UNIVERSIDADE FEDERAL FLUMINENSE

RAPHAEL BERNARDINO FERREIRA LIMA

Uma biblioteca na linguagem reversível Janus para implementação de procedimentos criptográficos

NITERÓI

2019

UNIVERSIDADE FEDERAL FLUMINENSE

RAPHAEL BERNARDINO FERREIRA LIMA

Uma biblioteca na linguagem reversível Janus para implementação de procedimentos criptográficos

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Sistemas de Computação.

Orientador:
LUIS ANTONIO BRASIL KOWADA

NITERÓI

2019

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

L732b Lima, Raphael Bernardino Ferreira
Uma biblioteca na linguagem reversível Janus para implementação de procedimentos criptográficos / Raphael Bernardino Ferreira Lima ; Luis Antonio Brasil Kowada, orientador. Niterói, 2019.
124 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2019.

DOI: http://dx.doi.org/10.22409/PGC.2019.m.14454501785

1. Programação. 2. Criptografia. 3. Produção intelectual. I. Kowada, Luis Antonio Brasil, orientador. II. Universidade Federal Fluminense. Instituto de Computação. III. Título.

CDD -

Bibliotecária responsável: Fabiana Menezes Santos da Silva - CRB7/5274

RAPHAEL BERNARDINO FERREIRA LIMA

Uma biblioteca na linguagem reversível Janus para implementação de procedimentos criptográficos

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Sistemas de Computação.

Aprovada em março de 2019.

BANCA EXAMINADORA

Prof. Luis Antonio Brasil Kowada - Orientador, UFF

Prof. Franklin Marquezino - UFRJ

Prof. Yuri Abitbol de Menezes Frota - UFF

Profa. Simone de Lima Martins - UFF

Niterói

2019

Dedico este trabalho a todas as pessoas que auxiliaram a concluí-lo.

Agradecimentos

Agradeço primeiramente a Deus por ter me dado o conhecimento necessário para a conclusão deste trabalho.

Agradeço aos meus pais que sempre estiveram presentes e ajudaram em todos os momentos de dificuldades.

Também agradeço ao orientador Luis Antonio Kowada por permitir a realização deste trabalho e auxiliar nas situações mais inesperadas e complexas. Agradeço ao Daniel Chicayban Bastos por ter gasto um tempo precioso para revisar esta dissertação, tecer comentários a cerca dos algoritmos e referências.

Agradeço a disposição dos membros da banca Franklin Marquezino, Yuri Abitbol e Simone de Lima em participar da avaliação deste trabalho e propor sugestões de melhoria do mesmo.

E, por fim, quero deixar registrado que sou muito grato por todo o apoio prestado pelos amigos e colegas da Universidade Federal Fluminense.

Resumo

Neste trabalho, propomos uma biblioteca de funções para a linguagem de programação reversível Janus. As cifras de criptografia simétrica AES e ChaCha são implementadas utilizando funções desta biblioteca e, consequentemente, da linguagem. A biblioteca conta com funções de matemática modular, geração de números pseudoaleatórios, algoritmo de Euclides estendido, operações binárias, cálculo de logaritmo e operações sobre vetores. As implementações são comparadas com versões em alto e baixo nível, a fim de medir a eficiência.

Palavras-chave: Programação reversível. Computação reversível. Criptografia. Algoritmos de chave simétrica.

Abstract

In this work, we propose a library to the reversible programming language Janus with a wide range of functions: mathematical modular operations, a pseudorandom number generator function, the extended Euclidean algorithm, binary operations, a generic logarithm function and vector operations. Symmetric key ciphers such as AES and ChaCha are implemented. To measure the efficiency of these implementations, the number of operations are compared to circuits and high-level versions.

Keywords: Reversible programming. Reversible computing. Cryptography. Symmetric-key algorithm.

Lista de Figuras

1.1 Gramática na forma EBNF para a linguagem Janus [32, 63]. 6

2.1 Previsão do tamanho das portas físicas dos transistores segundo os relatórios de 2013 e 2015 feitos pela ITRS. 10

2.2 Representação das portas lógicas (a) Toffoli, (b) Fredkin e (c) CNOT. . . . 12

2.3 Sintaxe e gramática inicial da linguagem Janus (Janus86). 16

2.4 Representação do fluxo utilizado nas estruturas condicionais e de repetição. 17

2.5 Sintaxe e gramática da versão mais nova da linguagem Janus (JanusP). . . 18

3.1 Representação da operação de deslocamento a direita. O *bit* menos significativo é perdido e um novo *bit "0"* é adicionado na posição do *bit* mais significativo. 33

3.2 Representação da operação de rotação a direita. O *bit* menos significativo se torna o *bit* mais significativo e todos os demais são reposicionados uma unidade à direita. 33

Lista de Tabelas

2.1 Comparação entre as linguagens JanusB e JanusP. 18

3.1 Tabela-verdade para a operação de conjunção lógica (AND). 29

3.2 Tabela-verdade para a operação de disjunção lógica (OR). 30

3.3 Tabela-verdade para a operação de disjunção exclusiva (XOR). 31

4.1 Tabela de substituição SBOX utilizada no procedimento SubBytes. 58

4.2 Tabela SBOX-Inv utilizada no procedimento inverso da SubBytes. 58

4.3 Tabela de constantes utilizada no procedimento AddRoundKey. 59

4.4 Quantidade de operações necessárias utilizando 8 bits para cada função. . . 74

4.5 Quantidade total de operações utilizadas em cada função do AES baseando-se na implementação ao longo desta dissertação. 76

4.6 Quantidade total de operações necessárias realizadas pelo circuito proposto por Grassl et al. [28] mapeadas para implementação desenvolvida nesta dissertação. 77

4.7 Tabela comparativa para ilustrar a quantidade de operações feitas em cada abordagem. A versão em circuitos possui foco em reduzir a quantidade de recursos utilizados, enquanto a versão Janus tem como objetivo reduzir a quantidade de operações. 79

4.8 Tabela ilustrativa do conteúdo do vetor no algoritmo Salsa. 81

4.9 Tabela ilustrativa da reestruturação feita no algoritmo ChaCha. 83

4.10 Comparação da quantidade de operações feitas na implementação proposta e na DIKU. 95

A.1 Tabela completa de constantes de rodada. 103

A.2 Tabela completa do corpo de Galois GF02. 104

A.3 Tabela completa do corpo de Galois GF03. 104

A.4 Tabela completa do corpo de Galois GF09. 104

A.5 Tabela completa do corpo de Galois GF11. 104

A.6 Tabela completa do corpo de Galois GF13. 105

A.7 Tabela completa do corpo de Galois GF14. 105

Sumário

1 Introdução **1**

1.1 Estado da arte 4

1.2 Proposta e objetivo 7

2 Linguagem de programação reversível **8**

2.1 Fundamentos de uma linguagem reversível 13

2.1.1 Updates reversíveis 13

2.1.2 Determinismo reverso 14

2.1.3 Acesso às semânticas inversas 15

2.1.4 Eficiência do programa reversível 15

2.2 Janus 16

3 Biblioteca Kairos **19**

3.1 Aritmética Modular 19

3.1.1 Adição e subtração modular 20

3.1.2 Multiplicação modular 22

3.1.3 Exponenciação modular 26

3.2 Operações Binárias 29

3.2.1 Operação AND 29

3.2.2 Operação OR 30

3.2.3 Operação XOR 31

3.2.4 Operação de complemento a um e a dois 31

3.2.5 Left-Rotate e Right-Rotate 33
3.3 Funções matemáticas 35
3.3.1 Função de fatoração 35
3.3.2 Função de raiz quadrada 37
3.3.3 Função de logaritmo base N 37
3.3.4 Algoritmo de Euclides estendido 39
3.3.5 Função de valor absoluto 42
3.3.6 Função teto (ceil) 43
3.3.7 Função piso (floor) 43
3.4 Operações sobre vetores (arrays) 44
3.4.1 Cópia de vetores 44
3.4.2 Rotação para esquerda e direita 45
3.5 Pseudorandom number generator (PRNG) 46

4 Implementações criptográficas 52

4.1 Introdução a chaves simétricas 54
4.2 Caso 1 - AES 56
4.2.1 Algoritmo do AES 56
4.2.2 Implementação do AES 62
4.2.2.1 Inicialização do AES 62
4.2.2.2 Encriptação e decriptação 65
4.2.2.3 Expansão de chaves 66
4.2.2.4 AddRoundKey 69
4.2.2.5 SubBytes e SubBytes-Inv 69
4.2.2.6 ShiftRows e ShiftRows-Inv 71
4.2.2.7 MixColumns e MixColumns-Inv 71
4.2.3 Comparação com a versão em circuitos 73

4.3 Caso 2 - ChaCha20 79

4.3.1 Algoritmo do ChaCha 80

4.3.2 Implementação do ChaCha 84

4.3.2.1 Inicialização do ChaCha 84

4.3.2.2 Core e Quarter Round 86

4.3.2.3 Encriptação e decriptação 88

4.3.3 Comparação com implementação DIKU 89

5 Conclusão **96**

Referências **98**

Apêndice A – Tabelas AES **103**

A.1 Tabela RCON completa 103

A.2 Tabelas do Corpo Finito de Galois 103

Apêndice B – Exemplo ChaCha da JanusP **106**

B.1 Implementação do ChaCha20 106

Capítulo 1

Introdução

Um dos problemas da computação convencional é a grande quantidade de calor produzida (e consequentemente da energia necessária). Este é um problema inerente do modelo usado atualmente, seja utilizando a arquitetura de von Neumann (também conhecida como arquitetura de Princeton), seja utilizando outras arquiteturas como Arquitetura de Harvard, cujos dados ficam em memória separada das instruções. Isto acontece porque este modelo convencional é irreversível, ou seja, as informações podem ser inseridas, apagadas ou modificadas. Landauer [38] mostrou em 1961 que qualquer operação que apaga informação necessariamente dissipa calor nos processos computacionais, visto que há aumento da entropia [25]. Mas existem modelos de arquitetura nos quais não há aumento de calor, que são os modelos de computação reversível. Os processos reversíveis não apagam informação de forma que é possível voltar para o estado inicial, descartando assim a necessidade de sobrescrever, ou apagar, os conteúdos. Se esse processo fosse totalmente adiabático (chamada computação adiabática), a energia contida no circuito poderia ser totalmente recuperada e reutilizada nos demais processos. Na prática, devido aos componentes e materiais utilizados, sempre há perda de calor e, consequentemente, de energia. Mas é muito menor do que da computação convencional.

