

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

LUAN CARDOSO DOS SANTOS

**IMPLEMENTAÇÃO DO ESQUEMA TOTALMENTE HOMÓMORFICO
SOBRE INTEIROS COM CHAVE REDUZIDA**

**MARÍLIA
2014**

LUAN CARDOSO DOS SANTOS

**IMPLEMENTAÇÃO DO ESQUEMA TOTALMENTE HOMOMÓRFICO
SOBRE INTEIROS COM CHAVE REDUZIDA**

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: Dr. Fábio Dacêncio Pereira

MARÍLIA
2014



CENTRO UNIVERSITÁRIO EURÍPIDES DE MARÍLIA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

TRABALHO DE CONCLUSÃO DE CURSO – AVALIAÇÃO FINAL

Luan Cardoso dos Santos

IMPLEMENTAÇÃO DO ESQUEMA TOTALMENTE HOMOMORFICO SOBRE INTEIROS COM CHAVE REDUZIDA

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: 6 (Bo)

Orientador: Fábio Dacêncio Pereira

1º. Examinador: Rodolfo Barros Chiaramonte

2º. Examinador: Bruno Marques dos Santos







Marília, 02 de dezembro de 2014.

SANTOS, Luan Cardoso dos

Implementação do esquema totalmente homomórfico sobre inteiros com chave reduzida/ Luan Cardoso dos Santos; orientador: Profº. Dr. Fábio Dacêncio Pereira. Marília, SP: [s.n.], 2014.

73 folhas

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

Dedico este trabalho a meu pai, Devaldite, sem o qual eu não seria metade do homem que sou hoje.

AGRADECIMENTOS

Agradeço a minha família, meus amigos e professores. Um agradecimento especial a meu professor orientador, Fábio Dacêncio, por ajudar a fomentar meu desenvolvimento acadêmico. Em especial agradeço também ao meu grande amigo e colega de pesquisa, Guilherme Rodrigues Bilar, por sua amizade e apoio em todos esses anos do curso superior. Ainda, agradeço a Michael Trautsch, pelo apoio nas horas mais difíceis, a pela sua amizade, verdadeira mesmo que a distância seja grande. E, agradeço a minha Mãe, Enezina, por ser a melhor mãe que qualquer pessoa poderia desejar.

Agradeço a Viv pela companhia, e a Fahl pela amizade. Agradeço ao rosto sem nome, por motivos que a mim escapam.

Agradeço, também, ao Stark. Dois anos tão curtos em sua companhia, mas, que carregarei com carinho o resto de minha vida.

E, a todos aqueles que passaram por minha vida nesses anos, sejam responsáveis por bons momentos ou por lágrimas, meu profundo obrigado, pois, colaboraram para que eu me tornasse quem eu sou hoje.

Por fim, agradeço ao CNPq pela bolsa concedida durante o último ano de curso, e aos professores do curso de ciência da computação, pela dedicação e excelência no trabalho de ensinar.

Ex Nihilo Omnia Fiunt

RESUMO

Nesta monografia é descrito o trabalho de implementação do esquema totalmente homomórfico de chave reduzida (DGHV sobre inteiros) proposto por Jean-Sébastien Coron, Avradip Mandal, David Naccache e Mehdi Tibouchi, que foi publicado na conferencia CRYPTO 2011, 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 $\tilde{O}(\lambda^{10})$, o que de acordo com Coron (CORON et al, 2011), torna inviável a aplicação em sistemas práticos. O esquema totalmente homomórfico DGHV com chave pública reduzida diminui o tamanho da chave pública gerada para aproximadamente $\tilde{O}(\lambda^7)$, criptografando de maneira quadrática os elementos da chave pública, ao invés de criptografá-los de maneira linear. Para esse trabalho, foram utilizadas as linguagens de programação Python, contando com a biblioteca de matemática e teoria numérica GMPY2.

Palavras-Chave: Criptografia; Homomorfismo; Pós-Quântica; Inteiros.

ABSTRACT

In this monograph it is described the implementation of a fully homomorphic cryptosystem with key reduction (DGHV over integers), proposed by Jean-Sébastien Coron, Avradip Mandal, David Naccache e Mehdi Tibouchi. This work was published in the CRYPTO 2011 conference. This cryptosystem can be related to the fully homomorphic scheme proposed by Gentry, this one being simpler, yet, this simplicity comes at the cost of having a public key of size in the order of $\tilde{O}(\lambda^{10})$, which, according to Coron (CORON et al, 2011), makes it not suitable for practical applications. The fully homomorphic DGHV scheme with key reduction reduces the key size to the order of $\tilde{O}(\lambda^7)$, encrypting in the public key elements in a quadratic for, instead of the linear way of the original work. In this paper, the Python programming language was used, with GMPY2, a mathematics and number theory library.

Keywords: Cryptography; Homomorphism; Post-quantum; Integers.

LISTA DE ILUSTRAÇÕES

Figura 1. Esquema de <i>bootstrapping</i>	22
Figura 2. Principais esquemas totalmente homomórficos.	22
Figura 3: Resultados do primeiro algoritmo de teste.....	36
Figura 4: Relação entre as primitivas do esquema.	37
Figura 5: Diagrama das classes <i>sk</i> , <i>pk</i> e <i>Parameter</i>	45
Figura 6: Tempo de multiplicação de Matrizes de números aleatórios	51
Figura 7: Relação de tempo de execução de código Numba/Python	51
Figura 8: Fatores de tempo do teste de <i>Multithreading</i>	54
Figura 9: Trabalhos relacionados	56
Figura 10: Comparação de tempos entre as implementações.....	57
Figura 11: Comparação entre primitivas das diferentes implementações	58

LISTA DE TABELAS

Tabela 1. Valores da portas lógicas XOR e AND	19
Tabela 2. Parâmetros	33
Tabela 3: Tempos de execução <i>Multithread (em segundos)</i>	53
Tabela 4: Implementação de Wang et.al.	55
Tabela 5: Tempos de execução.....	56

LISTA DE ALGORITMOS

Código Fonte 1: Primeira implementação de um esquema homomórfico	34
Código Fonte 2: Módulo.....	37
Código Fonte 3: <i>Keygen</i>	38
Código Fonte 4: Geração de inteiros aleatórios ímpares.....	39
Código Fonte 5: Geração de primos	40
Código Fonte 6: Geração do elemento X_0	40
Código Fonte 7: Geração de q e r	40
Código Fonte 8: Lista de elementos X_i	41
Código Fonte 9: Vetores s_b	42
Código Fonte 10: Matrix aleatória u	42
Código Fonte 11: Cálculo de u_{11}	43
Código Fonte 12: Encriptação da chave pública	44
Código Fonte 13: Classes e métodos de escrita.....	44
Código Fonte 14: <i>Encrypt</i>	46
Código Fonte 15: <i>Expand</i>	47
Código Fonte 16: <i>Decrypt</i>	48
Código Fonte 17: Funções lógicas.....	48
Código Fonte 18: Comparação Numba e Python	49
Código Fonte 19: Teste de <i>Multithreading</i>	52

LISTA DE ABREVIATURAS E SIGLAS

FHE	Criptografia totalmente homomórfica (Inglês: <i>Fully homophic Encryption</i>)
SHE	Criptografia parcialmente homomórfico (Inglês: <i>Somewhat homomorphic encryption</i>)
LWE	Sigla em inglês para <i>Learning with errors</i> , uma classe de problemas matemáticos baseado em aprendizagem de máquina.
RWLW	Sigla em inglês para <i>Ring Learning with errors</i> ,
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 <i>Low Level Virtual Machine</i> , sigla em inglês para máquina virtual de baixo nível.

SUMÁRIO

INTRODUÇÃO	16
1.1 Motivação e justificativa	16
1.2 Objetivos Gerais	17
1.3 Organização deste documento	17
1.4 Simbologia adotada	18
2 FUNDAMENTOS TEÓRICOS	18
2.1 Circuitos lógicos.....	18
2.2 Álgebra abstrata.....	19
2.3 Reticulados.....	19
2.4 Criptografia	20
2.4.1 Criptografia assimétrica.....	20
2.5 Homomorfismo	21
2.6 Criptografia totalmente homomórfica	21
2.6.1 A joalheria da Alice	23
2.6.2 Usos de criptografia homomórfica.....	25
3 FHE sobre números inteiros	26
3.1 Parâmetros.....	27
3.2 Construção	27
3.2.1 KeyGen(λ).	28
3.2.2 Encrypt ($\mathbf{pk}, \mathbf{m} \in \{0, 1\}$).....	28
3.2.3 Evaluate($\mathbf{pk}, \mathbf{C}, \mathbf{c1}, \dots, \mathbf{ct}$).....	28
3.2.4 Decrypt (sk, c).	28
3.3 Segurança do sistema criptográfico	29
3.4 Ataques.....	29
4 FHE sobre números inteiros com chave reduzida.....	29
4.1 Construção	30
4.1.1 KeyGen(1λ).	30
4.1.2 Encrypt (pk, $\mathbf{m} \in \{0, 1\}$).....	31

4.1.3	Evaluate	31
4.1.4	Expand pk, c^*	32
4.1.5	Decrypt sk, c^*, z	32
4.1.6	Recrypt pk, c^*, z	32
4.2	Parâmetros concretos e experimentais	33
4.3	Implementação do esquema FHE	33
4.3.1	Primeiro código – SHE simétrico	34
4.3.2	Relação entre as primitivas	36
4.3.3	Geração das chaves	37
4.3.4	Classes e Pickle	44
4.3.5	Encriptação	45
4.3.6	Expansão	47
4.3.7	Decriptação	47
4.3.8	Primitivas AND e XOR	48
4.3.9	Recrypt	49
4.4	Melhorando o Desempenho	49
4.4.1	Tecnologias para melhoria de desempenho	49
4.4.2	Proposta de melhoria de desempenho	54
5	Resultados	55
5.1	Trabalhos correlatos	55
5.2	Metodologia dos testes	56
5.3	Resultados	56
5.4	Conclusões	57
5.5	Publicações	58
5.6	Trabalhos futuros	58
	REFERÊNCIAS	59
	APÊNDICE A - Códigos	61
1	keyGen.py	61
2	fheKey.py	66
3	__main__.py	67
4	Encrypt.py	68
5	parameters.py	71
	APÊNDICE B – Configuração da IDE	73

INTRODUÇÃO

Desde a antiguidade, a privacidade tem sido uma necessidade constante que deu frutos as mais diversas técnicas e tecnologias que permitem que a informação seja compartilhada de forma segura entre apenas partes pré-estabelecidas. Desde as primeiras cifras, usadas pelos romanos, até os sistemas criptográficos modernos, um longo caminho foi percorrido.

A criptografia moderna tem por base problemas matemáticos relacionados à teoria dos números, onde, de forma simplista, o processo de quebra dessa segurança, apesar de possível, é impraticável devido às necessidades imensas de tempo e poder de processamento. Um exemplo disto é o algoritmo RSA, que usa a fatoração de números inteiros em números primos como seu problema-base (RIVEST et. al, 1978). Enquanto a multiplicação de dois números é um problema facilmente solucionado por computadores convencionais, a fatoração de um número é uma tarefa complexa, que não pode ser executada em tempo polinomial ao tamanho da entrada.

Porém, com o advento da teoria de computadores quânticos, e com a construção desses estando cada vez mais próxima da realidade, a ideia de que a fatoração de números inteiros é segura está se tornando nebulosa (SHOR, 1994). Com isso, uma nova classe de algoritmos surge, chamados algoritmos criptográficos pós-quânticos. Esses algoritmos usam problemas matemáticos diferentes como base de sua construção, mantendo o tempo de ataque exponencial, mesmo contra computadores quânticos (BERNSTEIN, 2008). Dentro desses ditos sistemas criptográficos pós-quânticos podemos destacar os sistemas completamente homomórficos, que possuem a interessante característica de permitir que computações sejam executadas nos dados criptográficos.

1.1 Motivação e justificativa

Sistemas criptográficos totalmente homomórficos são capazes de executar processamento arbitrário diretamente na mensagem cifrada, sem a necessidade de se decifrar essas informações. Tal característica, aliada ao fato de também serem sistemas criptográficos de chave assimétrica, abre uma pletera de usos diferentes.

O exemplo canônico para os usos de sistemas criptográficos é em sistemas de votação, onde, o processamento dos dados dos votos é feito de forma que esses dados nunca

sejam decriptados, garantindo assim o sigilo do voto, mas, permitindo que a contagem destes seja pública.

Ainda, um esquema de criptografia totalmente homomórfico é capaz de prover um serviço de banco de dados encriptados, onde, em um cenário hipotético, o servidor possuiria um banco de dados criptografados e, seria capaz de executar *queries* criptografadas e retornar dados criptografados para o cliente, sem, em nenhum momento, ter informações sobre a consulta, nem sobre a informação retornada por ela.

De forma geral, um esquema totalmente homomórfico prático permite que computação segura sobre dados privados possa ser executada em servidores não seguros, mantendo o sigilo dos dados processados. Entretanto, a criptografia totalmente homomórfica ainda é uma área nova dentro da criptografia, e, ainda são necessárias pesquisas e melhorias para trazer essa tecnologia ao âmbito prático do nosso dia a dia.

1.2 Objetivos Gerais

O objetivo dessa monografia é discutir de forma breve os fundamentos da criptografia homomórfica, e descrever, de forma completa um esquema totalmente homomórfico, assim como propor uma implementação de um sistema completamente homomórfico em uma linguagem de alto nível. O sistema escolhido foi o DGHV (DIJK et. al, 2010), com alterações propostas por Coron (CORON et. al, 2011). A linguagem escolhida para essa implementação foi Python 3, juntamente com a biblioteca de teoria numérica GMPY2. Além disso, também serão feitas propostas de *speedup* do código usando compilação em tempo de execução e paralelismo em CPU.

1.3 Organização deste documento

Esse trabalho está dividido 4 partes: Primeiramente serão apresentados conceitos e fundamentos necessários para a correta compreensão do esquema homomórfico, assim como uma breve introdução à criptografia. Em seguida, serão apresentados os dois esquemas homomórficos aqui abordados: O esquema DGHV original e o esquema modificado proposto por Coron, junto da implementação em Python. Por fim, serão apresentados os resultados obtidos nesse trabalho de graduação.

1.4 Simbologia adotada

Nesse trabalho será adotada a mesma notação matemática que nos trabalhos de Gentry (GENTRY e HALEVI, 2011) e Coron (CORON, 2011). Denota-se por $\lceil x \rceil$, $\lfloor x \rfloor$ e $\{x\}$ os arredondamentos de x respectivamente para o maior inteiro, para o menor inteiro e para o inteiro mais próximo. Dado um número real z e um número inteiro p , $[z]_p$ denota a redução de z módulo p com $-p/2 < [z]_p \leq p/2$. $[z]_p$ também é denotado por $z \bmod p$. Nas notações de intervalo, o símbolo de colchete indica intervalo fechado, enquanto parênteses indicam um intervalo aberto. Por exemplo $(a, b] = \{x \in \mathbb{R} | a < x \leq b\}$, e $[a, b) = \{x \in \mathbb{R} | a \leq x < b\}$.

2 FUNDAMENTOS TEÓRICOS

Nesse capítulo serão apresentados os conceitos básicos utilizados pelos esquemas de criptografia homomórfica, assim como descrições mais aprofundadas sobre criptografia em geral e criptografia homomórfica. Esse capítulo não tem por objetivo se aprofundar em bases matemáticas ou discutir em profundidade os assuntos apresentados, apenas disponibilizar ao leitor fundamentos para os conceitos a serem posteriormente apresentados.

2.1 Circuitos lógicos

Todas as operações executadas por computadores, desde a mais simples soma até complexas simulações de dobra de proteínas, em última análise, são simples operações lógicas e aritméticas básicas, tais como somar, complementar, comparar e mover bits. Essas funções são fisicamente realizadas por circuitos eletrônicos chamados circuitos lógicos. De forma simplista e informal, esse modelo de processamento baseado na lógica booleana é equivalente a uma máquina de Turing, sendo capaz de realizar qualquer algoritmo arbitrário.

Levando-se em conta a criptografia homomórfica, em específico os algoritmos que serão apresentados nesse trabalho, as computações a serem realizadas no texto cifrado devem ser convertidas em circuitos lógicos formados pelas portas XOR e AND.

