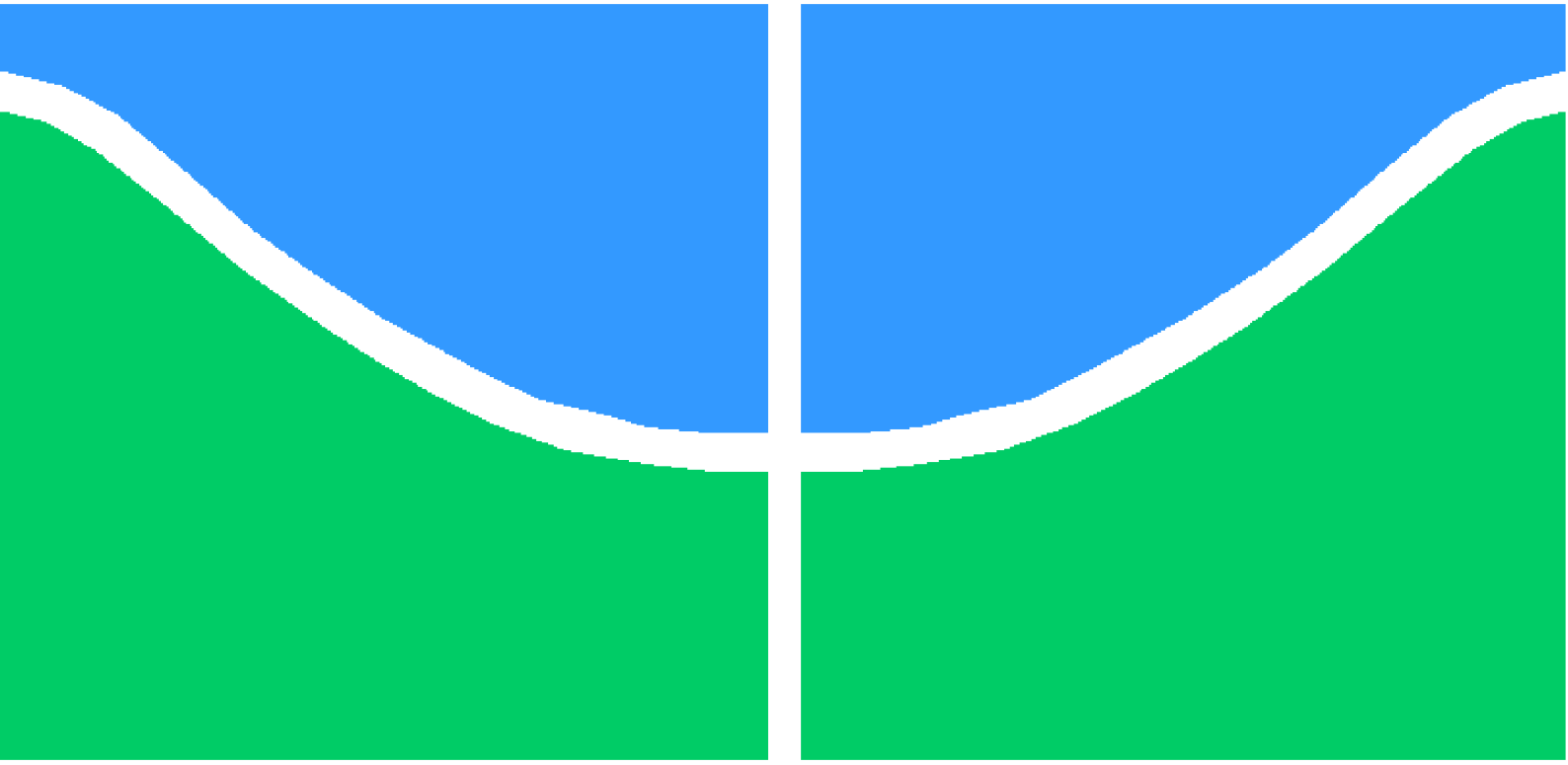

Universidade de Brasília - UnB
Faculdade UnB Gama - FGA
Engenharia de Software

Geração de Parâmetros de Domínio de Curvas Elípticas para Uso em Criptografia

Autor: Marcos da Silva Ramos
Orientador: Prof. Dr. Edson Alves Costa Júnior

Brasília, DF
2015

Marcos da Silva Ramos

Geração de Parâmetros de Domínio de Curvas Elípticas para Uso em Criptografia

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Universidade de Brasília - UnB

Faculdade UnB Gama - FGA

Orientador: Prof. Dr. Edson Alves Costa Júnior

Brasília, DF

2015

Marcos da Silva Ramos

Geração de Parâmetros de Domínio de Curvas Elípticas para Uso em Criptografia/ Marcos da Silva Ramos. – Brasília, DF, 2015-

85 p. : il. (algumas color.) ; 30 cm.

Orientador: Prof. Dr. Edson Alves Costa Júnior

Trabalho de Conclusão de Curso – Universidade de Brasília - UnB
Faculdade UnB Gama - FGA , 2015.

1. curva elíptica. 2. criptografia. I. Prof. Dr. Edson Alves Costa Júnior. II. Universidade de Brasília. III. Faculdade UnB Gama. IV. Geração de Parâmetros de Domínio de Curvas Elípticas para Uso em Criptografia

CDU 02:141:005.6

Marcos da Silva Ramos

Geração de Parâmetros de Domínio de Curvas Elípticas para Uso em Criptografia

Monografia submetida ao curso de graduação em Engenharia de Softwareda Universidade de Brasília, como requisito parcial para obtenção do Título de Bacharel em Engenharia de Software.

Trabalho aprovado. Brasília, DF, 03 de dezembro de 2015:

Prof. Dr. Edson Alves Costa Júnior
Orientador

Prof. Dr. Tiago Alves Fonseca
Convidado 1

Prof. Dr. Fernando William Cruz
Convidado 2

Brasília, DF
2015

Este trabalho é dedicado à minha família, que sempre me apoiou em todos os meus sonhos e onde muitas vezes busquei a força que faltava em mim.

Agradecimentos

Meus sinceros agradecimentos ao meu orientador Prof. Dr. Edson Alves da Costa Júnior, que não só é um excelente professor, como uma grande pessoa. Ao professor Dr. Tiago Alves pela ajuda e pelas inúmeras sugestões de melhoria deste trabalho. Aos meus amigos, por todo apoio recebido. Agradeço, em especial, a minha família, pelo apoio incondicional e por ter estado comigo em todos os momentos dessa jornada.

"Em algum lugar, algo incrível está esperando para ser descoberto"
(Carl Sagan)

Resumo

Curvas elípticas são belas e interessantes estruturas matemáticas. Possuem uma grande importância na Teoria dos Números, sendo usada, por exemplo, na prova do último Teorema de Fermat, na fatoração de inteiros e na criptografia. Seu uso em criptografia, descoberto em 1985, possibilitou a criação de uma alternativa às ferramentas já existentes, oferecendo uma série de vantagens em relação a seus concorrentes. Um ponto fundamental no uso de curvas elípticas é a escolha de seus parâmetros de domínio, que definem não só a forma da curva como também algumas propriedades que são úteis quando usadas em criptografia. Existem diversas especificações para a geração de curvas elípticas, cada uma delas buscando atender a uma demanda específica, indo de sistemas embarcados a sistemas bancários. O objetivo deste trabalho é mostrar todos os principais conceitos matemáticos que são empregados no processo de criação de curvas elípticas, explorar algoritmos já existentes e que já estão em uso e implementar um algoritmo para gerar tais parâmetros.

Palavras-chaves: curva elíptica. criptografia. teoria dos números.

Abstract

Elliptic curves are beautiful and interesting mathematical structures. They have a great importance in number theory, being used, for example, in the test Fermat's Last Theorem, in integer factorization and encryption. Its use in encryption, discovered in 1985, enabled the creation of new alternative to existing tools, offering a number of advantages in relation to its competitors. A key point in the use of elliptic curves is the choice of the domain parameters that define not only the shape of the curve but also some properties that are useful when used in cryptography. There are several specifications for elliptic curves generation, each one seeking to meet a specific cenario, ranging from embedded systems to banking system. The objective of this work is to show all major mathematical concepts which are employed in the creation process of elliptic curves, explore algorithms already existing and already in use and implement an algorithm to generate these parameters.

Key-words: elliptic curves. criptography. number theory.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

ECC	Elliptic Curve Criptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EC	Elliptic Curve
NIST	National Institute for Standards and Technology
RSA	Rivest-Shamir-Adleman cryptosystem
AES	Advanced Encryption System
DSA	Digital Signature Algorithm
ECIES	Elliptic Curve Integrated Encryption Scheme
ECDLP	Elliptic Curve Discrete Logarithm Problem

Lista de símbolos

Δ	Letra grega Delta
δ	Letra grega minúscula Delta
φ	Letra grega minúscula Phi
σ	Letra grega minúscula Sigma
π	Letra grega minúscula Pi
$\subseteq$	Subconjunto de
$\in$	Pertence
$\forall$	Para todo
$\exists$	Existe
$\equiv$	Congruente
$\rightarrow$	Implica que

Sumário

Introdução

Origens da Criptografia

O termo criptografia, que deriva do grego *kryptós* (oculto) e *gráphein* (escrever), se refere ao conjunto de técnicas, regras e algoritmos que permitem que uma mensagem seja enviada de um remetente ao seu destinatário sem que seu conteúdo seja revelado (SINGH, 2000). Atualmente este termo se tornou mais abrangente, também sendo utilizado para designar outras ferramentas como os algoritmos de assinatura digital, certificados digitais etc.

O conceito de criptografia esteve por muito tempo atrelado às técnicas de cifragem, mas durante a Segunda Guerra Mundial a pesquisa de algoritmos de criptografia se acentuou, dada a necessidade de se trocar informações estratégicas entre unidades de combate. Com o advento de computadores com capacidade de executar milhares de cálculos por segundo, matemáticos passaram a dar uma grande atenção à essa área, resultando na descoberta de várias novas técnicas (EICHER JODIE; OPOKU, 1997).

Até a década de 1970, os algoritmos de criptografia eram baseados apenas em técnicas que utilizavam uma mesma senha para cifrar e decifrar uma mensagem, que são os algoritmos de chave simétrica. O maior problema desta família de algoritmos está na distribuição segura desta chave entre todas as partes que devem cifrar e decifrar as mensagens. Para resolver esse problema foram criados, então, os algoritmos de chave assimétrica. Nestes algoritmos, cada parte possui um par de chaves, sendo uma pública, que pode ser conhecida por qualquer pessoa, e outra privada, de conhecimento exclusivo do usuário. Uma mensagem cifrada com uma chave pública só pode ser aberta pela chave privada correspondente, garantindo que apenas o destinatário correto seja capaz de ler a mensagem.

O primeiro padrão de criptografia assimétrica amplamente utilizado foi o RSA (KALISKI, 2006). Desenvolvido em 1976, foi o primeiro algoritmo assimétrico de criptografia utilizado em larga escala e que ainda continua sendo utilizado.

Como uma alternativa aos algoritmos que usam exponenciação modular, como o RSA, a criptografia de curvas elípticas foi proposta em 1987 por Neal Koblitz e Victor S. Miller (KOBLITZ, 1987). Ela utiliza curvas algébricas cúbicas e suas propriedades como o meio para gerar chaves criptográficas seguras. Sua principal vantagem é que, para oferecer o mesmo nível de segurança da RSA, ela necessita de chaves consideravelmente menores. Neste contexto, um ponto crucial para sua confiabilidade está no processo para a escolha dos parâmetros de domínio que caracterizam tais curvas. Este processo pode

ser tanto a escolha arbitrária dos valores destes parâmetros quanto a geração sistemática e automatizada por um algoritmo.

Atualmente, os padrões comerciais de criptografia mais utilizados nas aplicações ao redor do mundo foram criados pela entidade norte americana, o NIST (*National Institute of Standars and Technology*), incluindo parâmetros de domínio para curvas elípticas. Com as denúncias de espionagem ao governo brasileiro por parte dos EUA reveladas em 2012, várias ações tem sido tomadas para garantir a soberania nacional e a proteção de dados sigilosos. Todavia, mesmo após tal escândalo, o governo brasileiro não possui parâmetros de curvas elípticas publicados para uso em seus órgãos.

Objetivos

O objetivo geral deste trabalho é apresentar um algoritmo para gerar parâmetros de domínio de uma curva elíptica que atendam os requisitos de segurança reconhecidos internacionalmente.

Como objetivo específico, este trabalho busca desenvolver uma implementação *open source* de rotinas de geração de parâmetros de domínio de curvas elípticas e dos testes de validação de segurança dos parâmetros gerados.