Esta questão energética não é irrelevante. Atualmente, nos Estados Unidos, há um gasto anual de 3,82 trilhões de quilowatt-hora (kWh), ou 3,82 petawatt-hora (pWh), onde os computadores clássicos representam em torno de 15% [60] deste valor. E essa razão deve crescer nos próximos anos, visto que a cultura "inteligente" (*smartphones*, *smartwatches* e afins) está tomando força na sociedade. Outro fator crescente na sociedade moderna é o uso de moedas alternativas descentralizadas como o Bitcoin (BTC) [16]. Um estudo recente [17] relatou que 45 terawatt-hora são gastos apenas na criptomoeda Bitcoin e que tal utilização seria suficiente para iluminar diversos países pequenos como Hong Kong,

Iraque e Peru, ou até mesmo suprir mais que a metade da demanda da República Checa. Vale notar que o preço do Bitcoin não reflete o custo energético e sim a eficiência computacional dos equipamentos e tecnologias utilizadas ou, em outras palavras, o esforço gasto. Um estudo feito pela EliteFixtures [19] sobre o custo operacional para minerar 1 BTC foi feito baseado na dificuldade apresentada em janeiro de 2018, onde foram considerados os preços por kWh e quantidade de energia gasta pelos equipamentos AntMiner S9 que usa 17 megawatts e leva 548 dias, S7 que gasta 45 megawatts e demora 1580 dias, e o Avalon 6 que possui um gasto energético de 55 megawatts e consegue obter um Bitcoin no espaço de tempo igual a 2194 dias. Os resultados mostram que países como Venezuela, que possui uma grande parte da energia subsidiada pelo governo, possuem baixo custo de mineração, enquanto países mais restritos e fechados economicamente como Coreia do Sul possuem um alto custo. Michael Frank, um professor assistente no CISE (Computer and Information Science and Engineering), diz que, em uma tradução livre, "Os processadores mais rápidos disponíveis atualmente dissipam na ordem de 100 *watts* em forma de calor"[1]. A computação reversível idealizada na década de 60 tinha como pressuposto manipular 0s e 1s digitalmente, assim eles poderiam ser desfeitos ou revertidos. Esse processo difere da abordagem atual, na qual as operações são feitas e depois o resultado é descartado. Mas sob a ótica da computação clássica quando há a necessidade de "apagar" um valor, parte do circuito que contém a carga é aterrado, convertendo-a e destruindo toda e qualquer informação guardada, o que por sua vez resultará na dissipação da mesma em forma de calor. Segundo estudos [7] essa energia gasta pode ser reduzida para, no mínimo, $17meV$ em temperatura ambiente, porém a tecnologia *CMOS* atual é pior do que a imaginada por Landauer, fazendo com que esse valor seja em torno de $5keV$ por bit apagado. Quando pensamos em apenas uma operação, isso pode até não parecer ineficiente, ou ruim, mas lembre que são realizadas bilhões de operações por segundo nos computadores atuais, logo há bilhões de resultados sendo "apagados", ou descartados, a cada segundo fazendo com que essa abordagem seja ineficiente em termos energéticos. Além disso, devido a essa abordagem, uma limitação é criada onde a quantidade de *chips* que podem ser colocados juntos é reduzida, impactando indiretamente na performance e miniaturização de equipamentos. De forma mais clara, podemos citar como exemplo os *smartphones* que são extremamente prejudicados devido a gastarem muita energia, não poderem ter seus componentes organizados de forma mais próxima, ou eficiente, por razões físicas de dissipação do calor e que, consequentemente, sofrem um decaimento na performance geral do sistema.

[1] The fastest processors available today dissipate on the order of 100 watts of power in the form of heat.

Em paralelo com a pesquisa feita em circuitos reversíveis, surgiram algumas linguagens de programação que pretendem ser reversíveis, como, por exemplo, a linguagem de programação reversível Janus, proposta por Lutz e Derby em 1982 [41], e que, apesar de ser muito simples e precária, provou-se suficiente para o desenvolvimento de algoritmos complexos, como indica K. S. Perumalla [49]. Uma linguagem é dita reversível se suas construções podem ser escritas de forma reversível, ou seja, são capazes de recuperar o estado anterior sem que haja perda de informação. Programas irreversíveis podem ser escritos nessa linguagem de forma a reduzir o custo computacional, em termos de memória e tempo de execução, em relação às plataformas reversíveis. Como estas plataformas precisariam simular o ambiente e sua execução, haveria uma sobrecarga de memória e um aumento no tempo de execução, o que não irá ocorrer se o programa for escrito diretamente em uma linguagem reversível. Yokohama e Glück [65] propuseram algumas modificações e aprimoraram a descrição formal da linguagem, definindo uma gramática mais enxuta, clara e eficiente. Mas ainda há muito o que se fazer, já que a linguagem não possui diversas funções *built-in* que são imprescindíveis para a escrita de programas complexos de forma eficiente. Diante destes fatos, propomos a criação de uma biblioteca que não apenas auxiliasse o programador na escrita eficiente como também na compreensão de problemas eventuais, visto que esse terá acesso ao código-fonte.

No Capítulo 2 são apresentados os fundamentos de uma linguagem reversível explicando a definição, regras para montagem das mesmas, assim como, a linguagem de *workflow* reversível Janus e suas versões, apresentando a gramática e funcionamento dos *updates*. O objetivo desse capítulo é fazer com que os conceitos apresentados futuramente sejam mais facilmente compreendidos e tragam elucidação de alguns detalhes de uma linguagem reversível em relação às outras.

No Capítulo 3 é apresentada a implementação dos algoritmos propostos realizando uma análise comparativa com alguns circuitos descritos na literatura que possuem o mesmo objetivo. Como as implementações descritas nesta dissertação foram feitas em alto nível, ou seja, em uma linguagem de programação, não é esperado que a quantidade de operações ou espaço utilizado pelo algoritmo seja próximo da implementação de baixo nível, ou seja, em circuito. Porém isso nos traz pontos positivos de legibilidade, compreensão e, possivelmente, otimização visto que podem ser modificados com mais facilidade, rapidez e menor custo.

E por último, mas não menos importante, no Capítulo 4 são mostradas aplicações que utilizam alguns dos algoritmos apresentados no capítulo anterior. Grande parte das

aplicações são limitadas devido a alguns problemas técnicos da própria linguagem, como o suporte a apenas números inteiros. O nicho de criptografia foi escolhido devido a biblioteca proposta possuir foco em operações aritméticas, possuir algoritmos que não dependem de funções gráficas ou de desenho, acessos a disco e rede ou manipulação de estruturas de dados complexas. Uma conclusão contendo as contribuições, aprendizado e trabalhos futuros se encontra no Capítulo 5.

1.1 Estado da arte

Apesar do grande estudo e esforço feito no sentido da compreensão da computação reversível em baixo nível, houve certos avanços na abordagem mais compreensível, ou de alto nível, para humanos. Atualmente existem diversas linguagens de programação imperativas reversíveis e, devido a essa grande variedade, algumas orientadas a objetos como, por exemplo, ROOPL [32] começaram a aparecer recentemente. Dentre elas podemos citar algumas como Janus [41], R [24], psiLisp [3], Inv [45, 46], Gries' invertible language [29] e rFun [2, 64]. Além disso, arquiteturas reversíveis como Pendulum [62] e Bob [59], e seus respectivos conjunto de instruções denominados de PISA e BobISA, foram criadas.

A arquitetura Pendulum, que é baseada na arquitetura de von Neumann e que possui apenas uma memória, foi feita por Vieri devido às soluções existentes na época possuírem "defeitos", ou falta de preocupação, em certos aspectos como utilização apenas de portas Fredkin [52] e negligenciar o fluxo de controle nas operações [53], ou incompleta devido a não haver um mapeamento entre a ISA (*instruction set architecture*) e o RTL (*register transfer level*), porém neste último caso Hall [31] se baseou na PDP-10, uma arquitetura que também possui o mesmo problema em sua forma irreversível. Para resolver essas questões e dificuldades, Vieri utilizou uma arquitetura RISC (*Reduced instruction set computer*) e modificou a forma na qual as instruções eram interpretadas, de forma a não quebrar a reversibilidade das operações. Por este motivo, foi idealizado que a arquitetura Pendulum utilizaria três registradores para o controle de fluxo, sendo estes: PC (*program counter*) que armazena o endereço da instrução atual, BR (*branch register*) para armazenar o *offset* de salto e DIR (*direction bit*) que determina em que direção as instruções devem ser executadas e possui os valores -1 e 1. Caso o BR seja igual a zero, o valor de DIR é somado ao PC, ou seja, o *program counter* será incrementado ou decrementado dependendo da direção informada pelo registrador DIR. No caso do valor do *branch* ser diferente de zero, será calculado o *jump offset* necessário que possui a próxima instrução através do produto dos valores dos registradores BR e DIR. Portanto, os *jumps* são feitos em ambos

sentidos dependendo apenas do *bit* em BR e possibilitam que o registrador BR seja zerado a cada salto realizando a soma do produto encontrado anteriormente. Além disso, há a possibilidade de inverter o valor do registrador BR através do salto incondicional **BRA** e sua versão inversa **RBRA**, que é essencial para diversas linguagens poderem realizar as chamadas de procedimentos e suas versões inversas respectivas. Outras instruções como **SWAPBR** também são interessantes para implementar saltos dinâmicos, ou seja, que dependem da avaliação de uma variável em tempo de execução. A consulta e escrita em memória são feitas através da instrução **EXCH** enquanto que **DATA** armazena o valor em uma determinada célula de memória. As demais operações são similares às da arquitetura RISC, com alguns detalhes como, o **AND** é substituído por uma versão reversível denominada **ANDX** onde o resultado é armazenado em um terceiro registrador para garantir sua reversibilidade.

A arquitetura Bob, baseada na arquitetura abstrata Harvard com seu conjunto de instruções BobISA criado por Thomsen *et al.*, possui duas memórias: uma para armazenar os dados do programa e outra para as instruções do programa. Essa abordagem foi escolhida por simplificar o modelo reversível de forma que uma instrução em memória não possa modificar seu próprio conteúdo. Algumas diferenças em relação à Pendulum são que o registrador **BR** e a instrução **SWAPBR**, aqui chamada de **SWBR**, possuem 8 *bits*, portanto o salto máximo é de 127 linhas na direção informada pelo registrador **DIR**. Outras três instruções foram adicionadas, sendo a primeira instrução **RSWB**, que tem como objetivo realizar o *swap* e inverter o valor de **BR** diretamente. As outras duas instruções foram adicionadas com a finalidade de dobrar e dividir pela metade o valor contido no registrador de 4 *bits* com complemento-a-2 sendo, respectivamente, **MUL2** e **DIV2**.

A linguagem Janus, que será amplamente abordada ao longo desta dissertação, criada em 1982 por Lutz e Derby para uma aula no Caltech [2], recebeu este nome devido a representar o deus responsável pelo início e fim na mitologia greco-romana. Assim como o deus Janus (ou em português, Jano) a linguagem possui duas "faces" que servem para analisar o passado e futuro, ou seja, o estado inicial e final de uma aplicação ou programa. A linguagem procedural é estruturada através de procedimentos e conta com três tipos de dados: inteiros, vetores de inteiros com tamanho fixo e pilhas de inteiros que servem como vetores de tamanho dinâmico. Um programa escrito nesta linguagem conta com o procedimento principal seguido de outros procedimentos que não possuem retorno. Para obter o efeito de uma função, ou seja um procedimento com retorno, há a necessidade

[2] também conhecido como *California Institute of Technology*.

de passar como parâmetro adicional onde o resultado deve ser salvo. Além disso, essa linguagem é conhecida por ser *r-Turing complete*, ou seja, é capaz de simular qualquer máquina de Turing reversível. Em termos de compilação, Axelsen [1] realizou uma tradução para a arquitetura Pendulum, Brum [9] desenvolveu um compilador *offline* para a Janus86 e a linguagem também conta com um compilador *online* em desenvolvimento pelo departamento de informática da Universidade de Copenhague.

