FUNDAÇÃO DE ENSINO “EURÍPIDES SOARES DA ROCHA”
CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA – UNIVEM
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

GUILHERME RODRIGUES BILAR

IMPLEMENTAÇÃO DO ESQUEMA TOTALMENTE HOMOMÓRFICO SOBRE NÚMEROS INTEIROS UTILIZANDO PYTHON COM COMPRESSÃO DE CHAVE PÚBLICA

MARÍLIA
2014

GUILHERME RODRIGUES BILAR

IMPLEMENTAÇÃO DO ESQUEMA TOTALMENTE HOMOMÓRFICO SOBRE NÚMEROS INTEIROS UTILIZANDO PYTHON COM COMPRESSÃO DE CHAVE PÚBLICA

Trabalho de Curso apresentado ao Curso de Bacharelado em Ciência da Computação da Fundação de Ensino "Eurípides Soares da Rocha", mantenedora do Centro Universitário Eurípides de Marília – UNIVEM, como requisito parcial para obtenção do grau de Bacharel em Ciência da Computação.

Orientador
Profº: Fábio Dacêncio Pereira

MARÍLIA
2014

CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Guilherme Rodrigues Bilar

Implementação do esquema totalmente homomorfico sobre número inteiros utilizando Python com compressão de chave pública

Banca examinadora da monografia apresentada ao Curso de Bacharelado em Ciência da Computação do UNIVEM/F.E.E.S.R., para obtenção do Título de Bacharel em Ciência da Computação.

Nota: 10 (Dez)

Orientador: Fábio Dacêncio Pereira

1°. Examinador: Rodolfo Barros Chiaramonte

2°. Examinador: Bruno Marques dos Santos

Marília, 02 de dezembro de 2014.

BILAR, Guilherme Rodrigues

Implementação do Esquema Totalmente Homomórfico Sobre Números Inteiros Utilizando Python com Compressão de Chave Pública / Guilherme Rodrigues Bilar; orientador: Profº. MSc. Dr. Fábio Dacêncio Pereira, SP: [s.n.], 2014.

66 folhas

Monografia (Bacharelado em Ciência da Computação): Centro Universitário Eurípides de Marília.

Dedico este trabalho à meus pais me ajudaram a moldar-me no homem que sou hoje.

AGRADECIMENTOS

Agradeço a minha família, minha mãe Andrea, meu pai David por terem proporcionado a realização de dessa graduação, meu professor orientador Fabio Dacêncio Pereira por ter estimulado e fornecido as bases de minha pesquisa, tanto de iniciação científica quanto de trabalho de conclusão de curso, jamais teria chegado tão longe sem o seu apoio. Um agradecimento especial a Luan Cardoso dos Santos, meu amigo e colega de pesquisa, por sua amizade e apoio durante a minha formação superior, agradeço também aos meus amigos Rodrigo Tonon e Kathia Mascarenhas pelo estimulo, momentos de descontração e brincadeiras, tão importantes em momentos da vida como esse.

“Fascinating” - Spock

RESUMO

Foi implementado o esquema totalmente homomórfico com compressão de chave pública (DGVH sobre inteiros) proposto por Jean-Sébastian Coron, Avradip Mandal, David Naccache e Mehdi Tibouchi, que foi publicado na conferencia CRYPTO 2012, este mesmo esquema pode ser comparado com o esquema totalmente homomórfico de Gentry, que se trata de um esquema totalmente homomórfico mais simples, contudo essa simplicidade vem ao custo de que sua chave pública possui um tamanho estimado de 2 Gigabytes para o seu parâmetro de segurança mais elevado, o que de acordo com Coron et al, torna inviável a aplicação em sistemas práticos. O esquema totalmente homomórfico DGVH com chave pública comprimida diminui o tamanho da chave pública gerada para 18 Megabytes utilizando um gerador de números pseudoaleatórios. Para fazê-lo foi utilizada linguagem de programação Python, contando com a biblioteca de matemática e teoria numérica GMPY2.

Palavras-chave: Criptografia; Homomorfismo; Pós-Quântica; Inteiros; Proposta de Aceleração;

ABSTRACT

Has been implemented the fully homomorphic scheme with shorter key (DGVH with shorter key) proposed by Jean-Sébastian Coron, Avradip Mandal , David Naccache and Mehdi Tibouchi , which was published in the conference CRYPTO 2012. This same scheme can be compared with the Gentry's fully homomorphic scheme, being it a simpler fully homomorphic scheme, however this simplicity comes at the cost of the public key having an estimated size of 2 Gigabytes, which according to Coron et al, makes it impossible to use in practical systems. The DGVH scheme over integers with compressed public key decreases the size of the generated public key 18 Megabytes, using random numbers generators. To do so, the Python programming language was used, with the library of mathematics and number theory GMPY2.

Keywords: cryptography; homomorphic; post quantum; integer; propose of acceleration;

LISTA DE ILUSTRAÇÕES

Figura 1: Processo de compartilhamento e funcionamento de chaves simétricas 23
Figura 2: Processo de compartilhamento e funcionamento de chaves assimétricas 24
Figura 3: Processo de "ciphertext refresh" 30
Figura 4: Classificação dos Esquemas Totalmente Homomórficos 31
Figura 5: Esquemas Totalmente Homomórficos foram pesquisados durante o trabalho 42
Figura 6: Comparação de desempenho de tempo das primitivas com Toy 51
Figura 7: Comparação de desempenho de tempo das primitivas com Small 52
Figura 8: Comparação de desempenho de tempo das primitivas com Medium 53
Figura 9: Comparação do desempenho das primitivas entre os parâmetros de segurança das implementações 54
Figura 10: Comparação entre uma arquitetura multicore de uma CPU e a arquitetura many-cores de uma GPGPU 55
Figura 11: Comparação de diferentes arquiteturas 56
Figura 12: Esquematização da proposta de paralelismo 57

LISTA DE TABELAS

Tabela 1: Tempos de execução obtidos por Coron...40
Tabela 2: Tempo de execução em diferentes plataformas...41
Tabela 3: Parâmetros de Teste...49
Tabela 4: Tempo de execução das primitivas implementadas 50
Tabela 5: Relação de eficiência das primitivas 58

LISTA DE CÓDIGOS

Código Fonte 1: Gerador de Números Aleatórios 43
Código Fonte 2: Geração de Elementos Públicos 44
Código Fonte 3: Geração do Vetor S 44
Código Fonte 4: Geração do Elemento u1 44
Código Fonte 5: Encriptação da Chave 45
Código Fonte 6: Geração das Chaves 45
Código Fonte 7: Primitiva de Encrypt 46
Código Fonte 8: Primitiva de Expand 46
Código Fonte 9: Primitiva de Decrypt 47
Código Fonte 10: Funções Homomórficas 47

LISTA DE ABREVIATURAS E SIGLAS

FHE	Criptografia totalmente homomórfica (Inglês: *Fully homophic Encryption*)
SHE	Criptografia parcialmente homomórfica (Inglês: *Somewhat homomorphic encryption*)
LWE	Sigla em inglês para *Learning with errors*, uma classe de problemas matemáticos baseado em aprendizagem de maquina.
RWLW	Sigla em inglês para *Ring Learning with errors*,
MDC	Máximo divisor comum
MPZ	Sigla em inglês para inteiro de precisão múltipla
LLVM	Um projeto de infraestrutura para compilador, para otimização de código. Originalmente, acrônimo para *Low Level Virtual Machine*, sigla em inglês para máquina virtual de baixo nível
CPA	Do inglês *chosen-plaintext attacks*
FFT	Do inglês *Fast Fourier Transformation*
GPGPU	GPU's de propósito geral

SUMÁRIO

INTRODUÇÃO 16

1 CAPÍTULO 1: FUNDAMENTAÇÃO TEÓRICA 20

1.1 Circuitos Booleanos e Algébricos 20

1.2 Álgebra Abstrata 20

1.3 Reticulados 21

1.4 Criptografia Simétrica 22

1.5 Criptografia Assimétrica 23

1.6 Criptografia Parcialmente Homomórfica 24

1.7 Criptografia Totalmente Homomórfica 25

1.7.1 Segurança Contra Ataques 27

1.7.2 Ruído e Bootstraping 28

1.8 Classificação dos Esquemas Totalmente Homomórficos 30

1.8.1 Esquemas Totalmente Homomórficos Baseados em Reticulados 31

1.8.2 Esquemas Totalmente Homomórficos Baseados em Números Inteiros 32

1.8.3 Esquemas Totalmente Homomórficos Baseados em LWE 32

2 CAPÍTULO 2: HOMOMORFISMO COMPLETO SOBRE NÚMEROS INTEIROS 34

2.1 Primitivas do DGHV 35

2.1.1 KeyGen(λ) 35

2.1.2 Encrypt ($\mathbf{pk}, \boldsymbol{m} \in \{\mathbf{0}, \mathbf{1}\}$) 35

2.1.3 Evaluate($\mathbf{pk}, \boldsymbol{C}, \boldsymbol{c1}, \ldots, \boldsymbol{ct}$) 35

2.1.4 Decrypt (sk, c) 36

2.2 DGHV com compressão de chave e mudança de módulo 36

2.2.1 KeyGen($\mathbf{1\lambda}$) 36

2.2.2 Encrypt (*pk*, $\boldsymbol{m} \in \{\mathbf{0}, \mathbf{1}\}$) 37

2.2.3 Add($\boldsymbol{pk}, \boldsymbol{c1}*, \boldsymbol{c2}*$) 38

2.2.4 Mult($\boldsymbol{pk}, \boldsymbol{c1}*, \boldsymbol{c2}*$) 38

2.2.5 Expand$\boldsymbol{pk}, \boldsymbol{c}*$ 38

2.2.6 Decrypt$\boldsymbol{sk}, \boldsymbol{c}*, \boldsymbol{z}$ 38

2.2.7 Recrypt$\boldsymbol{pk}, \boldsymbol{c}*, \boldsymbol{z}$ 38

3 CAPÍTULO 3: PESQUISA E DESENVOLVIMENTO ... 40
3.1 Trabalhos Correlatos ... 40
3.2 Pesquisa Cientifica ... 41
3.3 Ferramentas ... 42
3.4 Implementação ... 42
3.4.1 Gerador de Números Aleatórios ... 43
3.4.2 Função Geradora de Chaves Assimétricas ... 43
3.4.3 Função de Encriptação ... 46
3.4.4 Função de Expansão ... 46
3.4.5 Função de Decriptação ... 47
3.4.6 Funções de Cálculo Homomórfico ... 47
4 CAPÍTULO 4: RESULTADOS ... 49
4.1 Métricas e Testes ... 49
4.1.1 Parâmetros de Teste ... 49
4.2 Resultados Finais ... 49
4.3 Trabalhos Futuros ... 54
4.3.1 Proposta de paralelismo ... 54
5 CAPÍTULO 5: CONCLUSÕES ... 57
5.1 Conclusões ... 57
5.2 Publicações ... 58
REFERÊNCIAS ... 58
Apêndice A: CODIGO DO ESQUEMA TOTALMENTE HOMOMORFICO ... 61
Apêndice B: TESTE DE POTENCIAÇÃO ... 65
Apêndice C: CONFIGURAÇÃO DO AMBIENTE ... 66

INTRODUÇÃO

A criptografia moderna tem como base problemas matemáticos relacionados a teoria de números, exemplo disto é o algoritmo RSA que usa a fatoração de números inteiros em números primos para a geração de chaves assimétricas (RIVEST et al., 1978) outro exemplo disso é o uso de logaritmos discretos na criptografia de curvas elípticas (MILLER, 1985), entretanto, com o surgimento da teoria dos computadores quântico e do algoritmo de Shor, esse tipo de solução torna-se vulnerável a ataques quânticos (SHOR, 1994). Para proteger dados, até mesmo da criptoanálise quântica houve o surgimento de uma nova área dentro da segurança da informação, chamada de Criptografia Pós-Quântica, que estuda a criação de algoritmos criptográficos resistentes a ataques quânticos, tendo como base outros problemas matemático-computacionais, como, por exemplo, algoritmos baseados em *hash*, em reticulados e em códigos. O tempo de ataque para todos esses algoritmos é exponencial, mesmo para um computador quântico (BERNSTEIN et al, 2008).

Entre características oferecidas pela a criptografia pós-quântica destaca-se o homomorfismo, onde um esquema totalmente homomórfico oferece a capacidade de manipularmos informação cifrada, de forma a realizar operações com as mesmas, tanto soma quanto multiplicação, sendo possível manter a integridade dos dados e garantir sua confidencialidade.

No que se refere a aplicações, o homomorfismo pode ser implantado no âmbito da nuvem, onde se tem três categorias distintas que as definem: Software como serviço (*SaaS – Software as a Service*), Plataforma como um serviço (*PaaS – Plataform as a Service*) e Infraestrutura como um serviço (*IaaS - Infrastrucute as a Service*), nestes três modelos de nuvem tem-se a presença de mecanismos criptográficos como o TCB (*Trusted Computer Base*) que provem o gerenciamento de chaves e a criação de canais de comunicação seguros entre cliente e servidor, no entanto, tais métodos de segurança apenas protegem os dados do ambiente externo, sendo que assim o servidor tem acesso total aos dados submetidos a ele, o que pode ser indesejável quando se trata de dados sigilosos, como por exemplo dados médicos sigilosos, dados militares ou econômicos, entre outros.

Tem-se então a necessidade da criptografia como serviço (*CaaS*), essa podendo ser aplicada em qualquer um tipos de nuvem tem a capacidade de proteger os dados até mesmo

do servidor em que estão hospedados, isso sendo possível graças ao homomorfismo, pode-se citar como possíveis aplicações de CaaS as seguintes:

- Mecanismo de buscas criptografados: Este permite realizar buscas na internet sem revelar informações sobre o que está sendo pesquisado;
- Banco de dados criptografados: Neste a nuvem manipula dados criptografados cabendo ao usuário decripta-los (BONEH, 2013);
- Computação sobre dados encriptados: Este permite processamento sobre dados encriptados, portanto o servidor que o processa não tem acesso aos dados que está processando.

A teoria do homomorfismo secreto (*privacy homomorphisms*) surgiu inicialmente em 1978, proposto por Rivest, Adleman e Dertouzos (RIVEST et al, 1978), este provera os conceitos básicos e as regras para que o homomorfismo fosse possível, muitos estudos foram realizados com base neste trabalho, no entanto, o primeiro esquema totalmente homomórfico foi apresentado por Gentry em 2009 (GENTRY, 2009), utilizando de reticulados, este suportava uma quantidade ilimitada de somas e multiplicações. A limitação quebrada foi a de que, a cada operação o ruído presente nas cifras se acumulava até que o processo de decriptação não conseguia mais retornar o dado original. Gentry então demostrou uma técnica para se reduzir o processo de *decrypt* de dados, sendo possível representá-lo como um polinômio de baixo grau nos bits do texto cifrado e da chave privada, dessa forma, permitindo que o esquema pudesse avaliar o seu próprio circuito de decriptação de forma homomórfica, e, como resultado, diminuindo o ruído apresentado no dado cifrado.

Com isso, o esquema de Gentry pôde executar computações arbitrarias nos dados cifrados, e, abriu caminho para que suas ideias fossem aplicadas a outros esquemas parcialmente homomórficos com o intuito de produzir esquemas FHE (do inglês: *Fully Homomorphic Encrypt;* do português: Esquemas Totalmente Homomórficos).

