bits* (microcontroladores presentes em sensores sem fio); (ii) formulação eficiente de aritmética em corpos binários para conjuntos vetoriais de arquiteturas de 64 *bits* e famílias mais recentes de processadores *desktop* dotadas de suporte nativo à multiplicação em corpos binários; (iii) técnicas para implementação serial e paralela de curvas elípticas binárias e emparelhamentos bilineares simétricos e assimétricos definidos sobre corpos primos ou binários. Estas contribuições permitiram obter significativos ganhos de desempenho e, conseqüentemente, uma série de recordes de velocidade para o cálculo de diversos algoritmos criptográficos relevantes em arquiteturas modernas que vão de sistemas embarcados de 8 *bits* a processadores com 8 *cores*.

Abstract

The development of asymmetric or public key cryptography made possible new applications of cryptography such as digital signatures and electronic commerce. Cryptography is now a vital component for providing confidentiality and authentication in communication infra-structures. Elliptic Curve Cryptography is among the most efficient public-key methods because of its low storage and computational requirements. The relatively recent advent of Pairing-Based Cryptography allowed the further construction of flexible and innovative cryptographic solutions like Identity-Based Cryptography and variants. However, the computational cost of pairing-based cryptosystems remains significantly higher than traditional public key cryptosystems and thus an important obstacle for adoption, specially in resource-constrained devices.

The main contributions of this work aim to improve the performance of curve-based cryptosystems, consisting of: (i) efficient implementation of binary fields in 8-bit microcontrollers embedded in sensor network nodes; (ii) efficient formulation of binary field arithmetic in terms of vector instructions present in 64-bit architectures, and on the recently-introduced native support for binary field multiplication in the latest Intel microarchitecture families; (iii) techniques for serial and parallel implementation of binary elliptic curves and symmetric and asymmetric pairings defined over prime and binary fields. These contributions produced important performance improvements and, consequently, several speed records for computing relevant cryptographic algorithms in modern computer architectures ranging from embedded 8-bit microcontrollers to 8-core processors.

Agradecimentos

Gostaria primeiramente de agradecer à minha noiva Fernanda Alcântara Andaló, que mesmo compartilhando das dificuldades da vida de um doutorando, sempre encontrou tempo para me escutar, compreender e transmitir carinho. Sua dedicação ao nosso projeto conjunto de vida me emociona e espero que estejamos próximos de forma definitiva muito em breve. Agradeço também aos meus bichos de estimação Jaiminho, Peta e Bruno, que souberam providenciar momentos de distração mesmo sem ter muita consciência do quanto isso era às vezes necessário.

Meu muito obrigado aos meus pais, José, Maria José, e irmãos, Pablo e Rodrigo, pelo suporte contínuo e pelo enorme esforço em percorrer um grande trajeto para assistir à minha defesa. Agradeço também à família Andaló, de minha noiva, pelos votos constantes de sucesso e à família Wagner pelas refeições esporádicas porém deliciosas que me ajudaram a atravessar o inverno canadense com saúde e disposição.

Agradeço a todos os membros da banca, pois é uma grande honra receber suas contribuições; aos meus co-autores, em especial Patrick Longa, Koray Karabina, Francisco Henríquez-Rodríguez, Darrel Hankerson, Conrado P. L. Gouvêa, Leonardo B. Oliveira e Jérémie Detrey por sua paciência e generosidade; aos colegas do Laboratório de Criptografia Aplicada da UNICAMP pelas discussões pertinentes; aos colegas do *Center for Applied Cryptographic Research*, em especial Márcio Juliato, e companheiros de escritório da Universidade de Waterloo pelo acolhimento hospitaleiro.

Um agradecimento especial ao Prof. Julio López, um grande orientador que não só me ajudou a percorrer os caminhos necessários para tornar esta tese possível, como soube me transmitir de forma eficiente sua paixão por aritmética criptográfica. Agradeço também ao Prof. Ricardo Dahab, sempre perspicaz em seus inúmeros conselhos, e ao Prof. Alfred Menezes, pelo exemplo incrível de competência e profissionalismo.

Fico particularmente feliz em agradecer aos meus amigos pela companhia e pela sabedoria em desviar estrategicamente o assunto de nossas conversas sempre que eu transparecia demasiada dedicação ao meu tema de pesquisa.

Registro meu reconhecimento à FAPESP, CNPq e CAPES pelo auxílio financeiro e aos demais que, de alguma maneira, me ajudaram na conclusão deste trabalho: docentes, funcionários e colegas do Instituto de Computação da UNICAMP.

Sumário

Capítulo 1

Introdução

A descoberta da criptografia de chave pública [4] revolucionou a forma de se construir sistemas criptográficos e possibilitou, de forma definitiva, a integração entre teoria criptográfica e implementação em aplicações reais. Particularmente, trouxe a possibilidade de se estabelecer serviços criptográficos como sigilo e assinatura irretratável em ambientes onde não existe qualquer relação de confiança entre os envolvidos ou canal seguro para distribuição de chaves. O antigo problema da distribuição de chaves converteu-se então na dificuldade de obtenção de uma chave pública autêntica. Como solução para este novo problema, um repositório público foi inicialmente proposto como ponto de distribuição de chaves [4]. Entretanto, não há como um repositório deste tipo fornecer autenticidade, pois possibilita que atacantes participem em protocolos personificando entidades legítimas. A autenticação mútua das chaves é crucial para que tais intervenções possam ser detectadas e evitadas.

O surgimento de infra-estruturas de chaves públicas [5] solucionou o problema de tituaridade de chaves públicas e impulsionou o comércio eletrônico. Por outro lado, criou diversos problemas adicionais. Em primeiro lugar, infra-estruturas de chaves públicas são concebidas para representar entidades do mundo real, suas filiações a organizações e serviços que fornecem. Os tipos de relações presentes são diversos, o que traz sérios problemas para o projeto de infra-estruturas que sejam tanto genéricas o suficiente para representar qualquer tipo de relação, quanto simples o suficiente para terem ampla aceitação, possibilidade de padronização e eficiência. Operações de validação de certificados e revogação de chaves públicas tendem a representar situações extremas [6].

A descoberta de sistemas criptográficos baseados no problema do logaritmo discreto em curvas elípticas [7, 8] produziu uma nova revolução na área. Ao apresentarem desempenho superior e exigirem chaves mais curtas para um mesmo nível de segurança que os métodos tradicionais de criptografia assimétrica, especialmente o algoritmo RSA [9] e variantes baseadas no problema da fatoração, alguns dos problemas inerentes às infra-estruturas de chaves públicas foram minimizados. Contudo, a dificuldade de gerência e a sobrecarga de desempenho decorrentes da utilização de certificados ainda impõe obstáculos na adoção de criptografia assimétrica em ambientes restritos, como computação móvel e em dispositivos de baixo poder computacional [10].

A busca de alternativas ao paradigma tradicional de infra-estruturas de chave pública resultou na descoberta de sistemas baseados em identidade. Foram concebidos inicialmente por Shamir [11], em 1984, para assinaturas digitais, mas as primeiras realizações funcionais e eficientes para cifração só foram apresentadas em 2001, por Boneh e Franklin [12], e Sakai, Ohgishi e Kasahara [13], a partir de emparelhamentos bilineares sobre curvas elípticas. A motivação original para sistemas baseados em identidade era aproveitar a autenticidade de informação publicamente conhecida para simplificar a autenticação de chaves públicas. Até então, a única aplicação de emparelhamentos bilineares em criptografia era atacar sistemas criptográficos de curvas elípticas [14, 15]. Após a aplicação de emparelhamentos para concretizar cifração baseada em identidade, uma gama de novos protocolos com propriedades inovadoras e especiais foi desenvolvida. Isto levou a uma flexibilização enorme das primitivas criptográficas conhecidas e ampliou os cenários de aplicação de criptografia assimétrica de forma considerável. Entre os protocolos baseados em problemas sobre grupos bilineares, destacam-se: acordo de chaves eficientes para múltiplas entidades [16], assinaturas curtas e agregadas [17] e paradigmas alternativos de certificação implícita [18].

1.1 Justificativa e objetivo

Apesar das propriedades inovadoras, o desempenho de sistemas criptográficos baseados em emparelhamentos ainda representa um obstáculo. Tipicamente, o cálculo de um emparelhamento bilinear ainda é comparável a uma decifração/assinatura RSA [19], e uma ordem de magnitude menos eficiente que uma multiplicação de ponto em curvas elípticas [20, 21]. Isto é natural, visto que os métodos mais estabelecidos de criptografia assimétrica puderam receber maior esforço de pesquisa, produzindo algoritmos cada vez mais eficientes. Esforço similar já é consistente em criptografia baseada em emparelhamentos [22], resultando em novos algoritmos [23, 1, 3, 24] e novas curvas adequadas à sua instanciação [25].

Este projeto teve como finalidade desenvolver algoritmos eficientes (seqüenciais e paralelos) e implementações em *software* otimizadas para criptografia de curvas elípticas e criptografia baseada em emparelhamentos. Vários níveis de aritmética são empregados: [19]: algoritmos para o cálculo do emparelhamento $e(P, Q)$ propriamente dito [23, 1], aritmética na curva elíptica onde o emparelhamento bilinear encontra-se definido [26], aritmética no corpo finito onde a curva elíptica está definida [27] e aritmética nas extensões

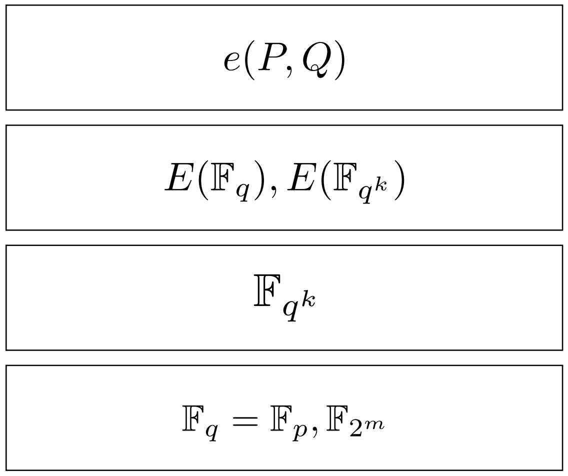

$e(P, Q)$
$E(\mathbb{F}_q), E(\mathbb{F}_{q^k})$
$\mathbb{F}_{q^k}$
$\mathbb{F}_q = \mathbb{F}_p, \mathbb{F}_{2^m}$

Figura 1.1: Níveis de aritmética envolvidos no cálculo de um emparelhamento bilinear.

deste corpo finito [28]. Estes níveis de aritmética encontram-se visualmente representados na Figura 1.1 em uma organização descendente partindo do cálculo do emparelhamento.

O objetivo principal deste trabalho consistiu em tornar estes métodos de criptografia mais eficientes nas arquiteturas modernas, abrangendo tanto pesquisa algorítmica quanto pesquisa aplicada de implementação. A implementação exigiu o projeto de técnicas de otimização de algoritmos em arquiteturas modernas (embarcadas, multiprocessadas) e concretizou os algoritmos em código funcional eficiente, fazendo o melhor uso possível dos recursos disponibilizados pelo *hardware*. Para isso, paralelismo em nível de tarefas e em nível de dados foram extensamente utilizados, incluindo a aplicação de multiprocessamento e conjuntos de instruções vetoriais.

1.2 Metodologia

Para atender os objetivos previamente mencionados, a metodologia utilizada compreendeu os seguintes passos:

1. Levantamento ferramental: pesquisar e experimentar com *frameworks*, linguagens e ferramentas para programação em arquiteturas multiprocessadas, incluindo compiladores, bibliotecas, simuladores e *profilers*.

   Como ferramenta principal para programação paralela em arquiteturas homogêneas, a tecnologia *OpenMP* [29] foi selecionada. O suporte à tecnologia encontra-se incluído em dois compiladores distintos, o *GNU Compiler Collection*, versão 4.2.0 e o *Intel C++ Compiler*, versão 10.01. O *framework* OpenMP suporta construções tanto para paralelismo de dados como para paralelismo de tarefas. A implementação dos algoritmos foi realizada principalmente na linguagem *C* e as rotinas críticas foram codificadas em *Assembly*.

2. Análise e seleção de arquiteturas: selecionar arquiteturas modernas relevantes e analisar características das arquiteturas selecionadas que possam posteriormente ser utilizadas para otimização das implementações dos algoritmos desenvolvidos.

   Dentre as opções de arquitetura multiprocessadas, destacam-se as arquiteturas Intel *Core* [30] e AMD K10 [31], que oferecem alto desempenho e as melhores ferramentas de desenvolvimento disponíveis. Para ambientes embutidos e computação móvel, destacam-se as arquiteturas *AVR ATmega128* [32] e *Intel XScale* [33]. Não houve tempo suficiente para se explorar o processador *Cell* [34] ou processamento em placas gráficas [35], apesar dos mesmos serem importantes no segmento de computação de alto desempenho.

3. Levantamento bibliográfico: determinar precisamente os componentes da aritmética em curvas elípticas e para o cálculo de emparelhamentos bilineares que são mais exigidos. Em busca de aperfeiçoamento das alternativas já propostas, pesquisar artigos que propõem algoritmos para criptografia de curvas elípticas e baseada em emparelhamentos. Pesquisar algoritmos relevantes que ainda não foram adaptados para ambientes paralelos.

4. Desenvolvimento de novos algoritmos: propor otimizações, variantes paralelas de algoritmos existentes ou novos algoritmos para criptografia de curvas elípticas e criptografia baseada em emparelhamentos bilineares, abrangendo os níveis de aritmética citados anteriormente.

5. Implementação eficiente dos algoritmos: a implementação em *software* de cada algoritmos desenvolvido utilizou a metodologia abaixo.

   (a) Detecção de gargalos: detectar porções do algoritmo que correspondem ao maior custo computacional;

   (b) Otimização: elaborar técnicas para otimizar as implementações dos algoritmos, considerando principalmente os gargalos de desempenho detectados;

   (c) Transformações algébricas: investigar os fundamentos matemáticos do algoritmo em busca de modificações que acelerem o tempo de execução de sua implementação ou permitam a extração de mais paralelismo;

   (d) Implementação: concretizar os algoritmos em implementações de alto desempenho que utilizem os recursos do processador com máxima eficiência.

6. Validação: a partir da análise de complexidade do algoritmo e de sua implementação concreta, observar e analisar os resultados em termos de tempo de execução, eficiência na utilização de recursos e escalabilidade. Demonstrar a correção das otimizações propostas também é desejável.

7. Construção de uma biblioteca em *software* contendo os algoritmos implementados. Esta biblioteca é descrita brevemente na seção 1.5.3.

1.3 Arquiteturas computacionais modernas

Tradicionalmente, cada nova geração de processadores de propósito geral obtém ganhos de desempenho significativos de duas formas: aprimoramentos no processo de fabricação e mudanças arquiteturais. Estas últimas são comumente relacionadas à extração de *paralelismo em nível de instruções*, ou seja, à execução concorrente de instruções que não possuem dependências de dados. Entretanto, estas otimizações atualmente se deparam com limitações críticas. A extração de paralelismo em nível de instrução claramente apresenta um limite superior, e muitas aplicações possuem um alto grau de dependência de dados que restringe este paralelismo [36]. O aperfeiçoamento do processo de fabricação também atinge limitações físicas, já que componentes cada vez menores dissipam potência em uma área cada vez menor [37]. Como obstáculo adicional, o poder de processamento vem crescendo muito mais do que a velocidade da memória, e o acesso à memória já é reconhecido como o maior gargalo de execução nas arquiteturas atuais [38]. Estas três limitações provocaram uma mudança radical no projeto de arquiteturas modernas, transportando a ênfase antes colocada em mecanismos para extração automática de paralelismo para mecanismos explícitos, na forma de multiprocessamento e vetorização. A disponibilidade crescente de componentes em áreas cada vez menores tem simultaneamente permitido o desenvolvimento de sistemas embarcados com poder computacional crescente.

1.3.1 Multiprocessamento

Arquiteturas multiprocessadas de propósito geral são chamadas *multi-core*. Em uma arquitetura *multi-core*, diversas unidades de processamento independentes conectam-se ao sistema de memória, compartilhando opcionalmente recursos do *hardware* [39]. Apesar de parecer uma forma barata e escalável de se aumentar desempenho, a introdução de *paralelismo em nível de thread* impõe uma mudança de paradigma de programação [40].

Enquanto nas máquinas uniprocessadas e seqüenciais, o desempenho dos programas crescia automaticamente a cada nova geração de processadores, em arquiteturas *multi-core*, o desempenho dos programas está diretamente relacionado ao grau de paralelismo em nível de *thread* que o programa apresenta. Como esse paralelismo é extraído explicitamente e requer análise profunda do problema e solução sendo tratados, e o modelo de execução em *multithreading* é não-determinístico, paralelizar algoritmos é uma tarefa não-trivial [41].

A exploração com sucesso do paralelismo também requer conhecimento da organização das unidades de processamento e da hierarquia de memória [42]. Quanto à organização das unidades de processamento, pode-se classificar as arquiteturas multiprocessadas em:

- *Homogêneas:* as unidades de processamento são idênticas e contam com os mesmos recursos. A extração de paralelismo torna-se mas simples, mas a divisão do problema deve produzir tarefas com as mesmas características;

- *Heterogêneas:* as unidades de processamento são especializadas para execução de tarefas com naturezas distintas. Permite-se tipicamente um maior controle do fluxo de execução mas pode-se introduzir dificuldade na exploração de paralelismo.

Já quanto à organização da hierarquia de memória, há a divisão em duas vertentes:

- *Acesso uniforme:* níveis inferiores de memória *cache* são compartilhados e o acesso à memória é uniforme a partir de um barramento único. Esta abordagem pressiona os níveis inferiores de memória *cache*, pois a ocupação é disputada por unidades de processamento distintas.

- *Acesso não-uniforme:* cada unidade ou subconjunto de unidades de processamento conta com memórias dedicadas de rápido acesso e latências variadas a porções distintas da memória global. Esta abordagem promete escalabilidade da hierarquia de memória, mas exige afinidade de tarefas por unidade de processamento;

Estas organizações definem o espectro de abordagens utilizadas no projeto de arquiteturas multiprocessadas modernas. A arquitetura *Core* [30] da Intel é composta por unidades de processamento idênticas e acesso uniforme à memória. A arquitetura Opteron [43] utiliza unidades de processamento também idênticas, mas acesso não-uniforme à memória. A arquitetura *UltraSparc T1* [44] da Sun emprega 8 unidades homogêneas e acesso uniforme, mas cada unidade é superescalar e capaz de executar 4 *threads* distintas, totalizando 32 *threads* simultâneas. O processador *Cell* [34] da IBM é um exemplo típico de arquitetura heterogênea que dispõe de um processador de propósito geral e 8 unidades de processamento adicionais especializadas em aritmética de alto desempenho. Esta especialização naturalmente impõe acesso não-uniforme à memória.

Os mesmos princípios regem o projeto de arquiteturas multiprocessadas no segmento de computação embutida. A heterogeneidade muitas vezes manifesta-se na presença de co-processadores [45, 33] e o acesso não-uniforme à memória é empregado a partir de *memórias de rascunho locais* [46] ou transferências explícitas de dados por um canal de comunicação dedicado [47].

1.3.2 Vetorização

As arquiteturas modernas também contam com instruções especializadas para processamento de múltiplos objetos de dados simultaneamente. Essas instruções são classificadas como SIMD (*Simple Instruction - Multiple Data*) [48], e são extremamente úteis na otimização de programas com alta densidade aritmética. Instruções vetoriais são outro recurso para exploração de paralelismo de dados, mas com uma granularidade muito mais fina do que o multiprocessamento. Conjuntos de instruções vetoriais bastante difundidos e presentes em arquiteturas modernas são MMX [49], SSE [50, 51] e AVX [52]. Há um incentivo ainda maior para a utilização de instruções dessa natureza, visto que a latência de execução dessas instruções tem diminuído a cada nova geração de processadores.

1.3.3 Sistemas embarcados

A disponibilidade farta de transístores têm também provocado uma expansão das funcionalidades possíveis em sistemas dedicados de pequeno porte. Esta tendência tecnológica é geralmente descrita através do termo *Internet of Things* [53], que captura precisamente a noção de um conjunto de inúmeros dispositivos distintos, cada um com sua função, interligados por um substrato compartilhado de comunicação. Sistemas embarcados ocupam os segmentos de menor custo do espectro de arquiteturas modernas e muitas vezes executam funções importantes que carecem naturalmente de serviços de segurança como autenticação e confidencialidade. Exemplos diretos de sistemas embarcados que desempenham funcionalidade crítica são nós de uma rede sensores [54] e etiquetas *RFID*(*Radio-Frequency IDentification*) [55].

1.3.4 Métricas de desempenho

A métrica tradicional para comparar o desempenho de diferentes implementações é a contagem direta de ciclos, desde que as implementações sejam executadas em plataformas representantes da mesma microarquitetura. A contagem de ciclos é tipicamente realizada através de um simulador de arquiteturas embarcadas ou pela leitura de registradores especiais nas arquiteturas de maior porte. Cabe a ressalva de que, dada a complexidade dos processadores atuais, envolvendo múltiplos componentes funcionando em paralelo a freqüências oscilantes de processamento, registrar a passagem de cada ciclo torna-se um problema complexo. A alternativa encontrada recentemente pelos fabricantes para manter razoavelmente a noção convencional de ciclo foi estabelecer o ciclo como uma medida de tempo real correspondente ao inverso da freqüência nominal do processador [56]. Há, portanto, que se ter o cuidado de realizar tomadas de tempo em ciclos com o processador ocioso e funcionando em sua freqüência nominal. Este procedimento simples foi utilizado para obtenções de resultados experimentais na maioria dos trabalhos apresentados nesta tese, com exceção da microarquitetura Intel *Westmere*. Esta microarquitetura possui suporte autônomo à gerência de freqüência do processador, podendo a freqüência inclusive superar a freqüência nominal (*overclocking*). Esta funcionalidade depende de diversos efeitos, como temperatura e a quantidade de trabalho distribuído entre as várias unidades de processamento, tornando difícil a determinação precisa da freqüência atual de execução e potencialmente distorcendo as tomadas de tempo. Foi possível contornar este obstáculo apenas a partir do desligamento desta funcionalidade na configuração das máquinas utilizadas para tomadas de tempo.

1.4 Fundamentação matemática

Nesta seção, apresentamos as áreas de criptografia de curvas elípticas e baseada em emparelhamentos bilineares, a fundamentação matemática para o Algoritmo de Miller [57, 58], empregado no cálculo de emparelhamentos, e a aplicação deste algoritmo para o cálculo de emparelhamentos bilineares propostos na literatura.

1.4.1 Curvas elípticas

Uma *curva elíptica* sobre um corpo $\mathbb{K}$ é definida pela *equação de Weierstraß*:

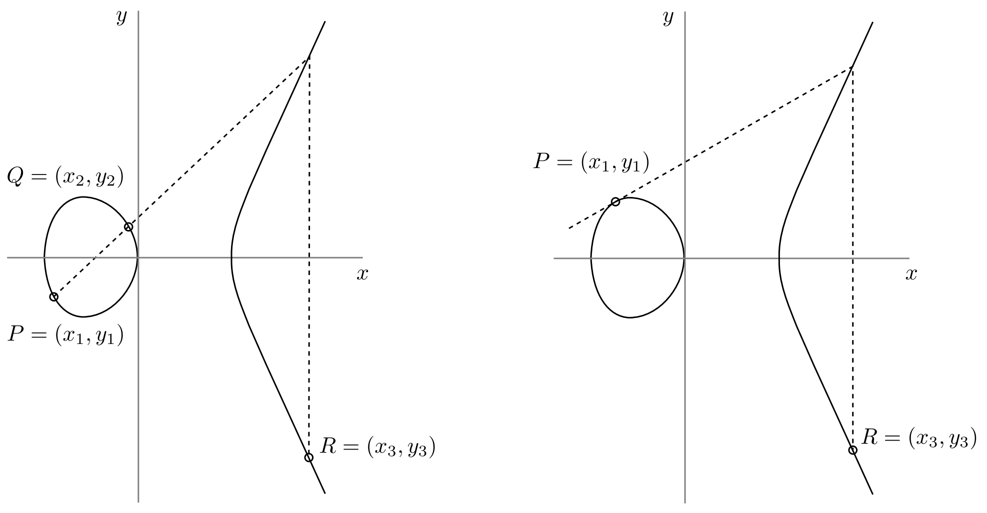
$$E : y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6, \tag{1.1}$$

onde $a_1, a_2, a_3, a_4, a_5, a_6 \in \mathbb{K}$, com *discriminante* $\Delta \neq 0$. Se $\bar{\mathbb{K}}$ é o fecho algébrico de $\mathbb{K}$ e $\mathbb{L}$ é uma extensão algébrica $\mathbb{K} \subset \mathbb{L} \subset \bar{\mathbb{K}}$, o conjunto de *pontos* $\mathbb{L}$*-racionais* em E é:

$$E(\mathbb{L}) = \{(x, y) \in \mathbb{L} \times \mathbb{L} : y^2 + a_1xy + a_3y - x^3 - a_2x^2 - a_4x - a_6 = 0\} \cup \{\infty\},$$

onde ∞ é o ponto no infinito.

Curvas elípticas fornecem uma regra geométrica para a adição de pontos. Sejam $P = (x_1, y_1)$ e $Q = (x_2, y_2)$ pontos distintos racionais sobre uma curva elíptica E. A soma $R = P + Q$ é definida como a reflexão sobre o eixo x do ponto R' de intersecção entre a



(a) Adição de pontos $R = P + Q$; (b) Duplicação de ponto $R = 2P$;

Figura 1.2: Adição e duplicação geométrica de pontos em curvas elípticas.

curva E e a linha que passa por P e Q. Quando $P = Q$, toma-se a intersecção R' entre a curva E e a tangente à curva E no ponto P. A figura 1.2 apresenta as regras para adição e duplicação de pontos em sua forma geométrica. O conjunto de pontos $E(\mathbb{K})$ em conjunção com a regra de adição forma o grupo abeliano $(E(\mathbb{K}), +)$ com o ponto no infinito ∞ como elemento de identidade.

O número de pontos da curva $E(\mathbb{K})$, denotado por $\#E(\mathbb{K})$, é chamado de *ordem* da curva sobre o corpo $\mathbb{K}$. A *condição de Hasse* afirma que $\#E(\mathbb{F}_{q^k}) = q^k + 1 - t$, onde $|t| \leq 2\sqrt{q^k}$ é chamado de *traço de Frobenius*. Curvas em que a característica q divide t são chamadas de *curvas supersingulares*.

A partir de substituição de variáveis, a equação (1.1) pode ser simplificada dependendo da característica q do corpo $\mathbb{K}$ [59]:

- Se $q \neq 2, 3$ primo, a curva elíptica é dita *curva prima* e tem equação:

$$E(\mathbb{F}_q) : y^2 = x^3 + ax + b; \tag{1.2}$$

- Se $q = 2$ e $a_1 \neq 0$, a curva elíptica é dita *curva binária ordinária* e tem equação:

$$E(\mathbb{F}_{2^m}) : y^2 + xy = x^3 + ax^2 + b; \tag{1.3}$$

- Se $q = 2$ e $a_1 = 0$, a curva elíptica é dita *curva binária supersingular* e tem equação:

$$E(\mathbb{F}_{2^m}) : y^2 + cy = x^3 + ax + b. \tag{1.4}$$

Dado um ponto elíptico $P \in E(\mathbb{F}_q)$ e um número inteiro l, a operação lP, chamada *multiplicação de ponto por escalar*, é definida pela relação de recorrência:

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

A multiplicação de ponto é a operação fundamental utilizada por protocolos baseados em curvas elípticas. Este operação pode ser calculada em diferentes sistemas de coordenadas [60, 61, 21] e com diferentes algoritmos, dependendo se há ou não conhecimento prévio do ponto a ser multiplicado [62, 63, 64, 65]. Os Capítulos 2, 3 e 4 discutem brevemente a escolha de algoritmos para multiplicação de ponto. Informações detalhadas para estes algoritmos podem ser encontradas em [59, 66].

Seja $n = \#E(\mathbb{F}_{q^k})$. A *ordem* de um ponto $P \in E$ é o menor inteiro $r > 0$ tal que $rP = \infty$. Temos que $r|n$. O *conjunto de pontos de torção* r de E, denotado por $E(\mathbb{K})[r]$, é o conjunto de pontos cuja ordem divide r, ou seja, o conjunto $\{P \in E(\mathbb{K}) | rP = \infty\}$. Destas definições, segue que $\langle P \rangle$, o grupo de pontos gerado por P, é um subgrupo de $E(\mathbb{K})[r]$, que por sua vez é um subgrupo de $E(\mathbb{K})[n]$. Dizemos que o subgrupo $\langle P \rangle$ tem *grau de mergulho* k se k é o menor inteiro tal que $r|q^k - 1$ [23].

1.4.2 Emparelhamentos bilineares

Sejam $\mathbb{G}_1$ e $\mathbb{G}_2$ grupos cíclicos aditivos e $\mathbb{G}_T$ um grupo cíclico multiplicativo tais que $|\mathbb{G}_1| = |\mathbb{G}_2| = |\mathbb{G}_T|$. Seja P o gerador de $\mathbb{G}_1$ e Q o gerador de $\mathbb{G}_2$. Um mapeamento $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ é dito um *emparelhamento bilinear admissível* [12] se satisfaz as seguintes propriedades:

1. *Bilinearidade:* dados $(V, W) \in \mathbb{G}_1 \times \mathbb{G}_2$, temos

$$e(P, Q + Z) = e(P, Q) \cdot e(P, Z) \text{ e } e(P + V, Q) = e(P, Q) \cdot e(V, Q).$$

   Conseqüentemente, para quaisquer $a, b \in \mathbb{Z}_{|\mathbb{G}_T|}$, temos:

$$e(aV, bW) = e(V, W)^{ab} = e(abV, W) = e(V, abW) = e(bV, aW).$$

2. *Não-degeneração:* $e(P, Q) \neq 1_{\mathbb{G}_T}$, onde $1_{\mathbb{G}_T}$ é a identidade do grupo $\mathbb{G}_T$.

3. *Eficiência:* O mapeamento e pode ser calculado eficientemente, ou seja, tem complexidade polinomial.

Tipicamente, $\mathbb{G}_1$ e $\mathbb{G}_2$ são subgrupos do grupo de pontos de uma curva elíptica sobre o fecho algébrico de um corpo finito $\mathbb{F}_q$ e $\mathbb{G}_T$ é um subgrupo do grupo multiplicativo de um corpo finito relacionado a $\mathbb{F}_q$ (uma de suas extensões, por exemplo). Esta definição genérica define um *emparelhamento assimétrico* [67]. A instanciação mais comum de um emparelhamento assimétrico é escolher $\mathbb{G}_1$ como um subgrupo de pontos na curva elíptica $E(\mathbb{F}_q)$, $\mathbb{G}_2$ como um subgrupo de pontos na curva elíptica $E(\mathbb{F}_{q^k})$ e $\mathbb{G}_T$ como o subgrupo de raízes da unidade em $\mathbb{F}^*_{q^k}$.

Se E é uma curva supersingular definida sobre $\mathbb{F}_q$ com grau de mergulho $k > 1$ e r é um divisor primo de $\#E(\mathbb{F}_q)$, o *emparelhamento simétrico* associado com E é o mapa bilinear $\hat{e} : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_T$ definido por $e(P_1, \psi(P_2))$, onde ψ é um *mapa de distorção*. O mapa de distorção é um homomorfismo de $\mathbb{G}_1$ em $\mathbb{G}_2$ cuja função é garantir que $\langle P \rangle$ e $\langle \psi(P) \rangle$ sejam conjuntos linearmente independentes [67].

1.4.3 Problemas subjacentes

Um conjunto de problemas clássicos dos quais se tem evidência de intratabilidade são utilizados explicitamente ou implicitamente por sistemas criptográficos de chave pública:

Problema do Logaritmo Discreto (Discrete Logarithm Problem – DLP): Seja $\mathbb{G}$ um grupo cíclico finito e g um gerador de $\mathbb{G}$. Dado $\langle g, g^a \rangle$, com escolha uniformemente aleatória de $a \in \mathbb{Z}_{|\mathbb{G}|}$, encontrar a.

Problema Diffie-Hellman Computacional (Computational Diffie Hellman Problem – CDHP): Seja $\mathbb{G}$ um grupo cíclico finito e g um gerador de $\mathbb{G}$. Dado $\langle g, g^a, g^b \rangle$ com escolha uniformemente aleatória de $a, b \in \mathbb{Z}_{|\mathbb{G}|}$, encontrar $g^{ab} \in \mathbb{G}$.

O algoritmo mais eficiente para cálculo de logaritmos discretos [68] é uma variação do algoritmo de cálculo de índices [69] e apresenta complexidade sub-exponencial. Para um grupo de pontos em curva elíptica, o DLP consiste em obter a a partir do resultado da operação de multiplicação de ponto aP. Existem evidências de que a técnica de cálculo de índices não possa ser estendida para grupos de pontos em curvas elípticas e, assim, a complexidade do problema DLP em curvas elípticas (*Elliptic Curve Discrete Logarithm Problem* – ECDLP) mantém-se estritamente exponencial [70].

A derivação de problemas análogos aos problemas ditos convencionais, no contexto de emparelhamentos bilineares, é direta:

Problema Diffie-Hellman Bilinear (Bilinear Diffie-Hellman Problem – BDHP): Seja $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ grupos adequados à instanciação de um emparelhamento bilinear admissível e e sejam P um gerador de $\mathbb{G}_1$ e Q um gerador de $\mathbb{G}_2$. Dado $\langle P, aP, bP, cP, Q, aQ, bQ, cQ \rangle$, com escolhas uniformemente aleatórias de $a, b, c \in \mathbb{Z}_{|\mathbb{G}_T|}$, calcular $e(P, Q)^{abc} \in \mathbb{G}_T$ [66].

Pode-se perceber que a intratabilidade do BDHP implica a dificuldade do DLP em $\mathbb{G}_1$, $\mathbb{G}_2$ e $\mathbb{G}_T$, visto que o cálculo de logaritmos discretos em $\mathbb{G}_T$ fornece um oráculo para o cálculo do ECDLP. Assim, a dificuldade do ECDLP nos grupos aditivos deve ser tal que o cálculo subexponencial do DLP em $\mathbb{G}_T$ permaneça difícil [71].

1.4.4 Cálculo de emparelhamentos

Por ser uma área de pesquisa nova, a criptografia baseada em emparelhamentos apresenta uma infinidade de opções do ponto de vista algorítmico. A escolha de parâmetros e algoritmos ainda encontra-se particularmente longe de consolidação. Desta forma, os trabalhos publicados na literatura refletem uma extensa experimentação com diversas escolhas de parâmetros, a fim de encontrar um ponto ótimo entre a escolha da curva e o desempenho da aritmética [25].

São conhecidos dois algoritmos para o cálculo de emparelhamentos bilineares: o algoritmo de Miller [57, 58] e as redes elípticas [72], tendo o primeiro recebido significativa atenção na literatura. Duas das instanciações de emparelhamentos consideradas mais eficientes são o emparelhamento η_T [1], uma modificação do emparelhamento de *Tate* [22] para curvas supersingulares definidas sobre um corpo finito binário; e o emparelhamento *R-ate* [3] sobre curvas ordinárias primas. Entretanto, diversas famílias de curvas são adequadas para instanciação de emparelhamentos, apresentando características distintas que influenciam o desempenho da aritmética empregada para o cálculo de emparelhamentos. Uma dessas características é o grau de mergulho da curva que controla diretamente o tamanho do corpo finito subjacente à curva e o grau da extensão do corpo utilizado na aritmética. Na literatura, já foram relatadas as descobertas de curvas com graus de mergulho diversos, como curvas MNT ($k = 3, 4, 6$) [73], curvas BN ($k = 12$) [74] e mais recentemente, curvas com grau de mergulho significativamente maior ($k = 8, 16, 18, 32, 36, 40$) [75]. O grau de mergulho representa um compromisso direto entre complexidade da aritmética na extensão do corpo e eficiência na aritmética do corpo base, sendo estes níveis de aritmética estimados como os mais solicitados durante o cálculo de um emparelhamento [19].

O cálculo de um emparelhamento bilinear pelo Algoritmo de Miller emprega teoria de divisores de pontos em curvas elípticas. Esta seção esboça as principais definições e fornece uma forma contemporânea do algoritmo.

Divisores

Um *divisor* é uma soma formal de pontos na curva $E(\bar{\mathbb{F}}_q)$:

$$\mathcal{D} = \sum_{P \in E} d_P(P), \tag{1.5}$$

onde d_P é um inteiro e (P) é um símbolo formal. O *grau* de um divisor é a soma de seus coeficientes não-nulos $deg(\mathcal{D}) = \sum_{P \in E} d_P$. Divisores podem envolver símbolos formais para vários pontos na curva elíptica, mas limita-se o enfoque a divisores de funções, por possuírem poucos símbolos. O divisor de uma função $f(x, y) = 0$ contém apenas os termos $d_P(P)$ tais que $P = (x, y)$ está na curva $E(x, y) = 0$ e na função $f(x, y) = 0$, ou seja, os pontos em que E e f se encontram. Para qualquer ponto P na curva elíptica tal que E e f não têm intersecção, $d_p = 0$. O *suporte de um divisor* $\mathcal{D}$ é o conjunto de pontos com coeficientes não-nulos $supp(\mathcal{D}) = \{P \in E | n_P \neq 0\}$.

Uma estrutura de grupo abeliano aditivo é definida no conjunto dos divisores pela soma dos coeficientes correspondentes em suas somas formais. Assim:

$$\begin{aligned} \sum_{P \in E} m_P(P) + \sum_{P \in E} n_P(P) &= \sum_{P \in E} (m_P + n_P)(P) \\ n\mathcal{D} &= \sum_{P \in E} (nd_P)(P). \end{aligned} \tag{1.6}$$

Funções racionais sobre uma curva elíptica

Uma função $f(x, y)$ em um corpo finito $\mathbb{F}_q$ é dita racional se f pode ser escrita como a razão entre polinômios $P(x, y)$ e $Q(x, y)$ em $\mathbb{F}_q$. Um função racional está sobre uma curva $E(\mathbb{F}_q)$ se $f(x, y) = 0$ e $E(x, y) = 0$ têm ao menos uma solução em comum. Utiliza-se a notação $P \in f \cap E$ para um ponto $P = (x, y)$ que satisfaz esta condição.

Seja $f(x, y)$ uma função racional sobre $E(x, y) = 0$. Para um ponto $P \in f \cap E$, P é chamado *zero* se $f(P) = 0$ e *pólo* se $f(P) = \infty$. Para todo ponto $P \in E(\mathbb{F}_q)$, existe uma função racional u com $u(P) = 0$ tal que toda função racional não-nula f pode ser escrita como $f(P) = u^d s(P)$ para algum inteiro d e uma função racional s, tal que $s(P) \notin \{0, \infty\}$. A função u é chamada *uniformizador* e pode ser qualquer reta $ax + by + c = 0$ que passa por P sem ser tangente a E em P. Define-se a *ordem de f em P*, denotada por $ord_P(f)$ como d. Se P é um zero da função f, então $ord_P(f) > 0$ e P tem *multiplicidade $ord_P(f)$*. Se P é um pólo de f, então a $ord_P(f) < 0$ e P tem *multiplicidade $-ord_P(f)$* [76].

Divisor de uma função racional

O divisor de uma função racional f é chamado *divisor principal*:

$$div(f) = \sum_{P \in E} ord_P(P). \tag{1.7}$$

Sabe-se que $deg(\mathcal{D}) = 0$ e $\sum_{P \in E} d_P P = \infty$ se e somente se $\mathcal{D}$ é principal. Dois divisores $\mathcal{C}$ e $\mathcal{D}$ são *equivalentes* ($\mathcal{C} \sim \mathcal{D}$) se a diferença $\mathcal{C} - \mathcal{D}$ é um divisor principal. Para avaliar uma função racional f em um divisor $\mathcal{D}$ com suporte disjunto ao suporte de $div(f)$, calcula-se [76]:

$$f(\mathcal{D}) = \prod_{P \in supp(\mathcal{D})} f(P)^{n_P}. \quad (1.8)$$

Seja $P \in E(\mathbb{F}_q)[r]$ e seja $\mathcal{D}_P$ um divisor equivalente a $(P) - (\infty)$. O divisor $r\mathcal{D}_P$ é principal e existe uma função $f_{r,P}$ tal que $div(f_{r,P}) = r\mathcal{D}_P = r(P) - r(\infty)$ [23].

1.4.5 Algoritmo de Miller

O Algoritmo de Miller [57, 58] constrói $f_{r,P}$ em estágios e avalia esta função sobre um divisor $\mathcal{D} \sim (Q) - (\infty)$ utilizando um método recursivo.

Primeiramente, seja $P \in E(\mathbb{F}_q)[r]$ e $Q \in \mathbb{F}_{q^k}$. Seja $\mathcal{D}$ o divisor $(Q+R)-(R)$ equivalente a $(Q) - (\infty)$. Para qualquer inteiro c, existe uma *função de Miller* $f_{c,P}$ com divisor $(f_{c,P}) = c(P) - (cP) - (c-1)(\infty)$. Então, para quaisquer inteiros a, b, com $g_{aP,bP}$ a linha entre os pontos aP e bP, temos que [23]:

$$f_{a+b,P}(\mathcal{D}) = f_{a,P}(\mathcal{D}) \cdot f_{b,P}(\mathcal{D}) \cdot \frac{g_{aP,bP}(\mathcal{D})}{g_{(a+b)P,-(a+b)P}(\mathcal{D})}. \quad (1.9)$$

Para calcular $\langle P, Q\rangle_r = f_{r,P}(\mathcal{D})$, o Algoritmo de Miller utiliza esta relação para construir e avaliar aplicações de funções de Miller no divisor $\mathcal{D}$, como mostrado no Algoritmo 1.1.

Algoritmo 1.1 Algoritmo de Miller [57, 58].

```
Entrada: $r = \sum_{i=0}^{\log_2(r)} r_i 2^i, P, \mathcal{D} \sim (Q) - (\infty)$.
Saída: $f_{r,P}(\mathcal{D})$.
 1: $T \leftarrow P$
 2: $f \leftarrow 1$
 3: for $i = \lfloor \log_2(r) \rfloor - 1$ downto $0$ do
 4:     $T \leftarrow 2T$
 5:     $f \leftarrow f^2 \cdot \frac{g_{T,T}(\mathcal{D})}{g_{2T,-2T}(\mathcal{D})}$
 6:     if $r_i = 1$ then
 7:         $T \leftarrow T + P$
 8:         $f \leftarrow f \cdot \frac{g_{T,P}(\mathcal{D})}{g_{T+P,-(T+P)}(\mathcal{D})}$
 9:     end if
10: end for
11: return $f$
```

Considerando que $g_{T,P}(\mathcal{D}) = l((Q+R)-(R)) = \frac{l(Q+R)}{l(R)}$, com l a linha que passa por T e P na adição $T+P$; e $g_{T,T}(\mathcal{D}) = \frac{v(Q+R)}{v(R)}$, com v a vertical que passa por T na duplicação $2T$, pode-se reescrever o Algoritmo de Miller na forma do Algoritmo 1.2.

Algoritmo 1.2 Algoritmo de Miller [57, 58].

```
Entrada: $r = \sum_{i=0}^{\log_2 r} r_i 2^i, P, Q + R, R$.
Saída: $\langle P, Q\rangle_r$.
 1: $T \leftarrow P$
 2: $f \leftarrow 1$
 3: for $i = \lfloor \log_2(r) \rfloor - 1$ downto $0$ do
 4:   $T \leftarrow 2T$
 5:   $f \leftarrow f^2 \cdot \frac{l(Q+R)v(R)}{v(Q+R)l(R)}$
 6:   if $r_i = 1$ then
 7:     $T \leftarrow T + P$
 8:     $f \leftarrow f \cdot \frac{l(Q+R)v(R)}{v(Q+R)l(R)}$
 9:   end if
10: end for
11: return $f$
```

1.4.6 Emparelhamento de Tate

O *emparelhamento de Tate* de ordem r é o mapa:

$$\begin{aligned} e_r &: E(\mathbb{F}_q)[r] \times E(\mathbb{F}_{q^k})[r] \rightarrow \mathbb{F}_{q^k}^* \\ e_r(P, Q) &= f_{r,P}(\mathcal{D})^{(q^k-1)/r}, \end{aligned} \tag{1.10}$$

para algum divisor $\mathcal{D} \sim (Q) - (\infty)$. Se escolhemos R com coordenadas em um subcorpo de $\mathbb{F}_{q^k}$, a potência com expoente $(q^k - 1)/r$ elimina todos os fatores $l(R), v(R)$ no Algoritmo 1.2. Isto acontece porque $(q - 1)|(q^k - 1)/r$ e, pelo pequeno Teorema de Fermat, $a^{(q-1)} = 1$. Fazendo $R = \infty$ no Algoritmo 1.2, temos ainda que $e_r(P, Q) = f_{r,P}(Q)^{(q^k-1)/r}$ [23].

Para um emparelhamento simétrico, pode-se escolher o mapa de distorção de forma que a coordenada x de $\psi(P)$ esteja em um subcorpo de $\mathbb{F}_{q^k}$. Assim, os fatores $v(Q)$ também serão eliminados pela exponenciação final [1].

Para um emparelhamento assimétrico, quando o grau de mergulho k é par, pode-se assumir que a extensão $\mathbb{F}_{q^k}$ é construída como uma extensão quadrática sobre $\mathbb{F}_{q^d}$, $d = k/2$. Desta forma, a potência $(q^k - 1)$ pode ser fatorada como $(q^d - 1)(q^d + 1)/r$ e calcula-se $(\frac{1}{v(Q)})^{q^d-1}$ como o conjugado $(\bar{v}(Q))^{q^d-1}$. Como Q é agora um ponto com coordenadas na extensão quadrática de $\mathbb{F}_{q^d}$, pode-se restringir Q à forma $(x_Q, y_Q) = ((a, 0), (0, b))$, $a, b \in \mathbb{F}_{q^d}$. Assim, o fator $\bar{v}(Q)$ é agora um elemento do subcorpo $\mathbb{F}_{q^d}$ também eliminado pela exponenciação final. Observa-se que, mesmo restringindo a forma do ponto Q, ainda é possível realizar aritmética na curva $E(\mathbb{F}_{q^k})$, pois o conjunto de pontos Q neste formato pode ser mapeado para um grupo isomórfico no *twist* da curva E. O *twist de grau d*

$E^t(\mathbb{F}_q)$ de uma curva $E(\mathbb{F}_q)$ é dado por $y^2 = x^3 + z^2ax + z^3b$ para todo não-resíduo z de grau d.

Finalmente, pode-se eliminar a última contribuição do laço de Miller, pois necessariamente esta contribuição ao acumulador irá estar em um subcorpo [23]. O Algoritmo 1.3 apresenta a especialização do Algoritmo de Miller para o cálculo do emparelhamento de Tate [19].

Algoritmo 1.3 Emparelhamento de Tate [19].

```
Entrada: $r = \sum_{i=0}^{\log_2 r} r_i 2^i, P, Q$.
Saída: $e_r(P, Q)$.
 1: $T \leftarrow P$
 2: $f \leftarrow 1$
 3: $s \leftarrow r - 1$
 4: for $i = \lfloor \log_2(s) \rfloor - 1$ downto 0 do
 5:   $T \leftarrow 2T$
 6:   $f \leftarrow f^2 \cdot l_{T,T}(Q)$
 7:   if $s_i = 1$ then
 8:     $T \leftarrow T + P$
 9:     $f \leftarrow f \cdot l_{T,P}(Q)$
10:   end if
11: end for
12: return $f^{(q^k - 1/r)}$
```

1.4.7 Emparelhamento de Weil

O *emparelhamento de Weil* [58] é definido classicamente como:

$$\begin{aligned} e_w &: E(\mathbb{F}_{q^k})[r] \times E(\mathbb{F}_{q^k})[r] \rightarrow \mathbb{F}_{q^k}^* \\ e_r(P, Q) &= \frac{f_{r,Q}(\mathcal{D_P})}{f_{r,P}(\mathcal{D_Q})}, \end{aligned} \tag{1.11}$$

para divisores $\mathcal{D_P} \sim (P) - (\infty)$ e $\mathcal{D_Q} \sim (Q) - (\infty)$. Uma das funções de Miller pode ser acelerada escolhendo o acumulador $P \in E(\mathbb{F}_q)[r]$, quando é comumente chamada de *Miller leve*, em contraste à função com acumulador $Q \in E(\mathbb{F}_{q^k})[r]$ chamada de *Miller completa* [77]. A princípio, não há exponenciação final como no emparelhamento de Tate, mas introduzir um expoente $(q^k - 1)$ permite eliminar as linhas verticais, como no caso anterior. O emparelhamento de Weil é examinado no Capítulo 7.

1.4.8 Emparelhamento η_T [1]

O emparelhamento η_T, proposto por Barreto et al. em 2004 é uma especialização do emparelhamento de Tate exclusiva para curvas supersingulares sobre corpos de característica pequena (2 ou 3). Esta especialização aplica o método *eta* desenvolvido por Duursma e Lee para curvas hiperelípticas [78]. Restringindo a definição para curvas supersingulares binárias da forma $y^3 + y = x^3 + x + b$, este emparelhamento define o mapa:

$$\begin{aligned} \eta_T &: E(\mathbb{F}_{2^m})[r] \times E(\mathbb{F}_{2^{km}})[r] \rightarrow \mathbb{F}^*_{2^{km}}, \\ \eta_T(P,Q) &= f_{T,P}(Q)^{(2^{km}-1)/r}, \end{aligned} \tag{1.12}$$

O Algoritmo de Miller pode ser também especializado para o cálculo do emparelhamento η_T. Para isso, é preciso explorar a forma simples dos divisores de funções em curvas supersingulares e embutir a definição do mapa de distorção associado no cálculo do emparelhamento. O Algoritmo 1.4 descreve o método proposto por [79] para calcular a fórmula fechada do emparelhamento apresentada em [1]. Este emparelhamento é revisitado nos Capítulos 5 e 7.

Algoritmo 1.4 Emparelhamento η_T em curva supersingular binária

Entrada: $P = (x_P, y_P), Q = (x_Q, y_Q) \in E(\mathbb{F}_{2^m})[r]$.
Saída: $\eta_T(P,Q) \in \mathbb{F}^*_{2^{4m}}$.
Nota: $\delta = b$ se $m \equiv 1,7 \pmod 8$, $\delta = (1-b)$ caso contrário; $\alpha = 0$ se $m \equiv 3 \pmod 4$, $\alpha = 1$ caso contrário; $\beta = b$ se $m \equiv 1,3 \pmod 8$, $\beta = (1-b)$ caso contrário.

```
1: $y_P \leftarrow y_P + 1 - \delta$
2: $u \leftarrow x_P + \alpha, v \leftarrow x_Q + \alpha$
3: $g_0 \leftarrow u \cdot v + y_P + y_Q + \beta$
4: $g_1 \leftarrow u + x_Q, g_2 \leftarrow v + x_P^2$
5: $G \leftarrow g_0 + g_1 s + t$
6: $L \leftarrow (g_0 + g_2) + (g_1 + 1)s + t$
7: $F \leftarrow L \cdot G$
8: for $i \leftarrow 1$ to $\frac{m-1}{2}$ do
9:     $x_P \leftarrow \sqrt{x_P}, y_P \leftarrow \sqrt{y_P}, x_Q \leftarrow x_Q^2, y_Q \leftarrow y_Q^2$
10:    $u \leftarrow x_P + \alpha, v \leftarrow x_Q + \alpha$
11:    $g_0 \leftarrow u \cdot v + y_P + y_Q + \beta$
12:    $g_1 \leftarrow u + x_Q$
13:    $G \leftarrow g_0 + g_1 s + t$
14:    $F \leftarrow F \cdot G$
15: end for
16: return $F^{(2^{2m}-1)(2^m+1\pm 2^{\frac{m+1}{2}})}$
```

1.4.9 Emparelhamento *ate* [2]

O emparelhamento *ate* proposto por Hess et al. em 2005 permuta os argumentos do emparelhamento de Tate e generaliza o emparelhamento *eta* para curvas não-supersingulares com traço de Frobenius pequeno. Seja $\pi_q : E \to E$ o *endomorfismo de Frobenius* definido por $\pi_q(x, y) = (x^q, y^q)$. Seja $\mathbb{G}_1 = E[r] \cap Ker(\pi_q - [1])$, $\mathbb{G}_2 = E[r] \cap Ker(\pi_q - [q])$, $T = t - 1$. Seja $N = gcd(T^k - 1, q^k - 1)$, $T^k - 1 = LN$. O emparelhamento *ate* é o mapa [2]:

$$\begin{aligned} a_T &: \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{F}^*_{q^k} \\ a_T(Q, P) &= f_{T,Q}(P)^{c_T(q^k-1)/N}, \end{aligned} \tag{1.13}$$

onde $c_T = \sum_{i=0}^{k-1-i} q^i \equiv kq^{k-1} \pmod r$. O emparelhamento *ate* é bilinear e não-degenerado se r não divide L. O Algoritmo 1.5 apresenta a especialização do Algoritmo de Miller para o cálculo do emparelhamento *ate* [19].

Algoritmo 1.5 Emparelhamento *ate* [19].

```
Entrada: $t = \sum_{i=0}^{\log_2 t} t_i 2^i, Q, P$.
Saída: $a_T(Q, P)$.
1: $T \leftarrow P$
2: $f \leftarrow 1$
3: $s \leftarrow t - 1$
4: for $i = \lfloor \log_2(s) \rfloor - 1$ downto $0$ do
5:    $T \leftarrow 2T$
6:    $f \leftarrow f^2 \cdot l_{T,T}(P)$
7:    if $t_i = 1, i \neq 0$ then
8:       $T \leftarrow T + P$
9:       $f \leftarrow f \cdot l_{T,Q}(T)$
10:   end if
11: end for
12: return $f^{(q^k-1/r)}$
```

1.4.10 Emparelhamento *R-ate* [3]

O emparelhamento *R-ate*, proposto por Lee, Lee e Park em 2008 é, por sua vez, uma generalização do emparelhamento *ate* que aprimora sua eficiência. Este emparelhamento calcula um emparelhamento bilinear a partir da razão entre dois emparelhamentos bilineares e requer duas aplicações do Algoritmo de Miller com um número pequeno de interações [3]. Para uma escolha cuidadosa de inteiros a, b, A, B, com $A = aB + b$, o emparelhamento *R-ate* é o mapa [3]:

$$\begin{aligned}
e_a &: E(\mathbb{F}_q^k)[r] \times E(\mathbb{F}_q)[r] \rightarrow \mathbb{F}_{q^k}^* \\
e_a(Q,P) &= f_{a,BQ}(P) \cdot f_{b,Q}(P) \cdot \frac{g_{aBQ,bQ}(P)}{g_{AQ}(P)} \\
e_a(Q,P) &= (f \cdot (f \cdot l_{aQ,Q}(P))^q \cdot l_{\pi(aQ+Q)}(P)^{(q^k-1)/r}. \qquad (1.14)
\end{aligned}$$

1.4.11 Emparelhamento otimal

Pode-se perceber que a principal otimização aplicada ao Algoritmo de Miller é a diminuição do número de iterações do laço principal do algoritmo. Vercauteren [24] estabelece que qualquer emparelhamento cujo laço de Miller pode ser truncado para $\log_2(r)/\phi(k)$ é chamado de *emparelhamento otimal* e prova ainda que $\log_2(r)/\phi(k)$ iterações é uma cota mínima para qualquer emparelhamento não-degenerado em curvas elípticas sem endomorfismos eficientemente computáveis que não sejam potências de Frobenius. Neste sentido, os emparelhamentos η_T e *R-ate* são otimais. O emparelhamento *ate* otimal estudado no Capítulo 6 é outro exemplo de emparelhamento otimal.