$prog$	::=	$p_{main}\ p^*$	(program)
t	::=	**int** | **stack**	(data type)
p_{main}	::=	**procedure main** `()` (**int** x(`[`$\overline{n}$`]`)$^?$ | **stack** x)* s	(main procedure)
p	::=	**procedure** q`(`$t\ x$, ..., $t\ x$`)` s	(procedure definition)
s	::=	$x\ \odot$ `=` e | x`[`e`]` $\odot$ `=` e	(assignment)
	|	**if** e **then** s **else** s **fi** e	(conditional)
	|	**from** e **do** s **loop** s **until** e	(loop)
	|	**push**`(`x, x`)` | **pop**`(`x, x`)`	(stack modification)
	|	**local** $t\ x$ `=` e s **delocal** $t\ x$ `=` e	(local variable block)
	|	**call** q`(`x, ..., x`)` | **uncall** q`(`x, ..., x`)`	(procedure invocation)
	|	**skip** | $s\ s$	(statement sequence)
e	::=	$\overline{n}$ | x | x`[`e`]` | $e\ \otimes\ e$ | **empty**`(`x`)` | **top**`(`x`)` | **nil**	(expression)
$\odot$	::=	`+` | `-` | `^`	(operator)
$\otimes$	::=	$\odot$ | `*` | `/` | `%` | `&` | `|` | `&&` | `||` | `<` | `>` | `=` | `!=` | `<=` | `>=`	(operator)

Figura 1.1: Gramática na forma EBNF para a linguagem Janus [32, 63].

Em relação à gramática visualizada na Figura 1.1, a estrutura condicional possui duas checagens **if** e **fi** sendo, respectivamente, as condições de "ida"e de "volta"e que devem ser logicamente iguais no momento da suas respectivas avaliações, ou seja, caso o valor lógico de **if** seja verdadeiro, o valor de **fi** também deverá ser verdadeiro. De forma análoga se **if** for falso, **fi** também deverá ser. A estrutura de repetição **from** possui a condição de parada definida através de **until**, e possui em seu corpo a possibilidade de duas execuções definidas através do **do** e **loop**. A expressão contida no **do** será executada caso a condição do **from** seja verdadeira, enquanto que a expressão do **loop** apenas será executada enquanto a condição definida no **until** não for alcançada. Isso possibilita a criação de **do .. while**, porém não possui uma *keyword* que force uma saída antes da condição de parada ser alcançada (um **break**, por exemplo) e assim impede que *loops* sejam controlados internamente. Mas note que no caso do programa ser executado de forma reversa essas condições serão invertidas e precisam ser válidas em ambos os sentidos. As pilhas são manipuladas através das *keywords* **push** que serve para empilhar, zerando a variável passada como primeiro parâmetro e empilhando na pilha informada no segundo parâmetro, e

pop para desempilhar os inteiros, atribuindo o valor desempilhado na primeira variável e removendo-a da pilha referida. Mas fique atento que poderá ocorrer um erro de execução caso a variável contenha um valor diferente de zero na operação de **pop**. Além disso, as variáveis locais definidas dentro dos procedimentos (exceto o principal) devem possuir o sufixo **local** para serem declaradas e ao final devem ser "limpas" utilizando **delocal**. Os sub-procedimentos são chamados através de **call** e podem ser executados de forma reversa através de **uncall**, que executará o procedimento de baixo para cima e realizando as operações inversas. Por exemplo, um procedimento que incrementa uma variável possuirá na sua versão inversa um decremento.

1.2 Proposta e objetivo

O foco deste trabalho constitui no enriquecimento da linguagem de programação reversível Janus, especificamente na versão mais nova, denominada JanusP, através da inclusão e expansão de funções como aritmética modular (soma, subtração, multiplicação e exponenciação), operações sobre *bits* AND, OR, XOR e NOT, rotações binárias sobre números inteiros à esquerda (*left-rotate*) e à direita (*right-rotate*), funções matemáticas como logaritmo, fatoração, raiz quadrada, geração de números pseudo-aleatórios, cálculo de máximo divisor comum e mínimo múltiplo comum. Com isso, espera-se que o desenvolvimento dos programas futuros criados nesta linguagem seja facilitado e não apenas aumente, como também facilite, o processo de aprendizagem nessas linguagens. A partir da biblioteca proposta, denominada Kairos, os futuros programadores poderão ver com exemplos práticos como as funções são implementadas, visto que a linguagem em questão não possui um manual didático ou muitos exemplos para aprendizagem. A biblioteca implementa as principais funções ariméticas envolvendo inteiros.

Na Seção 4.2 são apresentados circuitos reversíveis semelhantes que implementam as funções do algoritmo AES. A comparação de circuitos com as funções em alto nível é feita para que o entendimento possa ser feito de maneira mais natural e as pessoas que ainda não possuíram contato com programação, porém que conhecem, ou até mesmo que preferem, circuitos não sejam excluídas desse trabalho. Isso também se justifica pela questão da comparação sobre a eficiência das funções implementadas em alto nível ser mais clara e, possivelmente, mais elucidativa. Na Seção 4.3 é feita uma comparação da eficiência da nossa implementação com outra que utiliza a mesma linguagem e foi proposta por alunos da Universidade de Copenhague.

Capítulo 2

Linguagem de programação reversível

Uma linguagem reversível é definida a partir da ideia de que as operações feitas podem ser executadas de forma inversa e, assim, voltar ao estado inicial. Isso traz muitas vantagens em termos computacionais, dentre elas, a eliminação da necessidade de sobrescrever informações. Note que para cada vez que um dado é sobrescrito, essa informação é dissipada como calor e, por este motivo, a computação clássica tende a gerar muito mais calor do que a computação reversível. Landauer [38] define o limite teórico de que para cada bit "apagado" há um gasto de, no mínimo, $kT \ln 2$ *joules*, onde k é a constante de Boltzmann [1] e T é a temperatura na qual o sistema opera. Podemos concluir disto que a eficiência energética de circuitos ainda pode ser aumentada em relação ao modelo convencional e a computação reversível se torna uma opção economicamente interessante em relação aos computadores clássicos, visto que a computação reversível não está sujeita a essas definições ou limites mínimos.

Os programas feitos em uma linguagem reversível podem ser compreendidos a partir de autômatos finitos onde as transições ocorrem através de funções determinísticas. É importante ressaltar que as funções utilizadas precisam conseguir fazer o processo em ambos os sentidos, ou seja, avançar o estado e voltar ao estado anterior sem que haja dúvidas. Podemos tomar como exemplo a operação de adição, onde a função acrescentará 2 unidades em uma determinada variável. Então, em nosso estado inicial a variável possuirá valor 0, aplicaremos a função e obteremos o resultado 2. Para voltarmos ao estado inicial, precisaremos da função inversa, que será a operação de subtração que diminuirá 2 unidades da variável. Daí, teremos a variável com o valor 2, aplicaremos a função inversa e obteremos novamente o estado inicial, ou seja, com valor 0. Pode-se perceber que mesmo com um exemplo trivial as funções precisam ser cautelosamente

[1] Constante de Boltzmann é, aproximadamente, $8.6173303 \times 10^{-5}\ eV \cdot K^{-1}$

analisadas. Algumas operações como logaritmo ou raiz quadrada, por exemplo, não são triviais e necessitam de um cuidado redobrado. Outras funções triviais como divisão, por exemplo, precisam guardar informações adicionais para serem reversíveis, mas falaremos disso mais à frente.

Grande parte da pesquisa em computação reversível é focada principalmente em circuitos e algoritmos reversíveis. Certas áreas, como física e química, foram responsáveis por este desenvolvimento devido à busca pelo entendimento de certas interações atômicas em razão do tempo. A vantagem principal de investir em linguagens reversíveis é que mais pessoas conseguem produzir de forma mais rápida em relação à descrição dos circuitos e assim gerar um avanço maior em um intervalo de tempo menor. Outra vantagem é que mais testes podem ser feitos e os custos para produzir circuitos é reduzido. Porém não foi isso que ocorreu nas últimas décadas, fazendo com que tenhamos alto conhecimento em circuitos reversíveis e conhecimento reduzido em linguagens reversíveis. Por mais que os circuitos sejam a base da computação reversível, e portanto os mais necessários para o efetivo avanço tecnológico, isso provavelmente representará um atraso considerável no entendimento, privando certos nichos que poderiam vir a contribuir com algoritmos ou outras formas de implementação mais eficientes, fazendo com que áreas correlatas como a computação quântica sejam prejudicadas indiretamente. A "democratização" em linguagens de programação pode parecer ineficiente, mas acaba se tornando uma alternativa interessante para atrair mais pessoas, ou atenção, para os problemas cotidianos.

Um dos motivos para a grande evolução e pesquisa na área de circuitos talvez se justifique devido aos especialistas da indústria, pesquisadores de universidades e cientistas de governos indicarem que a miniaturização de circuitos não continuará por muito tempo. De fato, fazer transistores menores não traz melhorias tão grandes como antigamente. As características físicas também influenciaram na estagnação da velocidade do relógio (ou frequência do *clock*) por mais de uma década, o que obrigou a indústria a produzir processadores com múltiplos núcleos. Mas isso não resolve o problema, porque ao passo que são adicionados mais núcleos, mais regiões de "silício escuro" (áreas que precisam ser desligadas a fim de evitar o superaquecimento) são criadas. Por mais que muito esforço esteja sendo feito nesse sentido, não há como mudar as leis da física, portanto em algum momento breve os computadores não serão mais rápidos, baratos ou eficientes em termos energéticos. Quando esse ponto for atingido não haverá mais progresso na tecnologia convencional e a indústria precisará investir em soluções alternativas, e inovadoras, a fim de promover o avanço tecnológico.

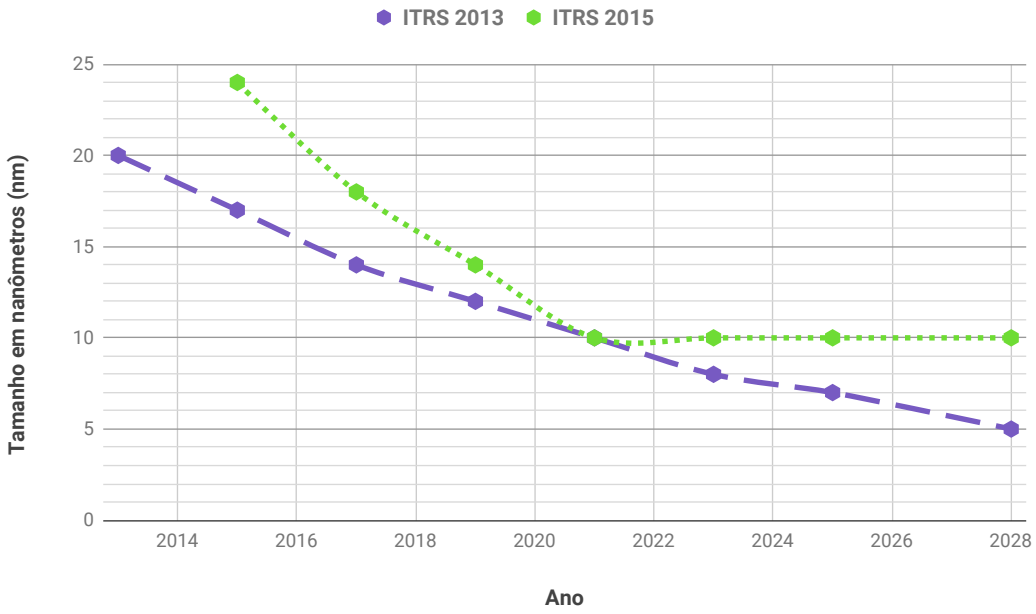

Figura 2.1: Previsão do tamanho das portas físicas dos transistores segundo os relatórios de 2013 e 2015 feitos pela ITRS.