OBJETIVOS ESPECÍFICOS

Este projeto visa pesquisar e implementar primitivas criptográficas da variante do esquema totalmente homomórfico DGHV proposta por Coron (CORON et al, 2012), essa prevê uma redução no tamanho das chaves públicas geradas pelo esquemas totalmente homomórfico bem como melhorias e alterações funcionais nas primitivas *Decrypt* e *Encrypt*, afim de finalmente propor um modelo de processamento paralelizado em GPU.

- Estudar os conceitos básicos da criptografia homomórfica e consequentemente o esquema totalmente homomórfico DGHV;
- Estudar a variante do DGHV proposta por Coron;
- Pesquisar implementações e trabalhos correlatos;
- Implementar as primitivas criptográficas;
- Comparar resultados e desempenho com trabalhos correlatos;
- Estudar os conceitos e arquitetura geral das GPUs;
- Propor um modelo de implementação paralela para GPU.

METODOLOGIA

O presente projeto pode ser divido em 4 partes sendo essas, Estudo Bibliográfico, Estudo da Ferramenta e seus Componentes, Implementação e Experimentação, e Validação e Verificação.

- **Estudo Bibliográfico**: Consiste em realizar um levantamento de trabalhos similares ou correlatos junto de um estudo profundo dos esquemas homomórficos relevantes relacionados ou trabalho, bem como o estudo de todo o fluxo de dados e dependências dos mesmos dentro do esquema homomórfico.
- **Estudo da ferramenta e seus componentes**: Consiste de uma série de atividades didáticas voltadas a aprendizagem das bibliotecas e ferramentas necessárias para a implementação do esquema homomórfico.
- **Implementação e Experimentação**: Foi escolhida como linguagem de programação Python, em conjunto com a biblioteca GMPY2, sendo assim iniciada a implementação do esquema totalmente homomórfico.

- **Validação e Verificação**: Foi realizada uma série de testes unitários e de integração das primitivas homomórficas implementadas, verificando assim a integridade dos resultados gerados, assim obtendo os tempos de execução de cada uma delas, junto dos resultados foi proposto um modelo para implementação em GPU.

1 CAPÍTULO 1: FUNDAMENTAÇÃO TEÓRICA

Neste capitulo serão apresentados os principais conceitos sobre esquemas homomórficos, bem como suas inovações, definições, classificações e base matemática que viabiliza a criptografia homomórfica.

1.1 Circuitos Booleanos e Algébricos

Definindo de maneira formal um circuito booleano é a representação física de um modelo matemático para a lógica digital booleana, este sendo formado por um conjunto de portas logicas conectadas por fios, modelado por um grafo orientado acíclico. O circuito é estimulado por suas entradas, onde são inseridos valores booleanos, valores que são processados pelas portas logicas interligadas, gerando assim um novo conjunto de valores booleanos como saída. Define-se como profundida de um circuito a distância entre as entradas e saídas do circuito. Tais circuitos possuem um modelo computacional completo e são capazes de computar algoritmos arbitrários (SKYUM e VALIANT, 1985).

Circuitos algébricos também possuem um modelo computacional completo, viabilizando assim a criação de uma função que seja um homomorfismo, até mesmo completo, para ser utilizado na criptografia, isso significa que é possível executar somar e multiplicações em textos criptografados bem como processar circuitos algébricos de maneira homomórfica.

1.2 Álgebra Abstrata

Nesta sessão serão apresentadas de maneira breve conceitos matemáticos relacionados a álgebra abstrata relevantes ao projeto.

Um anel é uma estrutura matemática consistente em um conjunto do grupo abeliano juntamente de duas operações binarias, normalmente soma (+) também pertencente ao grupo abeliano (BEACHY, 2014) e multiplicação (×), sendo que cada operação faz a combinação de dois elementos a fim de formar um terceiro e devem estar relacionadas pela propriedade

distributiva. Tais operações são familiares a muitas estruturas matemáticas como sistemas de números ou números inteiros.

Contido na teoria dos anéis tem-se o conceito de ideal, trata-se de um subconjunto especial fechado pertencente a um anel, este generaliza propriedades do conjunto dos números inteiros para com o anel.

A palavra homomorfismo tem origem grega, ὁμός (homos), significando "mesmo" e μορφή (morphe), significando "formato", na língua portuguesa define-se homomorfo como "Homomorfo: adj. (homo+morfo) Que tem a mesma forma" (MICHAELIS), dicionário da língua portuguesa), no entanto será utilizado o conceito de homomorfismo aplicado a álgebra abstrata, a mesma trata o homomorfismo como sendo funções naturais entre duas estruturas algébricas pertencentes ao mesmo tipo, como os anéis, transformam as somas e produtos de seu anel domínio em somas e produtos de suas imagens, sendo assim chamadas de funções homomórficas, por exemplo, dado o anel do conjunto de matrizes 2 x 2 de números reais ($\mathbb{R}$) dotado das operações de soma e multiplicação, é definida uma função f entre os anéis como a seguir:

$$f(r) = \begin{pmatrix} r & 0 \\ 0 & r \end{pmatrix}$$

Sendo r um número real, f é dita homomórfica se preservar as operações de adição e a multiplicação:

$$f(r+s) = \begin{pmatrix} r+s & 0 \\ 0 & r+s \end{pmatrix} = \begin{pmatrix} r & 0 \\ 0 & r \end{pmatrix} + \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix} = f(r) + f(s)$$

$$f(r \times s) = \begin{pmatrix} r \times s & 0 \\ 0 & r \times s \end{pmatrix} = \begin{pmatrix} r & 0 \\ 0 & r \end{pmatrix} \times \begin{pmatrix} s & 0 \\ 0 & s \end{pmatrix} = f(r) \times f(s)$$

1.3 Reticulados

Reticulados podem ser definidos como uma estrutura L=(L,R) tal que L é parcialmente ordenado por R e para cada dois elementos a, b de L se tem um menor limite superior e um maior limite inferior de {a,b}, assim como um reticulado pode ser validado como uma estrutura algébrica (L,∧,∨), onde L é um conjunto sendo ∧ e∨ duas operações sobre L sendo um reticulado para todos os elementos de L se validas as leis comutativas, associativas, de absorção e de idempotência.

1.4 Criptografia Simétrica

A criptografia de chave simétrica é foi e ainda é utilizado em muitos meios, Whiteman, Michael E. e Mattord, Herbert J. definem a criptografia simétrica como:

> Metodologia de criptografia que exige a mesma chave secreta para cifrar e decifrar a mensagem a está usando, o que é chamado de criptografia de chave privada ou criptografia simétrica. Métodos de criptografia simétrica utilizam operações matemáticas que podem ser programados na forma de algoritmos computacionais extremamente rápidos para que os processos de encriptação e decriptação sejam executados rapidamente mesmo por pequenos computadores. (WHITMAN, 2011, p. 364).

As chaves secretas mencionadas são geradas por uma parte do algoritmo criptográfico, essas são compostas por informações únicas e representam um parâmetro único de entrada do algoritmo de encriptação e decriptação, sem a qual o algoritmo produziria uma saída inútil. Na criptografia simétrica a chave utilizada tanto na encriptação quanto na decriptação de informações é a mesma, essa característica simétrica exige que no ato de comunicação, ambas as partes possuam a mesma chave.

Um exemplo muito utilizado na literatura criptográfica é o de Alice e Bob, neste cenário Alice e Bob são amigos e querem trocar mensagens pela internet de maneira segura, para isso Alice fazendo uso de criptografia simétrica gera uma chave secreta compartilhada, ela entrega uma cópia dessa chave pessoalmente para Bob (meio seguro), no dia seguinte Alice envia uma mensagem a Bob, essa mensagem é cifrada utilizando essa chave e enviada para Bob através da internet (meio inseguro), Bob recebe a mensagem e fazendo uso da chave dada por Alice, Bob decripta a mensagem e consegue compreende-la.

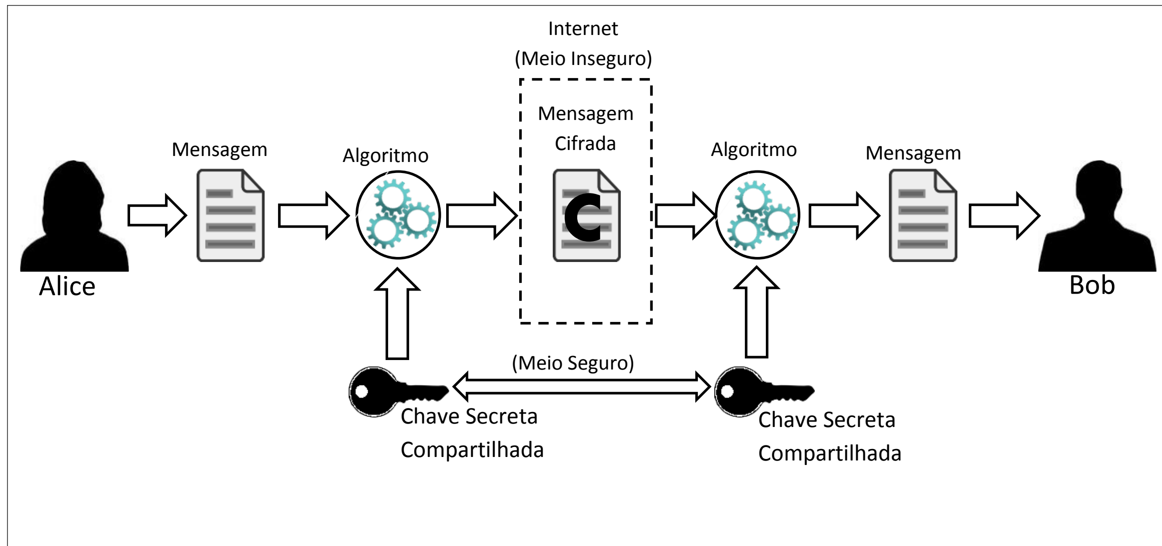

Figura 1: Processo de compartilhamento e funcionamento de chaves simétricas
Fonte: própria

A característica simétrica deste algoritmo pode acarretar em uma grave falha de segurança, caso a chave secreta compartilhada for comprometida, ou seja, obtida de maneira oculta por uma terceira parte, essa terceira parte poderá interceptar, ler e até mesmo alterar todas as mensagens transmitidas entre Alice e Bob, o uso da criptografia de chaves assimétricas evita que isso aconteça.

1.5 Criptografia Assimétrica

A criptografia de chaves assimétricas possui similaridades com em relação a criptografia de chaves simétricas, no entanto essa se diferencia pois faz uso de um par de chaves criptográficas logicamente relacionadas, uma sendo a chave pública e outra privada. Enquanto a chave pública pode ser livremente divulgada e vinculada até mesmo através de um meio inseguro, a chave privada deve ser mantida em segredo.

Em posse de um par de chaves, sendo a chave privada A e a chave pública B, e criptografa-se uma mensagem utilizando a chave A, esta mensagem somente poderá ser decriptada utilizando a chave B, se uma mensagem for criptografada com a chave B essa mensagem somente poderá ser decriptada pela a chave A. Essa característica assimétrica é o que garante a segurança deste modelo, já que isso caracteriza uma função de sentido único, de acordo com Whiteman, Michael E. e Mattord, Herbert J.:

> Os algoritmos assimétricos são funções de sentido único. A função de sentido único é simples de calcular em uma direção, mas complexo para calcular na direção oposta. Essa é à base da criptografia de chave pública (WHITMAN, 2011, p.366).

Exemplificando, Alice e Bob querem trocar mensagens de maneira segura pela internet, no entanto, desta vez Alice que mandar mensagens que apenas Bob seja capaz de decifrar, sendo assim Bob envia previamente a sua chave pública para Alice, que em posse da chave pública de Bob pode criptografar mensagens e envia-las para Bob sabendo que apenas ele será capaz de decifra-las já que Bob é o único em posse de sua chave privada.

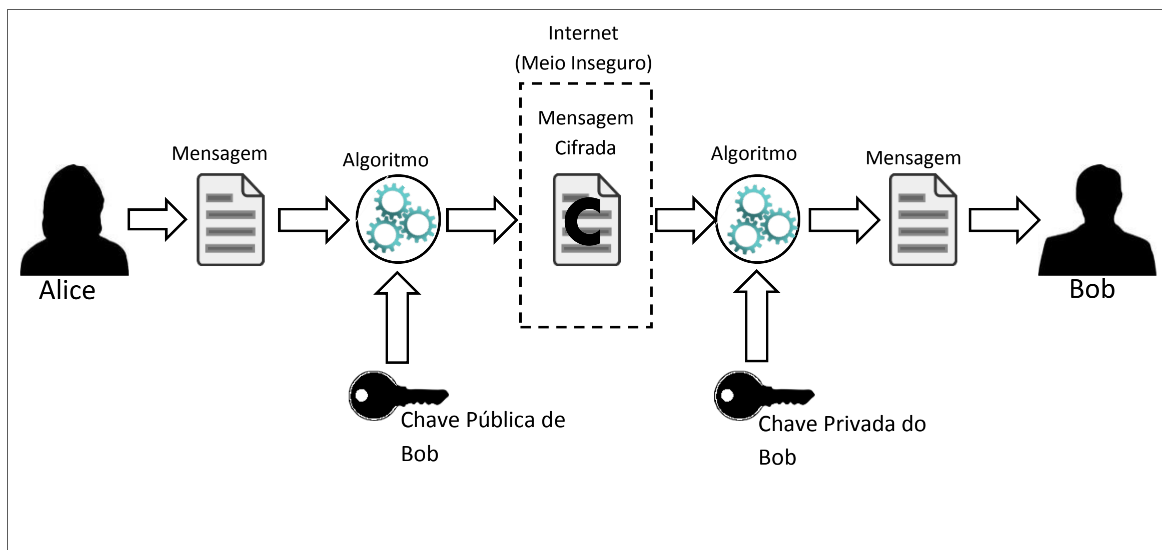

Figura 2: Processo de compartilhamento e funcionamento de chaves assimétricas
Fonte: própria

Outros exemplos e técnicas de trocas de chaves podem ser aplicados a fim de obter-se diferente níveis e tipos de segurança, estes não serão discutidos nesse projeto, já que o objetivo desta sessão é introduzir de maneira breve os conceitos de chave pública e privada, contudo podem ser encontrados na literatura da área.

1.6 Criptografia Parcialmente Homomórfica

Esquemas parcialmente homomórfico quando aplicados preservam de maneira homomórfica uma operação, mas não todas, exemplo deste é o modelo criptográfico RSA, esse é amplamente utilizado no ramo da segurança da informação por conta de seu algoritmo de chave pública assimétrico que possui homomorfismo para a operação de multiplicação (JONSSON e KALISKI, 2003), ente outros como o famoso Elgamal (ELGAMAL, 1985).

1.7 Criptografia Totalmente Homomórfica

Esquemas totalmente homomórfico ou esquemas completamente homomórficos, possuem a capacidade de preservar de maneira homomórfica as operações de soma e multiplicação, possibilitando assim a criação de circuitos digitais de profundidade arbitraria capazes de realizar o processamento de dados de forma homomórfica preservando e conservando os dados processados. A início a ideia de se realizar processamento sobre dados criptografados sem decripta-los aparenta ser um tanto quando paradoxal, no entanto essa impressão de impossível foi uma das motivações para a realização desta pesquisa.

Para compreender melhor os conceitos da criptografia homomórfica iremos utilizar um exemplo que foi inicialmente empregado por Gentry na literatura (GENTRY, 2010), tomando os personagens dos exemplos anteriores imagine que Alice é dona de uma joalheria e Bob é um de seus funcionários.