1.5 Contribuições

Esta seção apresenta uma relato não-cronológico dos resultados obtidos durante a execução deste trabalho, organizados em duas partes: implementação de criptografia de curvas elípticas e de criptografia baseada em emparelhamentos. Estas contribuições foram disponibilizadas na forma de uma biblioteca de *software* descrita ao final da seção. Nem todos os trabalhos discutidos nessa seção estão incluídos na tese, mas decidiu-se por divulgá-los para manter um relatório do progresso do autor.

1.5.1 Implementação de criptografia de curvas elípticas

A primeira etapa do trabalho consistiu na implementação de corpos e curvas elípticas binárias em sensores em fio. 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 [80, 81]. Atualmente, criptografia de curvas elípticas [8, 7] e criptografia baseada em emparelhamentos [10] têm sido apontadas como alternativas promissoras aos métodos convencionais de criptografia assimétrica em redes de sensores [82], 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.

Durante a execução desta etapa, foram detectadas 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 [82, 83].

Como resultados obtidos decorrentes desta etapa, pode-se citar os trabalhos [84, 85, 86] que implementam curvas elípticas em microcontroladores AVR ATmega128 de 8 *bits*:

1. **D. F. Aranha**, D. Câmara, J. López, L. Oliveira, R. Dahab. *Implementação eficiente de criptografia de curvas elípticas em sensores sem fio.* 8o. Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2008), 173–186, Gramado, Brasil, 2008;

2. **D. F. Aranha**, J. López, L. Oliveira, R. Dahab. *Efficient implementation of elliptic curves on sensor nodes.* Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc. (CHiLE 2009), Frutillar, Chile, 2009;

3. **D. F. Aranha**, R. Dahab, J. López, L. Oliveira. *Efficient implementation of elliptic curve cryptography in wireless sensors.* Advances in Mathematics of Communications, vol. 4, no. 2, 169–187, 2010.

Os dois primeiros aprimoram o estado-da-arte de implementações de criptografia de curvas elípticas em redes de sensores sem fio. Nós sensores representam um extremo no espectro de arquiteturas modernas, por terem recursos particularmente limitados e natureza descartável. Aproveitando as características peculiares da plataforma alvo, particularmente a configuração da hierarquia de memória, foi possível desenvolver otimizações para aritmética nos corpos finitos $\mathbb{F}_{2^{163}}$ e $\mathbb{F}_{2^{233}}$ e curva elípticas associadas, produzindo as implementações mais eficientes de quadrado, multiplicação, inversão e redução modular já publicadas para esta plataforma. A aritmética eficiente no corpo permitiu o cálculo de uma mutiplicação de ponto 61% mais rápida que a melhor implementação de curvas elípticas sobre corpos binários e 57% mais rápida que a melhor implementação para o caso primo, considerando o mesmo nível de segurança. O terceiro trabalho consolida estes resultados e é apresentado como Capítulo 2 desta tese.

O segundo conjunto de resultados trata da implementação de corpos binários em conjuntos de instruções vetoriais. Em particular, havia interesse em se aproveitar explicitamente instruções de permutação de *bytes* que codificam implicitamente acessos simultâneos a tabelas de constantes. Cronologicamente, este estudo foi iniciado com a implementação de um corpo razoavelmente grande ($\mathbb{F}_{2^{1223}}$) para o cálculo do emparelhamento η_T em arquiteturas de 64 *bits*. Posteriormente, verificou-se que as mesmas otimizações eram aplicáveis a corpos muito menores e foi desenvolvida uma formulação da aritmética com granularidade de 4 *bits* que emprega vastamente as instruções de permutação, produzindo inclusive um novo ainda que ineficiente algoritmo de multiplicação. Esta formulação foi complementada com um estudo detalhado dos impactos algorítmicos da disponibilidade repetina de suporte nativo à multiplicação em um corpo binário recentemente introduzida na arquitetura Intel [87]. A aceleração significativa da operação de multiplicação forçou uma reavaliação de estratégias de implementação para a operação de *halving* em curvas elípticas binárias e de paralelizações da operação de multiplicação de ponto. Estes resultados são apresentados nos artigos a seguir [88, 89]:

1. **D, F. Aranha**, J. López, D. Hankerson. *Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets.* 1st International Conference on Cryptology and Information Security (LATINCRYPT 2010), 144–161, Puebla, Mexico, 2010.

2. J. Taverne, A. Faz-Hernández, **D. F. Aranha**, F. Rodríguez-Henríquez, D. Hankerson, J. López.*Software Implementation of Binary Elliptic Curves: Impact of the Carry-less Multiplier on Scalar Multiplication.* 13th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2011), em publicação.

O primeiro trabalho é apresentado como o Capítulo 3 desta tese e obtém ganhos de desempenho em 8%-84% para diversas operações em um corpo binário. Esta formulação eficiente da aritmética permitiu aprimorar o estado da arte [90] para multiplicação de ponto em 27%-30%, desconsiderando o modo de operação em lote [61]. O custo médio de uma multiplicação de ponto em modo lote no nível de segurança de 128 *bits* foi superado em 10% apenas no segundo trabalho, após a introdução de suporte nativo à multiplicação em um corpo binário. Este último trabalho, apresentado como Capítulo 4, apresenta ainda resultados experimentais para implementações seriais e paralelas da operação de multiplicaçao de ponto nos níveis de segurança de 112, 128 e 192 *bits*.

1.5.2 Implementação de criptografia baseada em emparelhamentos bilineares

Em seguida, buscou-se estudar protocolos úteis para o fornecimento de serviços de segurança em redes de sensores sem fio. Um exemplos com estas características é o protocolo Sakai-Ohgishi-Kasahara [13] para acordo de chaves não-interativo, providenciando o acordo autenticado de chaves simétricas entre nós sensores sem exigir qualquer comunicação, o que permite importante economia de energia. Outro exemplo é o emprego de assinaturas digitais para transmissão autenticada de mensagens entre um nó específico da rede de sensores e um usuário ou aplicação final. Esta linha de pesquisa produziu os resultados abaixo [91, 92, 93]:

1. **D. F. Aranha**, L. B. Oliveira, J. López, R. Dahab. *NanoPBC: implementing cryptographic pairings on an 8-bit platform.* Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc. (CHiLE 2009), Frutillar, Chile, 2009;

2. L. B. Oliveira, **D. F. Aranha**, C. P. L. Gouvêa, Danilo F. Câmara, M. Scott, J. López, R. Dahab. *TinyPBC: Pairings for Authenticated Identity-Based Non-Interactive Key Distribution in Sensor Networks.* Computer Communications, vol. 34, no. 3, 485–493, 2011;

3. L. B. Oliveira, A. Kansal, C. P. L. Gouvêa, **D. F. Aranha**, J. López, B. Priyantha, M. Goraczko, F. Zhao. *Secure-TWS: Authenticating Node to Multi-user Communication in Shared Sensor Networks.* The Computer Journal, em publicação.

A intersecção entre este trabalho e os trabalhos acima consistiu principalmente na contribuição de implementações otimizadas dos algoritmos criptográficos envolvidos na plataforma AVR ATmega128. Os dois primeiros trabalhos obtém uma aceleração de até 61% no cálculo do emparelhamento η_T [1] sobre curvas supersingulares binárias a partir de uma generalização das técnicas previamente desenvolvidas para a implementação de corpos binários. O terceiro trabalho compara a implementação de diversos esquemas de assinatura sobre curvas elípticas e emparelhamentos bilineares, fornecendo um panorama completo da relação entre tempo de execução, consumo de memória e consumo de energia para o cálculo de assinaturas digitais. Os esquemas implementados foram ECDSA, Schnorr [94] sobre curvas elípticas e assinaturas curtas Boneh-Lynn-Shacham (BLS) [17] e Zhang-Safavi-Naini-Susilo (ZSS) [95], utilizando curvas elípticas definidas sobre corpos binários ou primos. De forma quase que surpreendente, foi detectado que assinaturas curtas não fornecem a economia de energia suficiente na plataforma alvo para justificar sua escolha em detrimento de esquemas convencionais como ECDSA ou Schnorr. Isto ocorre porque as curvas padrão utilizadas no ECDSA possuem tamanhos de parâmetros e algoritmos de multiplicação por escalar geralmente mais eficientes do que os disponíveis em curvas adequadas para a instanciação de emparelhamentos.

O cálculo de emparelhamento bilineares é a operação com maior custo computacional envolvida em criptografia baseada em emparelhamentos. Por essa razão, obstáculos significativos de desempenho ainda permanecem para adoção de criptografia baseada em emparelhamentos em dispositivos de baixo poder computacional, especialmente para altos níveis de segurança [10]. Dadas as tendências tecnológicas recentes da indústria de computação em migrar as arquiteturas computacionais para arquiteturas paralelas, algoritmos paralelos para cálculo de emparelhamentos em arquiteturas multiprocessadas são desejáveis. Este problema é sugerido como um problema em aberto em [96] e [67].

Desta forma, foi derivada uma formulação paralela do Algoritmo de Miller [58] empregado para o cálculo de emparelhamentos bilineares. Esta formulação fornece um algoritmo genérico independente da instanciação do emparelhamento e com ótima escalabilidade em curvas sobre corpos de característica pequena. Esta formulação foi ilustrada com a implementação paralela do emparelhamento η_T sobre curvas supersingulares binárias e do emparelhamento *ate* optimal sobre curvas Barreto-Naehrig (BN) [97], aprimorando o estado-da-arte do cálculo paralelo de emparelhamentos descrito nos trabalhos [98, 99]. Estes resultados são descritos em [100, 101, 102, 20, 103, 104]:

1. **D. F. Aranha**, J. López. *Paralelização em Software do Algoritmo de Miller.* 9o. Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2009), 27–40, Campinas, Brasil, 2009.

2. **D. F. Aranha**, J. López, D. Hankerson. *High-speed parallel software implementation of η_T pairing.* Software Performance Enhancement of Encryption and Decryption and Cryptographic Compilers/SPEED-CC, Berlim, Alemanha, 2009.

3. **D. F. Aranha**, J. López, D. Hankerson. *High-speed parallel software implementation of η_T pairing.* Cryptographer's Track - RSA Conference (CT-RSA 2010), 89–105, São Francisco, Estados Unidos, 2010.

4. **D. F. Aranha**, J.-L. Beuchat, J. Detrey, N. Estibals. *Optimal Eta Pairing on Supersingular Genus-2 Binary Hyperelliptic Curves.* Cryptology ePrint Archive, Report 2010/559.

5. **D. F. Aranha**, K. Karabina, P. Longa, C. H. Gebotys, J. López. *Faster Explicit Formulas for Computing Pairings over Ordinary Curves.* 30th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2011), 48–68 Talynn, Estônia, 2011.

6. **D. F. Aranha**, F. Henríquez-Rodríguez, E. Knapp, A. Menezes. *Parallelizing the Weil and Tate Pairings.* 13th Institute of Mathematics and its Applications' International Conference on Cryptography and Coding (IMA-CC 2011), em publicação.

O primeiro trabalho descreve a construção genérica com enfoque apenas na paralelização do Algoritmo de Miller. O segundo trabalho introduz uma série de técnicas para implementação de corpos binários em processadores equipados com conjuntos de instruções vetoriais. Estas técnicas colaboram para reduzir sobrecargas da paralelização genérica e produzem aprimoramentos em relação ao estado-da-arte de 28%, 44% e 66% empregando 2, 4 e 8 processadores, respectivamente. A aceleração do cálculo serial foi também significativa e sitou-se em pouco mais de 24%. Este conjunto aprimorado de resultados foi apresentado na edição 2009 do *workshop* bienal SPEED-CC. Como o *workshop* não produz anais convencionais – apenas um volume de registro – e a audiência deste é bastante restrita a especialistas na área de implementação, decidiu-se por enviar uma versão modificada do trabalho também para o *Cryptographers' Track* da *RSA Conference* 2010 (CT-RSA). Este trabalho encontra-se apresentado como Capítulo 5. As técnicas avançadas para a implementação de emparelhamento sobre corpos binários permitiram ainda o estabelecimento de um novo recorde de velocidade para emparelhamentos simétricos com a derivação e implementação de um novo emparelhamento *eta* sobre curvas supersingulares binarias de genus 2, discutida no quarto trabalho.

Após o estudo de implementações eficientes de emparelhamentos definidos sobre corpos binários, o quinto trabalho foi direcionado a corpos primos. Foi possível estabelecer um novo recorde de velocidade para o cálculo de emparelhamentos sobre curvas BN no nível de segurança de 128 *bits* com um aprimoramento de 28% a 34% em relação ao estado da arte. Este resultado foi obtido a partir da utilização de todas as técnicas propostas na literatura recente, além da introdução de novas técnicas como a aceleração da aritmética em corpos de extensão com a generalização da noção de redução modular preguiçosa, a concepção de novas fórmulas para cálculo comprimido de quadrados em grupos ciclotômicos e a eliminação de penalidades para parametrizações negativas da curva subjacente. Este trabalho encontra-se apresentado como Capítulo 6. O capítulo final da tese dedica-se a revisitar o problema do cálculo paralelo de emparelhamentos, aprimorando significativamente os resultados obtidos anteriormente com o emprego dos quadrados comprimidos em grupos ciclotômicos, do suporte nativo à multiplicação binária e de um compromisso entre tempo de execução e espaço de armazenamento para acelerar a exponenciação final do emparelhamento η_T. Além disso, como tentativa de se contornar os obstáculos à paralelização do Algoritmo de Miller, foram propostos dois emparelhamentos baseados no emparelhamento de Weil, sendo o primeiro deles uma instanciação direta de uma construção de Hess e o segundo derivado de uma nova construção demonstrada como bilinear. Esta nova construção permitiu incrementar a escalabilidade do cálculo do emparelhamento *ate* otimal primos em máquinas com até 8 unidades de processamento.

1.5.3 Biblioteca criptográfica

Um efeito colateral direto deste trabalho foi a fundação do projeto *RELIC* (*RELIC is an Efficient LIbrary for Cryptography*) [105]. O projeto teve como finalidade inicial a produção de uma biblioteca criptográfica para dispositivos embutidos que tivesse um menor consumo de memória que as alternativas. Atualmente, o projeto já constitui um *framework* completo para experimentação com implementação eficiente de algoritmos criptográficos, fornecendo ampla portabilidade e arquitetura modular especialmente projetada para permitir acelerações dependentes de arquitetura. São suportadas curvas elípticas e emparelhamentos sobre corpos binários e primos. A versão em repositório da biblioteca já atingiu a marca de 60 mil linhas de código, contando com mais de 5000 acessos de 900 visitantes únicos provenientes de 65 países, e mais de 1000 *downloads* distribuídos nas 7 versões já disponibilizadas.

1.6 Organização do documento

Este tese é organizada como uma coletânea em ordem não-cronológica de seis artigos, onde cada artigo é apresentado na forma de um capítulo auto-contido. O critério de seleção dos artigos foi o grau de envolvimento do autor, sendo selecionados apenas os trabalhos em que houve oportunidade de se participar na concepção, desenvolvimento, implementação e confecção do documento final. Os três primeiros capítulos descrevem contribuições para o problema de implementação eficiente de criptossistemas de curvas elípticas, enquanto os três últimos capítulos dedicam-se ao problema de implementação eficiente de emparelhamentos. A maior vantagem deste formato em coletânea é permitir a leitura não-linear e independente das contribuições originais, enquanto a maior desvantagem é a inevitável redundância em algumas das definições. O último capítulo apresenta conclusões e perspectivas de trabalho futuro.

Chapter 2

Efficient implementation of elliptic curve cryptography in wireless sensors

Diego F. Aranha, Leonardo B. Oliveira, Ricardo Dahab and Julio López

Abstract

The deployment of cryptography in sensor networks is a challenging task, given the limited computational power and the resource-constrained nature of the sensoring devices. This paper presents the implementation of elliptic curve cryptography in the MICAz Mote, a popular sensor platform. We present optimization techniques for arithmetic in binary fields, including squaring, multiplication and modular reduction at two different security levels. Our implementation of field multiplication and modular reduction algorithms focuses on the reduction of memory accesses and appears as the fastest result for this platform. Finite field arithmetic was implemented in C and Assembly and elliptic curve arithmetic was implemented in Koblitz and generic binary curves. We illustrate the performance of our implementation with timings for key agreement and digital signature protocols. In particular, a key agreement can be computed in 0.40 seconds and a digital signature can be computed and verified in 1 second at the 163-bit security level. Our results strongly indicate that binary curves are the most efficient alternative for the implementation of elliptic curve cryptography in this platform.

Publication

This work first appeared in Advances in Mathematics of Communication 4 (2), 2010, published by the American Institute of Mathematical Sciences and Shandong University and available at `http://dx.doi.org/10.3934/amc.2010.4.169`. ©2010 AIMS-SDU

2.1 Introduction

A Wireless Sensor Network (WSN) [54] is a wireless ad-hoc network consisting of resource-constrained sensoring devices (limited energy source, low communication bandwidth, small computational power) and one or more base stations. The base stations are more powerful and collect the data gathered by the sensor nodes so it can be analyzed. As any ad hoc network, routing is accomplished by the nodes themselves through hop-by-hop forwarding of data. Common WSN applications range from battlefield reconnaissance and emergency rescue operations to surveillance and environmental protection.

WSNs may be organized in different ways. In flat WSNs, all nodes play similar roles in sensing, data processing, and routing. In hierarchical WSNs, on the other hand, the network is typically organized into clusters, with ordinary cluster members and the cluster heads playing different roles. While ordinary cluster members are responsible for sensing, the cluster heads are responsible for additional tasks such as collecting and processing the sensing data from their cluster members, and forwarding the results towards the base stations.

Besides the vulnerabilities already present in ad-hoc networks, WSNs pose additional challenges: the sensor nodes are commonly distributed on locations physically accessible to adversaries; and the resources available in a sensor node are more limited than those in a conventional ad hoc network node, thus traditional solutions are not adequate. For example, the fact that sensor nodes should be discardable and consequently have low cost makes the integration of anti-tampering measures on these devices difficult.

Conventional public key cryptography systems such as RSA and DSA are impractical in this scenario due to the low processing power of sensor nodes. Until recently, security services such as confidentiality, authentication and integrity were achieved exclusively by symmetric techniques [80, 81]. Nowadays, however, elliptic curve cryptography (ECC) [8, 7] has emerged as a promising alternative to traditional public key methods on WSNs [82], because of its lower processing and storage requirements. These features motivate the search for increasingly efficient algorithms and implementations of ECC for such devices. The usual target platform is the MICAz Mote [106], a node commonly used on real WSN deployments, whose main characteristics are the low availability of RAM memory and the high cost of memory instructions, memory addressing and bitwise shifts by arbitrary amounts.

This work proposes optimizations for implementing ECC over binary fields, improving its limits of performance and viability. Experimental results show that binary elliptic curves offer significant computational advantages over prime curves when implemented in WSNs. Note that this observation contradicts a common misconception that sensor nodes are not sufficiently equipped to compute elliptic curve arithmetic over binary fields in an efficient way [82, 83].

Our main contributions in this work are:

- *Efficient implementations of multiplication, squaring, modular reduction and inversion in $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$:* optimized versions of known algorithms are presented, reducing the number of memory accesses to obtain performance gains. The new optimizations produce the fastest implementation of binary field arithmetic published for this platform;

- *Efficient implementation of elliptic curve cryptography:* point multiplication algorithms are implemented on Koblitz curves and generic binary curves. The time for a scalar multiplication of a random point in a binary curve is 61% faster than the best implementation so far [107] and 57% faster than the best implementation over a prime curve [108] at the 160-bit security level. We also present the first point multiplication timings at the 233-bit security level in this platform. Performance is illustrated by executions of key agreement and digital signature protocols.

The remaining sections of this paper are organized as follows. Related work is presented in Section 2.2 and elementary elliptic curve concepts are introduced in Section 2.3. The platform characteristics are presented in Section 2.4. Section 2.5 investigates efficient implementations of finite field arithmetic in the target platform while Section 2.6 investigates efficient elliptic curve arithmetic. Section 2.7 presents implementation results and Section 2.8 concludes the paper.

2.2 Related work

Cryptographic protocols are used to establish security services in WSNs. Key agreement is a fundamental protocol in this context because it can be used to negotiate cryptographic keys suitable for fast and energy-efficient symmetric algorithms. One possible solution for key agreement in WSNs is the deployment of pairing-based protocols, such as TinyTate [109] and TinyPBC [10], with the added advantage of not requiring communication. Here instead we focus on the performance side and assume that a simple one-pass Elliptic Curve Diffie-Hellman [110] protocol is employed for key agreement. With this assumption, different implementations of ECC can be compared by the cost of multiplying a random elliptic point by a random integer.

Gura et al. [82] presented the first implementation results of ECC and RSA on ATmega128 microcontrollers and demonstrated the superiority of the former over the latter. In Gura's work, prime field arithmetic was implemented in C and Assembly and a point multiplication took 0.81 seconds on a 8MHz device. Uhsadel et al. [111] later presented an expected time of 0.76 seconds for computing a point multiplication in a 7.3728MHz device. The fastest implementation of prime curves so far [108] explores the potential of elliptic curves with efficient computable endomorphisms defined over optimal prime fields and computes a point multiplication in 5.5 million cycles, or 0.745 second.

For binary curves, Malan et al. [112] implemented ECC using polynomial basis and presented results for the Diffie-Hellman key agreement protocol. A public key generation, which consists of a point multiplication, was computed in 34 seconds. Yan and Shi [113] implemented ECC over $\mathbb{F}_{2^{163}}$ and obtained a point multiplication in 13.9 seconds, suggesting that binary curves had too high a cost for sensors' current technology. Eberle et al. [83] implemented ECC in Assembly over $\mathbb{F}_{2^{163}}$ and obtained a point multiplication in 4.14 seconds, making use of architectural extensions for additional acceleration. NanoECC [114] specialized portions of the MIRACL arithmetic library [115] in the C programming language for efficient execution in sensor nodes, resulting in a point multiplication in 2.16 seconds over prime fields and 1.27 seconds over binary fields. Later, TinyECCK [116] presented an implementation of ECC over binary curves which takes into account the platform characteristics to optimize finite field arithmetic and obtained a point multiplication in 1.14 second. Recently, Kargl et al. [107] investigated algorithms resistant to simple power analysis and obtained a point multiplication in 0.7633 second on a 8MHz device. Table 2.2 presents the increasing efficiency of ECC in WSNs.

Finite field	**Work**	**Execution time (seconds)**
Binary	Malan et al. [112]	34
	Yan and Shi [113]	13.9
	Eberle et al. [83]	4.14
	NanoECC [114]	2.16
	TinyECCK [116]	1.14
	Kargl et al. [107]	0.83
Prime	Wang and Li. [117]	1.35
	NanoECC [114]	1.27
	Gura et al. [82]	0.87
	Uhsadel et al. [111]	0.76
	TinySA [108]	0.745

Table 2.1: Timings for scalar multiplication of a random point on a MICAz Mote at the 160-bit security level. The timings are normalized for a clock frequency of 7.3728MHz.

2.3 Elliptic curve cryptography

An *elliptic curve* E over a field $\mathbb{K}$ is the set of solutions $(x, y) \in \mathbb{K} \times \mathbb{K}$ which satisfy the *Weierstrass equation*

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

where $a_1, a_2, a_3, a_4, a_6 \in \mathbb{K}$ and the curve *discriminant* is $\Delta \neq 0$; together with a *point at infinity* denoted by $\mathcal{O}$. If $\mathbb{K}$ is a field of characteristic 2, then the curve is called a *binary elliptic curve* and there are two cases to consider. If $a_1 \neq 0$, then an admissible change of variables transforms E to the *non-supersingular binary elliptic curve* of equation

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

where $a, b \in \mathbb{F}_{2^m}$ and $\Delta = b$. A non-supersingular curve with $a \in \{0, 1\}$ and $b = 1$ is also a *Koblitz curve.* If $a_1 = 0$, then an admissible change of variables transforms E to the *supersingular binary elliptic curve*

$$y^2 + cy = x^3 + ax + b$$

where $a, b, c \in \mathbb{F}_{2^m}$ and $\Delta = c^4$.

The number of points on the curve $E(\mathbb{F}_{2^m})$, denoted by $\#E(\mathbb{F}_{2^m})$, is called the *curve order* over the field $\mathbb{F}_{2^m}$. The *Hasse bound* enunciates in this case that $n = 2^m + 1 - t$ and $|t| \leq 2\sqrt{2^m}$, where t is the *trace of Frobenius.* A curve can be generated with a prescribed order using the complex multiplication method [118] or the curve order can be explicitly computed in binary curves using the approach due to Satoh, Skjernaa and Taguchi [119]. Non-supersingularity comes from the fact that t is not a multiple of the characteristic 2 of the underlying finite field [59].

The set of points $\{(x, y) \in E(\mathbb{F}_{2^m})\} \cup \{\mathcal{O}\}$ under the addition operation $+$ (chord and tangent) forms an additive group, with $\mathcal{O}$ as the identity element. Given an elliptic point $P \in E(\mathbb{F}_{2^m})$ and an integer k, the operation kP, called *point multiplication*, is defined by the addition of the point P to itself $k - 1$ times:

$$kP = \underbrace{P + P + \ldots + P}_{k-1 \text{ additions}}.$$

Public key cryptography protocols, such as the Elliptic Curve Diffie-Hellman key agreement [110] and the Elliptic Curve Digital Signature Algorithm [110], employ point multiplication as a fundamental operation; and their security is based on the difficulty of solving the Elliptic Curve Discrete Logarithm Problem (ECDLP). This problem consists in finding the discrete logarithm k given a point kP. Criteria for selecting suitable secure curves are a complex subject and a matter of much discussion. We adopt the well-known standard NIST curves as a conservative choice, but we refer the reader to [110] for further details on how to generate efficient curves where instances of the ECDLP are computationally hard.

We restrict the discussion to non-supersingular curves because supersingular curves are not suitable for elliptic curve cryptosystems based on the ECDLP problem [120]. However, supersingular curves are particularly of interest in applications of pairing-based protocols on WSNs [10].

2.4 The platform

The MICAz Mote sensor node is equipped with an ATmega128 8-bit processor clocked at 7.3728MHz. The program code is loaded from an 128KB EEPROM chip and runtime memory is stored in a 4KB RAM chip [106]. The ATmega128 processor is a typical RISC architecture with 32 registers, but six of them are special pointer registers. Since at least one register is needed to store temporary results or data loaded from memory, 25 registers are generally available for arithmetic. The instruction set is also reduced, as only 1-bit shift/rotate instructions are natively supported. Bitwise shifts by arbitrary amounts can then be implemented with combinations of shift/rotate instructions and other instructions. The processor pipeline has two stages and memory instructions always cause pipeline stalls. Arithmetic instructions with register operands cost 1 cycle and memory instructions or memory addressing cost 2 processing cycles [32]. Table 2.2 presents the instructions provided by the platform which can be used for the implementation of binary field arithmetic.

Instruction	Description	Use	Cost
`rsl`, `lsl`	Right/left 1-bit shift	Multi-precision 1-bit shift	1 cycle
`rol`, `ror`	Right/left 1-bit rotate	Multi-precision 1-bit shift	1 cycle
`swap`	Swap high and low nibbles	Shift by 4 bits	1 cycle
`bld`, `bst`	Bit load/store from/to flag	Shift by 7 bits	1 cycle
`eor`	Bitwise exclusive `OR`	Binary field addition	1 cycle
`ld`, `st`	Memory load/store	Read operands/write results	2 cycles
`adiw`, `sbiw`	Pointer arithmetic	Memory addressing	2 cycles

Table 2.2: Relevant instructions for the implementation of binary field arithmetic.

2.5 Algorithms for finite field arithmetic

In this section we will represent the elements of $\mathbb{F}_{2^m}$ using a polynomial basis. Let $f(z)$ be an irreducible binary trinomial or pentanomial of degree m. The elements of $\mathbb{F}_{2^m}$ are the binary polynomials of degree at most $m-1$. A field element $a(z) = \sum_{i=0}^{m-1} a_i z^i$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_1, a_0)$ of length m. In a software implementation in an 8-bit processor, the element a is stored as a vector of $n = \lceil m/8 \rceil$ bytes. The field operations in $\mathbb{F}_{2^m}$ can be implemented by common processor instructions, such as logical shifts ($\gg$,$\ll$) and addition modulo 2 (XOR, $\oplus$).

2.5.1 Multiplication

The computation of kP is the most time-consuming operation on ECC and this operation depends directly on the finite field arithmetic. In particular, a fast field multiplication is critical for the performance of ECC.

Two different strategies are commonly considered for the implementation of multiplication in $\mathbb{F}_{2^m}$. The first one consists in applying the Karatsuba's algorithm [121] to divide the multiplication in sub-problems and solve each problem independently by the following formula [59] (with $a(z) = A_1 z^{\lceil m/2 \rceil} + A_0$ and $b(z) = B_1 z^{\lceil m/2 \rceil} + B_0$):

$$c(z) = a(z) \cdot b(z) = A_1 B_1 z^m + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] z^{\lceil m/2 \rceil} + A_0 B_0.$$

Naturally, Karatsuba multiplication imposes some overhead for the divide and conquer steps. The second one consists in applying a direct algorithm like the López-Dahab (LD) binary field multiplication (Algorithm 2.1) [27]. In this algorithm, the precomputation window is usually chosen as $t = 4$ and the precomputation table T has size $|T| = 16(n+1)$, since each element $T[i]$ requires at most $n+1$ bytes to store the result of $u(z)b(z)$. Operand a is scanned from left to right and processed in groups of 4 bits. In an 8-bit processor, the algorithm is comprised by two phases, where the lower halves of bytes of a are processed in the first phase and the higher halves are processed in the second phase. These phases are separated by an intermediate shift which implements multiplication by z^t.

Conventionally, the series of additions involved in the LD multiplication are implemented through additions over subparts of a double-precision vector. In order to reduce the number of memory accesses employed during these additions, we employ a rotating register window. This window simulates the series of additions by accumulating consecutive writes into registers. After a final result is obtained in the lowest precision register, this value is written into memory and this register is free to participate as the highest precision register. Figure 2.1 shows a rotating register window with $n + 1$ registers. We modify the LD multiplication algorithm by integrating a rotating register window. The

Algorithm 2.1 López-Dahab multiplication in $\mathbb{F}_{2^m}$ [27].

```
Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..2n-1]$.
 1: Compute $T(u) = u(z)b(z)$ for all polynomials $u(z)$ of degree lower than $t$.
 2: $c[0 \ldots 2n-1] \leftarrow 0$
 3: for $k \leftarrow 0$ to $n-1$ do
 4:     $u \leftarrow a[k] \gg t$
 5:     for $j \leftarrow 0$ to $n$ do
 6:         $c[j+k] \leftarrow c[j+k] \oplus T(u)[j]$
 7:     end for
 8: end for
 9: $c(z) \leftarrow c(z)z^t$
10: for $k \leftarrow 0$ to $n-1$ do
11:     $u \leftarrow a[k] \mod 2^t$
12:     for $j \leftarrow 0$ to $n$ do
13:         $c[j+k] \leftarrow c[j+k] \oplus T(u)[j]$
14:     end for
15: end for
16: return $c$
```

result of this integration is referred as *LD multiplication with registers* and shown as Algorithm 2.2. Figure 2.2 presents this modification graphically. These descriptions of the algorithm assumes that n general-purpose registers are available for arithmetic. If this is not the case, (e.g. multiplication in $\mathbb{F}_{2^{233}}$ on this platform) the accumulation in the register window must be divided in different blocks in a multistep fashion and each block processed with a different rotating register window. A slight overhead is introduced between the processing of consecutive blocks because some registers must be written into memory and freed before they can be used in a new rotating register window.

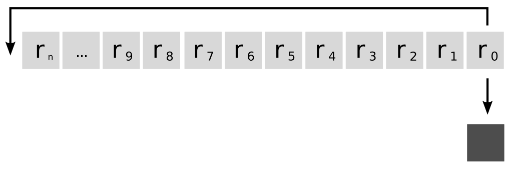


Figure 2.1: Rotating register window with $n + 1$ registers.

An additional suggested optimization is the separation of the precomputation table T in different blocks of 256 bytes, where each block is stored on a 256-byte aligned memory address. This optimization accelerates memory addressing because offsets lower than 256 can be computed by a simple 1-cycle addition instruction, avoiding expensive pointer

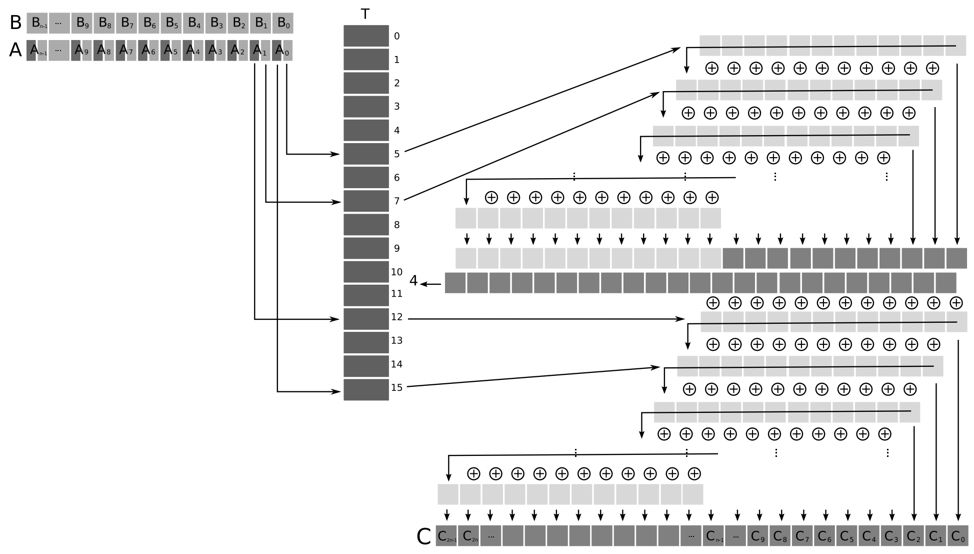


Figure 2.2: López-Dahab multiplication with registers of two field elements represented as n-byte vectors in an 8-bit processor.

Algorithm 2.2 Proposed optimization for multiplication in $\mathbb{F}_{2^m}$ using $n + 1$ registers.

```
Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..2n-1]$.
Note: $v_i$ denotes the vector of $n+1$ registers $(r_{i-1}, \ldots, r_0, r_n, \ldots, r_i)$.
 1: Compute $T(u) = u(z)b(z)$ for all polynomials $u(z)$ of degree lower than 4.
 2: Let $u_i$ be the 4 most significant bits of $a[i]$.
 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: $c \leftarrow ((r_{n-2}, \ldots, r_0, r_n) \mid\mid (c[n-1], \ldots, c[0])) \ll 4$
 8: Let $u_i$ be the 4 least significant bits of $a[i]$.
 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: $c[n \ldots 2n-1] \leftarrow c[n \ldots 2n-1] \oplus (r_{n-2}, \ldots, r_0, r_n)$
13: return $c$
```

arithmetic. Another optimization is to store the results of the first phase of the algorithm already shifted, eliminating some redundant memory reads to reload the intermediate result into registers for multi-precision shifting. A last optimization is the embedding of modular reduction at the end of the multiplication algorithm. This trick allows the reuse of values already loaded into registers to speed up modular reduction. The following analysis does not take these suggested optimizations into account.

Analysis of multiplication algorithms

Observing the fact that the more expensive instructions in the target platform are related to memory accesses, the behavior of different algorithms was analyzed to estimate their performance. This analysis traces the cost of different algorithms in terms of memory accesses (reads and writes) and arithmetic instructions (XOR).

Without considering partial multiplications, the Karatsuba algorithm in a binary field executes approximately $11n$ memory reads, $7n$ memory writes and $4n$ XOR instructions.

For LD multiplication, analysis shows that building the precomputation table requires n memory reads to obtain the values $b[i]$ and $|T|$ writes and $11n$ XOR instructions for filling the table. Inside each inner loop, the algorithm executes $2(n+1)$ memory reads, $n+1$ writes and $n+1$ XOR instructions. In each outer loop, the algorithm executes n memory accesses to read the values $a[k]$ and n iterations of the inner loop, totalizing $n+2n(n+1)$ reads, $n(n+1)$ writes and $n(n+1)$ XOR instructions. The logical shift of $c(z)$ computed at the intermediate stage requires $2n$ memory reads and writes. Considering the initialization of c, we have $3n+2(n+2n(n+1))$ memory reads, $|T|+2(2n)+2n(n+1)$ writes and $11n+2n(n+1)$ XOR instructions.

For the proposed optimization (Algorithm 2.2), building the precomputation table requires n memory reads to obtain the values $b[i]$ and $|T|$ writes and $11n$ XOR instructions for filling the table. Line 3 of the algorithm executes $n+1$ memory reads and 1 write on $c[0]$. Lines 4-6 execute $n+1$ memory reads, 1 write on $c[i]$ and $n+1$ XOR instructions, all this $n-1$ times. The intermediate shift executes n reads and $(2n)$ writes. Lines 9-11 execute $n+1$ memory reads, 1 read and write on $c[i]$ and $n+2$ XOR instructions, all this n times. The final operation costs n memory reads, writes and XOR instructions. The algorithm thus requires a total of $3n+n(n+1)+n(n+2)$ reads, $|T|+n+2n+2n$ writes and $11n+(n-1)(n+1)+n(n+2)+n$ XOR instructions.

Table 2.3 presents the costs associated with memory operations for LD multiplication, LD with registers multiplication and Karatsuba multiplication. Table 2.4 presents approximate costs of the algorithms in terms of executed memory instructions for the fields $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$.

We can see from Table 2.3 that the number of memory accesses for LD with registers is drastically reduced in comparison with the original algorithm, reducing the number of reads by half and the number of writes by a quadratic factor. The comparison between LD with registers and Karatsuba+LD with registers favors the first (lower number of writes) on both finite fields. One problem with this analysis is that it assumes that the

	Number of instructions in terms of vectors of n bytes		
Method	Reads	Writes	XOR
López-Dahab	$4n^2 + 9n$	$\lvert T\rvert + 2n^2 + 6n$	$2n^2 + 13n$
LD with registers	$2n^2 + 6n$	$\lvert T\rvert + 5n$	$2n^2 + 14n - 1$
Karatsuba	$11n + 3M(\lceil n/2 \rceil)$	$7n + 3M(\lceil n/2 \rceil)$	$4n + 3M(\lceil n/2 \rceil)$

Table 2.3: Costs in number of executed instructions for the multiplication algorithms in $\mathbb{F}_{2^m}$. $M(x)$ denotes the cost of a multiplication algorithm which multiplies two x-byte vectors.

	$n = 21$			$n = 30$		
Method	Reads	Writes	XOR	Reads	Writes	XOR
López-Dahab	1953	1452	1155	3870	2476	2190
LD with registers	1071	457	1175	1980	646	2219
Karatsuba+LD	1980	1647	1239	3310	2518	1984
Karatsuba+LD with registers	1155	888	1269	1898	1134	2025

Table 2.4: Costs in number of executed instructions for the multiplication algorithms in $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$. The Karatsuba algorithm in $\mathbb{F}_{2^{233}}$ executes two instances of cost $M(15)$ and one instance of cost $M(14)$ to better approximate the results.

processor has at least n general-purpose registers available for arithmetic. This is not true in $\mathbb{F}_{2^{233}}$, because the algorithm requires 31 registers for a full rotating register window. The decision between a multistep implementation of LD with registers and Karatsuba+LD with registers will depend on the actual implementation of the algorithms.

2.5.2 Modular reduction

The NIST irreducible polynomial for the finite field $\mathbb{F}_{2^{163}}$, $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$, allows a fast modular reduction algorithm. Algorithm 2.3 [116] presents an adaptation of this algorithm for 8-bit processors. In this algorithm, reducing a digit $c[i]$ of the upper half of the vector c requires six memory accesses to read and write $c[i]$ on lines 3-5. Four of them are redundant because ideally we only need to read and write $c[i]$ once. We eliminate these redundant accesses by employing a rotating register window of three registers which accumulate writes into registers before a final result can be written into memory. This optimization is given in Algorithm 2.4 along with the substitution of some bitwise shifts which are expensive in this platform for cheaper ones. Since the processor only supports 1-bit and 4-bit shifts natively, we further replace the various expensive shifts in the accumulate function R by table lookups on 256-byte tables. These tables are stored on 256-byte aligned memory addresses to speed up memory addressing. The new version of the accumulate function is depicted in Algorithm 2.5.

Algorithm 2.3 Fast modular reduction by $f(z) = z^{163} + z^7 + z^6 + z^3 + 1$.

Input: $c(z) = c[0..40]$.
Output: $c(z) \bmod f(z) = c[0..20]$.

```
 1: for $i \leftarrow 40$ downto 21 do
 2:     $t \leftarrow c[i]$
 3:     $c[i-19] \leftarrow c[i-19] \oplus (t \gg 4) \oplus (t \gg 5)$
 4:     $c[i-20] \leftarrow c[i-20] \oplus (t \ll 4) \oplus (t \ll 3) \oplus t \oplus (t \gg 3)$
 5:     $c[i-21] \leftarrow c[i-21] \oplus (t \ll 5)$
 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: $c[1] \leftarrow c[1] \oplus (t \gg 1) \oplus (t \gg 2)$
10: $c[20] \leftarrow c[20] \wedge$ 0x07
11: return $c$
```

Algorithm 2.4 Fast modular reduction in $\mathbb{F}_{2^{163}}$ with rotating register window.

Input: $c(z) = c[0..40]$.
Output: $c(z) \bmod f(z) = c[0..20]$.
Note: The accumulate function $R(r_0, r_1, r_2, t)$ executes:

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

```
 1: $r_b \leftarrow 0$, $r_c \leftarrow 0$
 2: for $i \leftarrow 40$ downto 25 by 3 do
 3:     $R(r_b, r_c, r_a, c[i])$, $c[i-19] \leftarrow c[i-19] \oplus r_b$
 4:     $R(r_c, r_a, r_b, c[i-1])$, $c[i-20] \leftarrow c[i-20] \oplus r_c$
 5:     $R(r_a, r_b, r_c, c[i-2])$, $c[i-21] \leftarrow c[i-21] \oplus r_a$
 6: end for
 7: $R(r_b, r_c, r_a, c[22])$, $c[3] \leftarrow c[3] \oplus r_b$
 8: $R(r_c, r_a, r_b, c[21])$, $c[2] \leftarrow c[2] \oplus r_c$
 9: $r_a \leftarrow c[1] \oplus r_a$
10: $r_b \leftarrow c[0] \oplus r_b$
11: $t \leftarrow c[20]$
12: $c[20] \leftarrow t \wedge$ 0x07
13: $t \leftarrow t \gg 3$
14: $c[0] \leftarrow r_b \oplus (t \ll 7) \oplus (t \ll 6) \oplus (t \ll 3) \oplus t$
15: $c[1] \leftarrow r_a \oplus (t \gg 1) \oplus (t \gg 2)$
16: return $c$
```

Algorithm 2.5 Optimized version of the accumulate function R.

```
Input: $r_0, r_1, r_2, t$.
Output: $r_0, r_1, r_2$.
1: $r_0 \leftarrow r_0 \oplus T_0[t]$
2: $r_1 \leftarrow r_1 \oplus T_1[t]$
3: $r_2 \leftarrow t \ll 5$
```

For the NIST irreducible polynomial in $\mathbb{F}_{2^{233}}$ on 8-bit processors, we present Algorithm 2.6, a direct adaptation of the standard algorithm. This algorithm only executes 1-bit or 7-bit shifts. These two shifts can be translated efficiently to the processor instruction set, because 1-bit shifts are supported natively and 7-bit shifts can be emulated efficiently. Hence lookup tables are not needed and the only optimization made during implementation of Algorithm 2.6 was complete unrolling of the main loop and straightforward elimination of consecutive redundant memory accesses.

Algorithm 2.6 Fast modular reduction by $f(z) = z^{233} + z^{74} + 1$.

```
Input: $c(z) = c[0..58]$.
Output: $c(z) \bmod f(z) = c[0..29]$.
1: for $i \leftarrow 58$ downto $32$ by $2$ do
2:     $t_0 \leftarrow c[i]$
3:     $t_1 \leftarrow c[i-1]$
4:     $c[i-19] \leftarrow c[i-19] \oplus (t_0 \gg 7)$
5:     $c[i-20] \leftarrow c[i-20] \oplus (t_0 \ll 1) \oplus (t_1 \gg 7)$
6:     $c[i-21] \leftarrow c[i-21] \oplus (t_1 \ll 1)$
7:     $c[i-29] \leftarrow c[i-29] \oplus (t_0 \gg 1)$
8:     $c[i-30] \leftarrow c[i-30] \oplus (t_0 \ll 7) \oplus (t_1 \gg 1)$
9:     $c[i-31] \leftarrow c[i-31] \oplus (t_1 \ll 7)$
10: end for
11: $t_0 \leftarrow c[30]$
12: $c[0] \leftarrow c[0] \oplus (t_0 \ll 7)$
13: $c[1] \leftarrow c[1] \oplus (t_0 \gg 1)$
14: $c[10] \leftarrow c[10] \oplus (t_0 \ll 1)$
15: $c[11] \leftarrow c[11] \oplus (t_0 \gg 7)$
16: $t_0 \leftarrow c[29] \gg 1$
17: $c[0] \leftarrow c[0] \oplus t_0$
18: $c[9] \leftarrow c[9] \oplus (t_0 \ll 2)$
19: $c[10] \leftarrow c[10] \oplus (t_0 \gg 6)$
20: $c[29] \leftarrow c[29] \wedge$ 0x01
21: return $c$
```

Analysis of modular reduction algorithms

As pointed by Seo et al. [116], Algorithm 2.3 executes many redundant memory accesses: 4 memory reads and 3 writes during each loop iteration and additional 4 reads and 3 writes on the final step, which sum up to 88 reads and 66 writes. The proposed optimization reduces the number of memory operations to 43 reads and 23 writes. Despite Algorithm 2.5 being specialized for the chosen polynomial, the register window technique can be applied to any irreducible polynomial with the non-null coefficients located in the first word. The implementation of Algorithm 2.6 also reduces the number of memory accesses, since a standard implementation executes 122 reads and 92 writes while our implementation executes 92 memory reads and 62 writes.

2.5.3 Squaring

The square of a finite field element $a(z) \in \mathbb{F}_{2^m}$ is given by $a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i} = a_{m-1}z^{2m-2} + \cdots + a_2z^4 + a_1z^2 + a_0$. The binary representation of $a(z)^2$ can be computed by inserting a “0” bit between each pair of successive bits on the binary representation of $a(z)$ and accelerated by introducing a 16-byte lookup table.

If modular reduction is computed in a separate step, reduntant memory operations are required to store the squaring result and reload this result for reduction. This can be improved by embedding the modular reduction step directly into the squaring algorithm. This way, the lower half of the digit vector a is expanded in the usual fashion and the upper half digits are expanded and immediately reduced. If modular reduction of a single byte requires expensive shifts, additional lookup tables can be used to store the expanded bytes already reduced. This is illustrated in Algorithm 2.7 which computes squaring in $\mathbb{F}_{2^{163}}$ using the same small rotating register window as Algorithm 2.5 and three additional 16-byte lookup tables T_0, T_1 and T_2. For squaring in $\mathbb{F}_{2^{233}}$, we also combine byte expansion of the digit vector's lower half with Algorithm 2.6 for fast reduction.

2.5.4 Inversion

For inversion in $\mathbb{F}_{2^m}$ we implemented the Extended Euclidean Algorithm for polynomials [59]. Since this algorithm requires flexible left shifts by arbitrary amounts, we implemented six dedicate shifting functions to shift a binary field element by every amount possible for an 8-bit processor. The core of a multi-precision left shift algorithm is the sequence of instructions which receives as input the amount to shift i, a register r and a carry register rc storing the bits shifted out in the last iteration; and produce $(r \ll i) \oplus rc$ as output and $r \gg (8-i)$ as new carry. Table 2.5 lists the required instructions and costs in cycles for shifting a single byte in each of the implemented multi-precision shifts by

i bits. Each instruction in the table cost 1 cycle, thus the cost to compute the core of a multi-precision left shift by i bits is just the number of rows in the i-th row of the table.

Algorithm 2.7 Squaring in $\mathbb{F}_{2^{163}}$.

Input: $a(z) = a[0..20]$.
Output: $c(z) = a(z)^2 \bmod f(z)$.
Note: The accumulate function $R(r_0, r_1, r_2, t)$ executes:
$r_0 \leftarrow r_0 \oplus T_0[t], r_1 \leftarrow r_1 \oplus T_1[t], r_2 \leftarrow r_2 \oplus T_2[t]$

```
1: For each 4-bit combination $u$, $T(u) = (0, u_3, 0, u_2, 0, u_1, 0, u_0)$.
2: for $i \leftarrow 0$ to 9 do
3:     $c[2i] \leftarrow T(a[i] \wedge \texttt{0x0F})$
4:     $c[2i+1] \leftarrow T(a[i] \gg 4)$
5: end for
6: $c[20] \leftarrow T(a[10] \wedge \texttt{0x0F})$
7: $r_b \leftarrow 0$, $r_c \leftarrow 0$, $j \leftarrow 20$
8: $t_0 \leftarrow a[20] \wedge \texttt{0x0F}$
9: $R(r_b, r_c, r_a, t_0)$, $c[21] \leftarrow r_b$
10: for $i \leftarrow 19$ downto 13 by 3 do
11:     $a_o \leftarrow a[i]$, $t_0 \leftarrow a_0 \gg 4$, $t_1 \leftarrow a_0 \wedge \texttt{0x0F}$
12:     $R(r_c, r_a, r_b, t_0)$, $c[j] \leftarrow c[j] \oplus r_c$
13:     $R(r_a, r_b, r_c, t_1)$, $c[j-1] \leftarrow c[j-1] \oplus r_a$
14:     $a_0 \leftarrow a[i-1]$, $t_0 \leftarrow a_0 \gg 4$, $t_1 = a_0 \wedge \texttt{0x0F}$
15:     $R(r_b, r_c, r_a, t_0)$, $c[j-2] \leftarrow c[j-2] \oplus r_b$
16:     $R(r_c, r_a, r_b, t_1)$, $c[j-3] \leftarrow c[j-3] \oplus r_c$
17:     $a_0 = a[i-2]$, $t_0 = a_0 \gg 4$, $t_1 = a_0 \wedge \texttt{0x0F}$
18:     $R(r_a, r_b, r_c, t_0)$, $c[j-4] \leftarrow c[j-4] \oplus r_a$
19:     $R(r_b, r_c, r_a, t_1)$, $c[j-5] \leftarrow c[j-5] \oplus r_b$
20:     $j \leftarrow j - 6$
21: end for
22: $t_0 = a[10] \gg 4$
23: $R(r_c, r_a, r_b, t_0), c[2] \leftarrow c[2] \oplus r_c$
24: $r_a \leftarrow c[1] \oplus r_a, r_b \leftarrow c[0] \oplus r_b$
25: $t \leftarrow c[21]$
26: $r_a \leftarrow r_a \oplus t \oplus (t \ll 3) \oplus (t \ll 4) \oplus (t \gg 3)$
27: $r_b \leftarrow r_b \oplus (t \ll 5)$
28: $t \leftarrow c[20]$
29: $c[20] \leftarrow t \wedge \texttt{0x07}$
30: $t \leftarrow t \gg 3$
31: $c[0] \leftarrow r_b \oplus (t \ll 7) \oplus (t \ll 6) \oplus (t \ll 3) \oplus t$
32: $c[1] \leftarrow r_a \oplus (t \gg 1) \oplus (t \gg 2)$
33: return $c$
```

i	**Intructions**
1	`rol r`

2	`clr rt`
	`lsl r`
	`rol rt`
	`lsl r`
	`rol rt`
	`eor r, rc`
	`mov rc, rt`

3	`clr rt`
	`lsl r`
	`rol rt`
	`lsl r`
	`rol rt`
	`lsl r`
	`rol rt`
	`eor r, rc`
	`mov rc, rt`

4	`swap r`
	`mov rt, r`
	`andi r, 0xF0`
	`andi rt, 0x0F`
	`eor r, rc`
	`mov rc, rt`

5	`swap r`
	`mov rt, r`
	`andi r, 0xF0`
	`andi rt, 0x0F`
	`lsl r`
	`rol rt`
	`eor r, rc`
	`mov rc, rt`

6	`bst rt, 0`
	`bld r, 6`
	`bst rt, 1`
	`bld r, 7`
	`lsr rt`
	`lsr rt`
	`eor r, rc`
	`mov rc, rt`

7	`bst rt, 0`
	`bld r, 7`
	`lsr rt`
	`eor r, rc`
	`mov rc, rt`

Table 2.5: Processor instructions used to efficiently implement multi-precision left shifts by i bits. The input register is `r`, the carry register is `rc` and a temporary register is `rt`. When $i = 1$, `rc` is represented by the carry processor flag.

2.6 Algorithms for elliptic curve arithmetic

We have selected fast algorithms for elliptic curve arithmetic in three situations: multiplying a random point P by a scalar k, multiplying the generator G by a scalar k and simultaneously multiplying two points P and Q by scalars k and l to obtain $kP + lQ$. Our implementation uses mixed addition with projective coordinates [60], given that the ratio of inversion to multiplication is 16.

For multiplying a random point by a scalar, we choose Solinas' *τ-adic non-adjacent form* (TNAF) representation [63] with $w = 4$ for Koblitz curves (4-TNAF method with 4 precomputation points) and the method due to López and Dahab [122] for random binary curves. Solinas' algorithm explores the optimizations provided by Koblitz curves and accelerates the computation of kP by substituting point doublings for applications of the efficiently computable endomorphism based on the Frobenius map $\tau(x, y) = (x^2, y^2)$. The method due to López and Dahab does not use precomputation, its execution time is constant and each iteration of the algorithm executes the same number of operations, independently of the bit pattern in k [59].

For multiplying the generator, we employ the same 4-TNAF method for Koblitz curves; and for generic curves, we employ the Comb method [64] with 16 precomputed points. Precomputed tables for the generator are stored in ROM memory to reduce RAM consumption. Larger precomputed tables can be used if program size is not an issue.

For simultaneous multiplication, we implement the interleaving method with 4-TNAFs for Koblitz curves and the interleaving of 4-NAFs with integers represented in *non-adjacent form* (NAF) for generic curves [65]. The same table built for multiplying the generator is used during simultaneous multiplication in Koblitz curves when point P or Q is the generator G. An additional small table of 4 points is precomputed for the generator and stored in ROM to provide the same situation with generic curves.