A previsão é de que em 2021 esse ponto seja atingido, e ao que tudo indica os fabricantes irão aumentar a densidade modificando a geometria do transistor de horizontal para vertical e construir múltiplas camadas de circuitos uma sobre a outra. O ITRS (International Technology Roadmap for Semiconductors) divulgou no relatório de 2013 que o comprimento dos transistores encolheria até 2028 (linha tracejada da Figura 2.1), porém em outro mais recente, de 2015, a previsão é de que haja uma estagnação logo em 2021 (linha pontilhada da Figura 2.1), bem antes do esperado segundo a previsão anterior. No último relatório publicado pela ITRS, hoje na qual faz parte de uma iniciativa IEEE e adota o nome de Rebooting Computing, é apontado que as melhorias na computação não são mais feitas de baixo para cima, por *switches* menores e memórias mais densas ou rápidas. O relatório adota uma abordagem *top-down*, tendo como foco as aplicações em *data centers*, Internet das Coisas (ou *Internet of Things*) e dispositivos móveis que impulsionam, e ditam, o *design* dos *chips*. De fato, empresas como Apple, Google e Qualcomm vem determinando quais tecnologias deverão ser produzidas e qual será o modo de produção. Isso talvez traga benefícios, visto que os componentes serão específicos e totalmente únicos para adaptar aos padrões impostos pelas empresas. Uma outra trilha denominada IRDS (International Roadmap for Devices and Systems) também adotará essa abordagem *top-down*, porém irá incluir os assuntos referentes à arquitetura de computadores a fim de promover uma visão mais abrangente e completa do ecossistema computacional, trazendo

dispositivos, componentes, sistemas, arquiteturas e *software* à discussão.

Atualmente algumas das abordagens descritas acima sobre mudar a geometria do circuito vêm sendo produzidas. Uma aposta da indústria é que a miniaturização poderia ser substituída pela integração 3D monolítica, onde as camadas de dispositivos seriam construídas umas sobre as outras e conectadas através de uma fiação densa. A transição ocorreria de forma lenta, ou seja, tecnologias antigas ainda seriam utilizadas. A indústria de memórias NAND já se voltou para arquiteturas 3D para impulsionar a miniaturização e aumentar a capacidade do flash NAND. Porém, segundo Gargani, presidente do ITRS, é possível que algumas empresas desejem e continuem o processo de miniaturização, apesar de não ser eficiente. Além disso, o ITRS prevê que as empresas de *chips* de ponta deixarão de adotar a tecnologia FinFET e migrarão para uma semelhante que possui além das portas laterais uma que se estende para baixo também. Depois será utilizada uma abordagem vertical na qual os transistores assumirão uma forma de pilar ou nanofios dispostos em pé. Isso permitirá que a indústria continue colocando mais transistores em uma determinada área e mantendo como verdade a lei de Moore, que diz que a cada dois anos a quantidade de transistores é dobrada.

Apesar do pouco avanço em linguagens reversíveis, algum progresso foi feito quando falamos em máquina de Turing reversível [4] ou compiladores que conseguem representar e inverter instruções de forma reversível, porém, como podemos imaginar, nem todas as operações podem ser transformadas em reversíveis de maneira trivial. Uma solução trivial é adicionar um *trace* de execução, conhecido como *Landauer embedding* [38], e guardar todas as modificações feitas ao longo da execução formando um histórico que poderá ser utilizado para reverter as operações. Apesar de muito poderoso, isso é ineficiente caso a execução seja muito demorada ou possua muitas chamadas de funções. O que geralmente buscamos fazer é executar pequenas funções e revertê-las quase de forma imediata, para evitar um histórico muito grande e manter o espaço necessário para a execução controlado e pequeno, porém deve-se ficar atento se estas funções precisam ser recalculadas, ou se os passos intermediários são reutilizados, frequentemente. Caso assim seja, é preferível que todas as operações sejam calculadas e depois o processo seja revertido para evitar que haja *slow down* ou *stalling*.

Analisando certos circuitos reversíveis podemos identificar com mais clareza se esse possui todas as suas operações reversíveis. Isso acontece porque todo circuito reversível pode ser associado com uma função bijetora onde a quantidade de bits de entrada é igual a de saída e todas as saídas são distintas. Caso o circuito possua mais bits de entrada do

que de saída, alguma informação foi perdida e portanto o circuito não é reversível. Isso se aplica também às portas (*gates*) lógicas como AND, OR e XOR que transformam duas entradas em apenas uma. Apesar da operação feita na porta lógica XOR ser reversível, ou seja, ao aplicarmos XOR com uma das entradas originais podemos obter a outra entrada original, a porta em si não é reversível. Como dito no capítulo anterior, algumas alternativas surgiram para evitar que essas portas lógicas destruam a informação gerando calor. Dentre essas as mais conhecidas são as portas lógicas de Fredkin, Toffoli e CNOT, que funcionam, respectivamente, como um SWAP controlado, AND controlado e um NOT controlado.

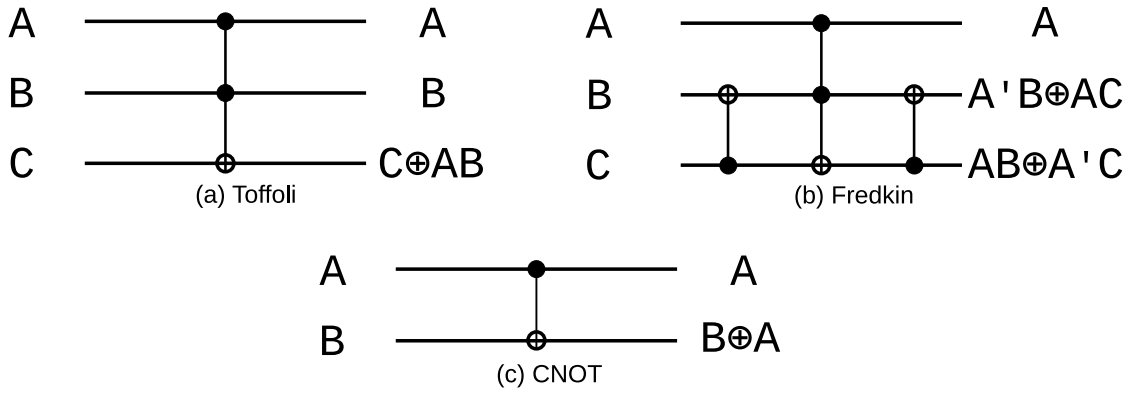

Figura 2.2: Representação das portas lógicas (a) Toffoli, (b) Fredkin e (c) CNOT.

A porta lógica CNOT possui 2 entradas e 2 saídas (veja a Figura 2.2), e pode ser entendida como um simples XOR, ou seja, caso a primeira entrada seja 0, o valor da segunda entrada não será modificado, e será aplicado um XOR na segunda entrada caso contrário. Na porta lógica de Fredkin existem 3 entradas e 3 saídas (ver Figura 2.2), sendo a primeira entrada chamada de controle. Caso o controle seja 0 o conteúdo das outras duas linhas não é modificado, ou seja, a saída é uma cópia da entrada. E no caso do controle ser igual a 1, os conteúdos das duas linhas são permutados. Com isso, não há descarte de informação, visto que a informação da entrada pode ser recuperada através da saída, e os "bits sujos" são utilizados para reverter a operação feita na porta lógica, consequentemente não dissipando calor. O processo feito na porta lógica Toffoli acontece de maneira similar, e também possui 3 entradas e 3 saídas (ver Figura 2.2) mas, caso as duas primeiras entradas possuam valor igual a 1, a terceira será modificada. Enquanto que no caso de uma das duas primeiras ser igual a 0, a cópia será feita. Essa última porta pode ser interpretada como um CNOT duplamente controlado.

2.1 Fundamentos de uma linguagem reversível

Yokohama et al. [63, 65] definem os fundamentos de uma linguagem reversível através de *updates* reversíveis, determinismo reverso, acesso às semânticas inversas e eficiência do programa reversível. De forma clara, as principais diferenças entre uma linguagem reversível e uma clássica são que todas as operações devem ser possíveis de serem revertidas, ou seja, qualquer estrutura de repetição ou condicional (ou de seleção), sequência de comandos e chamadas de métodos devem possuir suas operações respectivamente inversas.

2.1.1 Updates reversíveis

Todas as funções contidas na linguagem devem ser reversíveis e representadas com operações atômicas, e, portanto, nenhuma deve ferir as regras de reversibilidade. São apresentadas em duas definições para que isso ocorra.

Definição 1 (Funções injetivas): Uma função parcial θ: $(A \times B) \rightarrow C$, escrita como um operador binário infixo, é injetiva no primeiro argumento se, e somente se, $\forall a, a' \in A$, $\forall b \in B$, SE $a\ \theta\ b$ e $a'\ \theta\ b$ satisfazem à definição abaixo:

$$a\ \theta\ b = a'\ \theta\ b \Rightarrow a = a'$$

E para qualquer operador θ injetivo no primeiro argumento, existe um operador ϕ: $(C \times B) \rightarrow A$ tal que $\forall a \in A$, $\forall b \in B$

$$((a\ \theta\ b)\ \phi\ b) = a$$

Isso pode ser entendido de forma mais clara se pensarmos em funções como soma e OU-EXCLUSIVO, que são injetivas no primeiro argumento. Por exemplo, ao aplicarmos $a \oplus b$ e logo após aplicarmos novamente o OU-EXCLUSIVO com b, obteremos o resultado inicial a. Ou seja, operador inverso (ou função inversa) do OU-EXCLUSIVO é o próprio. O que não acontece com a soma, como descrito anteriormente, onde a função inversa é a subtração. Apesar disso, as funções de soma e subtração são injetivas, visto que possuem operadores θ e ϕ que possibilitam a reversibilidade.

$$((a \oplus b) \oplus b) = a$$

Nota-se também que a operação de multiplicação e divisão não são injetivas no primeiro argumento, visto que, ao escrevermos $n \cdot 0 = m \cdot 0$ não implica $n = m$. Para resolver esses eventuais problemas, uma outra definição é apresentada abaixo.