A função ou exclusivo ou disjunção exclusiva, também conhecida pelos acrônimos XOR e EXOR é uma operação lógica que retorna um valor verdadeiro se, e apenas se um dos operandos possuir valor verdadeiro. A função “E”, também chamada de conjunção lógica, é, similarmente, uma operação lógica que resulta em um valor verdadeiro se e apenas se todos

os operandos possuem valor verdadeiro. Na Tabela 1 é mostrada a tabela verdade para as portas lógicas XOR e AND.

Tabela 1. Valores das portas lógicas XOR e AND

a	b	a XOR b	a AND b
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

2.2 Álgebra abstrata

Dentro dos conceitos matemáticos nos quais se baseiam a criptografia homomórfica, um dos mais importantes é o conceito de anel algébrico.

Um anel é uma construção matemática abstrata, composto de um conjunto de elementos e duas operações, geralmente soma e multiplicação. Um anel é construído de tal forma que $(R, +)$ e $(R, *)$ é um grupo abeliano, e, as duas operações são relacionadas entre si pela propriedade distributiva (BEACHY, 2000).

Ainda, dado dois anéis R e S , um homomorfismo h é uma função entre os anéis que preserva as operações de soma e multiplicação, mapeando os resultados de um anel para o outro (GENTRY, 2009).

Matematicamente:

$$h(r1 + r2) = h(r1) + h(r2)$$

$$h(r1 \times r2) = h(r1) \times h(r2)$$

2.3 Reticulados

O esquema original, proposto por Gentry, foi baseado em reticulados ideias. Em termos gerais, reticulados ideais são uma classe especial de reticulados e uma generalização de reticulados cíclicos. Seu uso na criptografia deve-se ao fato de ser capaz de diminuir pela raiz quadrada o número de parâmetros necessários para se descrever um reticulado (GENTRY, 2009). Os reticulados também são utilizados como base para outros esquemas criptográficos pós-quânticos não necessariamente homomórficos, e, também, como base para os esquemas homomórficos LWE e RLWE (BRAKERSKI e VAIKUNTANATHAN, 2014).

2.4 Criptografia

Criptografia, cujo nome tem origem nas palavras gregas para “escrita escondida”, é o estudo dos princípios e as técnicas pelas quais dados podem ser tornados ilegíveis, para sua proteção, e, posteriormente tornados novamente legíveis pelo, e somente pelo destinatário.

Inicialmente, em tempos antigos, a criptografia era utilizada na troca de mensagens, principalmente nos assuntos ligados a guerra, com o principal intuito de manter sigilosas informações do inimigo. Um das mais clássicas técnicas utilizadas para cifrar uma mensagem era a “Cifra de César”, uma simples cifra que substituição onde cada letra da mensagem original era substituída por outra, deslocada um certo número de casas no alfabeto. Embora simples, essa cifra foi o passo inicial para a criação de toda uma área na teoria da informação.

Para se ilustrar a importância da criptografia, durante a segunda grande guerra, os alemães possuíam uma máquina de cifra, conhecida como máquina Enigma, que era utilizada para a codificação e decodificação de mensagens. A quebra da cifra da máquina Enigma foi um trunfo dos aliados, e, um dos fatores que levaram ao resultado da guerra, com a vitória aliada.

Nos dias atuais, a criptografia está presente no nosso dia a dia, mesmo que muitas vezes passe despercebida. De bancos a e-mails, de autenticação em sites de compra e transações financeiras a usos militares, aplicações que necessitem de segurança acabam por utilizar a criptografia.

2.4.1 Criptografia assimétrica

Os algoritmos de criptografia atuais podem ser divididos em dois grandes grupos: criptografia de chave simétrica e criptografia de chave assimétrica. O principal fator que diferencia uma da outra é que, enquanto na criptografia simétrica a mesma chave encripta e decripta a informação, na criptografia assimétrica, essas duas tarefas são realizadas por chaves distintas. Com isso, em um algoritmo de chaves assimétricas, uma chave pode ser tornada pública, enquanto outra é mantida secreta. Isso cria uma gama de usos que a criptografia de chave simétrica não abrange. Por exemplo, podem-se ter esquemas que são capazes de garantir tanto a confidencialidade quanto a autenticidade de uma mensagem (STALLINGS, 2007).

Um dos algoritmos utilizados em esquemas de chave pública é o algoritmo RSA (RIVEST, SHAMIR e ADLEMAN, 1978), criado por três professores do Instituto de

Tecnologia de Massachusetts (MIT), nomeadamente Ronald Rivest, Adi Shamir, e Leonard Adleman. O RSA é uma das mais bem sucedidas implementações de um sistema de chaves assimétrico, fundamentado em teorias clássicas dos números, nomeadamente, exponenciação, módulos e fatoração de números grandes.

2.5 Homomorfismo

Criptografia homomórfica é uma forma de criptografia que permite que computações sejam executadas na informação encriptada, de forma que o resultado seja a forma encriptada dessas mesmas computações executadas sobre a informação decriptada.

Hoje temos vários tipos de sistemas parcialmente homomórficos eficientes, e, certa quantidade de esquemas totalmente homomórficos, com a implementação menos eficiente. Embora sistemas criptográficos que possuam características homomórficas sejam sujeitos a ataques baseados exatamente nessa característica, o homomorfismo pode ser utilizado para se executar processamentos de forma segura.

Sistemas parcialmente homomórficos, ao contrário daqueles ditos totalmente homomórficos, ou possuem homomorfismo para apenas uma operação, ou então são apenas capazes de executar certa quantidade de processamento sobre o texto cifrado antes de se tornar impossível que esse seja decriptado. Como exemplo de sistemas parcialmente homomórficos, pode-se citar RSA, ElGamal (ELGAMAL, 1985), Goldwasser-Micali (GOLDWASSER e MICALI, 1984), Benaloh (BEHALOH, 1994) e Paillier (PAILLIER, 1999), esse último interessante devido a suas aplicações em sistemas de voto eletrônico e de dinheiro eletrônico, com foco em anonimidade.

2.6 Criptografia totalmente homomórfica

Como dito anteriormente, um sistema criptográfico totalmente homomórfico é aquele capaz de executar circuitos lógicos de comprimento abstrato sobre o texto cifrado. O primeiro esquema totalmente homomórfico foi proposto em 2010 por Craig Gentry em sua tese de Doutorado.

O esquema de Gentry partiu de um esquema parcialmente homomórfico (SHE), capaz de executar um número limitado de operações sobre o texto cifrado. Então, Gentry propôs algumas alterações sobre esse esquema que possibilitavam que ele fosse capaz de avaliar o próprio circuito de decriptação de forma homomórfica. Tal procedimento, chamado

de *bootstrapping*, efetivamente diminuí a ruído acumulado no texto cifrado durante cada operação homomórfica, permitindo então que circuitos lógicos de profundidade arbitrária fossem executados no texto cifrado. Na Figura 1 é ilustrado como esse procedimento foi aplicado por Gentry. O esquema de *bootstrapping* de Gentry foi posteriormente utilizado para a construção de outros esquemas criptográficos, tal como o próprio esquema DGHV.

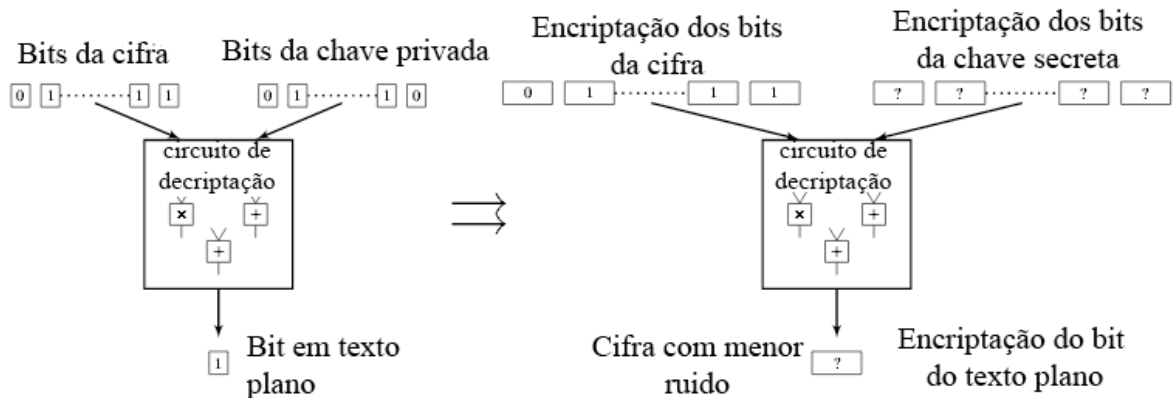


Figura 1. Esquema de *bootstrapping*,

Atualmente existem três principais vertentes para a construção de esquemas homomórficos. Os esquemas de Gentry e Gentry-Halevi, baseados em reticulados ideais para a sua construção; Os esquemas baseados no problema de aprendizado de máquina LWE, proposto por Brakerski; E, os esquemas baseados em anéis de números inteiros, nomeadamente o esquema DGHV. Na Figura 2 é apresentada uma árvore desses esquemas.

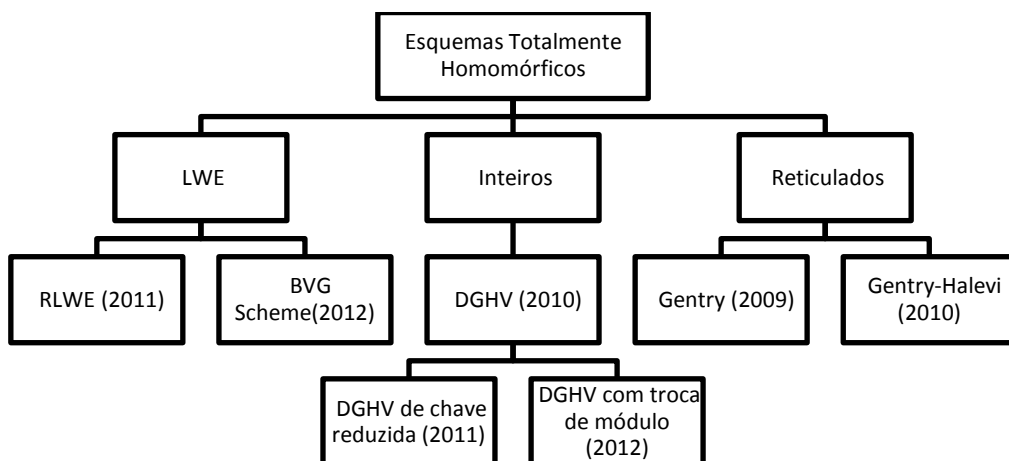


Figura 2. Principais esquemas totalmente homomórficos.

Fonte própria.

A seguir, é apresentado de forma informal o processo de criação desse esquema, em uma tradução livre da analogia disponível no trabalho de Gentry “*Computing arbitrary*

functions of encrypted data” (GENTRY, 2010).

2.6.1 A joalheria da Alice

À primeira vista, o conceito de criptografia totalmente homomórfica pode parecer inadequado, difícil de entender, ou até mesmo paradoxal. Como forma de criar algumas intuições e permitir que o esquema fosse compreendido de forma mais fácil e até mesmo lúdica, Gentry utilizou-se de uma analogia, onde o esquema FHE foi trazido para uma versão ficcional do “mundo real”:

“Alice é a dona de uma joalheria. Nessa joalheria temos materiais preciosos, como ouro, diamantes e prata. Alice deseja transformar esses materiais brutos em joias intrincadas, anéis e colares. Porém, Alice não confia nos seus empregados e assume que eles irão roubar tanto as joias ou os materiais brutos, se houver uma oportunidade. Em outras palavras, Alice deseja que os trabalhadores *processem* os materiais em objetos finais, mas sem permitir *acesso* aos materiais. Qual a solução de Alice para esse problema??

Felizmente, Alice possui um plano: Ela utilizará uma caixa transparente impenetrável, com luvas para manipular os objetos dentro. Alice então colocará os materiais brutos dentro da caixa, e, irá trancá-la. Então, os funcionários trabalharão esses materiais utilizando as luvas. Alice, possuindo a única chave, será a única que conseguirá remover os materiais de dentro dessa caixa. Nesse cenário, a caixa com luvas e materiais preciosos dentro, representa os dados iniciais criptografados m . As luvas representam a flexibilidade, ou homomorfismo do esquema, que permite que os dados sejam manipulados sem serem decriptados. E, por fim, a joia dentro da caixa representa a encriptação da função $f()$ aplicada sobre os dados m . Vale notar que, nesse cenário, a proteção dos dados é representada pela falta de acesso físico, e, não pelo funcionário sendo capaz de ver o que acontece dentro da sala. Para uma analogia dessa forma, um quarto escuro para revelação fotográfica pode ser mais interessante.

Então, anedoticamente, Alice, tendo achado a solução para suas aflições, compra caixas com luvas na indústria Acme. Infelizmente, as luvas nas caixas possuem um defeito de fabricação, que faz com que elas fiquem duras após um minuto de trabalho. Um grande problema para Alice, já que alguns dos designs necessitam de bem mais tempo do que isso para serem montados. Essa característica, retornando ao sistema parcialmente homomórfico usado como base para o FHE, é a representação do ruído acumulado no texto cifrado, que se acumula a cada operação homomórfica executada nele. Existe uma forma de Alice utilizar

essas caixas com defeito para montar as joias mais complexas?

Alice então nota que, apesar de defeituosas, as caixas possuem algumas características interessantes. Elas possuem slots de entrada, no qual é possível apenas colocar objetos, e não remover, como uma caixa de correios. Essas caixas também são flexíveis o suficiente para serem dobradas e colocadas uma dentro da outra. Será possível tirar proveito dessas características para resolver esses problemas?

Felizmente, Alice descobre uma forma, após um sonho de cavernas cheias de ouro, e, dragões devorando a própria cauda, uma forma de se utilizar as caixas para montar mesmo as joias mais complexas!

Como inicialmente imaginado, Alice entrega ao funcionário a caixa trancada, com os materiais brutos dentro. E, mais ainda, entrega várias outras caixas com chaves dentro delas mesmas. O funcionário pode, então, trabalhar na joia até o momento em que as luvas se tornam rígidas, que, em nossa analogia, representa o ruído no texto cifrado alcançando um patamar em que qualquer operação executada nele tornaria impossível a recuperação dos dados. O funcionário, então, coloca a caixa, com os materiais, e a joia parcialmente montada dentro de outra caixa. Como essa possui a chave para a primeira caixa, ele pode abrir a caixa interna, remover os materiais, e continuar a montagem. Alice observa que essa solução somente funciona se o funcionário pode abrir a caixa i dentro da caixa $i+1$ e poder executar um pouco de trabalho antes das luvas endurecem novamente. Contudo, contanto que a operação de destravar, e um pouco de trabalho na joia em si, possam ser executados em menos tempo do que o necessário para que as luvas parem de funcionar, qualquer tipo de joia, não importa a complexidade, pode ser feita dessa forma.

Nesse ponto, a analogia trás o conceito de *bootstrapping*, que foi proposto como executar, de forma homomórfica, o circuito de deciptação no texto encriptado, com o intuito de se reduzir o ruído acumulado. Apesar de essa analogia ser de fato, interessante como ferramenta para mais fácil entender o conceito de criptografia homomórfica, torna-se difícil ilustrar outros problemas enfrentados por Gentry. De forma simplista, o sistema SHE proposto por ele não era capaz de executar o seu circuito de deciptação de forma homomórfica. Gentry, então, propôs modificações nas primitivas responsáveis pela encriptação e geração das chaves, junto com alguns parâmetros, para tornar o circuito de deciptação o mais simples possível. Assim, o ruído gerado pela função de recriptação é menor do que o ruído que ela consegue remover do texto cifrado, deixando espaço para executar computações na informação cifrada. Voltando a analogia, isso seria o equivalente a se lubrificar as fechaduras das caixas para que, o processo de destranca-las seja executado mais rapidamente.”

2.6.2 Usos de criptografia homomórfica

Um sistema totalmente homomórfico tem muitas aplicações, sendo o exemplo canônico seu uso em sistemas de votação. Porém, um esquema totalmente homomórfico que seja eficiente pode ser utilizado para se delegar processamentos de dados sensíveis para ambientes em nuvem. *Queries* em bancos de dados criptografados, onde, uma consulta criptografada retorna dados, também criptografados, sem nunca deixar à vista os dados decryptados.