2.7 Implementation results

The compiler and assembler used is the GCC 4.1.2 suite for ATmega128 with optimization level `-O2`. The timings were measured with the software AVR Studio 4.14 [123]. This tool is a cycle-accurate simulator frequently used to prototype software for execution on the target platform. We have written a specialized library containing the software implementations.

Finite field arithmetic

The algorithms for squaring, multiplication, modular reduction and inversion in the finite field were implemented in the C language and Assembly. Table 2.6 presents the costs measured in cycles of each implemented operation in $\mathbb{F}_{2^{163}}$ and $\mathbb{F}_{2^{233}}$. Since the platform does not have cache memory or out-of-order execution, the finite field operations always cost the same number of cycles and the timings were taken exactly once, except for inversion. The timing for inversion was taken as the average of 50 timings measured on consecutive executions of the algorithm.

	$m = 163$		$m = 233$	
Algorithm	C language	*Assembly*	C language	*Assembly*
Squaring	629	430	908	463
Modular Squaring	1154	570	1340	956
LD Mult. with registers	13838	4508	–	8314
LD Mult. (new variant)	9738	–	18028	–
Karatsuba+LD with registers	12246	6968	25850	9261
Modular reduction	606	430	911	620
Inversion	243790	81365	473618	142986

Table 2.6: Timings in cycles for arithmetic algorithms in $\mathbb{F}_{2^m}$.

From Table 2.6, $m = 163$, we can observe that in the C language implementation, Karatsuba+LD with registers multiplication is more efficient than the direct application of LD with registers multiplication. This contradicts the preliminary analysis based on the number of memory accesses executed by each algorithm. This can be explained by the fact that the LD with registers multiplication uses 21 of the 32 general-purpose registers to store intermediate results during multiplication. Several additional registers are also needed to store memory addresses and temporary variables for arithmetic operations. The inefficiency found is thus originated from the difficulty of the C compiler to maintain all intermediate values on registers. To confirm this limitation, a new variant of LD with registers multiplication which reduces the number of temporary variables needed was also implemented. This variant processes 32 bits of the operand in each interaction compared to the original version of LD multiplication which processes 4 bits in each interaction. The new variant reduces the number of memory accesses while keeping a smaller number of temporary variables and thus exhibits the expected performance. For the squaring algorithm, we can see that embedding the modular reduction step reduces the cost of modular squaring significantly compared with the sequential execution of squaring plus modular reduction. Table 2.6, $m = 233$, shows that the Karatsuba algorithm in $\mathbb{F}_{2^{233}}$ indeed does not improve performance over the multistep implementation of LD with registers multiplication, even if the processor does not have enough registers to store the full rotating register window. The Assembly implementations demonstrate the compiler inefficiency in generating optimized code and allocating resources for the target platform, showing considerably faster timings.

Elliptic curve arithmetic

Point multiplication was implemented on elliptic curves standardized by NIST. Table 2.7 presents the execution time of the multiplication of a random point P by a random integer k of 163 or 233 bits, with the underlying finite field arithmetic implemented in C or Assembly. In each of the programming languages, the fastest field multiplication algorithm is used. The results were computed by the arithmetic mean of the timings measured on 50 consecutive executions of the algorithm.

	C language			*Assembly*		
Curve	kG	kP	$kP + lQ$	kG	kP	$kP + lQ$
NIST-K163 (Koblitz)	0.56	0.67	1.24	0.29	0.32	0.60
NIST-B163 (Generic)	0.77	1.55	2.21	0.37	0.74	1.04
NIST-K233 (Koblitz)	1.26	1.48	2.81	0.66	0.73	1.35
NIST-B233 (Generic)	1.94	3.90	5.35	0.94	1.89	2.52

Table 2.7: Timings in seconds for point multiplication.

Table 2.8 compares the performance of the proposed implementation with TinyECCK [116] and the work of Kargl et al. [107], the previously fastest binary curves implementation in C and Assembly published for this platform. For the C implementation, we achieve faster timings on all finite field arithmetic operations with improvements over 50%. For the Assembly implementation, we obtain speed improvements on field squaring and multiplication and exactly the same timing for modular reduction, but the polynomial used by Kargl et al.[107] is a trinomial carefully selected to support a faster modular reduction algorithm. The computation of kP on Koblitz curves implemented in C language was 41% faster than TinyECCK. By choosing the López-Dahab point multiplication algorithm with generic curves implemented in Assembly, we achieve a timing 11% faster than [107] while satisfying the timing-resistant property. If we relax this condition, we obtain a point multiplication 61% faster in Assembly by using Solinas' method. Comparing our Assembly implementation with TinyECCK and [107] with the same curve parameters, we achieve a 72% speedup and an 11% speedup for point multiplication, respectively.

	Proposed	**TinyECCK**	**Proposed**	**Kargl et al. [107]**
Algorithm	C language	C language	Assembly	Assembly
Modular Squaring	1154 c	2729 c	570 c	663
Multiplication	9738 c	19670 c	4508 c	5057 c
Modular reduction	606 c	1904 c	430 c	433 c
Inversion	243790 c	539132 c	81365 c	–
kP on Koblitz	0.67 s	1.14 s	0.32 s	–
kP on Generic	1.55 s	–	0.74 s	0.83 s

Table 2.8: Comparison between different implementations. The timings are presented in cycles (c) or seconds (s) on a 7.2838MHz device.

The fastest time for point multiplication previously published for this platform at the 160-bit security level was 0.745 second [108]. Compared to this implementation, which uses prime fields, the proposed optimizations result in a point multiplication 57% faster.

The implemented optimizations allow performance gains but provoke a collateral effect on memory consumption. Table 2.9 presents memory requirements for code size and RAM memory for the different implementations at the 160-bit security level. We can also observe that Assembly implementations are responsible for a significant expansion in program code size.

Cryptographic protocols

We now illustrate the performance obtained by our efficient implementation with some executions of cryptographic protocols for key agreement and digital signatures. Key agreement is employed in sensor networks for establishing symmetric keys which can be used for encryption or authentication. Digital signatures are employed for communication

	ROM memory	Static RAM	Stack RAM
Proposed (Koblitz) – C	22092	1028	1207
Proposed (Koblitz) – C+Assembly	25802	1732	1207
Proposed (Generic) – C	12848	881	682
Proposed (Generic) – C+Assembly	16218	1585	682
TinyECCK (C-only)	5592	–	618
Kargl et a. (C+Assembly) [107]	11264	–	–

Table 2.9: Cost in *bytes* of memory for implementations of scalar multiplication of a random point at the 160-bit security level.

between the sensor nodes and the base stations where data must be made available to multiple applications and users [124]. For key agreement between nodes, we implemented the Elliptic Curve Diffie & Hellman (ECDH) protocol [110], and for digital signatures, we implemented the Elliptic Curve Digital Signature Algorithm (ECDSA) [110]. We assume that public and private keys are generated and loaded into the nodes before the deployment of the sensor network. Hence timings for key generation and public key authentication are not presented or considered. Table 2.10 presents the timings for the ECDH protocol and Table 2.11 presents the timings for the ECDSA protocol, using the choice of algorithms discussed in Section 2.6. Results on these tables pose an interesting decision between deploying generic binary curves on the lower security level or deploying special curves on the higher security level.

	C language			*Assembly*		
Curve	Time	ROM	RAM	Time	ROM	RAM
NIST-K163	0.74	28.3	2.2	0.39	32.0	2.8
NIST-B163	1.62	24.0	1.1	0.81	27.8	1.9
NIST-K233	1.55	31.0	2.9	0.80	38.6	3.7
NIST-B233	3.97	26.9	1.5	1.96	34.6	2.2

Table 2.10: Timings for the ECDH protocol execution. Timings are given in seconds and ROM memory or Static+Stack RAM consumption are given in KB.

2.8 Conclusions

Despite several years of intense research, security and cryptography on WSNs still face several open problems. In this work, we presented efficient implementations of binary field algorithms such as squaring, multiplication, modular reduction and inversion. These implementations take into account the characteristics of the target platform (the MICAz Mote) to develop optimizations, specifically: (i) the cost of memory addressing; (ii) the

	C language			*Assembly*		
Curve	Time (S + V)	ROM	RAM	Time (S + V)	ROM	RAM
NIST-K163	0.67 + 1.23	31.8	2.9	0.36 + 0.63	35.3	3.7
NIST-B163	0.87 + 2.17	29.6	2.1	0.45 + 1.05	33.2	2.8
NIST-K233	1.46 + 2.76	34.6	3.1	0.78 + 1.39	42.2	3.8
NIST-B233	2.09 + 5.25	32.8	2.3	1.04 + 2.55	40.4	3.1

Table 2.11: Timings for the ECDSA protocol execution. Timings for signature (S) and verification (V) are given in seconds and ROM memory or Static+Stack RAM consumption are given in KB.

cost of memory instructions; (iii) the limited flexibility of bitwise shift instructions. We obtain the fastest binary field arithmetic implementations in C and Assembly published for the target platform. Significant performance benefits where achieved by the Assembly implementation, resulting from fine-grained resource allocation and instruction selection. These optimizations produced a point multiplication at the 160-bit security level under $\frac{1}{3}$ of a second, an improvement of 72% compared to the best implementation of a Koblitz curve previously published and an improvement of 61% compared to the best implementation of binary curves. When compared to the best implementation of prime curves, we obtain a performance gain of 57%. We also presented the first timings of elliptic curves at the higher 233-bit security level. For both security levels, we illustrate the performance obtained with executions of key agreement and digital signature protocols. In particular, a key agreement can be computed in under 0.40 second at the 163-bit security level and under 0.80 second at the 233-bit security level. A digital signature can be computed and verified in 1 second at the 163-bit security level and in 2.17 seconds at the 233-bit security level. We hope that our results can increase the efficiency and viability of elliptic curve cryptography on wireless sensor networks.

Acknowledgements

We would like to thank the referees for their valuable comments and suggestions. Diego F. Aranha is supported by FAPESP, grant no. 2007/06950-0. Julio López and Ricardo Dahab are partially supported by CNPq and FAPESP research grants.

Chapter 3

Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets

Diego F. Aranha, Julio López and Darrel Hankerson

Abstract

In this paper we describe an efficient software implementation of characteristic 2 fields making extensive use of vector instruction sets commonly found in desktop processors. Field elements are represented in a split form so performance-critical field operations can be formulated in terms of simple operations over 4-bit sets. In particular, we detail techniques for implementing field multiplication, squaring, square root extraction and present a constant-memory lookup-based multiplication strategy. Our representation makes extensive use of the *parallel table lookup* (PTLU) instruction recently introduced in popular desktop platforms and follows the trend of accelerating implementations of cryptography through PTLU-style instructions. We present timings for several binary fields commonly employed for curve-based cryptography and illustrate the presented techniques with executions of the ECDH and ECDSA protocols over binary curves at the 128-bit and 256-bit security levels standardized by NIST. Our implementation results are compared with publicly available benchmarking data.

Publication

This work was originally published in the Proceedings of the 1st International Conference on Cryptology and Information Security (LATINCRYPT 2010) available at `http://www.springer.com/computer/security+and+cryptology/book/978-3-642-14711-1`.

3.1 Introduction

Arithmetic in binary fields has significant cryptographic interest and finds several applications such as providing the underlying layer for arithmetic in elliptic curves – among them the highly efficient Koblitz family of anomalous binary curves [125] – and building blocks for the construction of symmetric ciphers [126], bilinear maps [1] and post-quantum cryptographic schemes [127, 128]. At the same time, modern processors are increasingly receiving new parallelism resources in the form of advanced vector processing units. Employing these resources in an efficient way is crucial to improve the performance of binary field arithmetic and consequently the performance of several cryptographic primitives.

As the main contribution, this work presents a high-speed software implementation of binary field arithmetic particularly appropriate for vector processors. Field arithmetic is expressed in terms of operations easily translated to contemporary vector instruction sets with a clear emphasis in arithmetic of 4-bit granularity to make proper use of the recently introduced powerful *parallel table lookup* (PTLU) instructions [129]. The presented techniques are compatible with the most efficient algorithms known for binary field arithmetic using a polynomial basis [27, 130] and with several of the implementation optimizations proposed in the literature [102]. These algorithms and optimizations can thus be seen in a new light under the framework we develop. Using the new PTLU instructions, we derive a new implementation strategy for binary field multiplication entirely based on table lookups. This strategy does not precompute tables and consequently consumes less memory than standard approaches: a single table of constants must be kept so memory consumption apart from space for temporary results is constant. The efficiency of our techniques and corresponding implementations is illustrated with performance figures for arithmetic in fields ranging from $\mathbb{F}_{2^{113}}$ to $\mathbb{F}_{2^{1223}}$ defined with square root friendly polynomials or NIST standard polynomials. Timings are also provided for key agreement and signature protocols at the two security levels adopted as standard for secure elliptic curve cryptography by the Brazilian National Public Key Infrastructure. We also present timings for curves supported by the eBACS [90] benchmarking project. The target platforms are several versions of the Intel Core microarchitecture.

The results of this work can improve the performance of cryptographic primitives employing binary field arithmetic in vector processors. The introduced techniques may also be important for exploring trade-offs occurring in the design of hardware arithmetic units or to optimize software implementations of other small-characteristic fields such as the ternary fields used in pairing-based cryptography [1]. Furthermore, our results follow the recent trend of optimizing implementations of cryptography through PTLU instructions [129, 131] and pose an interesting question about which operations should be supported by vector arithmetic units from a cost-benefit point of view.

This paper is organized as follows. Section 3.2 presents target platform characteristics. Section 3.3 presents our formulation and the techniques employed for efficient binary field arithmetic. Section 3.4 discusses experimental results for field and elliptic curve arithmetic and comparison with related work. The final section concludes the paper.

3.2 Platform Model

We assume that the target platform is equipped with a set of vector instructions, also called SIMD (Single Instruction, Multiple Data) because they operate in several data objects simultaneously. Currently, the most popular SIMD instruction sets are the Intel Streaming SIMD Extensions [50] and the AltiVec extensions introduced by Apple and IBM in the Power architecture specification [132]. Present technology provides instructions for orthogonal manipulation of 8, 16, 32 or 64-bit objects stored inside 128-bit architectural registers, but recently announced improvements in the form of Intel AVX and AMD SSE5 [133] extensions will support 256-bit registers in the future along with new operations like a native binary field multiplier [87].

To abstract the specific details of the underlying platform, vector instructions will be represented as mnemonics denoting a subset of operations supported in most instruction sets. Table 3.1 presents the mnemonics and the corresponding instructions in the SSE and AltiVec families, and thus shows the generality of our platform model. Experimental results will be provided only for SSSE3-capable Core 2/i7 platforms, however. In Table 2.2, memory access operations assume that memory addresses are always aligned in 16-byte boundaries so the faster load/store instructions can be used. Mnemonics for memory operations are reserved for loading/storing vector registers from/to word arrays. Bitwise shift instructions do not propagate bits between contiguous 32/64-bit data objects, requiring additional shifts and additions to be used as 128-bit bitwise shifts. Bytewise shift instructions (i.e. the shift amount is a multiple of 8) work across an entire vector register. We explicitly differentiate between bitwise and bytewise shifts because bytewise shifts may have smaller latencies in some vector instruction sets. Note that conventional shift mnemonics ($\ll, \gg$) are reserved for bitwise shifts of 64-bit words. Interleaving instructions alternately take bytes from the lower or higher parts of two registers to produce the output register. Two powerful instructions are discussed in more detail:

Memory alignment instructions extract a 128-bit section from the concatenation of two 128-bit registers, working as a fast shift with propagation of shifted out bytes between two vector registers.

Byte shuffling instructions take as inputs registers of bytes $r_a = (a_0, a_1, \ldots, a_{15})$ and $r_b = (b_0, b_1, \ldots, b_{15})$ and produce as result a permuted vector represented by the

16-byte register $r_c = (a_{b_0 \bmod 16}, a_{b_1 \bmod 16}, \ldots, a_{b_{15} \bmod 16})$. An often-missed use of these instructions is to perform 16 simultaneous lookups in a 16-byte lookup table, working as a legitimate PTLU instruction. This can be easily done by storing the lookup table in r_a and the lookup indexes in r_b and allows one to efficiently evaluate any function with 4-bit input and 8-bit output in parallel.

Table 3.1: Relevant vector instructions for the implementation of binary field arithmetic.

Mnemonic	**Description**	**SSE**	**AltiVec**
load,*store*	Memory load/store	`MOVDQA`	`LVX`
$\ll_{\nmid 8}, \gg_{\nmid 8}$	32/64-bit bitwise shifts	`PSLLQ`,`PSRLQ`	`VSLW`,`VSRW`
$\ll_8, \gg_8$	128-bit bytewise shift	`PSLLDQ`,`PSRLDQ`	`VPERM`
$\oplus, \wedge, \vee$	Bitwise `XOR`,`AND`,`OR`	`PXOR`,`PAND`,`POR`	`VAND`,`VOR`,`VXOR`
interlo,*interhi*	Byte interleaving	`PUNPCKLBW/HBW`	`VMRGLB`,`VMRGHB`
$\lhd$	Memory alignment	`PALIGNR`	`LVSL+VPERM`,`LVSR+VPERM`
shuffle,*lookup*	Byte shuffling	`PSHUFB`	`VPERM`

There is a visible recent effort from processor manufacturers to increase the performance and flexibility of shuffle instructions. Intel introduced a *Super Shuffle Engine* in the Core 2 45nm microarchitecture to reduce the latency of the `PSHUFB` instruction from 3 cycles to 1 cycle and doubled the throughput of this instruction in the Core i7 microarchitecture. AMD plans to introduce a new permutation instruction capable of operating at bit level with opcode `PPERM` in the upcoming SSE5 instruction set.

3.3 Binary Field Representation and Arithmetic

In this section we will represent the elements of the binary field $\mathbb{F}_{2^m}$ using a polynomial basis. Let $f(z)$ be an irreducible binary polynomial of degree m. The elements of $\mathbb{F}_{2^m}$ are the binary polynomials of degree at most $m-1$. A field element $a(z) = \sum_{i=0}^{m-1} a_i z^i$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_1, a_0)$ of length m. In a software implementation, these bit coefficients are typically packed and stored in a compact array $(a[0], \ldots, a[n-1])$ of n W-bit words, where W is the word size of the processor. For simplicity, it is assumed that n is even. We will instead use a split representation where a polynomial $a \in \mathbb{F}_{2^m}$ is divided into two polynomials:

$$a_L = \sum_{\substack{0 \le i < m, \\ 0 \le i \bmod 8 \le 3}} a_i z^i, \quad a_H = \sum_{\substack{0 \le i < m, \\ 4 \le i \bmod 8 \le 7}} a_i z^{i-4},$$

where a_L stores the low-order 4-bits of the contiguous bytes storing a in memory, and a_H stores the high-order 4-bits of the contiguous bytes storing a in memory. Using this representation, $a(z)$ can be simply written as:

$$a(z) = a_H(z)z^4 + a_L(z).$$

This representation allows performance-critical field operations to be formulated in terms of operations in groups of 4 bits (*nibbles*), taking advantage of the 4-bit granularity PTLU instructions. To minimize memory consumption, the algorithms will always receive the operands in the compact representation, convert them to the split representation using simple bit masks and return as result a field element stored in the compact form. In the following sections, algorithms for efficient squaring, square root extraction and multiplication using the split representation will be discussed.

3.3.1 Squaring

Since squaring in $\mathbb{F}_{2^m}$ is a linear map, the square of a field element $a(z)$ represented in split form is:

$$a(z)^2 = (a_H(z)z^4 + a_L(z))^2 = a_H(z)^2z^8 + a_L(z)^2.$$

Algorithmically, this means we can compute the $a(z)^2$ by adding the squares of a_L and a_H with an 8-bit offset. Squaring a_L and a_H in turn can be computed by the conventional method of inserting a zero bit between each pair of consecutive bits on their binary representations through a 16-byte table lookup [59]. Since these two polynomials have coefficients stored in 4-bit sets, the table lookups can be executed simultaneously using PTLU instructions. The proposed optimization is shown in Algorithm 3.1. The algorithm receives a field element a stored in a compact vector of n 64-bit words (or $\frac{n}{2}$ 128-bit values) and at each iteration of the algorithm, a 128-bit value $a[2i]$ is loaded from memory and converted to the split representation by a bit mask. Each group of nibbles is then expanded from 4 bits to 8 bits by a parallel table lookup. The final 8-bit offset addition is implemented by interleaving instructions which pick alternately the lower or higher bytes of a_L or a_H to form two consecutive 128-bit values $(t[2i], t[2i+1])$ produced as the result. The polynomial stored into t can be reduced modulo $f(z)$ to produce the final result $c(z)$.

3.3.2 Square Root

Square root extraction is an important operation for fast implementations of point halving [62] and the η_T pairing [1]. Square root is the inverse of squaring and consequently it is also a linear map. Using the split representation, we have:

$$\sqrt{a(z)} = \sqrt{a_H(z)z^4 + a_L(z)} = \sqrt{a_H(z)} \cdot z^2 + \sqrt{a_L(z)}.$$

Algorithm 3.1 Proposed optimization for the implementation of squaring in $\mathbb{F}_{2^m}$.

```
Input: $a(z) = a[0..n-1]$.
Output: $c(z) = c[0..n-1] = a(z)^2 \bmod f(z)$.
 1: ◇ Store in $table$ the squares $u(z)^2$ of all 4-bit polynomials $u(z)$.
 2: $table \leftarrow$ (0x5554515045444140,0x1514111005040100)
 3: $mask \leftarrow$ (0x0F0F0F0F0F0F0F0F,0x0F0F0F0F0F0F0F0F)
 4: for $i \leftarrow 0$ to $\frac{n}{2} - 1$ do
 5:     $a_0 \leftarrow load(a[2i])$
 6:     ◇ Convert to split representation.
 7:     $a_L \leftarrow a_0 \wedge mask$
 8:     $a_H \leftarrow a_0 \gg_{\nmid 8} 4$, $a_H \leftarrow a_H \wedge mask$
 9:     ◇ Perform parallel table lookups.
10:     $a_L \leftarrow lookup(table, a_L)$, $a_H \leftarrow lookup(table, a_H)$
11:     ◇ Simulate addition with 8-bit offset.
12:     $t[2i] \leftarrow interlo(a_L, a_H)$
13:     $t[2i+1] \leftarrow interhi(a_L, a_H)$
14: end for
15: return $c = t \bmod f(z)$
```

Given an element $a(z) \in \mathbb{F}_{2^m}$, the field element $c(z)$ such that $c(z)^2 = a(z) \bmod f(z)$ can be computed by the expression $c(z) = a_{even}(z) + \sqrt{z} \cdot a_{odd}(z) \bmod f(z)$, where $a_{even}(z)$ represents the concatenation of even coefficients of $a(z)$, $a_{odd}(z)$ represents the concatenation of odd coefficients of $a(z)$ and $\sqrt{z}$ is a constant depending on the irreducible polynomial $f(z)$ [62]. When f is chosen as a square root friendly polynomial [134], $\sqrt{z}$ has a sparse format and multiplication by this constant can be implemented with cheap shifted additions and no further reduction. For several of the NIST-standardized polynomials (e.g., $\mathbb{F}_{2^{283}}$ and $\mathbb{F}_{2^{571}}$), however, this is not the case and multiplication by $\sqrt{z}$ can be computed by a half-precision multiplication. A common optimization technique to treat this case is to precompute a multiplication table for $\sqrt{z}$.

Considering the above, the split representation induces a square root algorithm where $\sqrt{a_L(z)}$ and $\sqrt{a_H(z)}$ are computed separately and added with a 2-bit offset:

$$\begin{aligned} \sqrt{a(z)} &= \sqrt{a_H(z)}z^2 + \sqrt{a_L(z)} \\ &= \sqrt{z} \cdot (a_{L_{odd}}(z) + a_{H_{odd}}(z)z^2) + a_{L_{even}}(z) + a_{H_{even}}(z)z^2. \end{aligned}$$

Algorithm 3.2 presents our implementation of this method with vector instructions. The algorithm processes 128 bits of a in each iteration and progressively separates the coefficients of $a[2i]$ into even or odd coefficients. First, a permutation mask is used to divide $a[2i]$ in bytes of odd index and bytes of even index. This trick makes the final step easier. The bytes with even indexes are stored in the lower 64-bit part of a_0 and the bytes with odd indexes are stored in the higher 64-bit part of a_0. The high and low nibbles of a_0

are then split into a_L and a_H and additional lookup tables are applied to further separate the bits of a_L and a_H into $(a_{L_{odd}}(z), a_{L_{even}}(z), a_{H_{odd}}(z)z^2, a_{H_{even}}(z)z^2)$. Note that the 2-bit offset (or z^2 factor) is embedded in the $sqrt_H$ lookup table. At the end of the 128-bit section, a_0 stores the interleaving of coefficients from a_L and a_H packed into 4-bit sets. The remaining instructions in the 128-bit section separate the even and odd coefficients into u and v, which can be reordered and multiplied by $\sqrt{z}$. We implement these final steps in 64-bit mode to avoid expensive shifts in 128-bit mode.

Algorithm 3.2 Proposed optimization for square root in $\mathbb{F}_{2^m}$.

Input: $a(z) = a[0..n-1]$, exponents m and t of trinomial $f(z)$.

Output: $c(z) = c[0..n-1] = a(z)^{\frac{1}{2}} \bmod f(z)$.

```
1: ◇ Permutation mask to divide a 128-bit value in bytes with odd and even indexes.
2: $perm \leftarrow$ (0x0F0D0B0907050301,0x0E0C0A0806040200)
3: ◇ Table to divide a low nibble in bits with odd and even indexes.
4: $sqrt_L \leftarrow$ (0x3332232231302120,0x1312030211100100)
5: ◇ Table to divide a high nibble in bits with odd and even indexes ($sqrt_L \ll_{\nmid 8} 2$).
6: $sqrt_H \leftarrow$ (0xCCC88C88C4C08480,0x4C480C0844400400)
7: ◇ Bit masks to convert to split representation.
8: $mask_L \leftarrow$ (0x0F0F0F0F0F0F0F0F,0x0F0F0F0F0F0F0F0F)
9: $mask_H \leftarrow$ (0xF0F0F0F0F0F0F0F0,0xF0F0F0F0F0F0F0F0)
10: $a_{even} \leftarrow 0, a_{odd} \leftarrow 0$
11: for $i \leftarrow 0$ to $\frac{n}{2} - 1$ do
12:     $a_0 \leftarrow load(a[2i])$
13:     $a_0 \leftarrow shuffle(a_0, perm)$
14:     ◇ Convert permuted vector to split representation.
15:     $a_L \leftarrow a_0 \wedge mask_L$
16:     $a_H \leftarrow a_0 \wedge mask_H$, $a_H \leftarrow a_H \gg_{\nmid 8} 4$
17:     ◇ Extract $(a_{L_{odd}}(z) + a_{L_{even}(z)})$ in $a_L$ and $(a_{H_{odd}}(z)z^2 + a_{H_{even}(z)}z^2)$ in $a_H$.
18:     $a_L \leftarrow lookup(sqrt_L, a_L)$, $a_H \leftarrow lookup(sqrt_H, a_H)$
19:     ◇ Compute $(a_{L_{odd}}(z) + a_{L_{even}(z)} + a_{H_{odd}}(z)z^2 + a_{H_{even}(z)}z^2)$.
20:     $a_0 \leftarrow a_L \oplus a_H$
21:     ◇ Compute $u = a_{L_{even}}(z) + a_{H_{even}}(z)z^2$ and $v = a_{L_{odd}}(z) + a_{H_{odd}}(z)z^2$.
22:     $a_L \leftarrow a_0 \wedge mask_L$, $a_H \leftarrow a_0 \wedge mask_H$
23:     $u \leftarrow store(a_L)$
24:     $v \leftarrow store(a_H)$
25:     ◇ From now on, operate in 64-bit registers.
26:     $a_{even} \leftarrow a_{even} + (u[0] \vee (u[1] \ll 4))$
27:     $a_{odd} \leftarrow a_{odd} + (v[1] \vee (v[0] \gg 4))$
28: end for
29: return $c(z) = a_{even} + \sqrt{z} \cdot a_{odd} \bmod f(z)$
```

3.3.3 Multiplication

In this section, we discuss two strategies for implementing multiplication induced by the split representation.

Single operand in split representation

If the multiplier is represented in split form, the following expression for multiplication is obtained:

$$a(z) \cdot b(z) = b(z) \cdot (a_H(z)z^4 + a_L(z)) = b(z)z^4 a_H(z) + b(z)a_L(z). \tag{3.1}$$

The full multiplication result can then be obtained by adding the partial results of two smaller multiplications with a 4-bit offset. Since multiplication in a binary field rarely enjoys native support in common processors, one of the fastest ways of implementing multiplication with a polynomial basis is through the precomputation-based algorithm proposed by López and Dahab [27]. For computing $a(z) \cdot b(z)$, this algorithm builds a table of products of small polynomials of degree less than w by the full field element $b(z)$ and scans operand $a(z)$ in sets of w bits of each word at a time adding the intermediate results left-shifted by multiples of w. A common implementation choice is to use $w = 4$. The core operation of this algorithm is simulating a fast way of multiplying a small polynomial by a full field element using a lookup table. This allows the implementation to employ the XOR instruction with highest granularity included in the target platform, heavily benefiting from vector instruction sets. When left-shifts by 4 bits are expensive, as is the case with the target platform, a variant is also provided in [27] which employs two precomputation tables: one for $b(z)$ and other for $b(z)z^4$. This variant is clearly induced by Equation 3.1 and its implementation with vector registers is presented in Algorithm 3.3. In our implementation, all the shifted additions are done in registers to avoid costly memory operations and left-shifts by 4 bits are completely eliminated in favor of left-shifts by 8 bits. Recall that shifts by multiples of 8 bits can use the convenient and faster memory-alignment instructions. Operand a is scanned in 128-bit intervals to avoid 64-bit shifts of the partial result stored in registers.

Both operands in split representation

If both multiplicand and multiplier are represented in split form, the following expression is obtained:

$$a(z) \cdot b(z) = (b_H(z)z^4 + b_L(z)) \cdot (a_H(z)z^4 + a_L(z)).$$

Due to the sparseness of the split representation, a direct software implementation of this formula would lead to 4 applications of Algorithm 3.3. By using Karatsuba [121], we can

Algorithm 3.3 LD multiplication implemented with n 128-bit registers.

Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..n-1] = a(z) \cdot b(z)$.
Note: m_i denotes the vector of $\frac{n}{2}$ 128-bit registers $(r_{(i-1+n/2)}, \ldots, r_i)$.

```
1: Compute $T_0(u) = u(z) \cdot b(z), T_1(u) = u(z) \cdot (b(z)z^4)$ for all $u(z)$ of degree $< 4$.
2: $(r_{n-1} \ldots, r_0) \leftarrow 0$
3: for $k \leftarrow 56$ downto $0$ by $8$ do
4:    for $j \leftarrow 1$ to $n-1$ by $2$ do
5:       ◇ Process implictly 4 bits of $a_L$ and then 4 bits of $a_H$.
6:       Let $u = (u_3, u_2, u_1, u_0)$, where $u_t$ is bit $(k+t)$ of $a[j]$.
7:       $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_0(u)$
8:       Let $v = (v_3, v_2, v_1, v_0)$, where $v_t$ is bit $(k+t+4)$ of $a[j]$.
9:       $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_1(v)$
10:   end for
11:   $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
12: end for
13: for $k \leftarrow 56$ downto $0$ by $8$ do
14:   for $j \leftarrow 0$ to $n-2$ by $2$ do
15:      ◇ Process implictly 4 bits of $a_L$ and then 4 bits of $a_H$.
16:      Let $u = (u_3, u_2, u_1, u_0)$, where $u_t$ is bit $(k+t)$ of $a[j]$.
17:      $m_{j/2} \leftarrow m_{j/2} \oplus T_0(u)$
18:      Let $v = (v_3, v_2, v_1, v_0)$, where $v_t$ is bit $(k+t+4)$ of $a[j]$.
19:      $m_{j/2} \leftarrow m_{j/2} \oplus T_1(v)$
20:   end for
21:   if $k > 0$ then $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
22: end for
23: return $c = (r_{n-1} \ldots, r_0) \bmod f(z)$
```

lower the number of applications to 3:

$$a(z) \cdot b(z) = a_H b_H z^8 + \left[(a_H + a_L)(b_H + b_L) + a_H b_H + a_L b_L\right] z^4 + a_L b_L.$$

This can be improved by observing that a multiplication in split representation can be seen as a series of products of polynomials of degree less than w by a sparse polynomial with coefficients grouped in sets of size w. Products of a small polynomial by small sets of coefficients can be efficiently computed through table lookups and, moreover, by choosing $w = 4$ we can implement the 4-bit granular multiplication by simultaneous table lookups. Replacing lookups in a precomputation table (as in Algorithm 3.3) by lookups in a constant table leads to reduced memory consumption and increased arithmetic density. Higher arithmetic density has lower dependency on the performance of the memory subsystem and is consequently more attractive for implementation in vector processors. Unfortunately, there is no easy way of storing the table of constants without using memory and our approach does not benefit heavily of the feature. Algorithm 3.4 presents our shuffle-based approach to multiplication with the auxiliary function M_{split} defined in Algorithm 3.5. The disadvantage of this approach in comparison with [27] is that the core operation is sparser: now we multiply a small polynomial by a sparse field element, requiring more executions of this core operation to achieve the result. A fast shuffle instruction is thus required to implement this strategy in a competitive way. It is important to note that this approach is not immune to cache latency effects, since lookups using 4-bit sets of a on a table of constants stored into memory are still required. This is also the reason why operand a is processed in 64-bit mode inside M_{split}: quick access to these sets must be provided. Operand a is scanned in offsets a multiple of 8 so there is no need to apply masks to explicitly obtain a_L and a_H.

Algorithm 3.4 Left-to-right shuffle-based multiplication in $\mathbb{F}_{2^m}$.

```
Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..n-1] = a(z) \cdot b(z)$.
 1: ◇ Bit mask to convert to split representation.
 2: $mask \leftarrow$ (0x0F0F0F0F0F0F0F0F,0x0F0F0F0F0F0F0F0F)
 3: ◇ Prepare operands $a_L(z)$ and $b_L(z)$ in split representation.
 4: for $i \leftarrow 0$ to $\frac{n}{2}-1$ do $b_H[i] \leftarrow load(b[2i])$, $b_L[i] = b_H[i] \wedge mask$, $a_L[i] = a[i]$
 5: ◇ Compute $a_L(z)b_L(z)$.
 6: $(m_L[n-1], \ldots, m_L[0]) \leftarrow M_{split}(a_L, b_L)$
 7: ◇ Prepare operands $a_H(z)$ and $b_H(z)$ in split representation.
 8: for $i \leftarrow 0$ to $\frac{n}{2}-1$ do $b_H[i] \leftarrow (b_H[i] \gg_{\nmid 8} 4) \wedge mask$, $a_H[i] \leftarrow a[i] \gg 4$
 9: ◇ Compute $a_H(z)b_H(z)$.
10: $(m_H[n-1], \ldots, m_H[0]) \leftarrow M_{split}(a_H, b_H)$
11: ◇ Prepare operands $(a_L(z) + a_H(z))$ and $(b_L(z) + b_H(z))$ in split representation.
12: for $i \leftarrow 0$ to $\frac{n}{2}-1$ do $b_M[i] \leftarrow b_H[i] \oplus b_L[i]$, $a_M[i] = a_L[i] \oplus a_H[i]$
13: ◇ Compute $(a_L(z) + a_H(z))(b_L(z) + b_H(z))$.
14: $(m[n-1], \ldots, m[0]) \leftarrow M_{split}(a_M, b_M)$
15: ◇ Compute $(a_L(z) + a_H(z))(b_L(z) + b_H(z)) + a_L(z)b_L(z) + a_H(z)b_H(z)$.
16: for $i \leftarrow 0$ to $n-1$ do
17:     $m[i] \leftarrow m[i] \oplus (m_L[i] \oplus m_H[i])$
18: end for
19: ◇ Multiply $[(a_L(z) + a_H(z))(b_L(z) + b_H(z)) + a_L(z)b_L(z) + a_H(z)b_H(z)]$ by $z^4$.
20: $(m[n-1], \ldots, m[0]) \leftarrow (m[n-1], \ldots, m[0]) \ll_{\nmid 8} 4$
21: ◇ Multiply $a_H(z)b_H(z)$ by $z^8$.
22: $(m_H[n-1], \ldots, m_H[0]) \leftarrow (m_H[n-1], \ldots, m_H[0]) \lhd 8$
23: ◇ Compute $a_H b_H z^8 + [(a_H + a_L)(b_H + b_L) + a_H b_H + a_L b_L]\, z^4 + a_L b_L$.
24: for $i \leftarrow 0$ to $n-1$ do
25:     $m[i] \leftarrow m[i] \oplus (m_L[i] \oplus m_H[i])$
26: end for
27: return $c = (m[n-1], \ldots, m[0]) \bmod f(z)$
```

Algorithm 3.5 Auxiliary multiplication function $M_{split}(a, b)$.

```
Input: Operands $a, b$ in split representation.
Output: Result $a \cdot b$ stored in registers $(r_{n-1} \ldots, r_0)$.
 1: ◇ Table of constants storing all products of 4-bit × 4-bit polynomials.
 2: $table[0] \leftarrow$ (0x0000000000000000,0x0000000000000000)
 3: $table[1] \leftarrow$ (0x0F0E0D0C0B0A0908,0x0706050403020100)
 4: $table[2] \leftarrow$ (0x1E1C1A1816141210,0x0E0C0A0806040200)
 5: $table[3] \leftarrow$ (0x111217141D1E1B18,0x090A0F0C05060300)
 6: $table[4] \leftarrow$ (0x3C3834302C282420,0x1C1814100C080400)
 7: $table[5] \leftarrow$ (0x3336393C27222D28,0x1B1E11140F0A0500)
 8: $table[6] \leftarrow$ (0x22242E283A3C3630,0x12141E180A0C0600)
 9: $table[7] \leftarrow$ (0x2D2A232431363F38,0x15121B1C090E0700)
10: $table[8] \leftarrow$ (0x7870686058504840,0x3830282018100800)
11: $table[9] \leftarrow$ (0x777E656C535A4148,0x3F362D241B120900)
12: $table[10] \leftarrow$ (0x666C72784E445A50,0x363C22281E140A00)
13: $table[11] \leftarrow$ (0x69627F74454E5358,0x313A272C1D160B00)
14: $table[12] \leftarrow$ (0x44485C5074786C60,0x24283C3014180C00)
15: $table[13] \leftarrow$ (0x4B46515C7F726568,0x232E3934171A0D00)
16: $table[14] \leftarrow$ (0x5A544648626C7E70,0x2A243638121C0E00)
17: $table[15] \leftarrow$ (0x555A4B4469667778,0x2D22333C111E0F00)
18: $(r_{n-1} \ldots, r_0) \leftarrow 0$
19: for $k \leftarrow 56$ downto $0$ by $8$ do
20:     for $j \leftarrow 1$ to $n-1$ by $2$ do
21:         Let $u = (u_3, u_2, u_1, u_0)$, where $u_t$ is bit $(k+t)$ of $a[j]$.
22:         for $i \leftarrow 0$ to $\frac{n}{2} - 1$ do $r_i \leftarrow r_i \oplus \mathit{shuffle}(table[u], b[i])$
23:     end for
24:     $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
25: end for
26: for $k \leftarrow 56$ downto $0$ by $8$ do
27:     for $j \leftarrow 0$ to $n-2$ by $2$ do
28:         Let $u = (u_3, u_2, u_1, u_0)$, where $u_t$ is bit $(k+t)$ of $a[j]$.
29:         for $i \leftarrow 0$ to $\frac{n}{2} - 1$ do $r_i \leftarrow r_i \oplus \mathit{shuffle}(table[u], b[i])$
30:     end for
31:     if $k > 0$ then $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
32: end for
```

3.3.4 Modular Reduction and Inversion

Efficient modular reduction and inversion do not benefit from the split representation. The performance of both operations heavily depend on the performance of shifting and *ad-hoc* optimizations. For this reason, we only provide some of the general guidelines that were followed for the implementation of modular reduction for all the considered different choices of irreducible polynomial $f(z) = z^m + r(z)$:

- If $f(z)$ is a trinomial and $m - deg(r) \geq 128$, modular reduction should be implemented in 128-bit mode because of the small number of shifts required. Bytewise shifts should be used whenever possible. One example of modular reduction implemented with this guideline can be found in [102].

- If $f(z)$ is a pentanomial or if the degree of $r(z)$ is too big to satisfy the previous criteria, but still $m - deg(r) \geq 64$, modular reduction should be implemented in 64-bit mode but processing two words in parallel with 128-bit registers when possible.
- Consecutive writes can be accumulated into registers before being written to memory to avoid redundant memory writes.
- If the number of 128-bit digits n required to store a field element is small, squaring and multiplication results can be stored and immediately reduced using registers instead of using arrays. This optimization also reduces redundant memory operations for reloading and reducing an intermediate value.

For inversion in $\mathbb{F}_{2^m}$, the Extended Euclidean Algorithm variant for polynomials [59] can be implemented in 64-bit mode to make use of flexible shifts by arbitrary amounts. Our implementation of inversion was focused only in correctness and did not receive the amount of attention dedicated to the performance-critical operations.

3.4 Experimental Results

We now present and discuss our timings for finite field arithmetic and elliptic curve arithmetic.

3.4.1 Finite field arithmetic

We implemented arithmetic in the binary fields $\mathbb{F}_{2^m}, m \in \{113, 127, 163, 233, 239, 251, 283, 409, 571\}$, using the non-square root friendly polynomials defined by NIST or the Standards for Efficient Cryptography Group (SECG) [135] or the ECRYPT Benchmarking of Cryptographic Systems (eBACS) project [90]; and using the independent square root friendly polynomials given in [134, 136, 67]. All our choices for defining polynomials can be found in Table 3.2.

The C programming language was used in conjunction with compiler intrinsics for accessing vector instructions. The chosen compiler was GCC version 4.1.2 with backports from 4.3 because in our experiments it generated the fastest code from vector intrinsics, as already observed in [67]. Compiler flags included optimization level `-O2`, loop unrolling and tuning with the `-march=core2` switch. Considered target platforms are the Core 2 65nm (Core 2 I), Core 2 45nm (Core 2 II) and Core i7 45nm microarchitectures,

Table 3.2: Different choices of irreducible binary polynomials $f(z)$ of degree m. Polynomials are classified into Standard (as defined by NIST, SECG or eBACS) or square root friendly (SQRTF).

m	$f(z)$	$\sqrt{z} \bmod f(z)$	Type
113	$z^{113} + z^9 + 1$	$z^{57} + z^5$	SQRTF
127	$z^{127} + z^{63} + 1$	$z^{64} + z^{32}$	SQRTF
163	$z^{163} + z^7 + z^6 + z^3 + 1$	79 terms	Standard
	$z^{163} + z^{57} + z^{49} + z^{29} + 1$	$z^{82} + z^{29} + z^{25} + z^{15}$	SQRTF
233	$z^{233} + z^{74} + 1$	$z^{228} + z^{191} + z^{154} + z^{117} + z^{69} + z^{32}$	SQRTF
	$z^{233} + z^{159} + 1$	$z^{117} + z^{80}$	SQRTF
239	$z^{239} + z^{158} + 1$	$z^{199} + z^{120} + z^{118} + z^{39}$	Standard
	$z^{239} + z^{81} + 1$	$z^{120} + z^{41}$	SQRTF
251	$z^{251} + z^7 + z^4 + z^2 + 1$	166 terms	Standard
	$z^{251} + z^{89} + z^{81} + z^3 + 1$	$z^{126} + z^{45} + z^{41} + z^2$	SQRTF
283	$z^{283} + z^{12} + z^7 + z^5 + 1$	68 terms	Standard
	$z^{283} + z^{97} + z^{89} + z^{87} + 1$	$z^{142} + z^{49} + z^{45} + z^{44}$	SQRTF
409	$z^{409} + z^{87} + 1$	$z^{205} + z^{44}$	SQRTF
571	$z^{571} + z^{10} + z^5 + z^2 + 1$	273 terms	Standard
	$z^{571} + z^{193} + z^{185} + z^5 + 1$	$z^{286} + z^{97} + z^{93} + z^3$	SQRTF
1223	$z^{1223} + z^{255} + 1$	$z^{612} + z^{128}$	SQRTF

represented by a mobile Intel Core 2 T7200 2.0GHz, a Xeon X3320 2.5GHz and a Core i7 860 2.8GHz, respectively. Finite field arithmetic was implemented as an arithmetic backend of the RELIC toolkit [105] and the library was used for testing and benchmarking.

Table 3.3 presents our results for finite field arithmetic. The operations considered are squaring, multiplication, square root extraction and inversion. The standard polynomials are used in all operations with the exception of square root modulo a friendly polynomial. The differences between our implementations in the various microarchitectures can be explained by the lower cost of the bytewise shift and shuffle instructions after the introduction of the *Super Shuffle Engine* by Intel in the first release of the 45nm microarchitecture. Timings for the Core i7 are a little faster or competitive with timings for the Core 2 45nm architecture up to $m = 251$. For the bigger fields, minor compiler decisions focusing on Core 2 scheduling characteristics decrease performance in Core i7. To date, no released GCC version has specific tuning for the Core i7 microarchitecture. Our implementation shows some interesting results:

1. PTLU-based approaches like squaring, square root extraction and shuffle-based multiplication are significantly faster in the latest microarchitectures. The increased throughput of PTLU instructions is visible in simple operations like squaring and square root extraction and on the small field multipliers. Square root extraction

modulo friendly polynomials is the only PTLU-based approach that benefits from increased throughput in all field sizes. This happens because of its the lower dependence on memory performance compared to squaring or shuffle-based multiplication, since square root does not rely on a table of constants stored into memory and has a higher arithmetic density between fetching the operands and storing the results. Nevertheless, these approaches will likely benefit from future enhancements to the flexibility or speed of PTLU instructions.

2. Squaring and square root extraction with a friendly $f(z)$ are formulated and computed very efficiently using shuffling, producing multiplication-to-squaring ratios as high as 34. Table lookups are so fast that performance is mainly dependent on the choice of $f(z)$ and the resulting form of $\sqrt{z}$, as we can see in two cases: (i) comparing squaring in $\mathbb{F}_{2^{251}}$ and in $\mathbb{F}_{2^{409}}$, where the former requires the expansion of two vector registers, but expansion of three vector registers and reduction modulo a trinomial in the latter is cheaper; (ii) comparing square root with friendly $f(z)$ between $\mathbb{F}_{2^{571}}$ and $\mathbb{F}_{2^{1223}}$, where the former has half the field size, but the nice form of $\sqrt{z}$ in the latter makes square root extraction faster. Square root in field $\mathbb{F}_{2^{1223}}$ on the Core i7 is indeed an extreme example of this, since it is competitive with square root modulo friendly polynomials in much smaller fields.

3. Shuffle-based multiplication strategy (Algorithm 3.4) gets a significant speed improvement with faster shuffle instructions. However, memory access for addressing the table of 16 constants acts as a bottleneck for further improvements in the Core i7 architecture where shuffle throughput is doubled. A faster way to address this table must be made available so the higher arithmetic density can be explored without memory performance interference. It is still surprising that this algorithm requiring three times the number of operations of conventional López-Dahab multiplication (Algorithm 3.3) is only 50%-90% slower in the microarchitectures where shuffling is faster.

Our results lack a full comparison with other works because we were not able to find published timings for the same combinations of parameters and architectures using vector instruction sets. Comparing with the results given by [136] for a 64-bit C-only implementation of arithmetic in $\mathbb{F}_{2^{127}}$ on a Core 2 65nm machine: (i) both López-Dahab and shuffle-based multiplication approaches are faster by 25% and 3% than the reported 270 cycles, respectively, because of the higher addition granularity and fast shifting by 8 bits; (ii) square root extraction is 64% faster due to the way PTLU instructions are employed; (iii) inversion is 17% slower because it was not considered performance-critical in this work. The improvement with vector registers is still less than what could be

Table 3.3: Timings in cycles for our implementation of binary field arithmetic arithmetic in Intel platforms Core 2 65nm (Core 2 I), Core 2 45nm (Core 2 II) and Core i7 45nm. Square root friendly polynomials are only used when explicitly stated (or when the standard polynomial is also square root friendly). Results are the average of 10^4 executions of each algorithm with random inputs.

	Platform and field size					
	Core 2 I	Core 2 II	Core i7	Core 2 I	Core 2 II	Core i7
Field operation	$m = 113$			$m = 127$		
Squaring	34	25	24	25	16	16
Square root with standard f	-	-	-	-	-	-
Square root with friendly f	27	23	22	22	19	17
López-Dahab multiplication	147	127	116	200	179	165
Shuffle-based multiplication	257	187	176	262	191	169
Inversion	3675	3593	3446	4096	4042	3894
	$m = 163$			$m = 233$		
Squaring	60	49	48	50	36	34
Square root with standard f	195	163	167	128	124	117
Square root with friendly f	99	89	84	92	76	74
López-Dahab multiplication	240	218	216	276	241	246
Shuffle-based multiplication	506	336	313	695	429	403
Inversion	5984	5948	5756	9986	9922	10074
	$m = 239$			$m = 251$		
Squaring	53	43	47	72	59	58
Square root with standard f	113	104	94	258	237	231
Square root with friendly f	94	77	71	109	99	88
López-Dahab multiplication	284	251	261	350	325	323
Shuffle-based multiplication	686	441	412	738	468	475
Inversion	10259	10150	10101	10816	10471	10761
	$m = 283$			$m = 409$		
Squaring	73	58	55	67	52	50
Square root with standard f	309	278	292	171	155	135
Square root with friendly f	131	106	103	171	155	135
López-Dahab multiplication	418	397	400	751	690	715
Shuffle-based multiplication	1084	664	647	2239	1234	1265
Inversion	13522	13301	13608	23199	23050	24012
	$m = 571$			$m = 1223$		
Squaring	124	95	94	160	108	110
Square root with standard f	844	827	849	-	-	-
Square root with friendly f	211	191	174	166	140	114
López-Dahab multiplication	1247	1173	1197	4030*	3785*	3912*
Shuffle-based multiplication	4105	2163	2285	12204*	7219*	7367*
Inversion	38404	38173	40226	149763	149589	161577

(*) Karatsuba at depth 1 was used at the top level.

expected by doubling the register size, but this can be explained in terms of arithmetic density: López-Dahab multiplication benefits from larger ratios between field size and register size. We also refer the reader to [102] for a comparison of our implementation of arithmetic in $\mathbb{F}_{2^{1223}}$ using similar techniques with an efficient implementation of this field reported in [98].

A full comparison can however be provided in the fields $\mathbb{F}_{2^{113}}$ and $\mathbb{F}_{2^{251}}$ employed for curve-based key exchange benchmarking in the eBACS project [90]. We present in Table 3.4 timings obtained by the eBACS 64-bit version of the $\mathbf{mp}\mathbb{F}_q$ library [137] distributed as part of SUPERCOP version 20100509. This version employs the SSE2 vector instruction set inside the multiplier and relies on automatic tuning of the multiplication code. These timings were measured through the simple integration of $\mathbf{mp}\mathbb{F}_q$ as an arithmetic backend of RELIC so that the same compiling, testing and benchmarking procedures could be used. Comparison of results in Tables 3.3 and 3.4 for the corresponding fields favor our approaches with the exception of López-Dahab Multiplication in $\mathbb{F}_{2^{113}}$ on Core 2 I. Speedups range from 8% to 84%. This gives an idea of the improvements of our approach over the previous state-of-the-art in efficient implementations of finite field arithmetic using vector instructions. Note that although our code was hand-written, most of it closely follows the described algorithms. There is no obstacle to automatically generate an implementation with comparable performance.

Table 3.4: Timings in cycles of the $\mathbf{mp}\mathbb{F}_q$ library for binary field arithmetic arithmetic in Intel platforms Core 2 65nm (Core 2 I), Core 2 45nm (Core 2 II) and Core i7 45nm. Using notation of Table 3.3, results of square root extraction correspond to the square root friendly polynomial in $\mathbb{F}_{2^{113}}$ and to the standard polynomial in $\mathbb{F}_{2^{251}}$. Results are the average of 10^4 executions of each algorithm with random inputs.

	Platform and field size					
	Core 2 I	Core 2 II	Core i7	Core 2 I	Core 2 II	Core i7
Field operation	$m = 113$			$m = 251$		
Squaring	38	40	36	106	103	102
Square root	144	141	128	557	552	526
Multiplication	135	139	129	490	489	458
Inversion	5497	5435	5430	18425	16502	17907

3.4.2 Elliptic curve arithmetic

The performance obtained by our efficient implementation is illustrated with some executions of the Elliptic Curve Diffie-Hellman (ECDH) and Elliptic Curve Digital Signature Algorithm (ECDSA) protocols [110]. We have selected fast algorithms for elliptic curve arithmetic in three situations: multiplying a random point P by a scalar k, multiplying the generator G by a scalar k and simultaneously multiplying two points P and Q by scalars k and l to obtain $kP + lQ$. Our implementation uses mixed addition with projective coordinates [60], given that the inversion-to-multiplication ratio is between 22 and 41.

For multiplying a random point by a scalar, we choose Solinas' *τ-adic non-adjacent form* (TNAF) representation [63] with $w = 4$ for Koblitz curves (4-TNAF method with 4 precomputation points) and the method due to López and Dahab [122] for random binary curves. We did not implement point halving [62] because NIST field definitions are not optimal for square root computation, although conversion to the friendly representations used in Table 3.3 would be practical in some scenarios. For multiplying the generator, we employ the 6-TNAF method for Koblitz curves; and for generic curves, we employ the Comb method [64] with 256 precomputed points. For simultaneous multiplication, we implement the interleaving method with 4-TNAFs for Koblitz curves and with 4-NAFs (*width-4 non-adjacent form*) for generic curves [65].

Table 3.5 presents our timings for elliptic curve arithmetic using the fastest finite field implementations presented in the previous section. We provide timings for curves at the 128-bit and 256-bit security levels defined by NIST as reference for future implementations. The eBACS [90] website currently reports timings for key exchange of 855000, 748000 and 755000 cycles on platforms Core 2 I (`latour`), Core 2 II (`needme`) and Core i7 (`dragon`) respectively using CURVE2251. We improve these timings by 27%-30% with our implementation of CURVE2251 and show that we can instead compute key exchanges in curve NIST-B283 with performance comparable to the eBACS results. Bernstein [61] describes a bitsliced implementation of 251-bit binary Edwards curves (BBE) that computes a batch of 1024 point multiplications in parallel taking in average 314000 cycles per scalar multiplication in a Core 2 I (`latour`) platform. A question left open is the speed of a bitsliced approach using special curves. The straight-line requirements noted in [61] are naturally compatible with reducing side-channel exposure, but these do not appear to be compatible with traditional approaches on Koblitz curves. As a reference point, our results show that it's possible to match BBE's performance with a conventional implementation using the Koblitz curve SECG-K239 if one is willing to accept the following trade-offs: (i) allow curves with special algebraic structures; (ii) relax side-channel resistance; and (iii) lower the security level to a 239-bit curve. In order to comply with the security requirements in [61] while keeping the flexibility a non-bitsliced implementation provides, currently the best choice for binary curves is to use our implementation of CURVE2251 with the added performance impact.