Definição 2 (*update* reversível): Seja uma função f: $D \rightarrow B$, um operador θ: $(A \times B) \rightarrow C$ injetivo no primeiro argumento e uma função parcial g: $(A \times D) \rightarrow (C \times D)$. Um *update* reversível será equivalente a:

$$g(x, y) = (x \ \theta \ f(y), y)$$

Apesar da função g precisar ser injetiva, a função f não necessariamente precisará ser também. Ao salvarmos o valor de y, podemos aplicar qualquer função f e recuperarmos quaisquer valores iniciais, finais ou intermediários. Assim podemos escrever o *update* reversível através do operador inverso ϕ, apresentado anteriormente, como:

$$g^{-1}(x, y) = (x \ \phi \ f(y), y)$$

Por essas razões, na grande maioria das vezes, podemos reescrever um programa irreversível através de *updates* reversíveis e fazer com que este seja representado em um programa reversível. Deve-se ressaltar ainda que isso nem sempre manterá a mesma utilização de recursos (memória, uso de CPU, espaço e tempo de execução), visto que a função inversa também precisará ser escrita e executada para reverter o processo.

2.1.2 Determinismo reverso

Além disso, uma linguagem reversível deve ser determinística em ambos os sentidos. Para garantir isso, uma propriedade é definida de forma com que nenhuma declaração de procedimento ou variável seja irreversível.

Teorema 1 (determinismo *forward* e *backward*): Seja d uma declaração na linguagem reversível Janus, por exemplo. Onde *stmt* é o *statement* e *Stores* representa o armazenamento da informação.

$$\forall \sigma, \sigma', \sigma'' \in Stores \ \sigma \vdash_{stmt} d \Rightarrow \sigma' \wedge \sigma \vdash_{stmt} d \Rightarrow \sigma'' \Rightarrow \sigma' = \sigma''$$

$$\forall \sigma, \sigma', \sigma'' \in Stores \ \sigma' \vdash_{stmt} d \Rightarrow \sigma \wedge \sigma'' \vdash_{stmt} d \Rightarrow \sigma \Rightarrow \sigma' = \sigma''$$

Podemos interpretar a expressão acima seguindo a lógica de que se a partir de σ, aplicando a mesma declaração determinística e obtendo σ' e σ'', podemos dizer que $\sigma' = \sigma''$. E ainda, de forma reversa, se a partir de σ' e σ'' obtemos σ, pode-se concluir que $\sigma' = \sigma''$. De forma geral, podemos aplicar essa ideia para qualquer linguagem reversível, mesmo que o lado direito da declaração não seja avaliada. Isso é possível pois, mesmo que os operadores não sejam injetivos, e, portanto não reversíveis, a reversibilidade ainda será garantida devido à utilização de *updates* reversíveis.

2.1.3 Acesso às semânticas inversas

Essa propriedade diz respeito a existir uma função inversa para toda, e qualquer, função declarada na linguagem ou programa. Por exemplo, ao introduzirmos a operação de soma, precisamos necessariamente introduzir a operação de subtração. Na Janus esse conceito é estendido para chamadas de funções através de *call* e "deschamadas" com *uncall*. Fazendo isso, podemos facilitar a inversão de programas e assim obter melhores resultados, já que a complexidade de inversão do programa recai sobre o programador, fazendo com que o interpretador, ou compilador, seja mais simples e eficiente.

De maneira geral e clássica, para inverter um programa precisaríamos analisá-lo por completo (globalmente), inverter todas as operações e funções, verificar os valores e parâmetros iniciais e finais e, finalmente, gerar o programa reverso. Adotando essa abordagem, a inversão do programa acontece de maneira mais simples, pois é suficiente que analisemos localmente, ou seja, se para todo operador existe um outro inverso equivalente e todas as funções são formadas a partir de operações reversíveis, podemos gerar o programa reverso de maneira direta invertendo as chamadas de funções e as respectivas operações feitas.

2.1.4 Eficiência do programa reversível

Se além disso pudermos garantir que a complexidade das funções seja igual, teremos uma eficiência máxima. Se as demais propriedades forem garantidas, a complexidade computacional do programa inverso gerado será igual, dado que as funções inversas executam a mesma quantidade de passos.

2.2 Janus

A linguagem Janus, desenvolvida por Lutz e Derby [41], surgiu a partir da curiosidade dos autores para onde os *bits* iriam após as operações feitas, já que eles não podem ser descartados de forma direta. Durante um projeto de aula no Caltech, a linguagem reversível tomou vida e foi considerada pelos autores como sem futuro. Inicialmente o compilador gerava um código interno estruturado, e que não tinha pretensões de ser robusto ou sequer utilizado para grandes aplicações. O compilador contava com quatro grandes partes: o analisador léxico, um *parser* recursivo, um interpretador para procedimentos e um *scanner*. Mais tarde, em cerca de 2007, alguns pesquisadores do departamento de Ciência da Computação da Universidade de Copenhague (DIKU), liderados por Robert Glück e Tetsuo Yokoyama, encontraram essa linguagem e resolveram aprimorá-la.

Syntax Domains	**Grammar**
$p \in$ Progs[Janus]	$p ::= d^*$ (`procedure` id s)$^+$
$x \in$ Vars[Janus]	$d ::= x \mid x$`[`c`]`
$c \in$ Cons[Janus]	$s ::= x \oplus$`=` $e \mid x$`[`e`]` $\oplus$`=` $e \mid$
$id \in$ Idens[Janus]	`if` e `then` s `else` s `fi` $e \mid$
$s \in$ Stmts[Janus]	`from` e `do` s `loop` s `until` $e \mid$
$e \in$ Exps[Janus]	`call` $id \mid$ `uncall` $id \mid$ `skip` $\mid s\ s$
$\oplus \in$ ModOps[Janus]	$e ::= c \mid x \mid x$`[`e`]` $\mid e \odot e$
$\odot \in$ Ops[Janus]	$c ::=$ `0` $\mid$ `1` $\mid \cdots \mid$ `4294967295`
	$\oplus ::=$ `+` $\mid$ `-` $\mid$ `^`
	$\odot ::= \oplus \mid$ `*` $\mid$ `/` $\mid$ `%` $\mid$ `*/` $\mid$ `&` $\mid$ `|` $\mid$ `&&` $\mid$ `||` $\mid$
	`<` $\mid$ `>` $\mid$ `=` $\mid$ `!=` $\mid$ `<=` $\mid$ `>=`

Figura 2.3: Sintaxe e gramática inicial da linguagem Janus (Janus86).

A sintaxe dessa linguagem básica, Janus86, é definida por Yokohama e Glück [65] como vista na Figura 2.3, onde existem declarações de variáveis e procedimentos. Na declaração de variável podem ser definidas variáveis do tipo inteiro (não negativos) de 32 *bits* ou vetores unidimensionais de mesmo tipo. Os vetores começam na posição zero, como a grande maioria das linguagens convencionais, e os procedimentos possuem uma sequência de caracteres que servem como identificadores para o método declarado. Dentro de cada método declarado existem *statements* que devem ser propriamente reversíveis, como visto anteriormente. Um *statement* pode ser o corpo de uma função, uma estrutura de repetição ou seleção, uma atribuição ou operação ou, de forma mais geral, uma sequência de passos a serem seguidos. Em quaisquer casos de declaração, ou atribuição, a variável à esquerda não deve aparecer do lado direito, ou seja, a operação $x = x + 1$ deve ser trocada por x += 1. Note ainda na Figura 2.3 em $\oplus$ que nem todas as operações são permitidas,

onde as permitidas são adição, subtração e XOR. Multiplicações, divisões e outras não são definidas na gramática devido a possuírem casos específicos que tornam a operação irreversível como, por exemplo, multiplicação por zero ou divisão por zero que não possui resultado definido. Além disso, ainda no caso da divisão, pode haver perda de informação, visto que os tipos primitivos são apenas inteiros e, por isso, descartam a parte decimal.

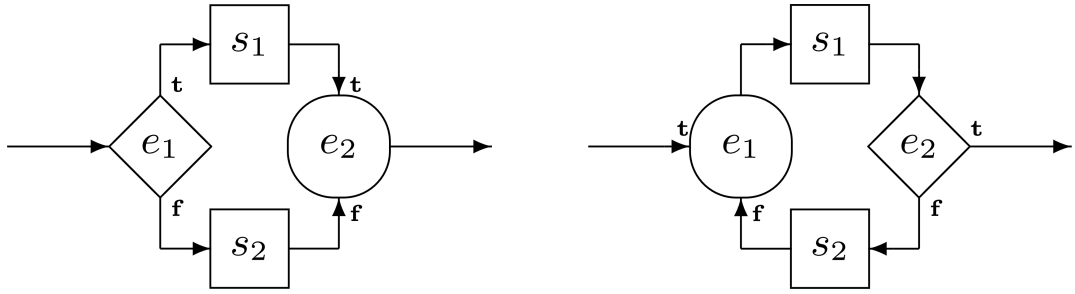

Figura 2.4: Representação do fluxo utilizado nas estruturas condicionais e de repetição.

Em relação às estruturas condicionais e de repetição, vistas na Figura 2.4, sempre haverá a checagem da expressão lógica e_1. Mas, no primeiro caso, haverá um desvio para s_1 no caso do valor lógico ser verdadeiro, ou s_2 caso contrário, terminando em e_2 realizando outra checagem e que deverá possuir o mesmo valor lógico de e_1. Já no segundo caso da estrutura de repetição, o *statement* s_1 será executado, e_2 será checado e caso seja falso o *statement* s_2 será executado, repetindo a iteração ($e_1 \rightarrow s_1 \rightarrow e_2 \rightarrow s_2$) até que e_2 seja verdade e o *loop* termine. Se e_1 for falso na estrutura de repetição, um erro será gerado.

A partir do estudo dessa linguagem, uma nova versão denominada JanusP foi desenvolvida com a colaboração destes pesquisadores em conjunto com outros pesquisadores do PIRC (Program Inversion and Reversible Computation). Assim a linguagem ganhou mais semelhanças com as linguagens modernas devido as variáveis serem locais ao invés de globais e possuir *stack* (pilhas que servem como vetores dinâmicos), porém ainda não há suporte para criação de classes, instância de objetos, ponto flutuante, *strings*, funções que retornam valores e afins. Um dos frutos do trabalho desses pesquisadores foi um compilador online para essa versão a partir da análise da linguagem JanusB (ou Janus86) por Claus Skou Nielsen e Michael Budde, e que atualmente é mantido por Michael Kirkedal Thomsen. O compilador online, para a versão JanusP, pode ser visualizado em `https://topps.diku.dk/pirc/?id=janusP`. Vale notar que todos os códigos desenvolvidos e apresentados ao longo desta tese foram feitos nesse compilador que se encontra em constante desenvolvimento.