Escopo

A proposta deste trabalho se limita ao estudo dos elementos necessários para a geração de parâmetros de domínio de curvas elípticas, os parâmetros atualmente em uso, juntamente com seus algoritmos geradores, e a proposta de implementação de um algoritmo para gerar estes parâmetros. As aplicações de tais curvas, como cifragem e assinatura digital, não fazem parte do escopo do trabalho, apesar de serem brevemente explicadas no Capítulo 1.

Estrutura do Trabalho

Este trabalho está dividido em 4 (quatro) capítulos. No Capítulo 1 são apresentados os principais conceitos matemáticos relacionados a curvas elípticas, uma visão geral sobre criptografia de chaves simétricas e assimétricas, algoritmos de cifragem e assinatura digital que utilizam curvas elípticas e os principais ataques a esses algoritmos. No Capítulo 2 são apresentadas todas as ferramentas utilizadas nos experimentos, as técnicas e metodologias empregadas e os algoritmos desenvolvidos. No Capítulo 3 são apresentados os resultados parciais da implementação dos algoritmos e uma breve avaliação do tempo de

execução dos algoritmos implementados. Por fim, o Capítulo 4 apresenta as considerações sobre o trabalho desenvolvido e descreve os trabalho futuros.

1 Fundamentação Teórica

A geração de parâmetros de curvas elípticas para criptografia depende de diversos conceitos algébricos, de forma a garantir que a curva resultante não seja suscetível a ataques matemáticos e computacionais, ou pelo menos garantir que tais ataques sejam dispendiosos ao ponto de torná-los impraticáveis. Neste capítulo serão mostrados os conceitos matemáticos necessários para a geração desses parâmetros.

1.1 Aritmética Modular

Na Teoria dos Números, ramo da matemática pura que estuda os números inteiros (NIVEN; ZUCKERMAN; S., 2014), uma importante ferramenta é a aritmética modular, a qual estuda as propriedades de conjuntos formados a partir dos restos da divisão de números inteiros por um número inteiro positivo n maior ou igual a 2, denominado módulo. A seguir, iremos discutir alguns aspectos da Teoria dos Números. Porém, antes de começar, é conveniente definir alguns termos e conceitos.

Uma congruência é uma relação entre dois números inteiros que, quando divididos por um mesmo número (chamado de módulo da congruência), deixam o mesmo resto. Em termos formais, dois números inteiros a e b são congruentes módulo n se n divide a diferença $a - b$. Esta relação pode ser expressa pela Equação 1.

$$a \equiv b \pmod{n} \tag{1}$$

Para um inteiro positivo n e a, b, c inteiros quaisquer, sendo a e b congruentes módulo n a congruência tem as propriedades listadas nas Equações 2, 3 e 4.

$$a + c \equiv b + c \pmod{n} \tag{2}$$
$$a - c \equiv b - c \pmod{n} \tag{3}$$
$$ac \equiv bc \pmod{n} \tag{4}$$

Em relação a divisão, ela não está definida em todos os casos. Se d é um inteiro que divide a e b, então vale a relação apresentada na Equação 5, onde (n, d) é o maior divisor comum entre n e d.

$$\frac{a}{d} \equiv \frac{b}{d} \left(\mod \frac{n}{(n,d)} \right) \tag{5}$$

O inverso multiplicativo de a modulo n, isto é, um inteiro b tal que $ab = 1 \pmod{n}$, existe apenas quando $(a, n) = 1$ (isto é, a e n são coprimos). O valor de b pode ser obtido através do Algoritmo de Euclides Estendido (HALIM, 2013).

A exponenciação modular calcula o resto de um número inteiro b quando elevado à um número inteiro k e reduzido módulo m. Escolhidos a base b, o expoente k e o módulo m, a exponenciação modular c é dada pela Equação 6.

$$c \equiv b^k \pmod{m} \tag{6}$$

1.2 Estruturas Algébricas

1.2.1 Operações Binárias

Seja L um conjunto não vazio. Uma operação sobre L é dita binária se ela tem como parâmetros dois elementos do conjunto L e os associa a um terceiro elemento em L, sendo que L opera em cada par $(a, b) \in L \times L$. Uma operação binária sobre L é grafada conforme mostrado na Equação 7.

$$\varphi : L \times L \to L \tag{7}$$

Uma operação binária é:

1. *Fechada*, se e somente se $\forall a, b, c \in L, (a\varphi b) \in L$;
2. *Comutativa*, se $\forall a, b \in L, (a\varphi b) = (b\varphi a)$;
3. *Associativa*, se $\forall a, b, c \in L, (a\varphi b)\varphi c = a\varphi(b\varphi c)$.

Uma operação binária possui um elemento neutro $e \in L$ se $\forall a \in L, e\varphi a = a\varphi e = a$.

Dado um elemento $a \in L$, a é invertível em relação à operação φ se existe um elemento $a' \in L$ que satisfaz a expressão $a\varphi a' = a'\varphi a = e$.

Um exemplo de operação binária é a soma no conjunto dos números naturais, como mostrado na Equação 8.

$$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \tag{8}$$

A imagem de um par $(a, b) \in \mathbb{N} \times \mathbb{N}$ pela operação $+$ é denotada por $a + b$ e é chamada de soma de a e b.

1.2.2 Monóides e Grupos

Seja L um conjunto não vazio. Uma estrutura algébrica (L, φ) (composta por um conjunto não vazio e uma operação binária) é um monóide se satisfaz as seguintes propriedades:

1. se a operação φ tem um elemento neutro em L;
2. se a operação φ é associativa;
3. se a operação φ possuir uma única operação binária.

Um grupo é um *monóide* (G, φ) que atende também à propriedade:

3. todo elemento $g \in G$ é invertível em relação a operação φ, isto é, todo elemento g de G possui um elemento simétrico g' tal que $(g\varphi g') = (g'\varphi g) = e$, onde e é o elemento neutro da operação φ.

Um grupo é dito *abeliano* se a operação for comutativa.

1.2.3 Anéis e Corpos

Seja L um conjunto não vazio, e $\sigma, \pi : L \times L \to L$ duas operações binárias em L denominadas, respectivamente, operações de adição e produto. A título de notação, será utilizada a expressão $a+b$ como equivalente a $a\sigma b$ e $a \times b$ como equivalente a $a\pi b$. O terno ordenado $(L, +, \times)$ é um anel se:

1. $(L, +)$ é um grupo abeliano;
2. $(L, +)$ possui um elemento neutro 0, *i.e*, $\forall a \in A, a + 0 = 0 + a = a$;
3. $(L, +)$ possui elementos simétricos para todos elementos de L, *i.e*, $\forall a \in A, \exists k \in A : a + k = 0$;
4. $(L, \times)$ é um monóide;
5. $(L, \times)$ é associativo, *i.e*, $\forall a, b, c \in L, (a \times b) \times c = a \times (b \times c)$;
6. $(L, +, \times)$ são distributivos, *i.e*, $\forall a, b, c \in A, a \times (b+c) = a \times b + a \times c$ e $(b+c) \times a = b \times a + c \times a$;

Um corpo é um anel cuja operação produto é comutativa e onde todos os elementos diferentes de zero (elemento neutro da adição) possuem um elemento inverso para a

operação produto. Formalmente, um anel comutativo P é chamado de *corpo* se atende à Equação 9.

$$\forall a \in P \neq 0, \exists a' \Rightarrow a \times a' = 1 \tag{9}$$

onde 1 é o elemento neutro da operação produto.

A *característica* de um corpo F é definido como o menor número de vezes que o elemento identidade do produto deve ser somando sucessivamente para obter o elemento identidade da soma. Um corpo tem característica zero se a soma nunca alcançar a identidade da soma. Um corpo é considerado *fechado* se todos os polinômios definidos em F possuem raíz em F

1.2.4 Endomorfismo de Frobenius

O endomorfismo de Frobenius é um endomorfismo especial de anéis comutativos com característica p. Ele eleva cada elemento de um anel à sua *p-ésima* potência.

Seja R um anel comutativo com característica p. O endomorfismo de Frobenius F é definido por:

$$F(r) = r^p, \forall r \in R \tag{10}$$

A multiplicação segue as mesmas propriedades de um anel:

$$F(r \times s) = (r \times s)^p = r^p \times s^p = F(r) \times F(s) \tag{11}$$

O mesmo acontece com a soma:

$$F(r + s) = (r + s)^p = r^p + s^p = F(r) + F(s) \tag{12}$$

1.2.5 Variedades e Curvas Algébricas

Uma *variedade algébrica* é o número de elementos de um conjunto de soluções de um sistema de equações polinomiais. Seja K um corpo fechado, o anel de polinômios $K[T_1, T_2, ..., T_n]$ nas n variáveis $T_1, T_2, ..., T_n$ com coeficientes em K e $\{f_i\}$ uma família de polinômios do anel. É denominada variedade algébrica V o subconjunto K_n formado pelos pontos que anulam (zeram) todos os polinômios da família $\{f_i\}$, como apresentado na Equação 13.

$$V = \{(x_1, ..., x_n) | f_i(x_1, ..., n) = 0; i = 1, ..., n\} \subseteq K \tag{13}$$

A dimensão de uma variedade algébrica V é o conjunto de funções racionais independentes em V.

Formalmente, denomina-se *curva algébrica* uma variedade algébrica de dimensão um. Esse conjunto de curvas descreve as figuras geométricas resultantes de uma seção cônica, a saber: círculos, hipérboles e eclipses (THOMAS, 2004).

1.3 Curvas Elípticas

1.3.1 Definição

Informalmente, curvas elípticas são formadas pelo conjunto de soluções de equações cúbicas definidas sobre um corpo K. Formalmente, uma curva elíptica é uma curva algébrica não singular sobre um corpo K de gênero 1 (SILVERMAN, 2009). Neste trabalho será somente abordado o caso em que K é um corpo finito.

A equação de Weierstrass é uma equação homogênea de grau 3 cuja forma é apresentada na Equação 14.

$$Y^2Z + a_1XYZ + a_3YZ^2 = X^3 + a_2X^2Z + a_4XZ^2 + a_6Z^3 \tag{14}$$

Uma curva de Weierstrass F é singular em um ponto P se as derivadas parciais se anulam nesse ponto, como apresentado na Equação 15.

$$\frac{\partial F}{\partial x}(P) = \frac{\partial F}{\partial y}(P) = \frac{\partial F}{\partial z}(P) = 0 \tag{15}$$

Se essa igualdade for atendida, a curva F tem então pelo menos um ponto singular e, se uma curva possuir ao menos um ponto singular, ela é classificada como uma curva singular.

Para um corpo F com característica $p > 0$, $p \neq 2$ e $p \neq 3$, a equação de Weierstrass pode ser simplificada para a forma normal de Weierstrass, que está descrita na Equação 16.

$$y^2 = x^3 + ax + b \tag{16}$$

A Figura 1 apresenta alguns exemplos de curvas elípticas usando a forma normal da equação de Weierestrass.

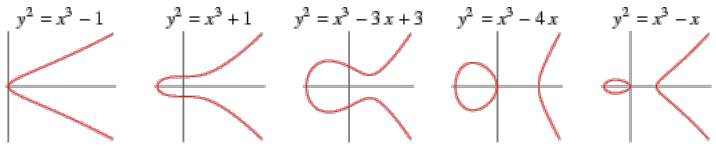

Figura 1: Exemplos de curvas elípticas.

Uma equação de Weierstrass pode ser singular, isto é, pode conter cúspides, pontos isolados ou cruzamentos. Se uma curva elíptica está na forma normal de Weierstrass é possível determinar se ela é singular através do cálculo do seu discriminante, cuja expressão é dada na Equação 17.

$$\Delta = -16(4a^3 + 27b^2) \tag{17}$$

Uma equação de Weierstrass é singular se e somente se seu discriminante Δ for igual a zero.

Uma curva elíptica E pode ser definida, de forma alternativa, como o conjunto de todas as as soluções de uma equação não-singular de Weierstrass mais um ponto especial O, chamado de ponto no infinito. O ponto no infinito é a solução trivial da equação de Weierstrass em sua forma homogênea (Equação 14).

1.3.2 Propriedades

Uma importante característica das curvas elípticas é que elas são grupos abelianos. Todavia, as operações algébricas usadas em curvas elípticas se diferem daquelas usadas, por exemplo, no conjunto dos números inteiros. A operação de soma de pontos de uma curva resulta em um terceiro ponto que também se encontra na mesma curva.

1.3.2.1 O j invariante

Seja E uma curva elíptica na forma normal de Weierstrass (Equação 14). Seu j invariante é dado pela Equação 18.

$$j = j(E) = 1728\left(\frac{4A^3}{4A^3 + 27B^2}\right) \tag{18}$$

Se o j invariante for zero, então a equação pode ser representada pela Equação 19.

$$E \rightarrow y^2 = x^3 + B \tag{19}$$

Se o j invariante for igual a 1728, então a equação assume a forma apresentada na Equação 20.

$$E \rightarrow y^2 = x^3 + Ax \tag{20}$$

Caso o j invariante seja diferente de zero e diferente de 1728 a equação terá a forma normal de Weierstrass (Equação 16).