3.5 Conclusion

In this work, we proposed novel techniques for exploring parallelism during the implementation of binary fields in computers equipped with modern vector instruction sets. It was made clear that field arithmetic can be efficiently formulated and implemented us-

Table 3.5: Timings given in 10^3 cycles for elliptic curve operations inside executions of the ECDH and ECDSA protocols measured in Intel platforms Core 2 65nm (Core 2 I), Core 2 45nm (Core 2 II) and Core i7 45nm. Results are the average of 10^4 executions of each algorithm with random inputs.

	Key Exchange (kP)			**Sign** (kG) + **Verify** ($kG + lP$)		
Curve	Core 2 I	Core 2 II	Core i7	Core 2 I	Core 2 II	Core i7
SECG-K239	270	247	260	196 + 419	175 + 384	184 + 400
CURVE2251	594	535	537	172 + 926	157 + 850	157 + 852
NIST-K283	411	378	386	298 + 632	270 + 586	275 + 599
NIST-B283	858	785	793	225 + 1212	210 + 1120	212 + 1140
NIST-K571	1782	1617	1656	1295 + 2840	1159 + 2580	1185 + 2645
NIST-B571	4754	4310	4440	1225 + 6206	1126 + 5683	1145 + 5822

ing these instructions. A competitive shuffle-based implementation of multiplication was introduced and our results show that it has two main requirements: (i) the availability of fast shuffle instructions; (ii) an efficient table addressing mechanism avoiding memory accesses completely. If both requirements can be satisfied in a low-cost way, we pose an interesting question on the design of future vector arithmetic units: is there a real need for a native binary field multiplier from a cost-benefit perspective? Supporting a single fast shuffle instruction can be cheaper and may allow very efficient implementations of the performance-critical operations in $\mathbb{F}_{2^m}$ and other fields of cryptographic interest such as $\mathbb{F}_{3^m}$.

We illustrated our implementation results with timings for ECDSA and ECDH protocols in two different NIST-standardized security levels, showing that a random point multiplication at the 128-bit security level can be computed in 411000 cycles in an Intel Core 2 65nm architecture. A comparison of our results with benchmark data from the eBACS project shows that our implementation of CURVE2251 is up to 30% faster. The timings on SECG-K239 provide a reference point for future comparisons with batch methods such as [61] under the assumption that the interest is in the fastest point multiplication on standardized curves over binary fields. This comparison will be interesting to revisit once the effects of the to-be-released native multiplier [87] and the AVX extensions [133] are evaluated in both implementations.

Acknowledgements

We would like to thank the anonymous reviewers for pointing out useful suggestions on how to compare our preliminary results with related work; and CNPq, CAPES and FAPESP, which support authors Diego F. Aranha and Julio López.

Chapter 4

Software implementation of binary elliptic curves: impact of the carry-less multiplier on scalar multiplication

Jonathan Taverne, Armando Faz-Hernández, Diego F. Aranha, Francisco Rodríguez-Henríquez, Darrel Hankerson and Julio López

Abstract

The availability of a new carry-less multiplication instruction in the latest Intel desktop processors significantly accelerates multiplication in binary fields and hence presents the opportunity for reevaluating algorithms for binary field arithmetic and scalar multiplication over elliptic curves. We describe how to best employ this instruction in field multiplication and the effect on performance of doubling and halving operations. Alternate strategies for implementing inversion and half-trace are examined that restore most of their competitiveness relative to the new multiplier. These improvements in field arithmetic are complemented by a study on serial and parallel approaches for Koblitz and random curves, where parallelization strategies are implemented and compared. The contributions are illustrated with experimental results improving the state-of-the-art performance of halving and doubling-based scalar multiplication on NIST curves at the 112- and 192-bit security levels, and a new speed record for side-channel resistant scalar multiplication in a random curve at the 128-bit security level.

Publication

This work was accepted for publication in the Proceedings of the 13th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2011). ©2011 IACR

4.1 Introduction

Improvements in the fabrication process of microprocessors allow the resulting higher transistor density to be converted into architectural features such as inclusion of new instructions or faster execution of the current instruction set. Limits on the conventional ways of increasing a processor's performance such as incrementing the clock rate, scaling the memory hierarchy [38] or improving support for instruction-level parallelism [36] have pushed manufacturers to embrace parallel processing as the mainstream computing paradigm and consequently amplify support for resources such as multiprocessing and vectorization. Examples of the latter are the recent inclusions of the SSE4 [51], AES [138] and AVX [52] instruction sets in the latest Intel microarchitectures.

Since the dawn of elliptic curve cryptography in 1985, several field arithmetic assumptions have been made by researchers and designers regarding its efficient implementation in software platforms. Some analysis (supported by experiments) assumed that inversion to multiplication ratios (I/M) were sufficiently small (e.g., $I/M \approx 3$) that point operations would be done in affine coordinates, favoring certain techniques. However, the small ratios were a mix of old hardware designs, slower multiplication algorithms compared with [27], and composite extension degree. It seems clear that sufficient progress was made in multiplication so there is incentive to use projective coordinates. Our interest in the face of much faster multiplication is at the other end—is I/M large enough to affect methods that commonly assumed this ratio is modest?

On the other hand, authors in [62] considered that the cost of a point halving computation was roughly equivalent to 2 field multiplications. The expensive computations in halving are a field multiplication, solving a quadratic $z^2+z=c$, and finding a square root over $\mathbb{F}_{2^m}$. However, quadratic solvers presented in [59] are multiplication-free and hence, provided that a fast binary field multiplier is available, there would be concern that the ratio of point halving to multiplication may be much larger than 2. Having a particularly fast multiplier would also push for computing square roots in $\mathbb{F}_{2^m}$ as efficiently as possible. Similarly, the common software design assumption that field squaring is essentially free (relative to multiplication) may no longer be valid.

A prevalent assumption is that large-characteristic fields are faster than binary field counterparts for software implementations of elliptic curve cryptography.[1] In spite of simpler arithmetic, binary field realizations could not be faster than large-characteristic analogues mostly due to the absence of a native *carry-less multiplier* in contemporary high-performance processors. However, using a bit-slicing technique, Bernstein [61] was able to compute a batch of 251-bit scalar multiplications on a binary Edwards curve, employing

[1]In hardware realizations, the opposite thesis is widely accepted: elliptic curve scalar point multiplication can be computed (much) faster using binary extension fields.

314,323 clock cycles per scalar multiplication, which, before the results presented in this work and to the best of our knowledge, was the fastest reported time for a software implementation of binary elliptic point multiplication.

In this work, we evaluate the impact of the recently introduced carry-less multiplication instruction [87] in the performance of binary field arithmetic and scalar multiplication over elliptic curves. We also consider parallel strategies in order to speed scalar multiplication when working on multi-core architectures. In contrast to parallelization applied to a batch of operations, the approach considered here applies to a single point multiplication. These approaches target different environments: batching makes sense when throughput is the measurement of interest, while the lower level parallelization is of interest when latency matters and the device is perhaps weak but has multiple processing units. Furthermore, throughout this paper we will assume that we are working in the unknown point scenario, i.e., where the elliptic curve point to be processed is not known in advance, thus precluding off-line precomputation. We will assume that there is sufficient memory space for storing a few multiples of the point to be processed and look-up tables for accelerating the computation of the underlying field arithmetic.

As the experimental results will show, our implementation of multiplication via this native support was significantly faster than previous timings reported in the literature. This motivated a study on alternative implementations of binary field arithmetic in hope of restoring the performance ratios among different operations in which the literature is traditionally based [59]. A direct consequence of this study is that performance analysis based on these conventional ratios [139] will remain valid in the new platform. Our main contributions are:

- A strategy to efficiently employ the native carry-less multiplier in binary field multiplication.
- Branchless and/or vectorized approaches for implementing half-trace computation, integer recoding and inversion. These approaches allow the halving operation to become again competitive with doubling in the face of a significantly faster multiplier, and help to reduce the impact of integer recoding and inversion in the overall speed of scalar multiplication, even when projective coordinates are used.
- Parallelization strategies for dual core execution of scalar multiplication algorithms in random and Koblitz binary elliptic curves.

We obtain a new state-of-the-art implementation of arithmetic in binary elliptic curves, including improved performance for NIST-standardized Koblitz curves and random curves suitable for halving and a new speed record for side-channel resistant point multiplication in a random curve at the 128-bit security level.

The remainder of the paper progresses as follows. Section 4.2 elaborates on exploiting carry-less multiplication for high-performance field multiplication along with implementation strategies for half-trace and inversion. Sections 4.3 and 4.4 discuss serial and parallel approaches for scalar multiplication. Section 4.5 presents extensive experimental results and comparison with related work. Section 4.6 concludes the paper with perspectives on the interplay between the proposed implementation strategies and future enhancements in the architecture under consideration.

4.2 Binary field arithmetic

A binary extension field $\mathbb{F}_{2^m}$ can be constructed by means of a degree-m polynomial f irreducible over $\mathbb{F}_2$ as $\mathbb{F}_{2^m} \cong \mathbb{F}_2[z]/(f(z))$. In the case of software implementations in modern desktop platforms, field elements $a \in \mathbb{F}_{2^m}$ can be represented as polynomials of degree at most $m-1$ with binary coefficients a_i packed in $n_{64} = \lceil \frac{m}{64} \rceil$ 64-bit processor words. In this context, the recently introduced carry-less multiplication instruction can play a significant role in order to efficiently implement a multiplier in $\mathbb{F}_{2^m}$. Along with field multiplication, other relevant field arithmetic operations such as squaring, square root, and half-trace, will be discussed in the rest of this section.

4.2.1 Multiplication

Field multiplication is the performance-critical operation for implementing several cryptographic primitives relying on binary fields, including arithmetic over elliptic curves and the Galois Counter Mode of operation (GCM). For accelerating the latter when used in combination with the AES block cipher [138], Intel introduced the carry-less multiplier in the Westmere microarchitecture as an instruction operating on 64-bit words stored in 128-bit vector registers with opcode *pclmulqdq* [87]. The instruction latency currently peaks at 15 cycles while reciprocal throughput ranks at 10 cycles. In other words, when operands are not in a dependency chain, effective latency is 10 cycles [140].

The instruction certainly looks expensive when compared to the 3-cycle 64-bit integer multiplier present in the same platform, which raises speculation whether Intel aimed for an area/performance trade-off or simply balanced the latency to the point where the carry-less multiplier did not interfere with the throughput of the hardware AES implementation. Either way, the instruction features suggest the following empirical guidelines for organizing the field multiplication code: (i) as memory access by vector instructions continues to be expensive [61], the maximum amount of work should be done in registers, for example through a Comba organization [141]; (ii) as the number of registers employed in multiplication should be minimized for avoiding false dependencies and maximize throughput, the multiplier should have 128-bit granularity; (iii) as the instruction latency allows, each 128-bit multiplication should be implemented with three carry-less multiplications in a Karatsuba fashion [121].

In fact, the overhead of Karatsuba multiplication is minimal in binary fields and the Karatsuba formula with the smaller number of multiplications for multiplying $\lceil \frac{n_{64}}{2} \rceil$ 128-bit digits proved to be optimal in all the considered field sizes. This observation comes in direct contrast to previous vectorized implementations of the *comb* method for binary field multiplication due to López and Dahab [27, Algorithm 5], where the memory-bound precomputation step severely limits the number of Karatsuba steps which can be employed, fixing the cutoff point to large fields [88] such as $\mathbb{F}_{2^{1223}}$. To summarize, multiplication was implemented as a 128-bit granular Karatsuba multiplier with each 128-digit multiplication solved by another Karatsuba instance requiring three carry-less multiplications, cheap additions and efficient shifts by multiples of 8 bits. A single 128-digit level of Karatsuba was used for fields $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{251}}$ where $\lceil \frac{n_{64}}{2} \rceil = 2$, while two instances were used for field $\mathbb{F}_{2^{409}}$ where $\lceil \frac{n_{64}}{2} \rceil = 4$. Particular approaches which led to lower performance in our experiments were organizations based on optimal Toom-Cook [142] due to the higher overhead brought by minor operations; and on a lower 64-bit granularity combined with alternative multiple-term Karatsuba formulas [143] due to register exhaustion to store all the intermediate values, causing a reduction in overall throughput.

4.2.2 Squaring, square-root and multi-squaring

Squaring and square-root are considered cheap operations in a binary field, especially when $\mathbb{F}_{2^m}$ is defined by a square-root friendly polynomial [134, 144], because they require only linear manipulation of individual coefficients [59]. These operations are traditionally implemented with the help of large precomputed tables, but vectorized implementations are possible with simultaneous table lookups through byte shuffling instructions [88]. This approach is enough to keep square and square-root efficient relative to multiplication even with a dramatic acceleration of field multiplication. For illustration, [88] reports multiplication-to-squaring ratios as high as 34 without a native multiplier, far from the conventional ratios of 5 [139] or 7 [59] and with a large room for future improvement.

Multi-squaring, or exponentiation to 2^k, can be efficiently implemented with a time-memory trade-off proposed as m-squaring in [144, 145] and here referred as *multi-squaring.* For a fixed k, a table T of $16\lceil \frac{m}{4} \rceil$ field elements can be precomputed such that $T[j, i_0+2i_1+4i_2+8i_3] = (i_0 z^{4j}+i_1 z^{4j+1}+i_2 z^{4j+2}+i_3 z^{4j+3})^{2^k}$ and $a^{2^k} = \sum_{j=0}^{\lceil \frac{m}{4} \rceil} T[j, \lfloor a/2^{4j} \rfloor \bmod 2^4]$. The threshold where multi-squaring became faster than simple consecutive squaring observed in our implementation was around $k \geq 6$ for $\mathbb{F}_{2^{233}}$ and $k \geq 10$ for $\mathbb{F}_{2^{409}}$.

4.2.3 Inversion

Inversion modulo $f(z)$ can be implemented via the polynomial version of the Extended Euclidean Algorithm (EEA), but the frequent branching and recurrent shifts by arbitrary amounts present a performance obstacle for vectorized implementations, which makes it difficult to write consistently fast EEA codes across different platforms. A branchless approach can be implemented through Itoh-Tsuji inversion [146] by computing $a^{-1} = a^{(2^{m-1}-1)2}$, as proposed in [147]. In contrast to the EEA method, the Itoh-Tsujii approach has the additional merit of being similarly fast (relative to multiplication) across common processors.

The overall cost of the method is $m - 1$ squarings and a number of multiplications dictated by the length of an addition chain for $m - 1$. The cost of squarings can be reduced by computing each required 2^i-power as a multi-squaring [145]. The choice of an addition chain allows the implementer to control the amount of required multiplications and the precomputed storage for multi-squaring, since the number of 2^i-powers involved can be balanced.

Previous work obtained inversion-to-multiplication ratios between 22 and 41 by implementing EEA in 64-bit mode [88], while the conventional ratios are between 5 and 10 [59, 139]. While we cannot reach the small ratios with Itoh-Tsujii for the parameters considered here, we can hope to do better than applying the method from [88] which will give significantly larger ratios with the carry-less multiplier. Hence the cost of squarings and multi-squarings should be minimized to the lowest possible allowed by storage capacity.

To summarize, we use addition chains of 10, 10 and 11 steps for computing field inversion over the fields $\mathbb{F}_{2^{233}}$, $\mathbb{F}_{2^{251}}$ and $\mathbb{F}_{2^{409}}$, respectively.[2] We extensively used the multi-squaring approach described in the preceding section. For example, in the case of $\mathbb{F}_{2^{233}}$, we selected the addition chain $1{\rightarrow}2{\rightarrow}3{\rightarrow}6{\rightarrow}7{\rightarrow}14{\rightarrow}28{\rightarrow}29{\rightarrow}58{\rightarrow}116{\rightarrow}232$, and used 3 pre-computed tables for computing the iterated squarings $a^{2^{29}}$, $a^{2^{58}}$ and $a^{2^{116}}$. The rest of the field squaring operations were computed by executing consecutive squarings. We recall that each table stores a total of $16\lceil \frac{m}{4} \rceil$ field elements.

4.2.4 Half-trace

Half-trace plays a central role in point halving and its performance is essential if halving is to be competitive against doubling. For an odd integer m, the half-trace function $H : \mathbb{F}_{2^m} \to \mathbb{F}_{2^m}$ is defined by $H(c) = \sum_{i=0}^{(m-1)/2} c^{2^{2i}}$ and satisfies the equation $\lambda^2 + \lambda = c + \mathrm{Tr}(c)$ required for point halving. One efficient desktop-targeted implementation of the

[2]In the case of inversion over $\mathbb{F}_{2^{409}}$, the minimal length addition chain to reach $m - 1 = 408$ has 10 steps. However, we preferred to use an 11-step chain to save one look-up table.

half-trace is described in [134] and presented as Algorithm 4.1, making extensive use of precomputations. This implementation is based on two main steps: the elimination of even power coefficients and the accumulation of half-trace precomputed values.

Step 5 in Algorithm 4.1, as shown in [59], consists in reducing the number of non-zero coefficients of c by removing the coefficients of even powers i via $H(z^i) = H(z^{i/2}) + z^{i/2} + \mathrm{Tr}(z^i)$. That will lead to memory and time savings during the last step of the half-trace computation, the accumulation (step 8). This is done by extraction of the odd and even bits and can benefit from vectorization in the same way as square-root in [88]. However, in the case of half-trace there is a bottleneck caused by data dependencies. For efficiency, the bank of 128-bit registers is used as much as possible, but at one point in the algorithm execution the number of available bits to process decreases. For 64-bit and 32-bit digits, the use of 128-bit registers is still beneficial, but for a smaller size, the conventional approach (not vectorized) becomes again competitive.

Once step 5 is completed, the direction taken in [59] remains in reducing memory needs. However another approach is followed in [134] which does not attempt to minimize memory requirements but rather it greedily strives to speed up the accumulation part (step 8). Precomputation is extended so as to reduce the number of accesses to the lookup table. The following values of the half-trace are stored: $H(l_0c^{8i+1}+l_1c^{8i+3}+l_2c^{8i+5}+l_3c^{8i+7})$ for all $i \geq 0$ such that $8i < m-3$ and $l_j \in \mathbb{F}_2$. The memory size in bytes taken by the precomputations follows the formula $16 \times n_{64} \times 8 \times \lceil \frac{m}{8} \rceil$.

Algorithm 4.1 Solve $x^2 + x = c$

```
Input: $c = \sum_{i=0}^{m-1} c_i z^i \in \mathbb{F}_{2^m}$ where $m$ is odd and $\mathrm{Tr}(c) = 0$
Output: a solution $s$ of $x^2 + x = c$
1: compute $H(l_0c^{8i+1} + l_1c^{8i+3} + l_2c^{8i+5} + l_3c^{8i+7})$ for $i \in I = \{0, \ldots, \lfloor \frac{m-3}{8} \rfloor\}$ and $l_j \in \mathbb{F}_2$
2: $s \leftarrow 0$
3: for $i = (m-1)/2$ downto $1$ do
4:     if $c_{2i} = 1$ then
5:         $c \leftarrow c + z^i$, $s \leftarrow s + z^i$
6:     end if
7: end for
8: return $s \leftarrow s + \sum_{i \in I} c^{8i+1}H(z^{8i+1}) + c^{8i+3}H(z^{8i+3}) + c^{8i+5}H(z^{8i+5}) + c^{8i+7}H(z^{8i+7})$
```

While considering different organizations of the half-trace code, we made the following serendipitous observation: inserting as many `xor` operations as the data dependencies permitted from the accumulation stage (step 8) into step 5 gave a substantial speed-up of 20% to 25% compared with code written in the order as described in Algorithm 4.1. Plausible explanations are compiler optimization and processor pipelining characteristics. The result is a half-trace-to-multiplication ratio near 1, and this ratio can be reduced if memory can be consumed more aggressively.

4.3 Random binary elliptic curves

Given a finite field $\mathbb{F}_q$ for $q = 2^m$, a non-supersingular elliptic curve $E(\mathbb{F}_q)$ is defined to be the set of points $(x, y) \in \mathbb{F}_q \times \mathbb{F}_q$ that satisfy the affine equation

$$y^2 + xy = x^3 + ax^2 + b, \tag{4.1}$$

where a and $0 \neq b \in \mathbb{F}_q$, together with the point at infinity denoted by $\mathcal{O}$. It is known that $E(\mathbb{F}_q)$ forms an additive Abelian group with respect to the elliptic point addition operation.

Let k be a positive integer and P a point on an elliptic curve. Then *elliptic curve scalar multiplication* is the operation that computes the multiple $Q = kP$, defined as the point resulting of adding P to itself $k-1$ times. One of the most basic methods for computing a scalar multiplication is based on a double-and-add variant of Horner's rule. As the name suggests, the two most prominent building blocks of this method are the *point doubling* and *point addition* primitives. By using the non-adjacent form (NAF) representation of the scalar k, the addition-subtraction method computes a scalar multiplication in about m doubles and $m/3$ additions [59]. The method can be extended to a *width-ω NAF* $k = \sum_{i=0}^{t-1} k_i 2^i$ where $k_i \in \{0, \pm 1, \ldots, \pm 2^m - 1\}$, $k_{t-1} \neq 0$, and at most one of any ω consecutive digits is nonzero. The length t is at most one larger than the bitsize of k, and the density is approximately $1/(\omega + 1)$; for $\omega = 2$, this is the same as NAF.

4.3.1 Sequential algorithms for random binary curves

The traditional left-to-right double-and-add method is illustrated in Algorithm 4.2 where $n = 0$ (that is, the computation corresponds to the left column) and the *width-ω NAF* $k = \sum_{i=0}^{t-1} k_i 2^i$ expression is computed from left to right, i.e., it starts processing k_{t-1} first, then k_{t-2} until it ends with the coefficient k_0. Step 1 computes $2^{\omega-2} - 1$ multiples of the point P. Based on the Montgomery trick, authors in [148] suggested a method to precompute the affine points in large-characteristic fields $\mathbb{F}_p$, employing only one inversion. Exporting that approach to $\mathbb{F}_{2^m}$, we obtained formulae that offer a saving of 4 multiplications and 15 squarings for $\omega = 4$ when compared with a naive method that would make use of the Montgomery trick in a trivial way (see Table 4.1 for a summary of the computational effort associated to this phase).

For a given ω, the evaluation stage of the algorithm has approximately $m/(\omega + 1)$ point additions, and hence increasing ω has diminishing returns. For the curves given by NIST [149] and with on-line precomputation, $\omega \leq 6$ is optimal in the sense that total point additions are minimized. In many cases, the recoding in ωNAF(k) is performed on-line and can be considered as part of the precomputation step.

Algorithm 4.2 Double-and-add, halve-and-add scalar multiplication: parallel

Input: ω, scalar k, $P \in E(\mathbb{F}_{2^m})$ of odd order r, constant n (e.g., from Table 4.1(b))
Output: kP

```
1: Compute $P_i = iP$ for
   $i \in I = \{1, 3, \ldots, 2^{\omega-1} - 1\}$
2: $Q_0 \leftarrow \mathcal{O}$
   {Barrier}
5: for $i = t$ downto $n$ do
6:    $Q_0 \leftarrow 2Q_0$
7:    if $k'_i > 0$ then
8:       $Q_0 \leftarrow Q_0 + P_{k'_i}$
9:    else if $k'_i < 0$ then
10:      $Q_0 \leftarrow Q_0 - P_{-k'_i}$
11:   end if
12: end for
   {Barrier}
21: return $Q \leftarrow Q_0 + \sum_{i \in I} iQ_i$
```

```
3: Recode: $k' = 2^n k \bmod r$ and obtain rep
   $\omega\mathrm{NAF}(k')/2^n = \sum_{i=0}^{t} k'_i 2^{i-n}$
4: Initialize $Q_i \leftarrow \mathcal{O}$ for $i \in I$
13: for $i = n - 1$ downto $0$ do
14:    $P \leftarrow P/2$
15:    if $k'_i > 0$ then
16:       $Q_{k'_i} \leftarrow Q_{k'_i} + P$
17:    else if $k'_i < 0$ then
18:       $Q_{-k'_i} \leftarrow Q_{-k'_i} - P$
19:    end if
20: end for
```

The most popular way to represent points in binary curves is López-Dahab projective coordinates that yield an effective cost for a mixed point addition and point doubling operation of about $8M + 5S \approx 9M$ and $4M + 5S \approx 5M$, respectively (see Tables 4.2 and 4.3). Kim and Kim [150] report alternate formulas for point doubling requiring four multiplications and five squarings, but two of the four multiplications are by the constant b, and these have the same cost as general multiplication with the native carry-less multiplier. For mixed addition, Kim and Kim require eight multiplications but save two field reductions when compared with López-Dahab, giving their method the edge. Hence, in this work we use López-Dahab for point doubling and Kim and Kim for point addition.

Right-to-left halve-and-add

Scalar multiplication based on point halving replaces point doubling by a potentially faster *halving* operation that produces Q from P with $P = 2Q$. The method was proposed independently by Knudsen [151] and Schroeppel [152] for curves $y^2 + xy = x^3 + ax^2 + b$ over $\mathbb{F}_{2^m}$. The method is simpler if the trace of a is 1, and this is the only case we consider. The expensive computations in halving are a field multiplication, solving a quadratic $z^2 + z = c$, and finding a square root. On the NIST random curves studied in this work, we found that the cost of halving is approximately $3M$, where M denotes the cost of a field multiplication.

Let the base point P have odd order r, and let t be the number of bits to represent

Table 4.1: Costs and parameter recommendations for $\omega \in \{3, 4, 5\}$.

ω	Algorithm 4.2		[59, Alg 3.70]	[59, Alg 3.70]'
	Precomp	Postcomp	Precomp	Postcomp
3	14M , 11S , I	43M , 26S	2M , 3S , I	26M , 13S
4	38M , 15S , I	116M , 79S	9M , 9S , I	79M , 45S
5	N/A	N/A	23M , 19S , 2I	200M , 117S

(a) Pre- and post-computation costs.

ω	Algorithm 4.2		Algorithm 4.3	
	B-233	B-409	K-233	K-409
3	128	242	131	207
4	132	240	135	210
5	N/A	N/A	136	213

(b) Recommended value for n.

r. For $0 < n \leq t$, let $\sum_{i=0}^{t} k'_i 2^i$ be given by the width-ω NAF of $2^n k \bmod r$. Then $k \equiv k'/2^n \equiv \sum_{i=0}^{t} k'_i 2^{i-n} \pmod r$ and the scalar multiplication can be split as

$$kP = (k'_t 2^{t-n} + \cdots + k'_n)P + (k'_{n-1} 2^{-1} + \cdots + k'_0 2^{-n})P. \tag{4.2}$$

When $n = t$, this gives the usual representation for point multiplication via halving, illustrated in Algorithm 4.2 (that is, the computation is essentially the right column). The cost for postcomputation appears in Table 4.1.

4.3.2 Parallel scalar multiplication on random binary curves

For parallelization, choose $n < t$ in (4.2) and process the first portion by a double-and-add method and the second portion by a method based on halve-and-add. Algorithm 4.2 illustrates a parallel approach suitable for two processors. Recommended values for n to balance cost between processors appear in Table 4.1.

4.3.3 Side-channel resistant multiplication on random curves

Another approach for scalar multiplication offering some resistance to side-channel attacks was proposed by López and Dahab [122] based on the Montgomery laddering technique. This approach requires $6M + 5S$ in $\mathbb{F}_{2^m}$ per iteration independently of the bit pattern in the scalar, and one of these multiplications is by the curve coefficient b. The curve being lately used for benchmarking purposes [90] at the 128-bit security level is an Edwards curve (CURVE2251) corresponding to the Weierstraß curve $y^2 + xy = x^3 + (z^{13} + z^9 + z^8 + z^7 + z^2 + z + 1)$. It is clear that this curve is especially tailored for this method due to the short length of b, reducing the cost of the algorithm to approximately $5.25M + 5S$

per iteration. At the same time, halving-based approaches are non-optimal for this curve due to the penalties introduced by the 4-cofactor [153]. Considering this and to partially satisfy the side-channel resistance offered by a bitsliced implementation such as [61], we restricted the choices of scalar multiplication at this security level to the Montgomery laddering approach.

4.4 Koblitz elliptic curves

A Koblitz curve $E_a(\mathbb{F}_q)$, also known as an Anomalous Binary Curve [125], is a special case of (4.1) where $b = 1$ and $a \in \{0, 1\}$. In a binary field, the map taking x to x^2 is an automorphism known as the Frobenius map. Since Koblitz curves are defined over the binary field $\mathbb{F}_2$, the Frobenius map and its inverse naturally extend to automorphisms of the curve denoted τ and τ^{-1}, respectively, where $\tau(x, y) = (x^2, y^2)$. Moreover, $(x^4, y^4) + 2(x, y) = \mu(x^2, y^2)$ for every (x, y) on E_a, where $\mu = (-1)^{1-a}$; that is, τ satisfies $\tau^2+2 = \mu\tau$ and we can associate τ with the complex number $\tau = \frac{\mu+\sqrt{-7}}{2}$.

Solinas [63] presents a τ-adic analogue of the usual NAF as follows. Since short representations are desirable, an element $\rho \in \mathbb{Z}[\tau]$ is found with $\rho \equiv k \pmod{\delta}$ of as small norm as possible, where $\delta = (\tau^m - 1)/(\tau - 1)$. Then for the subgroup of interest, $kP = \rho P$ and a width-ω τ-adic NAF ($\omega\tau$NAF) for ρ is obtained in a fashion that parallels the usual ωNAF. As in [63], define $\alpha_i = i \bmod \tau^\omega$ for $i \in \{1, 3, \ldots, 2^{\omega-1}-1\}$. A $\omega\tau$NAF of a nonzero element ρ is an expression $\rho = \sum_{i=0}^{l-1} u_i\tau^i$ where each $u_i \in \{0, \pm\alpha_1, \pm\alpha_3, \ldots, \pm\alpha_{2^{\omega-1}-1}\}$, $u_{l-1} \neq 0$, and at most one of any consecutive ω coefficients is nonzero. Scalar multiplication kP can be performed with the $\omega\tau$NAF expansion of ρ as

$$u_{l-1}\tau^{l-1}P + \cdots + u_2\tau^2P + u_1\tau P + u_0P \tag{4.3}$$

with $l - 1$ applications of τ and approximately $l/(\omega + 1)$ additions.

The length of the representation is at most $m + a$, and Solinas presents an efficient technique to find an estimate for ρ, denoted $\rho' = k$ partmod δ with $\rho' \equiv \rho \pmod{\delta}$, having expansion of length at most $m + a + 3$ [63, 154]. Under reasonable assumptions, the algorithm will usually produce an estimate giving length at most $m+1$. For simplicity, we will assume that the recodings obtained have this as an upper bound on length; small adjustments are necessary to process longer representations. Under these assumptions and properties of τ, scalars may be written $k = \sum_{i=0}^{m} u_i\tau^i = \sum_{i=0}^{m} u_i\tau^{-(m-i)}$ since $\tau^{-i} = \tau^{m-i}$ for all i.

4.4.1 Sequential algorithms for Koblitz curves

A traditional left-to-right τ-and-add method for (4.3) appears as [59, Alg 3.70], and is essentially the left-hand portion of Algorithm 4.3. Precomputation consists of $2^{\omega-2}-1$ multiples of the point P, each at a cost of approximately one point addition (see Table 4.1 for a summary of the computational effort associated to this phase).

Alternatively, we can process bits right-to-left and obtain a variant we shall denote as [59, Alg 3.70]$'$ (an analogue of [59, Alg 3.91]). The multiple points of precomputation P_u are exchanged for the same number of accumulators Q_u along with postcomputation of form $\sum \alpha_u Q_u$. The cost of postcomputation is likely more than the precomputation of the left-to-right variant; see Table 4.1 for a summary in the case where postcomputation uses projective additions. However, if the accumulator in Algorithm 4.3 is in projective coordinates, then the right-to-left variant has a less expensive evaluation phase since τ is applied to points in affine coordinates.

4.4.2 Parallel algorithm for Koblitz curves

The basic strategy in our parallel algorithm is to reformulate the scalar multiplication in terms of both the τ and the τ^{-1} operators as $k = \sum_{i=0}^{m} u_i\tau^i = u_0 + u_1\tau^1 + \cdots + u_n\tau^n + u_{n+1}\tau^{-(m-n-1)} + \cdots + u_m = \sum_{i=0}^{n} u_i\tau^i + \sum_{i=n+1}^{m} u_i\tau^{-(m-i)}$ where $0 < n < m$. Algorithm 4.3 illustrates a parallel approach suitable for two processors. Although similar in structure to Algorithm 4.2, a significant difference is the shared precomputation rather than the pre and postcomputation required in Algorithm 4.2.

The scalar representation is given by Solinas [63] and hence has an expected $m/(\omega+1)$ point additions in the evaluation-stage, and an extra point addition at the end. There are also approximately m applications of τ or its inverse. If the field representation is such that these operators have similar cost or are sufficiently inexpensive relative to field multiplication, then the evaluation stage can be a factor 2 faster than a corresponding non-parallel algorithm.

As discussed before, unlike the ordinary width-ω NAF, the τ-adic version requires a relatively expensive calculation to find a short ρ with $\rho \equiv k \pmod{\delta}$. Hence, (a portion of) the precomputation is "free" in the sense that it occurs during scalar recoding. This can encourage the use of a larger window size ω. The essential features exploited by Algorithm 4.3 are that the scalar can be efficiently represented in terms of the Frobenius map and that the map and its inverse can be efficiently computed, and hence the algorithm adapts to curves defined over small fields.

Algorithm 4.3 is attractive in the sense that two processors are directly supported without "extra" computations. However, if multiple applications of the "doubling step" are sufficiently inexpensive, then more processors and additional curves can be accommodated

Algorithm 4.3 $\omega\tau$NAF scalar multiplication: parallel

Input: ω, $k \in [1, r-1]$, $P \in E_a(\mathbb{F}_{2^m})$ of order r, constant n (e.g., from Table 4.1(b))
Output: kP

```
 1: $\rho \leftarrow k$ partmod $\delta$
 2: $\sum_{i=0}^{l-1} u_i\tau^i \leftarrow \omega\tau\mathrm{NAF}(\rho)$
 3: $P_u = \alpha_u P$,
    for $u \in \{1, 3, 5, \ldots, 2^{\omega-1} - 1\}$
    {Barrier}
 4: $Q_0 \leftarrow \mathcal{O}$
 5: for $i = n$ downto 0 do
 6:    $Q_0 \leftarrow \tau Q_0$
 7:    if $u_i = \alpha_j$ then
 8:       $Q_0 \leftarrow Q_0 + P_j$
 9:    else if $u_i = -\alpha_j$ then
10:       $Q_0 \leftarrow Q_0 - P_j$
11:    end if
12: end for
13: $Q_1 \leftarrow \mathcal{O}$
14: for $i = n + 1$ to $m$ do
15:    $Q_1 \leftarrow \tau^{-1} Q_1$
16:    if $u_i = \alpha_j$ then
17:       $Q_1 \leftarrow Q_1 + P_j$
18:    else if $u_i = -\alpha_j$ then
19:       $Q_1 \leftarrow Q_1 - P_j$
20:    end if
21: end for
    {Barrier}
22: return $Q \leftarrow Q_0 + Q_1$
```

in a straightforward fashion without sacrificing the high-level parallelism of Algorithm 4.3. As an example for Koblitz curves, a variant of Algorithm 4.3 discards the applications of τ^{-1} (which may be more expensive than τ) and finds $kP = k^1(\tau^j P) + k^0 P = \tau^j(k^1 P) + k^0 P$ for suitable k^i and $j \approx m/2$ with traditional methods to calculate $k^i P$. The application of τ^j is low cost if there is storage for a per-field matrix as it was first discussed in [144].

4.5 Experimental results

We consider example fields $\mathbb{F}_{2^m}$ for $m \in \{233, 251, 409\}$. These were chosen to address 112-bit and 192-bit security levels, according to the NIST recommendation, and the 251-bit binary Edwards elliptic curve presented in [61]. The field $\mathbb{F}_{2^{233}}$ was also chosen as more likely to expose any overhead penalty in the parallelization compared with larger fields from NIST. Our C library coded all the algorithms using the GNU C 4.6 (GCC) and Intel 12 (ICC) compilers, and the timings were obtained on a 3.326 GHz 32nm Intel *Westmere* processor i5 660.

Obtaining times useful for comparison across similar systems can be problematic. Intel, for example, introduced “Pentium 4" processors that were fundamentally different than earlier designs with the same name. The common method via time stamp counter (TSC) requires care on recent processors having “turbo" modes that increase the clock (on perhaps 1 of 2 cores) over the nominal clock implicit in TSC, giving an underestimate of actual cycles consumed. Benchmarking guidelines on eBACS [90], for example,

Table 4.2: Timings in clock cycles for field arithmetic operations. "op/M" denotes ratio to multiplication obtained from ICC.

Base field	$\mathbb{F}_{2^{233}}$			$\mathbb{F}_{2^{251}}$			$\mathbb{F}_{2^{409}}$		
operation	GCC	ICC	op/M	GCC	ICC	op/M	GCC	ICC	op/M
Multiplication	128	128	1.00	161	159	1.00	345	348	1.00
López-Dahab Mult.	256	367	2.87	338	429	2.70	637	761	2.19
Square root	67	60	0.47	155	144	0.91	59	56	0.16
Squaring	30	35	0.27	56	59	0.37	44	49	0.14
Half trace	167	150	1.17	219	212	1.33	322	320	0.92
Multi-Squaring	191	184	1.44	195	209	1.31	460	475	1.36
Inversion	2,951	2,914	22.77	3,710	3,878	24.39	9,241	9,350	26.87
4-τNAF	9,074	11,249	87.88	-	-	-	23,783	26,633	76.53
3-NAF	5,088	5,059	39.52	-	-	-	13,329	14,373	41.30
4-NAF	4,280	4,198	32.80	-	-	-	11,406	12,128	34.85
Recoding (halving)	1,543	1,509	11.79	-	-	-	3,382	3,087	8.87
Recoding (parallel)	999	1,043	8.15	-	-	-	2,272	2,188	6.29

recommend disabling such modes, and this is the method followed in this paper.

Timings for field arithmetic appear in Table 4.2. The López-Dahab multiplier described in [88] was implemented as a baseline to quantify the speedup due to the native multiplier. For the most part, timings for GCC and ICC are similar, although López-Dahab multiplication is an exception. The difference in multiplication times between $\mathbb{F}_{2^{233}} = \mathbb{F}_2[z]/(z^{233}+z^{74}+1)$ and $\mathbb{F}_{2^{251}} = \mathbb{F}_2[z]/(z^{251}+z^7+z^4+z^2+1)$ is in reduction. The relatively expensive square root in $\mathbb{F}_{2^{251}}$ is due to the representation chosen; if square roots are of interest, then there are reduction polynomials giving faster square root and similar numbers for other operations. Inversion via exponentiation (§4.2) gives I/M similar to that in [88] where an Euclidean algorithm variant was used with similar hardware but without the carry-less multiplier.

Table 4.4 shows timings obtained for different variants of sequential and parallel scalar multiplication. We observe that for ωNAF recoding with $\omega = 3, 4$, the halve-and-add algorithm is always faster than its double-and-add counterpart. This performance is a direct consequence of the timings reported in Table 4.3, where the cost of one point doubling is roughly 5.5 and 4.8 multiplications whereas the cost of a point halving is of only 3.3 and 2.5 multiplications in the fields $\mathbb{F}_{2^{233}}$ and $\mathbb{F}_{2^{409}}$, respectively. The parallel version that concurrently executes these algorithms in two threads computes one scalar multiplication with a latency that is roughly 37.7% and 37.0% smaller than that of the halve-and-add algorithm for the curves B-233 and B-409, respectively.

The bold entries for Koblitz curves identify fastest timings per category (i.e., considering the compiler, curve, and the specific value of ω used in the ωNAF recoding). For smaller ω, [59, Alg 3.70]$'$ has an edge over [59, Alg 3.70] because τ is applied to points in affine coordinates; this advantage diminishes with increasing ω due to postcomputation

Table 4.3: Timings in clock cycles for curve arithmetic operations. "op/M" denotes ratio to multiplication obtained from ICC.

Elliptic curve operations	B-233 GCC	B-233 ICC	B-233 op/M	B-409 GCC	B-409 ICC	B-409 op/M
Doubling (LD)	690	710	5.55	1,641	1,655	4.76
Addition (KIM Mixed)	1,194	1,171	9.15	2,987	3,000	8.62
Addition (LD Mixed)	1,243	1,233	9.63	3,072	3,079	8.85
Addition (LD General)	1,954	1,961	15.32	4,893	4,922	14.14
Halving	439	417	3.26	894	878	2.52

Table 4.4: Timings in 10^3 clock cycles for scalar multiplication in the unknown-point scenario.

ω	Scalar mult random curves	B-233 GCC	B-233 ICC	B-409 GCC	B-409 ICC
3	Double-and-add	240	238	984	989
3	Halve-and-add	196	192	755	756
3	(Dbl,Halve)-and-add	122	118	465	466
4	Double-and-add	231	229	941	944
4	Halve-and-add	188	182	706	705
4	(Dbl,Halve)-and-add	122	116	444	445

	Side-channel resistant scalar multiplication	CURVE2251 GCC	CURVE2251 ICC
	Montgomery laddering	296	282

ω	Scalar mult Koblitz curves	K-233 GCC	K-233 ICC	K-409 GCC	K-409 ICC
3	[59, Alg 3.70]	111	110	413	416
3	[59, Alg 3.70]′	98	**98**	**381**	389
3	(τ, τ)-and-add	**73**	74	**248**	248
3	Alg. 4.3	80	78	253	248
4	[59, Alg 3.70]	97	95	353	355
4	[59, Alg 3.70]′	90	**89**	**332**	339
4	(τ, τ)-and-add	68	**65**	216	214
4	Alg. 4.3	73	69	218	**214**
5	[59, Alg 3.70]	92	**90**	326	328
5	[59, Alg 3.70]′	95	93	**321**	332
5	(τ, τ)-and-add	63	**58**	197	**191**
5	Alg. 4.3	68	63	197	194

cost. "(τ, τ)-and-add" denotes the parallel variant described in §4.4.2. There is a storage penalty for a linear map, but applications of τ^{-1} are eliminated (of interest when τ is significantly less expensive). Given the modest cost of the multi-squaring operation (with an equivalent cost of less than 1.44 field multiplications, see Table 4.2), the (τ, τ)-and-add parallel variant is usually faster than Algorithm 4.3. When using $\omega = 5$, the parallel (τ, τ)-and-add algorithm computes one scalar multiplication with a latency that is roughly 35.5% and 40.5% smaller than that of the best sequential algorithm for the curves K-233 and K-409, respectively.

Per-field storage and coding techniques compute half-trace at cost comparable to field multiplication, and methods based on halving continue to be fastest for suitable random curves. However, the hardware multiplier and squaring (via shuffle) give a factor 2 advantage to Koblitz curves in the examples from NIST. This is larger than in [62, 59], where a 32-bit processor in the same general family as the i5 has half-trace at approximately half the cost of a field multiplication for B-233 and a factor 1.7 advantage to K-163 over B-163 (and the factor would have been smaller for K-233 and B-233). It is worth remarking that the parallel scalar multiplications versions shown in Table 4.4 look best for bigger curves and larger ω.

4.6 Conclusion and future work

In this work we achieve the fastest timings reported in the open literature for software computation of scalar multiplication in NIST and Edwards binary elliptic curves defined at the 112-bit, 128-bit and 192-bit security levels. The fastest curve implemented, namely NIST K-233, can compute one scalar multiplication in less than 17.5μs, a result that is not only much faster than previous software implementations of that curve, but is also quite competitive with the computation time achieved by state-of-the-art hardware accelerators working on similar or smaller curves [155, 144].

These fast timings were obtained through the usage of the native carry-less multiplier available in the newest Intel processors. At the same time, we strive to use the best algorithmic techniques, and the most efficient elliptic curve and finite field arithmetic formulae. Further, we proposed effective parallel formulations of scalar multiplication algorithms suitable for deployment in multi-core platforms.

The curves over binary fields permit relatively elegant parallelization with low synchronization cost, mainly due to the efficient halving or τ^{-1} operations. Parallelizing at lower levels in the arithmetic would be desirable, especially for curves over prime fields. Grabher *et al.* [96] apply parallelization for extension field multiplication, but times for a base field multiplication in a 256-bit prime field are relatively slow compared with Beuchat *et al.* [99]. On the other hand, a strategy that applies to all curves performs point doubles in one thread and point additions in another. The doubling thread stores intermediate values corresponding to nonzero digits of the NAF; the addition thread processes these points as they become available. Experimentally, synchronization cost is low, but so is the expected acceleration. Against the fastest times in Longa and Gebotys [21] for a curve over a 256-bit prime field, the technique would offer roughly 17% improvement, a disappointing return on processor investment.

The new native support for binary field multiplication allowed our implementation to improve by 10% the previous speed record for side-channel resistant scalar multiplication in random elliptic curves. It is hard to predict what will be the superior strategy between a conventional non-bitsliced or a bitsliced implementation on future revisions of the target platform: the latency of the carry-less multiplier instruction has clear room for improvement, while the new AVX instruction set has 256-bit registers. An issue with the current Sandy Bridge version of AVX is that `xor` throughput for operations with register operands was decreased significantly from 3 operations per cycle in SSE to 1 operation per cycle in AVX. The resulting performance of a bitsliced implementation will ultimately rely on the amount of work which can be scheduled to be done mostly in registers.

Acknowledgments

We wish to thank the University of Waterloo and especially Professor Alfred Menezes for useful discussions related to this work during a visit by three of the authors, where the idea of this project was discussed, planned and a portion of the development phase was done. Diego F. Aranha and Julio López thank CNPq, CAPES and FAPESP for financial support.

Chapter 5

High-speed Parallel Software Implementation of the η_T Pairing

Diego F. Aranha, Julio López and Darrel Hankerson

Abstract

We describe a high-speed software implementation of the η_T pairing over binary supersingular curves at the 128-bit security level. This implementation explores two types of parallelism found in modern multi-core platforms: vector instructions and multiprocessing. We first introduce novel techniques for implementing arithmetic in binary fields with vector instructions. We then devise a new parallelization of Miller's Algorithm to compute pairings. This parallelization provides an algorithm for pairing computation without increasing storage costs significantly. The combination of these acceleration techniques produce serial timings at least 24% faster and parallel timings 66% faster than the best previous result in an Intel Core platform, establishing a new state-of-the-art implementation of this pairing instantiation in this platform.

Publication

This work was originally published in the Proceedings of the Cryptographers' Track at the RSA Conference (CT-RSA 2010) available at `http://www.springer.com/computer/security+and+cryptology/book/978-3-642-11924-8`. ©2010 Springer-Verlag

5.1 Introduction

The computation of bilinear pairings is the most expensive operation in Pairing-based Cryptography, especially for high levels of security. For this reason, implementations must employ all the resources found in the target platform to obtain maximum efficiency. A resource being increasingly introduced in computing platforms is parallelism, in the form of vector instructions (data parallelism) and multiprocessing (task parallelism). This trend is observed even in the embedded space, with proposals of resource-constrained multi-core architectures and vector instruction sets for multimedia processing in portable devices.

This work describes a high-performance implementation of the η_T pairing [1] over binary supersingular curves at the 128-bit security level which employs these two forms of parallelism in a very efficient way. The target platform is the Intel Core architecture [30], the most popular 64-bit computing platform. Our main contributions are:

- *Novel techniques for implementing arithmetic in binary fields:* we explore powerful SIMD instructions to accelerate arithmetic in binary fields. We focus on the SSE family of vector instructions, but the same techniques can be employed with other SIMD instruction sets such as Altivec and the upcoming AMD SSE5.

- *Parallelization of Miller's Algorithm to compute pairings:* we develop a simple algorithm for parallel pairing computation which does not increase storage costs. Our parallelization is independent of the underlying pairing instantiation, allowing a parallel implementation to reach scalability in a variable number of processors unrelated to the pairing mathematical definition. This parallelization provides good scalability in fields of small characteristic.

- *Static load balancing technique:* we present a simple technique to balance the costs of parallel pairing computation between the available processing units. The technique is successfully applied for latency minimization, but its flexibility allows the implementation to determine controlled non-optimal partitions of the algorithm.

- *Experimental results:* speedups of parallel implementations over serial implementations are estimated and experimentally verified for platforms up to 8 processors. We also obtain an approximation of the performance up to 32 processing units and compare our serial and parallel execution times with the current state-of-the-art implementations with the same parameters.

The results of this work can improve serial and parallel implementations of pairings. The parallelization may be important to reduce the latency of pairing computation in two scenarios: (i) desktop-class processors running real-time applications with strict response time requirements; (ii) embedded multiprocessor architectures with weak processing units. The availability of parallel algorithms for application in these scenarios is suggested as an open problem by [96] and [67]. Our features of flexible load balancing and small storage overhead are critical for the second scenario, because they can support static scheduling schemes for compromises between pairing computation time and power consumption; and memory capacity is commonly restricted in embedded devices.

5.2 Finite Field Arithmetic

In this section we will represent the elements of $\mathbb{F}_{2^m}$ using a polynomial basis. Let $f(z)$ be an irreducible binary polynomial of degree m. The elements of $\mathbb{F}_{2^m}$ are the binary polynomials of degree at most $m-1$. A field element $a(z) = \sum_{i=0}^{m-1} a_i z^i$ is associated with the binary vector $a = (a_{m-1}, \ldots, a_1, a_0)$ of length m. In a software implementation, these bit coefficients are packed and stored in an array $(a[0], \ldots, a[n-1])$ of n W-bit words, where W is the word size of the processor. For simplicity, we assume that n is always even.

5.2.1 Vector Instruction Sets

Vector instructions, also called SIMD (Single Instruction, Multiple Data) because they operate in several data objects simultaneously, are widely supported in recent families of processor architectures. The number, functionality and efficiency of these instructions have been improved with each new generation of processors, and natural applications include multimedia processing, scientific applications or any software with high arithmetic density. Some well-known SIMD instruction sets are the Intel MMX and SSE [50] families, the Altivec extensions introduced by Apple and IBM in the Power architecture specification and AMD 3DNow. Instruction sets supported by current technology are restricted to 128-bit registers and provide simple orthogonal operations across 8, 16, 32 or 64-bit data units stored inside these registers, but future extensions such as Intel AVX and AMD SSE5 will support 256-bits registers with the added inclusion of a heavily-anticipated carry-less multiplier [87].

The Intel Core microarchitecture is equipped with several vector instruction sets which operate in 16 architectural 128-bit registers. A small subset of these instructions can be used to implement binary field arithmetic, some found in the Streaming SIMD Extensions 2 (SSE2) and others in the Supplementary SSE3 instructions (SSSE3). The SSE2 instruction set is also supported by the recent VIA Nano processors, AMD processors since the K8 family and Intel processors since the Pentium 4.

A non-exhaustive list of SSE2 instructions relevant for our work is given below. Each instruction described will be referred in the algorithms by the short mnemonic which follows the instruction opcode:

- `MOVDQU/MOVDQA` (*load*/*store*): implements load/store between unaligned/ aligned memory addresses and registers. In our implementation, all allocated memory is stored in 128-bit aligned base addresses so that the faster `MOVDQA` instruction can always be used.

- `PSLLQ/PSRLQ` ($\ll_{\nmid 8}, \gg_{\nmid 8}$): implements bitwise left/right shifts of a pair of 64-bit integers while shifting in zero bits. This instruction does not propagate bits from the lower 64-bit integer to the higher 64-bit integer, thus additional shifts and additions are required to implement bitwise shifts of 128-bit values.

- `PSLLDQ/PRLLDQ` ($\ll_8, \gg_8$): implements byte-wise left/right shifts of a 128-bit register. Since this instruction propagates bytes from the lower half to the higher half of a 128-bit register, this instruction is preferred over the previous one when the shift amount is a multiple of 8. Thus shifts by multiples of 8 bits should be used whenever possible. The latency of this instruction is 2 cycles in the first generation of Core 2 Conroe/Merom (65nm) processors and 1 cycle in the more recent Penryn/Wolfdale (45nm) microarchitecture.

- `PXOR/PAND/POR` ($\oplus, \wedge, \vee$): implements bitwise XOR/AND/OR of two 128-bit registers. These instructions have a high throughput, reaching 3 instructions per cycle when the operands are registers and there are no dependencies between consecutive operations.

- `PUNPCKLBW/PUNPCKHBW` (*interlo*/*interhi*): interleaves the lower/higher bytes in a register with the lower/higher bytes of another register.

We also find application for powerful but often-missed SSSE3 instructions:

- `PALIGNR` ($\lhd$): takes registers r_a and r_b, concatenate their values, and pull out a 128-bit section from an offset given by a constant immediate; in other words, implements a right byte-wise shift with propagation of shifted out bytes from r_a to r_b. This instruction can be used to implement a left shift by s bytes with the immediate $(16 - s)$.

- `PSHUFB` (*lookup* or *shuffle* depending on functionality): takes registers of bytes $r_a = a_0, a_1, \ldots, a_{15}$ and $r_b = b_0, b_1, \ldots, b_{15}$ and replaces r_a with the permutation $a_{b_0}, a_{b_1}, \ldots, a_{b_{15}}$; except that it replaces a_i with zero if the most significant bit of b_i

is set. A powerful use of this instruction is to perform 16 simultaneous lookups in a 16-byte lookup table. This can be easily done by storing the lookup table in r_a and the lookup indexes in r_b. Intel introduced a specific *Super Shuffle Engine* in the latest microarchitecture to reduce the latency of this instruction from 3 cycles to 1 cycle.

Alternate vector instruction sets present functional analogues of these instructions. In particular, the `PSHUFB` permutation instruction is implemented as `VPERM` in Altivec and as `PPERM` in SSE5, although the `PPERM` instruction is reportedly more powerful as it can also operate at bit level. SIMD instructions are critical for the performance of binary field arithmetic and can be easily accessed with compiler intrinsics. In the remainder of this section, the optimization techniques applied during the implementation of each field operation are detailed. We will describe algorithms in terms of *vector operations* using the mnemonics defined above so that algorithms can be easily transcribed to other target platforms. Specific instruction choices based on latency or functionality will be focused on the SSE family.