A gramática mais nova (ver Figura 2.5) não possui muitas diferenças em relação a anterior (ver Figura 2.3). Grande parte das mudanças foram feitas na estruturação da

$$
\begin{array}{rcll}
prog & ::= & p_{main}\ p^{*} & \text{(program)} \\
t & ::= & \textbf{int} \mid \textbf{stack} & \text{(data type)} \\
p_{main} & ::= & \textbf{procedure main}\ \texttt{()}\ (\textbf{int}\ x(\texttt{[}\overline{n}\texttt{]})^{?} \mid \textbf{stack}\ x)^{*}\ s & \text{(main procedure)} \\
p & ::= & \textbf{procedure}\ q\,\texttt{(}t\ x,\ \ldots,\ t\ x\texttt{)}\ s & \text{(procedure definition)} \\
s & ::= & x\ \odot\texttt{=}\ e \mid x\texttt{[}e\texttt{]}\ \odot\texttt{=}\ e & \text{(assignment)} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s\ \textbf{fi}\ e & \text{(conditional)} \\
 & \mid & \textbf{from}\ e\ \textbf{do}\ s\ \textbf{loop}\ s\ \textbf{until}\ e & \text{(loop)} \\
 & \mid & \textbf{push}\,\texttt{(}x,\ x\texttt{)} \mid \textbf{pop}\,\texttt{(}x,\ x\texttt{)} & \text{(stack modification)} \\
 & \mid & \textbf{local}\ t\ x\ \texttt{=}\ e\quad s\quad \textbf{delocal}\ t\ x\ \texttt{=}\ e & \text{(local variable block)} \\
 & \mid & \textbf{call}\ q\,\texttt{(}x,\ \ldots,\ x\texttt{)} \mid \textbf{uncall}\ q\,\texttt{(}x,\ \ldots,\ x\texttt{)} & \text{(procedure invocation)} \\
 & \mid & \textbf{skip} \mid s\ s & \text{(statement sequence)} \\
e & ::= & \overline{n} \mid x \mid x\texttt{[}e\texttt{]} \mid e\ \otimes\ e \mid \textbf{empty}\,\texttt{(}x\texttt{)} \mid \textbf{top}\,\texttt{(}x\texttt{)} \mid \textbf{nil} & \text{(expression)} \\
\odot & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{\^{}} & \text{(operator)} \\
\otimes & ::= & \odot \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} & \text{(operator)}
\end{array}
$$

Figura 2.5: Sintaxe e gramática da versão mais nova da linguagem Janus (JanusP).

linguagem no sentido de haver menos declarações de variáveis e manter o código mais organizado, visto que agora os métodos possuem variáveis de escopo, o que acaba por facilitar a depuração de erros. Anteriormente, programas muito grandes eram complicados de serem depurados já que todas variáveis eram globais e qualquer método poderia ter interferido no resultado. A adição de vetores dinâmicos também facilitou certas aplicações, porém como são utilizadas pilhas, o tempo de acesso aos valores contidos na pilha ainda não é ideal, visto que há a necessidade de empilhar e desempilhar para realizar uma busca. Na Tabela 2.1 é feita uma comparação sucinta do que foi dito anteriormente.

Tabela 2.1: Comparação entre as linguagens JanusB e JanusP.

	JanusB	**JanusP**
Escopo de variáveis	global	local
Tipos primitivos	Inteiros de 32 bits e vetores unidimensionais	Inteiros sem precisão fixa, vetores multidimensionais e pilha
Estrutura condicional	Sim	Sim
Estrutura de repetição	Sim	Sim
Vetor dinâmico	Não	Sim (pilha)
Operadores	Soma, subtração e XOR	Soma, subtração e XOR

Capítulo 3

Biblioteca Kairos

Neste capítulo são apresentadas as funções implementadas. Elas se referem a operações com inteiros como aritmética modular, operações com sequências binárias, funções matemáticas, operação sobre vetores e geração de números pseudoaleatórios. Na seção de aritmética modular são apresentadas as operações de soma e subtração, multiplicação e exponenciação.Nas operações binárias são implementadas AND, OR, XOR, complemento a 1 e a 2, e rotações à esquerda e à direita. Em seguida, são apresentadas funções matemáticas como valor absoluto, teto, piso, fatoração, logaritmo de base qualquer, raiz quadrada e o algoritmo de Euclides estendido. Nas operações sobre vetores, são implementadas as operações básicas de cópia e rotação. Ao final a função de geração de números pseudoaleatórios é apresentada.Esse conjunto de funções foi escolhido por serem mais comuns em sistemas de criptografia simétrica e assimétrica.

3.1 Aritmética Modular

A aritmética modular se caracteriza por operações sobre os inteiros, que são agrupados em classes de equivalência. Dado um inteiro positivo n, em cada classe a diferença entre os valores é sempre um múltiplo de n. Pode-se escolher qualquer representante da classe para cada operação. Em outras palavras podemos entender que a entrada é reduzida até que essa seja menor que o valor do módulo n. A relação de congruência é introduzida de forma com que os valores "repetidos" são ditos congruentes. Veja abaixo um exemplo considerando duas entradas de números inteiros positivos $a = 12$, $b = 24$ e $n = 12$:

$$12 \equiv 24 \bmod 12$$

O que acontece no exemplo acima é que ambos são reduzidos para o mesmo valor, zero, e por este motivo são ditos congruentes módulo 12. Caso o valor do módulo seja modificado para, por exemplo, $n = 24$ teremos agora que os números não são mais congruentes, pois o primeiro resultará em 12 enquanto o segundo continuará resultando em zero.

$$12 \not\equiv 24 \bmod 24$$

De maneira geral, dizemos que os números são congruentes se a diferença entre os mesmos é múltiplo do módulo. Ou seja, considerando que a, b e n são números inteiros, temos que:

$$a \equiv b \bmod n$$

se, e somente se, existir um valor inteiro k tal que $a - b = kn$. E, consequentemente, a pode ser escrito através de b e n, como:

$$a = b + kn$$

Com essas informações podemos perceber que qualquer operação modular pode ser feita, ou desfeita, de forma reversível caso guardemos os valores de b, k e n, devido a podermos representar os valores congruentes através destas informações.

3.1.1 Adição e subtração modular

A adição modular $a+b \bmod n$ e sua equivalente denominada subtração modular $a-b \bmod n$ podem ser vistas com clareza através de exemplos cotidianos como funcionamento de um relógio de ponteiro que varia de 1 a 12, dias da semana que possuem seu inicio no domingo e terminam no sábado e outros fenômenos cíclicos. Utilizando ainda os dias da semana, temos que uma operação de soma modular nos retornaria os dias seguintes e a subtração, os anteriores. Se fizermos um mapeamento dos dias da semana para números como, por exemplo, domingo = 0, segunda-feira = 1, terça-feira = 2, ... e sábado = 6. Podemos, por exemplo, considerar que estamos em uma segunda-feira e somarmos 321 dias. Note que isso é o equivalente a fazermos $(1 + 321) \bmod 7$, ou poderíamos reduzir as somas fazendo $(1 \bmod 7 + 321 \bmod 7) \bmod 7$ que resultaria em $(1+6) \bmod 7$ e, consequentemente, obteríamos o valor 0, ou domingo. De forma análoga, poderíamos ter subtraído para

determinar que dia da semana foi 321 dias atrás e obtido terça-feira.

Essas relações são cíclicas, ou seja, possuem um ínicio e um fim, quando chegam ao fim voltam ao inicio e se repetem. Isso traz certas dificuldades para esse tipo de operação, devido a termos que lidar com a congruência. Por este motivo, uma abordagem *naïve* foi utilizada onde os valores de entrada a e b devem ser menores que n e maiores que zero. O valor resultante da operação é salvo em a enquanto as demais variáveis não sofrem modificações. A última condição $a < b$ do Procedimento 1 pode não ser clara, mas é feita dessa forma porque só devemos reduzir o valor de a caso $a + b \geq n$ e, ao fazermos isso, temos que $a < b < n$.

```
1  procedure mod_add_n(int a, int b, int n)
2      a += b
3      if a >= n then
4          a -= n
5      fi a < b
```

Procedimento 1: Adição Modular

Uma versão mais elaborada foi criada para atender aos casos nos quais o valor de a ou b seja maior que n, ou menor que zero, onde os valores de seus multiplicadores ka e kb, respectivamente, são guardados a fim de que possam ser revertidos ao valor original. Isso se justifica devido a entradas congruentes poderem ser utilizadas, e o que as diferenciará será o valor de ka e kb. O ponto negativo dessa abordagem é que todos os valores precisam ser guardados em uma pilha, caso haja múltiplas chamadas desta função, para que o programa possa ser revertido posteriormente. Note também que nesta abordagem apresentada no Procedimento 2 os valores de a e b serão modificados caso sejam menores que zero ou maiores que n, e posteriormente o valor a guardará o resultado da operação.

```
1  procedure mod_add(int a, int ka, int b, int kb, int n)
2      ka += a / n
3      kb += b / n
4
5      a -= ka * n
6      b -= kb * n
7
8      a += b
9      if a >= n then
10         a -= n
11     fi a < b
```

Procedimento 2: Adição Modular

É recomendado que o programador utilize as funções modulares simples e que não fazem uso dos parâmetros adicionais. Caso assim seja, os valores sempre irão se manter abaixo de n. Em ambos os casos, as funções irão retornar valores entre 0 e n, logo não há risco das operações serem irreversíveis devido a divisões por zero.

3.1.2 Multiplicação modular

Antes de nos aprofundarmos nesta operação modular devemos relembrar que a multiplicação possui algumas restrições, como a multiplicação por zero, assim como a divisão que é indefinida para divisões por zero. A implementação da multiplicação simples, que não é provida pela linguagem através do símbolo *=, como visto anteriormente na gramática, pode ser implementada como mostrada no Procedimento 3. O operador de SWAP é representado como <=> na linguagem Janus.

```
1  procedure mul_nswap(int a, int b)
2      local int c = 0
3
4      if b != 0 then
5          c += a * b
6          c <=> a
7          c -= a / b
8      else
9          print ("parameter *b* equals zero!")
10         // a <=> b
11     fi b != 0
12
13     delocal int c = 0
```

Procedimento 3: Multiplicação com substituição sem *bit* adicional.

Note que a implementação apresentada no Procedimento 3 falha no caso em que $a = 1$ e $b = 0$, devido a utilização da divisão para zerar o conteúdo da variável local c, para contornar esse problema é necessário fazer com que o primeiro parâmetro sempre receba o menor valor absoluto. Para isso devemos utilizar a função *abs* que retorna o valor absoluto de um determinado número e compararmos se $a < b$ e assim fazer a chamada da função $mul_nswap(a, b)$, ou $mul_nswap(b, a)$ caso contrário. Uma outra abordagem é proposta para resolver esse problema, e que ao invés de fazer a comparação de qual valor absoluto é menor, apenas informa através de um variável extra que um *swap* deveria ter ocorrido.

No Procedimento 4 o terceiro parâmetro indica que houve uma operação onde o resultado está armazenado na segunda variável b. Como a linguagem não possui o tipo

```
1  procedure mul(int a, int b, int swapped)
2      local int c = 0
3
4      if b != 0 then
5          c += a * b
6          c <=> a
7          c -= a / b
8      else
9          swapped += 1
10     fi b != 0
11
12     delocal int c = 0
```

Procedimento 4: Multiplicação com substituição e *bit swap* adicional.

booleano, um valor inteiro foi utilizado, apesar de que em termos de espaço apenas um *bit* seria necessário. Uma outra alternativa seria armazenar o resultado no menor valor dos parâmetros, porém essa técnica foi desconsiderada devido a adicionar uma complexidade a mais na chamada da função, e que também recairia sobre a primeira abordagem.