2.6.2.1 Servidores na nuvem

Um sistema criptográfico totalmente homomórfico pode ser utilizado para delegar operações que não podem, por um motivo ou outro, serem executadas no computador local. Dados são enviados para um servidor na nuvem, e esse será responsável por realizar operações nesses dados criptografados, na forma de um circuito lógico que será executado sobre os bits criptografados. O Servidor, então, envia esses dados de volta para o usuário, que os decripta usando a sua chave secreta. Note que, a característica mais importante apresentada nesse cenário é que os dados nunca são decryptados fora do ambiente do usuário.

2.6.2.2 Validação de dados em ambientes não confiáveis

Similarmente ao uso em servidores na nuvem, um esquema FHE pode ser utilizado para validar informações em aplicações não confiáveis. Suponha um cenário em que um usuário deseje utilizar um serviço que necessite de seus dados pessoais para validação, porém, tal serviço não é confiável. Nesse caso, os seguintes passos devem ser tomados: O usuário fará a requisição de seus dados ao repositório central confiável, que, então, gerará um par de chaves criptográficas. Esse servidor, então, enviará os dados encriptados e chave pública de volta ao usuário. A aplicação não confiável receberá esses dados e computará funções de verificação de forma homomórfica, chegando então, a um resultado booleano e criptografado pela chave pública, representando se esses dados são válidos ou não. A aplicação não confiável, então, enviará os dados computados de volta ao servidor central, que, de posse da chave privada armazenada, irá decriptar o resultado e retornará o booleano correspondente.

Com isso, o usuário poderá utilizar o serviço sem por em risco suas informações pessoais.

2.6.2.3 Banco de dados Criptografado

Outro possível uso para um esquema de criptografia totalmente homomórfico são bancos de dados criptografados, onde, não somente os dados são criptografados, como o

índice deles e, mais ainda, a própria *query* de consulta ao banco de dados é criptografada. Dessa forma, um cliente é capaz de realizar consultas e recuperar dados em um banco de dados, sem que o servidor consiga ter qualquer informação sobre a *query* ou sobre os dados retornados por ela (BONEH et, al, 2013).

3 FHE SOBRE NÚMEROS INTEIROS

Gentry (GENTRY e HALEVI, 2011) inicia seu trabalho propondo o seguinte esquema de criptografia simétrica:

- *KeyGen*: A chave é um inteiro ímpar, escolhido em um intervalo $p \in [2^{n-1}, 2^n)$
- *Encrypt* (p, m): Para se criptografar um bit m , escolha o texto cifrado como um inteiro cujo módulo por p tenha a mesma paridade que o texto-puro. Em outras palavras, $c = pq + 2r + m$, onde os inteiros $\{q, r\}$ são escolhidos aleatoriamente em intervalos pré-definidos de forma que $2r$ é menor que $p/2$ em valores absolutos
- *Decrypt* (p, c): Calcule $(c \bmod p) \bmod 2$

Nesse esquema, se o ruído r for suficientemente menor do que a chave secreta, ele se torna parcialmente homomórfico, no sentido de que polinomiais de baixo grau podem ser calculados sobre os dados criptografados (Código Python na página 34).

Tal esquema de chave simétrica foi então transformado em um esquema de chave assimétrica, onde a chave pública é constituída de “encriptações de zero”, e, a encriptação de um bit m é definida como m somado a um subconjunto da chave pública.

Gentry define que DGHV sobre inteiros utiliza como base um conjunto de inteiros públicos, $x_i = p \cdot q_i + r_i, 0 \leq i \leq \tau$ onde o inteiro p é secreto (GENTRY e HALEVI, 2011) sendo dado um parâmetro de segurança λ , os seguintes parâmetros devem ser utilizados para a construção do sistema. É valido ressaltar que a maior motivação na criação desse esquema é a simplicidade conceitual, nomeadamente, demonstrar que mesmo algo “complexo” como um esquema de criptografia totalmente homomórfico pode ser conseguido partindo-se de sequencias elementares.

3.1 Parâmetros

Nessa seção definimos os parâmetros utilizados na construção do esquema DGHV (DIJK et. al, 2010). Os mesmos parâmetros serão utilizados na variante de Coron.

- 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.
- $\eta \geq \rho \cdot \Theta(\lambda \log^2 \lambda)$ para que seja possível realizar operações homomórficas para avaliar o “circuito de decifração reduzido”.
- $\gamma = \omega(\eta^2 \cdot \log \lambda)$ 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 -bits:

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

3.2 Construção

As primitivas deste esquema são quatro, *KeyGen*, *Encrypt*, *Decrypt* e *Evaluate*. *KeyGen* é a primitiva responsável pela geração de chaves do esquema, *Encrypt* responsável por gerar o texto cifrado, *Decrypt* responsável por decifrar o texto cifrado, até estas três primitivas são comuns e usuais aos esquemas homomórficos (GENTRY e HALEVI, 2011). A primitiva *Evaluate* foi inclusa com o propósito de reduzir o ruído gerado, Dijk (DIJK et al, 2010) descreve o algoritmo de *Evaluate* como:

“O algoritmo Evaluate tem como entrada uma chave pública pk , um circuito C , e uma tupla de textos

cifrados $\vec{c} = \langle c_1, \dots, c_t \rangle$ (um para cada entrada de bit de C), que tem como saída uma outra cifra c'' (DIJK et al, 2010, p.3-4)

Sendo assim, as primitivas para esse esquema são apresentadas a seguir:

3.2.1 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, recomendo a não ser que x_0 seja um número ímpar e ainda $r_\rho(x_0)$. Assim a chave pública é $pk = \langle x_0, x_1, \dots, x_\tau \rangle$.

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

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

$$c = \left[m + 2r + 2 \sum_{i \in S} x_i \right]_{x_0}$$

3.2.3 Evaluate($pk, C, c_1, \dots, c_t$).

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.

3.2.4 Decrypt(sk, c).

Tem como saída $m \leftarrow (c \bmod p) \bmod 2$. Note que $c \bmod p = c - p \cdot \lfloor c/p \rfloor$, e p é um número ímpar. Dessa forma, pode-se executar a decifração pela seguinte formula: $m \leftarrow [c]_2 \oplus [\lfloor c/p \rfloor]_2$.

3.3 Segurança do sistema criptográfico

Gentry demonstra que esse esquema criptográfico, devido as limitações de escopo de parâmetros e a relação entre eles, é seguro de acordo com o problema do MDC aproximado: Dado um conjunto de inteiros $x_0, x_1, \dots, x_t$, todos escolhidos de forma aleatória próximos a múltiplos de um inteiro longo p , encontrar esse “divisor comum próximo” p . (GENTRY e HALEVI, 2011)

3.4 Ataques

Ainda no seu trabalho, Gentry demonstra uma simples técnica de ataque de força bruta, onde o MDC para os ruídos r'_1, r'_2 é calculado. Essa técnica garante que o número p será encontrado, mas, possui tempo de execução de aproximadamente $2^{2\rho}$. A definição para o problema do MDC aproximado especifica para o esquema homomórfico sobre inteiros é: Dado um número polinomial de amostras da distribuição $\mathcal{D}_{\gamma, \rho}(p)$ para um inteiro ímpar aleatório de η bits p , retorne p .

Outros ataques foram propostos, mas, não serão discutidos no escopo desse trabalho.

4 FHE SOBRE NÚMEROS INTEIROS COM CHAVE REDUZIDA

Coron (CORON et.al, 2011) utilizou uma variante do esquema DGHV, onde foi adicionado um novo parâmetro β , sendo manipulados números inteiros $x'_{i,j}$ na forma de $x'_{i,j} = x_{i,0} \cdot x_{j,1} \bmod x_0$ para $1 \leq i, j \leq \beta$, assim sendo possível reduzir o tamanho das chaves públicas, como descrito:

Apenas 2β inteiros de $x'_{i,j}$ que precisam ser armazenados na chave pública, a fim de gerar os $\tau = \beta^2$ inteiros de $x'_{i,j}$ usados para a criptografia. Em outras palavras, estamos criptografando usando uma forma quadrática nos elementos chaves públicas, em vez de uma forma linear, o que permite reduzir o tamanho da chave pública de τ para $2\sqrt{\tau}$ inteiros de γ bits.

Essa técnica requer a utilização de um x_0 livre de erros, que é $x_0 = q_0 \cdot p$, pois caso contrário o erro iria crescer de maneira muito grande. Além disso, para a criptografia consideramos uma combinação linear de $x'_{i,j}$ com os coeficientes em $[0, 2^\alpha)$, em vez de bits; isto permite reduzir ainda mais o tamanho da chave pública. (CORON et.al, 2011, p. 4-5).

Para tornar o este esquema em um esquema totalmente homomórfico, foram adicionados mais três parâmetros concretos ao mesmo, sendo estes, θ , κ e Θ . Sendo dito:

Concretamente, um usa $\theta = \lambda, \kappa = \gamma + 2 + \lceil \log_2(\theta + 1) \rceil$, e $\Theta = \tilde{O}(\lambda^3)$. Um adiciona a chave pública um conjunto $y = \{y_1, \dots, y_\Theta\}$ dos números racionais em $[0, 2]$ com κ bits de precisão, de tal modo que há um subconjunto esparso $S \subset \{1, \dots, \Theta\}$ de tamanho θ com $\sum_{i \in S} y_i \approx 1/p \bmod 2$. O texto cifrado expandido é calculado usando y_i 's. A chave secreta sk é substituída pelo vector indicador do subconjunto S .

No entanto a adição de Θ elementos y_i , cada um de tamanho κ teríamos uma chave pública de tamanho $\tilde{O}(\lambda^8)$, em vez de $\tilde{O}(\lambda^7)$ como em nossa variante. Portanto, em vez de armazenar y_i na chave pública, nós geramos os y_i utilizando um gerador de números pseudoaleatórios $f(se)$. Então, apenas a semente se e y_1 tem necessidade de ser armazenada na chave pública, e os outros y_i 's podem ser recuperados durante a fase expansão do texto cifrado, aplicando-se $f(se)$ novamente.”(CORON et.al, 2011, p. 9).

4.1 Construção

Coron, em seu trabalho (CORON et.al, 2011), demonstra uma melhoria no esquema DGHV, principalmente com relação com comprimento da chave pública do esquema SHE. Originalmente a chave pública possuía um comprimento na ordem de $O(\lambda^{10})$, foi reduzida para $O(\lambda^7)$. A principal técnica para se conseguir essa diminuição foi armazenar apenas um subconjunto da chave pública, e, recuperar a chave completa *on-the-fly* por meio de multiplicações dos elementos desse subconjunto. Coron também demonstra que seu sistema modificado ainda é semanticamente seguro, porém sobre uma variante do problema do MDC aproximado.

A seguir é apresentada uma descrição completa do esquema proposto por Coron, conforme disponível em (CORON et.al, 2011).

4.1.1 KeyGen(1^λ).

Gera um número ímpar p com η bits. Escolhe um número inteiro $q_0 \in [0, 2^\eta/p)$, escolhido como um produto aleatório de λ^2 -bits primos, e $x_0 = q_0 \cdot p$. Gera β pares inteiros de $x_{i,0}, x_{i,1}$ dentro do intervalo de $1 \leq i \leq \beta$: $x_{i,b} = p \cdot q_{i,b} + r_{i,b}$, $1 \leq i \leq \beta$, $0 \leq b \leq 1$, onde,

$r_{i,b}$ são inteiros entre $(-2^\rho, 2^\rho)$ e $q_{i,b}$ são inteiros aleatórios de ordem $[0, q_0)$. Sendo assim $pk^* = (x_0, x_{1,0}, x_{1,1}, \dots, x_{\beta,0}, x_{\beta,1})$.

Também gera os vetores $s^{(0)}$ e $s^{(1)}$, de comprimento $\lceil \sqrt{\Theta} \rceil$, que seguem a condição de que $s_1^{(0)} = s_1^{(1)} = 1$, para cada um dos $\kappa \in [0, \sqrt{\theta})$ e $b = 0,1$, onde há pelo menos um bit não zero entre os $s_i^{(b)}$'s, $k\lceil \sqrt{B} \rceil < i \leq (k+1)\lceil \sqrt{B} \rceil$, com $B = \Theta/\theta$, e S sendo $S = \{(i,j): s_i^{(0)} \cdot s_j^{(1)} = 1\}$, contendo exatamente θ elementos.

Inicializa um gerador f de números pseudoaleatórios de todo o sistema com uma semente aleatória se , usando assim $f(se)$ para gerar $u_{i,j} \in [0, 2^{\kappa+1})$ para $1 \leq i,j \leq \lceil \sqrt{\Theta} \rceil$, $(i,j) \neq (1,1)$. Então, atribuindo $u_{1,1}$ de forma que:

$$\sum_{(i,j) \in S} u_{i,j} = x_p \text{ mod } 2^{\kappa+1}$$

Onde $x_p \leftarrow \lfloor 2^\kappa/p \rfloor$. Assim, computando a cifra de $\sigma^{(b)}$ dos vetores $s^{(b)}$, escolhendo para cada $i \in [1, \lceil \sqrt{\Theta} \rceil]$ e $b = 0,1$, inteiros aleatórios $r'_{i,b} \in (-2^\rho, 2^\rho)$ e $q'_{i,b} \in [0, q_0)$, é determinado que:

$$\sigma_i^{(b)} = s_i^{(b)} + 2r'_{i,b} + p \cdot q'_{i,b} \text{ mod } x_0$$

Desde modo, temos como saída da primitiva $KeyGen()$, a chave privada sendo $sk = (s^{(0)}, s^{(1)})$, e a chave pública como $pk = (pk^*, se, u_{1,1}, \sigma^{(0)}, \sigma^{(1)})$.

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

Escolha uma matriz de números aleatórios $b = (b_{i,j})_{1 \leq i,j \leq \beta} \in [0, 2^\alpha)^{\beta \times \beta}$ e um inteiro aleatório r no intervalo $(-2^{\rho'}, 2^{\rho'})$. Retorne o texto cifrado como:

$$c^* = m + 2r + 2 \sum_{1 \leq i,j \leq \beta} b_{ij} \cdot x_{i,0} \cdot x_{j,1} \text{ mod } x_0$$

4.1.3 Evaluate

$\text{Add}(pk, c_1^*, c_2^*)$. Retorna $c_1^* + c_2^* \bmod x_0$

$\text{Mult}(pk, c_1^*, c_2^*)$. Retorna $c_1^* \cdot c_2^* \bmod x_0$

4.1.4 Expand (pk, c^*).

Esse procedimento, a expansão do texto cifrado, recebe um texto cifrado c^* e computa a matriz associada z . Podemos pensar nesse procedimento como separado tanto do *Encrypt* quanto de *Decrypt*, já que pode ser executada de forma pública utilizando apenas o texto cifrado e dados públicos. Para tal, para cada $1 < i, j < \sqrt{\theta}$, primeiramente compute o número inteiro aleatório $u_{i,j}$ usando o gerador de números pseudoaleatórios $f(\text{se})$, então calcule $y_{i,j} = u_{i,j}/2^\kappa$ e então compute $z_{i,j}$:

$$z_{i,j} = [c^* \cdot y_{i,j}]_2$$

Mantendo apenas $n = \lceil \log_2(\theta + 1) \rceil$ bits de precisão após o ponto binário. Defina a matriz $z = (z_{i,j})$. Retorne o texto cifrado *expandido* $c = (c^*, z)$.

4.1.5 Decrypt (sk, c^*, z).

$$\text{Retornar } m \leftarrow \left[c^* - \left[\sum_{i,j} s_i^{(0)} \cdot s_j^{(1)} \cdot z_{ij} \right] \right]_2$$

4.1.6 Recrypt (pk, c^*, z).

Para se executar o procedimento de *recrypt*, aplique o circuito de decriptação ao texto cifrado expandido Z e às chaves secretas encriptadas $\sigma_i^{(b)}$. Retorne o resultado como o texto cifrado renovado c_{new}^* .

Assim define-se de maneira completa o 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*. Essas primitivas mapeiam no espaço do texto-puro, respectivamente, as funções lógicas XOR e AND.

4.2 Parâmetros concretos e experimentais

Com as análises executadas em sua variante do esquema DGHV, levando em conta os ataques conhecidos, Coron chegou aos valores da Tabela 2. Parâmetros para os parâmetros do esquema.