1.3.2.2 Teorema de Hasse

Seja E uma curva elíptica definida sobre F_q, um corpo finito com q elementos, onde q é uma potência p^n de um número primo p. O número de pontos de E, denominado $\#E(F_q)$, é aproximadamente $q+1$ pontos, com erro de $2\sqrt{q}$, conforme resumido na Equação 21.

$$|\#E(F_q) - q - 1| \leq 2\sqrt{q} \tag{21}$$

Este resultado, denominado *Teorema de Hasse* (JúNIOR, 2005), pode ser utilizado para aproximar (ou se determinar exatamente, quando combinado com outros resultados) o número exato de pontos da curva elíptica E.

1.3.2.3 Adição de Pontos

Sejam P e Q pontos numa curva elíptica E na forma normal de Weierstrass, e (x_p, y_p) e (x_q, y_q) suas respectivas coordenadas. A reta que passa pelos pontos P e Q intercepta a curva E no ponto R, cujo simétrico em relação ao eixo-x é o resultado da soma entre P e Q. Podemos expressar essa operação de adição através das igualdades apresentadas nas Equações 22 e 23.

$$x_r = m^2 - 2x_p \tag{22}$$

$$y_r = m(x_p - x_r) - y_p \tag{23}$$

onde m é o coeficiente linear da reta que passa por P e Q. O valor de m pode ser encontrado através das expressões apresentadas nas Equações 24 e 25.

$$m = \frac{y_q - y_p}{x_q - x_p}, \text{ se } P \neq Q \tag{24}$$

$$m = \frac{3x_p + A}{2y_p}, \text{ se } P = Q \tag{25}$$

1.3.3 Escolha de Parâmetros de Domínio

Gerar uma curva elíptica significa escolher parâmetros que atendam aos requisitos da aplicação dessa curva. No caso específico da criptografia, esses requisitos estão relacionados à capacidade da curva em gerar pontos que não possuam brechas de segurança (KOCHER, 1996) e cuja solução de suas operações sejam de elevado custo computacional (CRANDALL; POMERANCE, 2010).

1.3.3.1 Parâmetros de Domínio

Para se gerar uma curva é necessário, no mínimo, escolher os seguintes parâmetros:

p característica do corpo K;

A coeficiente A da equação da curva na forma reduzida (Equação 16);

B coeficiente B da equação da curva na forma reduzida (Equação 16);

x_0 coordenada x do ponto base P_0;

y_0 coordenada y do ponto base P_0;

p a ordem p do ponto base.

Além desses parâmetros, alguns algoritmos de geração de curvas podem selecionar outros parâmetros para, por exemplo, indexar pontos, comprimir coordenadas, etc.

1.3.3.2 Curvas Empregadas Atualmente

Existem diversas propostas de curvas elípticas para aplicações em criptografia. Aqui serão listadas apenas as mais relevantes, em termos de abrangência e adoção.

NIST. O padrão de assinatura digital DSA (NIST, 2014) recomenda uma série de curvas elípticas para o uso pelo governos dos EUA. Todas as curvas propostas sobre $GF(p)$ têm cofator 1 e a forma $E : y^2 = x^3 - 3x + B \pmod{p}$. São oferecidas curvas para criptografia de 192, 224, 256, 384 e 521 *bits*.

SECG. O Grupo de Padrões para Criptografia Eficiente (*Standards for Efficient Cryptography Group*) é um consórcio internacional fundado em 1998 pela Certicom com o objetivo de criar padrões para criptografia baseados em curvas elípticas. Suas curvas (SECG, 2010) tem tamanho em *bits* de 112, 128, 160, 192, 224, 256, 384 e 521. Os parâmetros escolhidos visam a a eficiência das implementações e, portanto, não são aleatórios.

ANSI. O padrão X9.62 (ANSI, 2005) menciona duas curvas sobre $GF(p)$, sendo uma tomada a partir de FIPS-186-2 e a outra com tamanho de 239 *bits*.

IETF. RFC 2409 (IETF, 1998a) e RFC 2412 (IETF, 1998b) suportam o terceiro e quarto Group de Oakley. Essas são curvas sobre $GF(2155)$ e $GF(2185)$, respectivamente.

ECC-Brainpool. As curvas Brainpool (ECC-BRAINPOOL, 2005) são empregadas pelo governo alemão na autenticação de documentos oficiais e na assinatura digital dos vistos para estrangeiros. O algoritmo de geração de curvas é aberto e pode ser implementado por terceiros. São definidas curvas para um corpo finito sobre um número primo p. São geradas curvas com tamanhos de chaves de 160, 192, 224, 256, 320, 384 e 512 *bits*.

1.3.4 Contagem de Pontos

Em todos os algoritmos de geração de parâmetros de domínio é necessário, em algum ponto, calcular a ordem $\#E(F_q)$ da curva, ou seja, calcular o número de pontos da curva. Através dessa informação é possível verificar algumas características relacionadas ao nível segurança oferecido por ela, sendo a principal delas a de verificar a dificuldade em se resolver o problema do logaritmo discreto.

Existem alguns métodos para realizar a contagem de pontos de uma curva elípticas, sendo os mais conhecidos o algoritmo de Força Bruta 1.3.4.1, o algoritmo de Schoof 1.3.4.2 e o algoritmo de Schoof-Elkies-Atkin 1.3.4.3.

1.3.4.1 Força Bruta

Trata-se da abordagem mais simples: percorre-se todos os elementos de F_q verificando quais satisfazem a equação normal de Weierstrass 16.

Este algoritmo tem complexidade $O(q)$.

1.3.4.2 Algoritmo de Schoof

Criado em 1985 por René Schoof, foi o primeiro algoritmo determinístico com tempo de execução polinomial. Até sua descoberta, os algoritmos para contagem de pontos possuíam tempo de execução exponencial. Partindo do princípio que a ordem $\#E(F_q)$ pode ser estimada através do Teorema de Hasse, o algoritmo de Schoof se limita a calcular $t = q+1-\#E(F_q)$, para $t \mod N$ onde $N > \sqrt[4]{q}$. Todavia, nesse ponto é também muito difícil encontrar o valor de t. A solução encontrada foi encontrar $t \mod m$ para um primo m pequeno, o que é fácil. É escolhida então uma lista $S = \{m_1, m_2, ..., m_k\}$ tal que $\prod m_i = N > \sqrt[4]{q}$. Dado $t \mod m_i$ para todo $m_i \in S$, é possível utilizar o Teorema Chinês dos Restos para calcular $t \mod N$.

Este algoritmo tem complexidade $O(n^5)$ para execução $O(n^3)$ no espaço.

1.3.4.3 Algoritmo de Schoof-Elkies-Atkin

Em 1990 uma versão melhorada do algoritmo de Schoof foir criada por Noam Elkies e Arthur O. L. Atkin. Este algoritmo faz a distinção entre os primos utilizados, apenas números primos específicos são utilizados. Um número primo m é chamado de primo de Elkies se a equação do endomofismo de Frobenius $\phi^2 - t\phi + q = 0$ funciona para F_m, caso contrário ele é chamado de primo de Atkin.

Este algoritmo tem complexidade $O(n^4)$ para execução $O(n^4)$ no espaço.

1.4 Criptografia

Antes do uso massivo de computadores, o conceito de criptografia era quase totalmente ligado ao conceito de cifragem, isto é, tornar uma mensagem compreensível em um texto aparentemente sem sentido (KAHN, 1996). O advento dos computadores permitiu uma extensa gama de estudos sobre criptografia e a criação de novas ferramentas criptográficas, como por exemplo, a assinatura digital. Através dela é possível prover os seguintes tipos de serviço:

- *confidencialidade*: apenas o destinatário deve ser capaz de compreender uma mensagem cifrada;

- *integridade*: o destinatário deve ser capaz de detectar quaisquer alterações na mensagem durante sua transmissão;
- *autenticação*: o destinatário deve ser capaz de identificar o remetente da mensagem;
- *irretratabilidade*: ao remetente deve ser impossível negar a autoria da mensagem.

Note que cada ferramenta criptográfica pode satisfazer um ou mais objetivos, não necessariamente todos.

1.4.1 Criptografia simétrica

Na criptografia de chave simétrica (DELFS; KNEBL, 2007) a mesma chave é utilizada tanto para cifrar quanto para decifrar uma mensagem. É considerada a forma mais simples de cifragem, pois existe apenas uma chave para realizar as duas operações. A chave representa um segredo compartilhado entre duas ou mais partes com a premissa de que somente essas partes o conheçam. Como vantagem apresenta a velocidade, chegando a ser de centenas a milhares de vezes mais rápida que algoritmos de chave assimétrica.

Sua desvantagem é a exigência de uma cópia da chave para cada uma das partes envolvidas, aumentando a probabilidade de serem descobertas por um potencial adversário. Por esse motivo, necessitam ser constatemente trocadas, o que gera mais um problema: a sua distribuição entre todas as partes interessadas.

Em um canal de comunicação seguro, normalmente é utilizado um algoritmo de criptografia assimétrico, que é mais lento, para a troca de chaves simétricas entre duas partes. A partir de então, a comunicação é acelerada utilizando uma chave simétrica compartilhada.

1.4.2 Criptografia asssimétrica

Também conhecida como criptografia de chave pública, é um método de criptografia que utiliza duas chaves: uma chave pública e uma chave privada. A chave pública é acessível e a aberta enquanto que a chave privada é de conhecimento apenas de seu dono. Uma mensagem cifrada com a chave pública só pode ser decifrada pela chave privada correspondente.

Dependendo de como for utilizada, a criptografia assimétrica pode ser utilizada para dois diferentes propósitos: cifragem para a troca de mensagens confidenciais e assinatura digital para autenticação de autoria de mensagem. Na troca de mensagens confidenciais, utiliza-se a chave pública para criptografar a mensagem original, garantindo que somente o destinatário poderá abri-la com sua chave privada. No caso da autenticação de autoria envia-se, junto à mensagem que se deseja autenticar, uma assinatura da

mensagem que é gerada a partir do corpo da mensagem e da chave privada do remetente. O destinatário pode verificar a integridade da mensagem utilizando a chave pública do remetente e, caso a mensagem corresponda com a assinatura gerada, pressupõe-se que a mensagem recebida realmente foi enviada pelo destinatário correto.

1.5 Cifragem ECIES

O algoritmo de cifragem ECIES (*Elliptic Curve Integrated Encryption Scheme*) (SHOUP, 2001) permite que duas partes, o remente U e o destinatário V, troquem uma mensagem M de forma confidencial.

Resumidamente, o algoritmo ECIES é composto de quatro etapas:

1. *configuração*: estabalecer parâmetros de domínio e de troca de chaves;
2. *implantação da chave*: o remente U deve receber a chave pública de V;
3. *cifragem de M*: U deve usar a chave pública de V para cifrar M;
4. *decifragem de M*: V deve usar sua chave privada para decifrar M e obter a mensagem original.

Cada uma dessas quatro operações possui diversos passos que devem ser seguidos, cujo detalhamento fogem ao escopo deste trabalho. Para maiores detalhes sobre tais passos e suas implementações ver (SECG, 2009).

1.6 Assinatura Digital (ECDSA)

O algoritmo digital de assinatura por curvas elípticas ECDSA (*Elliptic Curve Digital Signature Algorithm*) (JOHNSON; MENEZES; VANSTONE, 2001) é uma variante dos algoritmos de assinatura digital DSA (*Digital Signature Algorithm*) (NIST, 2014), que usa operações sobre pontos de curvas elípticas em vez de exponenciações modulares (Seção 1.1).

O uso de curvas elípticas em criptografia se baseia no problema do logaritmo discreto de curvas elípticas (ECDLP) (LAUTER; STANGE, 2009): encontrar um valor k entre 1 e q que satisfaça a equação $Q = kP_0$. Análises de algoritmos que resolvem ECDLP mostram que curvas elípticas oferecem um nível superior de segurança em relação à outros algoritmos de criptografia como, por exemplo, o algoritmo RSA (RSA LABS, 2012).

Sua principal vantagem sobre outras variantes é que, para um mesmo nível de segurança, o ECDSA utiliza chaves de tamanhos menores (ver Tabela 1).

1.6.1 Tamanho da Chave

Diferentes tipos de criptografia podem exigir diferentes tamanhos de chaves para oferecer o mesmo nível de segurança. A Tabela 1 lista um comparativo do tamanho em *bits* das chaves de diferentes algoritmos para um mesmo nível de segurança.