Sendo dona de uma joalheria Alice tem posse de itens valiosos, como pedras preciosas, ouro, prata, utilizando estes materiais, Bob, seu funcionário deve ser capaz de manipula-los e transformar essa matéria que lhe é disponibilizada em acessórios, como anéis, colares, brincos, etc. No entanto Alice não confia em Bob a ponto de conceder acesso direto a matéria prima, ela assume que ele iria rouba-la se fosse dada a oportunidade. Em suma, temos uma situação onde Alice quer que Bob seja capaz de processar os seus materiais sem ter acesso direto a eles.

Como solução, Alice utiliza uma caixa transparente e impenetrável com luvas que dão acesso ao seu interior, trancada por uma fechadura que apenas Alice tem posse da chave. Alice abre a caixa e coloca os materiais preciosos dentro, em seguida ela tranca a caixa e a entrega a Bob. Usando as luvas, Bob consegue manipular a matéria prima colocadas por Alice dentro da caixa, e assim montar os produtos finais desejados por ela. Sendo a caixa impenetrável, Bob não pode alcançar diretamente as joias, o que impossibilita o roubo, uma vez que Bob retorna a caixa para Alice, ela pode abrir a caixa, assim obtendo o produto final.

Neste exemplo, os materiais utilizados por Bob para produzir as joias representa um conjunto de dados encriptados, $m_1, \dots, m_t$, que podem ser acessados apenas pelo portador da chave de decriptação privada, representado por Alice. As luvas ligadas a caixa representam a maleabilidade proporcionada pelo homomorfismo, este que permite que os dados brutos sejam manipulados enquanto estão na sua forma cifrada. O produto final, ou joia, dentro da caixa representa o dado encriptado processado sobre uma função qualquer, $f(m_1, \dots, m_t)$, a

segurança do homomorfismo é denotada pela restrição física do funcionário aos dados que ele manipula.

Um esquema de encriptação homomórfico qualquer, ($\mathcal{E}$), irá necessitar ao menos de três algoritmos, primitivas criptográficas do esquema: $KeyGen_{\mathcal{E}}, Encrypt_{\mathcal{E}}$ e $Decrypt_{\mathcal{E}}$, estes devem executar em tempo polinomial de lambda (λ), onde λ é o parâmetro de segurança de todo o esquema que define o tamanho em comprimento de bits das chaves geradas pelo esquema (GENTRY, 2009). Em esquemas homomórficos assimétricos, também chamados de esquemas homomórficos de chave pública, a primitiva $KeyGen_{\mathcal{E}}$ utiliza o parâmetro λ para gerar um par de chaves assimétrico, como já foi visto uma chave pública (*pk*) que qualquer entidade pode ter acesso, e uma chave privada ou chave secreta (*sk*) que deve ser mantida em posse reservada a pessoa de interesse.

As primitivas, $Encrypt_{\mathcal{E}}$ e $Decrypt_{\mathcal{E}}$ são respectivamente responsáveis pelos processos de cifrar e decifrar os dados, fazendo uso das respectivas chaves necessárias para a tarefa. Em esquemas homomórficos assimétricos uma quarta primitiva é necessária, essa nomeada de $Evaluate_{\mathcal{E}}$, é associada a um conjunto de funções permitidas de execução ($\mathcal{F}_{\mathcal{E}}$), este tem como entrada a chave pública (pk) do esquema, uma função qualquer f que se deseja aplicar, e os dados cifrados onde a função f será aplicada ($c_1, \dots, c_t$), este então dá como saída um texto cifrado (c), que representa a encriptação de $f(m_1, \dots, m_t)$. Se a função f não está contida no conjunto de funções permitidas de execução $\mathcal{F}_{\mathcal{E}}$ não há garantias de que a função $Evaluate_{\mathcal{E}}$ irá gerar uma saída de utilização para o esquema (GENTRY, 2009).

Lembrando que a solução mais simples para Alice seria nunca dar a caixa para Bob, mas sim simplesmente Alice abri-la e ela mesma manipula a matéria prima dentro da caixa a fim de confeccionar o produto final, assim como ao invés da encriptação de $f(m_1, \dots, m_t)$ ser processada na primitiva $Evaluate_{\mathcal{E}}$ poderíamos modificá-la para a mesma dar como saída $f(c_1, \dots, c_t)$ sem processar as cifras, e então modificar a primitiva de $Decrypt_{\mathcal{E}}$ para processar individualmente a decriptação de *c*, a decriptação de c_1, até a decriptação de c_t, assim obtendo os valores $m_1, \dots, m_t$ e então aplicar a função f sobre essas mensagens, obtendo assim $f(m_1, \dots, m_t)$. No entanto uma solução trivial como essa não é o que a criptografia homomórfica busca, o objetivo aqui é o de processar o dado enquanto mante-se a sua confidencialidade, tanto para o meio externo, quanto para quem o está processando.

Algumas observações devem ser feitas a fim de formalizar o processamento de dados, estas formalizações devem ser feitas diretamente no algoritmo das primitivas do esquema criptográfico, a fim de prevenir e proteger o esquema contra ataques externos, essas

formalizações controlam e definem regras quanto ao tempo de execução, de maneira mais especifica, decriptar c deve levar a mesma quantidade de computação que decriptar c_1, ainda mais, c e $c_1$devem possuir o mesmo tamanho, sendo que com essas alterações as complexidade das primitivas $KeyGen_{\mathcal{E}}, Encrypt_{\mathcal{E}}\ e\ Decrypt_{\mathcal{E}}$ devem continuar sendo polinomial a λ (GENTRY, 2009).

Para trazer o total homomorfismo ao exemplo de Alice, Bob precisa ser capaz de montar todos os tipos de joias, independentemente de sua complexidade em tempo arbitrário, onde o trabalho que Bob precisa fazer sobre as matérias primas não tem influência sobre o tempo que Alice leva para abrir a caixa. O mesmo cabe ao esquema homomórfico $\mathcal{E}$, a primitiva $Evaluate_{\mathcal{E}}$ deve agora, assim como as outras primitivas do esquema, depender apenas do parâmetro de segurança λ, entretanto, este depende da complexidade da função f a ser aplicada. Para medir a complexidade da função f utiliza-se como métrica o tamanho de um circuito booleano S_f, este computa a função f utilizando portas lógicas (AND, OR, NOT), sendo assim, $Evaluate_{\mathcal{E}}$, é dita eficiente se existe um polinômio g, sendo que, para qualquer função f, que é representada por um circuito S_f, assim, $Evaluate_{\mathcal{E}}(pk, f, c_1, \dots, c_t)$, tem a complexidade de $S_f \cdot g(\lambda)$, além de que a primitiva $Evaluate_{\mathcal{E}}$ deve ter um tempo de execução *quasi-linear*, a fim de prevenir vazamentos de informação que fazem jus aos dados cifrados (GENTRY, 2010).

1.7.1 Segurança Contra Ataques

O homomorfismo destaca-se entre as outras soluções oferecidas pela criptografia pôs quântica pela sua própria característica e homomórfica vinda de sua solida base matemática, no entanto, deve-se atentar a certos requerimentos em sua aplicação em esquemas criptográficos. Esquemas criptográficos atuais devem possuir a propriedade de serem semanticamente seguros contra ataques CPA (*chosen-plaintext attacks*), que foi exemplificado da seguinte maneira por Janathan Katz e Yehuda Lindell.

> [...]a ideia básica por trás de um chosen-plaintext attack é que para o adversário $\mathcal{A}$ é permitido solicitar criptografias de várias mensagens que são escolhidas "on-the-fly" de forma adaptativa. Este é formalizado permitindo $\mathcal{A}$ interagir livremente com um oráculo criptografia, visto como uma “caixa-preta” que criptografa mensagens de escolha de $\mathcal{A}$ (essas criptografias são calculadas usando uma chave secreta k desconhecida para $\mathcal{A}$). Seguindo notação padrão em ciência da computação, denotamos por $\mathcal{A}^{\mathcal{O}(\cdot)}$ a computação de $\mathcal{A}$ dando acesso a um oráculo $\mathcal{O}$, e, portanto,

> neste caso, indicamos a computação de $\mathcal{A}$ com acesso a encriptação do oraculo por $\mathcal{A}^{Enc_k}(\cdot)$.Quando $\mathcal{A}$ consulta o oráculo, fornecendo-lhe apenas um simples mensagem de texto m como entrada, o oráculo retorna um texto cifrado $c \leftarrow Enc_k(m)$ como resposta. Quando Enc é aleatório, o oráculo usa moedas aleatórias frescas cada vez que ele responde a uma consulta.
>
> A noção de segurança exige que $\mathcal{A}$ não deve ser capaz de distinguir a encriptação de duas mensagens arbitrárias, mesmo quando é dado para $\mathcal{A}$ acesso um oráculo de criptografia (KATZ e LINDELL, 2007, p.95).

A presença da segurança semântica contra ataques do tipo CPA é de extrema importância para garantir um dos conceitos mais básicos da criptografia (GOLDWASSER e MICALI, 2010) em um esquema criptográfico tão elaborado quanto são os homomórficos. Também se deve se fazer presente a segurança semântica contra ataques de aproximação por MDC (Máximo Divisor Comum) (BERNSTEIN et al, 2008), sendo que o problema de aproximação por MDC deve ser manter difícil mesmo quando são dadas além de duas amostras.

1.7.2 Ruído e Bootstraping

Voltando a analogia, depois de resolver o problema na manufatura de suas joias com as caixas de vidro com luvas que impossibilita Bob de ter contato direto com as matérias primas assim evitando qualquer tipo de furto, Alice agora se depara com um novo problema, após receber uma nova remessa de caixas de vidro, Alice percebe que as luvas que dão acesso ao interior das caixas estão defeituosas: após alguns minutos de uso as luvas ficam extremamente duras, inviabilizando o uso das caixas e impossibilitando a montagem das joias.

O novo problema de Alice está associado ao conceito de ruído, que está presente no esquema homomórfico e é composto de dados aleatórios que devem seguir certas restrições que variam de esquema para esquema. Operações homomórficas como adição e multiplicação aumentam o tamanho do ruído das cifras a cada vez que são executadas sobre a mesma, resultando em cifras com um ruído tão grande que torna a primitiva *decrypt* incapaz de recobrar os dados originais de maneira confiável.

Um noite, Alice sonha com montes e montes de joias, ouro e riquezas empilhados até as alturas, quando de repente um enorme dragão aparece abrindo sua imensurável boca e devora todo o tesouro, e então o dragão começa a comer a sua própria cauda. No dia seguinte Alice reflete sobre o seu sonho e percebe que nele está contida a solução para seu problema com as caixas.

Como antes Alice entrega uma caixa para Bob, denominada caixa 1, contendo as matérias primas para a confecção das joias, contudo Alice entrega para Bob uma segunda caixa, denominada caixa 2, que contem em seu interior a chave da caixa 1, a caixa 3 contem no seu interior a chave da caixa 2 e assim por diante. Assim, Bob pode manipular a matéria prima da caixa 1 as luvas endureçam, então Bob coloca a caixa 1 um dentro da caixa 2, que já contém a chave que abre a caixa 1. Usando as luvas da caixa 2, Bob abre a caixa 1 utilizando a chave, retirando os materiais de dentro da caixa 1 e continua o seu trabalho dentro da caixa 2, uma vez que as luvas da caixa 2 endureçam, Bob coloca a caixa 2 no interior da caixa 3, abre a caixa 2 utilizando a chave que já estava contida dentro da caixa 3, e continua o seu trabalho, repetindo esse processo até que a joia seja finalizada.

O defeito apresentado nas luvas ligadas as caixas na analogia de Alice se faz equivalente a característica de nosso esquema homomórfico qualquer $\mathcal{E}$ de realizar um número limitado de operações matemáticas sobre os dados cifrados até que o ruído na cifra fique grande demais e seja impossível recuperar os dados de maneira confiável, para transforma-lo em um esquema totalmente homomórfico, o mesmo deve ser capaz de executar um número ilimitado de operações matemáticas sobre a cifra sem que isso comprometa a recuperação dos dados.

Gentry mostrou que é possível realizar essa transformação, inicialmente ele demostrou que é possível reduzir todo o processo de decriptação de forma que seja possível representar o mesmo na forma de um polinômio de baixo grau de maneira direta nos bits da cifra dos dados e da chave privada, assim, ao invés do processamento desse polinômio de decriptação ocorrer na cifra e na chave privada o mesmo ocorre de maneira homomórfica diretamente na primitiva $Encrypt_{\mathcal{E}}$, assim tem-se a recuperação da encriptação dos bits do dado encriptado, contudo, se o grau do polinômio de decriptação suficientemente pequeno o ruído resultante dessa nova cifra possuirá um tamanho inferior em comparação ao ruído da cifra original, Gentry chamou tal processo de "*ciphertext refresh*".

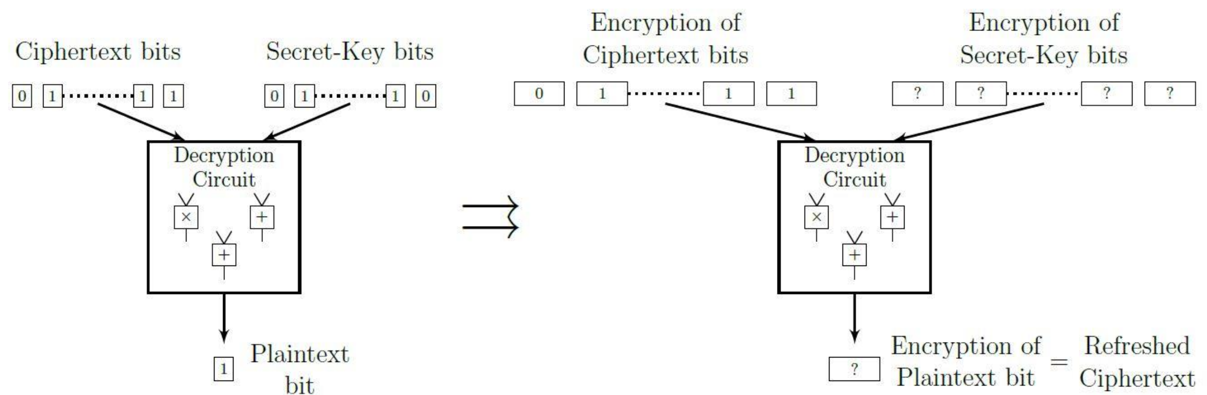

Figura 3: Processo de "ciphertext refresh"
Fonte: CORON et al, 2012

Duas cifras que passaram pelo processo de *ciphertext refresh*, estarão aptas a passar novamente por operações matemáticas de maneira homomórfica novamente, sem que o tamanho do ruído resultante nas cifras comprometa dos valores criptografados contidos nas cifras, dessa forma damos ao nosso esquema homomórfico qualquer $\mathcal{E}$ a característica do homomorfismo total.

1.8 Classificação dos Esquemas Totalmente Homomórficos

Por definição um esquema criptográfico é dito totalmente homomórfico quando este permite executar de forma implícita a adição e multiplicação de dados, manipulando apenas os textos cifrados. (AHITUV et al, 1987)

Atualmente os esquemas totalmente homomórficos são baseados em três classes de problemas matemáticos, como ilustra a figura 4: reticulados ideais, como fora proposto inicialmente por Gentry, anéis de números inteiros, proposto por Dijk, Gentry, Halevi e Vaikuntanathan, e, LWE (do inglês: *Learning With Erros*), proposto pro Brakerski e Vaikuntanathan (BRAKERSKI e VAIKUNTANATHAN, 2014).