5.2.2 Squaring

Since the square of a finite field element $a(z) \in \mathbb{F}_{2^m}$ is given by $a(z)^2 = \sum_{i=0}^{m-1} a_i z^{2i} = a_{m-1}z^{2m-2} + \cdots + a_2 z^4 + a_1 z^2 + a_0$, the binary representation of $a(z)^2$ can be computed by inserting a zero bit between each pair of consecutive bits on the binary representation of $a(z)$. This operation can be accelerated by introducing a lookup table as discussed in [59]. This method can be improved further if the table lookups can be executed simultaneously. This way, for an implementation which processes 4 bits per iteration, squaring can be implemented mainly in terms of permutation instructions which convert groups of 4 bits (*nibbles*) to the corresponding expanded bytes. The proposed optimization is shown in Algorithm 5.1. The algorithm receives a field element a stored in a vector of n 64-bit words (or $\frac{n}{2}$ 128-bit values) and expands the input into a double-precision vector t which can be reduced modulo $f(z)$. At each iteration of this algorithm, a 128-bit value $a[2i]$ is loaded from memory and separated by a bit mask into two registers containing the low nibbles (a_L) and the high nibbles (a_H). Each group of nibbles is then expanded from 4 bits to 8 bits by a parallel table lookup. The proper order of bytes is restored by interleaving instructions which pick alternately the lower or higher bytes of a_L or a_H to form two consecutive 128-bit values ($t[2i], t[2i+1]$) produced as the result.

Algorithm 5.1 Proposed implementation of squaring in $\mathbb{F}_{2^m}$.

```
Input: $a(z) = a[0..n-1]$.
Output: $c(z) = c[0..n-1] = a(z)^2 \bmod f(z)$.
 1: ▷ Store in $table$ the squares $u(z)^2$ of all 4-bit polynomials $u(z)$.
 2: $table \leftarrow$ 0x5554515045444140,0x1514111005040100
 3: $mask \leftarrow$ 0x0F0F0F0F0F0F0F0F,0x0F0F0F0F0F0F0F0F
 4: for $i \leftarrow 0$ to $\frac{n}{2} - 1$ do
 5:     $a_0 \leftarrow load(a[2i])$
 6:     $a_L \leftarrow a_0 \wedge mask$,   $a_L \leftarrow lookup(table, a_L)$
 7:     $a_H \leftarrow a_0 \gg_{\nmid 8} 4$,   $a_H \leftarrow a_H \wedge mask$,   $a_H \leftarrow lookup(table, a_H)$
 8:     $t[2i] \leftarrow store(interlo(a_L, a_H))$,   $t[2i+1] \leftarrow store(interhi(a_L, a_H))$
 9: end for
10: return $c = t \bmod f(z)$
```

5.2.3 Square Root

Given an element $a(z) \in \mathbb{F}_{2^m}$, the field element $c(z)$ such that $c(z)^2 = a(z) \bmod f(z)$ can be computed by $c(z) = a_{even} + \sqrt{z} \cdot a_{odd} \bmod f(z)$, where a_{even} represents the concatenation of even coefficients of $a(z)$, a_{odd} represents the concatenation of odd coefficients of $a(z)$ and $\sqrt{z}$ is a constant depending on the irreducible polynomial $f(z)$ [62]. When $f(z)$ is a suitable trinomial $f(z) = z^m + z^t + 1$ with odd exponents m, t, $\sqrt{z}$ has the sparse form $\sqrt{z} = z^{\frac{m+1}{2}} + z^{\frac{t+1}{2}}$ and multiplication by this constant can be computed with shifts and additions only.

This algorithm can also be implemented with simultaneous table lookups. Algorithm 5.2 presents our implementation of this method with vector instructions. The algorithm processes 128 bits of a in each iteration and progressively separates the coefficients of $a[2i]$ in even or odd coefficients. First, a permutation mask is used to divide $a[2i]$ in bytes of odd index and bytes of even index. The bytes with even indexes are stored in the lower 64-bit part of a_0 and the bytes with odd indexes are stored in the higher 64-bit part of a_0. The high and low nibbles of a_0 are then divided into a_L and a_H and additional lookup tables are applied to further separate the bits of a_L and a_H into bits with odd and even indexes. At the end of the 128-bit section, a_0 stores the interleaving of odd and even coefficients of a packed into groups of 4 bits. The remaining instructions in the 128-bit section separate the even and odd coefficients into u and v, which can be reordered and multiplied by $\sqrt{z}$ inside the 64-bit section. We implement these final steps in 64-bit mode to avoid expensive shifts in 128-bit registers.

5.2.4 Multiplication

Two different strategies are commonly considered for the implementation of multiplication in $\mathbb{F}_{2^m}$. The first one consists in applying the Karatsuba algorithm [121] to divide the

Algorithm 5.2 Proposed implementation of square root in $\mathbb{F}_{2^m}$.

Input: $a(z) = a[0..n-1]$, exponents m and t of trinomial $f(z)$.

Output: $c(z) = c[0..n-1] = a(z)^{\frac{1}{2}} \bmod f(z)$.

```
1: $\triangleright$ Permutation mask to divide a 128-bit value in bytes with odd and even indexes.
2: $perm \leftarrow$ 0x0F0D0B0907050301,0x0E0C0A0806040200
3: $\triangleright$ Tables to divide a low/high nibble in bits with odd and even indexes.
4: $sqrt_L \leftarrow$ 0x3332232231302120,0x1312030211100100
5: $\triangleright$ Table to divide a high nibble in bits with odd and even indexes ($sqrt_L \ll 2$).
6: $sqrt_H \leftarrow$ 0xCCC88C88C4C08480,0x4C480C0844400400
7: $\triangleright$ Bit masks to isolate bytes in lower or higher nibbles.
8: $mask_L \leftarrow$ 0x0F0F0F0F0F0F0F0F,0x0F0F0F0F0F0F0F0F
9: $mask_H \leftarrow$ 0xF0F0F0F0F0F0F0F0,0xF0F0F0F0F0F0F0F0
10: $c[0\ldots n-1] \leftarrow 0,\ h \leftarrow \frac{n+1}{2}, l \leftarrow \frac{t+1}{128}, s_1 \leftarrow \frac{m+1}{2} \bmod 64, s_2 \leftarrow \frac{t+1}{2} \bmod 64$
11: for $i \leftarrow 0$ to $\frac{n}{2} - 1$ do
12:     $a_0 \leftarrow load(a[2i]),\quad a_0 \leftarrow shuffle(a_0, perm)$
13:     $a_L \leftarrow a_0 \wedge mask_L,\ a_L \leftarrow lookup(sqrt_L, a_L),$
14:     $a_H \leftarrow a_0 \wedge mask_H,\ a_H \leftarrow a_H \gg_{\nmid 8} 4,\quad a_H \leftarrow lookup(sqrt_H, a_H)$
15:     $a_0 \leftarrow a_L \vee a_H,\qquad a_L \leftarrow a_0 \wedge mask_L,\ a_H \leftarrow a_0 \wedge mask_H$
16:     $u \leftarrow store(a_L),\ v \leftarrow store(a_H)$
17:     $\triangleright$ From now on, operate in 64-bit registers.
18:     $a_{even} \leftarrow u[0] \vee u[1] \ll 4,\ a_{odd} \leftarrow v[1] \vee v[0] \gg 4$
19:     $c[i] \leftarrow c[i] \oplus a_{even}$
20:     $c[i+h-1] \leftarrow c[h+i-1] \oplus (a_{odd} \ll s_1)$
21:     $c[i+h] \leftarrow c[h+i] \oplus (a_{odd} \gg (64 - s_1))$
22:     $c[i+l] \leftarrow c[i+l] \oplus (a_{odd} \ll s_2)$
23:     $c[i+l+1] \leftarrow c[i+l+1] \oplus (a_{odd} \gg (64 - s_2))$
24: end for
25: return $c$
```

multiplication in sub-problems and solve each problem independently [59] (for $a(z) = A_1 z^{\lceil m/2 \rceil} + A_0$ and $b(z) = B_1 z^{\lceil m/2 \rceil} + B_0$):

$$c(z) = a(z) \cdot b(z) = A_1 B_1 z^m + [(A_1 + A_0)(B_1 + B_0) + A_1 B_1 + A_0 B_0] z^{\lceil m/2 \rceil} + A_0 B_0.$$

The second one consists in applying a direct algorithm like the *comb* method proposed by López and Dahab in [27]. Conventionally, the series of additions involved in this method are implemented through additions over sub parts of a double-precision vector. In order to reduce the number of memory accesses during these additions, we employ n registers. These registers simulate the series of memory additions by accumulating consecutive writes, allowing the implementation to reach maximum XOR throughput. We also employ an additional table T_1 analogue to T_0 which stores $u(z) \cdot (b(z) \ll 4)$ to eliminate shifts by 4, as discussed in [27]. Recall that shifts by multiples of 8 bits are faster in the target platform. We assume that the length of operand $b[0..n-1]$ is at most $64n - 7$ bits;

if necessary, terms of higher degree can be processed separately at relatively low cost. The implemented LD multiplication algorithm is shown as Algorithm 5.3. The element $a(z)$ is processed in groups of 8 bits separated by intervals of 128 bits. This avoids shifts of the register vector since a 128-bit shift can be emulated by referencing m_{i+1} instead of m_i. The multiple precision shift by 8 bits of the register vector ($\lhd 8$) is implemented with 15-byte shifts with carry propagation ($\lhd$) of register pairs.

Algorithm 5.3 LD multiplication implemented with n 128-bit registers.

```
Input: $a(z) = a[0..n-1], b(z) = b[0..n-1]$.
Output: $c(z) = c[0..n-1]$.
Note: $m_i$ denotes the vector of $\frac{n}{2}$ 128-bit registers $(r_{(i-1+n/2)}, \ldots, r_i)$.

 1: Compute $T_0(u) = u(z) \cdot b(z), T_1(u) = u(z) \cdot (b(z) \ll 4)$ for all $u(z)$ of degree $< 4$.
 2: $(r_{n-1} \ldots, r_0) \leftarrow 0$
 3: for $k \leftarrow 56$ downto $0$ by $8$ do
 4:   for $j \leftarrow 1$ to $n-1$ by $2$ do
 5:     Let $u = (u_3, u_2, u_1, u_0)$, where $u_t$ is bit $(k+t)$ of $a[j]$.
 6:     Let $v = (v_3, v_2, v_1, v_0)$, where $v_t$ is bit $(k+t+4)$ of $a[j]$.
 7:     $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_0(u)$
 8:     $m_{(j-1)/2} \leftarrow m_{(j-1)/2} \oplus T_1(v)$
 9:   end for
10:   $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
11: end for
12: for $k \leftarrow 56$ downto $0$ by $8$ do
13:   for $j \leftarrow 0$ to $n-2$ by $2$ do
14:     Let $u = (u_3, u_2, u_1, u_0)$, where $u_t$ is bit $(k+t)$ of $a[j]$.
15:     Let $v = (v_3, v_2, v_1, v_0)$, where $v_t$ is bit $(k+t+4)$ of $a[j]$.
16:     $m_{j/2} \leftarrow m_{j/2} \oplus T_0(u)$
17:     $m_{j/2} \leftarrow m_{j/2} \oplus T_1(v)$
18:   end for
19:   if $k > 0$ then $(r_{n-1} \ldots, r_0) \leftarrow (r_{n-1} \ldots, r_0) \lhd 8$
20: end for
21: return $c = (r_{n-1} \ldots, r_0) \bmod f(z)$
```

5.2.5 Modular Reduction

Efficient modular reduction depends on the format of the trinomial or pentanomial $f(z)$. In general, it's better to choose $f(z)$ such that bitwise shifts amounts are multiples of 8 bits. If the non-null coefficients of $f(z)$ are located in the lower words of the array representation of $f(z)$, consecutive writes into memory can also be accumulated into registers to avoid redundant memory writes. We illustrate these optimizations with modular reduction by $f(z) = z^{1223} + z^{255} + 1$ in Algorithm 5.4. The algorithm receives as input a vector of n 128-bit elements and reduces this vector by accumulating four memory writes at a time in registers. Note also that shifts by multiples of 8 bits are used whenever possible.

Algorithm 5.4 Proposed modular reduction by $f(z) = z^{1223} + z^{255} + 1$.

Input: $t(z) = t[0..n-1]$ (vector of 128-bit elements).
Output: $c(z) \bmod f(z) = c[0..n-1]$.
Note: The accumulate function $R(r_3, r_2, r_1, r_0, t)$ executes:

```
    $s \leftarrow t \gg_{\nmid 8} 7, r_3 \leftarrow t \ll_{\nmid 8} 57$
    $r_3 \leftarrow r_3 \oplus (s \ll_8 64)$
    $r_2 \leftarrow r_2 \oplus (s \gg_8 64)$
    $r_1 \leftarrow r_1 \oplus (t \ll_8 56)$
    $r_0 \leftarrow r_0 \oplus (t \gg_8 72)$

 1: $r_0, r_1, r_2, r_3 \leftarrow 0$
 2: for $i \leftarrow 19$ downto $15$ by $4$ do
 3:     $R(r_3, r_2, r_1, r_0, t[i])$,        $t[i-7] \leftarrow t[i-7] \oplus r_0$
 4:     $R(r_0, r_3, r_2, r_1, t[i-1])$,   $t[i-8] \leftarrow t[i-8] \oplus r_1$
 5:     $R(r_1, r_0, r_3, r_2, t[i-2])$,   $t[i-9] \leftarrow t[i-9] \oplus r_2$
 6:     $R(r_2, r_1, r_0, r_3, t[i-3])$,   $t[i-10] \leftarrow t[i-10] \oplus r_3$
 7: end for
 8: $R(r_3, r_2, r_1, r_0, t[11])$,   $t[4] \leftarrow t[4] \oplus r_0$
 9: $R(r_0, r_3, r_2, r_1, t[10])$,   $t[3] \leftarrow t[3] \oplus r_1$
10: $t[2] \leftarrow t[2] \oplus r_2$,      $t[1] \leftarrow t[1] \oplus r_3$,    $t[0] \leftarrow t[0] \oplus r_0$
11: $r_0 \leftarrow m[9] \gg_8 64$,   $r_0 \leftarrow r_0 \gg_{\nmid 8} 7$,      $t[0] \leftarrow t[0] \oplus r_0$
12: $r_1 \leftarrow r_0 \ll_8 64$,      $r_1 \leftarrow r_1 \ll_{\nmid 8} 63$,    $t[1] \leftarrow t[1] \oplus r_1$
13: $r_1 \leftarrow r_0 \gg_{\nmid 8} 1$,       $t[2] \leftarrow t[2] \oplus r_1$
14: for $i \leftarrow 0$ to $9$ do $c[2i] \leftarrow store(t[i])$
15: $c[19] \leftarrow c[19] \wedge$ 0x7F
16: return $c$
```

5.2.6 Inversion

For inversion in $\mathbb{F}_{2^m}$ we implemented a variant of the Extended Euclidean Algorithm for polynomials [59] where the length of each temporary vector is tracked. Since this algorithm requires flexible left shifts by arbitrary amounts, we implemented the full algorithm in 64-bit mode. Some Assembly in the form of a compiler intrinsic was used to efficiently count the number of leading 0 bits to determine the highest set bit.

Table 5.1: Comparison of different software implementations of finite field arithmetic in two Intel Core 2 platforms. All timings are reported in cycles. Improvements are computed in comparison with the previous fastest result in a 65nm platform, since the related works do not present timings for field operations in a 45nm platform.

	Operation			
Implementation	$a^2 \bmod f$	$a^{\frac{1}{2}} \bmod f$	$a \cdot b \bmod f$	$a^{-1} \bmod f$
Hankerson et al. [67]	600	500	8200	162000
Beuchat et al. [98]	480	749	5438	–
This work (Core 2 65nm)	160	166	4030	149763
Improvement	66.7%	66.8%	25.9%	7.6%
This work (Core 2 45nm)	108	140	3785	149589

5.2.7 Implementation Timings

In this section, we present our timings for finite field arithmetic. We implemented arithmetic in $\mathbb{F}_{2^{1223}}$ with irreducible trinomial $f(z) = z^{1223} + z^{255} + 1$. This field is suitable for instantiations of the η_T pairing over supersingular binary curves at the 128-bit security level [67]. The C programming language was used in conjunction with compiler intrinsics for accessing vector instructions. The chosen compiler was GCC version 4.1.2 because it generated the fastest code from vector intrinsics, as already observed by [67]. The differences between our implementations in the 65nm and 45nm processors can be explained by the lower cost of the `PSLLDQ` and `PSHUFB` instructions in the newer generation after the introduction of the *Super Shuffle Engine* by Intel.

Field multiplication was implemented by a combination of one instance of Karatsuba and the LD method depicted as Algorithm 5.3. Karatsuba's splitting point was at 632 bits and the divide-and-conquer steps were also implemented with vector instructions. Note that our binary field multiplier precomputes two tables of 16 rows, while the multiplier implemented in [67] precomputes a single table. This increase in memory consumption is negligible when compared to the total memory capacity of the target platform.

5.3 Pairing Computation

Miller's Algorithm for pairing computation requires a rich mathematical framework. We briefly present some definitions and point the reader to more complete treatments of the subject presented in [22, 23].

5.3.1 Preliminary Definitions

An *admissible bilinear pairing* is an efficiently computable map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are additive groups of points in an elliptic curve E and $\mathbb{G}_T$ is a related multiplicative group. Let P, Q be r-torsion points. The computation of a bilinear pairing $e(P, Q)$ requires the construction and evaluation of a function $f_{r,P}$ such that $div(f_{r,P}) = r(P) - r(\mathcal{O})$ at a divisor $\mathcal{D}$ which is equivalent to $(Q) - (\mathcal{O})$. Miller constructs $f_{r,P}$ in stages by using a double-and-add method [58]. Let $g_{U,V} : E(\mathbb{F}_{q^k}) \rightarrow \mathbb{F}_{q^k}$ be the line equation through points U and V. If $U = V$, the line $g_{U,V}$ is the tangent to the curve at U. If $V = -U$, the line g_U is the shorthand for $g_{U,-U}$. A *Miller function* is any function $f_{c,P}$ with divisor $div(f_{c,P}) = c(P) - (cP) - (c-1)(\mathcal{O})$, $c \in \mathbb{Z}$. The following property is true for all integers $a, b \in \mathbb{Z}$ [23, Theorem 2]:

$$f_{a+b,P}(\mathcal{D}) = f_{a,P}(\mathcal{D}) \cdot f_{b,P}(\mathcal{D}) \cdot \frac{g_{aP,bP}(\mathcal{D})}{g_{(a+b)P}(\mathcal{D})}. \tag{5.1}$$

Direct corollaries are:

(i) $f_{1,P}(\mathcal{D}) = 1$.

(ii) $f_{a,P}(\mathcal{D}) = f_{a-1,P}(\mathcal{D}) \cdot \frac{g_{(a-1)P,P}(\mathcal{D})}{g_{aP}(\mathcal{D})}$.

(iii) $f_{2a,P}(\mathcal{D}) = f_{a,P}(\mathcal{D})^2 \cdot \frac{g_{aP,aP}(\mathcal{D})}{g_{2aP}(\mathcal{D})}$.

Miller's Algorithm is depicted in Algorithm 5.5. The work by Barreto et al. [23] later showed how to use the final exponentiation of the Tate pairing to eliminate the denominators involved in the algorithm and to evaluate $f_{r,P}$ at Q instead of the divisor $\mathcal{D}$. Additional optimizations published in the literature focus on minimizing the latency of the Miller loop, that is, reduce the length of r while keeping its low Hamming weight [1, 2, 3].

5.3.2 Related Work

In this work, we are interested in parallel algorithms for pairing computation with no static limits on scalability, or more precisely, algorithms in which the scalability is not restricted by the mathematical definition of the pairing. Practical limits will always exist when: (i) the communication cost is dominant; (ii) the cost of parallelization is higher than the cost of computation.

Several works already developed parallel strategies for the computation of pairings achieving mixed results. Grabher et al. [96] analyzes two approaches: parallel extension field arithmetic, which gives good results but has a clear limit on scalability; a parallel

Algorithm 5.5 Miller's Algorithm [58].

```
Input: $r = \sum_{i=0}^{\log_2(r)} r_i 2^i$, $P$, $\mathcal{D} = (Q + R) - (R)$
Output: $f_{r,P}(\mathcal{D})$.
 1: $T \leftarrow P$, $f \leftarrow 1$
 2: for $i = \lfloor \log_2(r) \rfloor - 1$ downto $0$ do
 3:     $f \leftarrow f^2 \cdot \frac{g_{T,T}(Q+R)g_{2T}(R)}{g_{2T}(Q+R)g_{T,T}(R)}$
 4:     $T \leftarrow 2T$
 5:     if $r_i = 1$ then
 6:         $f \leftarrow f \cdot \frac{g_{T,P}(Q+R)g_{T+P}(R)}{g_{T+P}(Q+R)g_{T,P}(R)}$
 7:         $T \leftarrow T + P$
 8:     end if
 9: end for
10: return $f$
```

Miller loop strategy for two processors, where lines 3-4 for all iterations in Miller's Algorithm are precomputed by one processor and both processors compute in parallel the iterations where $r_i = 1$. Because r frequently has a low Hamming weight, this strategy results in performance losses due to unbalanced computational costs between the processors.

Mitsunari [156] observes that the different iterations of the algorithm can be computed in parallel if the points T of different iterations are available and proposes a specialized version of the η_T pairing over $\mathbb{F}_{3^m}$ for parallel execution in 2 processors. In this version, all the values $(x_P{}^{\frac{1}{3}^i}, y_P{}^{\frac{1}{3}^i}, x_Q{}^{3^i}, y_Q{}^{3^i})$ used for line evaluation in the i-th iteration of the algorithm are precomputed and the Miller loop iterations are divided in sets of the same size. Hence load balancing is trivially achieved. Since the cost of cubing and cube root computation is small, this approach achieves good speedups ranging from 1.61 to 1.76 at two different security levels. However, it requires significant storage overhead, since $4 \cdot (\frac{m+1}{2})$ field elements must be precomputed and stored. This approach is generalized and extended in the work by Beuchat et al. [98], where results are presented for fields of characteristic 2 and 3 at the 128-bit security level. For characteristic 2, the speedups achieved by parallel execution reach 1.75, 2.53 and 2.57 for 2, 4, and 8 processors, respectively. For characteristic 3, the speedups reach 1.65, 2.26 and 2.79, respectively. This parallelization represents the current state-of-the-art in parallel implementations of cryptographic pairings.

Cesena and Avanzi [157, 158] propose a technique to compute pairings over trace zero varieties constructed from supersingular elliptic curves and extensions with degrees $a = 3$ or $a = 5$. This approach allows a pairing computation to be packed in a short parallel Miller loops by the action of the a-th power of Frobenius. The problem with this approach is again the scalability limit (restricted by the extension degree a). The speedup achieved with parallel execution in 3 processors is 1.11 over a serial implementation of the η_T pairing at the same security level [158].

5.3.3 Parallelization

In this section, a parallelization of Miller's Algorithm is derived. This parallelization can be used to accelerate serial pairing implementations or improve the scalability of parallel approaches restricted by the pairing definition. This formulation is similar to the parallelization presented by [156] and [98], but our method focuses on minimizing the number of points needed for parallel executions of different iterations of the algorithm. This allows us to eliminate the overhead of storing $4(\frac{m+1}{2})$ precomputed field elements.

Miller's Algorithm computes $f_{r,P}$ in $\log_2(r)$ iterations. For a parallel algorithm, we must divide these $\log_2(r)$ iterations between some number π of processors. To achieve this, first we need a simple property of Miller functions [3, 24].

Lemma 5.3.1. *Let P, Q be points on $E(\mathbb{F}_q)$, $\mathcal{D} \sim (Q) - (\infty)$ and $f_{c,P}$ denote a Miller function. For all integers $a, b \in \mathbb{Z}$, $f_{a\cdot b,P}(\mathcal{D}) = f_{b,P}(\mathcal{D})^a \cdot f_{a,bP}(\mathcal{D})$.*

We need this property because Equation (5.1) just divides a Miller's Algorithm instance computed in $\log_2(r)$ iterations in two instances computed in at least $\log_2(r) - 1$ iterations. If we could represent r as a product $r_0 \cdot r_1$, it would be possible to compute $f_{r,P}$ in two instances of $\frac{\log_2(r)}{2}$ iterations. Since for some pairing instantiations, r is a prime group order, we write r in the simple and flexible form $2^w r_1 + r_0$, with $w \sim \frac{\log_2(r)}{2}$. This way, we can compute:

$$f_{r,P}(\mathcal{D}) = f_{2^w r_1 + r_0,P}(\mathcal{D}) = f_{2^w r_1,P}(\mathcal{D}) \cdot f_{r_0,P}(\mathcal{D}) \cdot \frac{g_{(2^w r_1)P, r_0 P}(\mathcal{D})}{g_{rP}(\mathcal{D})}. \tag{5.2}$$

The previous Lemma provides two choices to further develop $f_{2^w r_1,P}(\mathcal{D})$:

(i) $f_{2^w r_1,P}(\mathcal{D}) = f_{r_1,P}(\mathcal{D})^{2^w} \cdot f_{2^w, r_1 P}(\mathcal{D})$.

(ii) $f_{2^w r_1,P}(\mathcal{D}) = f_{2^w,P}(\mathcal{D})^{r_1} \cdot f_{r_1, 2^w P}(\mathcal{D})$.

The choice can be made based on efficiency: (i) compute w squarings in the extension field $\mathbb{F}^*_{q^k}$ and a point multiplication by r_1; (ii) compute an exponentiation to r_1 in the extension field and a point multiplication by 2^w (or w repeated point doublings). In the general case, the most efficient strategy will depend on the curve and embedding degree. The higher the embedding degree, the higher the cost of exponentiation in the extension field in comparison with point multiplication in the elliptic curve. If r has low Hamming weight, the two strategies should have similar costs. We adopt the first strategy:

$$f_{r,P}(\mathcal{D}) = f_{r_1,P}(\mathcal{D})^{2^w} \cdot f_{2^w, r_1 P}(\mathcal{D}) \cdot f_{r_0,P}(\mathcal{D}) \cdot \frac{g_{(2^w r_1)P, r_0 P}(\mathcal{D})}{g_{rP}(\mathcal{D})}. \tag{5.3}$$

This formula is clearly suitable for parallel execution in $\pi = 3$ processors, since each Miller function can be computed in $\frac{\log_2(r)}{2}$ iterations. For our purposes, however, r will have low Hamming weight and r_0 will be very small. In this case, $f_{r,P}$ can be computed by two Miller functions of approximately $\frac{\log_2(r)}{2}$ iterations. The parameter w can be adjusted to balance the costs in both processors (w extension field squarings with a point multiplication by r_1).

This formula can also be applied recursively for $f_{r_1,P}$ and f_{2^w,r_1P} to develop a parallelization suitable for any number of processors. Observe that π also does not have to be a power of 2, because of the flexible way we write r to exploit parallelism. An important detail is that a parallel implementation will only have significant speedups if the cost of the Miller loop is dominant over the communication overhead or the parallelization overhead. It is also important to note that the higher the number of processors, the higher the number of squarings and the smaller the constants r_i involved in point multiplication. However, applying the formula recursively can increase the size of the integers which multiply P, because they will be a product of r_i constants. Thus, the scalability of this algorithm for π processors depends on the cost of squarings in the extension field, the cost of point multiplications by r_i in the elliptic curve and the actual length of the Miller loop. Fortunately, these parameters are constant and can be statically determined. If P is fixed (a private key, for example), the multiples r_iP can also be precomputed and stored with low storage overhead.

5.3.4 Parallel η_T Pairing

In this section, the performance gain of a parallel implementation of the η_T pairing over a serial implementation is investigated following the analysis by [67].

Let E be a supersingular curve with embedding degree $k = 4$ defined over $\mathbb{F}_{2^m}$ with equation $E/\mathbb{F}_{2^m} : y^2 + y = x^3 + x + b$. The order of E is $2^m + 1 \pm 2^{\frac{m+1}{2}}$. A quartic extension is built over $\mathbb{F}_{2^m}$ with basis $\{1, s, t, st\}$, where $s^2 = s + 1$ and $t^2 = t + s$. Let $P, Q \in E(\mathbb{F}_{2^m})$ be r-torsion points. An associated distortion map ψ from $E(\mathbb{F}_{2^m})[r]$ to $E(\mathbb{F}_{2^{4m}})$ is defined by $\psi : (x, y) \to (x + s^2, y + sx + t)$. For this family of curves, Barreto et al. [1] defined the optimized η_T pairing:

$$\begin{aligned} \eta_T &: E(\mathbb{F}_{2^m})[r] \times E(\mathbb{F}_{2^{4m}})[r] \to \mathbb{F}^*_{2^{4m}}, \\ \eta_T(P, Q) &= f_{T',P'}(Q')^M, \end{aligned} \tag{5.4}$$

with $Q' = \psi(Q)$, $T' = (-v)(2^m - \#E(\mathbb{F}_{2^m}))$, $P' = (-v)P$, $M = (2^{2m} - 1)(2^m + 1 \pm 2^{\frac{m+1}{2}})$ for a curve-dependent parameter $v \in \{-1, 1\}$.

At the 128-bit security level, the base field must have $m = 1223$ bits [67]. Let E_1 be the supersingular curve with embedding degree $k = 4$ defined over $\mathbb{F}_{2^{1223}}$ with equation

$E_1(\mathbb{F}_{2^{1223}}) : y^2 + y = x^3 + x$. The order of E_1 is $5r = 2^{1223} + 2^{612} + 1$, where r is a 1221-bit prime number. Applying the parallel form developed in Section 5.3.3, the pairing computation can be decomposed in:

$$f_{T',P'}(Q')^M = \left(f_{2^{612-w},P'}(Q')^{2^w} \cdot f_{2^w,2^{612-w}P'}(Q') \cdot \frac{g_{2^{612-w}P',P'}(Q')}{g_{T'P'}(Q')} \right)^M .$$

Since squarings in $\mathbb{F}_{2^{4m}}$ and point duplication in supersingular curves require only binary field squarings and these can be efficiently computed, the cost of parallelization is low, but further improvements are possible. Barreto *et al.* [1] proposed a closed formula for this pairing based on a reversed-loop approach with square roots which eliminates the extension field squarings in Miller's Algorithm. Beuchat et al. [79] encountered further algorithmic improvements and proposed a slightly faster formula for the η_T pairing computation. We can obtain a parallel algorithm directly from the parallel formula derived above by excluding the involved extension field squarings and simply dividing the loop iterations between the processors. This algorithm is shown as Algorithm 5.6. In this algorithm, each processor i starts the loop from the w_i counter, computing w_i squarings/square roots of overhead. Without extension field squarings to offset these operations, it makes sense to assign processor 1 the first line evaluation and to increase the loop parts executed by processors with small w_i. The total overhead is smaller because extension field squarings are not needed and point arithmetic in binary supersingular curves can be computed with inexpensive squarings and square roots. Observe that the combining step can be implemented in at least two different ways: (i) serial combining of results with $(\pi - 1)$ serial extension field multiplications executed in one processor; (ii) parallel logarithmic combining of results with latency of $\lceil \log_2(\pi) \rceil$ extension field multiplications. We adopt the parallel strategy for efficiency.

5.3.5 Performance Analysis

Now we proceed with performance analysis of Algorithm 5.6. Processor 1 has an initialization cost of 3 multiplications and 2 squarings. Processor i has a parallelization cost of $2w_i$ squarings and $2w_i$ square roots. Additional parallelization overhead is $\lceil \log_2(\pi) \rceil$ extension field multiplications to combine the results. A full extension field multiplication costs 9 field multiplications. Each iteration of the algorithm executes 2 square roots, 2 squarings, 1 field multiplication and 1 extension field multiplication. Exploring the sparsity of G_i, this extension field multiplication costs 6 field multiplications. The final exponentiation has a cost of 26 multiplications, 7 finite field squarings, 612 extension field squarings and 1 inversion. Each extension field squaring costs 4 finite field squarings [79].

Let $\widetilde{m}, \widetilde{s}, \widetilde{r}, \widetilde{i}$ be the cost of finite field operations: multiplication, squaring, square root and inversion, respectively. For our efficient implementation of finite field $\mathbb{F}_{2^{1223}}$ in an

Intel Core 2 65nm processor, we have $\widetilde{r} \approx \widetilde{s}$, $\widetilde{m} \approx 25\widetilde{s}$ and $\widetilde{i} \approx 37\widetilde{m}$. From these ratios, we will illustrate how to compute the optimal w_i values which balance the computational cost between processors. Let $c_\pi(i)$ be the computational cost of a processor $0 < i \leq \pi$ while executing its portion of the parallel algorithm. For $\pi = 2$ processors:

$$\begin{aligned} c_2(1) &= (3\widetilde{m} + 2\widetilde{s}) + (7\widetilde{m} + 4\widetilde{s})w_2 = 80\widetilde{s} + (186\widetilde{s})w_2 \\ c_2(2) &= (4\widetilde{s})w_2 + (7\widetilde{m} + 4\widetilde{s})(611 - w_2). \end{aligned}$$

Algorithm 5.6 Proposed parallelization of the η_T pairing (π processors).

```
Input: $P = (x_P, y_P), Q = (x_Q, y_Q) \in E(\mathbb{F}_{2^m})[r]$, starting point $w_i$ for processor $i$.
Output: $\eta_T(P, Q) \in \mathbb{F}^*_{2^{4m}}$.
 1: $y_P \leftarrow y_P + 1 - \delta$
 2: parallel section(processor $i$)
 3: if $i = 1$ then
 4:     $u_i \leftarrow x_P + \alpha, v_i \leftarrow x_Q + \alpha$
 5:     $g_{0i} \leftarrow u_i \cdot v_i + y_P + y_Q + \beta$
 6:     $g_{1i} \leftarrow u_i + x_Q, g_{2i} \leftarrow v_i + x_P^2$
 7:     $G_i \leftarrow g_{0i} + g_{1i}s + t$
 8:     $L_i \leftarrow (g_{0i} + g_{2i}) + (g_{1i} + 1)s + t$
 9:     $F_i \leftarrow L_i \cdot G_i$
10: else
11:     $F_i \leftarrow 1$
12: end if
13: $x_{Q_i} \leftarrow (x_Q)^{2^{w_i}}, y_{Q_i} \leftarrow (y_Q)^{2^{w_i}}$
14: $x_{P_i} \leftarrow (x_P)^{\frac{1}{2^{w_i}}}, y_{P_i} \leftarrow (y_P)^{\frac{1}{2^{w_i}}}$
15: for $j \leftarrow w_i$ to $w_{i+1} - 1$ do
16:     $x_{P_i} \leftarrow \sqrt{x_{P_i}}, y_{P_i} \leftarrow \sqrt{y_{P_i}}, x_{Q_i} \leftarrow x_{Q_i}^2, y_{Q_i} \leftarrow y_{Q_i}^2$
17:     $u_i \leftarrow x_{P_i} + \alpha, v_i \leftarrow x_{Q_i} + \alpha$
18:     $g_{0i} \leftarrow u_i \cdot v_i + y_{P_i} + y_{Q_i} + \beta$
19:     $g_{1i} \leftarrow u_i + x_{Q_i}$
20:     $G_i \leftarrow g_{0i} + g_{1i}s + t$
21:     $F_i \leftarrow F_i \cdot G_i$
22: end for
23: $F \leftarrow \prod_{i=1}^{\pi} F_i$
24: end parallel
25: return $F^M$
```

Naturally, we always have $w_1 = 0$ and $w_{\pi+1} = 611$. Solving $c_2(1) = c_2(2)$ for w_2, we can obtain the optimal $w_2 = 309$. For $\pi = 4$ processors, we solve $c_4(1) = c_4(2) = c_4(3) = c_4(4)$ to obtain $w_2 = 158, w_3 = 312, w_4 = 463$. Observe that by solving a simple system of equations it is always possible to balance the computational cost between the processors. Furthermore, the latency of the Miller loop will always be equal to $c_\pi(1)$. Let $c_1(1)$ be the

Table 5.2: Estimated speedups for our parallelization of the η_T pairing over supersingular binary curves at the 128-bit security level. The optimal partitions were computed by a Sage[1] script.

	Number π of processors					
Estimated speedup $s(\pi)$	**1**	**2**	**4**	**8**	**16**	**32**
Core 2 65nm	1.00	1.90	3.45	5.83	8.69	11.48
Core 2 45nm	1.00	1.92	3.54	6.11	9.34	12.66

cost of a serial implementation of the main loop, *par* be the parallelization overhead and *exp* be the cost of final exponentiation. Considering the additional $\lceil \log_2(\pi) \rceil$ extension field multiplications as parallelization overhead and $26\widetilde{m} + (7 + 2446)\widetilde{s} + \widetilde{i}$ as the cost of final exponentiation, the speedup for π processors is the ratio between the cost of the serial implementation over the cost of the parallel implementation:

$$s(\pi) = \frac{c_1(1) + exp}{c_\pi(1) + par + exp} = \frac{77 + 179 \cdot 611 + 3978}{c_\pi(1) + 225\lceil \log_2(\pi) \rceil + 3978}.$$

Table 5.2 presents speedups estimated by our performance analysis. Note that our efficient implementation of binary field arithmetic in a 45nm processor has a bigger multiplication-to-squaring ratio, concentrating higher computational costs in the main loop of the algorithm. This explains why the speedups should be higher in the 45nm processor.

5.4 Experimental Results

We implemented the parallel algorithm for the η_T pairing over our efficient binary field arithmetic in two Intel Core platforms: an Intel Core 2 Quad 65nm platform running at 2.4GHz (Platform 1) and a dual quad-core Intel Xeon 45nm processor running at 2.0GHz (Platform 2). The parallel sections were implemented with OpenMP[2] constructs. OpenMP is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran. We used a special version of the GCC 4.1.2 compiler included in Fedora Linux 8 with OpenMP support backported from GCC 4.2 and SSSE3 support backported from GCC 4.3. This way, we could use both multiprocessing support and fast code generation for SSE intrinsics.

The timings and speedups presented in Table 5.3 were measured on 10^4 executions of each algorithm. We present timings in millions of cycles to ignore differences in clock frequency between the target platforms. From the table, we can observe that real implementations can obtain speedups close to the estimated speedups derived in the previous section. We verified that threading creation and synchronization overhead stayed in the

Table 5.3: Experimental results for serial/parallel executions of the η_T pairing. Times are presented in millions of cycles and the speedups are computed by the ratio between execution times of serial implementations over execution times of parallel implementations. The columns marked with (*) present estimates based on per-thread data.

	Number of threads					
Platform 1 – Intel Core 2 65nm	**1**	**2**	**4**	**8***	**16***	**32***
Hankerson et al. [67] – latency	39	–	–	–	–	–
Beuchat et al. [98] – latency	26.86	16.13	10.13	–	–	–
Beuchat et al. [98] – speedup	1.00	1.67	2.65	–	–	–
This work – latency	18.76	10.08	5.72	3.55	2.51	2.14
This work – speedup	1.00	1.86	3.28	5.28	7.47	8.76
Improvement	30.2%	37.5%	43.5%	–	–	–
Platform 2 – Intel Core 2 45nm	**1**	**2**	**4**	**8**	**16***	**32***
Beuchat et al. [98] – latency	23.03	13.14	9.08	8.93	–	–
Beuchat et al. [98] – speedup	1.00	1.77	2.54	2.58	–	–
This work – latency	17.40	9.34	5.08	3.02	2.03	1.62
This work – speedup	1.00	1.86	3.42	5.76	8.57	10.74
Improvement	24.4%	28.9%	44.0%	66.2%	–	–

order of microseconds, being negligible compared to the pairing computation time. Timings for $\pi > 4$ processors in Platform 1 and $\pi > 8$ processors in Platform 2 were measured through a high-precision per-thread counter measured by the main thread. These timings might be an accurate approximation of future real implementations, but memory effects (such as cache locality) or scheduling influence may impose penalties.

Table 5.3 shows that the proposed parallelization presents good scalability. We improve the state-of-the-art serial and parallel execution times significantly. The fastest timing for computing the η_T pairing obtained by our implementation was 1.51 milliseconds using all 8 cores of Platform 2. The work by Beuchat *et al.* [98] reports a timing of 3.08 milliseconds in a Intel Core i7 45nm processor clocked at 2.9GHz. Note that we obtain a much faster timing with a lower clock frequency and without requiring the storage overhead of $4 \cdot (\frac{m+1}{2})$ field elements present in [98], which may reach 365KB for these parameters and be prohibitive in resource-constrained embedded devices.

[1]SAGE: Software for Algebra and Geometry Experimentation.
`http://www.sagemath.org`
[2]Open Multi-Processing. `http://www.openmp.org`

5.5 Conclusion and Future Work

In this work, we proposed novel techniques for exploring parallelism during the implementation of the η_T pairing over supersingular binary curves in modern multi-core computers. Powerful vector instructions of the SSE family were shown to accelerate considerably the arithmetic in binary fields. We obtained significant performance in computing the η_T pairing, using an efficient implementation of field multiplication, squaring and square root computation. The optimizations improved the state-of-the-art timings of this pairing instantiation at the 128-bit security level by 24% and 30% in two different Intel Core processors.

We also derived a parallelization of Miller's Algorithm to compute pairings. This parallelization is generic and can be applied to any pairing algorithm or instantiation. The construction also achieves good scalability in the symmetric case and this scalability is not restricted by the definition of the pairing. We illustrated the formulation when applied to the η_T pairing over supersingular binary curves and validated our performance analysis with a real implementation. The experimental results show that the actual implementation could sustain performance gains close to the estimated speedups. Parallel execution of the η_T pairing improved the state-of-the-art timings by at least 28%, 44% and 66% in 2, 4 and 8 cores respectively. This parallelization is suitable for embedded platforms and can be applied to reduce computation latency when response time is critical.

Future work can adapt the introduced techniques for the case $\mathbb{F}_{3^m}$. Improvements to the parallelization should focus on minimizing the serial region and parallelization cost. The proposed parallelization should also be applied to an optimal asymmetric pairing setting, where parallelization costs are clearly higher. Preliminary data for the R-ate pairing [3] over Barreto-Naehrig curves at the 128-bit security level points to a 10% speedup using 2 processor cores.

Chapter 6

Faster Explicit Formulas for Computing Pairings over Ordinary Curves

Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys and Julio López

Abstract

We describe efficient formulas for computing pairings on ordinary elliptic curves over prime fields. First, we generalize lazy reduction techniques, previously considered only for arithmetic in quadratic extensions, to the whole pairing computation, including towering and curve arithmetic. Second, we introduce a new compressed squaring formula for cyclotomic subgroups and a new technique to avoid performing an inversion in the final exponentiation when the curve is parameterized by a negative integer. The techniques are illustrated in the context of pairing computation over Barreto-Naehrig curves, where they have a particularly efficient realization, and are also combined with other important developments in the recent literature. The resulting formulas reduce the number of required operations and, consequently, execution time, improving on the state-of-the-art performance of cryptographic pairings by 28%-34% on several popular 64-bit computing platforms. In particular, our techniques allow to compute a pairing under 2 million cycles for the first time on such architectures.

Publication

This work was originally published in the Proceedings of the 30th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2011) and is available at `http://www.springer.com/computer/security+and+cryptology/book/978-3-642-20464-7`. ©2011 IACR.

6.1 Introduction

The performance of pairing computation has received increasing interest in the research community, mainly because Pairing-Based Cryptography enables efficient and elegant solutions to several longstanding problems in cryptography such as Identity-Based Encryption [12, 13], powerful non-interactive zero-knowledge proof systems [159] and efficient multi-party key agreements [16]. Recently, dramatic improvements over the figure of 10 million cycles presented in [67] made possible to compute a pairing at the 128-bit security level in 4.38 million cycles [160] when using high-speed vector floating-point operations, and 2.33 million cycles [99] when the fastest integer multiplier available in Intel 64-bit architectures is employed.

This work revisits the problem of efficiently computing pairings over large-characteristic fields and improves the state-of-the-art performance of cryptographic pairings by a significant margin. First of all, it builds on the latest advancements proposed by several authors:

- The Optimal Ate pairing [24] computed entirely on twists [26] with simplified final line evaluations [160] over a recently-introduced subclass [97] of the Barreto-Naehrig (BN) family of pairing-friendly elliptic curves [74].
- The implementation techniques described by [99] for accelerating quadratic extension field arithmetic, showing how to reduce expensive carry handling and function call overheads.

On the other hand, the following new techniques are introduced:

- The notion of lazy reduction, usually applied for arithmetic in quadratic extensions in the context of pairings, as discussed in [19], is generalized to the towering and curve arithmetic performed in the pairing computation. In a sense, this follows a direction opposite to the one taken by other authors. Instead of trying to encode arithmetic so that modular reductions are faster [161, 160], we insist on Montgomery reduction and focus our efforts on reducing *the need* of computing reductions. Moreover, for dealing with costly higher-precision additions inserted by lazy reduction, we develop a flexible methodology that keeps intermediate values under Montgomery reduction boundaries and maximizes the use of operations without carry checks. The traditional operation count model is also augmented to take into account modular reductions individually.
- Formulas for point doubling and point addition in Jacobian and homogeneous coordinates are carefully optimized by eliminating several commonly neglected operations that are not inexpensive on modern 64-bit platforms.

- The computation of the final exponentiation is improved with a new set of formulas for compressed squaring and efficient decompression in cyclotomic subgroups, and an arithmetic trick to remove a significant penalty incurred when computing pairings over curves parameterized by negative integers.

The described techniques produce significant savings, allowing our illustrative software implementation to compute a pairing under 2 million cycles and improve the state-of-the-art timings by 28%-34% on several different 64-bit computing platforms. Even though the techniques are applied on pairings over BN curves at the 128-bit security level, they can be easily extended to other settings using different curves and higher security levels [25].

This paper is organized as follows. Section 6.2 gives an overview of Miller's Algorithm when employed for computing the Optimal Ate pairing over Barreto-Naehrig curves. Section 6.3 presents the generalized lazy reduction technique and its application to the improvement of towering arithmetic performance. Different optimizations to curve arithmetic, including the application of lazy reduction, are discussed in Section 6.4. Section 6.5 describes our improvements on the final exponentiation. Section 6.6 summarizes operation counts and Section 6.7 describes our high-speed software implementation and comparison of results with the previously fastest implementation in the literature. Section 6.8 concludes the paper.

6.2 Preliminaries

An *admissible bilinear pairing* is a non-degenerate efficiently-computable map $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$, where $\mathbb{G}_1$ and $\mathbb{G}_2$ are additive groups of points in an elliptic curve E and $\mathbb{G}_T$ is a subgroup of the multiplicative group of a finite field. The core property of map e is linearity in both arguments, allowing the construction of novel cryptographic schemes with security relying on the hardness of the Discrete Logarithm Problem in $\mathbb{G}_1, \mathbb{G}_2$ and $\mathbb{G}_T$.

Barreto and Naehrig [74] described a parameterized family of elliptic curves $E_b : y^2 = x^3 + b, b \neq 0$ over a prime field $\mathbb{F}_p$, $p = 36u^4 + 36u^3 + 24u^2 + 6u + 1$, with prime order $n = 36u^4 + 36u^3 + 18u^2 + 6u + 1$, where $u \in \mathbb{Z}$ is an arbitrary integer. This family is rather large and easy to generate [97], providing a multitude of parameter choices; and, having embedding degree $k = 12$, is well-suited for computing asymmetric pairings at the 128-bit security level [19]. It admits several optimal derivations [24] of different variants of the Ate pairing [2] such as R-ate [3], Optimal Ate [24] and χ-ate [162].

Let $E[n]$ be the subgroup of n-torsion points of E and $E' : y^2 = x^3 + b/\xi$ be a sextic twist of E with ξ not a cube nor a square in $\mathbb{F}_{p^2}$. For the clear benefit of direct benchmarking, but also pointing that performance among variants is roughly the same, we restrict the discussion to computing the Optimal Ate pairing defined as in [160]:

$$\begin{array}{rcl} a_{opt} : \mathbb{G}_2 \times \mathbb{G}_1 & \to & \mathbb{G}_T \\ (Q, P) & \to & \left(f_{r,Q}(P) \cdot l_{[r]Q,\pi_p(Q)}(P) \cdot l_{[r]Q+\pi_p(Q),-\pi_p^2(Q)}(P)\right)^{\frac{p^{12}-1}{n}}, \end{array}$$

where $r = 6u + 2 \in \mathbb{Z}$; the map $\pi_p : E \to E$ is the Frobenius endomorphism $\pi_p(x, y) = (x^p, y^p)$; groups $\mathbb{G}_1, \mathbb{G}_2$ are determined by the eigenspaces of π_p as $\mathbb{G}_1 = E[n] \cap \mathrm{Ker}(\pi_p - [1]) = E(\mathbb{F}_p)[n]$ and $\mathbb{G}_2$ as the preimage $E'(\mathbb{F}_{p^2})[n]$ of $E[n] \cap \mathrm{Ker}(\pi_p - [p]) \subseteq E(\mathbb{F}_{p^{12}})[n]$ under the twisting isomorphism $\psi : E' \to E$; the group $\mathbb{G}_T$ is the subgroup of n-th roots of unity $\mu_n \subset \mathbb{F}^*_{p^{12}}$; $f_{r,Q}(P)$ is a normalized function with divisor $(f_{r,Q}) = r(Q) - ([r]Q) - (r - 1)(\mathcal{O})$ and $l_{Q_1,Q_2}(P)$ is the line arising in the addition of Q_1 and Q_2 evaluated at point P.

Miller [8, 58] proposed an algorithm that constructs $f_{r,P}$ in stages by using a double-and-add method. When generalizing the denominator-free version [23] of Miller's Algorithm for computing the pairing a_{opt} with the set of implementation-friendly parameters suggested by [97] at the 128-bit security level, we obtain Algorithm 6.1. For the BN curve we have $E : y^2 = x^3 + 2, u = -(2^{62} + 2^{55} + 1) < 0$. In order to accommodate the negative r (line 9 in Algorithm 6.1), it is required to compute a cheap negation in $\mathbb{G}_2$ to make the final accumulator T the result of $[-|r|]Q$, and an expensive inversion in the big field $\mathbb{G}_T$ to obtain the correct pairing value $f_{-|r|,Q}(P) = (f_{|r|,Q}(P))^{-1}$, instead of the value $f_{|r|,Q}(P)$ produced at the end of the algorithm. The expensive inversion will be handled later at Section 6.5 with the help of the final exponentiation.

6.3 Tower Extension Field Arithmetic

Miller's Algorithm [8, 58] employs arithmetic in $\mathbb{F}_{p^{12}}$ during the accumulation steps (lines 3,5,11-12 in Algorithm 6.1) and at the final exponentiation (line 13 in the same algorithm). Hence, to achieve a high-performance implementation of pairings it is crucial to perform arithmetic over extension fields efficiently. In particular, it has been recommended in [163] to represent $\mathbb{F}_{p^k}$ with a tower of extensions using irreducible binomials. Accordingly, in our targeted setting we represent $\mathbb{F}_{p^{12}}$ using the flexible towering scheme used in [164, 67, 99, 97] combined with the parameters suggested by [97]:

- $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2 - \beta)$, where $\beta = -1$.
- $\mathbb{F}_{p^4} = \mathbb{F}_{p^2}[s]/(s^2 - \xi)$, where $\xi = 1 + i$.
- $\mathbb{F}_{p^6} = \mathbb{F}_{p^2}[v]/(v^3 - \xi)$, where $\xi = 1 + i$.
- $\mathbb{F}_{p^{12}} = \mathbb{F}_{p^4}[t]/(t^3 - s)$ or $\mathbb{F}_{p^6}[w]/(w^2 - v)$.

Algorithm 6.1 Optimal Ate pairing on BN curves (generalized for $u < 0$).

```
Input: $P \in \mathbb{G}_1, Q \in \mathbb{G}_2, r = |6u+2| = \sum_{i=0}^{\log_2(r)} r_i 2^i$
Output: $a_{opt}(Q, P)$
 1: $T \leftarrow Q,\ f \leftarrow 1$
 2: for $i = \lfloor \log_2(r) \rfloor - 1$ downto $0$ do
 3:     $f \leftarrow f^2 \cdot l_{T,T}(P), T \leftarrow 2T$
 4:     if $r_i = 1$ then
 5:         $f \leftarrow f \cdot l_{T,Q}(P), T \leftarrow T + Q$
 6: end for
 7: $Q_1 \leftarrow \pi_p(Q), Q_2 \leftarrow \pi_p^2(Q)$
 8: if $u < 0$ then
 9:     $T \leftarrow -T, f \leftarrow f^{-1}$
10: end if
11: $f \leftarrow f \cdot l_{T,Q_1}(P), T \leftarrow T + Q_1$
12: $f \leftarrow f \cdot l_{T,-Q_2}(P), T \leftarrow T - Q_2$
13: $f \leftarrow f^{(p^{12}-1)/n}$
14: return $f$
```

It is possible to convert from one towering $\mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$ to the other $\mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^4} \rightarrow \mathbb{F}_{p^{12}}$ by simply permuting the order of coefficients. The choice $p \equiv 3 \pmod 4$ accelerates arithmetic in $\mathbb{F}_{p^2}$, since multiplications by $\beta = -1$ can be computed as simple subtractions [97].

6.3.1 Lazy Reduction for Tower Fields

The concept of lazy reduction goes back to at least [165] and has been advantageously exploited by many works in different scenarios [166, 167, 19]. Lim and Hwang [166] showed that multiplication in $\mathbb{F}_{p^k}$, when $\mathbb{F}_{p^k} = \mathbb{F}_p[x]/(x^k - w)$ is seen as a direct extension over $\mathbb{F}_p$ via the irreducible binomial $(x^k - w)$ with $w \in \mathbb{F}_p$, can be performed with k reductions modulo p. In contrast, it would normally require either k^2 reductions using conventional multiplication, or $k(k+1)/2$ reductions using Karatsuba multiplication. Lazy reduction was first employed in the context of pairing computation by [19] to eliminate reductions in $\mathbb{F}_{p^2}$ multiplication. If one considers the tower $\mathbb{F}_p \rightarrow \mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$, then this approach requires $2 \cdot 6 \cdot 3 = 36$ reductions modulo p, and $3 \cdot 6 \cdot 3 = 54$ integer multiplications for performing one multiplication in $\mathbb{F}_{p^{12}}$; see [19, 67, 99].

In this section, we generalize the lazy reduction technique to towering-friendly fields $\mathbb{F}_{p^k}$, $k = 2^i 3^j$, $i \geq 1, j \geq 0$, conveniently built with irreducible binomials [168]. We show that multiplication (and squaring) in a tower extension $\mathbb{F}_{p^k}$ only requires k reductions and still benefits from different arithmetic optimizations available in the literature to reduce the number of subfield multiplications or squarings. For instance, with our approach one now requires $2 \cdot 3 \cdot 2 = 12$ reductions modulo p and 54 integer multiplications using the tower $\mathbb{F}_p \to \mathbb{F}_{p^2} \to \mathbb{F}_{p^6} \to \mathbb{F}_{p^{12}}$ to compute one multiplication in $\mathbb{F}_{p^{12}}$; or 12 reductions modulo p and 36 integer multiplications to compute one squaring in $\mathbb{F}_{p^{12}}$. Although wider in generality, these techniques are analyzed in detail in the context of Montgomery multiplication and Montgomery reduction [169], which are commonly used in the context of pairings over ordinary curves. We explicitly state our formulas for the towering construction $\mathbb{F}_p \to \mathbb{F}_{p^2} \to \mathbb{F}_{p^6} \to \mathbb{F}_{p^{12}}$ in Section 6.3.3. To remove ambiguity, the term *reduction modulo p* always refers to modular reduction of double-precision integers.