Tabela 2. Parâmetros

Parâmetro	λ	ρ	η	γ	β	θ	θ
Toy	42	16	1088	$1,6 \times 10^5$	12	144	15
Small	52	24	1632	$8,6 \times 10^5$	23	533	15
Medium	62	32	2176	$4,2 \times 10^6$	44	1972	15
Large	72	39	2652	$1,9 \times 10^7$	88	7897	15

Fonte: Coron 2011

Vale ressaltar que, embora λ seja nomeado “parâmetro de segurança”, ele não deve ser tomado da mesma forma que os valores em bits de segurança comuns em outros esquemas criptográficos. No caso do esquema homomórfico em questão, é mais adequado chamar os parâmetros de “níveis de segurança”, sendo eles, por si só, inadequados para comparação com outros esquemas não homomórficos.

4.3 Implementação do esquema FHE

Para a implementação desse sistema criptográfico, foi escolhida a linguagem de programação Python 3.3. Tal escolha se deve, primeiramente, pelo Python ser uma linguagem fácil de programação, com características excelentes para a natureza científica e exploratória do trabalho, assim como ter uma fácil leitura.

Nos primeiros testes usando esquemas SHE, a primitiva de inteiros nativa da linguagem, apesar de possuir a interessante característica de ser implementado como um *long* de tamanho arbitrário, não era compatível com os tempos obtidos nos trabalhos de Coron. Para sanar essa limitação, foi escolhida a biblioteca de teoria numérica GMPY2, que possui um tipo inteiro de precisão múltipla (MPZ). Como as operações com esse tipo numérico são realizadas por código C, a velocidade de execução se aproximou das esperadas. Ainda, essa biblioteca possui funções de geração de números primos que se mostraram úteis a implementação.

4.3.1 Primeiro código – SHE simétrico

A primeira implementação feita foi o esquema de chave simétrica, proposto por (GENTRY e HALEVI, 2011) no início de seu trabalho. O algoritmo utilizado para testes foi o seguinte (Código Fonte 1):

Código Fonte 1: Primeira implementação de um esquema homomórfico

```

1  #!/ python 3.3
2  #imports
3  from gmpy2 import mpz
4  import random
5  import time
6
7  toy    ={'lambda':42, 'rho':16, 'eta':1088, 'gamma':160000, 'beta':12,
   'gamma':144}
8  small  ={'lambda':52, 'rho':24, 'eta':1632, 'gamma':860000, 'beta':23,
   'gamma':533}
9  medium={ 'lambda':62, 'rho':32, 'eta':2176, 'gamma':4200000, 'beta':44,
   'gamma':1972}
10 large  ={'lambda':72, 'rho':39, 'eta':2652, 'gamma':19000000, 'beta':88,
    'gamma':7897}
11
12 def keygen (par):
13     p=random.randint(-(2**par['eta']-1), 2**par['eta']-1)
14     return p
15
16 def encrypt(m, p, par):
17     q=random.randint(0,2**par['gamma']/p-1)
18     r=random.randint(-(2**par['rho'])-1,2**par['rho']-1)
19     c=q*p+2*r+m
20     return c
21
22 def decrypt(c, p):
23     m= (c%p)%2
24     return m
25
26 def test():
27     t=-time.clock()
28     key=keygen(medium)
29     print('gerada chave = ...', str(key)[-20:], sep='')
30
31     c1=encrypt(1, key, medium)
32     print('enc(1) = ', str(c1), sep='')
33     c2=encrypt(0, key, medium)
34     print('enc(0) = ', str(c2), sep='')
35
36     print('dec(1) = ', decrypt(c1, key), sep='')

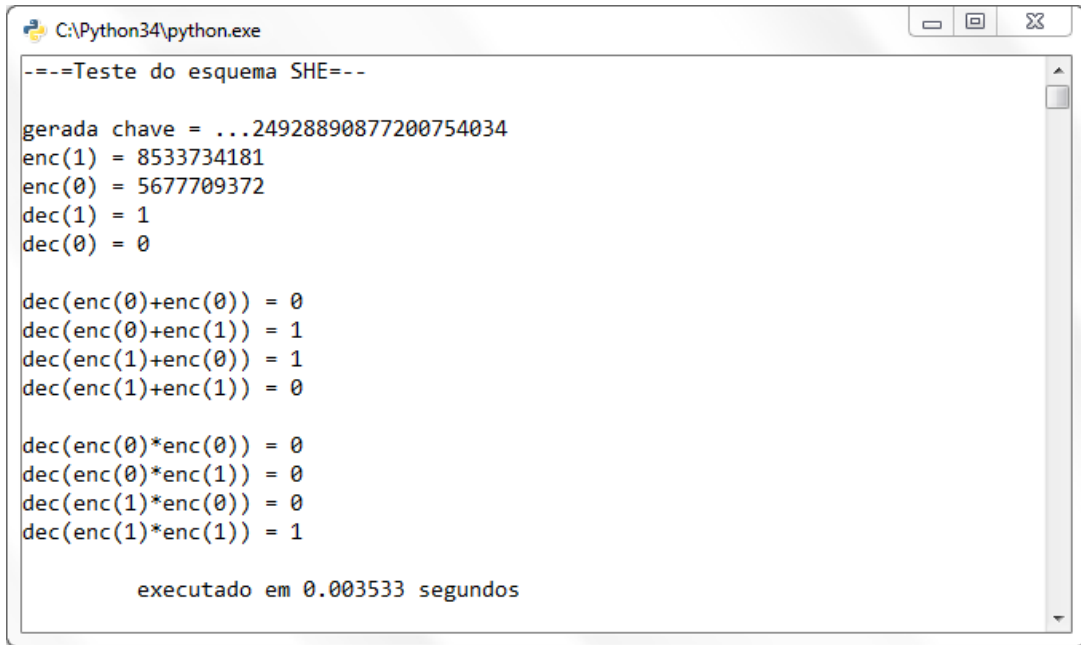
```

```

37     print('dec(0) = ', decrypt(c2, key), sep='')
38
39     print(' ')
40
41     print('dec(enc(0)+enc(0)) =', decrypt(c2+c2, key))
42     print('dec(enc(0)+enc(1)) =', decrypt(c2+c1, key))
43     print('dec(enc(1)+enc(0)) =', decrypt(c1+c2, key))
44     print('dec(enc(1)+enc(1)) =', decrypt(c1+c1, key))
45
46
47     print(' ')
48
49     print('dec(enc(0)*enc(0)) =', decrypt(c2*c2, key))
50     print('dec(enc(0)*enc(1)) =', decrypt(c2*c1, key))
51     print('dec(enc(1)*enc(0)) =', decrypt(c1*c2, key))
52     print('dec(enc(1)*enc(1)) =', decrypt(c1*c1, key))
53
54     t+=time.clock()
55     print("\n\t executado em %f segundos" %(t))
56
57 if __name__ == '__main__':
58     print('---Teste do esquema SHE---\n')
59     test()
60     a=input(' ')

```

Ao ser executado, no interpretador padrão do Python, o resultado obtido é aquele da Figura 3. Note-se que, além de encriptar e decriptar corretamente os bits, a propriedade homomórfica é válida para operações de soma e multiplicação, mapeando respectivamente XOR e AND no espaço do texto plano. A função *keygen* recebe um dicionário com os parâmetros disponibilizados por Coron (CORON et.al, 2011). Destes, utiliza apenas η para retornar um valor no intervalo de $[-2^\eta - 1, 2^\eta]$. Note que a função *randint* tem um intervalo de retorno de $[x, y]$. A função *encrypt* escolhe aleatoriamente um valor q e r , respectivamente nos intervalos de $[0, 2^\gamma/p - 1)$ e $(-2^\rho, 2^\rho)$. O valor encriptado é então retornado como $qp + 2r + m$. A função de *decrypt* retorna o texto plano como $(c \bmod p) \bmod 2$, sendo c o texto cifrado e p a chave. Já a função de teste, por sua vez, como o nome indica, simplesmente executa a geração da chave, encriptação, decriptação e decriptação de *chiphertexts* derivados, e apresenta esses dados.



```
C:\Python34\python.exe

---Teste do esquema SHE---

gerada chave = ...24928890877200754034
enc(1) = 8533734181
enc(0) = 5677709372
dec(1) = 1
dec(0) = 0

dec(enc(0)+enc(0)) = 0
dec(enc(0)+enc(1)) = 1
dec(enc(1)+enc(0)) = 1
dec(enc(1)+enc(1)) = 0

dec(enc(0)*enc(0)) = 0
dec(enc(0)*enc(1)) = 0
dec(enc(1)*enc(0)) = 0
dec(enc(1)*enc(1)) = 1

executado em 0.003533 segundos
```

Figura 3: Resultados do primeiro algoritmo de teste

Fonte: própria

Existe a ressalva de que, apesar de funcionar conforme esperado, esse código possui como objetivo apenas ser didático, não sendo seguro, ou mesmo útil de um ponto de vista estritamente prático.

4.3.2 Relação entre as primitivas

A função *Keygen* utiliza um objeto da classe *Parâmetro* para gerar o par de chaves criptográficas. A chave pública é utilizada pelas funções de encriptação e expansão para gerar, respectivamente o texto cifrado e a matriz associada *Z*. A função de deciptação utiliza o texto cifrado, a matriz associada *Z* e a chave privada para recuperar o bit original. Por fim, as funções homomórficas *add* e *mult* trabalham sobre os Textos cifrados, e, produzem novas cifras. A Figura 4 ilustra de forma simplificada essas relações.

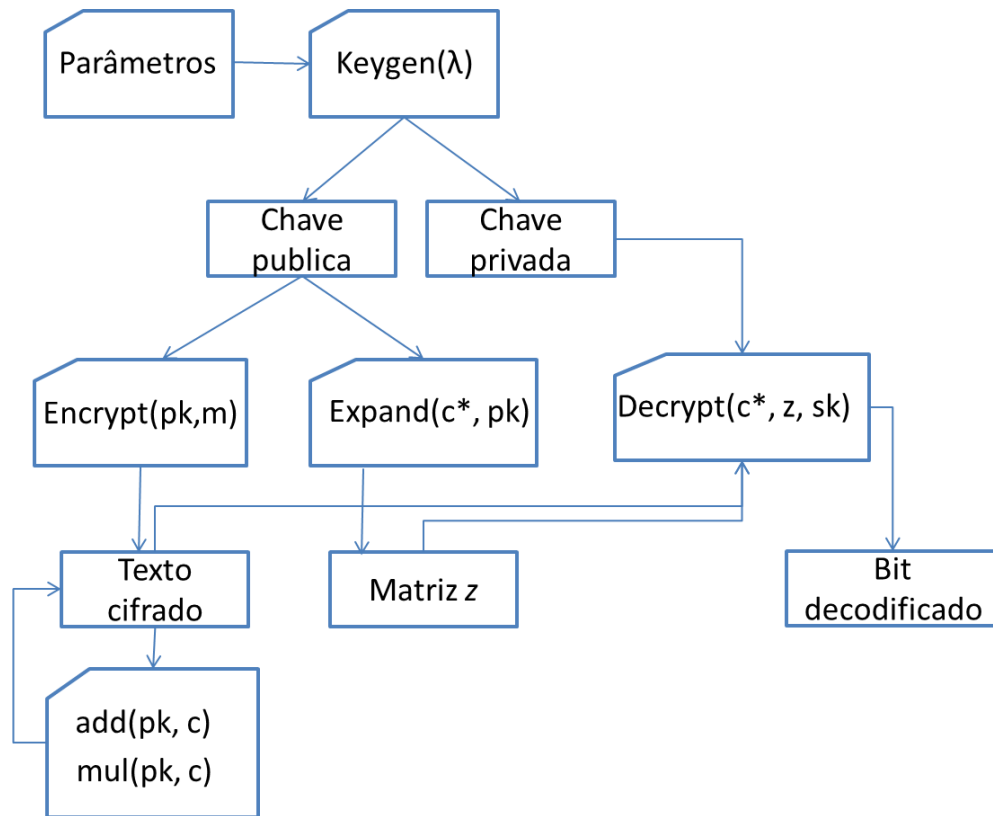


Figura 4: Relação entre as primitivas do esquema.

Fonte: própria

4.3.3 Geração das chaves

Primeiramente, deve-se atentar ao fato de que o operador de resto de divisão da linguagem Python retorna um valor inteiro dentro do intervalo $[0, n)$, enquanto, nos trabalhos relacionados a criptografia homomórfica utilizados na produção deste, a redução de um valor módulo n resulta em um inteiro no intervalo $(-n/2, n/2]$. Além disso, a divisão inteira de z por p é definida por $q_z(p) = \lfloor z/p \rfloor$. Com isso, a operação de redução de módulo no intervalo desejado é definida como $r_p(z) = z - q_p(z) \cdot p \mid r_p(z) \in (-p/2, p/2]$.

Para tal, foi utilizado o Código Fonte 2: Módulo. Note que sendo Python uma linguagem de tipagem dinâmica, foi utilizado na função *qNear* o operador de divisão inteira, representado por `//`.

Código Fonte 2: Módulo

```

1 def qNear(a,b):
2     '''retorna quociente de a por b arredondado para o inteiro mais
   proximo'''
3     return (2*a+b)//(2*b)
4

```

```

5  def modNear(a,b):
6      '''retorna a mod b no intervalo de [-(b/2), b/2)'''
7      return a-b*qNear(a,b)

```

Para a geração do par de chaves criptográficas, o código a seguir foi utilizado (Código Fonte 3: *Keygen*):

Código Fonte 3: *Keygen*

```

1  def keygen(file, size='toy'):
2      P.setPar(size)
3      tempo=-time.time()
4
5      p = genZi(P._eta)
6      #print("p computado")
7      q0, x0 = genX0(p, P._gamma, P._lambda)
8      #print("q0,x0 computado")
9      listaX = genX(P._beta, P._rho, p, q0)
10     #print("listax computado")
11     pkAsk = listaX
12     pkAsk.insert(0, x0)
13     print("pk* computado")
14     while True:
15         s0,s1=genSk(P._theta, P._thetam)
16         if (s0.count(1)*s1.count(1)==15): break
17
18     se=int(time.time()*1000) #seed para RNG
19     _kappa=P._gamma+6 #variavel para o RNG, conforme pg9 coron
20     u11=genU11(se, s0, s1, P._theta, _kappa, p)
21     sigma0 = encryptVector(s0, p, q0, x0, P._rho)
22     signal = encryptVector(s1, p, q0, x0, P._rho)
23     tempo+=time.time()
24     #pickle files
25
26     public=fheKey.pk(pkAsk, se, u11, sigma0, signal, P)
27     secret=fheKey.sk(s0, s1, P)
28     fheKey.write(public, 'pk_pickle_'+file)
29     fheKey.write(secret, 'sk_pickle_'+file)
30
31     #write key to file
32     f = open(file, 'w')
33     f.write(("keygen executado em " +str(tempo) +" segundos"+"'\n\n'))
34     f.write(('p==' + str(p)+''\n\n'))
35     f.write(('q0==' +str(q0)+''\n\n'))
36     f.write(('pk*==' +str(pkAsk)+''\n\n'))
37     f.write(('s0 ==' +str(s0)+''\n\n'))

```

```

38     f.write(('s1 =='+str(s1)+'\n\n'))
39     f.write(('se =='+str(se)+'\n\n'))
40     f.write(('u11 =='+str(u11)+'\n\n'))
41     f.write(('sigma0 =='+str(sigma0)+'\n\n'))
42     f.write(('sigma1 =='+str(sigma1)+'\n\n'))
43     f.close
44     print('arquivo salvo', file)

```

Inicialmente, é computado o valor de p , a partir de η (linha 5). A função `genZi` (Código Fonte 4: Geração de inteiros aleatórios ímpares retorna um número inteiro ímpar de η bits, utilizando como semente o tempo atual em milissegundos e funções da biblioteca `GMPY2`.

Utilizar o tempo atual como semente para o gerador de números pseudoaleatórios pode não gerar uma quantidade adequada de entropia para aplicações criptográficas. Por exemplo, `gpg4win`, uma implementação para Windows do *Gnu Privacy Guard* e o software *TrueCrypt*, utilizam o tráfego de rede, e, dados como coordenadas de mouse e ativação do teclado para gerar aleatoriedade. Porém, como esse trabalho não tem por objetivo criar uma implementação final para o mercado, mas sim algo didático, e, como a criptografia totalmente homomórfica *per se* ainda não é adequada para aplicações práticas, o uso do tempo será adotado pela simplicidade de implantação.