Tabela 1: Comparação entre o tamanho das chaves para um mesmo nível de segurança (ENISA, 2014).

Algoritmo	**Tamanho em *bits***				
Chave Simétrica	80	112	128	160	256
ECDSA, primo p	160	224	256	320	512
RSA, tamanho do módulo	1024	2048	3072	7628	15360

1.6.2 Algoritmo de Geração Assinatura

O algoritmo de geração de assinatura se baseia no problema do logaritmo discreto. Se duas partes desejam trocar mensagens digitalmente assinadas, elas precisam usar três parâmetros em acordo: uma curva elíptica E, um ponto base G e um número inteiro n tal que $nG = O$.

Uma das partes gera então um par de chaves criptográficas, sendo a chave primária d selecionada aleatóriamente no intervalo $[1, n-1]$, e a chave pública computada através da expressão $Q = dG$. Uma vez geradas as chaves, são executados os seguintes passos:

1. calcular $h = \text{hash}(msg)$, onde msg é a mensagem a ser assinada e $hash$ é uma função hash criptográfica, como SHA-1 (NIST, 2012) ou MD5 (IETF, 1992);
2. seja z número formado pelos l *bits* mais a esquerda de h, onde l é o tamanho em *bits* de n;
3. selecionar um inteiro aleatório e seguro k no intervalo $[1, n-1]$;
4. calcular o ponto na curva $E : (x, y) = kG$;
5. calcular $r = x \pmod{n}$. Se $r = 0$, retornar ao passo 3.
6. calcular $s = k - 1(z + rd) \pmod{n}$. Se $s = 0$, retornar ao passo 3.

A assinatura gerada será o dada pelo par (r, s).

1.6.3 Algoritmo de Verificação de Assinatura

Para autenticar a assinatura recebida, o destinatário deve receber uma cópia da chave publica Q, gerada na seção anterior. É possível, então, verificar se Q é um ponto válido de E seguindo os seguintes passos:

1. verificar se $Q \neq O$;
2. verificar se Q está na curva E;
3. verificar se $nQ = O$.

Uma vez que a chave pública Q foi validada, para validar a assinatura recebida é necessário seguir os seguintes passos:

1. verificar que r e s são inteiro no intervalo $[1, n-1]$;
2. calcular $e = \text{hash(msg)}$, onde *hash* é a mesma função utilizada na geração da assinatura;
3. seja z número formado pelos l *bits* mais a esquerda de e, onde l é o tamanho em *bits* de n;
4. calcular $w = s^{-1} \pmod{n}$;
5. calcular $u_1 = zw \pmod{n}$ e $u_2 = rw \pmod{n}$;
6. calcular o ponto $(x, y) = u_1 G + u_2 Q$;

A assinatura será válida se e somente se $r \equiv x \pmod{n}$.

1.7 Ataques Conhecidos

1.7.1 Ataques de Canal Lateral

Ataques de canal lateral (JOYE, 2003) são baseados no tempo de processamento de operações matemáticas dentro de um microprocessador e no consumo de energia de tais operações. Em resumo, cada operação matemática que é calculada dentro de um microprocessador possui uma assinatura única de tempo de execução e de consumo de energia. Analisando essas assinaturas e possuindo algumas informações sobre os parâmetros de domínio, é possível reconstruir as operações matemáticas realizadas pelo computador e então obter as chaves criptográficas empregadas.

1.7.1.1 Análise de Tempo

O ataque de análise de tempo (KOCHER, 1996) usa o fato de que toda operação matemática possui um tempo específico de execução em um processador e que este tempo pode variar dependendo do tamanho e do tipo da entrada do operador. De posse de informações precisas sobre a execução de cada operação, é possível inferir informações sobre o que está sendo calculado pelo processador.

1.7.1.2 Análise de Carga

O ataque de análise de carga (KOCHER; JAFFE; JUN, 1999) se baseia no fato de que cada operação realizada por um processador consome uma quantidade específica de energia elétrica. Dessa forma, registrando todos os picos de consumo do processador durante a fase de cifragem/decifragem de uma mensagem é possível reconstruir todas as operações que foram realizadas e a partir daí, recuperar, com um alto grau de precisão, o valor da chave privada.

1.7.2 Ataques Usando Computação Quântica

Ataques à ECDLP usando computação quântica (YANG, 2013) são baseados no algoritmo de Shor (SHOR, 1994), cuja função é encontrar os fatores primos de um número inteiro N qualquer. Usando essa propriedade é possível resolver o problema do logaritmo discreto, como descrito em (PROOS; ZALKA, 2008) e (EICHER JODIE; OPOKU, 1997).

2 Metodologia

Para o processo de geração dos parâmetros de uma curva elíptica, foi utilizado como referência o documento de padrões de curva geração de curvas Brainpool (ECC-BRAINPOOL, 2005), pelo fato de não estar sob nenhuma licença que restringe seu uso para fins de pesquisa, e também por ser um padrão consolidado e utilizado em aplicações comerciais.

2.1 Ferramentas Utilizadas

Nesta seção é descrito todo o ambiente de desenvolvimento, desde as configurações físicas do computador ao software utilizado.

As especificações do computador utilizado no desenvolvimento são descritas pela Tabela 2, as especificações do computador que funcionou como segundo nó na distribuição dos processos são descritas pela Tabela 3 e a listagem dos softwares empregados no ambiente de desenvolvimento são descritas na Tabela 4 e, por fim, as bibliotecas e ferramentas de apoio são descritas na Tabela 5.

Tabela 2: Configuração de hardware do computador de desenvolvimento principal, também utilizado como um nó no processamento de dados distribuído.

Item	**Especificação**
Placa Mãe	ASUS B85M-G
Processador	Intel Core i5 3.5GHz 4690
Memória	Kingston HyperX 4GBx2 DDR3 1600MHz Dual Channel
Armazenamento	Kingston 240GB SSD; Seagate 1TB 7.200RPM,HDD
Adaptador de Vídeo	Asus STRIX GTX 970 4GB

Tabela 3: Configuração de hardware do segundo nó de processamento.

Item	**Especificação**
Placa Mãe	Intel HM77
Processador	Intel Core i5 1.7GHz 3317U
Memória	Kingston 4GBx2 DDR3 1366MHz Dual Channel
Armazenamento	eSATA 32GB SSD; Seagate 512GB 5.400RPM,HDD
Adaptador de Vídeo	AMD Radeon 7570M + Intel HD Graphics 4000

Tabela 4: Ferramentas de desenvolvimento empregadas no processo de experimentação.

Ferramenta	**Nome**	**Versão**	**Comentário**
Sistema Operacional	Arch Linux	-	Arch Linux é uma distribuição *rolling release*, por essa razão, não possui uma versão oficial.
Linguagem	C++	11	O padrão C++11 foi escolhido por possuir mecanismos na própria linguagem que facilitam o desenvolvimento.
Compilador	LLVM	3.6.1	A saída de compilação é melhor estruturada e é compatível com os argumentos das ferramentas GNU.
Builder	GNU Make	4.1	
Editores de Texto	VIM, Geany	7.4, 1.24	-

Tabela 5: Bibliotecas e ferramentas utilizadas no desenvolvimento.

Nome	**Versão**	**Comentário**
GNU Multiple Precision Arithmetic Library	6.0.0	Biblioteca para tratar números com tamanho arbitrário e realização de operações algébricas diversas.
Python	3.4	Usada principalmente para prototipações rápidas.
GNU Parallel	5.0.0	Ferramenta para paralelização e distribuição de processos.
MIRACL	7.0.0	Biblioteca com diversos utilitários para criptografia e contagem de pontos de curvas elípticas.

2.2 Técnicas e Metodologias

A implementação dos softwares para experimentação do processo de geração de parâmetros de uma curva elíptica teve como objetivo validar a viabilidade de implementação e de assertividade dos algoritmos escolhidos. Foram criados diversos módulos, onde cada um deles é responsável pela implementação de um algoritmo específico.

O desenvolvimento de cada módulo foi realizado em quatro etapas, descritas a seguir:

1. *Levantamento teórico.* O primeiro passo para a implementação de cada módulo.

Como cada módulo implementa um único algoritmo, para cada um deles foi estudada a teoria que guiava sua implementação.

2. *Prototipação.* Nesta fase os algoritmos foram escritos em linguagem de script para avaliar a dificuldade de implementação e analisar a qualidade de seus dados de saída.

3. *Implementação.* Uma vez que os algoritmos escolhidos foram validados e que seus resultados de saída eram condizentes com os resultados esperados, o próximo passo foi reescrevê-los em uma linguagem com maior desempenho. Para alcançar o desempenho necessário e conseguir lidar com números de tamanho arbitrário, os scripts foram então reescritos em linguagem compilada.

2.3 Algoritmos Desenvolvidos

2.3.1 Operações sobre Curvas

Tais operações são baseadas nas estruturas matemáticas e cálculos apresentados na Seção 1.3.

1. cálculo do j invariante;
2. cálculo do discriminante;
3. soma entre dois pontos;
4. multiplicação de um ponto por um escalar.

As implementações destes algoritmos são apresentadas no Apêndice F.

2.3.2 Geração de Curvas

Com exceção dos três primeiros, que são atômicos, estes algoritmos são compostos por outros algoritmos menores, que podem ser implementados de tal forma a serem reutilizados. São eles:

Extração dos n bits mais a direita de um número inteiro (Código 2.1): dado um número inteiro qualquer, esse algoritmo cria um novo número inteiro, cujo valor é dado pelos *n bits* mais à direita do número original.

Código 2.1: Algoritmo para extração dos *n bits* mais a direita de s.

```
1 extract_bits(s: inteiro, n: inteiro): inteiro
2     seja t, inteiro
3     t ← 2n − 1
4     seja r, inteiro
5     r ← t & s
```

```
6     retorne r
```

Conversão de um vetor de bytes para um número inteiro (Código 2.2): dado vetor de *bytes*, este algoritmo obtém os valores de cada *byte* e os expande em um número inteiro.

Código 2.2: Algoritmo para expandir os *bits* de uma um vetor de *bytes* em um inteiro.

```
1 expand_s(s: byte[]): inteiro
2     seja n, inteiro
3     n ← length(s)
4     seja k, inteiro
5     para i ← 0, enquanto i < n, faca:
6         k ← k & (s[i] << i*8)
7         i ← i + 1
8     retorne k
```

Conversão de um número inteiro para um vetor de bytes (Código 2.3): dado um número inteiro, este algoritmo obtém os *bytes* desse número e os extrai para um vetor de *bytes*. `BitLen` é uma função da biblioteca GnuMP que retorna o número de *bits* necessários para se representar um determinado número.

Código 2.3: Algoritmo para extração dos *bits* de um número inteiro para um vetor de *bytes*.

```
1 expand_i(k: inteiro): byte[]
2     seja nbytes, inteiro
3     nbytes ← BitLen(k) / 8
4     seja r, byte[nbytes]
5     para i ← 0, enquanto i < nbytes, faca:
6         r[i] ← k & (0xFF << i)
7         i ← i + 1
8     retorne r
```

Atualização de uma seed (*semente*) (Código 2.4): dado um vetor de *bytes*, esse algoritmo gera uma nova *seed* pseudo-aleatória.

Código 2.4: Algoritmo para atualização de uma *seed*.

```
1 update_seed(s: byte[]): byte[]
2     seja z, inteiro
3     z ← expand_s(s)
4     seja t, byte[]
5     t ← expand_i((z + 1) mod 2^160))
6     retorne t
```

De acordo com o guia de geração de curvas Brainpool (ECC-BRAINPOOL, 2005), para gerar uma curva com L *bits* de segurança, são necessários quatro algoritmos principais para a geração de uma curva elíptica: Inicialização, Geração de números pseudo-aleatórios,

Geração de Números Primos e Geração de curvas pseudo-aleatórias. Estas algoritmos são descritos a seguir.

Inicialização (Código 2.5). dado um número inteiro L, que é o tamanho em *bits* das chaves criptográficas, este algoritmo calcula dois parâmetros, v e w, onde v corresponde ao número de hashes *SHA-1* que devem ser calculadas em cada passo do gerador pseudo-aleatório de números e w corresponde a quantos *bits* sobram ou faltam para o próximo valor de v.

Código 2.5: Algoritmo para obtenção dos parâmetros iniciais.

```
1 pre(L: inteiro): inteiro, inteiro
2     seja v, inteiro
3     seja w, inteiro
4     v ← Ceil((L − 1)/160)
5     w ← L − 160 ∗ v
6     retorne v, w
```

Geração de números pseudo-aletórios (Código 2.6): dada uma *array* de *bytes* `seed`, dois inteiros `v` e `w`, este algoritmo gera um novo número pseudo-aleatório.