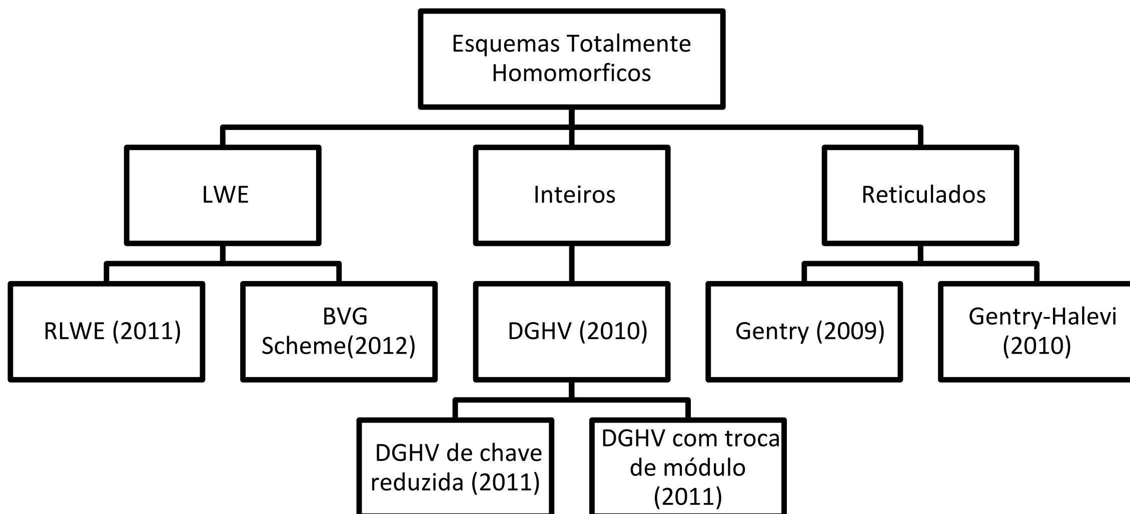

Figura 4: Classificação dos Esquemas Totalmente Homomórficos
Fonte: BILAR, G. R.; SANTOS, L. C.; PEREIRA, F. D., 2014.

1.8.1 Esquemas Totalmente Homomórficos Baseados em Reticulados

O primeiro esquema totalmente homomórfico conhecido foi o proposto por Gentry em sua tese original, apresentada em 2009, o mesmo incluía pela primeira vez a primitiva *recrypt*, que aplicava a técnica de *ciphertext refresh* aplicando assim o conceito de *bootstraping* ao esquema, dando a um esquema originalmente parcialmente homomórfico a característica de ser totalmente homomórfico (GENTRY, 2009). Este esquema criado por Gentry utiliza como base reticulados ideais e realizava um número limitado de somas e multiplicações sobre o dado cifrado, sua implementação obteve destaque sendo reconhecida como um grande avanço em sua área.

Utilizando a tese de Gentry e seu esquema com *bootstraping* como base, Halevi em parceria com o próprio Gentry implementaram em 2010 o esquema totalmente homomórfico Gentry-Halevi, este basicamente o esquema original de Gentry com inclusão de algumas melhorias, dentre essas se destaca uma optimização na geração de chaves do esquema SHE (do inglês: *Somewhat Homomorphic Scheme*), sendo que esta não mais requer um polinômio de inversão total (GENTRY e HALEVI, 2010). Os trabalhos iniciais de Gentry, tornaram-se referência e um ponto de partida para pesquisas relativas a criptografia homomórfica.

1.8.2 Esquemas Totalmente Homomórficos Baseados em Números Inteiros

O uso de reticulados na implementação de um esquema totalmente homomórfico não se mostrou uma das melhores alternativas, de acordo com Coron (CORON et al., 2011) e de Gentry (DIJK et al., 2010), pois as chaves públicas geradas tinham tamanho estimado na ordem de $\tilde{O}(\lambda^{10})$ e o tempo de execução das primitivas dos esquemas criptográficos não atingiam o desempenho desejado para aplicação nos cenários desejados.

Então, Dijk, Gentry, Halevi e Vaikuntanathan, em 2010 propuseram um esquema totalmente homomórfico (DGHV) que utiliza apenas álgebra modular sobre um anel de números inteiros, que se provou tem uma matemática um tanto quanto menos complexa quando comparada com os esquemas baseados em reticulados. Este mesmo esquema foi analisado por Coron, que propôs duas variantes para o mesmo que tornam o custo computacional de certas primitivas menor, e que diminuem o tamanho proibitivo da chave pública do esquema original utilizando diferentes técnicas de redução e compressão.

A primeira variante de Coron, chamada de DGHV com chave reduzida, conta com a adição de novos parâmetros quadráticos adicionados as primitivas do esquema, armazenando assim apenas um pequeno conjunto valores relacionados a chave pública para então gerar a chave pública completa em tempo de execução, utilizando essa técnica, Coron demostrou a redução do tamanho da chave pública de uma ordem de $\tilde{O}(\lambda^{10})$ para $\tilde{O}(\lambda^{7})$(CORON et al., 2011).

A segunda variante de Coron, chamada de DGHV com compactação de chave pública com troca de modulo, onde os elementos da chave são gerados *on-the-fly* por meio de um RNG (do inglês: *Random Number Generator*) e então armazena apenas as sementes e os fatores de correção para os números pseudoaleatórios, o que permite recuperar o valor dos parâmetros da chave pública em tempo de execução, utilizando essa técnica, Coron reduziu ainda mais o tamanho da chave pública gerada pelo esquema, sendo de uma ordem de $\tilde{O}(\lambda^{7})$ para $\tilde{O}(\lambda^{5})$ (CORON et al., 2012).

1.8.3 Esquemas Totalmente Homomórficos Baseados em LWE

Brakerski e Vaikuntanathan desenvolveram um esquema baseado no problema de aprendizagem com erros (sigla em inglês: LWE) e aprendizagem com erro em anel (sigla em inglês: RLEW), os mesmos introduziram no meio cientifico uma nova técnica de redução de dimensão e uma nova técnica de troca de módulo. O grande diferencial desse esquema é que o

ruído no texto cifrado cresce de forma *quasi-linear*, e a técnica de *bootstraping*, antes essencial para outros esquemas totalmente homomórficos, passa a não parte necessária para o funcionamento do sistema, mas sim uma otimização em quesito de desempenho das primitivas criptográficas do esquema. (BRAKERSKI e VAIKUNTANATHAN, 2014)

2 CAPÍTULO 2: HOMOMORFISMO COMPLETO SOBRE NÚMEROS INTEIROS

Neste capitulo iremos descrever o esquema original sobre números inteiros utilizado por Coron como base para seu trabalho tanto quanto para a criação de sua segunda variante. Gentry define que o DGHV sobre inteiros utiliza como base um conjunto de inteiros públicos, $x_i = p.q_i + r_i, 0 \leq i \leq \tau$ onde o inteiro p é secreto, sendo dado um parâmetro de segurança λ, os seguintes parâmetros devem ser utilizados para compor o esquema SHE, que então deve ser aprimorado para gerar o FHE sobre inteiros :

- γ é o comprimento em bits de x_i's.
- η é o comprimento em bits da chave secreta p.
- ρ é o comprimento em bits do ruído r_i.
- τ é o número de x_i's na chave pública.
- ρ'é um parâmetro de ruído secundário utilizado para cifrar.

Que devem seguir as seguintes restrições:

- $\rho = \omega(log\lambda)$, para proteção contra ataques de força bruta direcionados ao ruído
- $\eta \geq \rho.\Theta(\lambda log^2\lambda)$, para que seja possível realizar operações homomórficos para avaliar o "circuito de decriptação reduzido"
- $\gamma = \omega(\eta^2.log\lambda)$, para frustrar ataques baseados em retículos com aproximação pelo problema de MDC
- $\tau \geq \gamma + \omega(log\lambda)$, para reduzir a aproximação por MDC
- $\rho' = \rho + \omega(log\lambda)$, para o parâmetro de ruído secundário

Dados esses parâmetros é possível gerar as primitivas do esquema DGHV, sendo que para um inteiro ímpar p de η-bit, seja utilizada uma distribuição sobre inteiros de γ-bit:

$$\mathcal{D}_{\gamma,\rho}(p) = \{ \, Escolha \; q \leftarrow \mathbb{Z} \cap \left[0, \frac{2^\gamma}{p}\right), r \leftarrow \mathbb{Z} \cap (-2^\rho, 2^\rho) : Saída \; x = q.p + r\}$$

2.1 Primitivas do DGHV

As primitivas deste esquema são quatro, *KeyGen, Encrypt, Decrypt* e *Evaluate*. *KeyGen* é a primitiva responsável pela geração do par de chaves do esquema, *Encrypt* responsável por gerar o texto cifrado, *Decrypt* responsável por decifrar o texto cifrado, e *Evaluate*, que executa, de forma pública, um circuito logico sobre uma tupla de bits cifrados, e, que retorna o equivalente cifrado desse circuito aplicado aos dados originais.

Sendo assim obtemos as primitivas como:

2.1.1 KeyGen(λ)

Sendo a chave privada um inteiro ímpar de η-bits, $p \leftarrow (2\mathbb{Z}+1) \cap [2^{\eta-1}, 2^{\eta})$, para a chave pública amostramos $x_i \leftarrow \mathcal{D}_{\gamma,\rho}(p)$ para$i = 0, \dots, \tau$. De forma que $x_0$seja maior, recomeçando a não ser que x_0 seja um número ímpar e ainda $r_p(x_0)$. Assim a chave pública é pk=$\langle x_0, x_1, \dots, x_\tau \rangle$.

2.1.2 Encrypt (pk, $\boldsymbol{m} \in \{\mathbf{0}, \mathbf{1}\}$)

Onde, escolhe-se um subconjunto $S \subseteq \{1,2, \dots, \tau\}$ aleatório, e um inteiro aleatório *r* entre $\left(-2^{\rho'}, 2^{\rho}\right)$, e gera o texto cifrado *c* como:

$$c = \left[m + 2r + 2\sum_{i \in S} x_i \right]_{x0}$$

2.1.3 Evaluate(pk, $\boldsymbol{C}, \boldsymbol{c_1}, \dots, \boldsymbol{c_t}$)

Onde, dado o circuito C com$t$ bits de entrada e *t* textos cifrados c_i, aplicar a adição e multiplicação das portas lógicas de C nos textos cifrados, executando todas as operações sobre os inteiros, e retornar o inteiro resultante.

2.1.4 Decrypt (sk, c)

Tem como saída $m \leftarrow (c \bmod p) \bmod 2$. Note que $c \, mod \, p = c - p \, . \lfloor c/p \rceil$ep é impar, invés podendo computar: $m \leftarrow [c]_2 \oplus [\lfloor c/p \rceil]_2$.

Assim é descrito o esquema criptográfico parcialmente homomórfico, sendo este semanticamente seguro contra ataques de aproximação de MDC.

2.2 DGHV com compressão de chave e mudança de módulo

No trabalho intitulado "*Public Key Compression and Modulus Switching for Fully Homomorphic Encryption over the Integers*", além de demonstrar um ataque a esse sistema com complexidade de $\tilde{O}(2^{\rho})$, Coron obteve uma implementação em SAGE cuja chave pública possuía 10.1 MB de tamanho sem o uso de framework BGV e 18 MB com o uso do framework BGV, ao contrário de 802MB do seu trabalho anterior. Nesse trabalho, o comprimento da chave pública foi reduzido ainda mais de $\tilde{O}(\lambda^7)$ para $\tilde{O}(\lambda^5)$.

A principal inovação proposta por Coron nesse esquema, é que, ao invés de se armazenar os elementos da chave e criptografia desses elementos é armazenada apenas a correção do valor com relação a um gerador de números aleatórios. Dessa forma, os dados a serem armazenados são menores, e, o dado completo é recuperado "*on-the-fly*" pelas primitivas de *encrypt, recrypt, decrypt* e *expand.* Além disso, é descrita uma técnica de troca de módulo, que permite a esse esquema utilizar o Framework sem *boostraping* proposto por Brakerski, Gentry e Vaikuntanathan.

É apresentada a seguir a descrição completa do esquema proposto por Coron em seu trabalho:

2.2.1 KeyGen(1^{λ})

Gerar um número ímpar p com η bits. Escolhe um número inteiro $q_0 \in [0, 2^{\gamma}/p)$, sendo $x_0 = q_0 \cdot p$. Inicializa um gerador de números pseudoaleatórios f_1 com uma semente aleatória se_1. Aplicando assim $f_1(se_1)$ para gerar o conjunto de inteiros $\chi_i \in [0, 2^{\gamma}/p)$ para $1 \leq i \leq \tau$, computando assim:

$$\delta_i = \langle \chi_i \rangle_p + \xi_i \cdot p - r_i$$

onde $r_i \leftarrow \mathbb{Z} \cap (-2^\rho, 2^\rho)$ e $\xi_i \leftarrow \mathbb{Z} \cap \left[0, \frac{2^{\lambda+\eta}}{p}\right)$. Sendo $pk^* = (\mathrm{se}_1, x_0, \delta_i, \ldots, \delta_\tau)$ sendo os inteiros correspondentes a x_i para$1 \leq i \leq \tau$, definido por $x_i = X_i - \delta_i$.

Adicionalmente gera um vetor randômico de bits s de comprimento Θ, sujeito as condições que de $s_1 = 1$, para cada $k \in \lfloor 0, \theta)$, onde se tem no máximo um bits não zero entre s_i para cada $k \cdot B + 1 \leq i < (k+1) \cdot B + 1$, onde $B = \lfloor \Theta/\theta \rfloor$, e que o peso de Hamming de s seja exatamente θ.

Inicializa um gerador de números pseudoaleatórios f_2, utilizando uma semente aleatória se_2, gerando inteiros $u_i \in [0, 2^{k+1})$ para $2 \leq i \leq \Theta,$ onde $k \coloneqq \gamma + n + 2$. Assim, obtemos u_1 da seguinte maneira:

$$\sum_{i=1}^{\Theta} si \cdot u_i = x_p \, mod \, 2^{k+1}$$

onde $x_p \leftarrow \lfloor 2^\kappa / p \rceil$.

Inicializa-se um gerador de números pseudoaleatórios f_3, utilizando uma semente aleatória se_3, computando assim codificações de σ do vetor s da seguinte maneira: utiliza-se $f_3(se_3)$ para gerar inteiros $\chi'_i \in [0, 2^\gamma[\, paracada \, 1 \leq i \leq \Theta$, gerando assim inteiros aleatórios $r'_i \in (-2^\rho, 2^\rho)$ e $\xi'_i \in [0, 2^{\gamma+\eta}/p)$, assim tendo:

$$\delta'_i = \langle \chi'_i \rangle_p + \xi'_i \cdot p - 2 \cdot r'_i - s_i$$

A codificação correspondente de $\sigma_i$des_i sendo definida como:

$$\sigma_i = \chi'_i - \delta'_i$$

Obtém-se assim como saída a chave privada $sk = s$ e a chave pública $pk = (pk^*, se_2, u_1, se_3, \delta')$.