Código Fonte 4: Geração de inteiros aleatórios ímpares

```

1  def genZi(eta):
2      '''retorna um inteiro aleatório ímpar de eta bits''' ##
3      eta=mpz(eta)
4      seed=int(time.time()*1000000) #time em microseconds
5      state=gmpy2.random_state(seed)
6      while True:
7          p=gmpy2.mpz_rrandomb(state, eta)
8          if gmpy2.is_odd(p): return p

```

Depois, os valores de q_0 e x_0 são gerados. A variável q_0 é escolhida como o produto de dois números primos aleatórios de λ^2 bits, dentro do intervalo $(0, 2^\gamma/p]$, e, $x_0 = q_0 \cdot p$. Para tal, o Código Fonte 5: Geração de primos é utilizado para gerar números provavelmente primos de b bits, e, o código fonte Código Fonte 6: Geração do elemento X_0 para calcular os valores de q_0 e x_0 .

Código Fonte 5: Geração de primos

```

1  def genPrime(b):
2      '''função retorna um primo MPZ de b bits''' ##
3      while True:
4          seed=int(time.time()*1000000) #time em microseconds
5          state=gmpy2.random_state(seed)
6          prime=gmpy2.mpz_rrandomb(state,b)
7          prime=gmpy2.next_prime(prime)
8          if len(prime)==b:
9              printDebug('gerado número primo de ', b, 'b')
10             return prime

```

Código Fonte 6: Geração do elemento X_0

```

1  def genX0(p, _gamma, _lambda):
2      '''gera o elemento q0 no intervalo [0, (2**gamma)/p) como sendo o
3      produto de
4      dois primos de lambda**2 bits e retorna x0 como sendo q0*p
5      o retorno da função é uma tupla (q0, x0)
6      '''
7      teto=gmpy2.f_div(2**mpz(_gamma), p) #divisão com retorno de
8      inteiro
9      while True:
10         q0=genPrime(_lambda**2)*genPrime(_lambda**2)
11         if q0 < teto: break
12         x0=gmpy2.mul(p, q0)
13         printDebug('genX0 executado, com parametro x0==', len(x0),
14         'e q0==', len(q0), 'bits')
15         printDebug('q0==', q0, '\nX0==', x0)
16
17     return q0, x0

```

O próximo passo é a geração de β pares de elementos x_i que formação a chave pública parcial pk_{Ask} . De forma a facilitar a leitura do código, as duas funções mostradas no código fonte Código Fonte 7 encapsulam a geração dos elementos aleatórios q e r , com $q \in (0, q_0)$ e $r \in (-2^\rho, 2^\rho)$

Código Fonte 7: Geração de q e r

```

1  def grand(q0):
2      '''retorna um número aleatorio no intervalo 0, q0'''
3      return random.randint(0, q0)
4
5  def rrand(rho):
6      '''retorna um número aleatorio entre -(2^rho) e 2^rho'''

```

```

7     limit=2**rho
8     return random.randint(-limit, limit)

```

O Código Fonte 8 gera uma lista de pares x_i que compõem a chave privada parcial, seguindo a seguinte formula:

$$x_{i,b} = p \cdot q_{i,b} + r_{i,b} \mid 1 \leq i \leq \beta, 0 \leq b \leq 1$$

Essa lista, então, é concatenada com o valor x_0 anteriormente calculado, e, armazenada como a variável $pkAsk$ (linha 12 do *keygen*)

Código Fonte 8: Lista de elementos X_i

```

1  def genX(beta, rho, p, q0):
2      '''Gera uma lista de beta pares x[i,0], x[i,1] seguindo a formula
3          x[i,b]=p.q[i,b]+r[i,b] | 1<i<beta, 0<b<1
4          onde q[i,b] são aleatorios no intervalo [0, q0] e r[i,b] são
5          inteiros aleatorios no intervalo (-2**p, 2**p)
6      '''
7      x=list()
8      for i in range(beta*2):
9          result = gmpy2.mul(p, qrand(q0))
10         result = gmpy2.add(result, rrand(rho))
11
12         x.append(result)
13
14     printDebug('gerada lista de elementos xi com', len(x), 'elementos
15     (beta=', beta, ')')
16
17     return x

```

A próxima etapa na geração das chaves é gerar a chave privada. O código fonte Código Fonte 9: Vetores s_b gera dois vetores de números binários s_b . Os vetores s_b são compostos por bits aleatórios, mas, algumas restrições são aplicadas em sua construção:

- O primeiro elemento de ambos os vetores deve ser 1.
- Para cada $k \in [0, \sqrt{\theta})$ e $b = \{0,1\}$, existe no máximo um bit *set* nos vetores $s_i^{(b)}$, com $k\lfloor\sqrt{B}\rfloor < i \leq (k+1)\lfloor\sqrt{B}\rfloor$ e com $B = \theta/\theta$.
- O conjunto $S = \{(i,j): s_i^{(0)} \cdot s_j^{(1)} = 1\}$, contém exatamente θ elementos.

Convém relembrar que, para os quatro diferentes conjuntos de parâmetros adotados, $\theta = 15$.

Código Fonte 9: Vetores s_b

```

1  def genSk(theta, thetam):
2      l=math.ceil(math.sqrt(theta)) #comprimento do vetor s
3      s0 = [1]+[0]*(l-1)
4      s1 = [1]+[0]*(l-1)
5      k=range(0, 4)
6      ks0=random.sample(k, 2)
7      ks1=random.sample(k, 4)
8      B=math.floor(math.sqrt(theta/thetam))
9      for k in ks0:
10         idx=random.randint((k*B)+1, (k+1)*B)
11         s0[idx-1]=1
12     for k in ks1:
13         idx=random.randint((k*B)+1, (k+1)*B)
14         s1[idx-1]=1
15     return s0, s1

```

Para a geração do elemento u_{11} , é necessário a consulta de elementos da matriz U . Tal matriz é muito grande para ser mantida em memória, por tanto, uma semente aleatória *se* é adicionada como parâmetro da chave pública, o que permite que os elementos da matriz U sejam calculados *on the fly* para a geração de u_{11} . O código fonte Código Fonte 10 recebe a posição do elemento na matriz U , a semente aleatória e os parâmetros da chave e retorna o elemento u correspondente.

Código Fonte 10: Matrix aleatória u

```

1  def randomMatrix(i, j, kappa, se, sqrtTheta):
2      '''gera a matrix para u11 on the fly'''
3      if i==0 and j==0: return 0
4      random.seed(se)
5      itera = i*sqrtTheta+j
6      for i in range(itera):
7          a=random.getrandbits(kappa+1)
8      return a

```

Com a matriz U sendo gerado em tempo de execução pelo código fonte Código Fonte 10, o código fonte 11 retorna o valor do elemento u_{11} como sendo o resultado da seguinte formula:

$$\sum_{(i,j) \in S} u_{i,j} = x_p \bmod 2^{\kappa+1}$$

Onde $x_p \leftarrow \lfloor 2^k/p \rfloor$.

Código Fonte 11: Calculo de u_{11}

```

1  def genU11(se, s0, s1, theta, kappa, p):
2      '''função para gerar o elemento u(1,1)'''
3      #gerar indices de elementos 1 dos vetores s0 e s1
4      si, sj = list(), list()
5      for i in range(len(s0)):
6          if s0[i]==1: si.append(i)
7      for j in range(len(s1)):
8          if s1[j]==1: sj.append(j)
9      #gerar matriz de raiz de theta elementos
10     l=math.ceil(math.sqrt(theta))
11     mx= 2**mpz(kappa+1)
12     #computa somatorio
13     xp=gmpr2.div(2**kappa,p)
14     xp=mpz(gmpr2.round_away(xp))
15
16     soma=modNear(xp, mx)
17     somatorio=0
18     for i in si:
19         for j in sj:
20             somatorio += randomMatrix(i, j, kappa, se, l)
21     u=soma-somatorio
22     return u

```

O próximo passo na geração de chaves é a geração de vetores criptografados dos elementos da chave pública. Esse procedimento é necessário para que, utilizando os bits da chave privada criptografados, o esquema criptográfico possa executar o próprio circuito de decriptação, resultando na primitiva de *Recrypt*, necessária para o esquema ser totalmente homomórfico. Para gerar as listas “sigma0” e “sigma1”, a função *encryptVector* no código fonte

Escolhendo para cada $i \in [1, \lceil \sqrt{\theta} \rceil]$ e $b = \{0,1\}$, inteiros aleatórios $r'_{i,b} \in (-2^\rho, 2^\rho)$ e $q'_{i,b} \in [0, q_0)$,

Código Fonte 12: Encriptação da chave pública é utilizada. Para criptografar um bit s , a seguinte fórmula é utilizada:

$$\sigma_i^{(b)} = s_i^{(b)} + 2r'_{i,b} + p \cdot q'_{i,b} \bmod x_0$$

Escolhendo para cada $i \in [1, \lceil \sqrt{\Theta} \rceil]$ e $b = \{0,1\}$, inteiros aleatórios $r'_{i,b} \in (-2^\rho, 2^\rho)$ e $q'_{i,b} \in [0, q_0)$,

Código Fonte 12: Encriptação da chave pública

```

1  def encryptVector(s, p, q0, x0, rho):
2      sigma=list()
3      p2=2**rho
4      for i in s:
5          r = random.randint(-p2, p2)
6          q = random.randint(0, q0-1)
7          c = modNear((i+(2*r)+(p*q)),x0)
8          sigma.append(c)
9      return sigma

```

Com isso, a chave pública e privada estão calculadas, sendo a chave pública as variáveis `pkAsk`, `se`, `u11`, `sigma0`, `sigma1`. A chave privada é composta pelas listas `s0` e `s1`. Em ambos foi adicionado um objeto da classe parâmetros (`P`) para melhor controle. O resto do código apenas é responsável pela saída para monitor e arquivos no HD, em ASCII para debug,

4.3.4 Classes e Pickle

Durante as fases iniciais de implementação do código, os números gerados eram salvos no HD local usando arquivos de texto puro. Apesar de interessante de um ponto de debug de código, não é a melhor alternativa. Python possui uma biblioteca destinada a serialização e permanência de dados chamada *Pickle*. Para facilitar a escrita e leitura das chaves criptográficas, as classes *pk* e *sk* foram criadas, e, métodos *read* e *write* escritos em volta do *pickle*. O código fonte Código Fonte 13 mostra a implementação dessas classes e métodos, que são utilizados nas outras primitivas. A Figura 5 mostra o diagrama dessas classes

Código Fonte 13: Classes e métodos de escrita

```

1  import pickle
2  '''
3  biblioteca que implementa as classes para a chave pública e privada, além
4  das funções necessarias para escrever e ler esses objetos, usando pickle
5  '''

```

```

6  class pk():
7      '''classe para armazenar a chave privada'''
8      def __init__(self, pkAsk, se, u11, sigma0, sigma1, P):
9          self.pkAsk=pkAsk
10         self.se=se
11         self.u11=u11
12         self.sigma0=sigma0
13         self.sigma1=sigma1
14         self.P=P
15
16  class sk():
17      '''classe para armazenar a chave pública'''
18      def __init__(self, sz, su, P):
19          self.s0=sz
20          self.s1=su
21          self.P=P
22
23  def write(obj, name):
24      '''faz uma coserva de obj no arquivo name'''
25      with open(name, 'wb') as arq:
26          pickle.dump(obj, arq)
27
28  def read(name):
29      '''abre o pote name e retorna o pickles'''
30      with open(name, 'rb') as arq:
31          return pickle.load(arq)

```

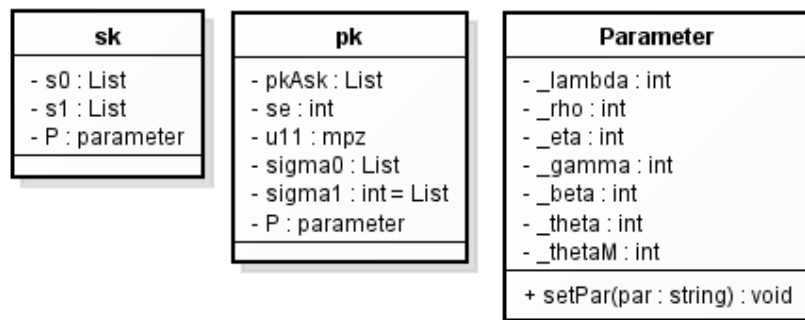


Figura 5: Diagrama das classes *sk*, *pk* e *Parameter*

4.3.5 Encriptação

O código fonte Código Fonte 14 descreve a primitiva de encriptação. Inicialmente, o escopo da mensagem m é validado, e, então, criada uma matriz $\beta \times \beta$ e preenchida com elementos aleatórios no intervalo $[0, 2^\alpha)$. O parâmetro $\rho' \leftarrow 2^\rho$ e o inteiro aleatório r está no

intervalo $(-2^{\rho'}, 2^{\rho'})$. A função então calcula o somatório como sendo:

$$\sum_{1 \leq i, j \leq \beta} b_{ij} \cdot x_{i,0} \cdot x_{j,1}$$

Por fim, o *cyphertext* é calculado como:

$$c^* = m + 2r + 2(\text{somatório}) \bmod x_0$$

Código Fonte 14: *Encrypt*

```

1  def encrypt(pk, m):
2      if m != 0 and m != 1:
3          print('not a valid m')
4          return 0
5
6      alpha = pk.P._lambda  ##o fator de segurança alpha é igual a lambda
7      matrix = [[0 for i in range(pk.P._beta)] for j in range(pk.P._beta)]
8      for i in range(pk.P._beta):
9          for j in range(pk.P._beta):
10             matrix[i][j]=random.randint(0, 2**alpha -1)
11             #print(matrix[i][j], end=' ')
12             #print(' ')
13     pprime=2**pk.P._rho
14     r=random.randint(-pprime, pprime)
15     pkAsk=(pk.pkAsk).copy()
16     x0=pkAsk.pop(0)
17     xi0=pkAsk[:2]
18     xj1=pkAsk[1::2]
19
20     ## iniciando processo de encrypt
21     ## computar somatorio
22     somatorio=mpz(0)
23     for i in range(pk.P._beta):
24         for j in range(pk.P._beta):
25             x=gmpy2.mul(xj1[j], xi0[i])
26             somatorio += gmpy2.mul(matrix[i][j], x)
27     c=(m+(2*r)+(2*somatorio))
28     c=modNear(c,x0)
29     return c

```

4.3.6 Expansão

O Código Fonte 15 é o algoritmo de responsável pelo procedimento de expansão.

Para tal, para cada $1 < i, j < \sqrt{\theta}$, primeiramente compute o número inteiro aleatório $u_{i,j}$ usando o gerador de números pseudoaleatórios $f(se)$, então calcule $y_{i,j} = u_{i,j}/2^\kappa$ e então compute $z_{i,j}$:

$$z_{i,j} = [c^* \cdot y_{i,j}]_2$$

Mantendo apenas $n = \lceil \log_2(\theta + 1) \rceil$ bits de precisão após o ponto binário. Defina a matriz $z = (z_{i,j})$.

Código Fonte 15: *Expand*

```

1  def expand(pk, cAsk):
2      #cria uma matriz de sqrt(theta) elementos
3      l=int(math.sqrt(pk.P._theta))
4      y=[[0 for i in range(l)] for i in range(l)]
5      kappa=pk.P._gamma+6
6
7      n=4 ##coron 441, pg10
8      gmpy2.get_context().precision=n #seta precisao do mpfr para 2
9      #computa matriz y[i][j]
10     for i in range(l):
11         for j in range(l):
12             y[i][j]=float(gmpy2.mpfr(randomMatrix(i, j, kappa, pk.se,
13 l)/(2**kappa)))
14             y[0][0]=float(gmpy2.mpfr(pk.u11/2**kappa))
15
16     #seta precisão do mpfr de novo para o valor correto
17     gmpy2.get_context().precision=1024 #valor feito com testes.
18     #gera matrix z e computa elementos
19     expand = [[1 for i in range(l)] for i in range(l)]
20     for i in range(l):
21         for j in range(l):
22             expand[i][j]=float((cAsk * y[i][j])%2)
23     return expand

```

4.3.7 Decriptação

A função de decriptação, código fonte Código Fonte 16, recebe como parâmetro a

chave privada, a mensagem cifrada e a matriz associada z .