Theorem 6.3.1. *Let $k = 2^i 3^j$, $i, j \in \mathbb{Z}$ and $i \geq 1, j \geq 0$. Let*

$$\mathbb{F}_p = \mathbb{F}_{p^{k_0}} \to \mathbb{F}_{p^{k_1}} = \mathbb{F}_{p^2} \to \cdots \to \mathbb{F}_{p^{k_{i+j-2}}} \to \mathbb{F}_{p^{k_{i+j-1}}} \to \mathbb{F}_{p^{k_{i+j}}} = \mathbb{F}_{p^k}$$

be a tower extension, where each extension $\mathbb{F}_{p^{k_{\ell+1}}}/\mathbb{F}_{p^{k_\ell}}$ is of degree either 2 or 3, which can be constructed using a second degree irreducible binomial $x^2 - \beta_\ell$, $\beta_\ell \in \mathbb{F}_{p^{k_\ell}}$, or a third degree irreducible binomial $x^3 - \beta_\ell$, $\beta_\ell \in \mathbb{F}_{p^{k_\ell}}$, respectively. Suppose that β_ℓ can be chosen such that, for all $a \in \mathbb{F}_{p^{k_\ell}}$, $a \cdot \beta_\ell$ can be computed without any reduction modulo p. Then multiplication in $\mathbb{F}_{p^k}$ can be computed with $3^i 6^j$ integer multiplications and $k = 2^i 3^j$ reductions modulo p for any k.

Proof. We prove this by induction on $i + j$. The base case is $i + j = 1$ ($i = 1$ and $j = 0$). That is, $k = 2$, and we have a tower $\mathbb{F}_p \to \mathbb{F}_{p^2}$ with $\mathbb{F}_{p^2} = \mathbb{F}_p[x]/(x^2 - \beta)$. For any $a = a_0 + a_1 x, b = b_0 + b_1 x \in \mathbb{F}_{p^2}$, $a_i, b_i \in \mathbb{F}_p$, we can write

$$a \cdot b = (a_0 b_0 + a_1 b_1 \beta) + ((a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1)x,$$

which can be computed with 3 integer multiplications and 2 reductions modulo p (note that we ignore multiplication by β, by our assumption).

Next, consider

$$\mathbb{F}_p \to \mathbb{F}_{p^2} \to \cdots \to \mathbb{F}_{p^{k_{i+j}}} \to \mathbb{F}_{p^{k_{i+j+1}}},$$

where $k_{i+j+1} = 2^{i+1} 3^j$, or $k_{i+j+1} = 2^i 3^{j+1}$. In the former case, let $\mathbb{F}_{p^{k_{i+j+1}}} = \mathbb{F}_{p^{k_{i+j}}}[x]/(x^2 - \beta)$ and $a = a_0 + a_1 x$, $b = b_0 + b_1 x \in \mathbb{F}_{p^{k_{i+j+1}}}$, a_i, $b_i \in \mathbb{F}_{p^{k_{i+j}}}$. Then

$$a \cdot b = (a_0 b_0 + a_1 b_1 \beta) + [(a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1]\, x, \tag{6.1}$$

which can be computed with 3 multiplications in $\mathbb{F}_{p^{k_{i+j}}}$, namely $a_0 b_0$, $a_1 b_1 \beta$ and $(a_0 + a_1)(b_0 + b_1)$ (again, we ignore multiplication by β). By the induction hypothesis, each multiplication in $\mathbb{F}_{p^{k_{i+j}}}$ requires $3^i 6^j$ integer multiplications, and $2^i 3^j$ reductions modulo p. Also, three reductions modulo p, when computing $a_0 b_0$, $a_1 b_1 \beta$ and $(a_0 + a_1)(b_0 + b_1)$, can be minimized to two reductions modulo p (see (6.1)). Hence, multiplication in $\mathbb{F}_{p^{k_{i+j+1}}}$ can be computed with $3 \cdot 3^i 6^j = 3^{i+1} 6^j$ integer multiplications and $2 \cdot 2^i 3^j = 2^{i+1} 3^j$ reductions modulo p.

The latter case, $k_{i+j+1} = 2^i 3^{j+1}$, can be proved similarly, by considering $\mathbb{F}_{p^{k_{i+j+1}}} = \mathbb{F}_{p^{k_{i+j}}}[x]/(x^3 - \beta)$, and the Karatsuba multiplication formula for degree 3 extensions instead of (6.1). $\square$

It is also straightforward to generalize the procedure above to any formula other than Karatsuba which also involves only sums (or subtractions) of products of the form $\sum \pm a_i b_j$, with $a_i, b_j \in \mathbb{F}_{p^{k_l}}$, such as complex squaring or the Chung-Hasan asymmetric squaring formulas [170].

For efficiency purposes, we suggest a different treatment for the highest layer in the tower arithmetic. Theorem 6.3.1 implies that reductions can be completely delayed to the end of the last layer by applying lazy reduction, but in some cases (when the optimal k is already reached and no reductions can be saved) it will be more efficient to perform reductions immediately after multiplications or squarings. This will be illustrated with the computation of squaring in $\mathbb{F}_{p^{12}}$ in Section 6.3.3.

In the Miller Loop, reductions can also be delayed from the underlying $\mathbb{F}_{p^2}$ field during multiplication and squaring to the arithmetic layer immediately above (i.e., the point arithmetic and line evaluation). Similarly to the tower extension, on this upper layer reductions should only be delayed in the cases where this technique leads to fewer reductions. For details, see Section 6.4.

There are some penalties when delaying reductions. In particular, *single-precision* operations (with operands occupying $n = \lceil \lceil \log_2 p \rceil / w \rceil$ words, where w is the computer word-size) are replaced by *double-precision* operations (with operands occupying $2n$ words). However, this disadvantage can be minimized in terms of speed by selecting a field size smaller than the word-size boundary because this technique can be exploited more extensively for optimizing double-precision arithmetic.

6.3.2 Selecting a Field Size Smaller than the Word-Size Boundary

If the modulus p is selected so that $l = \lceil \log_2 p \rceil < N$, where $N = n \cdot w$, n is the exact number of words required to represent p, i.e., $n = \lceil l/w \rceil$, and w is the computer word-size, then several consecutive additions without carry-out in the most significant word (MSW) can be performed before a multiplication of the form $c = a \cdot b$, where $a, b \in [0, 2^N - 1]$ such that $c < 2^{2N}$. In the case of Montgomery reduction, the restriction is given by the upper bound $c < 2^N \cdot p$. Similarly, when delaying reductions the result of a multiplication without reduction has maximum value $(p-1)^2 < 2^{2N}$ (assuming that $a, b \in [0, p]$) and several consecutive double-precision additions without carry-outs in the MSW (and, in some cases, subtractions without borrow-outs in the MSW) can be performed before reduction. When using Montgomery reduction up to $\sim \lfloor 2^N/p \rfloor$ additions can be performed without carry checks.

Furthermore, cheaper single- and double-precision operations exploiting this "extra room" can be combined for maximal performance. The challenge is to optimally balance their use in the tower arithmetic since both may interfere with each other. For instance, if intermediate values are allowed to grow up to $2p$ before multiplication (instead of p) then the maximum result would be $4p^2$. This strategy makes use of cheaper single-precision additions without carry checks but limits the number of double-precision additions that can be executed without carry checks after multiplication with delayed reduction. As it will be evident later, to maximize the gain obtained with the proposed methodology one should take into account relative costs of operations and maximum bounds.

In the case of double-precision arithmetic, different optimizing alternatives are available. Let us analyze them in the context of Montgomery arithmetic. First, as pointed out by [99], if $c > 2^N \cdot p$, where c is the result of a double-precision addition, then c can be restored with a cheaper single-precision subtraction by $2^N \cdot p$ (note that the first half of this value consists of zeroes only). Second, different options are available to convert negative numbers to positive after double-precision subtraction. In particular, let us consider the computation $c = a + l \cdot b$, where $a, b \in [0, mp^2]$, $m \in \mathbb{Z}^+$ and $l < 0 \in \mathbb{Z}$ s.t. $|lmp| < 2^N$, which is a recurrent operation (for instance, when $l = \beta$). For this operation, we have explored the following alternatives, which can be integrated in the tower arithmetic with different advantages: **Option 1:** $r = c + (2^N \cdot p/2^h)$, $r \in [0, mp^2 + 2^N \cdot p/2^h]$, h is a small integer s.t. $|lmp^2| < 2^N \cdot p/2^h < 2^N \cdot p - mp^2$.
Option 2: if $c < 0$ then $r = c + 2^N \cdot p$, $r \in [0, 2^N \cdot p]$.
Option 3: $r = c - lmp^2$, $r \in [0, (|l| + 1)mp^2]$, s.t. $(|l| + 1)mp < 2^N$.
Option 4: if $c < 0$ then $r = c - lmp^2, r \in [0, |lmp^2|]$.

In particular, Options 2 and 4 require conditional checks that make the corresponding operations more expensive. Nevertheless, these options may be valuable when negative values cannot be corrected with other options without violating the upper bound. Also note that Option 2 can make use of a cheaper single-precision subtraction for converting negative results to positive. Options 1 and 3 are particularly efficient because no conditional checks are required. Moreover, if l is small enough (and h maximized for Option 1) several following operations can avoid carry checks. Between both, Option 1 is generally more efficient because adding $2^N \cdot p/2^h$ requires less than double-precision if $h \leq w$, where w is the computer word-size.

Next, we demonstrate how the different design options discussed in this section can be exploited with a clever selection of parameters and applied to different operations combining single- and double-precision arithmetic to speed up the extension field arithmetic.

6.3.3 Analysis for Selected Parameters

For our illustrative analysis, we use the tower $\mathbb{F}_{p^2} \rightarrow \mathbb{F}_{p^6} \rightarrow \mathbb{F}_{p^{12}}$ constructed with the irreducible binomials described at the beginning of this section. When targeting the 128-bit security level, single- and double-precision operations are defined by operands with sizes $N = 256$ and $2N = 512$, respectively. For our selected prime, $\lceil \log_2 p \rceil = 254$ and $2^N \cdot p \approx 6.8p^2$. Notation is fixed as following: (i) $+, -, \times$ are operators not involving carry handling or modular reduction for boundary keeping; (ii) $\oplus, \ominus, \otimes$ are operators producing reduced results through carry handling or modular reduction; (iii) a superscript in an operator is used to denote the extension degree involved in the operation; (iv) notation $a_{i,j}$ is used to address j-th subfield element in extension field element a_i; (v) lower case t and upper case T variables represent single- and double-precision integers or extension field elements composed of single and double-precision integers, respectively. The precision of the operators is determined by the precision of the operands and result. Note that, as stated before, if $c > 2^N \cdot p$ after adding $c = a + b$ in double-precision, we correct the result by computing $c - 2^N \cdot p$. Similar to subtraction, we refer to the latter as "Option 2".

The following notation is used for the cost of operations: (i) m, s, a denote the cost of multiplication, squaring and addition in $\mathbb{F}_p$, respectively; (ii) $\tilde{m}, \tilde{s}, \tilde{a}, \tilde{i}$ denote the cost of multiplication, squaring, addition and inversion in $\mathbb{F}_{p^2}$, respectively; (iii) m_u, s_u, r denote the cost of unreduced multiplication and squaring producing double-precision results, and modular reduction of double-precision integers, respectively; (iv) $\tilde{m}_u, \tilde{s}_u, \tilde{r}$ denote the cost of unreduced multiplication and squaring, and modular reduction of double-precision elements in $\mathbb{F}_{p^2}$, respectively. For the remainder of the paper, and unless explicitly stated otherwise, we assume that double-precision addition has the cost of $2a$ and $2\tilde{a}$ in $\mathbb{F}_p$ and $\mathbb{F}_{p^2}$, respectively, which approximately follows what we observe in practice.

We will now illustrate a selection of operations for efficient multiplication in $\mathbb{F}_{p^{12}}$, beginning with multiplication in $\mathbb{F}_{p^2}$. Let $a, b, c \in \mathbb{F}_{p^2}$ such that $a = a_0 + a_1 i, b = b_0 + b_1 i, c = a \cdot b = c_0 + c_1 i$. The required operations for computing $\mathbb{F}_{p^2}$ multiplication are detailed in Algorithm 6.2. As explained in Beuchat *et al.* [99, Section 5.2], when using the Karatsuba method and $a_i, b_i \in \mathbb{F}_p$, $c_1 = (a_0 + a_1)(b_0 + b_1) - a_0 b_0 - a_1 b_1 = a_0 b_1 + a_1 b_0 < 2p^2 < 2^N \cdot p$, additions are single-precision, reduction after multiplication can be delayed and hence subtractions are double-precision (steps 1-3 in Algorithm 6.2). Obviously, these operations do not require carry checks. For $c_0 = a_0 b_0 - a_1 b_1$, c_0 is in interval $[-p^2, p^2]$ and a negative result can be converted to positive using **Option 1** with $h = 2$ or **Option 2**, for which the final c_0 is in the range $[0, (2^N \cdot p/4) + p^2] \subset [0, 2^N \cdot p]$ or $[0, 2^N \cdot p]$, respectively (step 4 in Algorithm 6.2). Following Theorem 6.3.1, all reductions can be completely delayed to the next arithmetic layer (higher extension or curve arithmetic).

Let us now define multiplication in $\mathbb{F}_{p^6}$. Let $a, b, c \in \mathbb{F}_{p^6}$ such that $a = (a_0 + a_1 v +$

Algorithm 6.2 Multiplication in $\mathbb{F}_{p^2}$ without reduction ($\times^2$, cost $\tilde{m}_u = 3m_u + 8a$)

```
Input: $a = (a_0 + a_1 i)$ and $b = (b_0 + b_1 i) \in \mathbb{F}_{p^2}$
Output: $c = a \cdot b = (c_0 + c_1 i) \in \mathbb{F}_{p^2}$
1: $T_0 \leftarrow a_0 \times b_0$, $T_1 \leftarrow a_1 \times b_1$, $t_0 \leftarrow a_0 + a_1$, $t_1 \leftarrow b_0 + b_1$
2: $T_2 \leftarrow t_0 \times t_1$, $T_3 \leftarrow T_0 + T_1$
3: $T_3 \leftarrow T_2 - T_3$
4: $T_4 \leftarrow T_0 \ominus T_1$                                        (Option 1 or 2)
5: return $c = (T_4 + T_3 i)$
```

$a_2v^2), b = (b_0 + b_1v + b_2v^2), c = a \cdot b = (c_0 + c_1v + c_2v^2)$. The required operations for computing $\mathbb{F}_{p^6}$ multiplication are detailed in Algorithm 6.3. In this case, $c_0 = v_0 + \xi[(a_1 + a_2)(b_1 + b_2) - v_1 - v_2], c_1 = (a_0 + a_1)(b_0 + b_1) - v_0 - v_1 + \xi v_2$ and $c_2 = (a_0 + a_2)(b_0 + b_2) - v_0 - v_2 + v_1$, where $v_0 = a_0b_0, v_1 = a_1b_1$ and $v_2 = a_2b_2$. First, note that the pattern $s_x = (a_i + a_j)(b_i + b_j) - v_i - v_j$ repeats for each $c_x, 0 \leq x \leq 2$. After multiplications using Alg. 6.2 with **Option 1** ($h = 2$), we have $v_{i,0}, v_{j,0} \in [0, (2^N \cdot p/4) + p^2]$ and $v_{i,1}, v_{j,1} \in [0, 2p^2]$ (step 1 of Alg. 6.3). Outputs of single-precision additions of the forms $(a_i + a_j)$ and $(b_i + b_j)$ are in the range $[0, 2p]$ and hence do not produce carries (steps 2, 9 and 17 of Alg. 6.3). Corresponding $\mathbb{F}_{p^2}$ multiplications $r_x = (a_i + a_j)(b_i + b_j)$ using Alg. 6.2 with **Option 2** give results in the ranges $r_{x,0} \in [0, 2^N \cdot p]$ and $r_{x,1} \in [0, 8p^2]$ (steps 3, 10 and 18). Although $max(r_{x,1}) = 8p^2 > 2^N \cdot p$, note that $8p^2 < 2^{2N}$ and $s_{x,1} = a_{i,0}b_{j,1} + a_{i,1}b_{j,0} + a_{j,0}b_{i,1} + a_{j,1}b_{i,0} \in [0, 4p^2]$ since $s_x = a_ib_j + a_jb_i$. Hence, for $0 \leq x \leq 2$, double-precision subtractions for computing $s_{x,1}$ using Karatsuba do not require carry checks (steps 4 and 6, 11 and 13, 19 and 21). For computing $s_{x,0} = r_{x,0} - (v_{i,0} + v_{j,0})$, addition does not require carry check (output range $[0, 2(2^N \cdot p/4 + p^2)] \subset [0, 2^N \cdot p]$) and subtraction gives result in the range $[0, 2^N \cdot p]$ when using **Option 2** (steps 5, 12 and 20). For computing c_0, multiplication by ξ, i.e., $S_0 = \xi s_0$ involves the operations $S_{0,0} = s_{0,0} - s_{0,1}$ and $S_{0,1} = s_{0,0} + s_{0,1}$, which are computed in double-precision using **Option 2** to get the output range $[0, 2^N \cdot p]$ (step 7). Similarly, final additions with v_0 require **Option 2** to get again the output range $[0, 2^N \cdot p]$ (step 8). For computing c_1, $S_1 = \xi v_2$ is computed as $S_{1,0} = v_{2,0} - v_{2,1}$ and $S_{1,1} = v_{2,0} + v_{2,1}$, where the former requires a double-precision subtraction using **Option 1** ($h = 1$) to get a result in the range $[0, 2^N \cdot p/2 + 2^N \cdot p/4 + p^2] \subset [0, 2^N \cdot p]$ (step 14) and the latter requires a double-precision addition with no carry check to get a result in the range $[0, (2^N \cdot p/4) + 3p^2] \subset [0, 2^N \cdot p]$ (step 15). Then, $c_{1,0} = s_{1,0} + S_{1,0}$ and $c_{1,1} = s_{1,1} + S_{1,1}$ involve double-precision additions using **Option 2** to obtain results in the range $[0, 2^N \cdot p]$ (step 16). Results $c_{2,0} = s_{2,0} + v_{1,0}$ and $c_{2,1} = s_{2,1} + v_{1,1}$ require a double-precision addition using **Option 2** (final output range $[0, 2^N \cdot p]$, step 22) and a double-precision addition without carry check (final output range $[0, 6p^2] \subset [0, 2^N \cdot p]$, step 23), respectively. Modular reductions have been delayed again to the last layer $\mathbb{F}_{p^{12}}$.

Finally, let $a, b, c \in \mathbb{F}_{p^{12}}$ such that $a = a_0 + a_1w, b = b_0 + b_1w, c = a \cdot b = c_0 + c_1w$.

Algorithm 6.3 Multiplication in $\mathbb{F}_{p^6}$ without reduction ($\times^6$, cost of $6\tilde{m}_u + 28\tilde{a}$)

```
Input: $a = (a_0 + a_1 v + a_2 v^2)$ and $b = (b_0 + b_1 v + b_2 v^2) \in \mathbb{F}_{p^6}$
Output: $c = a \cdot b = (c_0 + c_1 v + c_2 v^2) \in \mathbb{F}_{p^6}$
 1: $T_0 \leftarrow a_0 \times^2 b_0$, $T_1 \leftarrow a_1 \times^2 b_1$, $T_2 \leftarrow a_2 \times^2 b_2$        (Option 1, $h = 2$)
 2: $t_0 \leftarrow a_1 +^2 a_2$, $t_1 \leftarrow b_1 +^2 b_2$
 3: $T_3 \leftarrow t_0 \times^2 t_1$        (Option 2)
 4: $T_4 \leftarrow T_1 +^2 T_2$
 5: $T_{3,0} \leftarrow T_{3,0} \ominus T_{4,0}$        (Option 2)
 6: $T_{3,1} \leftarrow T_{3,1} - T_{4,1}$
 7: $T_{4,0} \leftarrow T_{3,0} \ominus T_{3,1}$, $T_{4,1} \leftarrow T_{3,0} \oplus T_{3,1}$ ($\equiv T_4 \leftarrow \xi \cdot T_3$)        (Option 2)
 8: $T_5 \leftarrow T_4 \oplus^2 T_0$        (Option 2)
 9: $t_0 \leftarrow a_0 +^2 a_1$, $t_1 \leftarrow b_0 +^2 b_1$
10: $T_3 \leftarrow t_0 \times^2 t_1$        (Option 2)
11: $T_4 \leftarrow T_0 +^2 T_1$
12: $T_{3,0} \leftarrow T_{3,0} \ominus T_{4,0}$        (Option 2)
13: $T_{3,1} \leftarrow T_{3,1} - T_{4,1}$
14: $T_{4,0} \leftarrow T_{2,0} \ominus T_{2,1}$        (Option 1, $h = 1$)
15: $T_{4,1} \leftarrow T_{2,0} + T_{2,1}$ (steps 14-15 $\equiv T_4 \leftarrow \xi \cdot T_2$)
16: $T_6 \leftarrow T_3 \oplus^2 T_4$        (Option 2)
17: $t_0 \leftarrow a_0 +^2 a_2$, $t_1 \leftarrow b_0 +^2 b_2$
18: $T_3 \leftarrow t_0 \times^2 t_1$        (Option 2)
19: $T_4 \leftarrow T_0 +^2 T_2$
20: $T_{3,0} \leftarrow T_{3,0} \ominus T_{4,0}$        (Option 2)
21: $T_{3,1} \leftarrow T_{3,1} - T_{4,1}$
22: $T_{7,0} \leftarrow T_{3,0} \oplus T_{1,0}$        (Option 2)
23: $T_{7,1} \leftarrow T_{3,1} + T_{1,1}$
24: return $c = (T_5 + T_6 v + T_7 v^2)$
```

Algorithm 6.4 details the required operations for computing multiplication. In this case, $c_1 = (a_0 + a_1)(b_0 + b_1) - a_1 b_1 - a_0 b_0$. At step 1, $\mathbb{F}_{p^6}$ multiplications $a_0 b_0$ and $a_1 b_1$ give outputs in range $\subset [0, 2^N \cdot p]$ using Algorithm 6.3. Additions $a_0 + a_1$ and $b_0 + b_1$ are single-precision reduced modulo p so that multiplication $(a_0 + a_1)(b_0 + b_1)$ in step 2 gives output in range $\subset [0, 2^N \cdot p]$ using Algorithm 6.3. Then, subtractions by $a_1 b_1$ and $a_0 b_0$ use double-precision operations with **Option 2** to have an output range $[0, 2^N \cdot p]$ so that we can apply Montgomery reduction at step 5 to obtain the result modulo p. For $c_0 = a_0 b_0 + v a_1 b_1$, multiplication by v, i.e., $T = v \cdot v_1$, where $v_i = a_i b_i$, involves the double-precision operations $T_{0,0} = v_{2,0} - v_{2,1}, T_{0,1} = v_{2,0} + v_{2,1}, T_1 = v_0$ and $T_2 = v_1$, all performed with **Option 2** to obtain the output range $[0, 2^N \cdot p]$ (steps 6-7). Final addition with $a_0 b_0$ uses double-precision with **Option 2** again so that we can apply Montgomery reduction at step 9 to obtain the result modulo p. We remark that, by applying the lazy reduction technique using the operation sequence above, we have reduced the number of reductions in $\mathbb{F}_{p^6}$ from 3 to only 2, or the number of total modular reductions in $\mathbb{F}_p$ from 54 (or 36 if lazy reduction is employed in $\mathbb{F}_{p^2}$) to only $k = 12$.

Algorithm 6.4 Multiplication in $\mathbb{F}_{p^{12}}$ ($\times^{12}$, cost of $18\tilde{m}_u + 6\tilde{r} + 110\tilde{a}$)

```
Input: $a = (a_0 + a_1 w)$ and $b = (b_0 + b_1 w) \in \mathbb{F}_{p^{12}}$
Output: $c = a \cdot b = (c_0 + c_1 w) \in \mathbb{F}_{p^{12}}$
 1: $T_0 \leftarrow a_0 \times^6 b_0$, $T_1 \leftarrow a_1 \times^6 b_1$, $t_0 \leftarrow a_0 \oplus^6 a_1$, $t_1 \leftarrow b_0 \oplus^6 b_1$
 2: $T_2 \leftarrow t_0 \times^6 t_1$
 3: $T_3 \leftarrow T_0 \oplus^6 T_1$                                        (Option 2)
 4: $T_2 \leftarrow T_2 \ominus^6 T_3$                                        (Option 2)
 5: $c_1 \leftarrow T_2 \bmod^6 p$
 6: $T_{2,0,0} \leftarrow T_{1,2,0} \ominus T_{1,2,1}$, $T_{2,0,1} \leftarrow T_{1,2,0} \oplus T_{1,2,1}$          (Option 2)
 7: $T_{2,1} \leftarrow T_{1,0}$, $T_{2,2} \leftarrow T_{1,1}$ (steps 6-7 $\equiv T_2 \leftarrow v \cdot T_1$)
 8: $T_2 \leftarrow T_0 \oplus^6 T_2$                                        (Option 2)
 9: $c_0 \leftarrow T_2 \bmod^6 p$
10: return $c = (c_0 + c_1 w)$
```

As previously stated, there are situations when it is more efficient to perform reductions right after multiplications and squarings in the last arithmetic layer of the tower construction. We illustrate the latter with squaring in $\mathbb{F}_{p^{12}}$. As shown in Algorithm 6.5, a total of 2 reductions in $\mathbb{F}_{p^6}$ are required when performing $\mathbb{F}_{p^6}$ multiplications in step 4. If lazy reduction was applied, the number of reductions would stay at 2, and worse, the total cost would be increased because some operations would require double-precision. The reader should note that the approach suggested by [97], where the formulas in [170] are employed for computing squarings in internal cubic extensions of $\mathbb{F}_{p^{12}}$, saves $1\tilde{m}$ in comparison with Algorithm 6.5. However, we experimented such approach with several combinations of formulas and towering, and it remained consistently slower than Algorithm 6.5 due to an increase in the number of additions.

6.4 Miller Loop

In this section, we present our optimizations to the curve arithmetic. To be consistent with other results in the literature, we do not distinguish between simple- and double-precision additions in the formulas below.

Recently, Costello *et al.* [26, Section 5] proposed the use of homogeneous coordinates to perform the curve arithmetic entirely on the twist. Their formula for computing a point doubling and line evaluation costs $2\tilde{m} + 7\tilde{s} + 23\tilde{a} + 4m + 1m_{b'}$. The twisting of point P, given in our case by $(x_P/w^2, y_P/w^3) = (\frac{x_P}{\xi}v^2, \frac{y_P}{\xi}vw)$, is eliminated by multiplying the

Algorithm 6.5 Squaring in $\mathbb{F}_{p^{12}}$ (cost of $12\tilde{m}_u + 6\tilde{r} + 73\tilde{a}$)

```
Input: $a = (a_0 + a_1 w) \in \mathbb{F}_{p^{12}}$
Output: $c = a^2 = (c_0 + c_1 w) \in \mathbb{F}_{p^{12}}$
1: $t_0 \leftarrow a_0 \oplus^6 a_1, t_{1,0,0} \leftarrow a_{1,2,0} \ominus a_{1,2,1},\ t_{1,0,1} \leftarrow a_{1,2,0} \oplus a_{1,2,1}$
2: $t_{1,1} \leftarrow a_{1,0},\ t_{1,2} \leftarrow a_{1,1}$ (steps 2-3 $\equiv t_1 \leftarrow v \cdot a_1$)
3: $t_1 \leftarrow a_0 \oplus^6 t_1$
4: $c_1 \leftarrow (a_0 \times^6 a_1) \bmod^6 p,\ t_0 \leftarrow (t_0 \times^6 t_1) \bmod^6 p$
5: $t_{1,0,0} \leftarrow c_{1,2,0} \ominus c_{1,2,1},\ t_{1,0,1} \leftarrow c_{1,2,0} \oplus c_{1,2,1}$
6: $t_{1,1} \leftarrow c_{1,0},\ t_{1,2} \leftarrow c_{1,1}$ (steps 6-7 $\equiv t_1 \leftarrow v \cdot c_1$)
7: $t_1 \leftarrow t_1 \oplus^6 c_1$
8: $c_0 \leftarrow t_0 \ominus^6 t_1,\ c_1 \leftarrow c_1 \oplus^6 c_1$
9: return $c = (c_0 + c_1 w)$
```

whole line evaluation by ξ and relying on the final exponentiation to eliminate this extra factor [26]. Clearly, the main drawback of this formula is the high number of additions. We present the following revised formula:

$$X_3 = \frac{X_1 Y_1}{2}\left(Y_1^2 - 9b'Z_1^2\right), \quad Y_3 = \left[\frac{1}{2}\left(Y_1^2 + 9b'Z_1^2\right)\right]^2 - 27b'^2 Z_1^4, \quad Z_3 = 2Y_1^3 Z_1,$$
$$l = (-2Y_1 Z_1 y_P)vw + \left(3X_1^2 x_P\right)v^2 + \xi\left(3b'Z_1^2 - Y_1^2\right). \tag{6.2}$$

This doubling formula gives the cost of $3\tilde{m} + 6\tilde{s} + 17\tilde{a} + 4m + m_{b'} + m_{\xi}$. Moreover, if the parameter b' is cleverly selected as in [97], multiplication by b' can be performed with minimal number of additions and subtractions. For instance, if one fixes $b = 2$ then $b' = 2/(1+i) = 1-i$. Accordingly, the following execution has a cost of $3\tilde{m}+6\tilde{s}+19\tilde{a}+4m$ (note that computations for E and $l_{0,0}$ are over $\mathbb{F}_p$ and $\overline{y_P} = -y_P$ is precomputed):

$$\begin{gathered} A = X_1 \cdot Y_1/2,\ B = Y_1^2,\ C = Z_1^2,\ D = 3C,\ E_0 = D_0 + D_1, \\ E_1 = D_1 - D_0,\ F = 3E,\ X_3 = A \cdot (B - F),\ G = (B+F)/2, \\ Y_3 = G^2 - 3E^2,\ H = (Y_1 + Z_1)^2 - (B + C), \\ Z_3 = B \cdot H,\ I = E - B,\ J = X_1^2 \\ l_{0,0,0} = I_0 - I_1,\ l_{0,0,1} = I_0 + I_1,\ l_{1,1} = H \cdot \overline{y_P},\ l_{0,2} = 3J \cdot x_P. \end{gathered} \tag{6.3}$$

We point out that in practice we have observed that $\tilde{m} - \tilde{s} \approx 3\tilde{a}$. Hence, it is more efficient to compute $X_1 Y_1$ directly than using $(X_1 + Y_1)^2, B$ and J. If this was not the case, the formula could be computed with cost $2\tilde{m} + 7\tilde{s} + 23\tilde{a} + 4m$.

Remarkably, the technique proposed in Section 6.3 for delaying reductions can also be applied to the point arithmetic over a quadratic extension field. Reductions can be delayed to the end of each $\mathbb{F}_{p^2}$ multiplication/squaring and then delayed further for those sums of products that allow reducing the number of reductions. Although not plentiful (given the nature of most curve arithmetic formulas which have consecutive and redundant multiplications/squarings), there are a few places where this technique can be applied. For instance, doubling formula (6.2) requires 25 $\mathbb{F}_p$ reductions (3 per $\mathbb{F}_{p^2}$ multiplication using Karatsuba, 2 per $\mathbb{F}_{p^2}$ squaring and 1 per $\mathbb{F}_p$ multiplication). First, by delaying reductions inside $\mathbb{F}_{p^2}$ arithmetic the number of reductions per multiplication goes down to only 2, with 22 reductions in total. Moreover, reductions corresponding to G^2 and $3E^2$ in Y_3 (see execution (6.3)) can be further delayed and merged, eliminating the need of two reductions. In total, the number of reductions is now 20. Similar optimizations can be applied to other point/line evaluation formulas (see extended version [171] for optimizations to formulas using Jacobian and homogeneous coordinates).

For accumulating line evaluations into the Miller variable, $\mathbb{F}_{p^{12}}$ is represented using the towering $\mathbb{F}_{p^2} \to \mathbb{F}_{p^4} \to \mathbb{F}_{p^{12}}$ and a special (dense×sparse)-multiplication costing $13\tilde{m}_u + 6\tilde{r} + 61\tilde{a}$ is used. During the first iteration of the loop, a squaring in $\mathbb{F}_{p^{12}}$ can be eliminated since the Miller variable is initialized as 1 (line 1 in Algorithm 6.1) and a special (sparse×sparse) multiplication costing $7\tilde{m}_u + 5\tilde{r} + 30\tilde{a}$ is used to multiply the first two line evaluations, resulting in the revised Algorithm 6.6. This sparser multiplication is also used for multiplying the two final line evaluations in step 10 of the algorithm.

6.5 Final Exponentiation

The fastest way known for computing the final exponentiation is described in [172]. The power $\frac{p^{12}-1}{n}$ is factored into an easy exponent $(p^6 - 1)$ which requires a conjugation and an inversion; another easy exponent $(p^2 + 1)$ which requires a p^2-power Frobenius and a multiplication; and a hard exponent $(p^4 - p^2 + 1)/n$ which can be performed in the cyclotomic subgroup $\mathbb{G}_{\phi_6}(\mathbb{F}_{p^2})$. For computing this last power, one can write the hard exponent as follows [19]:

$$(p^4 - p^2 + 1)/n = \lambda_3 p^3 + \lambda_2 p^2 + \lambda_1 p + \lambda_0,$$

where

$$\begin{aligned} \lambda_3(u) = 1 \quad &, \quad \lambda_2(u) = 6u^2 + 1, \\ \lambda_1(u) = -36u^3 - 18u^2 - 12u + 1 \quad &, \quad \lambda_0(u) = -36u^3 - 30u^2 - 18u - 2, \end{aligned}$$

and compute the individual powers by a multi-addition chain, requiring three consecutive exponentiations by the absolute value of the curve parameter $|u|$, 13 multiplications, 4 squarings, 4 p-power Frobenius, 2 p^2-power Frobenius and a single p^3-power Frobenius in $\mathbb{F}_{p^{12}}$. These powers of Frobenius can be efficiently computed with the formulas in [99].

In the following subsections, we explain how to remove the expensive inversion in $\mathbb{F}_{p^{12}}$ mentioned at the end of Section 6.2; and how the cyclotomic subgroup structure allows faster compressed squarings and consequently faster exponentiation by $|u|$.

6.5.1 Removing the Inversion Penalty

From Algorithm 6.1, the Optimal Ate pairing when $u < 0$ can be computed as

$$a_{opt}(Q,P) = \left[g^{-1} \cdot h\right]^{\frac{p^{12}-1}{n}}, \tag{6.4}$$

with $r = 6u+2$, $g = f_{|r|,Q}(P)$ and $h = l_{[-|r|]Q,\pi_p(Q)}(P) \cdot l_{[-|r|Q]+\pi_p(Q),-\pi_p^2(Q)}(P)$. Lemma 6.5.1 below allows one to replace the expensive inversion g^{-1} with a simple conjugation with no change in the result. This is depicted in line 9 of Algorithm 6.6.

Lemma 6.5.1. *The pairing $a_{opt}(Q,P)$ can be computed as $\left[g^{p^6} \cdot h\right]^{\frac{p^{12}-1}{n}}$, with g, h defined as above.*

Proof. By distributing the power $(p^{12}-1)/n$ in terms g, h in Equation (6.4):

$$\begin{aligned} a_{opt}(Q,P) &= g^{\frac{1-p^{12}}{n}} \cdot h^{\frac{p^{12}-1}{n}} = g^{\frac{(1-p^6)(1+p^6)}{n}} \cdot h^{\frac{p^{12}-1}{n}} \\ &= g^{\frac{(p^{12}-p^6)(1+p^6)}{n}} \cdot h^{\frac{p^{12}-1}{n}} = g^{\frac{p^6(p^6-1)(p^6+1)}{n}} \cdot h^{\frac{p^{12}-1}{n}} = \left[g^{p^6} \cdot h\right]^{\frac{p^{12}-1}{n}} \end{aligned}$$

□

6.5.2 Computing u-th powers in $\mathbb{G}_{\phi_6}(\mathbb{F}_{p^2})$

Let

$$g = \sum_{i=0}^{2} (g_{2i} + g_{2i+1}s)t^i \in \mathbb{G}_{\phi_6}(\mathbb{F}_{p^2}) \text{ and } g^2 = \sum_{i=0}^{2} (h_{2i} + h_{2i+1}s)t^i$$

with $g_i, h_i \in \mathbb{F}_{p^2}$. In [173], it was shown that one can compress g to $\mathcal{C}(g) = [g_2, g_3, g_4, g_5]$, and the compressed representation of g^2 is computed as $\mathcal{C}(g^2) = [h_2, h_3, h_4, h_5]$, where h_i is computed as follows:

$$\begin{aligned} h_2 &= 2(g_2 + 3\xi B_{4,5}), & h_3 &= 3(A_{4,5} - (\xi+1)B_{4,5}) - 2g_3, \\ h_4 &= 3(A_{2,3} - (\xi+1)B_{2,3}) - 2g_4, & h_5 &= 2(g_5 + 3B_{2,3}), \end{aligned} \tag{6.5}$$

where $A_{i,j} = (g_i + g_j)(g_i + \xi g_j)$ and $B_{i,j} = g_i g_j$. The above formula requires 4 multiplications in $\mathbb{F}_{p^2}$. Considering the lazy reduction technique discussed in Section 6.3.3, we propose another formula that is slightly faster and has a cost of $6\tilde{s}_u + 4\tilde{r} + 31\tilde{a}$. The formula is given as follows:

$$\begin{array}{ll} h_2 = 2g_2 + 3(S_{4,5} - S_4 - S_5)\xi, & h_3 = 3(S_4 + S_5\xi) - 2g_3, \\ h_4 = 3(S_2 + S_3\xi) - 2g_4, & h_5 = 2g_5 + 3(S_{2,3} - S_2 - S_3), \end{array} \tag{6.6}$$

where $S_{i,j} = (g_i + g_j)^2$ and $S_i = g_i^2$; also see extended version [171] for the correctness of our formula and an explicit implementation.

When g is raised to a power via a square-and-multiply exponentiation algorithm, full representation of elements (decompression) is required because, if $\mathcal{C}$ is used as the compression map, it is not known how to perform multiplication given the compressed representation of elements. Given a compressed representation of $g \in \mathbb{G}_{\phi_6}(\mathbb{F}_{p^2}) \setminus \{1\}$, $\mathcal{C}(g) = [g_2, g_3, g_4, g_5]$, the decompression map $\mathcal{D}$ is evaluated as follows (see [173] for more details):

$$\mathcal{D}([g_2, g_3, g_4, g_5]) = (g_0 + g_1 s) + (g_2 + g_3 s)t + (g_4 + g_5 s)t^2,$$

$$\begin{cases} g_1 = \frac{g_5^2\xi + 3g_4^2 - 2g_3}{4g_2}, \quad g_0 = (2g_1^2 + g_2 g_5 - 3g_3 g_4)\xi + 1, & \text{if } g_2 \neq 0; \\ g_1 = \frac{2g_4 g_5}{g_3}, \quad g_0 = (2g_1^2 - 3g_3 g_4)\xi + 1, & \text{if } g_2 = 0. \end{cases}$$

In particular, $g^{|u|}$ can be computed in three steps:

1. Compute $\mathcal{C}(g^{2^i})$ for $1 \le i \le 62$ using (6.6) and store $\mathcal{C}(g^{2^{55}})$ and $\mathcal{C}(g^{2^{62}})$.
2. Compute $\mathcal{D}(\mathcal{C}(g^{2^{55}})) = g^{2^{55}}$ and $\mathcal{D}(\mathcal{C}(g^{2^{62}})) = g^{2^{62}}$.
3. Compute $g^{|u|} = g^{2^{62}} \cdot g^{2^{55}} \cdot g$.

Step 1 requires 62 squarings in $\mathbb{G}_{\phi_6}(\mathbb{F}_{p^2})$. Using Montgomery's simultaneous inversion trick [174], Step 2 requires $9\tilde{m} + 6\tilde{s} + 22\tilde{a} + \tilde{i}$. Step 3 requires 2 multiplications in $\mathbb{F}_{p^{12}}$. The total cost is:

$$\begin{aligned} Exp &= 62 \cdot (6\tilde{s}_u + 4\tilde{r} + 31\tilde{a}) + (9\tilde{m} + 6\tilde{s} + 22\tilde{a} + \tilde{i}) + 2 \cdot (18\tilde{m}_u + 6\tilde{r} + 110\tilde{a}) \\ &= 45\tilde{m}_u + 378\tilde{s}_u + 275\tilde{r} + 2164\tilde{a} + \tilde{i}, \end{aligned}$$

Granger-Scott's [175] formula for squaring can be implemented at a cost of $9\tilde{s}_u + 6\tilde{r} + 46\tilde{a}$ if lazy reduction techniques are employed. With this approach, an exponentiation costs:

$$\begin{aligned} Exp' &= 62 \cdot (9\tilde{s}_u + 6\tilde{r} + 46\tilde{a}) + 2 \cdot (18\tilde{m}_u + 6\tilde{r} + 110\tilde{a}) \\ &= 36\tilde{m}_u + 558\tilde{s}_u + 399\tilde{r} + 3072\tilde{a}. \end{aligned}$$

Hence, the faster compressed squaring formulas reduce by 33% the number of squarings and by 30% the number of additions in $\mathbb{F}_{p^2}$.

Algorithm 6.6 Revised Optimal Ate pairing on BN curves (generalized for $u < 0$).

```
Input: $P \in \mathbb{G}_1, Q \in \mathbb{G}_2, r = |6u+2| = \sum_{i=0}^{\log_2(r)} r_i 2^i$
Output: $a_{opt}(Q, P)$
 1: $d \leftarrow l_{Q,Q}(P)$, $T \leftarrow 2Q, e \leftarrow 1$
 2: if $r_{\lfloor \log_2(r) \rfloor - 1} = 1$ then $e \leftarrow l_{T,Q}(P), T \leftarrow T + Q$
 3: $f \leftarrow d \cdot e$
 4: for $i = \lfloor \log_2(r) \rfloor - 2$ downto $0$ do
 5:     $f \leftarrow f^2 \cdot l_{T,T}(P), T \leftarrow 2T$
 6:     if $r_i = 1$ then $f \leftarrow f \cdot l_{T,Q}(P), T \leftarrow T + Q$
 7: end for
 8: $Q_1 \leftarrow \pi_p(Q), Q_2 \leftarrow \pi_p^2(Q)$
 9: if $u < 0$ then $T \leftarrow -T, f \leftarrow f^{p^6}$
10: $d \leftarrow l_{T,Q_1}(P), T \leftarrow T + Q_1, e \leftarrow l_{T,-Q_2}(P), T \leftarrow T - Q_2$, $f \leftarrow f \cdot (d \cdot e)$
11: $f \leftarrow f^{(p^6-1)(p^2+1)(p^4-p^2+1)/n}$
12: return $f$
```

6.6 Computational Cost

We now consider all the improvements described in the previous sections and present a detailed operation count. Table 6.1 shows the exact operation count for each operation executed in Miller's Algorithm.

Table 6.1: Operation counts for arithmetic required by Miller's Algorithm. (†) Work [99] counts these additions in a different way. Considering their criteria, costs for multiplication and squaring in $\mathbb{F}_{p^2}$ are $3m_u + 2r + 4a$ and $2m_u + 2r + 2a$, respectively.

$E'(\mathbb{F}_{p^2})$-Arithmetic	**Operation Count**
Doubling/Eval.	$3\tilde{m}_u + 6\tilde{s}_u + 8\tilde{r} + 22\tilde{a} + 4m$
Addition/Eval.	$11\tilde{m}_u + 2\tilde{s}_u + 11\tilde{r} + 12\tilde{a} + 4m$
p-power Frobenius	$6m_u + 4r + 18a$
p^2-power Frobenius	$2m + 2a$
Negation	$\tilde{a}$
$\mathbb{F}_{p^2}$-Arithmetic	**Operation Count**
Add./Sub./Neg.	$\tilde{a} = 2a$
Conjugation	a
Multiplication	$\tilde{m} = \tilde{m}_u + \tilde{r} = 3m_u + 2r + 8a^{\dagger}$
Squaring	$\tilde{s} = \tilde{s}_u + \tilde{r} = 2m_u + 2r + 3a^{\dagger}$
Multiplication by β	a
Multiplication by ξ	$2a$
Inversion	$\tilde{i}$

$\mathbb{F}_{p^{12}}$-Arithmetic	**Operation Count**
Add./Sub.	$6\tilde{a}$
Conjugation	$3\tilde{a}$
Multiplication	$18\tilde{m}_u + 6\tilde{r} + 110\tilde{a}$
Sparse Mult.	$13\tilde{m}_u + 6\tilde{r} + 61\tilde{a}$
Sparser Mult.	$7\tilde{m}_u + 5\tilde{r} + 30\tilde{a}$
Squaring	$12\tilde{m}_u + 6\tilde{r} + 73\tilde{a}$
Cyc. Squaring	$9\tilde{s}_u + 6\tilde{r} + 46\tilde{a}$
Comp. Squaring	$6\tilde{s}_u + 4\tilde{r} + 31\tilde{a}$
Simult. Decomp.	$9\tilde{m} + 6\tilde{s} + 22\tilde{a} + \tilde{i}$
p-power Frobenius	$15m_u + 10r + 46a$
p^2-power Frobenius	$10m + 2\tilde{a}$
Inversion	$25\tilde{m}_u + 9\tilde{s}_u + 24\tilde{r}$ $+112\tilde{a} + \tilde{i}$

For the selected parameters and with the presented improvements, the Miller Loop in Algorithm 6.6 executes 64 point doublings with line evaluations, 6 point additions with line evaluations (4 inside Miller Loop and 2 more at the final steps), 1 negation in $\mathbb{F}_{p^2}$ to precompute $\overline{y_P}$, 1 p-power Frobenius, 1 p^2-power Frobenius and 2 negations in $E(\mathbb{F}_{p^2})$; and 1 conjugation, 1 multiplication, 66 sparse multiplications, 2 sparser multiplications and 63 squarings in $\mathbb{F}_{p^{12}}$. The cost of the Miller Loop is:

$$\begin{aligned} ML &= 64 \cdot (3\tilde{m}_u + 6\tilde{s}_u + 8\tilde{r} + 22\tilde{a} + 4m) + 6 \cdot (11\tilde{m}_u + 2\tilde{s}_u + 11\tilde{r} + 12\tilde{a} + 4m) \\ &+ \tilde{a} + 6m_u + 4r + 18a + 2m + 2a + 2\tilde{a} + 3\tilde{a} + (18\tilde{m}_u + 6\tilde{r} + 110\tilde{a}) \\ &+ 66 \cdot (13\tilde{m}_u + 6\tilde{r} + 61\tilde{a}) + 2 \cdot (7\tilde{m}_u + 5\tilde{r} + 30\tilde{a}) + 63 \cdot (12\tilde{m}_u + 6\tilde{r} + 73\tilde{a}) \\ &= 1904\tilde{m}_u + 396\tilde{s}_u + 1368\tilde{r} + 10281\tilde{a} + 282m + 6m_u + 4r + 20a. \end{aligned}$$

The final exponentiation executes in total 1 inversion, 4 conjugations, 15 multiplications, 3 u-th powers, 4 cyclotomic squarings, 5 p-power Frobenius, 3 p^2-power Frobenius:

$$\begin{aligned} FE &= 25\tilde{m}_u + 9\tilde{s}_u + 24\tilde{r} + 112\tilde{a} + \tilde{i} + 4 \cdot 3\tilde{a} + 15 \cdot (18\tilde{m}_u + 6\tilde{r} + 110\tilde{a}) \\ &+ 3 \cdot Exp + 4 \cdot (9\tilde{s}_u + 6\tilde{r} + 46\tilde{a}) + 5 \cdot (15m_u + 10r + 46a) + 3 \cdot (10m + 2\tilde{a}) \\ &= 430\tilde{m}_u + 1179\tilde{s}_u + 963\tilde{r} + 8456\tilde{a} + 4\tilde{i} + 30m + 75m_u + 50r + 230a. \end{aligned}$$

Table 6.2 gives a first-order comparison between our implementation and the best implementation available in the literature of the Optimal Ate pairing at the 128-bit security level in the same platform. For the related work, we suppose that lazy reduction is always used in $\mathbb{F}_{p^2}$ and then each multiplication or squaring essentially computes a modular reduction (that is, $\tilde{m} = \tilde{m}_u + \tilde{r} = 3m_u + 2r$ and $\tilde{s} = \tilde{s}_u + \tilde{r} = 2m_u + 2r$). Note that our generalization of the lazy reduction techniques to the whole pairing computation brings the number of modular reductions from the expected 7818 (if lazy reduction was only used for $\mathbb{F}_{p^2}$ arithmetic) to just 4662, avoiding more than 40% of the total required modular reductions. The number of multiplications is also reduced by 13% and the number of additions is increased by 26% due to lazy reduction trade-offs. Our operation count for the pairing computation is apparently more expensive than Pereira *et al.* [97]. However, the reader should note that, when we consider the real cost of additions in $\mathbb{F}_p$, we cannot exploit the squaring formula in $\mathbb{F}_{p^{12}}$ by [170] (see Section 6.3.3) and a point doubling formula with fewer multiplications (see Section 6.4), given the significant increase in the number of additions.

6.7 Implementation Results

A software implementation was realized to confirm the performance benefits resulting from the introduced techniques. We implemented $\mathbb{F}_{p^2}$ arithmetic directly in Assembly, largely following advice from [99] to optimize carry handling and eliminate function call

Table 6.2: Comparison of operation counts for different implementations of the Optimal Ate pairing at the 128-bit security level.

Work	Phase	**Operations in $\mathbb{F}_{p^2}$**	**Operations in $\mathbb{F}_p$**
Beuchat *et al.*[99]	ML	$1952(\tilde{m}_u + \tilde{r}) + 568(\tilde{s}_u + \tilde{r}) + 6912\tilde{a}$	$6992m_u + 5040r$
	FE	$403(\tilde{m}_u + \tilde{r}) + 1719(\tilde{s}_u + \tilde{r}) + 7021\tilde{a}$	$4647m_u + 4244r$
	ML+FE	$2355(\tilde{m}_u + \tilde{r}) + 2287(\tilde{s}_u + \tilde{r}) + 13933\tilde{a}$	$11639m_u + 9284r$
This work	ML	$1904\tilde{m}_u + 396\tilde{s}_u + 1368\tilde{r} + 10281\tilde{a}$	$6504m_u + 2736r$
	FE	$430\tilde{m}_u + 1179\tilde{s}_u + 963\tilde{r} + 8456\tilde{a}$	$3648m_u + 1926r$
	ML+FE	$2334\tilde{m}_u + 1575\tilde{s}_u + 2331\tilde{r} + 18737\tilde{a}$	$10152m_u + 4662r$

Table 6.3: Cumulative performance improvement when using new arithmetic in cyclotomic subgroups (Section 6.5.2) and generalized lazy reduction (Section 6.3.1) on several Intel and AMD 64-bit architectures. Improvements are calculated relatively to the Basic Implementation. Timings are presented in millions of clock cycles.

	This work							
Method	Phenom II	Impr.	Core i5	Impr.	Opteron	Impr.	Core 2	Impr.
Basic Implementation	1.907	-	2.162	-	2.127	-	2.829	-
Cyclotomic Formulas	1.777	7%	2.020	7%	2.005	6%	2.677	5%
Lazy Reduction	1.562	18%	1.688	22%	1.710	20%	2.194	22%

overheads. Higher-level algorithms were implemented using the C programming language compiled with the GCC compiler using `-O3` optimization level. Table 6.3 presents the relevant timings in millions of cycles. Basic Implementation employs homogeneous projective coordinates and lazy reduction below $\mathbb{F}_{p^2}$. Faster arithmetic in cyclotomic subgroups accelerates the Basic Implementation by 5%-7% and, in conjunction with generalized lazy reduction, it improves the Basic Implementation by 18%-22%.

Table 6.4 compares our implementation with related work. To ensure that machines with different configurations but belonging to the same microarchitecture had compatible performance (as is the case with Core i5 and Core i7), software from [99] was benchmarked and the results compared with the ones reported in [99]. Machines considered equivalent by this criteria are presented in the same column. We note that Phenom II was not considered in the original study and that we could not find a Core 2 Duo machine producing the same timings as in [99]. For this reason, timings for these two architectures were taken independently by the authors using the available software. Observe that the Basic Implementation in Table 6.3 consistently outperforms Beuchat *et al.* due to our careful implementation of an optimal choice of parameters ($E(\mathbb{F}_p) : y^2 = x^3 + 2, p = 3 \bmod 4$) [97] combined with optimized curve arithmetic in homogeneous coordinates [26]. When lazy reduction and faster cyclotomic formulas are enabled, pairing computation becomes faster than the best previous result by 28%-34%. For extended benchmark results and comparisons with previous works on different 64-bit processors, the reader is referred to our online database [176].

Table 6.4: Comparison between implementations on 64-bit architectures. Timings are presented in clock cycles.

	Work/Platform			
	Beuchat *et al.* [99]			
Operation	Phenom II	Core i7	Opteron	Core 2 Duo
Multiplication in $\mathbb{F}_{p^2}$	440	435	443	590
Squaring in $\mathbb{F}_{p^2}$	353	342	355	479
Miller Loop	1,338,000	1,330,000	1,360,000	1,781,000
Final Exponentiation	1,020,000	1,000,000	1,040,000	1,370,000
Optimal Ate Pairing	2,358,000	2,330,000	2,400,000	3,151,000
	This work			
Operation	Phenom II	Core i5	Opteron	Core 2 Duo
Multiplication in $\mathbb{F}_{p^2}$	368	412	390	560
Squaring in $\mathbb{F}_{p^2}$	288	328	295	451
Miller Loop	898,000	978,000	988,000	1,275,000
Final Exponentiation	664,000	710,000	722,000	919,000
Optimal Ate Pairing	1,562,000	1,688,000	1,710,000	2,194,000
Improvement	34%	28%	29%	30%

6.8 Conclusion

In this work, we revisited the problem of computing optimal pairings on ordinary pairing-friendly curves over prime fields. Several new techniques were introduced for pairing computation, comprised mainly in the generalization of lazy reduction techniques to arithmetic in extensions above $\mathbb{F}_{p^2}$ and inside curve arithmetic; and improvements to the final exponentiation consisting of a formula for compressed squaring in cyclotomic subgroups and an arithmetic trick to remove penalties from negative curve parameterizations. The faster arithmetic in the cyclotomic subgroup improved pairing performance by 5%-7% and the generalized lazy reduction technique was able to eliminate 40% of the required modular reductions, improving pairing performance by further 11%-17%. The introduced techniques allow for the first time a pairing computation under 2 million cycles on 64-bit desktop computing platforms, improving the state-of-the-art by 28%-34%. The performance improvements are expected to be even higher on embedded architectures, where the ratio between multiplication and addition is typically higher.

Acknowledgements

We would like to express our gratitude to Alfred Menezes, Craig Costello, Michael Scott, Paulo S. L. M. Barreto, Geovandro C. C. F. Pereira and Conrado P. L. Gouvêa for useful discussions during the preparation of this work. The authors thank the Natural Sciences and Engineering Research Council of Canada (NSERC), the Ontario Centres of Excellence (OCE), CNPq, CAPES and FAPESP for partially supporting this work.