Código 2.6: Geração de números pseudo-aleatórios.

```
1  gen_number(seed: byte[], v: inteiro, w: inteiro): inteiro
2      seja z, inteiro
3      z ← expand_s(seed)
4
5      seja tmp, byte[]
6      tmp ← SHA1(seed)
7
8      seja h, byte[v]
9      h[0] ← expand_i(extract_bits(expand_s(tmp), w))
10
11     para i ← 1, enquanto i < v, faca:
12         seja si, byte[]
13         si ← expand_i((z + i) mod 2^160)
14         h[i] ← SHA1(si)
15         i ← i + 1
16
17     seja k, byte[]
18     k ← h[0] | h[1] | h[2] ... | h[v]
19
20     seja result, inteiro
21     result ← expand_s(k)
22
23     retorne result
```

Geração de números primos (Código 2.7): dado um número inteiro positivo `L` e um *array* de *bytes*, este algoritmo gera um novo número primo pseudo-aleatório a partir da *seed* `s` com um tamanho total de `L` *bits*.

Código 2.7: Algoritmo para a geração de números primos pseudo-aleatórios.

```
1  prime_gen(L: inteiro, s: byte[]): inteiro
2      seja v, inteiro
3      seja w, inteiro
4
5      v, w ← pre(L)
6
7      seja c, inteiro
8      c ← gen_number(s, v, w)
9
10     seja p, inteiro
11     p ← NextPrime(c, 3, 4)
12
13     seja lower, inteiro
14     seja upper, inteiro
15
16     lower ← 2 * (L − 1) − 1
17     upper ← 2 * L
18
19     se p < lower ou p > upper:
20         s ← update_seed(s)
21         reinicie
22
23     se IsPrime(p) = false:
24         s ← update_seed(s)
25         renicie
26
27     retorne p
```

Verificação de Integridade da Curva (Código 2.8): Verifica a integridade de uma curva, dados os parâmetros A e B.

Código 2.8: Verificação de Integridade da Curva.

```
1  check_integrity(A: inteiro, B:inteiro, L:inteiro, p:inteiro): booleano
2
3    se #E(GF(p)) < p:
4      retorne falso
5
6    se B ≡ B^2 (mod p):
7      retorne falso
8
9    se AZ^4 (mod p) = ∅:
10     retorne falso
```

```
11
12  retorne verdadeiro
```

Verificação de Segurança da Curva (Código 2.9): Verifica a segurança de uma curva, dados os parâmetros A e B.

Código 2.9: Verificação de Segurança da Curva.

```
1  check_sec(A: inteiro, B:inteiro, p:inteiro, q:inteiro): booleano
2
3    seja l, inteiro
4    l ← degree(A, B, p)
5
6    se (q - 1) / l ≥ 100:
7      retorne falso
8
9    se p < 10^{7}:
10     retorne falso
11
12   se #E(GF(p)) = p:
13     retorn falso
14
15   se #E(GF(p)) != q:
16     retorne falso
17
18   retorne verdadeiro
```

Geração de parâmetros de domínio (Código 2.10): dado um inteiro `L`, que é o tamanho em *bits* do número primo a ser gerado, e uma *array* de *bytes* `s`, que é a *seed* para gerar números pseudo-aleatórios, este algoritmo gera parâmetros de domínio de uma curva elíptica.

Código 2.10: Geração de parâmetros de domínio de uma curva elíptica sobre GF(p).

```
1  gen_curve(L: inteiro, s: byte[]): inteiro, byte[], inteiro, byte[],
       inteiro, inteiro
2      seja v, inteiro
3      seja w, inteiro
4      v, w ← pre(L)
5
6      seja A, inteiro
7      A = gen_number(s, v, w)
8
9      seja seed_A, byte[]
10     seed_a ← s
11
12     # gerar B
13     s ← update_seed(s)
```

```
14
15      seja B, inteiro
16      B = gen_number(s, v, w)
17
18      se B ≡ B^2 (mod p):
19          s ← update_seed(s)
20          reinicie
21
22      seja seed_B, byte[]
23      seed_B ← s
24
25      se −16 ∗ (4 ∗ A^3 + 27 ∗ B^2) = 0 entao:
26          s ← update_seed(s)
27          reinicie
28
29      # passo 20: verificacao se seguranca
30      SecCheck(A, B)
31
32      s ← update_seed(s)
33      seja seed_BP, byte[]
34      seed_BP ← s
35
36      seja Mult, inteiro
37      Mult = gen_number(s, v, w)
38
39      # passo 29: contar pontos
40      CheckIntegrity(A, B, Mult)
41
42      retorne A, seed_A, B, seed_B, Mult, P0
```

2.4 Contagem de Pontos

Os algoritmos para geração de parâmetros de domínio de uma curva elíptica são potencialmente complexos, com uma grande ordem de complexidade tanto de execução quanto de consumo de memória. Além disso, a maior parte das ferramentas disponíveis para realizar os cálculos envolvidos não são otimizados para aproveitar computadores que possuem processadores com vários núcleos ou mesmo o uso de *threads*.

Para chegar no resultado final, foram realizadas algumas tentativas com diferentes ferramentas.

2.4.1 Ferramentas

2.4.1.1 Python-Schoof

Esta implementação em Python do algoritmo de Schoof é de fácil entendimento e de simples implementação. Seu código é aberto e muito bem documentado, além de ser facilmente extensível.

Seu objetivo é demonstrar o funcionamento do algoritmo de forma acadêmica, por essa razão o desempenho de execução não recebeu atenção. Ela funciona bem para curvas pequenas, onde p possui cerca de 4 a 6 *bits*. Acima disso o tempo de execução se torna impraticável para fins práticos. Além dessa limitação, por limitações do próprio Python, a implementação não aceita números maiores que 32 *bits*, o que invalida sua escolha já que curva com menor valor de p possui 160 *bits*.

2.4.1.2 C++/Python-Schoof

Por causa das limitações da implementação do Python-Schoof, a segunda tentativa foi a de reimplementar todo o código fonte em linguagem C++, utilizando parte do *framework* que foi desenvolvido durante este trabalho, para resolver os problemas relacionados ao tamanho dos números possíveis.

A implementação Python-Schoof, todavia, utiliza metaprogramação em quase todo seu código, dificultando o processo de reescrita para C++, que possui uma sintaxe mais rígida. Como o objetivo do trabalho não é apenas implementar um algoritmo de contagem de pontos, esta opção também foi descartada.

2.4.1.3 MIRACLE/sea

A biblioteca MIRACL é uma biblioteca matemática para diversas operações algébricas, além de possuir várias ferramentas voltadas à criptografia, dentre elas, a contagem de pontos. É uma biblioteca que vem sendo utilizada há anos, sendo a pioneira a implementar e disponibilizar seu código fonte.

Seus principais defeitos são o fato de não aproveitar os núcleos de processadores com vários núcleos para paralelizar os cálculos que não possuem dependência e o de que a maior parte de suas ferramentas são disponibilizadas na forma de binários, dificultando a integração com outras ferramentas.

Esta biblioteca, no entanto, foi a única capaz de contar pontos para curvas de até 512 *bits* sem apresentar os problemas enfrentados pelas outras e, portanto, foi a escolhida.

2.4.2 Adaptações da Biblioteca MIRACL

Antes de executar o algoritmo de contagem de pontos foi necessário realizar algumas adaptações na ferramenta MIRACL para tornar mais simples a sua utilização, pois ela possui algumas limitações, dentre elas:

- Ausência de um *Makefile*: para gerar os binários necessários, é necessário compilar e *linkar* manualmente os arquivos de código fonte corretos entre mais de 100 arquivos.
- Excesso de saída no terminal: muita informação desnecessária era impressa na saída padrão do programa, dificultando a integração com ferramentas externas que dependem das saídas do dele.
- Execução em um único *thread*: boa parte da biblioteca foi escrita na década de 90, quando a maioria dos computadores possuíam processadores com apenas um núcleo. Na execução de algoritmos com intenso de uso de *CPU* em um processador com vários núcleos, os algoritmos vão utilizar uma única *thread*, ainda que o algoritmo seja paralelizável.

Para atenuar os problemas listados acima foram feitas algumas adaptações no próprio código fonte da biblioteca assim como a utilização de ferramentas externas para otimizar algumas tarefas.

- Automatização da *build*: foi criado um *script* que compila todos os arquivos necessários e gera todos os executáveis.
- Limpeza na saída dos binários: foram removidas todas saídas de depuração do código, sendo deixadas apenas as impressões de resultado e mensagens de erro.
- Paralelização: para a execução paralela de alguns algoritmos foi utilizada a ferramenta GNU Parallel. Esta biblioteca permite paralelizar tarefas tanto em um único computador como em um cluster.

2.4.3 Execução

No algoritmo de Schoof-Elkins-Atkins é necessário executar três passos:

- **Gerar uma lista de polinômios de Müller.** É o passo mais demorado mas é executado apenas uma vez. É gerada uma lista com vários polinômios que são utilizados posteriormente pelo algoritmo de contagem de pontos.
- **Pré-processar o parâmetro p.** É o passo mais rápido. Utilizando a lista de polinômios gerada no passo anterior, este algoritmo gera uma lista dos primos Elkins

e Atkins, como descrito na Seção 1.3.4.3. Deve ser executado uma vez para cada número primo.

- **Contar pontos.** Não tão demorado quanto a geração dos polinômios mas para curvas maiores o cálculo pode demorar várias horas. O algoritmo empregado é o Schoof-Elkins-Atkins e utiliza os primos pré-processados no passo anterior.

O algoritmo de geração de Polinômios de Müller foi executado de forma paralela utilizando a biblioteca GNU Parallel. Além da paralelização usando processos também foi realizada a paralelização de computadores. Dessa forma, o algoritmo pode ser processado utilizando um total de oito núcleos de processamento. A Figura 2 mostra como os computadores foram organizados em uma rede local para executar esta tarefa.

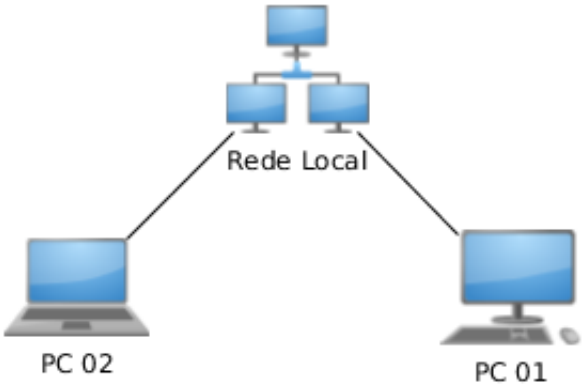

Figura 2: Esquema de um sistema distribuído para cálculo de polinômios de Mueller utilizando GNU Parallel.

3 Resultados

Durante esta fase de desenvolvimento, foi possível implementar as operações sobre curvas (Seção 2.3.1), o algoritmo para geração de números primos (Código 2.7) completamente e o algoritmo de geração de curvas (Código 2.10) parcialmente. Para apoio destes, todos os algoritmos auxiliares (Códigos 2.1, 2.2, 2.3, 2.4, 2.5 e 2.6) também foram completamente implementados.

Os algoritmos foram implementados em linguagem C++ usando a biblioteca GnuMP para lidar operações aritméticas e também para lidar com números de tamanho arbitrário. Alguns binários gerados a partir biblioteca MIRACL foram utilizados para realizar a contagem de pontos.

3.1 Melhorias das ferramentas e técnicas utilizadas

3.1.1 Refatorações no algoritmo Brainpool

A descrição do algoritmo Brainpool é de natureza matemática e, por essa razão, possui alguns detalhes que podem ser repetitivos ou mesmo desnecessários quando observados do ponto de vista da computação. Para facilitar sua compreensão e torná-lo mais próximo de uma implementação real, foram realizadas algumas modificações. São elas:

- **Extração do algoritmo atualizador de *seeds*.** Na versão original da Brainpool, este algoritmo é executado toda vez que a curva é rejeitada em algum passo sendo necessário recalcular a *seed* para gerar novos parâmetros A, B e p. Essa refatoração permitiu reutilizar o mesmo algoritmo em outros passos;
- **Extração do algoritmo de geração de números aleatórios.** Este algoritmo é frequentemente repetido na versão original, em diversos passos, mesmo que ele sempre realize a mesma tarefa, que é a de gerar números aleatórios. Essa refatoração permitiu diminuir a quantidade de passos do algoritmo original;
- **Extração do algoritmo gerador de parâmetros iniciais.** Executado tanto para a geração de números primos quanto para a geração de parâmetros. Sua refatoração elimina a repetição no início dos outros algoritmos.