2.2.2 Encrypt (pk, $m \in \{0, 1\}$)

Recuperar os inteiros x_i dapk^*. Escolhendo assim um vetor aleatório inteiro $\boldsymbol{b} = (b_i)_{1 \leq i \leq \tau} \in [0, 2^\alpha)^\tau$ e um inteiro aleatório r em $(-2^{\rho'}, 2^{\rho'})$. Computando a cifra como:

$$c^* = m + 2r + 2\sum_{i=1}^{\tau} b_i \cdot x_i \; mod \; x_0$$

2.2.3 Add(pk, c_1^*, c_2^*)

Retorna $c_1^* + \; c_2^* mod x_0$

2.2.4 Mult(pk, c_1^*, c_2^*)

Retorna $c_1^* \cdot \; c_2^* mod x_0$

2.2.5 Expand(pk, c^*)

O procedimento de expansão do texto cifrado recebe um texto cifrado c^* e computa a matriz associada expandindo o texto cifrado. Para tal, cada $1 < i < \Theta$, primeiramente computa-se o número inteiro aleatório u_i usando o gerador de números pseudoaleatórios $f_2(se_2)$, então calcule $y_{i,j} = \; u_{i,j}/2^{\kappa}$ e então compute z_i :

$$z_i = [c^* \cdot \; y_i]_2$$

Mantendo apenas $n = \lceil log_2(\theta + 1) \rceil$ bits de precisão após o ponto binário. Definindo o vetorz $= (z_i)$. Retornando o texto cifrado expandido $c = (c^*, z)$

2.2.6 Decrypt(sk, c^*, z)

Retorna$m \leftarrow \left[c^* - \left\lfloor \sum_{i=1}^{\Theta} s_i \cdot z_i \right\rceil\right]_2$

2.2.7 Recrypt(pk, c^*, z)

Recupera os bits da chave privada criptografada σ_i da *pk*. Aplicando o circuito de decodificação ao texto cifrado expandido z e aos bits da chave secreta criptografada σ_i. Tendo como saída um texto cifrado com o ruído reduzido como c_{novo}^*.

Assim finalizamos a descrição completa do esquema DGHV Totalmente Homomórfico de chave pública reduzida, a primitiva *Evaluate*, foi fragmentada, dando

origem a outras a outras duas primitivas de nome *Add* e *Mult*, que emulam, respectivamente, o comportamento das portas logicas *XOR* e *AND*.

3 CAPÍTULO 3: PESQUISA E DESENVOLVIMENTO

Neste capitulo será descrito todo o processo de pesquisa, trabalhos correlatos e a implementação da segunda variante de Coron e as ferramentas envolvidas no processo.

3.1 Trabalhos Correlatos

Em sua pesquisa Coron implementou toda a sua proposta de variante para o esquema DGHV utilizando como SAGE (*System for Algebra and Geometry Experimentation*), um software matemático que utiliza Python, com suporte a construções procedurais, funcionais e orientadas a objeto, o código da mesma foi disponibilizado a comunidade online para consulta, em três versões, uma sem a minimização do processo de decriptação e sem o processo de *bootstrapping*, mas que conta com a técnica de compressão de chave pública descrita no capítulo anterior, uma segunda versão que conta com o processo de decriptação minimizado e a técnica de compressão de chave pública e ainda uma terceira versão que com a minimização do processo de decriptação de dados, a técnica de compressão de chave pública e o processo de *bootstrapping,* a tabela 1 ilustra o tempo de execução das primitivas criptográficas do sistema obtidas por Coron.

Tabela 1: Tempos de execução obtidos por Coron

Parâmetro de Segurança	*KeyGen*	*Encrypt*	*Decrypt*	*Expand*	*Recrypt*
Toy	0,06 s	0,05 s	0,00 s	0,01 s	0,41 s
Small	1,3 s	1,0 s	0,00 s	0,15 s	4,5 s
Medium	28 s	21 s	0,01 s	2,7 s	51 s
Large	10 min	7 min 15 s	0,05 s	51 s	11 min 34 s

Fonte: CORON et al, 2012

O fato deste ser um dos poucos trabalhos correlatos encontrados na literatura que realmente tem como o objetivo a implementação de um esquema totalmente homomórfico serviu como uma motivação a mais sendo ao mesmo tempo um desafio para a implementação deste esquema.

Também foi encontrado um trabalho tendo como foco a aceleração em GPU das primitivas de *Encrypt*, *Decrypt* e *Recrypt* do esquema totalmente homomórfico Gentry-Halevi, modificações foram realizadas na mesma utilizando técnicas matemáticas como FFT

(*Fast Fourier Transformation*) a fim de deixar as mesmas elegíveis ao paralelismo oferecido por GPU's, obtendo os tempos de execução apresentados na tabela 2.

Tabela 2: Tempo de execução em diferentes plataformas

	CPU	GPU
Encrypt	1,69 s	0,22 s
Decrypt	18,5 ms	2,5 ms
Recrypt	27,68 s	4,2 s

Fonte: WANG W., HU Y., CHEN L.,HUANG X.,SUNAR B., 2012

Os tempos de execução apresentados na tabela 2 foram obtidos em uma GPU NVIDIA Tesla C2050, essa possui uma memória dedicada de 6 gigabytes com barramento GDDR5, contando com 448 núcleos de processamento dedicado do tipo CUDA.

3.2 Pesquisa Cientifica

Durante o trabalho o contato inicial com esquemas homomórficos deu-se com o esquema inicial de Gentry sobre reticulados ideais, um código de estudo foi obtido no site da empresa IBM, no entanto foi constatado que o tempo de execução e o tamanho das chaves públicas geradas não eram elegíveis a uma aplicação prática do mesmo, sendo que o tempo de execução de algumas primitivas criptográficas chegava ao tempo de horas e as chaves públicas geradas pelo esquema ocupavam a casa dos 2 gigabytes de memória, então foi iniciada uma busca por esquemas totalmente homomórficos baseados em outros problemas matemáticos que poderiam resolver ou amenizar as dificuldades encontradas com os reticulados ideais.

Dentre os esquemas totalmente homomórficos pesquisados o trabalho sobre números inteiros desenvolvido por Dijk (DIJK et al, 2010) mostrou-se promissor, sendo preliminarmente implementado o seu SHE e executado utilizando baixos parâmetros de segurança com a finalidade de teste, a experiência adquirida nessa processo proporcionou uma busca mais aprofundada dos esquemas totalmente homomórficos que utilizam a mesma base matemática para o seu funcionamento, vale destacar que apesar de uma maior facilidade de implementação e familiaridade com a matemática do esquema, fatores como tempo de execução e tamanho da chave pública gerada continuavam a ser um desafio.

Após uma pesquisa mais aprofundada foram encontrados as variantes do esquema totalmente homomórfico sobre inteiros propostas por Coron (CORON et al, 2011) (CORON et al, 2012) seu estudo sobre o esquema homomórfico propõe uma redução no tamanho das

chaves públicas geradas pelo esquema, bem como melhorias em certas primitivas criptográficas através de técnicas de compressão, redução de chave e mudança de módulo, sendo a técnica de compressão de chave e mudança de módulo a escolhida para a implementação neste trabalho, decorrido do fato de que a mesma apresenta o maior fator de redução da chave pública e uma representação da primitiva de *decrypt* normalizada.

Sendo assim a figura 5 ilustra todos os esquemas totalmente homomórficos que estiveram envolvidos durante a pesquisa e desenvolvimento deste trabalho.

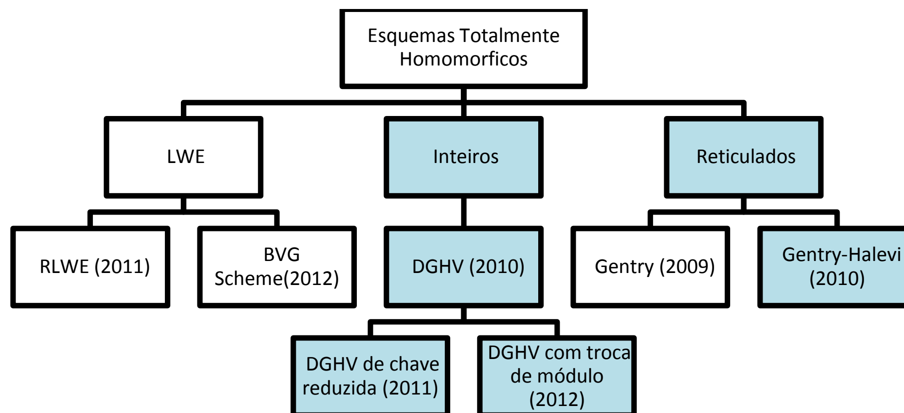

Figura 5: Esquemas Totalmente Homomórficos foram pesquisados durante o trabalho
Fonte: própria

3.3 Ferramentas

Para a implementação, além da linguagem de programação Python 3.3, também foi utilizada a biblioteca GMPY 2, que dá suporte a processamentos matemáticos rápidos, com velocidade comparável a código C, possibilitando a utilização do tipo numérico MPZ, este sendo um número inteiro de precisão arbitraria, tornando possível a manipulação dos grandes números gerados pelas primitivas criptográfica e ainda garantindo uma velocidade de cálculo maior do que a do tipo *long* contido nativamente na linguagem Python.

3.4 Implementação

Nesta sessão serão descritos os trechos de códigos produzidos durante a implementação das primitivas criptográficas do esquema homomórfico, dando destaque as primitivas de *decrypt*, *encrypt*, *keygen*, *expand*, as funções homomórficas de *add* e *mult*, e o

gerador de números aleatórios utilizado, os mesmos fazem uso das ferramentas apresentadas na sessão anterior.

3.4.1 Gerador de Números Aleatórios

Inicialmente, foi criada uma classe geradora de números pseudoaleatórios com *set* e *get*, além de um iterador para o Python, como apresentado no código abaixo.

Código Fonte 1: Gerador de Números Aleatórios

```
1  class rng:
2      "gerador de numeros-pseudo aleatorios"
3      def __init__(self,seed,  gam, n):
4          '''gera números no range [0, 2**gam], gera n elementos'''
5          self.seed, self.gam, self.n = seed, gam, n
6          self.list=[None for i in range(self.n)]
7      def __getitem__(self, index): return self.list[index]
8      def __setitem__(self, index, valor):
9          self.list[index]=valor
10     def __iter__(self):
11         random.seed(self.seed)
12         for i in range(self.n):
13             rand=mpz(random.getrandbits(self.gam))
14             if self.list[i]!=None: yield self.list[i]
15             else:
16                 yield rand
```

3.4.2 Função Geradora de Chaves Assimétricas

Para executar a geração do par de chaves criptográficas, é utilizada a classe pk, onde está contido o gerador do par de chaves assimétrico, bem como os elementos dos quais o mesmo necessita.

Código Fonte 2: Geração de Elementos Públicos

```
5  class  pk:
6      '''classe para os elementos da chave, públicos e provados'''
7      def __init__(self, pr):
8          t=-time.clock()
9          self.pr=pr
10         while True:
11             self.p=gmpy2.next_prime(random.getrandbits(pr['eta']))
12             if len(self.p)==pr['eta']: break
13         #print (self.p)
14         self.q0=random.randint(0, (mpz(2)**pr['gam'])/self.p)
15         self.x0=self.q0*self.p
```

No código fonte 2 é realizada a geração de elementos públicos da chave, sendo estes x_0 e q_0, sendo $q_0 \in [0, 2^{\gamma}/p)$ e $x_0 = q_0 \cdot p$, com p sendo um número ímpar de η bits.

Código Fonte 3: Geração do Vetor S

```
22          '''geração do vetor s'''
            assert pr['Theta']%15==0 #Theta tem que ser um multiplo de
23 theta
24          B=int(pr['Theta']/15)
25          self.s=[1]+[0 for i in range(B-1)]
26          for i in range(15-1):
27              self.s=self.s+randList(B)
28
```

No código fonte 3 é calculado o vetor s, sendo este um vetor randômico de bits s com comprimento Θ. Sendo que o mesmo é restringido pelas linhas 24, 25 e 26, obedecendo as restrições matemáticas do esquemas sendo $s_1 = 1$, para cada $k \in [0, \theta)$, onde se tem no máximo um bits não zero entre s_i para cada $k \cdot B + 1 \leq i < (k+1) \cdot B + 1$, onde $B = \lfloor \Theta/\theta \rfloor$, e que o peso de Hamming de s seja exatamente θ.

Código Fonte 4: Geração do Elemento u1

```
29          '''geração do elemento u1'''
30          self.kappa=pr['gam']+pr['eta']+2
31          self.se2=int(time.time())
32          self.f2=rng(self.se2, self.kappa+1, pr['Theta'])
33          self.f2[0]=0 #seta elemento u1 como sendo zero
34          somatorio=0
35          i=0
36          for u in self.f2:
37              somatorio += u*self.s[i]
38              i+=1
```

```
39          soma=mpz(round((mpz(2)**self.kappa)/self.p))
40          self.u1=soma-somatorio
```

Com uma segunda semente aleatória alimentando o gerador de números aleatórios gera-se inteiros $u_i \in [0, 2^{k+1})$ para $2 \leq i \leq \Theta$, onde $k \coloneqq \gamma + n + 2$, como ilustra das linhas 30 a 32 do código, e então gerasse o elemento u_1 como:

$$\sum_{i=1}^{\Theta} si \cdot u_i = x_p \ mod \ 2^{k+1}$$

onde $x_p \leftarrow \lfloor 2^{\kappa}/p \rceil$ como ilustra as linhas 36 a 40.

Código Fonte 5: Encriptação da Chave

```
42          '''geração de deltaPrime = encriptações do vetor sigma'''
43          self.se3=int(time.time())
44          self.f3=rng(self.se2, pr['gam'], pr['Theta'])
            self.deltaPrime=[chi%self.p + random.randint(0,
    mpz(2)**(pr['gam']+pr['eta']/self.p)/self.p) * self.p -
    2*random.randint(-(mpz(2)**pr['rho']), mpz(2)**pr['rho']) -si for
45  chi,si in zip(self.f3, self.s)]
```

No código fonte 5 é calculada a encriptação do vetor s, elemento contido na chave pública, sendo calculado como $\delta'_i = \langle \chi'_i \rangle_p + \xi'_i \cdot p - 2 \cdot r'_i - s_i$ onde utilizasse o gerador de números aleatórios alimentado de uma terceira semente como ilustra as linhas 43 e 44, na linha 45 todo o delta linha é calculado.

Código Fonte 6: Geração das Chaves

```
47          '''geração da chave para picles'''
48          self.pkAsk={'se1':self.se1, 'x0':self.x0, 'delta': self.delta}
            self.pk={'pkAsk':self.pkAsk, 'se2': self.se2, 'u1': self.u1,
49  'se3': self.se3, 'deltaPrime': self.deltaPrime, 'pr':self.pr}
50
51          self.sk=self.s
52          t+=time.clock()
53          print('keygem em: ', t, 'segundos  par==', pr['tp'])
54
55      def returnKeys(self):
56          return self.sk, self.pk
```

O código 6 gera o par de chaves como um dicionário, que combina os elementos gerados dando origem ao par de chaves criptográficas, que podem ser recuperadas por meio da função *returnKeys*.

3.4.3 Função de Encriptação

Código Fonte 7: Primitiva de Encrypt

```
1  def encrypt(pk, m):
2      assert m==0 or m==1
3      x0=pk['pkAsk']['x0']
4      f1=rng(pk['pkAsk']['se1'], pk['pr']['gam'], pk['pr']['tau'])
5      soma=0
6      for chi, delta in zip(f1, pk['pkAsk']['delta']):
           soma += (chi-delta)*random.randint(0,
7  mpz(2)**pk['pr']['alpha'])
       r=random.randint(-mpz(2)**pk['pr']['rho'],
8  mpz(2)**pk['pr']['rho'])
9      c=modNear(m+2*r+2*soma, x0)
10     return c
```

A função de *encrypt* recebe como parâmetros a chave pública, e o bit a ser criptografado, primeiramente, é verificado se o inteiro *m* pertence ao intervalo [0,1]. Em seguida, o gerador de números pseudoaleatórios é iniciado, e, com ele, recuperado o vetor *xi* a partir dos elementos de Delta. Então, a encriptação do bit m é calculada como sendo:

$$c^* = m + 2r + 2\sum_{i=1}^{\tau} b_i \cdot x_i \bmod x_0$$

onde valor de *c* é retornado como um inteiro de precisão múltipla.