Primeiramente é calculada a soma dos resultados de multiplicações entre a chave privada e a matriz associada, seguindo a seguinte formula:

$$soma = \sum_{i,j} s_i^{(0)} \cdot s_j^{(1)} \cdot z_{ij}$$

Então, a mensagem original m é recuperada como:

$$m \leftarrow [c^* - \lfloor soma \rfloor]_2$$

Código Fonte 16: Decrypt

```

1  def decrypt(sk, cAsk, z):
2      soma=0
3      l=int(math.sqrt(sk.P._theta))
4      for i in range(l):
5          for j in range(l):
6              soma+=(sk.s0[i]*sk.s1[j]*z[i][j])
7      soma=mpz(gmpy2.round_away(soma))
8      m=(cAsk-soma)%2
9      return m

```

4.3.8 Primitivas AND e XOR

Uma das características importantes de um sistema criptográfico homomórfico é a capacidade de se executar processamento sobre os textos cifrados. Nos esquemas totalmente homomórficos, as portas lógicas AND e XOR são mapeadas, respectivamente, pela multiplicação e pela soma dos textos cifrados, seguido de uma redução modular pelo elemento x_0 da chave pública parcial pk^* . No código fonte Código Fonte 17 temos as implementação dessas funções.

Código Fonte 17: Funções lógicas

```

1  def add(pk, c1, c2):
2      soma=gmpy2.add(c1,c2)
3      return modNear(soma,pk.pkAsk[0])
4
5  def mul(pk, c1, c2):

```

```

6     mul= gmpy2.mul(c1,c2)
7     return modNear(mul,pk.pkAsk[0])

```

4.3.9 Recrypt

A primitiva de *Recrypt*, por fins práticos, não foi implementada, por ser ela uma representação homomórfica do circuito de *decrypt*, executado sobre os bits do texto cifrado e sobre encriptações da chave privada.

4.4 Melhorando o Desempenho

Como forma de se melhorar o desempenho de execução do código, será apresentado algumas propostas de modificação do código. São elas o uso de *threads* e de um compilador *Just-in-time* com otimização automática do código.

4.4.1 Tecnologias para melhoria de desempenho

A biblioteca Numba, para Python dispõe de uma solução que facilita a criação de código com melhor desempenho do que Python puro. Tal tecnologia envolve o uso de um decorador de função, e, o Numba gera uma representação da função em LLVM, que, em teoria, é de execução mais rápida do que código Python puro. Para analisar, de forma simples, a relação entre código Python e código Python com Numba, o Código Fonte 18: Comparação Numba e Python foi utilizado para gerar o gráfico da figura Figura 6 e Figura 7: Relação de tempo de execução de código Numba/Python. O código foi executado na IDE Spyder, com Numba Pro versão *Student*, em um computador com Windows 7 x64, CPU core i5@2.6GHz, e 4GB de memória RAM.

Código Fonte 18: Comparação Numba e Python

```

1  # -*- coding: utf-8 -*-
2  import random
3  from numbapro import *
4  from time import clock
5
6  def Mult(X,Y,r): #Multiplicação de matrizes
7      r=[[0 for i in range(len(X))] for i in range(len(X))]
8      for i in range(len(X)):
9          for j in range(len(Y[0])):

```

```

10         for k in range(len(Y)):
11             r[i][j] += X[i][k] * Y[k][j]
12
13     @autojit
14     def MultJ(X,Y,r):
15         for i in range(len(X)):
16             for j in range(len(Y[0])):
17                 for k in range(len(Y)):
18                     r[i][j] += X[i][k] * Y[k][j]
19
20     def matrix(l):
21         X = [[random.randint(0,1000) for i in range(l)] for i in range(l)]
22         Y = [[random.randint(0,1000) for i in range(l)] for i in range(l)]
23         r=[[0 for i in range(l)] for i in range(l)]
24
25         t1=-clock()
26         R=Mult(X,Y,r)
27         t1+=clock()
28
29         t2=-clock()
30         Rj=MultJ(X,Y,r)
31         t2+=clock()
32
33         print("Matrix %d x %d: Python= %f s ; Numba= %f s" %(l,l, t1, t2))
34
35     if __name__=='__main__':
36         mtxsize=[10,20,30,40,50,60,70,80,90,100,150,200,250,300,350,400,450,500]
37         for l in mtxsize:
38             matrix(l)

```

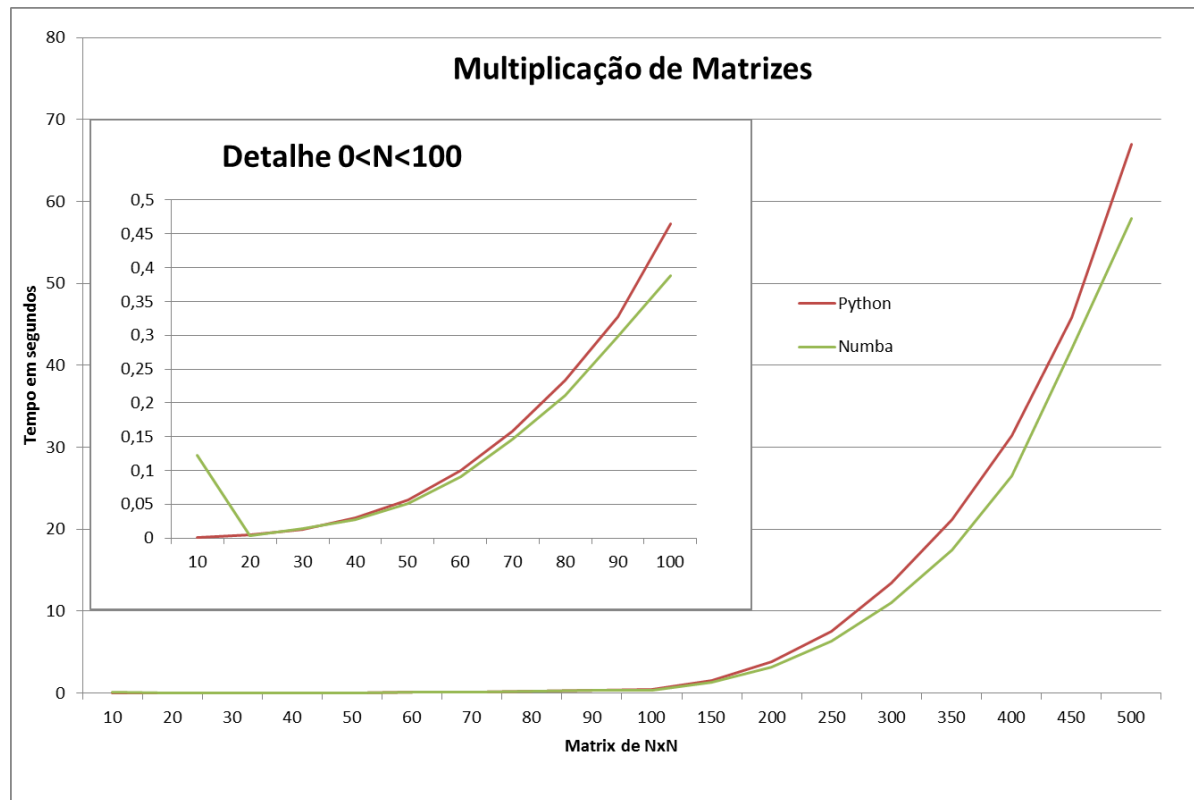


Figura 6: Tempo de multiplicação de Matrizes de números aleatórios

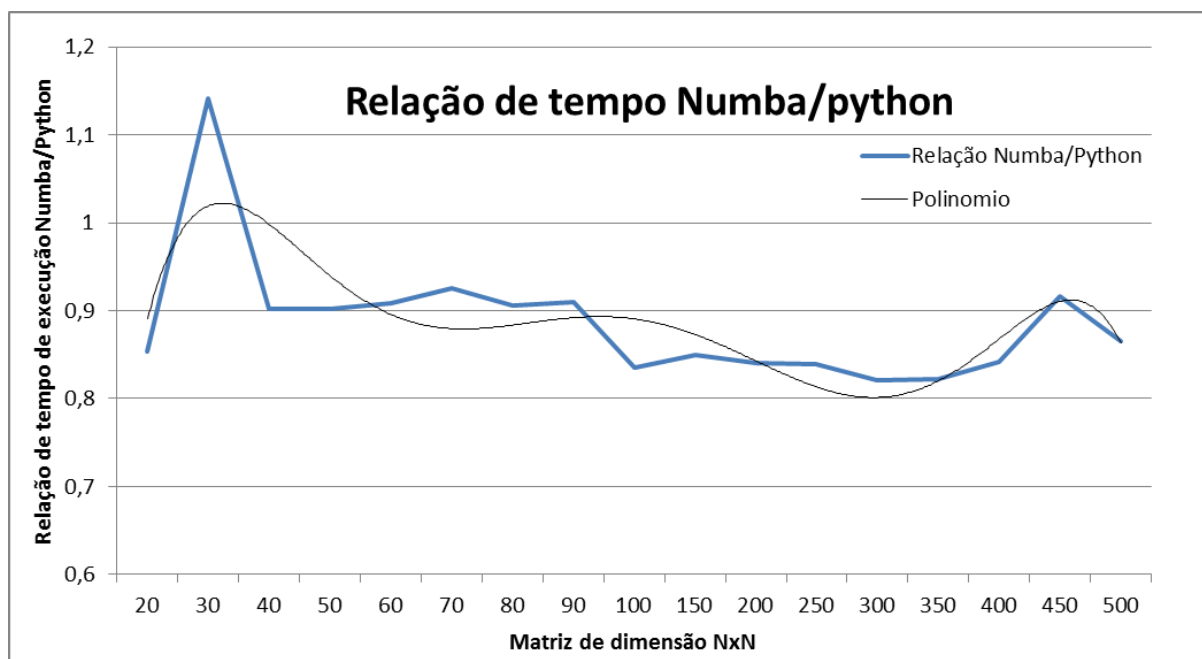


Figura 7: Relação de tempo de execução de código Numba/Python

Pelos resultados obtidos, a execução do código de multiplicação de matrizes toma 88% do tempo necessário com relação a execução do código puramente em Python. De fato, a

única diferença entre as funções de multiplicação é o decorador adicionado a função, tornando o uso do Numba extremamente fácil em primeira análise.

Entretanto, várias limitações são impostas ao uso dessa tecnologia. Em testes, foi visto que o uso de *list comprehensions*, ou seja declarações de laços dentro de listas, não funciona com o decorador autoJit, assim como alocação dinâmica de variáveis ou troca de tipos de variáveis. Tais limitações necessitam que, dependendo da implementação original, grandes porções de código sejam modificadas para se adequar ao JIT. Além é claro da limitação no uso de alguns estilos de programação próprios do Python, como utilizar listas com alocação dinâmica de memória.

Outra forma possível de melhorar o desempenho do código é por meio do uso de múltiplas *threads* para a execução. O Python possui uma função de mapeamento de funções e entradas chamada *map()* implementada de forma *multithread* na biblioteca *multiprocessing.dummy*. Para se executar um pequeno teste do comportamento de *multithread* no Python, o código fonte Código Fonte 19: Teste de *Multithreading* foi usado, e, os resultados obtidos mostrados na Tabela 3, com os fatores de tempo em relação à execução sem threads concorrentes são mostrados no gráfico da Figura 8.

Código Fonte 19: Teste de *Multithreading*

```

1  from time import clock
2  import random
3  from multiprocessing import Pool
4  from multiprocessing.dummy import Pool as ThreadPool
5
6  def doMath(x):
7      r=((x**7)%77)**3/x
8      return r
9
10 if __name__=='__main__':
11
12     for tlista in [10,100,1000,10000,100000,1000000,5000000]:
13         l=[random.randint(1,1000000) for i in range (tlista)]
14         t1=-clock()
15         r=[]
16         for i in l:
17             r.append(doMath(i))
18         t1+=clock()
19
20         t2=-clock()
21         pool=ThreadPool(4) #Cria N threads, onde N é o número de cores

```

```

22     r2=pool.map(doMath, l) #
23     t2+=clock()
24
25     t3=-clock()
26     pool=ThreadPool(8) #Cria N threads, onde N é o número de cores
27     r3=pool.map(doMath, l) #
28     t3+=clock()
29
30     t4=-clock()
31     pool=ThreadPool(16) #Cria N threads, onde N é o número de cores
32     r4=pool.map(doMath, l) #
33     t4+=clock()
34     print('Len=(%d) Tempo normal=%f ; %f (4) | %f(8) | %f (16)'
35           %(tlista, t1, t2, t3, t4))

```

Tabela 3: Tempos de execução Multithread (em segundos)

len(lista)	N de threads			
	1	4	8	16
10	0,000029	0,006711	0,004196	0,004353
100	0,000248	0,002982	0,003543	0,006894
1000	0,00223	0,004796	0,006732	0,008902
10000	0,022507	0,022182	0,028914	0,033569
100000	0,235911	0,203993	0,208345	0,220024
1000000	2,163537	2,017519	1,983868	2,008886
5000000	10,62872	10,00738	9,68792	10,1853
Tempo de execução em segundos				

O uso de múltiplos threads não é uma bala mágica que resolve todo e qualquer problema. No teste executado, vê-se que só existe ganho de desempenho em listas com tamanho superior à 10^6 elementos. Apesar de ter havido um esforço para que a simulação fosse parecida com aquilo que é visto no homomorfismo, testes isolando partes do código e executando em várias threads devem ser feitos para se determinar se existe uma aceleração do código.

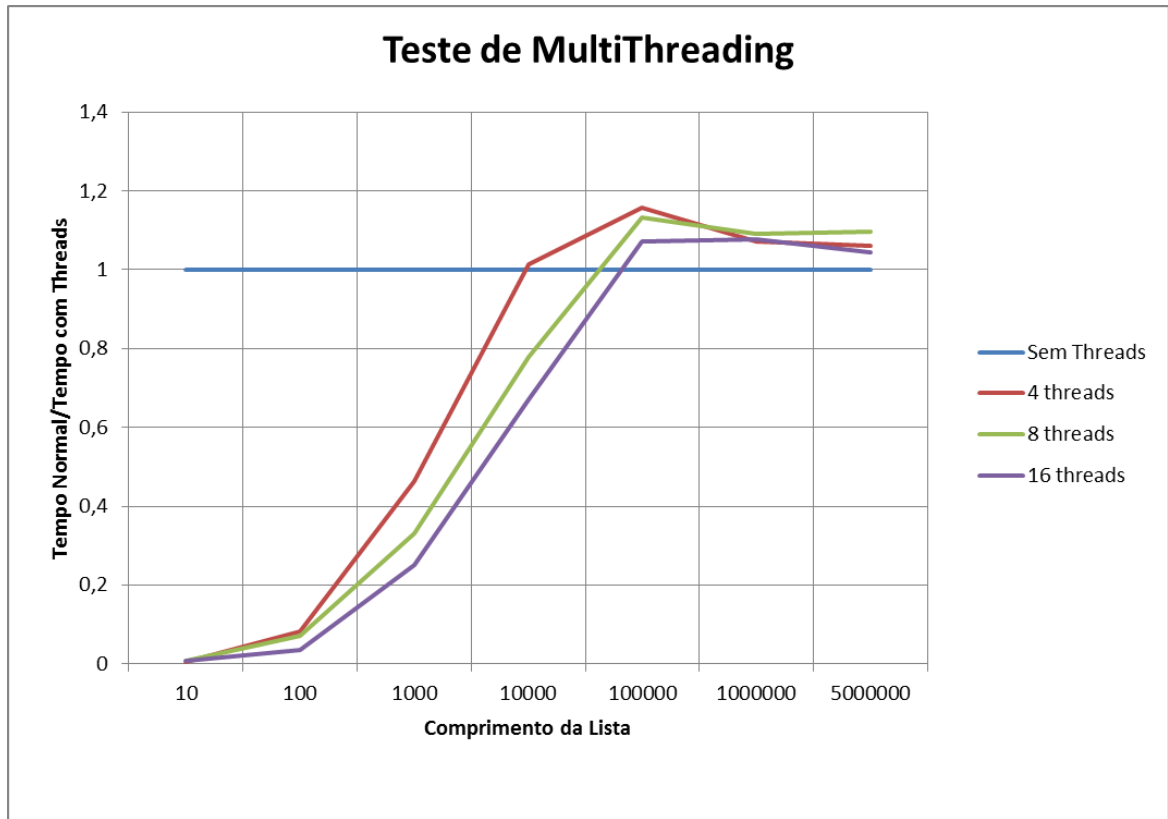


Figura 8: Fatores de tempo do teste de *Multithreading*

4.4.2 Proposta de melhoria de desempenho

Alguns trechos de código do esquema homomórfico, por possuírem características específicas, se mostram propícios para futuras implementações de paralelismo. Na primitiva de *Keygen*, são eles:

- A geração de números primos de λ^2 -bits, já que, devido ao tamanho deles, o teste de primariedade ocupa um bom tempo de CPU
- A geração dos elementos $\sigma_i^{(b)}$, já que o processamento de cada elemento do vetor é independente dos outros elementos, sendo assim possível que sejam executados em paralelo.