Chapter 7

Parallelizing the Weil and Tate Pairings

Diego F. Aranha, Francisco Henríquez-Rodríguez, Edward Knapp and Alfred Menezes

Abstract

In the past year, the speed record for pairing implementations on desktop-class machines has been broken several times. The speed records for asymmetric pairings were set on a single processor. In this paper, we describe our parallel implementation of the optimal ate pairing over Barreto-Naehrig (BN) curves that is about 1.23 times faster using two cores of an Intel Core i5 or Core i7 machine, and 1.45 times faster using 4 cores of the Core i7 than the state-of-the-art implementation on a single core. We instantiate Hess's general Weil pairing construction and introduce a new optimal Weil pairing tailored for parallel execution. Our experimental results suggest that the new Weil pairing is 1.25 times faster than the optimal ate pairing on 8-core extensions of the aforementioned machines. Finally, we combine previous techniques for parallelizing the eta pairing on a supersingular elliptic curve with embedding degree 4, and achieve an estimated 1.24-fold speedup on an 8-core extension of an Intel Core i7 over the previous best technique.

Publication

This work corresponds to the submited version accepted for publication in the Proceedings of the 13th Institute of Mathematics and its Applications' International Conference on Cryptography and Coding (IMA-CC 2011).

7.1 Introduction

Since the publication in 2001 of Boneh and Franklin's seminal paper on identity-based encryption [12], pairings have been used extensively to design ingenious protocols for meeting security objectives that are seemingly unachievable using conventional cryptographic techniques. Researchers have made remarkable progress in designing and implementing both symmetric and asymmetric pairings. For asymmetric pairings based on Barreto-Naehrig (BN) elliptic curves at the 128-bit security level, there have recently been several notable improvements on the 10-million cycle Core 2 implementation reported in [67]. Naehrig et al. [160] exploited high-speed vector floating-point operations and were able to compute a pairing in 4.47 million cycles on a Core 2. Shortly thereafter, Beuchat et al. [99] and Aranha et al. [20] reported timings of 2.33 million cycles and 1.70 million cycles, respectively, when employing the fastest integer multiplier available on the Intel Core i7 64-bit platform.

The aforementioned timings were all for a single-core implementation. In this paper, we continue the work initiated by Grabher, Großschädl and Page [96] on implementing pairings on multi-core platforms. This work is especially challenging for asymmetric pairings because of the apparent paucity of opportunities available for parallelizing Miller's basic algorithm. In particular, it seems hopeless to expect the optimal 2-fold speedup for known algorithms when going from 1 core to 2 cores on existing computing platforms. Furthermore, effective usage of parallel computation resources depends on expensive operating system calls for thread creation or synchronization and on the relative immaturity of development tools such as compilers, profilers, and debuggers. Concurrent programming is difficult in general, due to fundamentally nondeterministic nature of the multi-threading programming model [41], but it becomes even harder when the computational cost of what is being computed is not several orders of magnitude higher than the parallelization overhead itself.

Our results

We focus our attention on two of the fastest known symmetric and asymmetric pairings at the 128-bit security level, namely the eta pairing [1] over a supersingular elliptic curve with embedding degree 4, and Vercauteren's optimal ate pairing [24] over BN elliptic curves. It is worthwhile studying both symmetric and asymmetric pairings because they provide different functionalities (see [96]). Our target platforms are popular Intel architectures. For the eta pairing, we combine techniques from [98] and [102] and achieve an estimated 1.24-fold speedup on an 8-core extension of an Intel Core i7 over the previous best technique. For the optimal ate pairing, we exploit a method introduced in [102] for parallelizing a Miller function evaluation and a new 'delayed squaring' technique. Our implementation is about 1.23 times faster using two cores of an Intel Core i5 or Core i7 machine, and 1.45 times faster using 4 cores of the Core i7 than the state-of-the-art implementation on a single core [20]. We observe that the straightforward methods for parallelizing extension field multiplication that were deemed effective in [96] fail on the platforms under consideration because our field multiplication is much faster, whereby the cost of managing the resulting threads dominates the cost of useful computation.

The limited success in parallelizing the optimal ate pairing on BN curves is due in part to the apparent difficulty in parallelizing the final exponentiation. This motivated us to consider the Weil pairing, whose potential speed advantages over the Tate pairing due to the absence of a final exponentiation in the former were first considered in [177]. We study two optimal Weil pairings, both of which can be computed using the equivalent of four independent Miller functions each having optimal length, and without an expensive final exponentiation. The first pairing is an instantiation of Hess's general Weil pairing construction [178], while the second pairing is an elegant new construction tailored for parallel execution. These pairings are faster than previous variants of the Weil pairing proposed in [179] and [178]. Our experimental results suggest that the new Weil pairing is 1.25 times faster than the optimal ate pairing on 8-core extensions of the Intel Core i5 and Core i7 machines.

We emphasize that our implementations are for a *single* pairing evaluation on multiple cores. If a protocol requires multiple pairing evaluations, the best strategy may be to simply execute each pairing on a single core of a multi-core platform — the optimal strategy depends on several factors including the number of available cores. Thus, our work is primarily directed at protocols that require a single pairing evaluation in applications that have stringent response time requirements or where the processing power of individual cores in a multi-core platform is low. Some examples of protocols that require a single pairing evaluation are the encryption and decryption procedures in the Boneh-Franklin identity-based encryption scheme [12], signature verification in the Boneh-Boyen short signature scheme [180], the Sakai-Ohgishi-Kasahara non-interactive key agreement scheme [13], and Scott's identity-based key agreement scheme [181].

Other platforms

In order to exploit the fine-grained parallelism inherent in hardware platforms, a designer must carefully craft the circuit's control unit and schedule its hundreds of thousands of micro instructions [182], an extremely challenging and complex task. FPGA devices can achieve substantial performance improvements and power efficiency for cryptographic applications. However, their reconfigurable design feature results in unavoidable highly-redundant architectures that cause an overhead area factor between 20 and 40 when compared to static ASIC designs [183]. Mainly because of this, contemporary FPGA devices can run at maximum clock frequencies of less than 600MHz (although this value is rarely or never achieved in actual cryptographic designs). Thus, it is not entirely surprising that timings for the FPGA implementations of the optimal ate pairing over BN curves reported by Duquesne and Guillermin [184] and Yao et al. [185] are slightly slower than the ones for software implementation by Aranha et al. [20].

In contrast, ASIC designs and multi-core processors can easily operate at higher frequencies. Fan et al. [161] and Kammler et al. [186] presented ASIC and ASIP designs of pairings over BN curves at the 128-bit security level. The timings achieved by these designs are both slower than the ones reported in the aforementioned FPGA designs [184, 185] and software implementation [20]. On the other hand, modern multi-core processors are supported by standard C/C++ compilers such as GCC and Intel's ICC that can be combined with OpenMP to add parallelism. Even though achieving optimal performance using these tools is a challenging task, the software implementor's work is significantly easier than that of the hardware designer. In practice, the achievable parallelism on the multi-core processors tends to be coarse-grained, but this should be compared with the high frequencies of operation that these platforms enjoy, the periodic addition by major manufacturers of faster and more powerful sets of instructions, and the constant reduction of the retail price due to the large market for these processors.

We believe that deciding whether multi-core processors or FPGA/ASIC hardware devices are the best choice for parallel realizations of pairings is far from clear. Perhaps in the near future, the preferred design option will be a combination of both platforms as some hybrid computer manufacturers whose architectures combine both technologies seem to suggest. This combination of technologies can also be of interest for emerging resource-constrained and embedded multi-core architectures such as the dual-core Cortex ARM. It is conceivable that such constrained processors can be supported by fast hardware accelerators attached to them.

We expect that our study of parallel pairing implementation on desktop processors presented will be useful in predicting the performance that parallel pairing implementations can achieve in resource-constrained embedded systems. Even though the parallelization overhead is likely to be much more expensive in embedded systems than on desktop processors, the pairing computation time will be slower due to the usage of smaller processor word sizes and less sophisticated multipliers. Hence, we expect that the ratio between the pairing computation time and the parallelization overhead reported in our work will remain roughly the same as the ones that we can expect in resource-constrained platforms. Because of that, we anticipate that many of the observations and conclusions for pairing parallelization on desktop processors that we arrive at can be extrapolated to the embedded-system scenario.

Outline

The remainder of this paper is organized as follows. The optimal ate and eta pairings are reviewed in §7.2. Our parallelization of the optimal ate pairing is described in §7.3, and the optimal Weil pairings are presented in §7.4. Our parallel implementation of the optimal ate and optimal Weil pairings is described in §7.5. Our improvements to the parallel implementation of the eta pairing are presented in §7.6. We draw our conclusions in §7.7.

7.2 Background on pairings

Let E be an elliptic curve defined over the finite field $\mathbb{F}_q$, and let r be a prime with $r \mid \#E(\mathbb{F}_q)$ and $\gcd(r,q)=1$. The *embedding degree* k is the smallest positive integer with $r \mid (q^k-1)$. We will assume that k is even, whence $E[r] \subseteq E(\mathbb{F}_{q^k})$.

7.2.1 Miller functions

Let $R \in E(\mathbb{F}_{q^k})$ and let s be a non-negative integer. A *Miller function* $f_{s,R}$ [58] of *length* s is a function in $\mathbb{F}_{q^k}(E)$ with divisor $(f_{s,R}) = s(R) - (sR) - (s-1)(\infty)$. Note that $f_{s,R}$ is uniquely defined up to multiplication by nonzero constants in $\mathbb{F}_{q^k}$. The length s of a Miller function determines the number $\lfloor \log_2 s \rfloor$ of doubling steps, and the Hamming weight of s determines the number of addition steps in Miller's algorithm for computing $f_{s,R}$ [58]. We will always assume that Miller functions are minimally defined; that is, if $R \in E(\mathbb{F}_{q^e})$, then $f_{s,R}$ is selected from the function field $\mathbb{F}_{q^e}(E)$. Let u_∞ be an $\mathbb{F}_q$-rational uniformizing parameter for ∞. A function $f \in \mathbb{F}_{q^k}(E)$ is said to be *normalized* if $lc_\infty(f)=1$, where $lc_\infty(f) = (u_\infty^{-t} f)(\infty)$ and t is the order of f at ∞. Furthermore, f is said to be *semi-normalized* if $lc_\infty(f)$ belongs to a proper subfield of $\mathbb{F}_{q^k}$.

7.2.2 The Tate pairing

Let $\mathbb{G}_T$ denote the order-r subgroup of $\mathbb{F}_{q^k}^*$. The (reduced) Tate pairing $e_r : E[r] \times E[r] \to \mathbb{G}_T$ can be defined by

$$e_r : (P,Q) \mapsto \left(\frac{f_{r,P}(Q+R)}{f_{r,P}(R)} \right)^{(q^k-1)/r} \tag{7.1}$$

where $R \in E(\mathbb{F}_{q^k})$ satisfies $R \notin \{\infty, P, -Q, P-Q\}$. Several variants of the Tate pairing have been defined in the literature, e.g., [1, 2, 3, 24, 162, 178]. All these pairings have the property that they are fixed powers of the Tate pairing with domain restricted to the product of two order-r subgroups of $E[r]$. In §7.2.3 and §7.2.4, we describe two of the fastest asymmetric and symmetric pairings — the optimal ate pairing on BN curves, and the eta pairing on $k = 4$ supersingular curves.

7.2.3 The optimal ate pairing

A BN elliptic curve $E : Y^2 = X^3 + b$ [74] of order r is defined over a prime field $\mathbb{F}_p$, where $p(z) = 36z^4 + 36z^3 + 24z^2 + 6z + 1$ and where $r = \#E(\mathbb{F}_p) = 36z^4 + 36z^3 + 18z^2 + 6z + 1$ is prime. These curves have embedding degree $k = 12$. The integer z is called the BN parameter.

Let $\pi : (x, y) \mapsto (x^p, y^p)$ be the Frobenius endomorphism, and let $\mathbb{G}_1 = \{P \in E[r] : \pi(P) = P\} = E(\mathbb{F}_p)[r]$; $\mathbb{G}_1$ is the 1-eigenspace of π acting on $E[r]$. There is a unique sextic twist $\tilde{E}$ of E over $\mathbb{F}_{p^2}$ with $r \mid \#\tilde{E}(\mathbb{F}_{p^2})$ [2]; let $\Psi : \tilde{E} \to E$ be the associated twisting isomorphism. Let $\tilde{Q} \in \tilde{E}(\mathbb{F}_{p^2})$ be a point of order r; then $Q = \Psi(\tilde{Q}) \notin E(\mathbb{F}_p)$. The group $\mathbb{G}_2 = \langle Q \rangle$ is the p-eigenspace of π acting on $E[r]$. Points in $\mathbb{G}_2$ have x-coordinates in $\mathbb{F}_{p^6}$, a property that is exploited in the important denominator elimination speedup [187]. For future reference, we note that for a suitable third root of unity $\delta \in \mathbb{F}_p$, the automorphism $\phi : (x, y) \mapsto (\delta x, -y)$ has order 6 and satisfies $\phi(P) = p^2 P$ for all $P \in \mathbb{G}_1$.

The optimal ate pairing [24] is $a_{\text{opt}} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ defined by

$$a_{\text{opt}} : (P, Q) \mapsto \left(f_{6z+2,Q}(P) \cdot \ell_{(6z+2)Q,\pi(Q)}(P) \cdot \ell_{(6z+2)Q+\pi(Q),-\pi^2(Q)}(P)\right)^{(p^{12}-1)/r}, \tag{7.2}$$

where $\ell_{A,B}$ denotes the line through points A and B, and where $f_{6z+2,Q}$ and the line functions are semi-normalized. This pairing is called "optimal" because the length $6z + 2$ of the Miller function appearing in (7.2) has bitlength roughly one-fourth that of the length r of the Miller function in the Tate pairing definition (7.1) [24]. The exponent $(p^{12} - 1)/r$ in (7.2) can be written as $(p^6 - 1)(p^2 + 1)(p^4 - p^2 + 1)/r$. Since p-th powering is inexpensive in $\mathbb{F}_{p^{12}}$, the exponentiation by $(p^6 - 1)(p^2 + 1)$ is said to be the *easy part* of the final exponentiation in (7.2); the exponentiation by $(p^4 - p^2 + 1)/r$ is called the *hard part.*

7.2.4 The eta pairing

Consider the supersingular elliptic curve $E : Y^2 + Y = X^3 + X$ defined over $\mathbb{F}_{2^m}$, with m odd. For simplicity, we further assume that $m \equiv 7 \pmod 8$. We have $\#E(\mathbb{F}_{2^m}) = 2^m + 1 + 2^{(m+1)/2}$. Let r be a large prime divisor of $\#E(\mathbb{F}_{2^m})$. The embedding degree is $k = 4$. The extension field $\mathbb{F}_{2^{4m}}$ is represented using tower extensions $\mathbb{F}_{2^{2m}} = \mathbb{F}_{2^m}[s]/(s^2 + s + 1)$ and $\mathbb{F}_{2^{4m}} = \mathbb{F}_{2^{2m}}[t]/(t^2 + t + s)$. A distortion map on $\mathbb{G}_1 = E(\mathbb{F}_{2^m})[r]$ is $\psi : (x, y) \mapsto (x + s^2, y + sx + t)$. Let $\mathbb{G}_T$ be the order-r subgroup of $\mathbb{F}^*_{2^{4m}}$. The eta pairing of Barreto et al. [1] is $\eta_T : \mathbb{G}_1 \times \mathbb{G}_1 \to \mathbb{G}_T$ defined by

$$\eta_T : (P, Q) \mapsto f_{T,-P}(\psi(Q))^M, \tag{7.3}$$

where $T = 2^{(m+1)/2} + 1$ and $M = (2^{2m} - 1)(2^m - 2^{(m+1)/2} + 1)$. Note that if $r \approx 2^m$, then the length of the Miller function appearing in (7.3) is approximately half that of the length r of the Miller function in the Tate pairing definition (7.1). Observe also that the final exponentiation by M can be computed relatively quickly since squaring in $\mathbb{F}_{2^{4m}}$ is an inexpensive operation.

7.3 Parallelizing the optimal ate pairing

In this section, we shall assume that all Miller functions and line functions are semi-normalized. Equations involving Miller and line functions hold up to multiplication by nonzero constants. The following are two well-known properties of Miller functions.

Lemma 1 (Miller [58]). *Let a and b be non-negative integers, and let $R \in E(\mathbb{F}_{q^k})$. Then*

(i) $f_{a+b,R} = f_{a,R} \cdot f_{b,R} \cdot \ell_{aR,bR}/v_{(a+b)R}$, *where v_P denotes the vertical line through P; and*

(ii) $f_{ab,R} = f_{b,R}^{a} \cdot f_{a,bR}$.

The method of [102] for parallelizing the computation of a Miller function $f_{s,R}$ is the following. We first write $s = 2^w s_1 + s_0$, where $s_0 < 2^w$. Applying Lemma 1, we obtain

$$f_{s,R} = f_{s_1,R}^{2^w} \cdot f_{2^w,s_1R} \cdot f_{s_0,R} \cdot \frac{\ell_{2^w s_1 R, s_0 R}}{v_{sR}}. \tag{7.4}$$

If s_0 is small, then the Miller function $f_{s_0,R}$ can be computed relatively cheaply. Thus the computation of $f_{s,R}$ can be parallelized by computing $f_{s_1,R}^{2^w}$ on one processor and f_{2^w,s_1R} on a second processor. The parameter w should be carefully selected in order to balance the time of the two function computations. The relevant criteria for selecting w include the Hamming weight of s_1 (which determines the number of additions in the Miller loop for the first function), and the cost of the w-fold squaring in the first function relative to the cost of computing s_1R in the second function. The w-fold squaring in $f_{s_1,R}^{2^w}$ can be sped up by first computing $\alpha = f_{s_1,R}^{(p^6-1)(p^2+1)}$ (recall that exponentiation by $(p^6-1)(p^2+1)$ is the easy part of the final exponentiation), followed by α^{2^w}. The advantage of this 'delayed squaring' trick is that α belongs to the order-$(p^4 - p^2 + 1)$ cyclotomic subgroup of $\mathbb{F}_{p^{12}}^*$ whence Karabina's squaring method [173] can be deployed at a cost of 12 $\mathbb{F}_p$ multiplications plus some small overhead — this is considerably less than squaring a general element in $\mathbb{F}_{p^{12}}$ which costs 24 $\mathbb{F}_p$ multiplications.

Each of the two expensive Miller function computations in (7.4) can be recursively parallelized. For this purpose, one writes

$$s = s_t 2^{w_t} + \cdots + s_2 2^{w_2} + s_1 2^{w_1} + s_0,$$

where $s_i 2^{w_i} = (s \bmod 2^{w_{i+1}}) - (s \bmod 2^{w_i})$ for some $w_t > \cdots > w_2 > w_1 > w_0 = 0$. We also note that the lines that appear in (7.2) should be scheduled for execution on the processor that handles s_0 since this processor does not have any overhead of computing consecutive squarings.

Remark 1. We were unable to find any effective method for parallelizing the hard part of the final exponentiation. The exponent $(p^4 - p^2 + 1)/r$ can be decomposed into a multi-addition chain requiring the consecutive z-th powers α^z, α^{z^2} and α^{z^3} where $\alpha \in \mathbb{F}_{p^{12}}$ [172]. However, the extremely low Hamming weight of z limits the potential for parallelization. Furthermore, techniques that exploit very fast squaring (e.g., [188]) and fixed bases (e.g., [64]) are not applicable.

Remark 2. We measured the parallelization overhead in the target platforms using OpenMP Microbenchmarks [189] and observed costs on the order of 1μs for multi-threading primitives such as thread creation or synchronization; the costs confirm those reported in [39]. These overheads also scaled linearly with the number of threads involved. During 1μs on a 2.53GHz machine, it is possible to perform 6 $\mathbb{F}_{p^2}$ multiplications or 8 $\mathbb{F}_{p^2}$ squarings [20]. On the other hand, the most expensive $\mathbb{F}_{p^{12}}$ operation within the Miller loop of a pairing computation is a sparse multiplication costing 13 $\mathbb{F}_{p^2}$ multiplications. Hence, it seems that any potential speedup in a parallel implementation of $\mathbb{F}_{p^{12}}$ arithmetic (for example, by assigning each of the three $\mathbb{F}_{p^6}$ multiplications required for an $\mathbb{F}_{p^{12}}$ multiplication to different processors), as suggested by Grabher, Großschädl and Page [96], would be nullified by the overheads. Furthermore, this approach is clearly not scalable to many processors.

7.4 Optimal Weil pairings

In this section, E is BN curve defined over $\mathbb{F}_p$ with BN parameter z. Unless otherwise stated, all functions are assumed to be normalized. By a 'pairing' we will mean a non-degenerate bilinear pairing from $\mathbb{G}_1 \times \mathbb{G}_2$ to $\mathbb{G}_T$. Note that if e is a pairing and $\gcd(\ell, r) = 1$, then e^ℓ is also a pairing. It is understood that pairing values are defined to be 1 if either input point is equal to ∞.

The classical Weil pairing [58] is $e_W : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ defined by

$$e_W : (P, Q) \mapsto -\frac{f_{r,P}(Q)}{f_{r,Q}(P)}. \tag{7.5}$$

Note that the Weil pairing does not have a final exponentiation. The two Miller functions in (7.5) each have length approximately z^4 and can be independently computed on two processors.

7.4.1 Hess's Weil pairing construction

Hess [178] developed a framework for designing Tate and Weil-type pairings. Let s be an integer. Let $h = \sum_{i=0}^{d} h_i x^i \in \mathbb{Z}[x]$ with $h(s) \equiv 0 \pmod{r}$, and let $m = h(s)/r$; we assume that $r \nmid m$. For $R \in E[r]$, Lemma 1(ii) gives

$$f_{r,R}^{m} = f_{mr,R} = f_{\sum_{i=0}^{d} h_i s^i, R}.$$

As shown by Vercauteren [24, Theorem 1], repeated application of Lemma 1 yields

$$f_{r,R}^{m} = \prod_{i=1}^{d} f_{s^i,R}^{h_i} \cdot \left(\prod_{i=0}^{d} f_{h_i, s^i R} \cdot \prod_{i=0}^{d-1} \frac{\ell_{s_{i+1}R, h_i s^i R}}{v_{s_i R}} \right) \tag{7.6}$$

where $s_i = \sum_{j=i}^{d} h_j s^j$. Let $f_{s,h,R}$ denote the expression within the parenthesis in (7.6); $f_{s,h,R}$ is called an *extended Miller function* and can be verified to have divisor $\sum_{i=0}^{d} h_i((s^i R) - (\infty))$. Note that $f_{s,R} = f_{s,s-x,R}$. Hess's result for Weil pairings specialized to BN curves is the following.

Theorem 1 (Theorem 1 in [178]). *Let s be a primitive 6th root of unity modulo r^2 with $s \equiv p^2 \pmod{r}$. Let $h \in \mathbb{Z}[x]$ with $h(s) \equiv 0 \pmod{r}$ and $h(s) \not\equiv 0 \pmod{r^2}$. Then $e_{s,h} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ defined by*

$$e_{s,h} : (P, Q) \mapsto \left(\frac{f_{s,h,P}(Q)}{f_{s,h,Q}(P)} \right)^{6} \tag{7.7}$$

is a pairing. In particular, there exists a sixth root of unity $w \in \mathbb{F}_p$ such that taking $h(x) = p^2 - x$ gives the eil pairing $e_{\text{eil}} : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ defined by

$$e_{\text{eil}} : (P, Q) \mapsto w \frac{f_{p^2,P}(Q)}{f_{p^2,Q}(P)}. \tag{7.8}$$

7.4.2 The $\boldsymbol{\alpha}$ Weil pairing

Taking $h(x) = (2z+1) + (6z^2+2z)x$ in Theorem 1 yields the pairing $\alpha' : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ defined by

$$\alpha' : (P, Q) \mapsto w \frac{f_{2z+1,P}(Q)}{f_{2z+1,Q}(P)} \cdot \frac{f_{6z^2+2z,p^2P}(Q)}{f_{6z^2+2z,p^2Q}(P)} \cdot \frac{\ell_{(6z^2+2z)p^2P,(2z+1)P}(Q)}{\ell_{(6z^2+2z)p^2Q,(2z+1)Q}(P)} \tag{7.9}$$

for some sixth root of unity $w \in \mathbb{F}_p$. Since $6 \mid p^6 - 1$ and $r \nmid (p^6-1)(p^2+1)$, it follows that the map $\alpha : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ defined by

$$\alpha = (\alpha')^{(p^6-1)(p^2+1)} : (P, Q) \mapsto \left(\frac{f_{2z+1,P}(Q)}{f_{2z+1,Q}(P)} \cdot \frac{f_{6z^2+2z,p^2P}(Q)}{f_{6z^2+2z,p^2Q}(P)} \right)^{(p^6-1)(p^2+1)} \tag{7.10}$$

is also a pairing. The advantage of α over α' is that field elements that lie in a proper subfield of $\mathbb{F}_{p^{12}}$ can be ignored. In particular, the four Miller functions in (7.10) only need to be semi-normalized, the important denominator elimination speedup of [187] can be applied, and the two line functions in (7.9) can be ignored. Furthermore, the delayed squaring technique of §7.3 can be employed as described below. In order to shorten the length of the Miller functions $f_{6z^2+2z,R}$ for $R \in \{p^2P, p^2Q\}$ in (7.10), we can use Lemma 1(ii) to write

$$f_{6z^2+2z,R} = f_{z,(6z+2)R} \cdot f^{z}_{6z+2,R}.$$

The revised formula for the α pairing now has 6 Miller functions, and it may appear that at least 6 processors would be necessary to effectively parallelize the pairing. However, we observe that

$$f_{6z^2+2z,p^2Q}(P) = f^{p^2}_{6z^2+2z,Q}(P) \tag{7.11}$$

since $p^2Q = \pi^2(Q)$ and $\pi(P) = P$. Lemma 2 shows that an analogous conclusion can be drawn with P and Q exchanged.

Lemma 2. *For all $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$ we have*

$$f^{(p^6-1)(p^2+1)}_{6z^2+2z,p^2P}(Q) = f^{p^2(p^6-1)(p^2+1)}_{6z^2+2z,P}(Q). \tag{7.12}$$

Proof. To simplify the notation, we set $a = 6z^2 + 2z$ and $c = (p^6 - 1)(p^2 + 1)$. Two applications of Lemma 1(ii) yields

$$f^{c}_{ap^2,P} \;=\; f^{ac}_{p^2,P} \cdot f^{c}_{a,p^2P} \;=\; f^{cp^2}_{a,P} \cdot f^{c}_{p^2,aP}. \tag{7.13}$$

Let $\tilde{\pi} : (x, y) \mapsto (x^{p^2}, y^{p^2})$ be the p^2-power Frobenius acting on the twist $\tilde{E}$. Let $\tilde{\mathbb{G}}_1$ and $\tilde{\mathbb{G}}_2$ be the 1-eigenspace and the p^2-eigenspace, respectively, of $\tilde{\pi}$ acting on $\tilde{E}[r]$. Then $\Psi^{-1}(\mathbb{G}_1) = \tilde{\mathbb{G}}_2$ and $\Psi^{-1}(\mathbb{G}_2) = \tilde{\mathbb{G}}_1$, where $\Psi : \tilde{E} \to E$ is the twisting isomorphism[1]. Lemma 6 of [190] applied to $\tilde{E}$ shows that $(\tilde{P}, \tilde{Q}) \mapsto \tilde{f}_{p^2,\tilde{P}}(\tilde{Q})$ is a pairing on $\tilde{\mathbb{G}}_2 \times \tilde{\mathbb{G}}_1$, where $\tilde{f}$ is a normalized Miller function associated with $\tilde{E}$. Thus

$$\tilde{f}_{p^2,a\tilde{P}}(\tilde{Q}) = \tilde{f}^{a}_{p^2,\tilde{P}}(\tilde{Q}). \tag{7.14}$$

Now, it follows from the work of [191] (see also the proof of Theorem 1 in [26]) that $\tilde{f}^{c}_{p^2,\tilde{P}}(\tilde{Q}) = f^{c}_{p^2,P}(Q)$ where $P = \Psi^{-1}(\tilde{P}) \in \mathbb{G}_1$ and $Q = \Psi^{-1}(\tilde{Q}) \in \mathbb{G}_2$. Hence (7.14) can be written as

$$f^{c}_{p^2,aP}(Q) = f^{ac}_{p^2,P}(Q).$$

The result now follows from (7.13). □

[1]We couldn't find the statement $\Psi(\tilde{\mathbb{G}}_2) = \mathbb{G}_1$ in the literature. For completeness, we include a proof in Appendix 7.A.

Using (7.11) and (7.12), we obtain the following alternate formulation of the α pairing:

$$\alpha : (P,Q) \mapsto \left(\frac{f_{2z+1,P}(Q)}{f_{2z+1,Q}(P)} \cdot \left(\frac{f_{z,(6z+2)P}(Q) \cdot f^{z}_{6z+2,P}(Q)}{f_{z,(6z+2)Q}(P) \cdot f^{z}_{6z+2,Q}(P)} \right)^{p^2} \right)^{(p^6-1)(p^2+1)}. \tag{7.15}$$

Now, if $f_{z,R}$ is computed first, then $f_{2z+1,R}$ and $f_{6z+2,R}$ can be computed thereafter with little additional work. Thus, there are effectively only 4 Miller functions in (7.15). Each of these Miller functions has length approximately z, and therefore the α pairing is considered optimal in the sense of [24].

Figure 7.1 illustrates the execution path when the 4 Miller functions in (7.15) are computed in parallel using 4 processors. A further optimization is to raise the Miller functions to the power $(p^6-1)(p^2+1)$ as soon as they are computed — this enables the use of Karabina's fast squaring when computing $f^{z}_{6z+2,P}(Q)$ and $f^{z}_{6z+2,Q}(P)$. Note also that since $(6z+2)+p-p^2+p^3 \equiv 0 \pmod{r}$, we have $(6z+2)Q = -\pi(Q)+\pi^2(Q)-\pi^3(Q)$ and thus $(6z+2)Q$ can be computed very quickly. The fourth processor in Figure 7.1 is the bottleneck because of the exponentiation by z.

$$\begin{tikzcd}
\text{1.}\ (6z+2)P \arrow[rr] & & f^{p^2}_{z,(6z+2)P}(Q) \arrow[rr] & & f_{2z+1,P}(Q) \cdot f^{p^2}_{6z^2+2z,P}(Q) \\
\text{2.}\ f_{2z+1,P}(Q) \arrow[r] \arrow[rrr, bend right] & f_{6z+2,P}(Q) \arrow[r] & f^{z}_{6z+2,P}(Q) \arrow[r] & f_{2z+1,P}(Q) \cdot f^{zp^2}_{6z+2,P}(Q) \arrow[ur] & \\
\text{3.}\ (6z+2)Q \arrow[rr] & & f^{p^2}_{z,(6z+2)Q}(P) \arrow[rr] & & f_{2z+1,Q}(P) \cdot f^{p^2}_{6z^2+2z,Q}(P) \\
\text{4.}\ f_{2z+1,Q}(P) \arrow[r] \arrow[rrr, bend right] & f_{6z+2,Q}(P) \arrow[r] & f^{z}_{6z+2,Q}(P) \arrow[r] & f_{2z+1,Q}(P) \cdot f^{zp^2}_{6z+2,Q}(P) \arrow[ur] &
\end{tikzcd}$$

Figure 7.1: Execution path for computing the α pairing on 4 processors.

In §7.4.3, we present a variant of the Weil pairing that is slightly faster than the α pairing.

7.4.3 The $\boldsymbol{\beta}$ Weil pairing

Consider $h(x) = (6z+2)+x-x^2+x^3$. From (7.6) we obtain

$$f^{-1}_{p,h,R} = f^{-m}_{r,R} \cdot f_{p,R} \cdot f^{-1}_{p^2,R} \cdot f_{p^3,R} \tag{7.16}$$

and

$$f_{p,h,R} = f_{6z+2,R} \cdot f_{-1,p^2R} \cdot \frac{\ell_{(6z+2)R,(p-p^2+p^3)R}}{v_\infty} \cdot \frac{\ell_{pR,(-p^2+p^3)R}}{v_{(p-p^2+p^3)R}} \cdot \frac{\ell_{-p^2R,p^3R}}{v_{(-p^2+p^3)R}}. \tag{7.17}$$

Theorem 2. *For $h(x) = (6z+2) + x - x^2 + x^3$, the map $\beta' : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ defined by*

$$\beta' : \ (P,Q) \mapsto w \left(\frac{f_{p,h,P}(Q)}{f_{p,h,Q}(P)} \right)^p \frac{f_{p,h,pP}(Q)}{f_{p,h,Q}(pP)} \tag{7.18}$$

is a pairing, where $w \in \mathbb{F}_p$ is some sixth root of unity.

Proof. For simplicity, multiplicative factors that are sixth roots of unity will be omitted in the proof. For $y \in \{r, p, p^2, p^3\}$, define the functions

$$\gamma_y : (P,Q) \mapsto \frac{f_{y,P}(Q)^p}{f_{y,Q}(P)^p} \cdot \frac{f_{y,pP}(Q)}{f_{y,Q}(pP)}$$

on $\mathbb{G}_1 \times \mathbb{G}_2$. Since $f_{p,h,P}^{-1} = f_{r,P}^{-m} \cdot f_{p,P} \cdot f_{p^2,P}^{-1} \cdot f_{p^3,P}$, it follows that

$$\beta'(P,Q)^{-1} = \gamma_r(P,Q)^{-m} \cdot \gamma_p(P,Q) \cdot \gamma_{p^2}(P,Q)^{-1} \cdot \gamma_{p^3}(P,Q)$$

for all $(P,Q) \in \mathbb{G}_1 \times \mathbb{G}_2$. The conclusion that β' is a pairing immediately follows if it can be shown that γ_r, γ_p, γ_{p^2} and γ_{p^3} are pairings.

Now, $(P,Q) \mapsto f_{r,P}(Q)/f_{r,Q}(P)$ and $(P,Q) \mapsto f_{p^2,P}(Q)/f_{p^2,Q}(P)$ are, respectively, the classic Weil pairing (7.5) and the eil pairing (7.8). It follows that γ_r and γ_{p^2} are also pairings.

Using the facts that $f_{p^2,R} = f_{p,R}^p \cdot f_{p,pR}$ (Lemma 1(ii)) and that $(P,Q) \mapsto f_{p,Q}(P)$ is a pairing (Lemma 6 of [190]), the eil pairing can be written as

$$\frac{f_{p^2,P}(Q)}{f_{p^2,Q}(P)} \ = \ \frac{f_{p,P}(Q)^p}{f_{p,Q}(P)^p} \cdot \frac{f_{p,pP}(Q)}{f_{p,pQ}(P)} \ = \ \frac{f_{p,P}(Q)^p}{f_{p,Q}(P)^p} \cdot \frac{f_{p,pP}(Q)}{f_{p,Q}(pP)} \ = \ \gamma_p(P,Q),$$

and hence γ_p is a pairing.

Finally, we note that $(P,Q) \mapsto f_{p^2,Q}(P)$ is a pairing since $f_{p^2,Q} = f_{p,Q}^p \cdot f_{p,pQ}$ and $(P,Q) \mapsto f_{p,Q}$ is a pairing. Using this observation and the fact that $f_{p^3,R} = f_{p,R}^{p^2} \cdot f_{p^2,pR}$, one can check that

$$\gamma_{p^3}(P,Q) = \gamma_p(P,Q)^{p^2} \cdot \gamma_{p^2}(pP,Q).$$

Hence γ_{p^3} is a pairing. □

Remark 3. The proof of Theorem 2 can easily be modified for all polynomials $h(x) \in \mathbb{Z}[x]$ that satisfy $h(p) \equiv 0 \pmod{r}$.

Since $6 \mid p^6 - 1$ and $r \nmid (p^6-1)(p^2+1)$, the map $\beta : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ defined by

$$\beta = (\beta')^{(p^6-1)(p^2+1)} : (P,Q) \mapsto \left(\left(\frac{f_{p,h,P}(Q)}{f_{p,h,Q}(P)} \right)^p \frac{f_{p,h,pP}(Q)}{f_{p,h,Q}(pP)} \right)^{(p^6-1)(p^2+1)} \tag{7.19}$$

is also a pairing. Since each extended Miller function in (7.19) is essentially a Miller function of length approximately z (see (7.17)), the β pairing is considered optimal. As was the case with the α pairing, the exponentiation by $(p^6-1)(p^2+1)$ means that the four extended Miller functions in (7.19) only need to be semi-normalized and denominator elimination can be applied. Moreover, the vertical lines $v_{(p-p^2+p^3)R}$, $v_{(-p^2+p^3)R}$, f_{-1,p^2R} and $\ell_{(6z+2)R,(p-p^2+p^3)R}$ for $R \in \{P, pP, Q\}$ in (7.17) can be ignored. Once pP has been computed, the remaining line functions $\ell_{pR,(-p^2+p^3)R}$ and ℓ_{-p^2R,p^3R} for $R \in \{P, pP, Q\}$ can be computed at very little additional cost since $p^2P = \phi(P)$, $p^3P = \phi(pP)$, and $pQ = \pi(Q)$. Furthermore, the delayed squaring technique of §7.3 can be employed if the extended Miller functions are divided using (7.4).

Figure 7.2 illustrates the execution path when the 4 extended Miller functions in (7.19) are computed in parallel using 4 processors. The fourth processor is the bottleneck, and thus it is desirable to accelerate the computation of pP. To this effect, we observe that

$$p \equiv 2z(p^2-2) + p^2 - 1 \pmod{r},$$

whence

$$pP \;=\; 2z(p^2-2)P + p^2P - P \;=\; 2z(\phi(P) - 2P) + \phi(P) - P. \tag{7.20}$$

Thus, computing pP has roughly the same cost as zP.

$$\begin{tikzcd}
1.\ f^p_{6z+2,P}(Q) \arrow[rr] & & f^p_{6z+2,P}(Q)\cdot f_{6z+2,pP}(Q) \\
2.\ pP \arrow[r] & f_{6z+2,pP}(Q) \arrow[ur] & \\
3.\ f^p_{6z+2,Q}(P) \arrow[rr] & & f^p_{6z+2,Q}(P)\cdot f_{6z+2,Q}(pP) \\
4.\ pP \arrow[r] & f_{6z+2,Q}(pP) \arrow[ur] &
\end{tikzcd}$$

Figure 7.2: Execution path for computing the β pairing on 4 processors.

7.5 Parallel implementation of the BN pairings

The parallelization approaches described in §7.3 and §7.4 were implemented on top of the state-of-the-art implementation of an optimal ate pairing at the 128-bit security level described in [20]. The underlying elliptic curve is a BN curve with parameter $z = -(2^{62} + 2^{55} + 1)$ [97].

Let π denote the number of available processors on the target platform. To select parameters $(w_{\pi-1}, \ldots, w_2, w_1, w_0)$ that split the Miller loop, we employ the load balancing scheme suggested in [102] with fine granularity, taking into account the relative cost of inversions, multiplications, squarings, additions and modular reductions on the target platform. With the optimal parameters determined, elementary operation counting makes it possible to estimate the performance improvement of the corresponding implementation. Figure 7.3 presents the estimated speedups for the parallelization approaches discussed in this work in comparison with the optimal ate serial implementation of [20]. Notice how the performance of the α and β Weil pairings scales better with the number of processing cores. Scaling stills suffers from the same saturation effect experienced by the ate pairing variants, but at a higher number of cores.

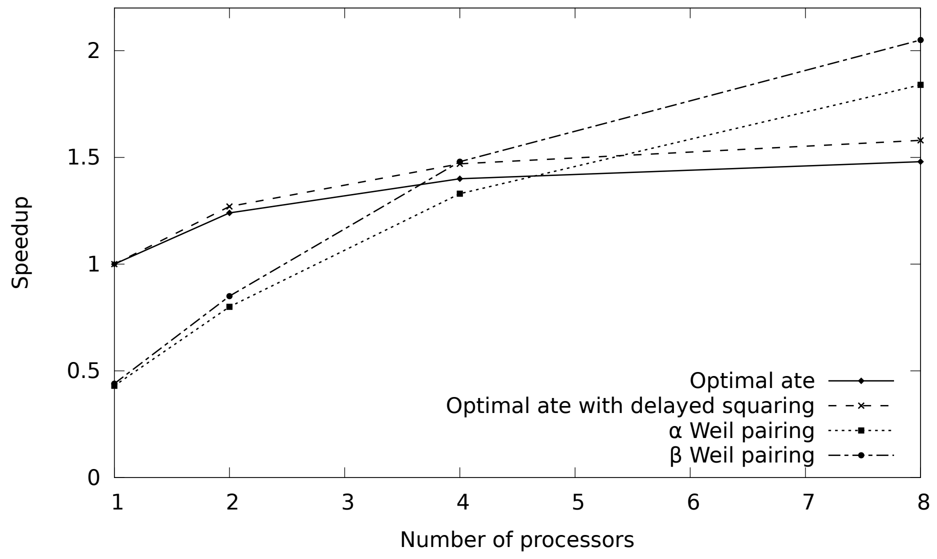


Figure 7.3: Estimated speedups for several parallelization approaches of BN pairings. Speedups are computed in relation to a serial implementation of the optimal ate pairing.

The parallel implementation was realized on platforms — a 2-core Intel Core i5 M540 32nm 2.53GHz machine ("Platform 1") and a 4-core Intel Core i7 Sandy Bridge 32nm 2.0GHz machine ("Platform 2"), using GCC v4.5.2 as compiler with optimization flags `-O3 -funroll-loops`. Parallel sections were implemented through the compiler's native OpenMP support. The same split in the Miller loop was used in both machines, as they have similar field operation costs. Table 7.1 presents the experimental results, including both the speedups estimated by operation counting and actual timings. For the optimal ate pairing, the table confirms that delayed squaring yields a slightly better scaling, and that the increase in overall speedup starts to stagnate at 8 cores. Even with these obstacles, the pairing latency of the optimal ate pairing is reduced by 18-20% when using 2 processor cores, a significant improvement over the 10% reported as a preliminary result for the R-ate pairing in [102]. The measured timings are compatible with the estimated speedups — the differences are due to the parallelization overheads that are not accounted for in the model for speedup estimation. The gap between the estimated and measured timings increases with the number of cores due to the linear increase in overhead.

	Number of threads			
Estimated speedup	**1**	**2**	**4**	**8**
Optimal ate	1.00	1.24	1.40	1.48
Optimal ate with delayed squaring	1.00	1.27	1.47	1.58
α Weil	0.43	0.80	1.33	1.84
β Weil	0.44	0.86	1.48	2.05
Platform 1: Intel Core i5 Westmere 32nm	**1**	**2**	**4***	**8***
Optimal ate – latency	1688	1394	1254	1204
Optimal ate – speedup	1.00	1.21	1.35	1.40
Optimal ate with delayed squaring – latency	–	1371	1189	1151
Optimal ate with delayed squaring – speedup	–	1.23	1.42	1.47
α Weil – latency	–	–	1318	1006
α Weil – speedup	–	–	1.28	1.68
β Weil – latency	–	–	1227	915
β Weil – speedup	–	–	1.38	1.84
Platform 2: Intel Core i7 Sandy Bridge 32nm	**1**	**2**	**4**	**8***
Optimal ate – latency	1562	1287	1137	1107
Optimal ate – speedup	1.00	1.21	1.37	1.41
Optimal ate with delayed squaring – latency	–	1260	1080	1056
Optimal ate with delayed squaring – speedup	–	1.24	1.45	1.48
α Weil – latency	–	–	1272	936
α Weil – speedup	–	–	1.23	1.67
β Weil – latency	–	–	1104	840
β Weil – speedup	–	–	1.41	1.86

Table 7.1: Experimental results for serial/parallel executions of BN pairings. Times are presented in thousands of clock cycles and the speedups are computed as the ratio of the execution time of a serial implementation and of a parallel implementation. The dashes represent data points where there is no expected improvement over the serial implementation. The columns marked with (*) present estimates based on per-thread data.

The performance of the β Weil pairing is generally superior to the α Weil pairing due to the difference in the cost of computing zP in the former and an exponentiation by z in the cyclotomic subgroup in the latter (see Figures 7.1 and 7.2). It is important to observe that since the β pairing is tailored for parallel execution, any future improvements in the parallelization of the Miller loop in the optimal ate variants can be directly applied to the β pairing. In the columns of Table 7.1 marked with (*), we present estimates for machines with higher numbers of cores. These estimates were obtained by running multiples threads per core and then measuring the cost of the most expensive thread. This serves as an accurate prediction of performance scaling in future machines, assuming that critical platform characteristics such as the memory organization and multi-threading overhead will not change dramatically.

7.6 Parallel implementation of the eta pairing

Four approaches to parallelizing the eta pairing are outlined in §§7.6.1–7.6.4. Our implementation is then described in §7.6.5.

7.6.1 Algorithm 1

Aranha et al. [102] applied the parallelization method given in (7.4) to the reverse-loop eta pairing algorithm presented in [79], obtaining Algorithm 7.1. The Miller loop is split across π processors inside a parallel execution section. The intermediate results F_i, $0 \leq i \leq \pi - 1$, from each core are multiplied together in parallel in step 17 at a cost of $\lceil \log_2 \pi \rceil$ $\mathbb{F}_{2^{4m}}$-multiplications. The starting points $(w_0, w_1, \ldots, w_{\pi-1})$ can be determined so that the precomputation cost of processor i to compute $(x_Q)^{2^{w_i}}$, $(y_Q)^{2^{w_i}}$, $(x_P)^{1/2^{w_i}}$, $(y_P)^{1/2^{w_i}}$ in step 10 can be balanced with the number of iterations, $w_{i+1} - w_i$, so that all processors incur the same cost [102]. This balancing can be deduced statically with a simple platform-dependent operation count. The main advantage of Algorithm 1 is the negligible storage requirements, which makes it well suited for embedded platforms.

7.6.2 Algorithm 2

An alternate parallelization strategy proposed in [98] is to precompute all the squares and square-roots $(x_Q)^{2^j}$, $(y_Q)^{2^j}$, $(x_P)^{1/2^j}$, $(y_P)^{1/2^j}$ for $0 \leq j < (m-1)/2$ and split the Miller loop in equal parts so that $w_{i+1} - w_i = \lceil (m-1)/2\pi \rceil$ without the need for any static scheduling. This strategy requires storage capacity for $4 \cdot (m-1)/2 \approx 2m$ field elements, and therefore is more useful for the desktop platform scenario where storage is abundant. However, Algorithm 2 was found to be slower that Algorithm 1 because the precomputation cost is higher. Since this precomputation step is executed serially in Algorithm 2, it can be viewed as the equivalent of all the processors incurring a higher precomputation cost compared with Algorithm 1.

Algorithm 7.1 for parallelization of the eta pairing on π processors.

Input: $P = (x_P, y_P)$, $Q = (x_Q, y_Q) \in E(\mathbb{F}_{2^m})[r]$, starting point w_i for processor $i \in [0, \pi - 1]$.
Output: $\eta_T(P, Q) \in \mathbb{F}^*_{2^{4m}}$.

```
 1: $y_P \leftarrow y_P + 1$
 2: **parallel section** (processor $i$)
 3: **if** $i = 0$ **then**
 4:    $u_i \leftarrow x_P, \quad v_i \leftarrow x_Q$
 5:    $g_{0i} \leftarrow u_i \cdot v_i + y_P + y_Q + 1, \quad g_{1i} \leftarrow u_i + x_Q, \quad g_{2i} \leftarrow v_i + x_P^2$
 6:    $G_i \leftarrow g_{0i} + g_{1i}s + t, \quad L_i \leftarrow (g_{0i} + g_{2i}) + (g_{1i} + 1)s + t, \quad F_i \leftarrow L_i \cdot G_i$
 7: **else**
 8:    $F_i \leftarrow 1$
 9: **end if**
10: $x_{Q_i} \leftarrow (x_Q)^{2^{w_i}}, \quad y_{Q_i} \leftarrow (y_Q)^{2^{w_i}}, \quad x_{Pi} \leftarrow (x_P)^{1/2^{w_i}}, \quad y_{Pi} \leftarrow (y_P)^{1/2^{w_i}}$
11: **for** $j \leftarrow w_i$ **to** $w_{i+1} - 1$ **do**
12:    $x_{Pi} \leftarrow \sqrt{x_{Pi}}, \quad y_{Pi} \leftarrow \sqrt{y_{Pi}}, \quad x_{Q_i} \leftarrow x_{Q_i}^2, \quad y_{Q_i} \leftarrow y_{Q_i}^2$
13:    $u_i \leftarrow x_{Pi}, \quad v_i \leftarrow x_{Q_i}$
14:    $g_{0i} \leftarrow u_i \cdot v_i + y_{Pi} + y_{Q_i} + 1, \quad g_{1i} \leftarrow u_i + x_{Q_i}$
15:    $G_i \leftarrow g_{0i} + g_{1i}s + t, \quad F_i \leftarrow F_i \cdot G_i$
16: **end for**
17: $F \leftarrow \prod_{i=0}^{\pi-1} F_i$
18: **end parallel**
19: **return** $F^M$
```

7.6.3 Algorithm 3

A straightforward improvement to Algorithm 2 is to further parallelize the precomputation step using at most 4 processors, where each processor takes care of one iterated squaring or square-root routine.

7.6.4 Algorithm 4

An improvement to Algorithm 3 is to utilize a recently-introduced time-memory trade-off for computing consecutive squares [145]. Since squaring and square-root are linear maps on $\mathbb{F}_{2^m} = \mathbb{F}_2[x]/(f(x))$, for any fixed power 2^k, a table T of $16\lceil m/4 \rceil$ $\mathbb{F}_{2^m}$ elements can be precomputed so that

$$T[j, i_0 + 2i_1 + 4i_2 + 8i_3] = (i_0 x^{4j} + i_1 x^{4j+1} + i_2 x^{4j+2} + i_3 x^{4j+3})^{2^k} \qquad (7.21)$$

for all $0 \leq j < \lceil m/4 \rceil$ and $i_0, i_1, i_2, i_3 \in \{0, 1\}$. After the table has been computed, any power α^{2^k} can be quickly computed as

$$\alpha^{2^k} = \sum_{j=0}^{\lceil m/4 \rceil} T[j, \lfloor a/2^{4j} \rfloor \bmod 16],$$

where a denotes the integer whose base-2 digits are the coefficients of the polynomial $\alpha \in \mathbb{F}_2[x]$ (which has degree at most m-1). A similar table can be computed for 2^{-k} powers. The objective of this technique is to permit the computation of any 2^k or 2^{-k}-powers in constant time independent of how large k is. When applied to the precomputation step required for parallelization, this in turn allows each processor to use the same partition size at a storage cost of $2 \cdot 16 \cdot m/4 = 8m$ field elements per processor, divided into two tables for squarings and square-roots. This has the effect of making load balancing trivial while speeding up the precomputation step. The technique has further application in reducing an important part of the serial cost of parallel pairing computation — the final exponentiation by M. The time-memory trade-off can also be used to accelerate the $2^{(m+1)/2}$-th power in $\mathbb{F}_{2^{4m}}$ and inversion with the Itoh-Tsujii algorithm [146, 98].

7.6.5 Implementation report

For concreteness, we focus our attention on the supersingular elliptic curve $E_1 : Y^2 + Y = X^3 + X$ over $\mathbb{F}_{2^{1223}}$. This curve is a candidate for the 128-bit security level [67]. We have $\#E_1(\mathbb{F}_{2^{1223}}) = 5r$ where r is a 1221-bit prime. The representation chosen for the underlying field is $\mathbb{F}_{2^{1223}} = \mathbb{F}_2[x]/(x^{1223} + x^{255} + 1)$.

We implemented Algorithms 1–4 on an Intel Core i5 Westmere 32nm processor ("Platform 1") and an Intel Core i7 Sandy Bridge 32nm processor ("Platform 2"), both equipped with the new carry-less multiplier [87]. The finite field implementation closely followed [88] with the exception of multiplication, where a multi-level Karatsuba multiplier of 128-bit granularity that exploits the new native multiplication instruction was deployed. The first level is the traditional 2-way Karatsuba, where each of the three multiplications is solved by a 5-way Karatsuba formula [143]. Each of the final 128-digit multiplications is computed by another Karatsuba instance requiring 3 executions of the carry-less multiplication instruction.

Let M, S, R, T, I be the respective costs of multiplication, squaring, square-root, repeated squaring or square-root computation using a precomputed table as given in (7.21), and inversion in $\mathbb{F}_{2^m}$. The following ratios were obtained from our implementation of arithmetic in $\mathbb{F}_{2^{1223}}$ on Platform 1: $M = 14S$ and $S \approx R$; for Algorithm 1 we have $I = 95M$, and for Algorithm 4 we have $T = 73S$ and $I = 51M$. Using these ratios, the speedups for the eta pairing of the four parallelization approaches over a serial implementation were estimated; the results are depicted in Figure 7.4. It can be observed from Figure 7.4 that the proposed parallelization is expected to be faster and more scalable, at the expense of storage requirements for $8\pi m$ field elements. On Platform 2, the ratios were: $M = 18S$, $S \approx R$, $I = 78M$ for Algorithm 1, and $T = 81S$ and $I = 38M$ for Algorithm 4.

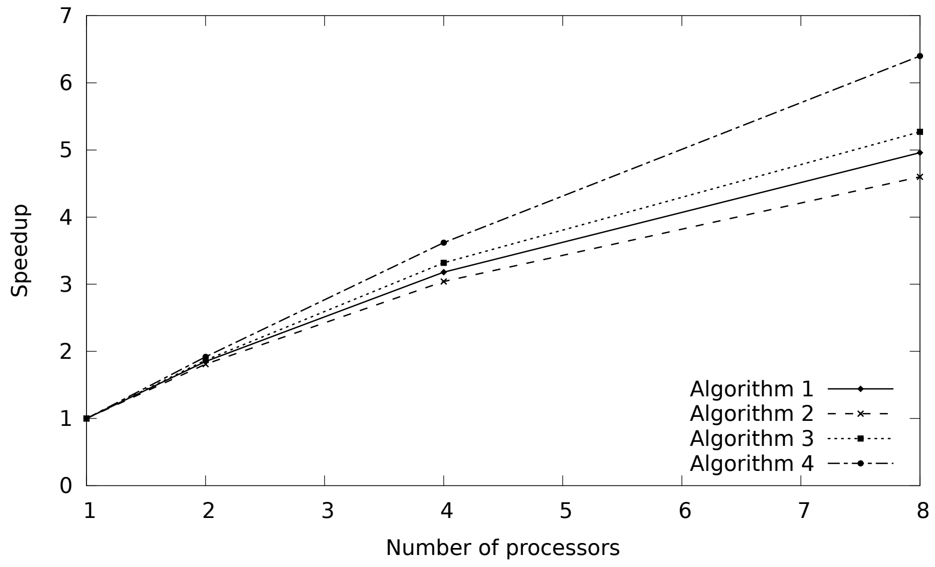


Figure 7.4: Estimated speedups of the eta pairing on Platform 1 for four parallelization algorithms using up to $\pi = 8$ processors.