3.4.4 Função de Expansão

Código Fonte 8: Primitiva de Expand

```
1  def expand(pk, c):
2
3      n=4 #bits of precision for the u1/2**kappa (u1)has len=148450bits
4      pr=pk['pr']
5      kappa=pr['gam']+pr['eta']+2
6      f2=rng(pk['se2'], kappa+1, pr['Theta'])
7      f2[0]=pk['u1']
8      gmpy2.get_context().precision=300000
9      k=mpz(2)**kappa
```

```
10        y=[u/k for u in f2]
11        z=[modNear(c*yi, 2) for yi in y]
12        gmpy2.get_context().precision=n
13        zprime=[float(zi) for zi in z]
14        gmpy2.get_context().precision=300000
15        return zprime
```

A função de expansão recebe como parâmetros a chave pública, e a cifra *c*, e, a partir deles, computa o vetor associado *z*. Essa função é separada das primitivas *encrypt* e *decrypt*, já que pode ser computada de forma pública a partir apenas de dados públicos. Para tal, são recuperados os inteiros u usando RNG(*se2*). Então, o vetor z é calculada como $z_{i,j} = [c^* \cdot y_{i,j}]_2$.

3.4.5 Função de Decriptação

Código Fonte 9: Primitiva de Decrypt

```
1  def decrypt(sk, c, z):
2      e=zip(sk, z)
3      soma=0
4      for si, zi in zip(sk, z):
5          soma+= si*zi
6      soma=int(round(soma))
7      m=(c-soma)%2
8      return m
```

A função de decodificação recebe como parâmetros a chave secreta, a mensagem cifrada e a matriz *z*. O bit é decodificado de acordo com a seguinte formula:

$$m \leftarrow [c^* - \left\lfloor \sum_{i=1}^{\Theta} s_i \cdot z_i \right\rceil]_2$$

3.4.6 Funções de Cálculo Homomórfico

Código Fonte 10: Funções Homomórficas

```
1  def add(pk, c1, c2):
2      return modNear(c1+c2, pk['pkAsk']['x0'])
3
4  def mul(pk, c1, c2):
5      return modNear(c1*c2, pk['pkAsk']['x0'])
```

Essas funções são o equivalente as portas logicas *XOR* e *AND* para os bits do texto cifrado. *XOR* é definido como uma soma de dois textos cifrados módulo x0 e *AND* é definido como uma multiplicação de dois textos cifrados módulo x0.

4 CAPÍTULO 4: RESULTADOS

4.1 Métricas e Testes

A literatura utiliza comumente a medida do tempo de execução de cada uma das primitivas a fim de quantificar e avaliar o desempenho de cada uma das primitivas criptográficas. As primitivas são executadas de maneira repetitiva e tem o seu tempo de execução contabilizado por software, em posse do tempo de execução e do número de vezes de execução das primitivas aplica-se uma média aritmética simples sobre os mesmos, obtendo assim o tempo médio de execução das respectivas primitivas que pode ser utilizado de maneira comparativa entre variadas implementações e variados esquemas homomórficos.

4.1.1 Parâmetros de Teste

Em seu trabalho Coron propôs parâmetros de teste para a realização do teste das primitivas da variante de seu esquema, sendo estes os apresentados na seguinte tabela:

Tabela 3: Parâmetros de Teste

Parâmetros	λ	ρ	η	μ	$\gamma \times 10^6$	Θ
Toy	42	16	336	56	0.061	195
Small	52	20	390	65	0.27	735
Medium	62	26	438	73	1.02	2925
Large	72	34	492	82	2.20	5700

Fonte: Coron et al, 2012

Os valores destes parâmetros são restritos, sendo *Toy* possuindo um lambda equivalente a 42 bits de segurança, *Small* equivalente a 52 bits de segurança, *Medium* equivalente a 62 bits de segurança e *Large* equivalente a 72 bits de segurança, tal parâmetro define o nível de segurança atribuída diretamente ao par de chaves gerados pela primitiva de geração de chaves *KeyGen*, os outros parâmetros são referentes a matemática contida nas equações apresentadas no capitulo 2 e possuem caráter funcional para o esquema homomórfico.

4.2 Resultados Finais

Nesta sessão serão apresentados os resultados obtidos utilizando as métricas e parâmetros descritos acima, os tempos de execução obtidos foram gerados em um único

núcleo de um processador Core i5 m520 com 2.40Ghz de frequência, em um computador com sistema operacional de 64 bits e 4 GB de memória, são apresentados na tabela 2.

Tabela 4: Tempo de execução das primitivas implementadas

Parâmetro de Segurança	*KeyGen*	*Encrypt*	*Decrypt*	*Evaluate*
Toy	0.6 s	0.02 s	0.0 s	0.2 s
Small	3.6 s	0.6 s	0.0 s	1.9 s
Medium	1 min 48 s	55 s	0.0 s	14.7 s
Large	*	*	*	*

Fonte: própria

Pode-se verificar um estouro de memória representado pelo símbolo (*), o mesmo ocorre devido ao tamanho dos números inteiros gerados com o parâmetro de segurança *Large*, sendo este o maior proposto por Coron (CORON et al, 2012), também é possível verificar o tempo constante de 0 segundos relativo a primitiva de decifração do esquema, o mesmo é uma das características essenciais ressaltadas por Coron em seu trabalho como explicado no capítulo 1 (um) deste trabalho.

Os gráficos das figuras 6,7 e 8 ilustram uma comparação de desempenho de tempo de execução das primitivas criptográficas entre a implementação feita em SAGE disponível na literatura com a implementação realizada em Python 3.X deste trabalho.

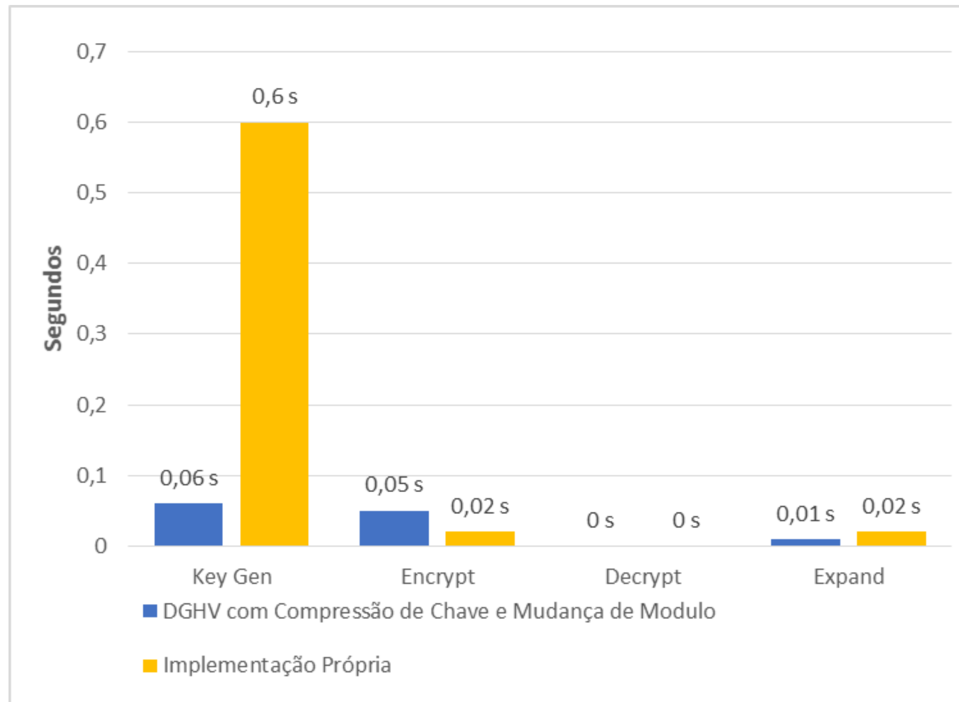

Figura 6: Comparação de desempenho de tempo das primitivas com Toy
Fonte: própria

O gráfico da figura 6 ilustra a comparação de tempo de execução das primitivas quando alimentadas com o parâmetro de segurança *toy*, através da representação gráfica é possível verificar a primitiva *decrypt* teve o mesmo comportamento que o esperado com relação a implementação original do esquema, surpreendentemente a primitiva de *encrypt* deve um desempenho melhor do que a da implementação original, o que se deve ao tipo de dado gerado pela biblioteca utilizada e o tamanho do dado gerado pelo parâmetro *toy*, podemos também constatar a inversão destes fatores influenciando a primitiva de *expand e keygen*.

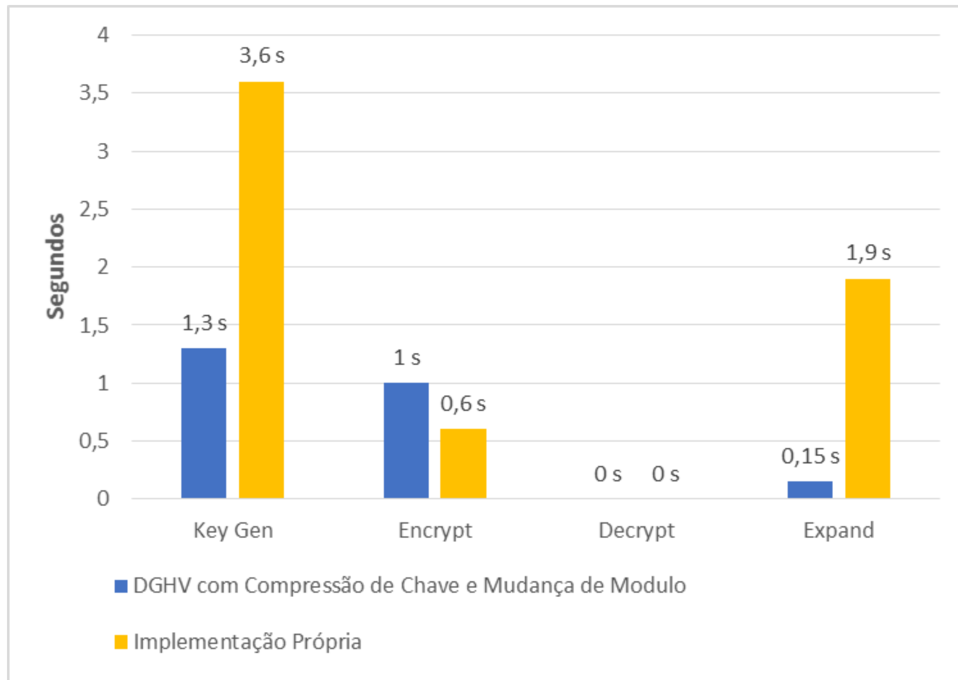

Figura 7: Comparação de desempenho de tempo das primitivas com Small
Fonte: própria

O gráfico da figura 7 ilustra a comparação de tempo de execução das primitivas quando alimentadas com o parâmetro de segurança *small*, é possível observar um bruto aumento no tempo de execução das primitivas de *keygen* e *expand*, isso se deve a manipulação dos dados que tem um tamanho consideravelmente maior que os dados do parâmetro de segurança *toy*. As primitivas de *encrypt* e *decrypt* continuam apesentando um tempo de execução inferior a implementação original, mais uma vez isso vem do fato do tipo de dado gerado pela biblioteca matemática escolhida e do baixo grau de computacional das mesmas.

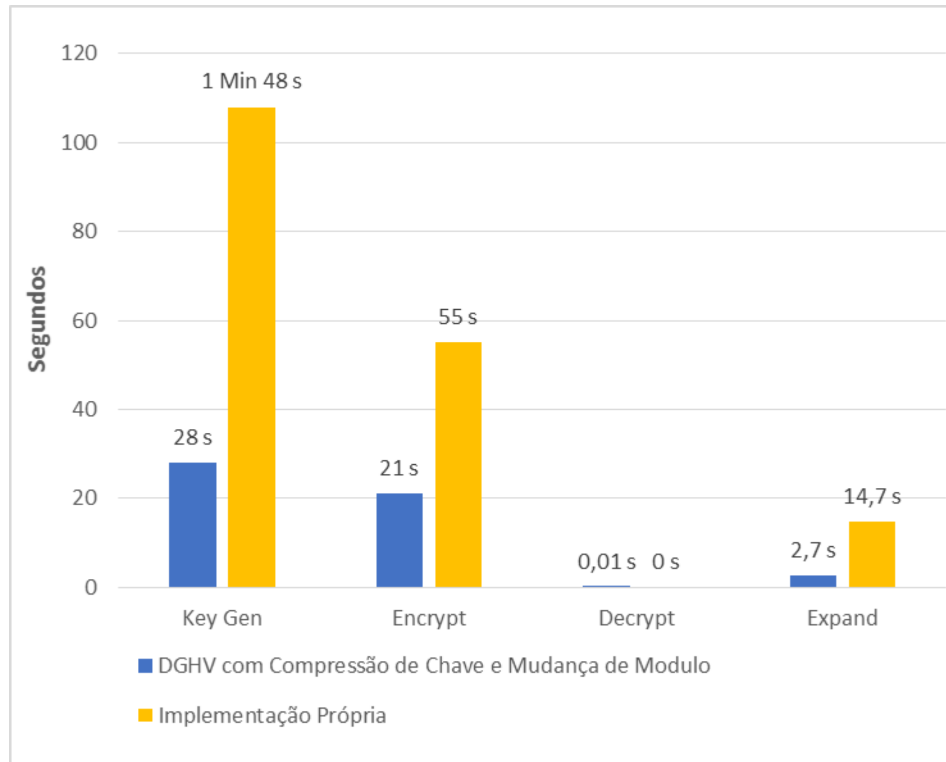

Figura 8: Comparação de desempenho de tempo das primitivas com Medium
Fonte: própria

O gráfico da figura 8 ilustra a comparação de tempo de execução das primitivas quando alimentadas com o parâmetro de segurança *medium*, é possível observar um bruto aumento no tempo de execução das primitivas de *keygen*, *expand*, e *encrypt* com relação a implementação de original, o mesmo deve-se ao tamanho dos dados que estão sendo manipulado, com o parâmetro de segurança *medium*, estamos falando de números inteiros que contem centenas de milhares de casas decimais, o que causa lentidão no processamento dos mesmos, entretanto destaca-se que a primitiva de *decrypt* manteve a característica principal proposta pelo esquema homomórfico implementado de manter um tempo de execução linear mesmo quando os parâmetros de segurança são maiores.