3.1.2 Script de construção para a MIRACL

Ao obter uma cópia do arquivos de código fonte da biblioteca MIRACL o usuário não possui muitas opções para compilá-lo. A recomendação da própria documentação é

utilizar o Microsoft Visual Studio, caso o usuário utilize o sistema operacional Windows, ou então compilar cada arquivo manualmente caso o usuário utilize outras plataformas, como o Linux. A única forma de automatização presente para sistemas operacionais baseados em Linux é um script que gera a biblioteca base da MIRACL. Todavia cada binário pode depender de diversos arquivos diferentes, que devem ser compilados manualmente para só então serem gerados os binários desejados.

Para solucionar estes problemas foi criado um script que realizar as seguintes tarefas:

- Compila a biblioteca base da MIRACL, gerando a biblioteca estática principal;
- Compila todos os arquivos de código fonte;
- Gera todos os arquivos executáveis a partir dos objetos gerados na compilação e montando-os junto com a biblioteca estática gerada.

Dessa forma é possível obter todos os executáveis executando apenas um comando.

3.1.3 Melhoria nos *outputs* das ferramentas para contagem de pontos

Para facilitar a integração das ferramentas empregadas nos algoritmos desenvolvidos foram realizadas diversas modificações e melhorias nas saídas do executáveis da MIRACL, permitindo que o algoritmo de contagem de pontos possa executá-los e capturar a saída que contém o resultado.

3.1.4 Scripts para contagem de pontos

Além do fato de que a biblioteca MIRACL não é amigável em relação ao seu processo de compilação existe também o problema de que seus binários requerem um alto grau de conhecimento técnico para utilizá-los. Para facilitar o uso dos executáveis da MIRACL foram criados diversos scripts que encapsulam os executáveis e, além de executá-los, realizam também outras tarefas. Os seguintes scripts foram criados:

- **Script para gerar Polinômios de Müller.** De todo o processo de geração de curvas elípticas esta é a etapa mais demorada. Este script, além de executar o binário propriamente dito, realiza outras tarefas:
  - Geração dos polinômios de forma paralela, utilizando o GNU Parallel;
  - Configuração dos nós para o caso da execução em diferentes computadores;
  - Concatenação dos resultados para gerar a lista final com os polinômios.

- **Pré-processamento de números primos.** Este script automaticamente carrega a lista de polinômios gerada anteriormente e, dado um número primo como parâmetro, executa o pré-processamento de seus polinômios;
- **Contagem de Pontos.** Informando os parâmetros da curva que terá sua ordem calculada, este script automaticamente carrega o arquivo com os dados pré-processados que foi gerado no passo anterior (ou então o gera se não existir), executa o executável *sea* da MIRACL para contar os pontos e salva o resultado em um arquivo.

3.2 Implementação e Execução dos Algoritmos

3.2.1 Geração de Polinômios de Müller

A geração dos Polinômios de Müller é realizada através do executável `mueller`. Sua execução realizada por intermédio do script citado na seção anterior. A Figura 3 mostra tempo de execução em minutos para cada primo até 521.

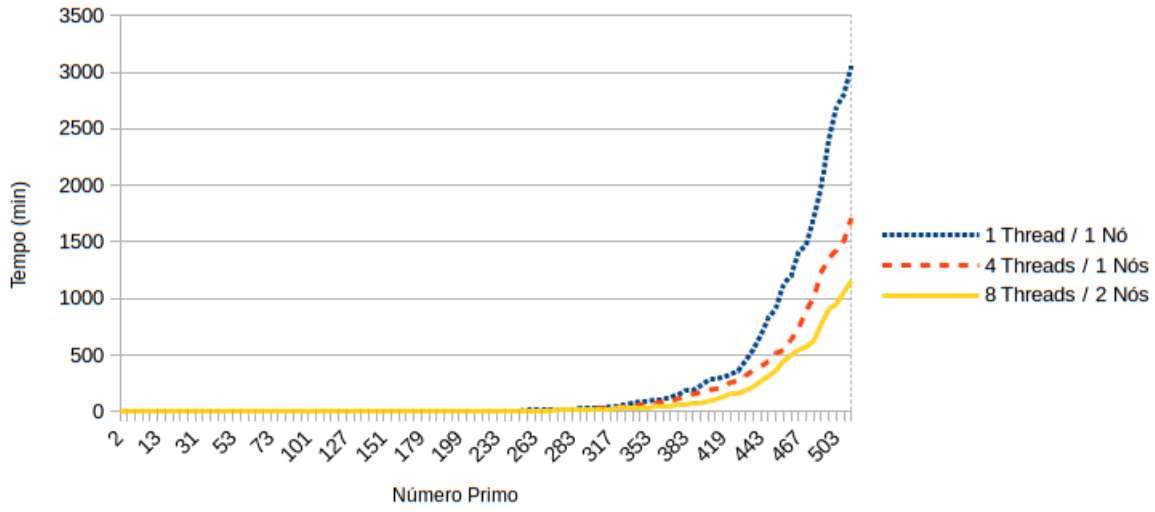

Figura 3: Tempo de execução do algoritmo para gerar Polinômios de Müller, comparando a velocidade utilizando diferentes números de *threads* (T) e nós (N).

3.2.2 Contagem de Pontos

A contagem de pontos é realizada através do executável `sea`, cuja execução é controlada por um script. A Figura 4 mostra o tempo de execução médio para a contagem de pontos utilizando tanto o algoritmo de Schoof quanto o algoritmo de Schoof-Elkins-Atkins. Não foi possível adquirir mais dados sobre o tempo de execução do algoritmo de Schoof pois seu executável passou a apresentar travamentos para curvas com parâmetro p maior que 320 *bits*.

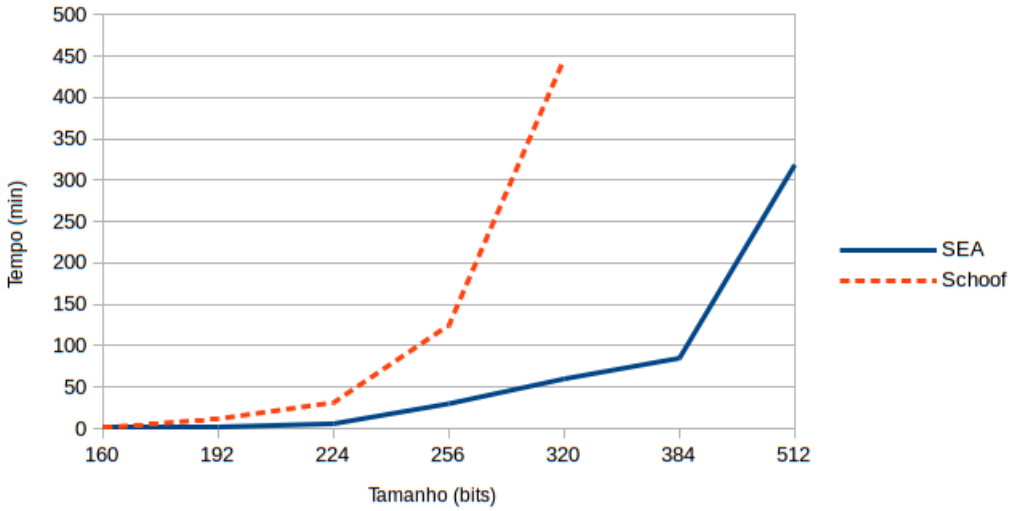

Figura 4: Tempo de execução média entre o algoritmo de Schoof e SEA em relação o tamanho do primo p.

3.2.3 Algoritmos Brainpool

Extração dos n bits mais a direita de um número (Algoritmo 2.1). A saída da execução do algoritmo é apresentado no Código 3.1 e sua implementação é apresentada no Apêndice A.

Código 3.1: Entrada e saída do Código 1

```
1 Input: s = FF 22 26 CD 8E 25 4B 5E F1 C0 CB 6A E4 D9 00 66 23 E8 4B DA
2        n = 80
3 Output: CB 6A E4 D9 00 66 23 E8 4B DA
```

Conversão de um vetor de bytes para um número inteiro (Algoritmo 2.2). A saída de execução do algoritmo é apresentado no Código 3.2 é apresentado no Apêndice A.

Código 3.2: Entrada e saída do Código 2

```
1 Input: s = 8D 00 79 F8 43 54 34 E1 43 AD DA 02 61 C0 20 96 E8 6A 8B 56
2 Output: 804978323752281613485744381124601214816980994902
```

Conversão de um número inteiro para um vetor de bytes (Algoritmo 2.3). A saída de execução do algoritmo é apresentado no Código 3.3 é apresentado no Apêndice A.

Código 3.3: Entrada e saída do Código 3

```
1 Input: 99999999999999999999999999999
2 Output: 01 43 1E 0F AE 6D 72 17 CA 9F FF FF FF
```

Atualização de uma seed (*semente*) (Algoritmo 2.4). A saída de execução do algoritmo é apresentado no Código 3.4 é apresentado no Apêndice D.

Código 3.4: Entrada e saída do Código 4

```
1 Input: 52 3F CE D3 D9 BC A6 D5
2 Output: 52 3F CE D3 D9 BC A6 D6
```

Inicialização (Algoritmo 2.5). A saída de execução do algoritmo é apresentado no Código 3.5 é apresentado no Apêndice E.

Código 3.5: Entrada e saída do Código 5

```
1 Input: 512
2 Output: v = 3, w = 192
```

Geração de números pseudo-aletórios (Algoritmo 2.6). A saída de execução do algoritmo é apresentado no Código 3.6 é apresentado no Apêndice B.

Código 3.6: Entrada e saída do Código 6

```
1 Input: s = 7F B0 55 71 68 32 A1 43 22 72 23 C1 99 B3 9D 02 A3 FE D4 03 ,
      v = 0, w = 320
2 Output: 3051081469519950502060536657487508821012954296 03
```

Geração de números primos (Algoritmo 2.7). A saída de execução do algoritmo é apresentado no Código 3.7 é apresentado no Apêndice C.

Código 3.7: Entrada e saída do Código 7

```
1  Input: L = 512, s = E1 01 3C E2 1B 5D DA 34 2E 77 23 9E 3D 37 46 40 87
      32 5B E7
2 Output: 772356886603345274343506969552059879425589594679
```

3.2.3.1 Tempo de execução

Para verificar o tempo de execução de cada algoritmo, cada um deles foi executado 100 vezes. Os parâmetros foram escolhidos de acordo com o pior caso possível. Os resultados obtidos são listados na Tabela 6.

Tabela 6: Tempo médio de execução dos algoritmos após 100 execuções.

Algoritmo	**Parâmetros**	**Média (ms)**	**Desvio Padrão (ms)**
2.1 (Apêndice A)	s: 8 bytes aleatórios; n: 40	1,704	0,205
2.2 (Apêndice A)	s: 20 bytes aleatórios	1,650	0,111
2.3 (Apêndice A)	k: 999999999999999999999999	1,626	0,059
2.4 (Apêndice D)	s: 20 bytes aleatórios	1,665	0,065
2.5 (Apêndice E)	L: 512	1,661	0,057
2.6 (Apêndice B)	s: 7F B0 55 71 68 32 A1 43 22 72 23 C1 99 B3 9D 02 A3 FE D4 03; v: 0; w: 320	2,879	0,897
2.7 (Apêndice C)	s: 8 bytes aleatórios; L: 512	15,827	11,04

3.3 Geração de curvas

O algoritmo de geração de curvas foi executado cinco vezes, tendo seu tempo de execução coletado em cada uma delas. Para todas as execuções foram utilizados os parâmetros referentes ao tamanho de cada curva assim como descritos na documentação Brainpool. Em cada uma das execuções são realizados os seguintes passos:

- Pré-processamento de polinômios para o número primo escolhido;
- Execução dos algoritmos da própria Brainpool;
- Cálculo da ordem da curva (cálculo da quantidade de pontos);
- Verificações de segurança;

Os tempos de execução são listados na Tabela 7. A Figura 5 possui os dados dessa tabela em forma de um gráfico de linha. Percebe-se que o tempo de execução do algoritmo está diretamente relacionado com o tempo de execução da contagem de pontos, apresentado na Figura 4.

Tabela 7: Tempo médio em minutos para a geração de parâmetros de domínio de uma curva elíptica utilizando os parâmetros especificados pela documentação da Brainpool para cada tamanho de curva.

#	**160**	**192**	**224**	**256**	**320**	**384**	**512**
1	2,0	4,1	17,3	43,2	108,4	129,6	400,4
2	2,9	3,4	8,9	46,9	105,9	122,2	389,2
3	2,7	3,7	13,0	47,0	106,2	138,4	401,3
4	1,4	2,7	13,6	38,3	106,9	113,6	370,3
5	2,2	2,4	16,1	38,6	106,4	141,1	384,7
Média	**2,2**	**3,3**	**13,8**	**42,8**	**106,8**	**129,0**	**389,2**

3.4 Discussão dos resultados

As melhorias realizadas na ferramentas utilizadas, além dos scripts criados, facilitam o desenvolvimento de soluções relacionadas não somente à curvas elípticas como diversas outras áreas da criptografia. Os passos necessários para se obter todos os executáveis da biblioteca MIRACL caíram de 14 para apenas 1.