Table 7.2 presents experimental results for this parallelization and updates results from [88] by accounting for the native support for binary field multiplication. The speedups obtained by the actual implementation closely match what was estimated in Figure 7.4. The one exception is for Algorithm 4 running in 8 threads on Platform 1 where the competition for cache occupancy among the processors for quick access to the precomputed tables degrades performance. One can expect that this effect will be reduced in a proper 8-core machine because the memory hierarchy would be better prepared for parallel access. The native support for binary field multiplication has two effects: dramatically improving the serial pairing computation time from 17.4 million cycles in [102] to 7.2 million cycles (Platform 1) and 6.7 million cycles (Platform 2) in Algorithm 1; and reducing the relative cost of the Miller loop compared to the final exponentiation, which further reduces the 8-core estimated speedups for Platform 1 from 6.11 in [102] to 4.96 in Algorithm 1. The improvements in Algorithm 4 eliminate an important serial cost in the final exponentiation. For Platform 1, this resulted in an increase in the 8-core estimated speedup to 6.40, and an acceleration of the serial pairing latency by about 4.5%.

7.7 Concluding remarks

Our work has demonstrated that asymmetric pairings derived from BN curves can be significantly accelerated on multi-processor machines. Furthermore, our experiments suggest

	Number of threads			
Platform 1: Intel Core i5 Westmere 32nm	**1**	**2**	**4***	**8***
Algorithm 1 – estimated speedup	1.00	1.85	3.18	4.96
Algorithm 1 – latency	7197	3900	2292	1507
Algorithm 1 – speedup	1.00	1.84	3.14	4.77
Algorithm 4 – estimated speedup	1.00	1.92	3.62	6.40
Algorithm 4 – latency	6864	3751	1983	1305
Algorithm 4 – speedup	1.00	1.83	3.46	5.26
Platform 2: Intel Core i7 Sandy Bridge 32nm	**1**	**2**	**4**	**8***
Algorithm 1 – estimated speedup	1.00	1.87	3.29	5.26
Algorithm 1 – latency	6648	3622	2072	1284
Algorithm 1 – speedup	1.00	1.84	3.21	5.17
Algorithm 4 – estimated speedup	1.00	1.94	3.67	6.54
Algorithm 4 – latency	6455	3370	1794	1034
Algorithm 4 – speedup	1.00	1.92	3.60	6.24

Table 7.2: Experimental results for serial/parallel executions the eta pairing. Times are presented in thousands of clock cycles and the speedups are computed as the ratio of the execution time of a serial implementation and of a parallel implementation. The columns marked with (*) present estimates based on per-thread data.

that there are variants of the Weil pairing that are more amenable to parallelization than the optimal ate pairing. Unlike the case with asymmetric pairings, our parallel implementations of the eta pairing on $k = 4$ supersingular elliptic curves come close to achieving the ideal parallelization factor. Nevertheless, we found that asymmetric pairings derived from BN curves are faster than the eta pairing on multi-processor machines. Of course, these conclusions are heavily dependent on the characteristics of the hardware we employed and may change on future generations of multi-core architectures.

7.A Relationship between $\mathbb{G}_1$, $\mathbb{G}_2$, $\tilde{\mathbb{G}}_1$ and $\tilde{\mathbb{G}}_2$

Let $E : Y^2 = X^3 + b$ be a BN curve defined over $\mathbb{F}_p$ with $r = \#E(\mathbb{F}_p)$. We have $\mathbb{F}_{p^{12}} = \mathbb{F}_{p^2}[w]/(w^6 - \xi)$ where $\xi \in \mathbb{F}_{p^2}$ is a non-square and a non-cube [74]. Without loss of generality, we can assume that the equation for the degree-6 twist $\tilde{E}$ over $\mathbb{F}_{p^2}$ for which $r \mid \#\tilde{E}(\mathbb{F}_{p^2})$ is $Y^2 = X^3 + b/\xi$. The twisting isomorphism is $\Psi : \tilde{E} \to E$, $(x, y) \mapsto (w^2x, w^3y)$.

Lemma 3. *Let $\mathbb{G}_1$ and $\mathbb{G}_2$ be the 1- and p-eigenspaces of the p-power Frobenius π acting on $E[r]$, and let $\tilde{\mathbb{G}}_1$ and $\tilde{\mathbb{G}}_2$ be the 1- and p^2-eigenspaces of the p^2-power Frobenius $\tilde{\pi}$ acting on $\tilde{E}[r]$. Then $\Psi(\tilde{\mathbb{G}}_1) = \mathbb{G}_2$ and $\Psi(\tilde{\mathbb{G}}_2) = \mathbb{G}_1$.*

Proof. $\Psi(\tilde{\mathbb{G}}_1) = \mathbb{G}_2$ was proven in [74]. We shall show that $\Psi(\tilde{\mathbb{G}}_2) = \mathbb{G}_1$.

We have $w^{p^2} = w(w^6)^{(p^2-1)/6} = w \cdot \xi^{(p^2-1)/6}$. Let $c = \xi^{(p^2-1)/6}$ so $w^{p^2} = cw$. Then $c^6 = \xi^{p^2-1} \in \mathbb{F}_{p^2}^*$, so c is a 6th root of unity and in fact is in $\mathbb{F}_p$ since $p \equiv 1 \pmod 6$. Furthermore, c has order 6 since ξ is a non-square non-cube in $\mathbb{F}_{p^2}$.

Let $P = (w^2x, w^3y) \in \mathbb{G}_1$ and $\tilde{P} = \Psi^{-1}(P) = (x, y)$. Then

$$\begin{aligned}
\Psi(\tilde{\pi}(\tilde{P})) &= (w^2x^{p^2}, w^3y^{p^2}) \\
&= (w^{2p^2}c^{-2p^2}x^{p^2}, w^{3p^2}c^{-3p^2}y^{p^2}) \\
&= \pi^2(c^{-2}w^2x, c^{-3}w^3y) \\
&= \pi^2(\chi(P)) \\
&= \chi(P)
\end{aligned}$$

where $\chi : (x, y) \mapsto (c^{-2}x, c^{-3}y)$ is an order-6 automorphism of E defined over $\mathbb{F}_p$ and thus satisfies $\chi(P) = p^2P$ or $\chi(P) = p^{10}P$. If $\chi(P) = p^{10}P$, then we have $\Psi(\tilde{\pi}(\tilde{P})) = p^{10}P$, and applying Ψ^{-1} to both sides gives $\tilde{\pi}(\tilde{P}) = p^{10}\tilde{P}$ — this is impossible since p^{10} is not an eigenvalue of $\tilde{\pi}$ acting on $\tilde{E}[r]$. Hence we must have $\chi(P) = p^2P$, whence $\Psi(\tilde{\pi}(\tilde{P})) = p^2P$. Applying Ψ^{-1} to both sides gives $\tilde{\pi}(\tilde{P}) = p^2\tilde{P}$, so $\tilde{P} \in \tilde{\mathbb{G}}_2$. We have established that $\Psi^{-1}(P) \in \tilde{\mathbb{G}}_2$, so we conclude that $\Psi(\tilde{\mathbb{G}}_2) = \mathbb{G}_1$. □

Capítulo 8

Conclusões

Apresentamos nos capítulos anteriores vários trabalhos relacionados à implementação eficiente em *software* de algoritmos critográficos. As contribuições foram agrupadas em dois conjuntos de resultados: implementação eficiente de criptossistemas de curvas elípticas e implementação eficiente de criptografia baseada em emparelhamentos. Procurou-se organizar os trabalhos apresentados segundo um fluxo lógico que parte de dispositivos no segmento mais restrito do espectro de arquiteturas computacionais até atingir as arquiteturas multiprocessadas que vêm ditando as tendências tecnológicas recentes.

No Capítulo 2, são apresentadas técnicas para implementação de aritmética em corpos binários em microcontroladores de 8 *bits* que aproveitam ao máximo os recursos da plataforma subjacente em busca de desempenho. A implementação de curvas elípticas no nível de segurança de 80 *bits* sobre estes corpos permite calcular uma multiplicação de ponto aleatório, operação fundamental de protocolos baseados em curvas elípticas, em menos de $\frac{1}{3}$ segundo. Este resultado contraria diversas observações levantadas em trabalhos anteriores sobre a inviabilidade de curvas binárias do ponto de vista de desempenho para dispositivos limitados, e aperfeiçoa o estado-da-arte em pelo menos 57% para multiplicações de ponto curvas elípticas neste nível de segurança. Ainda que inicialmente dedicadas à plataforma alvo, as otimizações desenvolvidas podem ser empregadas em arquiteturas que compartilhem de características como conjunto de instruções limitado e latência proibitiva em operações de acesso à memória, típicas de outras arquiteturas no segmento de microcontroladores. Um obstáculo claro detectado neste trabalho foi a falta de disponibilidade de memória RAM na plataforma alvo para realizar implementações em níveis de segurança mais altos. Em compensação, uma vantagem do cenário de rede de sensores é não exigir proteção contra vazamento de informações por canais laterais, visto que o adversário já possui acesso físico irrestrito aos dispositivos de monitoramento.

No Capítulo 3, o mesmo problema é transportado para arquiteturas *desktop* equipadas com conjuntos de instruções vetoriais. Uma formulação de aritmética em corpos binários explorando explicitamente as novas instruções de permutação de *bytes* é desenvolvida, acelerando a latência de operações em um corpo binário em até 84% e induzindo um novo ainda que ineficiente algoritmo de multiplicação no corpo finito. Esta nova formulação de aritmética é ilustrada com a implementação da multiplicação de ponto em uma curva elíptica no nível de segurança de 128 *bits*, mais adequado para a plataforma-alvo, e acelera em até 30% o estado-da-arte de implementações desta primitiva que não funcionam em modo lote. Devido ao desempenho superior de abordagens baseadas em modo lote, não houve quebra de recordes de velocidade, mas o trabalho permitiu estabelecer novos pontos de referência para o desempenho em modo lote. Seria interessante como trabalho futuro estender a abordagem à corpos finitos de catacterística 3 e avaliar o desempenho do novo algoritmo de multiplicação neste cenário.

No Capítulo 4, é analisado o impacto algorítmico causado pela introdução de suporte nativo à multiplicação em corpos binários nas microarquiteturas mais recentes da Intel. Diversas estratégias de implementação de operações como *half-trace* e inversão permitiram restaurar o desempenho de abordagens para multiplicação de ponto baseadas em *halving*, face à disponibilidade de uma operação de duplicação de ponto significativamente mais eficiente. Os impactos da nova instrução são ainda explorados a partir de extensas implementações seriais e paralelas de curvas elípticas binárias padronizadas nos níveis de segurança de 112 e 192 *bits* e um novo recorde de velocidade é finalmente obtido para multiplicações de ponto em curvas binárias resistentes ao vazamento de informações por meio de canal lateral, com uma modesta aceleração de 10% em relação ao estado-da-arte em modo lote previamente discutido. Outra conseqüência da nova instrução, além da aceleração da implementação, é reduzir o vazamento de informações por canais laterais ligados à hierarquia de memória presentes nos algoritmos que exigem fases de pré-computação. e modo lote precisa ser novamente revisitada após a introdução de instruções vetoriais de 256 *bits* trazidas pelas extensões AVX. É também necessário fornecer um panorama mais completo do desempenho de curvas elípticas no nível de segurança de 128 *bits*, visto que apenas um cenário foi considerado neste trabalho.

As contribuições que lidam com o desempenho de emparelhamentos foram apresentadas a partir do Capítulo 5. Neste capítulo, a implementação eficiente em corpos binários permitiu estabelecer um novo recorde de velocidade para emparelhamentos simétricos, com uma aceleração de até 30% em relação ao estado-da-arte prévio. Além disso, é apresentada uma estratégia escalável de paralelização do Algoritmo de Miller útil para aplicações em que a latência de comunicação é crítica. Esta estratégia é excepcional para o caso simétrico binário e acelera o estado-da-arte em 28%, 44% e 66% em arquiteturas com 2, 4 e 8 unidades de processamento. A aplicação da mesma estratégia para o caso assimétrico fornece um ganho modesto de desempenho de apenas 10% com o emprego de 2 unidades de processamento. Como trabalhos futuros, permanece a necessidade de se revisitar essas estratégias para o caso simétrico definido sobre corpos ternários.

No Capítulo 6, são desenvolvidas novas técnicas para a implementação serial de emparelhamentos primos no nível de segurança de 128 *bits*. As contribuições principais são: (i) a generalização da noção de *redução modular preguiçosa* para aritmética em corpos de extensão e em *twists* de curvas elípticas; (ii) o desenvolvimento de novas fórmulas para o cálculo de quadrados comprimidos sucessivos em subgrupos ciclotômicos de corpos de extensão; (iii) a eliminação de penalidades de desempenho em parametrizações negativas de curvas Barreto-Naehrig. Estas novas técnicas estabelecem um novo recorde de velocidade para o cálculo de qualquer emparelhamento, aprimorando o melhor resultado anterior em até 34%. Como trabalho futuro natural, é importante examinar o impacto dessas novas técnicas na escolha de parâmetros para o cálculo de emparelhamentos em níveis superiores de segurança, onde uma primeira análise sugere curvas com maior grau de mergulho.

No último capítulo de resultados, o Capítulo 7, todas as técnicas desenvolvidas no trabalho são empregadas para o aprimoramento dos resultados obtidos nos dois últimos capítulos. Em primeiro lugar, é mostrado como as fórmulas para o cálculo comprimido de quadrados em subgrupos ciclotômicos aumentam a escalabilidade da estratégia de paralelização apresentada no Capitulo 5 para o caso assimétrico, atingindo até 20% de ganho de desempenho quando estão disponíveis 2 unidades de processamento. Há um estudo mais detalhado da granularidade ideal e dos obstáculos para a paralelização de emparelhamentos assimétricos e são apresentados dois novos emparelhamentos derivados do emparelhamento de Weil, dos quais um deles apresenta escalabilidade superior às outras opções quando há disponibilidade de 8 unidades de processamento. Finalmente, o desempenho de emparelhamentos simétricos sobre curvas elípticas supersingulares binárias é revisitado, considerando principalmente o impacto do novo suporte nativo à multiplicação em corpos binários discutido no Capítulo 4, propondo-se uma simplificação da partição paralela do Algoritmo de Miller que restaura a escalabilidade observada na implementação original.

Dentre os vários recordes de velocidade para implementações em *software* no nível de segurança de 128 *bits* apresentados neste trabalho, nenhum foi superado até a publicação desta tese. São estes: recorde de velocidade para a multiplicação em curvas binárias genéricas resistente a canais laterais e para o cálculo serial ou paralelo de emparelhamentos bilineares nos cenários simétrico e assimétrico. O recorde para o caso assimétrico se aplica também a implementações em *hardware*. Várias das contribuições desta tese estão disponíveis na biblioteca criptográfica RELIC, da qual o autor é fundador e desenvolvedor líder.

Referências Bibliográficas

[1] P. S. L. M. Barreto, S. Galbraith, C. Ó hÉigeartaigh, and M. Scott. Efficient Pairing Computation on Supersingular Abelian Varieties. *Designs, Codes and Cryptography*, 42(3):239–271, 2007.

[2] F. Hess, N. P. Smart, and F. Vercauteren. The Eta Pairing Revisited. *IEEE Transactions on Information Theory*, 52:4595–4602, 2006.

[3] H. Lee E. Lee and C. Park. Efficient and Generalized Pairing Computation on Abelian Varieties. *IEEE Transactions on Information Theory*, 55(4):1793–1803, 2009.

[4] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, IT-22:644–654, November 1976.

[5] L. M. Kohnfelder. Towards a practical public-key cryptosystem. B.S. Thesis, supervised by L. Adleman, May 1978.

[6] P. Gutman. PKI: it's not dead, just resting. *IEEE Computer*, 35(8):41–49, 2002.

[7] N. Koblitz. Elliptic Curve Cryptosystems. *Mathematics of computation*, 48:203–9, 1987.

[8] V. Miller. Uses of Elliptic Curves in Cryptography. In H. C. Williams, editor, *5th Annual International Cryptology Conference (CRYPTO 85)*, volume 218 of *LNCS*, pages 417–426. Springer, 1986.

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

[10] L. B. Oliveira, M. Scott, J. López, and R. Dahab. TinyPBC: Pairings for Authenticated Identity-Based Non-Interactive Key Distribution in Sensor Networks. In *5th International Conference on Networked Sensing Systems (INSS 2008)*, pages 173–179, 2008.

[11] A. Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and D. Chaum, editors, *4th Annual International Cryptology Conference (CRYPTO 84)*, volume 196 of *LNCS*, pages 47–53. Springer, 1984.

[12] D. Boneh and M. K. Franklin. Identity-Based Encryption from the Weil Pairing. In J. Kilian, editor, *21st Annual International Cryptology Conference (CRYPTO 2001)*, volume 2139 of *LNCS*, pages 213–229. Springer, 2001.

[13] R. Sakai, K. Ohgishi, and M. Kasahara. Cryptosystems Based on Pairing over Elliptic Curve (in Japanese). In *The 2001 Symposium on Cryptography and Information Security*, January 2001.

[14] A. Menezes, T. Okamoto, and S. A. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. In *23rd annual ACM Symposium on Theory of Computing (STOC 91)*, pages 80–89, New Yourk, USA, 1992. ACM.

[15] G. Frey and M. M{uller and H. R}uck. The Tate pairing and the discrete logarithm applied to elliptic curve cryptosystems. *IEEE Transtactions on Information Theory*, 45(5):1717–1719, 1999.

[16] A. Joux. A One Round Protocol for Tripartite Diffie-Hellman. *Journal of Cryptology*, 17(4):263–276, 2004.

[17] D. Boneh, B. Lynn, and H. Shacham. Short Signatures from the Weil Pairing. *Journal of Cryptology*, 17(4):297–319, 2004.

[18] Sattam S. Al-Riyami and Kenneth G. Paterson. Certificateless Public Key Cryptography. In C.-S. Laih, editor, *9th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2003)*, volume 2894 of *LNCS*, pages 452–473. Springer, 2003.

[19] M. Scott. Implementing Cryptographic Pairings. In *First International Conference on Pairing-Based Cryptography (Pairing 2007)*, volume 4575 of *LNCS*, pages 177–196. Springer, 2007.

[20] D. F. Aranha, K. Karabina, P. Longa, C. H. Gebotys, and J. López. Faster Explicit Formulas for Computing Pairings over Ordinary Curves. In K. Patterson, editor, *30th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2011)*, volume 6632 of *LNCS*, pages 48–68. Springer, 2010.

[21] P. Longa and C. H. Gebotys. Efficient techniques for high-speed elliptic curve cryptography. In S. Mangard and F.-X. Standaert, editors, *12th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2010)*, volume 6225 of *LNCS*, pages 80–94. Springer, 2010.

[22] Paulo S. L. M. Barreto, Ben Lynn, and Michael Scott. Efficient Implementation of Pairing-Based Cryptosystems. *Journal of Cryptology*, 17(4):321–334, 2004.

[23] Paulo S. L. M. Barreto, H. Y. Kim, B. Lynn, and M. Scott. Efficient Algorithms for Pairing-Based Cryptosystems. In M. Yung, editor, *22th Annual International Cryptology Conference (CRYPTO 2002)*, volume 2442 of *LNCS*, pages 354–368. Springer, 2002.

[24] F. Vercauteren. Optimal pairings. *IEEE Trans. Inf. Theor.*, 56(1):455–461, 2010.

[25] D. Freeman, M. Scott, and E. Teske. A Taxonomy of Pairing-Friendly Elliptic Curves. *Journal of Cryptology*, 23(2):224–280, 2010.

[26] C. Costello, T. Lange, and M. Naehrig. Faster Pairing Computations on Curves with High-Degree Twists. In P. Q. Nguyen and D. Pointcheval, editors, *13th International Conference on Practice and Theory in Public Key Cryptography (PKC 2010)*, volume 6056 of *LNCS*, pages 224–242. Springer, 2010.

[27] J. López and R. Dahab. High-Speed Software Multiplication in GF(2^m). In B. K. Roy and E. Okamoto, editors, *1st International Conference in Cryptology in India (INDOCRYPT 2000)*, volume 1977 of *LNCS*, pages 203–212. Springer, 2000.

[28] A. J. Devegili, C. Ó hÉigeartaigh, M. Scott, and R. Dahab. Multiplication and Squaring on Pairing-Friendly Fields. Cryptology ePrint Archive, Report 2006/471, 2006. `http://eprint.iacr.org/`.

[29] OpenMP Architecture Review Board. OpenMP Application Program Interface. `http://www.openmp.org/drupal/mp-documents/spec25.pdf`, 2005.

[30] O. Wechsler. Inside Intel Core Microarchitecture: Setting new Standards for Energy-efficient Performance. *Technology@Intel Magazine*, 2006.

[31] Advanced Micro Devices. Software optimization guide for amd family 10h and 12h processors. Technical Report, 2006. `http://developer.amd.com/`.

[32] Atmel Corporation. *8 bit AVR Microcontroller ATmega128(L) manual.* Atmel, 2467M-AVR-11/04 edition, November 2004.

[33] Intel. The Intel XScale Microarchitecture Technical Summary. `http://www.intel.com`.

[34] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[35] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, and A. Lefohn amd Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[36] D. W. Wall. Limits of Instruction-Level Parallelism. In *4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS 91)*, volume 26, pages 176–189, New York, NY, 1991. ACM.

[37] V. V. and M. Franz. Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37(3):195–237, 2005.

[38] Wm. A. Wulf and S. A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[39] S. Akhter and J. Roberts. *Multi-Core Programming.* Intel Press, 2006.

[40] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, 2005.

[41] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, 2006.

[42] C. Hughes and T. Hughes. *Professional Multicore Programming: Design and Implementation for C++ Developers.* Wrox Press Ltd., Birmingham, UK, UK, 2008.

[43] C. N. Keltcher, K. J. Mcgrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, 2003.

[44] R. Hetheringtonh. The UltraSPARC T1 Processor: Power-Efficient Throughput Computing. Sun White Paper, December 2005.

[45] K. Hirata and J. Goodacre. ARM MPCore; The streamlined and scalable ARM11 processor core. In *ASP-DAC*, pages 747–748, 2007.

[46] MIPS Technologies. MIPS32 1004K Coherent Processing System Core. The MIPS32 1004K Product Brief, 2008.

[47] F. Zhao N. B. Priyantha and D. Lymberopolous. mPlatform: A reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes. Tech Report, 2006. `http://research.microsoft.com/pubs/70353/tr-2006-142.pdf`.

[48] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, September 1972.

[49] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[50] Intel. Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference. `http://www.intel.com`, 2002.

[51] Intel. Intel SSE4 Programming Reference. Technical Report. `http://software.intel.com/`.

[52] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo. Intel AVX: New frontiers in performance improvement and energy efficiency. White paper. `http://software.intel.com/`.

[53] F. Mattern and C. Floerkemeier. *From the Internet of Computers to the Internet of Things*, volume 6462 of *LNCS*, pages 242–259. Springer, 2010.

[54] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In K. G. Shin, Y. Zhang, R. Bagrodia, and R. Govindan, editors, *15th Annual International Conference on Mobile Computing and Networking (MOBICOM 99)*, pages 263–270, Seattle, WA USA, 1999. ACM.

[55] Jeremey Landt. Shrouds of Time: The History of RFID. Technical report, The Association for Automatic Identification and Data Capture Technologies, 2001.

[56] Intel. Intel Architecture Software Developer's Manual Volume 3: System Programming Guide. `http://www.intel.com`, 2006.

[57] V. Miller. Short programs for functions on curves. Unpublished manuscript, 1986.

[58] V. S. Miller. The Weil Pairing, and Its Efficient Calculation. *Journal of Cryptology*, 17(4):235–261, 2004.

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

[60] J. López and R. Dahab. Improved Algorithms for Elliptic Curve Arithmetic in GF(2^n). In S. E. Tavares and H. Meijer, editors, *5th Annual International Workshop Selected Areas in Cryptography (SAC 98)*, volume 1556 of *LNCS*, pages 201–212. Springer, 1998.

[61] D. J. Bernstein. Batch Binary Edwards. In S. Halevi, editor, *29th Annual International Cryptology Conference (CRYPTO 2009)*, volume 5677 of *LNCS*, pages 317–336. Springer, 2009.

[62] K. Fong, D. Hankerson, J. López, and A. Menezes. Field Inversion and Point Halving Revisited. *IEEE Transactions on Computers*, 53(8):1047–1059, 2004.

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

[64] C. H. Lim and P. J. Lee. More Flexible Exponentiation with Precomputation. In Y. Desmedt, editor, *14th Annual International Cryptology Conference (CRYPTO 1994)*, volume 839 of *LNCS*, pages 95–107, London, UK, 1994. Springer.

[65] R. Gallant, R. Lambert, and S. Vanstone. Faster Point Multiplication on Elliptic Curves with Efficient Endomorphisms. In J. Kilian, editor, *21st Annual International Cryptology Conference (CRYPTO 2001)*, volume 2139 of *LNCS*, pages 190–200. Springer, 2001.

[66] H. Cohen and G. Frey, editors. *Handbook of Elliptic and Hyperelliptic Curve Cryptography.* CRC Press, 2005.

[67] D. Hankerson, A. Menezes, and M. Scott. Software Implementation of Pairings. In *Identity-Based Cryptography*, chapter 12, pages 188–206. IOS Press, 2008.

[68] D. M. Gordon. Discrete logarithms in GF(p) using the number field sieve. *SIAM Journal on Discrete Mathematics*, 6(1):124–138, 1993.

[69] L. M. Adleman. A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography (Abstract). In *Foundations of Computer Science (FOCS 79)*, pages 55–60. IEEE, 1979.

[70] J. H. Silverman and J. Suzuki. Elliptic curve discrete logarithms and the index calculus. In K. Ohta and D. Pei, editors, *International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT 98)*, volume 1514 of *LNCS*, pages 110–125, London, UK, 1998. Springer.

[71] S.D. Galbraith, K.G. Paterson, and N.P. Smart. Pairings for Cryptographers. *Discrete Applied Mathematics*, 156(16), 2008.

[72] K. E. Stange. The Tate Pairing via Elliptic Nets. In T. Takagi, T. Okamoto, E. Okamoto, and T. Okamoto, editors, *First International Conference on Pairing-Based Cryptography (Pairing 2007)*, volume 4575 of *LNCS*, pages 329–348. Springer, 2007.

[73] A. Miyaji, M. Nakabayashi, and S. Takano. New Explicit Conditions of Elliptic Curve Traces for FR-Reduction. *Transactions on Comm./Elec./Information and Systems*, 2001.

[74] P. S. L. M. Barreto and M. Naehrig. Pairing-Friendly Elliptic Curves of Prime Order. In B. Preneel and S. E. Tavares, editors, *12th International Workshop on Selected Areas in Cryptography (SAC 2005)*, volume 3897 of *LNCS*, pages 319–331. Springer, 2005.

[75] E. J. Kachisa, E. F. Schaefer, and M. Scott. Constructing Brezing-Weng pairing friendly elliptic curves using elements in the cyclotomic field. In S. D. Galbraith and K. G. Paterson, editors, *2nd International Conference on Pairing-Based Cryptography (Pairing 2008)*, volume 5209 of *LNCS*, pages 126–135. Springer, 2008.

[76] M. Maas. Pairing-based Cryptography. Master's thesis, Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, 2004.

[77] C.-M. Park, M.-H. Kim, and M. Yung. A remark on implementing the weil pairing. In D. Feng, D. Lin, and M. Yung, editors, *1st SKLOIS Conference on Information Security and Cryptology*, volume 3822 of *LNCS*, pages 313–323. Springer, 2005.

[78] I. M. Duursma and H. S. Lee. Tate Pairing Implementation for Hyperelliptic Curves $\mathrm{y}^2 = \mathrm{x}^\mathrm{p}$-x + d. In C.-S. Laih, editor, *9th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT 2003)*, volume 2894 of *LNCS*, pages 111–123, 2003.

[79] J. Beuchat, N. Brisebarre, J. Detrey, E. Okamoto, and F. Rodríguez-Henríquez. A Comparison Between Hardware Accelerators for the Modified Tate Pairing over $\mathbb{F}_{2^m}$ and $\mathbb{F}_{3^m}$. In S. D. Galbraith and K. G. Paterson, editors, *2nd International Conference on Pairing-Based Cryptography (Pairing 2008)*, volume 5209 of *LNCS*, pages 297–315. Springer, 2008.

[80] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. *Wireless Networks*, 8(5):521–534, September 2002.

[81] C. Karlof, N. Sastry, and D. Wagner. TinySec: A Link Layer Security Architecture for Wireless Sensor Networks. In *2nd ACM SenSys*, pages 162–175, Nov 2004.

[82] N. Gura, A. Patel, A. Wander, H. Eberle, and S. C. Shantz. Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In M. Joye and J.-J. Quisquater, editors, *6th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2004)*, volume 3156 of *LNCS*, pages 119–132. Springer, 2004.

[83] H. Eberle, A. Wander, N. Gura, S. Chang-Shantz, and V. Gupta. Architectural Extensions for Elliptic Curve Cryptography over GF(2^m) on 8-bit Microprocessors. In *16th IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP 2005)*, pages 343–349. IEEE, 2005.

[84] D. F. Aranha, D. Câmara, J. López, L. B. Oliveira, and R. Dahab. Implementação eficiente de criptografia de curvas elípticas em sensores sem fio. In *VIII Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2008)*, pages 173–186, 2008.

[85] D. F. Aranha, J. López, L. B. Oliveira, and R. Dahab. Efficient implementation of elliptic curves on sensor nodes. In *Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc (CHiLE 2009)*, 2009.

[86] D. F. Aranha, L. B. Oliveira, J. López, and R. Dahab. Efficient implementation of elliptic curve cryptography in wireless sensors. *Advances in Mathematics of Communications*, 4(2):169–187, 2010.

[87] S. Gueron and M. E. Kounavis. Carry-Less Multiplication and Its Usage for Computing The GCM Mode. White paper. `http://software.intel.com/`.

[88] D. F. Aranha, J. López, and D. Hankerson. Efficient Software Implementation of Binary Field Arithmetic Using Vector Instruction Sets. In M. Abdalla and P. S. L. M. Barreto, editors, *1st International Conference on Cryptology and Information Security (LATINCRYPT 2010)*, volume 6212 of *LNCS*, pages 144–161, 2010.

[89] J. Taverne, A. Faz-Hernández, D. F. Aranha, F. Rodríguez-Henríquez, D. Hankerson, and J. López. Software implementation of binary elliptic curves: Impact of the carry-less multiplier on scalar multiplication. In *13th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2011)*, 2011. To appear.

[90] D. J. Bernstein and T. Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `http://bench.cr.yp.to`, accessed 25 May 2010.

[91] D. F. Aranha, L. B. Oliveira, J. López, and R. Dahab. NanoPBC: implementing cryptographic pairings on an 8-bit platform. In *Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc (CHiLE 2009)*, 2009.

[92] L. B. Oliveira, D. F. Aranha, C. P. L. Gouvêa, M. Scott, D. F. Câmara, J. López, and R. Dahab. TinyPBC: Pairings for Authenticated Identity-Based Non-Interactive Key Distribution in Sensor Networks. *Computer Communications*, 4(2):169–187, 2010.

[93] L. B. Oliveira, A. Kansal, C. P. L. Gouvêa, D. F. Aranha, J. López, B. Priyantha, M. Goraczko, and F. Zhao. Secure-tws: Authenticating node to multi-user communication in shared sensor networks. *The Computer Journal*, 2011. To appear.

[94] C.-P. Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO '89*, volume 435 of *LNCS*, pages 239–252. Springer, 1989.

[95] Fangguo Zhang, Reihaneh Safavi-Naini, and Willy Susilo. An efficient signature scheme from bilinear pairings and its applications. In Feng Bao, Robert H. Deng, and Jianying Zhou, editors, *Public Key Cryptography (PKC 2004)*, volume 2947 of *LNCS*, pages 277–290. Springer, 2004.

[96] P. Grabher, J. Groszschaedl, and D. Page. On Software Parallel Implementation of Cryptographic Pairings. In R. M. Avanzi, L. K., and F. Sica, editors, *15th International Workshop on Selected Areas in Cryptography (SAC 2008)*, volume 5381 of *LNCS*, pages 34–49. Springer, 2008.

[97] P. S. L. M. Barreto, M. Naehrig, G. C. C. F. Pereira, and M. A. Simplício Jr. A Family of Implementation-Friendly BN Elliptic Curves. Cryptology ePrint Archive, Report 2010/429, 2010. `http://eprint.iacr.org/`.

[98] J. Beuchat, E. López-Trejo, L. Martínez-Ramos, S. Mitsunari, and F. Rodríguez-Henríquez. Multi-core Implementation of the Tate Pairing over Supersingular Elliptic Curves. In J. A. Garay, A. Miyaji, and A. Otsuka, editors, *8th International Conference on Cryptology and Network Security (CANS 2009)*, volume 5888 of *LNCS*, pages 413–432. Springer, 2009.

[99] J.-L. Beuchat, J.E. González Díaz, S. Mitsunari, E. Okamoto, F. Rodríguez-Henríquez, and T. Teruya. High-speed software implementation of the optimal ate pairing over Barreto–Naehrig curves. In M. Joye, A. Miyaji, and A. Otsuka, editors, *4th International Conference on Pairing-Based Cryptography (Pairing 2010)*, number 6487 in LNCS, pages 21–39. Springer, 2010.

[100] D. F. Aranha and J. López. Paralelização em software do Algoritmo de Miller. In *IX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSEG 2009)*, pages 27–40, 2009.

[101] D. F. Aranha, J. López, and D. Hankerson. High-Speed Parallel Software Implementation of the η_T Pairing. In *Software Performance Enhancement of Encryption and Decryption and Cryptographic Compilers (SPEED-CC 2009)*, pages 73–88, 2009. `http://www.hyperelliptic.org/SPEED/record09.pdf`.

[102] D. F. Aranha, J. López, and D. Hankerson. High-Speed Parallel Software Implementation of the η_T Pairing. In J. Pieprzyk, editor, *Cryptographers' Track at RSA Conference (CT-RSA 2010)*, volume 5985 of *LNCS*, pages 89–105. Springer, 2010.

[103] Diego F. Aranha, Jean-Luc Beuchat, Jérémie Detrey, and Nicolas Estibals. Optimal eta pairing on supersingular genus-2 binary hyperelliptic curves. Cryptology ePrint Archive, Report 2010/559, 2010. `http://eprint.iacr.org/`.

[104] D. F. Aranha, F. Rodríguez-Henríquez, E. Knapp, and A. Menezes. Parallelizing the Weil and Tate Pairings. To appear in Proceedings of the 13th IMA International Conference on Cryptography and Coding (IMA-CC 2011).

[105] D. F. Aranha and C. P. L. Gouvêa. RELIC is an Efficient LIbrary for Cryptography. `http://code.google.com/p/relic-toolkit/`.

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

[107] A. Kargl, S. Pyka, and H. Seuschek. Fast Arithmetic on ATmega128 for Elliptic Curve Cryptography. Cryptology ePrint Archive, Report 2008/442, 2008. `http://eprint.iacr.org/`.

[108] J. Großschädl. TinySA: a security architecture for wireless sensor networks. In *Conference on Emerging Network Experiment and Technology (CoNEXT 2006)*, page 55. ACM, 2006.

[109] L. B. Oliveira, D. F. Aranha, E. Morais, F. Daguano, J. López, and R. Dahab. TinyTate: Computing the Tate Pairing in Resource-Constrained Sensor Nodes. In *IEEE International Symposium on Network Computing and Applications (NCA 2007)*, pages 318–323. IEEE, 2007.

[110] Certicom Research. SEC 1: Elliptic Curve Cryptography. `http://www.secg.org`, 2000.

[111] L. Uhsadel, A. Poschmann, and C. Paar. Enabling Full-Size Public-Key Algorithms on 8-Bit Sensor Nodes. In F. Stajano, C. Meadows, S. Capkun, and T. Moore, editors, *4th European Workshop on Security and Privacy in Ad-hoc and Sensor Networks (ESAS 2007)*, volume 4572 of *LNCS*, pages 73–86. Springer, 2007.

[112] D. J. Malan, M. Welsh, and M. D. Smith. A Public-Key Infrastructure for Key Distribution in TinyOS Based on Elliptic Curve Cryptography. In *Proceedings of SECON'04*, Santa Clara, California, October 2004.

[113] H. Yan and Z. J. Shi. Studying Software Implementations of Elliptic Curve Cryptography. In *3rd International Conference on Information Technology: New Generations (ITNG 2006)*, pages 78–83, Washington, USA, 2006. IEEE.

[114] P. Szczechowiak, L. B. Oliveira, M. Scott, M. Collier, and R. Dahab. NanoECC: Testing the Limits of Elliptic Curve Cryptography in Sensor Networks. In R. Verdone, editor, *European Conference on Wireless Sensor Networks (EWSN'08)*, volume 4193 of *LNCS*, pages 305–320. Springer, 2008.

[115] M. Scott. MIRACL – Multiprecision Integer and Rational Arithmetic C/C++ Library. `http://www.shamus.ie/`.

[116] S. C. Seo, D. Han, and S. Hong. TinyECCK: Efficient Elliptic Curve Cryptography Implementation over G(2) on 8-Bit Micaz Mote. *IEICE Transactions*, 91-D(5):1338–1347, 2008.

[117] H. Wang and Q. Li. Efficient Implementation of Public Key Cryptosystems on Mote Sensors. In P. Ning, S. Qing, and N. Li, editors, *8th International Conference on Information and Communications Security (ICICS 2006)*, volume 4307 of *LNCS*, pages 519–528, Raleigh, NC, 2006. Springer.

[118] G.-J. Lay and H. G. Zimmer. Constructing elliptic curves with given group order over large finite fields. In L. M. Adleman and M.-D. A. Huang, editors, *1st International Symposium on Algorithmic Number Theory (ANTS-I)*, volume 877 of *LNCS*, pages 250–263. Springer, 1994.

[119] B. Skjernaa T. Satoh and Y. Taguchi. Fast computation of canonical lifts of elliptic curves and its application to point counting. *Finite Fields Appl.*, 9:89–101, 2003.

[120] A. Menezes, T. Okamoto, and S. Vanstone. Reducing elliptic curve logarithms to logarithms in a finite field. *IEEE Trans. Inform. Theory*, 39:1639–1646, 1993.

[121] A. Karatsuba and Y. Ofman. Multiplication of Many-Digital Numbers by Automatic Computers. *Doklady Akad. Nauk SSSR*, (145):293–294, 1962. Translation in Physics-Doklady 7, 595-596, 1963.

[122] J. López and R. Dahab. Fast Multiplication on Elliptic Curves over GF(2^m) without Precomputation. In Ç. K. Koç and C. Paar, editors, *1st International Workshop on Cryptographic Hardware and Embedded Systems (CHES 1999)*, volume 1717 of *LNCS*, pages 316–327. Springer, 1999.

[123] Atmel Corporation. AVR Studio 4.14. `http://www.atmel.com/`, 2005.

[124] L. B. Oliveira, A. Kansal, B. Priyantha, M. Goraczko, and Zhao F. Secure-TWS: Authenticating Node to Multi-user Communication in Shared Sensor Networks. In *The 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*. ACM, 2009.

[125] N. Koblitz. CM-Curves with Good Cryptographic Properties. In *11th Annual International Cryptology Conference (CRYPTO 1991)*, volume 576 of *LNCS*, pages 279–287. Springer, 1991.

[126] J. Daemen and V. Rijmen. *The Design of Rijndael*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[127] R. Misoczki and P. S. L. M. Barreto. Compact McEliece Keys from Goppa Codes. In M. J. Jacobson Jr., V. Rijmen, and R. Safavi-Naini, editors, *16th Annual International Workshop on Selected Areas in Cryptography (SAC 2009)*, volume 5867 of *LNCS*, pages 376–392. Springer, 2009.

[128] A. I.-T. Chen, M.-S. Chen, T.-R. Chen, C.-M. Cheng, J. Ding, E. L.-H. Kuo, F. Y.-S. Lee, and B.-Y. Yang. SSE Implementation of Multivariate PKCs on Modern x86 CPUs. In C. Clavier and K. Gaj, editors, *11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009)*, volume 5747 of *LNCS*, pages 33–48. Springer, 2009.

[129] A. M. Fiskiran and R. B. Lee. Fast Parallel Table Lookups to Accelerate Symmetric-Key Cryptography. In *International Symposium on Information Technology: Coding and Computing (ITCC 2005)*, volume 1, pages 526–531. IEEE, 2005.

[130] D. Hankerson, J. López, and A. Menezes. Software Implementation of Elliptic Curve Cryptography over Binary Fields. In Ç. K. Koç and C. Paar, editors, *2nd International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2000)*, volume 1965 of *LNCS*, pages 1–24. Springer, 2000.

[131] Y. Hilewitz, Y. L. Yin, and B. Lee R. Accelerating the Whirlpool Hash Function Using Parallel Table Lookup and Fast Cyclical Permutation. In K. Nyberg, editor, *15th International Workshop on Fast Software Encryption (FSE 2005)*, volume 5086 of *LNCS*, pages 173–188, 2008.

[132] K. Diefendorff, P. K. Dubey, R. H., and H. Scales. AltiVec Extension to PowerPC Accelerates Media Processing. *IEEE Micro*, 20(2):85–95, 2000.

[133] AMD Technology. AMD64 Architecture Programmer's Manual Volume 6: 128-bit and 256-bit XOP, FMA4 and CVT16 Instruction. `http://support.amd.com/us/Processor_TechDocs/43479.pdf`.

[134] R. M. Avanzi. Another Look at Square Roots (and Other Less Common Operations) in Fields of Even Characteristic. In C. M. Adams, A. Miri, and M. J. Wiener, editors, *14th International Workshop on Selected Areas in Cryptography (SAC 2007)*, volume 4876 of *LNCS*, pages 138–154. Springer, 2007.

[135] Certicom Research. SEC 2: Recommended Elliptic Curve Domain Parameters. `http://www.secg.org`, 2000.

[136] D. Hankerson, K. Karabina, and A. Menezes. Analyzing the Galbraith-Lin-Scott Point Multiplication Method for Elliptic Curves over Binary Fields. *IEEE Transactions on Computers*, 58(10):1411–1420, 2009.

[137] P. Gaudry and E. Thomé. The mpFq library and implementing curve-based key exchanges. In *Software Performance Enhancement of Encryption and Decryption (SPEED 2007)*, pages 49–64, 2009. `http://www.hyperelliptic.org/SPEED/record.pdf`.

[138] S. Gueron. Intel Advanced Encryption Standard (AES) Instructions Set. White paper. `http://software.intel.com/`.

[139] D.J. Bernstein and T. Lange. Analysis and optimization of elliptic-curve single-scalar multiplication. In *Proceedings 8th International Conference on Finite Fields and Applications (Fq8)*, volume 461, pages 1–20. AMS, 2008.

[140] A. Fog. Instruction tables: List of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs. `http://www.agner.org/optimize/instruction_tables.pdf`, accessed 01 Mar 2011.

[141] P. G. Comba. Exponentiation Cryptosystems on the IBM PC. *IBM Systems Journal*, 29(4):526–538, 1990.

[142] M. Bodrato. Towards optimal Toom-Cook multiplication for univariate and multivariate polynomials in characteristic 2 and 0. In Claude Carlet and Berk Sunar, editors, *1st International Workshop on the Arithmetic of Finite Fields (WAIFI 2007)*, volume 4547 of *LNCS*, pages 116–133. Springer, 2007.

[143] P.L. Montgomery. Five, six, and seven-term Karatsuba-like formulae. *IEEE Transactions on Computers*, 54(3):362–369, 2005.

[144] Omran Ahmadi, Darrel Hankerson, and Francisco Rodríguez-Henríquez. Parallel formulations of scalar multiplication on Koblitz curves. *Journal of Universal Computer Science*, 14(3):481–504, 2008.

[145] J. W. Bos, T. Kleinjung, R. Niederhagen, and P. Schwabe. Ecc2k-130 on cell cpus. In J. Bernstein D and T. Lange, editors, *3rd International Conference on Cryptology in Africa (AFRICACRYPT 2010)*, volume 6055 of *LNCS*, pages 225–242. Springer, 2010.

[146] Toshiya Itoh and Shigeo Tsujii. A fast algorithm for computing multiplicative inverses in GF(2^m) using normal bases. *Inf. Comput.*, 78(3):171–177, 1988.

[147] J. Guajardo and C. Paar. Itoh-Tsujii inversion in standard basis and its application in cryptography and codes. *Designs, Codes and Cryptography*, 25(2):207–216, 2002.

[148] E. Dahmen, K. Okeya, and D. Schepers. Affine precomputation with sole inversion in elliptic curve cryptography. In J. Pieprzyk, H. Ghodosi, and E. Dawson, editors, *12th Australasian Conference on Information Security and Privacy (ACISP 2007)*, volume 4586 of *LNCS*, pages 245–258. Springer, 2007.

[149] National Institute of Standards and Technology. Recommended elliptic curves for federal government use. NIST Special Publication, 1999. `http://csrc.nist.gov/csrc/fedstandards.html`.

[150] K. H. Kim and S. I. Kim. A new method for speeding up arithmetic on elliptic curves over binary fields. Cryptology ePrint Archive, Report 2007/181, 2007. `http://eprint.iacr.org/`.

[151] E. W. Knudsen. Elliptic scalar multiplication using point halving. In K.-Y. Lam, E. Okamoto, and C. Xing, editors, *International Conference on the Theory and Applications of Cryptology and Information Security (ASIACRYPT 1999)*, volume 1716 of *LNCS*, pages 135–149. Springer, 1999.

[152] R. Schroeppel. Elliptic curves: Twice as fast! Presentation at the 20th Annual International Cryptology Conference (CRYPTO 2000) Rump Session, 2000.

[153] B. King and B. Rubin. Improvements to the point halving algorithm. In H. Wang, J. Pieprzyk, and V. Varadharajan, editors, *9th Australasian Conference on Information Security and Privacy (ACISP 2004)*, volume 3108 of *LNCS*, pages 262–276. Springer, 2004.

[154] I. F. Blake, V. K. Murty, and G. Xu. A note on window τ-NAF algorithm. *Inf. Process. Lett.*, 95(5):496–502, 2005.

[155] K. Järvinen. Optimized FPGA-based elliptic curve cryptography processor for high-speed applications. *Integration, the VLSI Journal*, 44(1):12–21, 2010.

[156] S. Mitsunari. A Fast Implementation of η_T Pairing in Characteristic Three on Intel Core 2 Duo Processor. Cryptology ePrint Archive, Report 2009/032, 2009. `http://eprint.iacr.org/`.

[157] E. Cesena. Pairing with Supersingular Trace Zero Varieties Revisited. Cryptology ePrint Archive, Report 2008/404, 2008. `http://eprint.iacr.org/`.

[158] E. Cesena and R. Avanzi. Trace Zero Varieties in Pairing-based Cryptography. In *Conference on Hyperelliptic curves, discrete Logarithms, Encryption, etc (CHiLE 2009)*, 2009. `http://inst-mat.utalca.cl/chile2009/Slides/Roberto_Avanzi_2.pdf`.

[159] J. Groth and A. Sahai. Efficient Non-interactive Proof Systems for Bilinear Groups. In N. P. Smart, editor, *27th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2008)*, volume 4965 of *LNCS*. Springer, 2008.

[160] M. Naehrig, R. Niederhagen, and P. Schwabe. New Software Speed Records for Cryptographic Pairings. In M. Abdalla and P. S. L. M. Barreto, editors, *First International Conference on Cryptology and Information Security in Latin America (LATINCRYPT 2010)*, volume 6212 of *LNCS*, pages 109–123. Springer, 2010.

[161] J. Fan, F. Vercauteren, and I. Verbauwhede. Faster F_p-arithmetic for Cryptographic Pairings on Barreto-Naehrig Curves. In *11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009)*, volume 5747 of *LNCS*, pages 240–253. Springer, 2009.

[162] Y. Nogami, M. Akane, Y. Sakemi, H. Kato, and Y. Morikawa. Integer Variable χ-Based Ate Pairing. In S. D. Galbraith and K. G. Paterson, editors, *2nd International Conference on Pairing-Based Cryptography (Pairing 2008)*, volume 5209 of *LNCS*, pages 178–191. Springer, 2008.

[163] IEEE. P1363.3: Standard for Identity-Based Cryptographic Techniques using Pairings. Draft.

[164] A. J. Devegili, M. Scott, and R. Dahab. Implementing Cryptographic Pairings over Barreto-Naehrig Curves. In T. Takagi, T. Okamoto, E. Okamoto, and T. Okamoto, editors, *First International Conference on Pairing-Based Cryptography (Pairing 2007)*, volume 4575 of *LNCS*, pages 197–207. Springer, 2007.

[165] D. Weber and T. F. Denny. The solution of mccurley's discrete log challenge. In Hugo Krawczyk, editor, *18th Annual International Cryptology Conference (CRYPTO '98)*, volume 1462 of *LNCS*, pages 458–471. Springer, 1998.

[166] C. H. Lim and H. S. Hwang. Fast implementation of elliptic curve arithmetic in GF(p^n). In H. Imai and Y. Zheng, editors, *Third International Workshop on Practice and Theory in Public Key Cryptography (PKC 2000)*, volume 1751 of *LNCS*, pages 405–421. Springer, 2000.

[167] Roberto Maria Avanzi. Aspects of hyperelliptic curves over large prime fields in software implementations. In Marc Joye and Jean-Jacques Quisquater, editors, *6th International Workshop on Cryptographic Hardware and Embedded Systems - CHES 2004*, volume 3156 of *LNCS*, pages 148–162. Springer, 2004.

[168] N. Benger and M. Scott. Constructing Tower Extensions of Finite Fields for Implementation of Pairing-Based Cryptography. In M. A. Hasan and T. Helleseth, editors, *Third International Workshop on Arithmetic of Finite Fields (WAIFI 2010)*, volume 6087 of *LNCS*. Springer, 2010.

[169] Peter L. Montgomery. Modular Multiplication Without Trial Division. *Mathematics of Computation*, 44(170):pp. 519–521, 1985.

[170] J. Chung and M. . Hasan. Asymmetric Squaring Formulae. In *18th IEEE Symposium on Computer Arithmetic (ARITH-18 2007)*, pages 113–122, 2007.

[171] Diego F. Aranha, Koray Karabina, Patrick Longa, Catherine H. Gebotys, and Julio López. Faster explicit formulas for computing pairings over ordinary curves. Cryptology ePrint Archive, Report 2010/526, 2010. `http://eprint.iacr.org/`.

[172] M. Scott, N. Benger, M. Charlemagne, L. J. Dominguez Perez, and E. J. Kachisa. On the Final Exponentiation for Calculating Pairings on Ordinary Elliptic Curves. In H. Shacham and B. Waters, editors, *3rd International Conference on Pairing-Based Cryptography (Pairing 2009)*, volume 5671 of *LNCS*, pages 78–88. Springer, 2009.

[173] Koray Karabina. Squaring in cyclotomic subgroups. Cryptology ePrint Archive, Report 2010/542, 2010. `http://eprint.iacr.org/`.

[174] P. Montgomery. Speeding the Pollard and Elliptic Curve Methods of Factorization. *Mathematics of Computation*, 48:243–264, 1987.

[175] R. Granger and M. Scott. Faster Squaring in the Cyclotomic Subgroup of Sixth Degree Extensions. In P. Q. Nguyen and D. Pointcheval, editors, *13th International Conference on Practice and Theory in Public Key Cryptography (PKC 2010)*, volume 6056 of *LNCS*, pages 209–223. Springer, 2010.

[176] P. Longa. Speed Benchmarks for Pairings over Ordinary Curves. Available at `http://www.patricklonga.bravehost.com/speed_pairing.html#speed`.

[177] N. Koblitz and A. Menezes. Pairing-based cryptography at high security levels. In Nigel P. Smart, editor, *10th IMA International Conference on Cryptography and Coding (IMA-CC 2005)*, volume 3796 of *LNCS*. Springer, 2005.

[178] F. Hess. Pairing lattices. In S. D. Galbraith and K. G. Paterson, editors, *2nd International Conference on Pairing-Based Cryptography (Pairing 2008)*, volume 5209 of *LNCS*, pages 18–38. Springer, 2008.

[179] C. Zhao, F. Zhang, and D. Xie. Reducing the complexity of the weil pairing computation. Cryptology ePrint Archive, Report 2008/121, 2008. `http://eprint.iacr.org/`.

[180] Dan Boneh and Xavier Boyen. Short signatures without random oracles and the SDH assumption in bilinear groups. *Journal of Cryptology*, 21(2):149–177, 2008.

[181] M. Scott. Authenticated ID-based key exchange and remote log-in with simple token and PIN number. Cryptology ePrint Archive: Report 2002/164, 2002. `http://eprint.iacr.org/`.

[182] N. Estibals. Compact hardware for computing the Tate pairing over 128-bit-security supersingular curves. In M. Joye, A. Miyaji, and A. Otsuka, editors, *4th International Conference on Pairing-Based Cryptography (Pairing 2010)*, number 6487 in LNCS, pages 397–416. Springer, 2010.

[183] T. Güneysu. Utilizing hard cores for modern fpga devices for high-performance cryptography. *Journal of Cryptographic Engineering*, 1:37–55, 2011.

[184] S. Duquesne and N. Guillermin. A fpga pairing implementation using the residue number system. Cryptology ePrint Archive, Report 2011/176, 2011. `http://eprint.iacr.org/`.

[185] G. Yao, J. Fan, R. Cheung, and I. Verbauwhede. A high speed pairing coprocessor using rns and lazy reduction. Cryptology ePrint Archive: Report 2011/258, 2011. `http://eprint.iacr.org/`.

[186] D. Kammler, D. Zhang, P. Schwabe, H. Scharwaechter, M. Langenberg, D. Auras, G. Ascheid, and R. Mathar. Designing an ASIP for cryptographic pairings over Barreto–Naehrig curves. In C. Clavier and K. Gaj, editors, *11th International Workshop on Cryptographic Hardware and Embedded Systems (CHES 2009)*, number 5747 in LNCS, pages 254–271. Springer, 2009.

[187] P. S. L. M. Barreto, B. Lynn, and M. Scott. On the selection of pairing-friendly groups. In M. Matsui and R. J. Zuccherato, editors, *10th International Workshop on Selected Areas in Cryptography (SAC 2003)*, volume 3006 of *LNCS*, pages 17–25, 2003.

[188] J. von zur Gathen. Efficient and optimal exponentiation in finite fields. *Computational Complexity*, 1:360–394, 1991.

[189] OpenMP Microbenchmarks v2.0.

[190] R. Granger, F. Hess, R. Oyono, N. Thériault, and F. Vercauteren. Ate pairing on hyperelliptic curves. In M. Naor, editor, *26th International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2007)*, number 4515 in Lecture Notes in Computer Science, pages 430–447. Springer, 2007.

[191] Y. Nogami M. Akane and Y. Morikawa. Fast ate pairing computation of embedding degree 12 using subfield-twisted elliptic curve. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E92.A:508–516, 2009.