Na primitiva de *Encrypt*, os pontos promissores são:

- A geração da matriz b
- O calculo de soma $\sum_{1 \leq i, j \leq \beta} b_{ij} \cdot x_{i,0} \cdot x_{j,1}$, que pode ter as multiplicações feitas em diferentes threads e então somadas ao finalizar a *pool*.

Similarmente, a primitiva de *Decrypt* tem como um ponto promissor para

paralelização a soma $\sum_{i,j} s_i^{(0)} \cdot s_j^{(1)} \cdot z_{ij}$.

Com relação às outras primitivas, não foram identificados pontos de interesse para paralelização do código, sendo, portanto, necessários testes com diferentes parâmetros, com o objetivo de se determinar se é possível haver um ganho de desempenho nessas primitivas.

5 RESULTADOS

5.1 Trabalhos correlatos

Dentre os trabalhos correlatos nessa área de pesquisa, relacionados a este, pode-se citar, principalmente, os trabalhos publicados pelo Dr. Jean-Sebasti  n Coron (CORON et.al, 2011) e pelo Dr. Craig Gentry. Gentry foi o respons  vel pela publica  o do primeiro esquema de criptografia totalmente homom  rfico, com base em reticulados (GENTRY e HALEVI, 2011) e trabalhou na publica  o do esquema com base em n  meros inteiros (DIJIK et.al, 2010) que foi o utilizado nesse trabalho.

Com rela  o    acelera  o do esquema homom  rfico, Wang, Lianmu e Xinming [4] fizeram uma implementa  o do FHE com reticulados de Gentry, e, aceleram a execu  o por meio do uso de GPUs e t  cnicas matem  ticas como a transformada r  pida de Fourier, obtendo resultados promissores, mostrados na Tabela 4. Os tempos obtidos foram obtidos em uma GPGPU NVIDIA Tesla C2050 com 448 CUDA Cores.

Tabela 4: Implementa  o de Wang et.al.

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

Fonte: Wei Wang, Yin Hu, Lianmu Chen, Xinming Huang e Berk Sunar, 2012

Os trabalhos correlatos utilizados na escrita desse trabalho foram aqueles da classe de FHE sobre inteiros, conforme ilustrado na Figura 9. O esquema de Gentry-Halevi foi utilizado nas etapas iniciais da pesquisa, sendo posteriormente favorecido o esquema DGHV e as suas variantes.

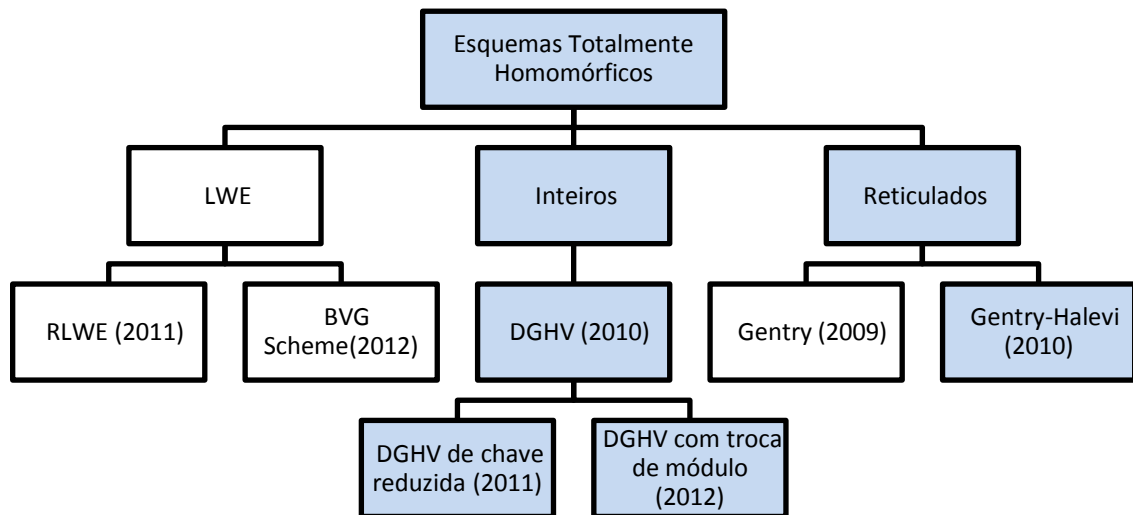


Figura 9: Trabalhos relacionados

Fonte própria

5.2 Metodologia dos testes

Todos os testes realizados nesse trabalho foram executados em um notebook com processador core i5 M520 @2.4 GHz, 4GB de memória RAM e sistema operacional Windows 7 x64, exceto onde indicado diferente.

Para o registro de tempo, foi utilizado a função *clock()*, da biblioteca padrão *time* do Python, e, calculada a média de quinze execuções, como forma de se obter uma leitura correta do tempo de execução.

5.3 Resultados

Em testes, foram obtidos os tempos de execução de acordo com a Tabela 5. Note que, para os parâmetros *Medium* e *Large*, não foi possível a medida de tempo devido ao estouro de memória disponível na máquina de testes.

Tabela 5: Tempos de execução

Parâmetro	<i>KeyGen</i>	<i>Encrypt</i>	<i>Decrypt</i>	<i>Expand</i>
<i>Toy</i>	0.6 s	0.00236 s	0.0001 s	1.28103 s
<i>Small</i>	10 s	0.01294 s	0.0002 s	8.08505 s
<i>Medium</i>	1 min 1 s	*	*	*
<i>Large</i>	11 min 21s	*	*	*

*estouro de memória

A comparação com resultados obtidos por Coron está ilustrada na Figura 10.

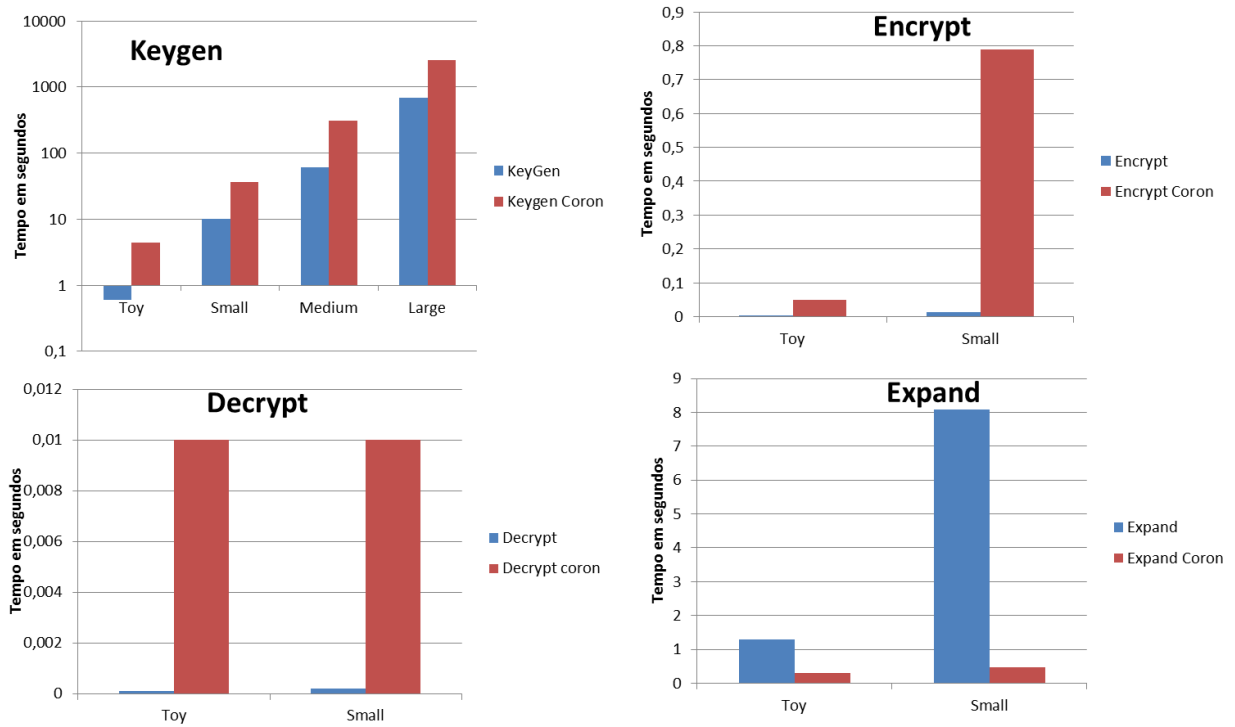


Figura 10: Comparação de tempos entre as implementações.

5.4 Conclusões

Como pode ser observado na Figura 10, a exceção da primitiva de *expand*, os resultados foram promissores, sendo ligeiramente mais rápidos que os obtidos por Coron. Deve-se notar que, apesar de velocidades compatíveis, conclusões mais profundas não podem ser obtidas desses dados, já que a implementação de Coron foi executada em uma máquina virtual SAGE 4.5.3 rodando em um processador Intel Core2 Duo E8500@ 3.12 GHz.

Por outro lado, esse trabalho prova o conceito de que um esquema criptográfico tal como os esquemas totalmente homomórficos podem ser implementados em linguagem de alto nível modernas, como o Python, com a ajuda de bibliotecas específicas para a matemática com inteiros longos envolvida.

A Figura 11 mostra dois gráficos onde se pode ver a relação de tempo entre as diferentes primitivas das duas implementações, nos parâmetros *toy* e *small*. Apesar de um campo frutífero em pesquisas, o custo computacional da criptografia totalmente homomórfica sobre números inteiros ainda é proibitivo para aplicações fora do âmbito acadêmico. Porém, é um consenso entre os autores e pesquisadores da área que, dentro de alguns anos, é possível

que essas novas técnicas de criptografia passem a ter usos práticos.

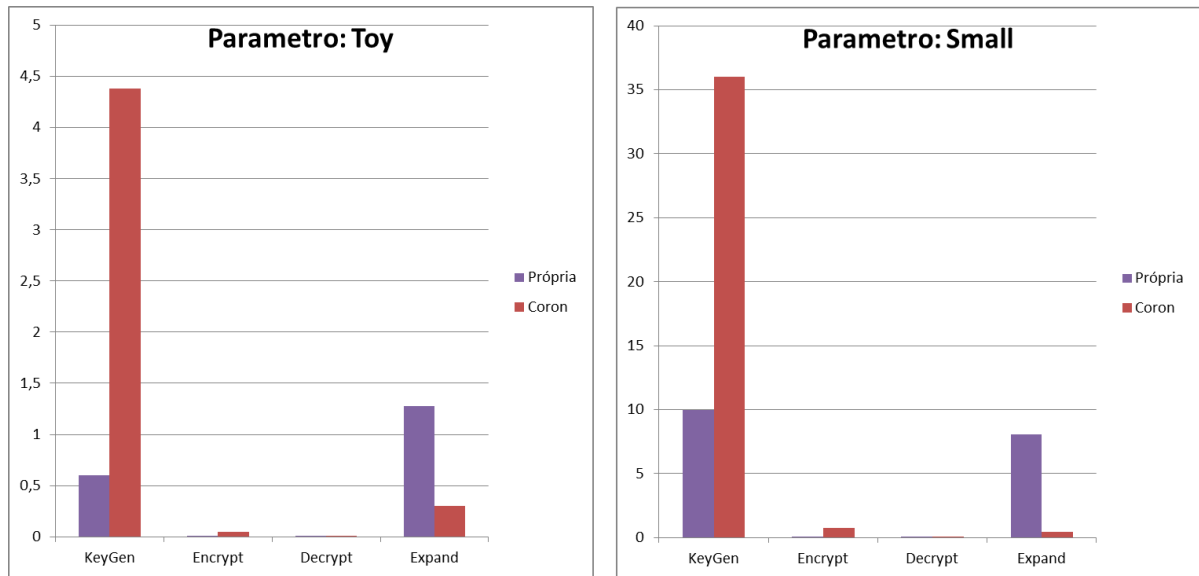


Figura 11: Comparação entre primitivas das diferentes implementações

5.5 Publicações

Durante a produção desse trabalho, foram feitas duas publicações com os resultados obtidos. A primeira no XV Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD) em 2014, onde um resumo foi publicado nos anais do evento e um banner apresentado entre as seções 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 do trabalho em seção técnica. Além disso, no SBSeg, o trabalho, que foi publicado em conjunto com Guilherme Rodrigues Bilar, foi agraciado com um prêmio de honra ao mérito, como um dos melhores trabalhos publicados no evento.

5.6 Trabalhos futuros

Sobre os resultados desse trabalho, refatoração do código, com o objetivo de melhorar desempenho ou legibilidade são possíveis linhas para futuros trabalhos. De forma similar, o código produzido pode ser utilizado por outros como base e documentação para implementações em outras tecnologias, tais como código para GPGPU e para FPGA.

Além disso, em trabalhos futuros a implementação da primitiva de *recrypt* irá agregar valor ao código atual.

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., 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.

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, Craig. **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.

PAILLIER, Pascal. **Public-key cryptosystems based on composite degree residuosity classes**. In: Advances in cryptology—EUROCRYPT'99. Springer Berlin Heidelberg, 1999. p. 223-238.

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.** In Shor, P. W., editor, 35th Annual Symposium on Foundations of Computer Science (Santa Fe, NM, 1994), páginas 124–134. IEEE Comput. Soc. Press, 1994.

STALLINGS, William. **Criptografia e Segurança de Redes: Princípios e Prática.** 4ª Edição. São Paulo: Pearson, 2007

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.

APÊNDICE A - Códigos

Código completo gerado durante o TCC.