A refatoração do algoritmo Brainpool o aproximou de um cenário real de implementação, onde trechos que são frequentemente repetidos são isolados para então serem reutilizados, tornando o algoritmo mais enxuto e de fácil compreensão, facilitando sua implementação utilizando outras linguagens e/ou bibliotecas algébricas.

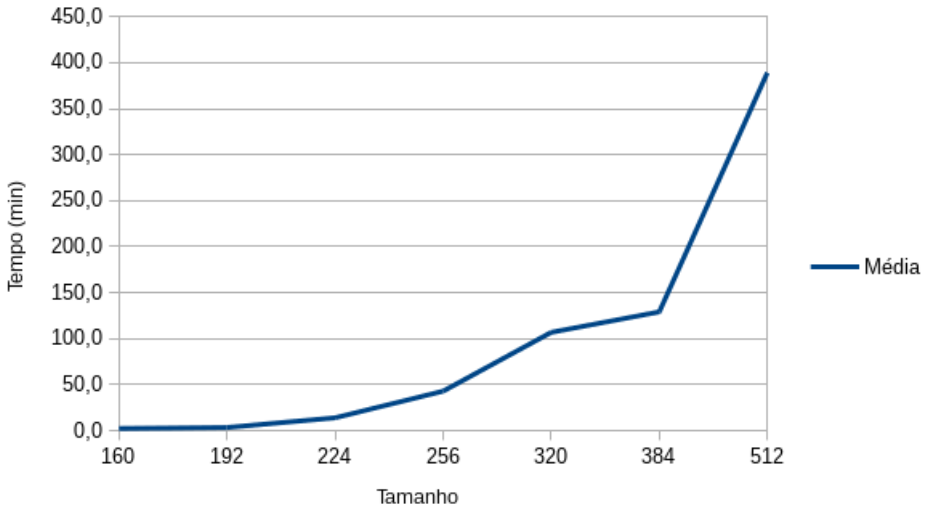

Figura 5: Visualização dos dados da Tabela 7.

O uso de paralelismo e distribuição de processamento para tarefas que exigem muito poder computacional mostrou-se uma necessidade, especialmente para o processo de contagem de pontos. Quando comparado com a execução em um único processo em um único nó (computador) o algoritmo de geração de Polinômios de Müller apresentou um ganho de aproximadamente 60% no desempenho quando paralelizado em 4 *threads* em um mesmo nó e de aproximadamente 90% quando paralelizado em 8 *threads* distribuídos em dois nós.

A contagem de pontos se mostrou eficiente e estável mesmo para curvas muito grandes. Dado que os equipamentos utilizados para executar o algoritmo não são especializados nesta tarefa, o tempo de execução de 5 horas para o pior caso pode ser considerado aceitável.

Por fim, o algoritmo de geração de curvas Brainpool foi completamente implementado e, alimentando o algoritmo com os mesmo parâmetros presentes em sua especificação original, gera as mesmas curvas, validando a implementação desenvolvida e também este trabalho.

4 Considerações Finais

O presente trabalho apresentou os principais conceitos matemáticos para o problema da geração de parâmetros de domínio para curvas elípticas, a implementação de um algoritmo para essa tarefa e seus algoritmos auxiliares, como o algoritmo de contagem de pontos.

Foram analisados os tempos de execução dos diferentes algoritmos o que leva a conclusão que em um algoritmo de geração de parâmetros de domínio o principal passo, e o que consome mais tempo, é o algoritmo de contagem de pontos. Apesar do custo inicial alto para calcular a lista dos Polinômios de Müller, o algoritmo de Schoof-Elkins-Atkins se mostrou o mais rápido, apresentando tempos de execução aceitáveis.

As melhorias apresentadas beneficiam não somente a execução do presente trabalho, como também qualquer indivíduo que venha a trabalhar com curvas elípticas e/ou com a biblioteca MIRACL.

Como principal contribuição deste trabalho fica a primeira implementação gratuita e *open source* do algoritmo Brainpool. Além da implementação do algoritmo Brainpool, também foi desenvolvido um *framework* de cálculo algébrico de alto nível, utilizando C++ e GNU MP, que permite realizar cálculos com números de tamanho arbitrário utilizando orientação a objetos.

4.1 Trabalhos Futuros

Apesar de ter servido ao propósito deste trabalho, a biblioteca MIRACL apresenta algumas deficiências em relação a sua escalabilidade e integração com outros sistemas. Como continuação deste trabalho seria interessante implementar o algoritmo de contagem de pontos dentro do próprio *framework* já desenvolvido, eliminando a dependência dos binários modificados da MIRACL.

Referências

ANSI. *Public Key Cryptography for the Financial Services Industry, The Elliptic Curve Digital Signature Algorithm (ECDSA)*. [S.l.], 2005. Citado na página 36.

CRANDALL, R.; POMERANCE, C. *Prime Numbers: A computational perspective*. 2. ed. [S.l.]: Springer, 2010. Citado na página 35.

DELFS, H.; KNEBL, H. *Introduction to Cryptography: Principles and Applications*. 2. ed. [S.l.]: Springer, 2007. Citado na página 38.

ECC-BRAINPOOL. *ECC Brainpool Standard Curves and Curve Generation*. [S.l.], 2005. Citado 3 vezes nas páginas 36, 43 e 46.

EICHER JODIE; OPOKU, Y. *Using the Quantum Computer to Break Elliptic Curve Cryptosystems*. [S.l.], 1997. Citado 2 vezes nas páginas 27 e 42.

ENISA. *Study on cryptographic protocols*. [S.l.], 2014. Citado 2 vezes nas páginas 17 e 40.

HALIM, S. *Competitive Programming*. 3. ed. [S.l.]: paperback, 2013. Citado na página 30.

IETF. *The MD5 Message-Digest Algorithm*. [S.l.], 1992. Citado na página 40.

IETF. *The Internet Key Exchange (IKE)*. [S.l.], 1998. Citado na página 36.

IETF. *The OAKLEY Key Determination Protocol*. [S.l.], 1998. Citado na página 36.

JOHNSON, D.; MENEZES, A.; VANSTONE, S. The elliptic curve digital signature algorithm (ecdsa). In: *International Journal of Information Security*. [S.l.: s.n.], 2001. v. 1, p. 36–63. Citado na página 39.

JOYE, M. Elliptic curves and side-channel analysis. *ST Journal of System Research*, v. 4, p. 283–306, 2003. Citado na página 41.

JúNIOR, E. A. da C. *Curvas Elipticas e Aplicacoes a Criptografia*. Dissertação (Mestrado) — Universidade de Brasília, Brasília, Brasil, 2005. Citado na página 34.

KAHN, D. *The Codebreakers – The Story of Secret Writing*. Revised sub edition. [S.l.]: Scribner, 1996. Citado na página 37.

KALISKI, B. *The Mathematics of the RSA Public-Key Cryptosystem*. [S.l.], 2006. Citado na página 27.

KOBLITZ, N. Elliptic curve cryptosystems. *Mathematics of Computation*, v. 83, p. 203–209, 1987. Citado na página 27.

KOCHER, P. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: *Advances in Cryptology — CRYPTO'96*. [S.l.: s.n.], 1996. p. 104–113. Citado 2 vezes nas páginas 35 e 41.

KOCHER, P.; JAFFE, J.; JUN, B. Differential power analysis. In: *Advances in Cryptology — CRYPTO'96*. [S.l.: s.n.], 1999. p. 388–397. Citado na página 41.

LAUTER, K. E.; STANGE, K. E. The elliptic curve discrete logarithm problem and equivalent hard problems for elliptic divisibility sequences. In: . [S.l.: s.n.], 2009. p. 309–327. Citado na página 39.

NIST. *Secure Hash Standard*. [S.l.], 2012. Citado na página 40.

NIST. *FIPS 186-4 - Digital Signature Standard (DSS)*. [S.l.], 2014. Citado 2 vezes nas páginas 36 e 39.

NIVEN, I.; ZUCKERMAN; S., H. *An Introduction to the Theory of Numbers*. 6. ed. [S.l.]: Wiley, 2014. Citado na página 29.

PROOS, J.; ZALKA, C. *Shor's discrete logarithm quantum algorithm for elliptic curves*. [S.l.], 2008. Citado na página 42.

RSA LABS. *RSA Cryptography Standard*. [S.l.], 2012. Citado na página 39.

SECG. *Standards for Efficient Cryptography*. 1. ed. [S.l.], 2009. Citado na página 39.

SECG. *Recommended Elliptic Curve Domain Parameters*. 2. ed. [S.l.], 2010. Citado na página 36.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *35th Annual Symposium on Foundations of Computer Science*, 1994. Citado na página 42.

SHOUP, V. A proposal for an iso standard for public key encryption. In: . [S.l.: s.n.], 2001. Citado na página 39.

SILVERMAN, J. H. *The Arithmetic of Elliptic Curve*. 2. ed. [S.l.]: Springer, 2009. Citado na página 33.

SINGH, S. *The Code Book: The Science of Secrecy from Ancient Egypt to Quantum Cryptography*. Reprint edition. [S.l.]: Anchor, 2000. Citado na página 27.

THOMAS, G. B. *Cálculo*. 10. ed. [S.l.]: Addison Wesley, 2004. Citado na página 32.

YANG, S. Y. *Quantum Attacks on Public-Key Cryptosystems*. 2. ed. [S.l.]: Springer, 2013. Citado na página 42.

Apêndices

APÊNDICE A – conversion.cpp/.h

conversion.h

```
#ifndef _CONVERSION_H
#define _CONVERSION_H

#include "str.h"

#include <gmp.h>
#include <gmpxx.h>

/* Expande os bits da string s para um inteiro */
mpz_class* expand(const str* s);

/* Expande os bits do inteiro t para uma string */
str* expand(const mpz_class* t);

/* Extrai os n primeiro bits da string s em uma nova string */
str* extract_right_bits(const str* s, unsigned int n);

#endif
```

conversion.cpp

```
#include "conversion.h"

#include <cmath>

#ifdef DEBUG
#include <iostream>
#endif

mpz_class* expand(const str* s){
  mpz_t z;
  mpz_init2(z, s->length * 8);
  mpz_import(z, s->length, 1, sizeof(unsigned char), 0, 0, s->data); //
      verificar a ordem dos MSB e LSB

  mpz_class* r = new mpz_class(z);
  return r;
}

str* expand(const mpz_class* t) {
  size_t nbits = mpz_sizeinbase(t->get_mpz_t(), 2);
  int nbytes = (int) ceil((double) nbits / 8.0);
```

```
21
22 #ifdef DEBUG
23   std::cout << "nbits:  " << nbits << std::endl;
24   std::cout << "nbytes: " << nbytes << std::endl;
25 #endif
26   str* z = new str(nbytes);
27
28   mpz_export(z->data, NULL, 1, sizeof(unsigned char), 0, 0, t->get_mpz_t
       ());
29
30   return z;
31 }
32
33 str* extract_right_bits(const str* s, unsigned int n) {
34     mpz_class t;
35     mpz_ui_pow_ui(t.get_mpz_t(), 2, n);
36
37   mpz_class mask(t - 1);
38   //mpz_class nbits(*n);
39   mpz_class* tmp = expand(s);
40   mpz_class* r = new mpz_class();
41
42   mpz_and(r->get_mpz_t(), tmp->get_mpz_t(), mask.get_mpz_t());
43   return expand(r);
44 }
```

APÊNDICE B – gen.cpp/.h

gen.h

```
#ifndef _GEN_H_
#define _GEN_H_

#include <gmp.h>
#include <gmpxx.h>

#include "str.h"

mpz_class* gen_number(const str* h, const str* seed, unsigned int v,
    unsigned int w);

#endif
```

gen.cpp

```
#include "gen.h"

#include "conversion.h"
#include "sha1.h"

mpz_class* gen_number(const str* h, const str* seed, unsigned int v,
    unsigned int w) {
    str* h_0 = extract_right_bits(h, w);
    mpz_class* z = expand(seed);

    str* hi = new str[v + 1];
    hi[0] = *h_0;


    for (int i = 1; i < v + 1; i++) {
        mpz_class z_plus_i((*z) + i), p, r;
        mpz_ui_pow_ui(p.get_mpz_t(), 2, 160);
        mpz_mod(r.get_mpz_t(), z_plus_i.get_mpz_t(), p.get_mpz_t()); //
            r = (z+i) % 2^160

        str* si = expand(&r);
        str* tmp = sha1(si);
        hi[i] = *tmp;
    }

    delete z;