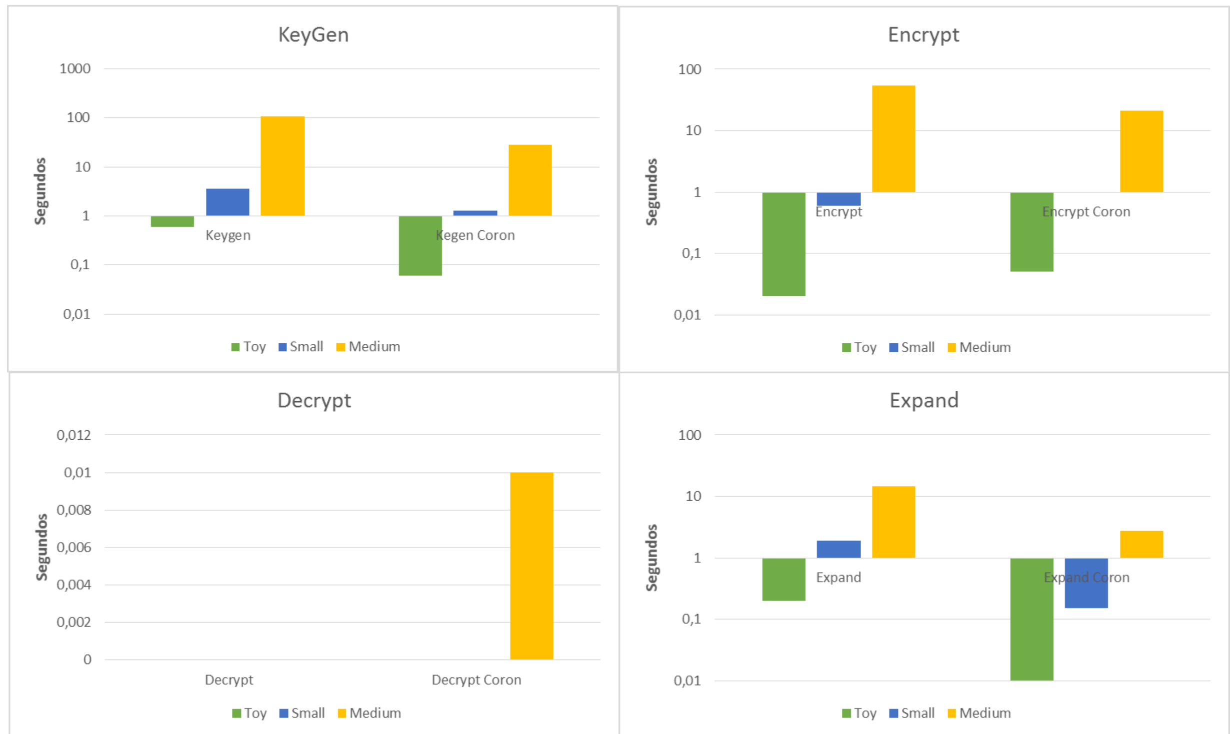

Figura 9: Comparação do desempenho das primitivas entre os parâmetros de segurança das implementações
Fonte: própria

A figura 9 representa o desempenho das primitivas criptográficas do esquema com relação aos parâmetros de segurança testados em escala logarítmica, observa-se o comportamento similar da primitiva de *keygen* com relação ao esquema original seguindo as mesmas tendências de crescimento de tempo, no entanto tendo um pior desempenho de tempo de execução, o mesmo ocorre com as primitivas de *encrypt* e *expand*. A primitiva de *decrypt* comportou-se da mesma maneira em ambos os esquemas homomórficos, tendo um tempo de execução próximo de 0, devido a representação de seu circuito logico.

4.3 Trabalhos Futuros

Nesta sessão serão discutidos os possíveis trabalhos futuros que podem dar continuidade a esta pesquisa, dentre eles destacam-se o paralelismo e a refatoração do código.

4.3.1 Proposta de paralelismo

Um dos desafios atuais para a criptografia homomórfica é o desempenho de tempo das primitivas homomórficas, muitas são as técnicas que podem ser aplicadas para solucionar este problema e trazer o homomorfismo total a realidade de aplicação, no entanto umas das técnicas que vem ganhando destaque entre as outras é a de paralelização de código em

Unidades de Processamento Gráfico (GPGPU), sendo este um hardware de custo relativamente mais baixo que o de hardware dedicados a tarefas especificas, sendo que o mesmo possui um desempenho comprovado no que se refere a melhoria de desempenho.

As GPGPU's são composta por vários processadores especializados em renderização 3D ligados entre si de maneira paralela, tal arquitetura proporciona um nível surpreendente de paralelismo dinâmico, essa característica somada ao poder superior de seus cores de processamento que são muito mais rápidos que os cores convencionais de CPU's torna as GPGPU's uma excelente plataforma para a manipulação de matrizes, vetores e outros complicados cálculos matemáticos, essa diferença fica mais evidente quando representamos e comparamos ambas as arquiteturas de maneira ilustrativa como é feito na figura 10.

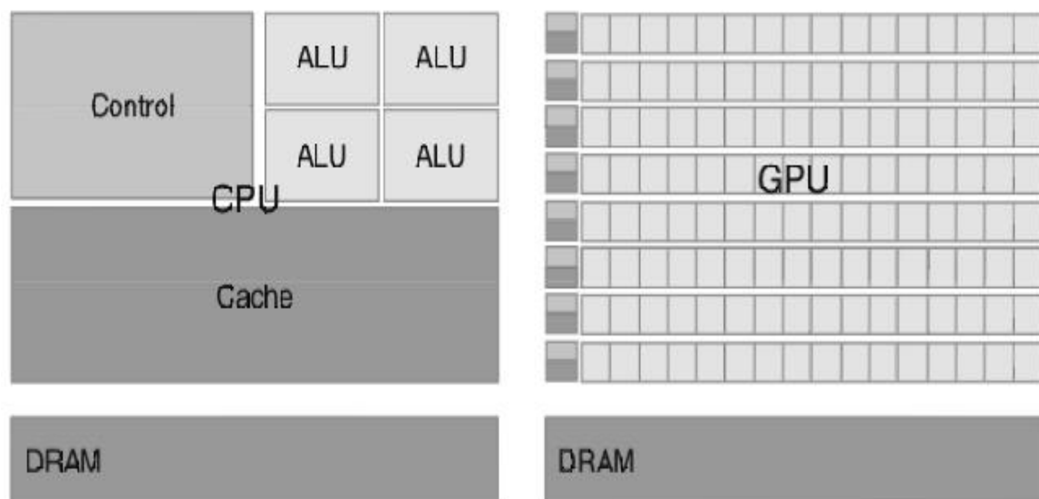

Figura 10: Comparação entre uma arquitetura multicore de uma CPU e a arquitetura many-cores de uma GPGPU
Fonte: DAVID et al, 2010

Para uma comparação mais direta, foi realizado um teste de potenciação em ambas as arquiteturas, sendo que temos uma execução *single core* para a CPU, uma segunda execução paralela para a CPU e uma terceira execução paralela para a GPU, obtendo os resultados apresentados na figura 11.

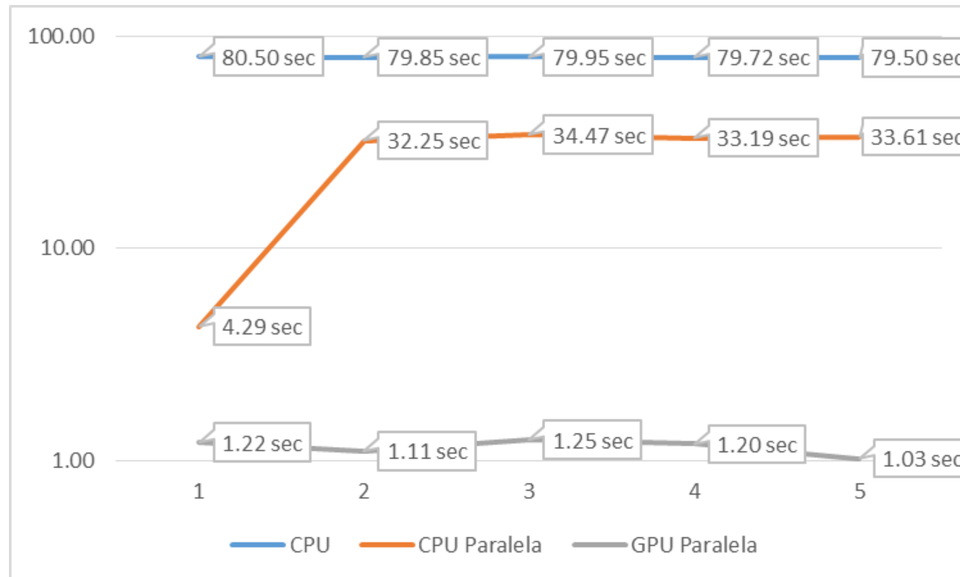

Figura 11: Comparação de diferentes arquiteturas
Fonte: própria

Pode-se notar o desempenho superior da aplicação do mesmo código quando aplicado a uma GPU, em comparação com as outras duas implementações a implementação paralela em GPU foi a mais eficiente, apresentando um fator de tempo aproximado de 69,09 vezes menor que a implementação em *single core* em CPU e 24,08 vezes menor em comparação com a implementação em CPU paralela, o que demostra a eficácia de determinadas aplicações em arquiteturas distribuídas e com múltiplos núcleos de processamento, como é o caso das GPGPU's.

Foi comprovado por Wang, (WANG et al, 2012), que é possível acelerar as funções primitivas de um esquema homomórfico na arquitetura *many-cores* presente em GPGPUs comerciais, em seu trabalho apresentado na HPEC 2012 (*IEEE Conference on High Performance Extreme Computing*), onde foi possível parte implementar o esquema totalmente homomórfico proposto por Gentry e Halevi (GENTRY, HALEVI, 2010), foram aceleradas as primitivas de *Encrypt*, *Decrypt* e *Recrypt*, a *KeyGen* não foi acelerada como explica:

> Para uma aplicação prática, a geração de chave pode ser processada offline, e nós não precisamos acelerar essa parte (WANG et al, 2012, p.4).

Como modelo de execução para arquitetura paralelizada utilizaremos uma técnica de paralelismo de dados, técnica que consiste em executar os algoritmos das primitivas de maneira simultânea em diferentes blocos de processamento, a entrada de dados é distribuída e atribuída entre estes diversos blocos, por meio de uma fila de *work-itens*, sendo possível

assim reduzir o tempo de execução das primitivas do esquema homomórfico, o resultado final poderá ser obtido através da soma dos resultados obtidos, como ilustra a figura 12.

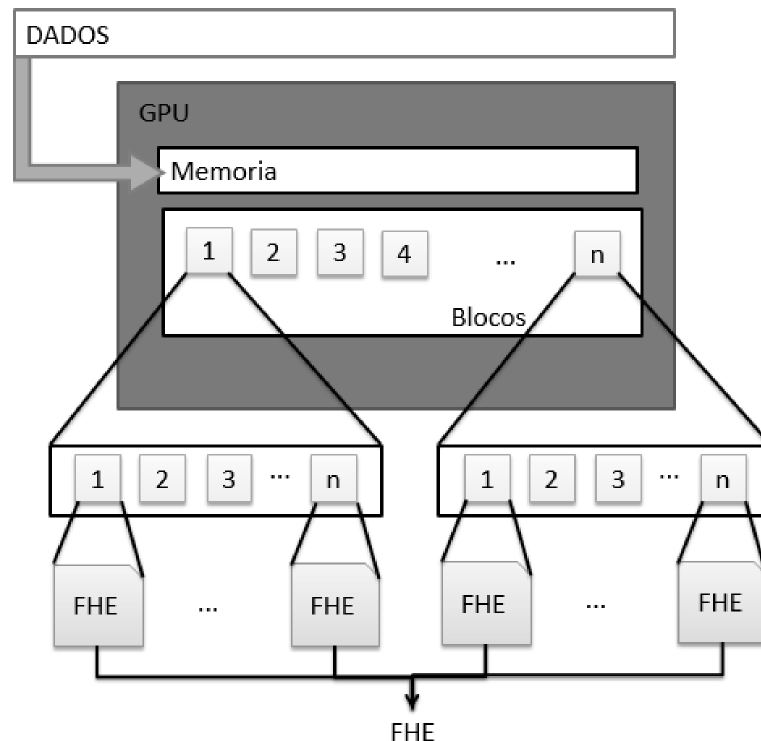

Figura 12: Esquematização da proposta de paralelismo
Fonte: própria

Os dados são inseridos na memória do dispositivo, de lá são atribuídos a unidades de processamento pertencentes a um bloco ou grid pertencente a arquitetura da GPU, dentro de cada uma destas unidade de processamento possui-se uma instancia de diferentes primitivas do esquema homomórfico, assim o ocorre o processamento paralelo simultâneo.

5 CAPÍTULO 5: CONCLUSÕES

5.1 Conclusões

A implementação realizada apresenta algumas limitações com relação a desempenho de tempo de execução das primitivas, como podemos verificar nos gráficos apresentados no capítulo anterior as primitivas com pior desempenho quando comparadas com a implementação foram as de *keygen* e *expand,* os dados da tabela 4 apresentam o fator que quantifica essa relação de desempenho, pela quantidade de vezes que cada uma das operações realizadas poderiam ser executadas na mesma quantidade de tempo que as operações da implementação realizada.

Tabela 5: Relação de eficiência das primitivas

	Keygen	*Expand*
Toy	2	10
Small	2,76	12,66
Medium	3,85	5,44

Isso ocorre devido as diferenças tantos das tecnologias e linguagens escolhidas para implementação, quanto da diferença do hardware utilizado e as condições do ambiente de testes utilizado para a execução dos algoritmos criptográficos, sendo que a implementação própria foi implementada em Python 3.X, utilizando a biblioteca GMPY2, o que acaba por criar uma camada a mais no código que prejudica o desempenho, somado a esse fator temos o tamanho dos números inteiros que estão sendo manipulados, quanto maior o parâmetro de segurança, maior serão os dados gerados como parâmetros de entrada das primitivas e maiores serão os resultados gerados pelas mesmas.

Os ambiente de teste para a implementação própria contava com um processador Core Intel I5 M520 com 2.40GHz de frequência, em um computador com sistema operacional de 64 bits e 4 GB de memória RAM. A implementação original de Coron (CORON et al, 2012) teve como ambiente de teste um processador Intel Core2 Duo E8400 com 3GHz de frequência de trabalho, onde outros dados como sistema operacional ou quantidade de memória principal disponível não foram divulgados.

5.2 Publicações

Os resultados e as pesquisas realizadas durante o decorrer deste trabalho de conclusão de curso foram contemplados com publicações em eventos nacionais de área especifica da computação como o XV Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD) em 2014, com um resumo sendo este publicado nos anais do evento e uma apresentação em forma de banner nas técnicas. A segunda publicação foi no XIV Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg), com publicação nos anais do evento e apresentação oral do trabalho em seção técnica sendo este publicado em conjunto com Luan Cardoso dos Santos, foi agraciado com um prêmio de honra ao mérito, como um dos melhores trabalhos publicados no evento.

REFERÊNCIAS

BEACHY, JOHN A. , **Introductory Lectures on Rings and Modules**, 0.3 Abelian Groups. Disponivel em http://www.math.niu.edu/~beachy/rings_modules/notes/03.pdf. Acesso em 19/01/2014

BENALOH, Josh. **“Dense probabilistic encryption”**. In: Proceedings of the workshop on selected areas of cryptography. 1994. p. 120-128.

BERNSTEIN, D. J., BUCHMANN, J. A., DAHMEN E. **“Post-Quantum Cryptography”**, Chicago and Darmstadt, 2008.

BONEH, Dan et al. **“Private database queries using somewhat homomorphic encryption”**. In: Applied Cryptography and Network Security. Springer Berlin Heidelberg, 2013. p. 102-118.

BRAKERSKI, Zvika; VAIKUNTANATHAN, Vinod. **“Efficient fully homomorphic encryption from (standard) LWE”**. SIAM Journal on Computing, v. 43, n. 2, p. 831-871, 2014.

CORON, J. S., NACCACHE, D., & TIBOUCHI, M. (2012). **“Public key compression and modulus switching for fully homomorphic encryption over the integers”**. No Advances in Cryptology–EUROCRYPT 2012 (p. 446-464). Springer Berlin Heidelberg.

CORON, J.S., MANDAL, A., NACCACHE, D. e TIBOUCHI, M., **“Fully Homomorphic Encryption over the Integers with Shorter Public Keys”**. In P. Rogaway (Ed.), CRYPTO 2011, LNCS, vol. 6841, Springer, pp. 487-504. Full version available at IACR eprint, 2011.