A divisão que, convencionalmente, é representada pelo símbolo /= nas linguagens de programação irreversíveis possui a sua implementação seguindo a estrutura mostrada no Procedimento 5. O operador de resto da divisão, ou módulo, é representado como % na linguagem Janus.

```
1  procedure div(int x, int y, int r)
2      local int z = 0
3
4      if y != 0 then
5          z += x / y
6          r += x % y
7          x <=> z
8          z -= (x * y) + r
9      else
10         print ("division by zero!")
11     fi y != 0
12
13     delocal int z = 0
```

Procedimento 5: Divisão tradicional reversível.

Sobre a operação de multiplicação modular, que não possui uma equivalente em contraponto da soma modular, nota-se uma grande semelhança com a operação de adição. Afinal, a multiplicação pode ser entendida como diversas somas e a divisão como sub-

trações sucessivas. Como o módulo representa o resto da divisão, não há uma operação divisão modular, pois a mesma não faria sentido. Dito isso, a operação de multiplicação modular é representada por:

$$a \cdot b \bmod n$$

Por se tratar de uma generalização da soma modular feita de formas consecutivas três implementações serão apresentadas, onde cada uma possui sua vantagem, ou melhor se adapta, em uma determinada situação. A primeira abordagem, apresentada no Procedimento 6, utiliza a operação de soma modular enunciada anteriormente em b, realiza a multiplicação (simples) entre a e b e depois efetua a soma modular utilizando o parâmetro a que foi modificado através da operação de multiplicação anterior. Note também que a função não trata o caso em que $b = 0$, que seria feito através de estruturas condicionais ou uma variável adicional.

```
1  procedure mod_mul1(int a, int ka, int b, int kb, int n)
2      local int c = 0
3      local int kc = 0
4      local int s = 0
5
6      if b != 0 then
7          call mul(a, b, s)
8          call mod_add(a, ka, c, kc, n)
9      fi b != 0
10
11     delocal int s = 0
12     delocal int kc = 0
13     delocal int c = 0
```

Procedimento 6: Multiplicação modular que reduz operandos usando adições.

A primeira abordagem, apresentada no Procedimento 6, é simples e efetiva nos casos em que o valor de a é menor que b ou em que há a utilização de poucas iterações. Essa abordagem não precisaria realizar a primeira soma modular (no operador b), porém a fim de reduzir o valor do mesmo, facilitar a multiplicação e, possivelmente, reduzir a quantidade de *bits* utilizados para armazenar as demais variáveis auxiliares, essa operação foi incluída. Os fatores multiplicativos ka e kb também são conservados para possibilitar a operação reversa e dar mais informações para o programador, se assim for necessário.

A segunda abordagem apresentada no Procedimento 7, apesar de menos eficiente, é mais direta e utiliza apenas um parâmetro adicional, em vez de dois, que serve para

guardar o resultado da operação modular. O parâmetro *swapped* se faz necessário por ser uma abordagem que não leva em consideração se $a < b$. A economia de espaço talvez seja interessante para aplicações que façam uso dessa função constantemente ou precisem do valor final da multiplicação juntamente com o resto obtido.

```
1  procedure mod_mul2(int a, int b, int r, int n, int swapped)
2      call mul(a, b, swapped)
3      if swapped = 0 then
4          call div(a, n, r)
5      else
6          call div(b, n, r)
7      fi swapped = 0
```

Procedimento 7: Multiplicação modular que reduz operandos usando divisões.

Ao realizar a operação de multiplicação, o valor de a será acumulado de forma multiplicativa com b e ao efetuar a operação de divisão o primeiro parâmetro possuirá o resultado do quociente enquanto que o terceiro parâmetro representará o resto da divisão. A partir do conhecimento do quociente e resto, juntamente como o divisor, é possível reconstruir o valor original. Além disso, há a necessidade de considerar se houve uma troca de a com b, se $b = 0$, como foi feito anteriormente no caso da multiplicação simples.

```
1  procedure mod_mul3(int a, int b, int n, stack s)
2      local int ainv = 0
3      local int m = 0
4      local int g = 0
5      local int ninv = 0
6
7      call ext_gcd(s, a, n, g, ainv, ninv)
8      m += (a * b) % n
9      b -= (ainv * m) % n
10     uncall ext_gcd(s, a, n, g, ainv, ninv)
11     b <=> m
12
13     delocal int ninv = 0
14     delocal int g = 0
15     delocal int m = 0
16     delocal int ainv = 0
```

Procedimento 8: Multiplicação modular que usa algoritmo de Euclides estendido.

A terceira opção apresentada no Procedimento 8 possui uma complexidade algorítmica maior pois faz uso do algoritmo de Euclides estendido, que será apresentado mais à frente na seção 3.3.4, e necessita da criação de uma pilha temporária. Essa restrição foi imposta

na linguagem recentemente em uma última atualização, fazendo com que essa opção seja a menos indicada.

A única vantagem dessa abordagem se dá pelo motivo que a pilha é zerada após sua execução e, portanto, não acarretará lixo ao longo da execução fazendo com que a pilha possa ser reutilizada diversas vezes. Em contraponto, a limitação imposta é que as entradas a e b devem ser menores que n. Mas note que essa vantagem também existe na primeira abordagem, caso as mesmas limitações sejam atendidas.

3.1.3 Exponenciação modular

Nos mais diversos ramos da computação a exponenciação é utilizada, assim uma operação eficiente influi diretamente na performance da aplicação, visto que essa é uma das operações aritméticas que mais demandam poder de processamento pois derivam de sucessivas multiplicações. A abordagem mais simples, e comumente feita por programadores inexperientes, é mostrada no Procedimento 9. A implementação utiliza a função de multiplicação demonstrada na Seção 3.1.2.

```
1   procedure pow(int base, int exponent, int r)
2       local int i = 0
3
4       if base != 0 then
5           if exponent < 0 then
6               print ("exponent must be a positive integer.")
7           else
8               r += 1
9               from i = 0
10              do skip
11              loop
12                  call mul_nswap(r, base)
13                  i += 1
14              until i >= exponent
15              i -= exponent
16          fi exponent < 0
17      fi base != 0
18
19      delocal int i = 0
```

Procedimento 9: Exponenciação simples que usa a multiplicação sem *swap*.

Para implementar a exponenciação modular, poderíamos utilizar a exponenciação entre inteiros e calcular o módulo do resultado obtido. Mas é mais eficiente utilizar a

multiplicação modular para isso. A grande vantagem em relação a abordagem anterior é a quantidade de tempo gasto para realizar as multiplicações, visto que os números são reduzidos a cada iteração. Essa operação é matematicamente escrita como:

$$a^b \bmod n$$

A implementação proposta foi dividida, por questão de espaço na página, nos Procedimentos 10 e 11, e utiliza a segunda abordagem da função de multiplicação modular (apresentada no Procedimento 7) devido a sua generalização e eficiência, além disso também evita que o controle seja feito internamente e gere complicações no entendimento ou clareza do código. Essa abordagem pode ser substituída por outra sem grandes problemas, caso o programador assim deseje. Mas vale notar que deverão ser feitas checagens com o objetivo de verificar se a multiplicação é "segura", ou seja, b não é igual a zero.

```
1  procedure mod_pow(stack s, int base, int exponent, int n, int res)
2      local int i = exponent
3      local int b = 0
4      local int r = 0
5      local int swap = 0
6      local int rtemp = 0
7
8      if n = 0 then
9          print ("division by zero!")
10     fi n = 0
11
12     if n > 1 && exponent >= 0 then
13         rtemp += 1
14         from i = exponent
15         do
16             if i % 2 = 1 then
17                 call mod_mul2(rtemp, base, r, n, swap)
18                 push(swap, s)
19                 rtemp <=> r
20                 push(r, s)
21             fi i % 2 = 1
22
23             call div(i, 2, r)
24             push(r, s)
```

Procedimento 10: Exponenciação modular que usa a multiplicação com *swap*.

Ao longo da execução desta função, os valores da base, expoente e módulo são utilizados como referência e, portanto, não são modificados. A pilha que guarda os passos

```
25                b += base
26                call mul(base, b, swap)
27                push(b, s)
28                push(swap, s)
29
30                call div(base, n, r)
31                base <=> r
32                push(r, s)
33            until i = 0
34
35            res += rtemp
36
37            from i = 0
38            do
39                pop(r, s)
40                base <=> r
41                uncall div(base, n, r)
42                pop(swap, s)
43                pop(b, s)
44                uncall mul(base, b, swap)
45                b -= base
46
47                pop(r, s)
48                uncall div(i, 2, r)
49
50                if i % 2 = 1 then
51                    pop(r, s)
52                    rtemp <=> r
53                    pop(swap, s)
54                    uncall mod_mul2(rtemp, base, r, n, swap)
55                fi i % 2 = 1
56            until i = exponent
57            rtemp -= 1
58        else
59            print("modulus and exponent must be greater than zero.")
60        fi n > 1 && exponent >= 0
61
62        delocal int rtemp = 0
63        delocal int swap = 0
64        delocal int r = 0
65        delocal int b = 0
66        delocal int i = exponent
```

Procedimento 11: Continuação do procedimento de exponenciação modular.

necessários para reverter a operação precisa ser declarada no corpo principal do programa devido a uma limitação da linguagem. Mas note que essa prática não acontecia anterior-

mente, pois havia a possibilidade de inicializar a *stack* dentro de qualquer procedimento e desalocar os recursos utilizados caso a pilha estivesse vazia ao final da execução.

3.2 Operações Binárias

Nesta seção serão descritas as operações lógicas de conjunção (AND), disjunção (OR), disjunção exclusiva (XOR) e complemento. Assim como as operações de rotação sobre bits denominadas de rotação à esquerda (left-rotate) e rotação à direita (right-rotate), ou deslocamento circular à esquerda e à direita, respectivamente.

As operações AND e OR são irreversíveis caso não guardem os resultados, devido a não serem funções injetivas. Algo que não acontece com as operações XOR e NOT, que podem ser aplicadas diretamente sem modificação, devido à função inversa ser ela própria. No caso das rotações, não há *carry* e nenhuma informação é perdida porque os bits são apenas reorganizados. Se a implementação utilizasse *shifts* não-circulares teríamos perda de informação, como acontece com as funções *built-in* de deslocamento na Janus.

3.2.1 Operação AND

A conjunção lógica é definida como a interseção de conjuntos na matemática ou o produto binário entre dois bits na computação. A grande maioria dos programas possui essa função básica que serve para avaliar expressões lógicas em estruturas condicionais. Se precisarmos avaliar uma expressão lógica podemos fazê-la de imediato, porém se quisermos armazenar o resultado, não poderemos guardá-lo sem um aumento no espaço utilizado. Isso acontece porque a porta lógica AND não possui inversa, e portanto, para revertemos precisaremos dos valores iniciais e finais.

Tabela 3.1: Tabela-verdade para a operação de conjunção lógica (AND).

a	**b**	$\boldsymbol{a \wedge b}$
1	1	1
1	0	0
0	1	0
0	0	0

Além disso, podemos perceber através da tabela-verdade Tabela 3.1 que em três casos os valores são iguais, o que impossibilita o trabalho de inversão se apenas um dos valores iniciais for conhecido. Isso reforça a ideia de que os valores iniciais precisam ser guardados.