```

```
26      h = str_concat(hi, v + 1);
27      delete[] hi;
28
29      mpz_class* c = expand(h);
30      delete h;
31
32      return c;
33  }
```

APÊNDICE C – nonce.cpp/.h

prime.h

```
#ifndef _PRIME_H_
#define _PRIME_H_

#include "str.h"

#include <gmp.h>
#include <gmpxx.h>

struct _Prime {
    mpz_class* value;
    str* seed;

    _Prime(const mpz_class* p, const str* s) {
        seed = new str(s);
        value = new mpz_class();
        mpz_set(value->get_mpz_t(), p->get_mpz_t());
    }

    ~_Prime() {
        delete value;
        delete seed;
    }
};

typedef struct _Prime Prime;

Prime* prime_gen(unsigned int bits, const str* seed = nullptr);

#endif
```

prime.cpp

```
#include "prime.h"

#include "hex.h"
#include "nonce.h"
#include "sha1.h"
#include "conversion.h"
#include "seed.h"
#include "gen.h"

#include <iostream>
```

```
11 #include <cmath>
12 #include <sstream>
13 //#include <openssl/sha.h>
14
15 //#define DEBUG
16
17 void log(std::string s) {
18 #ifdef DEBUG
19   std::cout << s << std::endl;
20 #endif
21 }
22
23 const std::string SHIFT = "          ";
24
25 bool check_range(mpz_class* p, unsigned int* L) {
26   mpz_t low, high, tmp;
27
28     // calcular low
29     mpz_init(low);
30     mpz_init(tmp);
31     mpz_ui_pow_ui(tmp, 2, *L - 1); // (2^(L-1))
32     mpz_sub_ui(low, tmp, 1); // 2^(L-1) - 1
33
34     // calcular high
35     mpz_init(high);
36     mpz_ui_pow_ui(high, 2, *L);
37
38     int higher_than_low;
39     int lower_than_high;
40
41     higher_than_low = mpz_cmp(low, p->get_mpz_t());
42     lower_than_high = mpz_cmp(high, p->get_mpz_t());
43
44     mpz_clear(tmp);
45     mpz_clear(high);
46     mpz_clear(low);
47
48   return higher_than_low == -1 && lower_than_high == 1;
49 }
50
51 bool check_primality(mpz_class* p) {
52   int is_prime;
53   is_prime = mpz_probab_prime_p(p->get_mpz_t(), 25);
54   return is_prime > 0;
55 }
56
57 Prime* prime_gen(unsigned int L, const str* s) {
```

```
58
59  bool should_update_seed = false;
60
61  // Passo 0
62  if (L == 0 ) {
63    return nullptr;
64  }
65
66  unsigned int v = (int) floor((double) (L - 1) / 160.0);
67  unsigned int w = L - 160 * v;
68
69    //std::cout << "v: " << v << std::endl;
70    //std::cout << "w: " << w << std::endl;
71
72    str* seed;
73  if (s != nullptr) {
74    seed = new str(s);
75  }
76    else {
77        seed = nonce_gen(160/8);
78    }
79
80    //std::cout << "Seed: " << to_hex_string(seed) << std::endl;
81    //std::cout << "Bits: " << L << std::endl;
82
83  while (true) {
84    if (should_update_seed) {
85      str* tmp = update_seed(seed);
86      delete seed;
87      seed = tmp;
88    }
89        // Step 2
90    str* h = nullptr;
91        h = sha1(seed);
92
93        // Steps 3 to 7
94        mpz_class* c = gen_number(h, seed, v, w);
95
96        // Step 8
97        mpz_class* p = new mpz_class();
98
99        do {
100           mpz_nextprime(p->get_mpz_t(), c->get_mpz_t());
101           mpz_set(c->get_mpz_t(), p->get_mpz_t());
102       } while(mpz_congruent_ui_p(p->get_mpz_t(), 3, 4) == 0);
103
104   delete c;
```

```
105
106         // Step 9
107     if (!check_range(p, &L)) {
108       //std::cout << "Passo 9 falhou. Atualizando seed e reiniciando."
              << std::endl;
109       delete p;
110       should_update_seed = true;
111       continue;
112     }
113
114         // Step 10
115     if (!check_primality(p)) {
116       //std::cout << "Passo 10 falhou. Atualizando seed e reiniciando."
              << std::endl;
117       delete p;
118       should_update_seed = true;
119       continue;
120     }
121         Prime* prime = new Prime(p, seed);
122
123     delete seed;
124     delete p;
125
126         return prime;
127   }
128
129   return nullptr;
130 }
```

APÊNDICE D – seed.cpp/.h

seed.h

```
#ifndef _SEED_H
#define _SEED_H

#include "str.h"

str* update_seed(const str* s);

#endif
```

seed.cpp

```
#include "seed.h"

#include "conversion.h"

#include <gmp.h>
#include <gmpxx.h>
#include "hex.h"
//#define DEBUG

#ifdef DEBUG
#include <iostream>
#endif

str* update_seed(const str* s) {
  // expandir os bits de s como um inteiro z
  mpz_class* z = expand(s);
  mpz_class z_plus_1((*z)+1);

#ifdef DEBUG
  std::cout << "z:   " << z << std::endl;
  std::cout << "z+1: " << z_plus_1 << std::endl;
#endif

  // calcular 2^160
  mpz_class k(0);
  mpz_ui_pow_ui(k.get_mpz_t(), 2, 160);

#ifdef DEBUG
  std::cout << "2^160: " << k<< std::endl;
#endif

```

```
32   // calcular (z+1) mod 2^160
33   mpz_class tmp;
34   mpz_mod(tmp.get_mpz_t(), z_plus_1.get_mpz_t(), k.get_mpz_t());
35
36 #ifdef DEBUG
37   std::cout << "tmp: " << tmp<< std::endl;
38 #endif
39
40   // expandir os bits the tmp como uma string t
41   str* t = expand(&tmp);
42
43
44 #ifdef DEBUG
45   std::cout << "seed: " << to_hex_string(t) << std::endl;
46 #endif
47
48   return t;
49 }
```

APÊNDICE E – pre.cpp/.h

pre.h

```
#include <utility>

std::pair<unsigned int, unsigned int> run_5();
```

pre.cpp

```
#include "pre.h"


std::pair<unsigned int, unsigned int> run_5() {
    int L = 512;
    unsigned int v = (int) floor((double) (L - 1) / 160.0);
    unsigned int w = L - 160 * (v - 1);

    std::pair<unsigned int, unsigned int>  res(v, w);

    return res;
}
```

APÊNDICE F – weierstrass.cpp/.h

weierstrass.h

```
1 #ifndef _WEIERSTRASS_H_
2 #define _WEIERSTRASS_H_
3
4 #include <gmp.h>
5 #include <gmpxx.h>
6 #include <iostream>
7
8 class Point {
9 public:
10     Point();
11     Point(const mpf_class* a_x, const mpf_class* a_y);
12     Point(const double a_x, const double a_y);
13     ~Point();
14
15     mpf_class* get_x() const;
16     mpf_class* get_y() const;
17
18     friend std::ostream& operator<<(std::ostream& os, const Point& p);
19     friend bool operator== (const Point& a, const Point& b);
20
21 private:
22     mpf_class* x;
23     mpf_class* y;
24 };
25
26 class WeierstrassEquation {
27 public:
28     WeierstrassEquation(const mpf_class* a, const mpf_class* b);
29     WeierstrassEquation(const long a, const long b);
30     WeierstrassEquation(const double a, const double b);
31     ~WeierstrassEquation();
32
33     mpf_class* y(const mpf_class* x);
34     mpf_class* j_invariant();
35     Point* sum_points(const Point* point_a, const Point* point_b =
           nullptr);
36     Point* mul_point(const Point* point, const mpz_class* k);
37
38     friend std::ostream& operator<<(std::ostream& os, const
           WeierstrassEquation& eq);
39
```

```
40 private:
41     mpf_class* a;
42     mpf_class* b;
43 };
44
45 #endif
```

weierstrass.cpp

```
1  #include "weierstrass.h"
2
3  /*******************************************
4   *                 Point                   *
5   *******************************************/
6
7  using namespace std;
8
9  Point::Point() {
10     x = new mpf_class(0.0);
11     y = new mpf_class(0.0);
12 }
13
14 Point::Point(const mpf_class* a_x, const mpf_class* a_y) {
15     x = new mpf_class(*a_x);
16     y = new mpf_class(*a_y);
17 }
18
19 Point::Point(const double a_x, const double a_y) {
20     x = new mpf_class(a_x);
21     y = new mpf_class(a_y);
22 }
23
24 Point::~Point() {
25     delete x;
26     delete y;
27 }
28
29 mpf_class* Point::get_x() const {
30     return x;
31 }
32
33 mpf_class* Point::get_y() const {
34     return y;
35 }
36
37 std::ostream& operator<<(std::ostream& os, const Point& p) {
38     os << "[x: " << *(p.x) << "; y: " << *(p.y) << "]";
39     return os;
```

```
40 }
41
42 bool operator== (const Point& a, const Point& b) {
43     bool x = *(a.x) == *(b.x);
44     bool y = *(a.y) == *(b.y);
45     return x && y;
46 }
47
48 /*****************************************
49  *           WeierstrassEquation         *
50  *****************************************/
51
52 WeierstrassEquation::WeierstrassEquation(const mpf_class* a, const
       mpf_class* b) {
53     this->a = new mpf_class(*a);
54     this->b = new mpf_class(*b);
55 }
56
57 WeierstrassEquation::WeierstrassEquation(const long a, const long b) {
58     this->a = new mpf_class(a);
59     this->b = new mpf_class(b);
60 }
61
62 WeierstrassEquation::WeierstrassEquation(const double a, const double b)
       {
63     this->a = new mpf_class(a);
64     this->b = new mpf_class(b);
65 }
66
67 WeierstrassEquation::~WeierstrassEquation() {
68     delete this->a;
69     delete this->b;
70 }
71
72 mpf_class* WeierstrassEquation::y(const mpf_class* x) {
73     mpf_class ta, tb, tc;
74
75     mpf_pow_ui(ta.get_mpf_t(), x->get_mpf_t(), 3);
76     tb = *(this->a) * (*x);
77     tc = *(this->b);
78
79     mpf_class tmp = ta + tb + tc;
80     mpf_class* result = new mpf_class();
81
82     mpf_sqrt(result->get_mpf_t(), tmp.get_mpf_t());
83
84     return result;
```

```
85 }
86
87 mpf_class* WeierstrassEquation::j_invariant() {
88     mpf_class ta, tb;
89
90     mpf_pow_ui(ta.get_mpf_t(), a->get_mpf_t(), 3);
91     mpf_pow_ui(tb.get_mpf_t(), b->get_mpf_t(), 2);
92
93     ta = 4 * ta;
94     tb = 27 * tb;
95
96     mpf_class* result = new mpf_class(1728 * (ta / (ta + tb)));
97
98     return result;
99 }
100
101 Point* WeierstrassEquation::sum_points(const Point* point_a, const Point
      * point_b) {
102     Point pa = *point_a;
103     Point pb;
104     if(point_b == nullptr) {
105         pb = pa;
106     }
107     else {
108         pb = *point_b;
109     }
110
111     mpf_class x_a = *(pa.get_x());
112     mpf_class y_a = *(pa.get_y());
113     mpf_class x_b = *(pb.get_x());
114     mpf_class y_b = *(pb.get_y());
115     mpf_class m = 0;
116
117     if(pa == pb) {
118         mpf_class k = 3 * (x_a * x_a) + *(this->a);
119         mpf_class l = 2 * y_a;
120         m = k / l;
121     }
122     else {
123         mpf_class k = y_b - y_a;
124         mpf_class l = x_b - x_a;
125         m = k / l;
126     }
127
128     mpf_class x_c = (m * m) - 2 * x_a;
129     mpf_class y_c = m * (x_a - x_c) - y_a;
130
```

```
131     Point* r = new Point(&x_c, &y_c);
132 
133     return r;
134 }
135 
136 Point* WeierstrassEquation::mul_point(const Point* point, const
        mpz_class* k) {
137     return nullptr;
138 }
139 
140 std::ostream& operator<<(std::ostream& os, const WeierstrassEquation& eq
        ) {
141     if(*(eq.a) == 0 && *(eq.b) == 0) {
142         os << "[y^2 = x^3]";
143     }
144 
145     else if(*(eq.a) == 0) {
146         os << "[y^2 = x^3 + " << *(eq.b) << "]";
147     }
148 
149     else if(*(eq.b) == 0) {
150         os << "[y^2 = x^3 + " << *(eq.a) << "x]";
151     }
152 
153     else {
154         os << "[y^2 = x^3 + " << *(eq.a) << "x + " << *(eq.b) << "]";
155     }
156 
157     return os;
158 }
```