DIJK., M. VAN, GENTRY, C., HALEVI, S. e VAIKUNTANATHAN, V., **“Fully homomorphic encryption over the integers”**. In H. Gilbert (Ed.), EUROCRYPT 2010, LNCS, vol. 6110, Springer, pp. 24-43, 2010.

DAVID, B. Kirk; WEN-MEI, HWU W. **“Programming Massively Parallel Processors: A Hands-on Approach. Burlington”**, MA, USA, 2010.

ELGAMAL, Taher. **A public key cryptosystem and a signature scheme based on discrete logarithms**. In: Advances in Cryptology. Springer Berlin Heidelberg, 1985. p. 10-18.

GENTRY, C., e HALEVI, S., "**Implementing Gentry's fully-homomorphic encryption scheme**," Advances in Cryptology-EUROCRYPT 2011, pp. 129-148, 2011.

GENTRY, C., **“Fully Homomorphic Encryption Using Ideal Lattices”**, Symposium on Theory of Computing - STOC, pp.169-178, 2009.

GENTRY, C. **“A fully homomorphic encryption scheme”**. Ph.D. thesis, Stanford University, 2009, Disponivel em:http://crypto.stanford.edu/craig.

GENTRY, C. **“Computing arbitrary functions of encrypted data”**. Communications of the ACM, v. 53, n. 3, p. 97-105, 2010.

GOLDWASSER, Shafi; MICALI, Silvio. **“Probabilistic encryption. Journal of computer and system sciences”**, v. 28, n. 2, p. 270-299, 1984.

JONSSON, Jakob; KALISKI, Burt. **“Public-key cryptography standards (PKCS)# 1: RSA cryptography specifications version 2.1.”** 2003.

KATZ, Jonathan; LINDELL, Yehuda. **“Introduction to modern cryptography: principles and protocols”**. CRC Press, 2007.

MILLER, V. **"Uses of elliptic curves in cryptography. In Advances in Cryptology, Crypto 85",** Lecture Notes in Computer Science, pages 417–426. Springer, 1985

RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. M. **"A method for obtaining digital signatures and public-key cryptosystems"**. Commun. ACM, 21(2):120–126, 1978.

SHOR, P. W. **"Algorithms for quantum computation: discrete logarithms and factoring"**. Em Shor, P. W., editor, 35th Annual Symposium on Foundations of Computer Science (Santa Fé, NM, 1994), páginas 124–134. IEEE Comput. Soc. Press, 1994.

STALLINGS, William. **"Criptografia e Segurança de Redes: Princípios e Pratica"**. 4º Edição. São Paulo: Pearson, 2007

SKYUM, Sven; VALIANT, Leslie G. **"A complexity theory based on Boolean algebra"**. Journal of the ACM (JACM), v. 32, n. 2, p. 484-502, 1985.

WANG, W., HU, Y., CHEN, L., HUANG, X., e SUNAR, B., **"Accelerating fully homomorphic encryption using gpu,"** in 2012 IEEE Conference on High Performance Extreme Computing (HPEC). IEEE, pp. 1–5m, 2012.

WATANABE, Yodai; SHIKATA, Junji; IMAI, Hideki. **"Equivalence between semantic security and indistinguishability against chosen ciphertext attacks. In: Public Key Cryptography"**—PKC 2003. Springer Berlin Heidelberg, 2002. p. 71-84.

WHITMAN, Michael; MATTORD, Herbert. **"Principles of information security".** Cengage Learning, 2011.

APÊNDICE A: CODIGO DO ESQUEMA TOTALMENTE HOMOMORFICO

Arquivo FHE_2.py

```
from gmpy2 import mpz
import gmpy2
import random
import time
from utils import *

class rng:
    "gerador de numeros-pseudo aleatorios"
    def __init__(self,seed,  gam, n):
        '''gera numeros no range [0, 2**gam], gera n elementos'''
        self.seed, self.gam, self.n = seed, gam, n
        self.list=[None for i in range(self.n)]
    def __getitem__(self, index): return self.list[index]
    def __setitem__(self, index, valor):
        self.list[index]=valor
    def __iter__(self):
        random.seed(self.seed)
        for i in range(self.n):
            rand=mpz(random.getrandbits(self.gam))
            if self.list[i]!=None: yield self.list[i]
            else:
                yield rand

def randList(B):
    l=[0 for i in range(B)]
    l[random.randint(0,B-1)]=1
    return l

class  pk:
    '''classe para os elementos da chave, publicos e provados'''
    def __init__(self, pr):
        t=-time.clock()
        self.pr=pr
        while True:
            self.p=gmpy2.next_prime(random.getrandbits(pr['eta']))
            if len(self.p)==pr['eta']: break
        #print (self.p)
        self.q0=random.randint(0,  (mpz(2)**pr['gam'])/self.p)
        self.x0=self.q0*self.p

        '''geração da lista de ruido delta'''
        self.se1=int(time.time()) #usando tempo como seed para os rngs.
TODO: passar tempo para função de Hash com retorno int
        self.f1=rng(self.se1, pr['gam'], pr['tau'])
        self.delta=[(chi%self.p)+self.p-
random.randint(0,mpz(2)**(pr['lam']+pr['eta']))* self.p - random.randint(-
(mpz(2)**pr['rho']), mpz(2)**pr['rho']) for chi in self.f1]

        '''geração do vetor s'''
        assert pr['Theta']%15==0 #Theta tem que ser um multipli de theta
        B=int(pr['Theta']/15)
        self.s=[1]+[0 for i in range(B-1)]
        for i in range(15-1):
            self.s=self.s+randList(B)
```

```
        '''geração do elemento u1'''
        self.kappa=pr['gam']+pr['eta']+2
        self.se2=int(time.time())
        self.f2=rng(self.se2, self.kappa+1, pr['Theta'])
        self.f2[0]=0 #seta elemento u1 como sendo zero
        somatorio=0
        i=0
        for u in self.f2:
            somatorio += u*self.s[i]
            i+=1
        soma=mpz(round((mpz(2)**self.kappa)/self.p))
        self.u1=soma-somatorio

        '''geração de deltaPrime = encryptações do vetor sigma'''
        self.se3=int(time.time())
        self.f3=rng(self.se2, pr['gam'], pr['Theta'])
        self.deltaPrime=[chi%self.p + random.randint(0,
mpz(2)**(pr['gam']+pr['eta']/self.p)/self.p) * self.p - 2*random.randint(-
(mpz(2)**pr['rho']), mpz(2)**pr['rho']) -si for chi,si in zip(self.f3,
self.s)]

        '''geração da chave para picles'''
        self.pkAsk={'se1':self.se1, 'x0':self.x0, 'delta': self.delta}
        self.pk={'pkAsk':self.pkAsk, 'se2': self.se2, 'u1': self.u1, 'se3':
self.se3, 'deltaPrime': self.deltaPrime, 'pr':self.pr}

        self.sk=self.s
        t+=time.clock()
        #print('keygem em: ', t, 'segundos  par==', pr['ty'])


    def returnKeys(self):
        return self.sk, self.pk

def encrypt(pk, m): #todo: criar classe para cyphertext, com noise, e
depth, assim como CORON et. al.
    assert m==0 or m==1
    x0=pk['pkAsk']['x0']
    f1=rng(pk['pkAsk']['se1'], pk['pr']['gam'], pk['pr']['tau'])
    soma=0
    for chi, delta in zip(f1, pk['pkAsk']['delta']):
        soma += (chi-delta)*random.randint(0, mpz(2)**pk['pr']['alpha'])
    r=random.randint(-mpz(2)**pk['pr']['rho'], mpz(2)**pk['pr']['rho'])
    c=modNear(m+2*r+2*soma, x0)
    return c

def add(pk, c1, c2):
    return modNear(c1+c2, pk['pkAsk']['x0'])

def mul(pk, c1, c2):
    return modNear(c1*c2, pk['pkAsk']['x0'])

def expand(pk, c):
    #mensagem do programador: sacrifique um bode quando chamar essa função
    n=4 #bits of precision for the u1/2**kappa (u1)has len=148450bits
    pr=pk['pr']
    kappa=pr['gam']+pr['eta']+2
    f2=rng(pk['se2'], kappa+1, pr['Theta'])
    f2[0]=pk['u1']
    gmpy2.get_context().precision=300000
    k=mpz(2)**kappa
```

```
    y=[u/k for u in f2]
    z=[modNear(c*yi, 2) for yi in y]
    gmpy2.get_context().precision=n
    zprime=[float(zi) for zi in z]
    gmpy2.get_context().precision=300000
    return zprime

def decrypt(sk, c, z):
    e=zip(sk, z)
    soma=0
    for si, zi in zip(sk, z):
        soma+= si*zi
    soma=int(round(soma))
    m=(c-soma)%2
    return m

def recrypt(pk, c, z):
    pass

def keyValidation(pk_, sk):
    '''verifica se a chave passada gera decriptações validas de '0' e '1'
'''
    c=encrypt(pk_, 0)
    c1=encrypt(pk_, 1)
    z=expand(pk_, c)
    z1=expand(pk_,c1)
    m=decrypt(sk, c, z)
    m1=decrypt(sk, c1, z1)
    if m==0 and m1==1: return True
    else: return False

def testeKeyGen():
    '''teste apenas para geração de chaves'''
    #parametros de acordo com coron.
    toy=    {'ty':"toy",   'lam':42,'rho':26,'eta':988, 'gam':147456,
'Theta':150, 'pksize':0.076519, 'seclevel':42.0,'alpha':936, 'tau':158}
    small=  {'ty':"small", 'lam':52,'rho':41,'eta':1558,'gam':843033,
'Theta':555, 'pksize':0.437567, 'seclevel':52.0,'alpha':1476,'tau':572}
    medium= {'ty':"medium",'lam':62,'rho':56,'eta':2128,'gam':4251866,
'Theta':2070,'pksize':2.207241, 'seclevel':62.0,'alpha':2016,'tau':2110}
    large=  {'ty':"large",
'lam':72,'rho':71,'eta':2698,'gam':19575950,'Theta':7965,'pksize':10.303797
,'seclevel':72.0,'alpha':2556,'tau':7659}

    print('iniciando teste de geração de chaves para Coron 440')
    for par in [toy, small, medium, large]:
        kt=-time.clock()
        key= pk(par)
        kt+=time.clock()
        print('%s gerada em %f segundos' %(par['ty'], kt))

def teste2():
    '''teste apenas para geração de chaves'''
    #parametros de acordo com coron.
    toy=    {'ty':"toy",   'lam':42,'rho':26,'eta':988, 'gam':147456,
'Theta':150, 'pksize':0.076519, 'seclevel':42.0,'alpha':936, 'tau':158}
    key= pk(toy)
    sk, pk_ =key.returnKeys()
    e=encrypt(pk_, 1)
    print (len(e)*64)
```

```
def teste():
    import gc
    print('pyFHE, baseado no trabalho 440 de coron')
    #parametros de acordo com coron.
    toy=      {'ty':"toy",   'lam':42,'rho':26,'eta':988, 'gam':147456,
'Theta':150, 'pksize':0.076519, 'seclevel':42.0,'alpha':936, 'tau':158}
    small=  {'ty':"small", 'lam':52,'rho':41,'eta':1558,'gam':843033,
'Theta':555, 'pksize':0.437567, 'seclevel':52.0,'alpha':1476,'tau':572}
    medium= {'ty':"medium",'lam':62,'rho':56,'eta':2128,'gam':4251866,
'Theta':2070,'pksize':2.207241, 'seclevel':62.0,'alpha':2016,'tau':2110}
    large=  {'ty':"large",
'lam':72,'rho':71,'eta':2698,'gam':19575950,'Theta':7965,'pksize':10.303797
,'seclevel':72.0,'alpha':2556,'tau':7659}

    print('Test')
    it=0
    for par in [toy, small, medium, large]:
        kt=-time.clock()
        key= pk(par)
        kt+=time.clock()
        sk, pk_ = key.returnKeys()
        while not keyValidation(pk_, sk):
            key= pk(par)
            sk, pk_ = key.returnKeys()
            print('key not valid')
        print('key found!')
        enctime=-time.clock()
        c=encrypt(pk_, 0)
        enctime+=time.clock()

        #c1=encrypt(pk_, 1)
        etime=-time.clock()
        z=expand(pk_, c)
        etime+=time.clock()
        #z1=expand(pk_,c1)
        mtime=-time.clock()
        m=decrypt(sk, c, z)
        mtime+=time.clock()
        #m1=decrypt(sk, c1, z1)

        #ca=add(pk_, c, c1)
        #cm=mul(pk_, c, c1)
        #ma=decrypt(sk, ca, expand(pk_, ca))
        #m=decrypt(sk, cm, expand(pk_, cm))
        f=open('fhe440.txt', 'a')
        f.write(par[it]['ty'])
        f.write("=="+str(kt)+"\n")
        f.write('Kg= %f, encrypt = %f, decrypt = %f, expand = %f \n\n'
%(kt, enctime, mtime, etime))
        f.close()

        #print("0=>%d  1=>%d  | 0+1=>%d  | 0x1=>%d" %(m, m1, ma, mm))
        gc.collect()
        #write(pk_, 'pk_'+key.pr['ty']+'.bin')
        #write(sk, 'sk_'+key.pr['ty']+'.bin')

if __name__=='__main__':
    #teste()
    print('codigo rodando')
    #testeKeyGen()
    teste2()
```

APÊNDICE B: TESTE DE POTENCIAÇÃO

```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""

import numbapro
import random
import time
from numpy import array

@numbapro.jit
def jited(a,b):
    return a**b

@numbapro.vectorize(['float64(float64, float64)'], target='parallel')
def jitedP(a,b):
    return a**b

def nonJited(a,b):
    return a**b

@numbapro.vectorize(['float64(float64, float64)'], target='gpu')
def jiteGPU(a,b):
    return a**b

if __name__=='__main__':
    print('runing test!')
    print('teste de velocidade de jit: 500 exponenciações de 2^(8e6)')
    for i in range(5):
        t1=-time.clock()
        for i in range(500):
            jited(2,8000000+i)
        t1+=time.clock()

        t2=-time.clock()
        for i in range(500):
            nonJited(2,8000000+i)
        t2+=time.clock()

        t3=-time.clock()
        for i in range(500):
            jitedP(2,8000000+i)
        t3+=time.clock()

        t4=-time.clock()
        for i in range(500):
            jiteGPU(2,8000000+i)
        t4+=time.clock()

        print('jit == %f nonjit == %f jitVecParalel == %f jitVecGPU == %f'
%(t1, t2, t3, t4))
```

APÊNDICE C: CONFIGURAÇÃO DO AMBIENTE

Para a execução desse trabalho, foi utilizada a linguagem de programação Python 3.3. A versão mais recente do Python pode ser baixada no link https://www.python.org/downloads/.

Para os cálculos matemáticos, foi utilizado a biblioteca GMPY2, cuja documentação está disponível em https://gmpy2.readthedocs.org/en/latest/. Download disponível em https://pypi.python.org/pypi/gmpy2. A instalação pode ser feito pelo instalador do Windows. Em caso de sistemas Unix, o comando `python setup.py install` pode ser executado na pasta onde se encontram os arquivos.

A versão mais atual do Numba, necessária para utilizar a compilação dinâmica usando LLVM, pode ser baixada em https://github.com/numba/numba.

Nesse trabalho foi utilizada o NumbaPro, cuja licença de uso acadêmico foi disponibilizada pela Continuum. A documentação básica do NumbaPro se encontra disponível em http://docs.continuum.io/numbapro/ . Para se utilizar o NumbaPro, o recomendado é usar a IDE pré-configurada *Anaconda Accelerate,* disponível em https://store.continuum.io/cshop/accelerate/.