1 keyGen.py

```

1  #FHE on Integers
2  '''
3  http://eprint.iacr.org/2011/441.pdf
4  http://eprint.iacr.org/2011/440.pdf
5  https://github.com/coron/fhe
6  '''
7
8  from gmpy2 import mpz
9  import random
10 import gmpy2
11 import time
12 import math
13 import parameters
14 import fheKey
15
16 debug = False
17
18 def printDebug(*args):
19     '''função para executar os prists de debug'''
20     if debug:
21         for i in args:
22             print(i, end=' ') # lint:ok
23         print('')
24
25 P=parameters.par()
26 P.setPar(tipo='toy') #seta parametos para o código
27
28 def qNear(a,b):
29     '''retorna quociente de a por b arredondado para o inteiro mais
30     proximo'''
31     return (2*a+b)//(2*b)
32
33 def modNear(a,b):
34     '''retorna a mod b no intervalo de [-(b/2), b/2]'''
35     return a-b*qNear(a,b)
36
37 def genZi(eta):
38     '''retorna um inteiro aleatorio impar de eta bits''' ##
39     eta=mpz(eta)

```

```

40     seed=int(time.time()*1000000) #time em microseconds
41     state=gmpy2.random_state(seed)
42     while True:
43         p=gmpy2.mpz_rrandomb(state, eta)
44         if gmpy2.is_odd(p): return p
45
46 def genPrime(b):
47     '''função retorna um primo MPZ de b bits''' ##
48     while True:
49         seed=int(time.time()*1000000) #time em microseconds
50         state=gmpy2.random_state(seed)
51         prime=gmpy2.mpz_rrandomb(state,b)
52         prime=gmpy2.next_prime(prime)
53         if len(prime)==b:
54             printDebug('gerado número primo de ', b, 'b')
55             return prime
56
57
58
59 def genX0(p, _gamma, _lambda):
60     '''gera o elemento q0 no intervalo [0, (2**gamma)/p) como sendo
61     o produto de
62     dois primos de lambda**2 bits e retorna x0 como sendo q0*p
63     o retorno da função é uma tupla (q0, x0)
64     '''
65     teto=gmpy2.f_div(2**mpz(_gamma), p) #divisão com retorno de
66     inteiro
67     while True:
68         q0=genPrime(_lambda**2)*genPrime(_lambda**2)
69         if q0 < teto: break
70     x0=gmpy2.mul(p, q0)
71     printDebug('genX0 executado, com parametro x0==', len(x0),
72     'e q0==', len(q0), 'bits')
73     printDebug('q0==', q0, '\nX0==', x0)
74
75     return q0, x0
76
77 def grand(q0):
78     '''retorna um número aleatorio no intervalo 0, q0'''
79     return random.randint(0, q0)
80
81 def rrand(rho):
82     '''retorna um número aleatorio entre -(2^rho) e 2^rho'''
83     limit=2**rho
84     return random.randint(-limit, limit)

```

```

def genX(beta, rho, p, q0): #TODO: talvez refazer isso com List
85 comprehension?
    '''Gera uma lista de beta pares x[i,0], x[i,1] seguindo a
86 formula
87     x[i,b]=p.q[i,b]+r[i,b] | 1<i<beta, 0<b<1
88     onde q[i,b] são aleatorios no intervalo [0, q0] e r[i,b] são
89     inteiros aleatorios no intervalo (-2**p, 2**p)
90     '''
91     x=list()
92     for i in range(beta*2):
93         result = gmpy2.mul(p, grand(q0))
94         result = gmpy2.add(result, rrand(rho))
95         x.append(result)
96     printDebug('gerada lista de elementos xi com', len(x),
97 'elementos (beta=', beta, ')')
98     return x
99
def genSk(theta, thetam): #TODO code verification
100     l=math.ceil(math.sqrt(theta)) #comprimento do vetor s
101     s0 = [1]+[0]*(l-1)
102     s1 = [1]+[0]*(l-1)
103     k=range(0, 4)
104     ks0=random.sample(k, 2)
105     ks1=random.sample(k, 4)
106     B=math.floor(math.sqrt(theta/thetam))
107     for k in ks0:
108         idx=random.randint((k*B)+1, (k+1)*B)
109         s0[idx-1]=1
110     for k in ks1:
111         idx=random.randint((k*B)+1, (k+1)*B)
112         s1[idx-1]=1
113     return s0, s1
114
def randomMatrix(i, j, kappa, se, sqrtTheta):
115     '''gera a matrix para u11 on the fly'''
116     if i==0 and j==0: return 0
117     random.seed(se)
118     itera = i*sqrtTheta+j
119     for i in range(itera):
120         a=random.getrandbits(kappa+1)
121     return a
122
def genU11(se, s0, s1, theta, kappa, p):
123
124     '''função para gerar o elemento u(1,1)'''
125     #gerar indices de elementos 1 dos vetores s0 e s1
126     si, sj = list(), list()
127     for i in range(len(s0)):
128         if s0[i]==1: si.append(i)
129

```

```

130     for j in range(len(s1)):
131         if s1[j]==1: sj.append(j)
132         #gerar matriz de raiz de theta elementos
133         l=math.ceil(math.sqrt(theta))
134         mx= 2**mpz(kappa+1)
135         #computa somatorio
136         xp=gmpy2.div(2**kappa,p)
137         xp=mpz(gmpy2.round_away(xp))
138
139         soma=modNear(xp, mx)
140         somatorio=0
141         for i in si:
142             for j in sj:
143                 somatorio += randomMatrix(i, j, kappa, se, l)
144         u=soma-somatorio
145         return u
146
147 def genU11_(se, s0, s1, theta, kappa, p):
148     '''função para gerar o elemento u(1,1)''' #old function
149     #gerar indices de elementos 1 dos vetores s0 e s1
150     si, sj = list(), list()
151     for i in range(len(s0)):
152         if s0[i]==1: si.append(i)
153     for j in range(len(s1)):
154         if s1[j]==1: sj.append(j)
155     #gerar matriz de raiz de theta elementos
156     l=math.ceil(math.sqrt(theta))
157     mx= 2**mpz(kappa+1)
158     #computa somatorio
159     xp=gmpy2.div(2**kappa,p)
160     xp=mpz(gmpy2.round_away(xp))
161     soma=xp%mx
162     somatorio=0
163     for i in si:
164         for j in sj:
165             somatorio += randomMatrix(i, j, kappa, se, l)
166     u=soma-somatorio
167     return u
168
169 def encryptVector(s, p, q0, x0, rho):
170     sigma=list()
171     p2=2**rho
172     for i in s:
173         r = random.randint(-p2, p2)
174         q = random.randint(0, q0-1)
175         c = modNear((i+(2*r)+(p*q)),x0)

```

```

176         sigma.append(c)
177     return sigma
178
179 def keygen(file, size='toy'):
180     P.setPar(size)
181     tempo=-time.time()
182
183     p = genZi(P._eta)
184     #print("p computado")
185     q0, x0 = genX0(p, P._gamma, P._lambda)
186     #print("q0,x0 computado")
187     listaX = genX(P._beta, P._rho, p, q0)
188     #print("listax computado")
189     pkAsk = listaX
190     pkAsk.insert(0, x0)
191     print("pk* computado")
192     while True:
193         s0,s1=genSk(P._theta, P._thetam)
194         if (s0.count(1)*s1.count(1)==15): break
195
196     se=int(time.time()*1000) #seed para RNG
197     _kappa=P._gamma+6 #variavel para o RNG, conforme pg9 coron
198     u11=genU11(se, s0, s1, P._theta, _kappa, p)
199     sigma0 = encryptVector(s0, p, q0, x0, P._rho)
200     sigma1 = encryptVector(s1, p, q0, x0, P._rho)
201     tempo+=time.time()
202     #pickle files
203
204     public=fheKey.pk(pkAsk, se, u11, sigma0, sigma1, P)
205     secret=fheKey.sk(s0, s1, P)
206     fheKey.write(public, 'pk_pickle_'+file)
207     fheKey.write(secret, 'sk_pickle_'+file)
208
209     #write key to file
210     f = open(file, 'w')
211     f.write(("keygen executado em " +str(tempo) +"
segundos"+'\n\n'))
212     f.write(('p==' + str(p)+'\n\n'))
213     f.write(('q0==' +str(q0)+'\n\n'))
214     f.write(('pk*==' +str(pkAsk)+'\n\n'))
215     f.write(('s0 ==' +str(s0)+'\n\n'))
216     f.write(('s1 ==' +str(s1)+'\n\n'))
217     f.write(('se ==' +str(se)+'\n\n'))
218     f.write(('u11 ==' +str(u11)+'\n\n'))
219     f.write(('sigma0 ==' +str(sigma0)+'\n\n'))
220     f.write(('sigma1 ==' +str(sigma1)+'\n\n'))
221     f.close

```

```

222     print('arquivo salvo', file)
223
224 def testRoutine():
225     f=open('tempoKeygen441.txt', 'w')
226     t=-time.clock()
227     keygen('toy.txt', 'toy')
228     t+=time.clock()
229     f.write('toy == '+str(t)+'\n')
230
231     t=-time.clock()
232     keygen('small.txt', 'small')
233     t+=time.clock()
234     f.write('small == '+str(t)+'\n')
235
236     t=-time.clock()
237     keygen('medium.txt', 'medium')
238     t+=time.clock()
239     f.write('medium == '+str(t)+'\n')
240
241     t=-time.clock()
242     keygen('large.txt', 'large')
243     t+=time.clock()
244     f.write('large == '+str(t)+'\n')
245     f.close()
246
247 if __name__=='__main__':
248     print("Biblioteca geradora de chaves. Função principal:
249     keygen(file, [parametro de segurança])")
250     a=input("deseja executar a rotina de testes? <y/n> ")
251     if a=='y':
252         testRoutine()
253         print('finalizado!')
254     else: quit(0)

```

2 fleKey.py

```

1 import pickle
2 '''
3 biblioteca que implementa as classes para a chave pública e privada,
4 além
5 das funções necessarias para escrever e ler esses objetos, usando
6 pickle
7 '''
8 class pk():
9     '''clase para armazenar a chave privada'''

```

```

8     def __init__(self, pkAsk, se, u11, sigma0, sigma1, P):
9         self.pkAsk=pkAsk
10        self.se=se
11        self.u11=u11
12        self.sigma0=sigma0
13        self.sigma1=sigma1
14        self.P=P
15
16    class sk():
17        '''classe para armazenar a chave pública'''
18        def __init__(self, sz, su, P):
19            self.s0=sz
20            self.s1=su
21            self.P=P
22
23    def write(obj, name):
24        '''faz uma coserva de obj no arquivo name'''
25        with open(name, 'wb') as arq:
26            pickle.dump(obj, arq)
27
28    def read(name):
29        '''abre o pote name e retorna o pickles'''
30        with open(name, 'rb') as arq:
31            return pickle.load(arq)
32
33    if __name__=='__main__':
34        '''classe de testes'''
35        publicKey=pk(1,2,3,4,5, 'parametros')
36        secretkey=sk(7,8, 'parametros')

```

3 __main__.py

```

1    '''versão v.02'''
2    import keyGen
3    from os import system
4
5    def cls():
6        system('cls')
7
8    def firstScreen():
9        cls()
10       print("FHE over Integers")
11       print("1-Geração de chaves")
12       print("2-Teste com geração de chave")

```

```

13     print("3-Teste com chave previamente gerada")
14     print("0-Sair")
15     a = input()
16     return int(a)
17
18 def key():
19     a=input('<Toy, Small, Medium, Large>? ').upper()
20     if a=='T': keyGen.keygen('toy.txt', 'toy')
21     elif a=='S': keyGen.keygen('small.txt', 'small')
22     elif a=='M': keyGen.keygen('medium.txt', 'medium')
23     elif a=='L': keyGen.keygen('large.txt', 'large')
24     b = input('any to continue...') # lint:ok
25
26 if __name__=='__main__':
27     while True:
28         a=firstScreen()
29         if a==1: key()
30         elif a==2: print('not yet')
31         elif a==3: print('not yet')
32         elif a==0: quit(0) # lint:ok
33         else: print('Are you sure you know how to handle me?')

```

4 Encrypt.py

```

1 import fheKey
2 import random
3 from gmpy2 import mpz
4 import gmpy2
5 import time
6 from keyGen import randomMatrix, modNear
7 import math
8 import gc
9 debug = False
10
11 def printDebug(*args):
12     if debug:
13         for i in args:
14             print (i, end=' ') # lint:ok
15
16 def printM(y):
17     if debug:
18         for i in y:
19             for j in i:
20                 print (j, end=' ') # lint:ok

```

```

21         print(' ')
22
23     def encrypt(pk, m):
24         if m != 0 and m != 1:
25             print('not a valid m')
26             return 0
27
28         alpha = pk.P._lambda ##o fator de segurança alpha é igual a
29         lambda, de acordo com Coron et. al.
30         matrix = [[0 for i in range(pk.P._beta)] for j in
31         range(pk.P._beta)]
32         for i in range(pk.P._beta):
33             for j in range(pk.P._beta):
34                 matrix[i][j]=random.randint(0, 2**alpha -1)
35                 #print(matrix[i][j], end=' ')
36                 #print(' ')
37             pprime=2**pk.P._rho
38             r=random.randint(-pprime, pprime)
39             pkAsk=(pk.pkAsk).copy()
40             x0=pkAsk.pop(0)
41             xi0=pkAsk[:2]
42             xj1=pkAsk[1::2]
43
44             ## iniciando processo de encrypt
45             ## computar somatorio
46             somatorio=mpz(0)
47             for i in range(pk.P._beta):
48                 for j in range(pk.P._beta):
49                     x=gmpy2.mul(xj1[j], xi0[i])
50                     somatorio += gmpy2.mul(matrix[i][j], x)
51             c=(m+(2*r)+(2*somatorio))
52             c=modNear(c,x0)
53             return c
54
55     def expand(pk, cAsk):
56         #cria uma matriz de sqrt(theta) elementos
57         l=int(math.sqrt(pk.P._theta))
58         y=[[0 for i in range(l)] for i in range(l)]
59         kappa=pk.P._gamma+6
60
61         n=4 ##coron 441, pg10
62         gmpy2.get_context().precision=n #seta precisao do mpfr para 2
63         #computa matriz y[i][j]
64         for i in range(l):
65             for j in range(l):
66                 y[i][j]=float(gmpy2.mpfr(randomMatrix(i, j, kappa,
67 pk.se, l)/(2**kappa)))

```

```

66     y[0][0]=float(gmpy2.mpfr(pk.u11/2**kappa))
67
68     #seta precisão do mpfr de novo para o valor correto
69     gmpy2.get_context().precision=1024 #valor feito com testes.
70     #gera matrix z e computa elementos
71     expand = [[1 for i in range(1)] for i in range(1)]
72     for i in range(1):
73         for j in range(1):
74             expand[i][j]=float((cAsk * y[i][j])%2)
75     return expand
76
77 def decrypt(sk, cAsk, z):
78     soma=0
79     l=int(math.sqrt(sk.P._theta))
80     for i in range(1):
81         for j in range(1):
82             soma+=(sk.s0[i]*sk.s1[j]*z[i][j])
83     soma=gmpy2.round_away(soma)
84     m=(cAsk-soma)%2
85     return m
86
87 def add(pk, c1, c2):
88     soma=gmpy2.add(c1,c2)
89     return modNear(soma,pk.pkAsk[0])
90
91 def mul(pk, c1, c2):
92     mul= gmpy2.mul(c1,c2)
93     return modNear(mul,pk.pkAsk[0])
94
95
96 def teste():
97     '''teste para funções com key'''
98     par=['toy','small', 'medium', 'large']
99     for i in par:
100         f=open('temposEncryptDecryptExpand.txt', 'a')
101         print('executing ', i)
102         f.write(i+'\n')
103         pk=fheKey.read('pk_pickle_'+i+'.txt')
104         sk=fheKey.read('sk_pickle_'+i+'.txt')
105         t=-time.clock()
106         e0=encrypt(pk, 0)
107         t+=time.clock()
108         f.write('encrypt=='+str(t)+'\n')
109
110         #e1=encrypt(pk, 1)
111         #print('bit0==', e0)

```

```

112         #print('bit1==', e1)
113         #added=add(pk, e0, e1)
114         #mult=mul(pk, e0, e1)
115
116         t--time.clock()
117         ze0=expand(pk, e0)
118         t+=time.clock()
119         f.write('expand=='+str(t)+'\n')
120         #ze1=expand(pk, e1)
121
122         #zadd=expand(pk, added)
123         #zmul=expand(pk, mult)
124         t--time.clock()
125         d0=decrypt(sk, e0, ze0)
126         t+=time.clock()
127         f.write('decrypt=='+str(t)+'\n')
128
129         #d1=decrypt(sk, e1, ze1)
130         #print('isso deve ser um 1+0 =>', decrypt(sk, added, zadd))
131         #print('isso deve ser um 1*0 =>', decrypt(sk, mult, zmul),
132         '\n')
133         f.close()
134
135 if __name__=='__main__':
136     print('iniciada rotina de testes:')
137     teste()
138     gc.collect()

```

5 parameters.py

```

1 class par:
2     '''classe para armazenar os parametros utilizados no sistema
3     homomórfico
4     parametros indicados no trabalho de coron et. al.
5     '''
6     _lambda=42
7     _rho    =16
8     _eta    =1088
9     _gamma  =160000
10    _beta   =12
11    _theta  =144
12    _thetam=15
13    def setPar(self, tipo):
14        if tipo == 'toy':
15            self._lambda=42

```

```

15         self._rho      =16
16         self._eta      =1088
17         self._gamma    =160000
18         self._beta     =12
19         self._theta    =144
20
21     elif tipo == 'small':
22         self._lambda=52
23         self._rho    =24
24         self._eta    =1632
25         self._gamma  =86000
26         self._beta   =23
27         self._theta  =533
28
29     elif tipo == 'medium':
30         self._lambda=62
31         self._rho    =32
32         self._eta    =2176
33         self._gamma  =4200000
34         self._beta   =44
35         self._theta  =1972
36
37     elif tipo == 'large':
38         self._lambda=72
39         self._rho    =39
40         self._eta    =2652
41         self._gamma  =19000000
42         self._beta   =88
43         self._theta  =7897
44     else:
45         print('invalid argument')
46         quit(0) # lint:ok
47
48 def __init__(self, tipo='toy'):
49     self.setPar(tipo)

```

APÊNDICE B – Configuração da IDE

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 feita 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/>.