Uma implementação *naïve* é mostrada no Procedimento 12, onde os parâmetros de entrada a e b, assim como o parâmetro de saída c são armazenados. Na chamada da função a operação $c += a \ \& \ b$ será executada, enquanto que para revertermos esse valor precisaremos apenas zerar a variável c fazendo $c -= a \ \& \ b$.

```
1  procedure and(int a, int b, int c)
2      c += a & b
```

Procedimento 12: Implementação da operação AND.

3.2.2 Operação OR

A disjunção lógica é definida como a união de conjuntos na matemática ou a soma binária subtraída do produto entre dois bits na computação. De forma análoga à operação de conjunção lógica, a grande maioria dos programas a utiliza e a porta lógica OR não possui uma inversa direta. Portanto, assim como fizemos anteriormente, iremos guardar os valores de entrada e saída para que possamos, eventualmente, voltar ao estado inicial.

A tabela-verdade Tabela 3.2 possui três casos nos quais os valores são iguais, o que impossibilita o trabalho de inversão se conhecermos apenas um dos valores iniciais.

Tabela 3.2: Tabela-verdade para a operação de disjunção lógica (OR).

a	**b**	$\boldsymbol{a \vee b}$
1	1	1
1	0	1
0	1	1
0	0	0

A implementação *naïve* dessa operação é mostrada no Procedimento 13, onde os parâmetros de entrada a e b e o parâmetro de saída c são armazenados. Na chamada da função a operação $c += a \mid b$ será executada, enquanto que para revertermos esse valor precisaremos apenas zerar a variável c fazendo $c -= a \mid b$.

```
1  procedure or(int a, int b, int c)
2      c += a | b
```

Procedimento 13: Implementação da operação OR.

3.2.3 Operação XOR

Na disjunção exclusiva podemos fazer uma interpretação a cerca das operações como uma soma módulo 2 na computação e na matemática como a união menos a interseção. Como os operadores são binários essa operação é feita de forma trivial e inversível devido a podermos sobrescrever um dos valores e recuperá-lo utilizando a outra entrada. Ou seja, o espaço utilizado se mantém constante ao longo de sua execução.

Analisando a tabela-verdade Tabela 3.3 podemos notar que existem dois casos. O primeiro que acontece quando os bits são iguais e resultam em 0, e o outro quando os bits são diferentes e o resultado é igual a 1. Com isso, podemos recuperar o valor sabendo apenas o resultado e um dos operandos.

Tabela 3.3: Tabela-verdade para a operação de disjunção exclusiva (XOR).

a	**b**	$\boldsymbol{a \oplus b}$
1	1	0
1	0	1
0	1	1
0	0	0

Na linguagem Janus existe essa operação *built-in*, portanto não há grande necessidade de utilizar a função proposta. Porém, caso o programador se sinta mais confortável, ou queira organizar o código utilizando apenas chamadas de funções, o Procedimento 14 pode se mostrar uma alternativa interessante.

```
1  procedure xor(int a, int b)
2      a ^= b
```

Procedimento 14: Implementação da operação XOR.

3.2.4 Operação de complemento a um e a dois

O complemento, por se tratar de uma operação unária, também será reversível de maneira trivial, visto que a inversa será a própria. Essa operação pode ser implementada de duas formas: considerando os bits de sinal no inteiro (*signed int*) ou não (*unsigned int*). A operação que será utilizada fica a cargo do programador, assim como garantir que os valores utilizados possuam, ou não, sinal. A linguagem JanusP não possui quaisquer limitações, ou restrições, sobre valores dos inteiros, possibilitando que existam tanto valores negativos quanto positivos.

```
1  procedure ones_complement(int x)
2      local int y = 0
3      y += -1 - x
4      x <=> y
5      delocal int y = -1 - x
```

Procedimento 15: Implementação de complemento a um.

Apesar da operação de complemento a um não ser muito popular, esta consiste em inverter todos os bits de um número, em sua representação binária, a fim de que os números positivos sejam convertidos em negativos e vice-versa. A sua adoção não foi muito popular devido a certos problemas como a representação do valor zero que resulta em zero positivo ou negativo, manipulações no *carry* e afins. Contudo sua implementação, considerando inteiros com sinal, é apresentada no Procedimento 15.

```
1  procedure twos_complement(int x, int bits)
2      local int m = 1 << bits
3      local int y = 0
4      y += m - x
5      x <=> y
6      delocal int y = m - x
7      delocal int m = 1 << bits
```

Procedimento 16: Implementação de complemento a dois sem sinal.

A operação de complemento a 2, que consiste em inverter todos os bits e depois realizar um incremento possui sua implementação dependente da quantidade de bits que serão utilizados para representar o número, no caso do bit de sinal ser desconsiderado. A implementação proposta pode ser visualizada no Procedimento 16. O operador $<<$ representa a exponenciação feita utilizando um deslocamento à esquerda.

```
1  procedure twos_complement_signal(int x)
2      local int y = 0
3      y -= x
4      x <=> y
5      delocal int y = 0 - x
```

Procedimento 17: Implementação de complemento a dois com sinal.

Caso seja necessário considerar o bit de sinal, a implementação fica ainda mais simples. Isso se deve ao fato de que a própria Janus irá lidar com os bits de sinal, então o cálculo, na verdade, ficará como $y = 0 - x = -x$. Além disso a quantidade de bits para representar

os valores também não é necessária, visto que a linguagem utiliza, por padrão, precisão arbitrária em suas operações. A implementação pode ser visualizada no Procedimento 17.

3.2.5 Left-Rotate e Right-Rotate

As operações de rotação são similares às operações de deslocamento de bits encontradas em diversas linguagens de programação clássica, geralmente representadas por $<<$ para deslocamentos à esquerda e à direita utiliza-se a notação $>>$. Quando aplicadas em números binários, os deslocamentos funcionam como multiplicação (à esquerda) e divisão (à direita), porém há perda de informação nesses casos. Na divisão, por exemplo, as perdas ocorrem devido aos bits menos significativos serem descartados e serem introduzidos 0s nos bits mais significativos (ver Figura 3.1). De forma análoga, a operação de deslocamento para a esquerda poderá sofrer perda de informação no *bit* mais significativo caso alguma limitação de tamanho ao número resultante seja imposta.

Figura 3.1: Representação da operação de deslocamento a direita. O *bit* menos significativo é perdido e um novo *bit "0"* é adicionado na posição do *bit* mais significativo.

Para evitar isso, são utilizados deslocamentos circulares que fazem com que os bits menos significativos se tornem bits mais significativos, no caso da divisão (ver Figura 3.2). Neste caso não há perda de informação, visto que os *bits* são apenas reposicionados sem nenhum acréscimo, ou remoção, de informação ao conteúdo original. A rotação à esquerda também ocorre sem nenhum problema, visto que o reposicionamento irá ocorrer na ordem inversa, ou seja, os *bits* serão reposicionados em um unidade para a esquerda, enquanto que o *bit* mais significativo se tornará o *bit* menos significativo.

Figura 3.2: Representação da operação de rotação a direita. O *bit* menos significativo se torna o *bit* mais significativo e todos os demais são reposicionados uma unidade à direita.

A implementação das rotações supracitadas, à esquerda e direita, são demonstradas abaixo. O parâmetro d é a entrada do número inteiro a ser rotacionado, n se refere à quantidade de *bits* que o número possui e m representa a quantidade de rotações que serão realizadas.

Na primeira parte do Procedimento 18, o cálculo do *bit* mais significativo é calculado em t_0, enquanto s_0 representa o valor de significância respectivo. As demais partes, t_1 e

```
1  procedure left_rotate(int d, int n, int m)
2      local int t0 = 0
3      local int t1 = 0
4      local int t2 = 0
5      local int t3 = 0
6      local int s0 = 0
7
8      t0 += d / (1 << n)
9      s0 += t0 * (1 << n)
10
11     t1 += (d - s0) / (1 << (n - m))
12     t2 += d - s0 - (t1 * (1 << (n - m)))
13
14     t3 += s0 + (t2 * (1 << m)) + t1
15
16     d -= t2
17     d -= t1 * (1 << (n - m))
18     d -= s0
19     d <=> t3
20
21     t2 -= (d - s0 - t1) / (1 << m)
22     t1 -= d % (1 << m)
23
24     s0 -= t0 * (1 << n)
25     t0 -= d / (1 << n)
26
27     delocal int s0 = 0
28     delocal int t3 = 0
29     delocal int t2 = 0
30     delocal int t1 = 0
31     delocal int t0 = 0
```

Procedimento 18: Implementação proposta para a rotação à esquerda.

t_2, representam a parte que será reposicionada em suas posições inicial e final, respectivamente. Ao final é calculado t_3 que representa a rotação feita utilizando s_0, t_1 e t_2 com seus valores de significância. A entrada d é zerada utilizando as informações que compõe t_3 e a realização de um *swap* entre d e t_3 ocorre a fim de manter o espaço constante.

```
1  procedure right_rotate(int d, int n, int m)
2      uncall left_rotate(d, n, m)
```

Procedimento 19: Implementação proposta para a rotação à direita.

Note que a função de rotação a direita, apresentada no Procedimento 19, fica simples de ser implementada devido as rotações serem simétricas. O programador poderá utilizar

as funções através de *calls* e *uncalls* sem grandes problemas, porém o compilador deverá otimizá-las, ou seja, trocar *uncall right_rotate()* por um *call left_rotate()*, por exemplo.

3.3 Funções matemáticas

Nessa seção são descritas as operações matemáticas mais complexas como raiz quadrada, fatoração, teto, piso, valor absoluto e logaritmo. Além disso, o algoritmo de Euclides estendido é apresentado a fim de complementar o conteúdo de aritmética modular, pois o mesmo pode ser utilizado para determinar o inverso multiplicativo modular e calcular o máximo divisor comum, simultaneamente.

3.3.1 Função de fatoração

Essa função tem como propósito decompor um número natural em seus fatores primos. Os fatores podem se repetir e são limitados pelo tamanho do vetor de entrada $fact[]$. É recomendável que quanto maior o número seja, maior seja o tamanho do vetor de entrada, visto que é provável que esse tenha mais fatores (repetidos ou não).

```
 1  procedure fat(int num, int fact[])
 2      local int try = 0
 3      local int i = 0
 4      local int z = 0
 5
 6      if num = 1 then
 7          i += 1
 8          fact[i] += 1
 9      else
10          from (try = 0) && (num > 1)
11          loop
12              call next_try(try)
13              from fact[i] != try
14              loop
15                  i += 1
16                  fact[i] += try
17                  z += num / try
18                  z <=> num
19                  z -= num * try
20              until (num % try) != 0
21          until (try * try) > num
```

Procedimento 20: Função de fatoração adaptada da proposta de Lutz e Derby [41].

```
22          if num != 1 then
23              i += 1
24              fact[i] <=> num
25          else
26              num -= 1
27          fi fact[i] != fact[i - 1]
28
29          if (fact[i - 1] * fact[i - 1]) < fact[i] then
30              from (try * try) > fact[i]
31              do skip
32              loop
33                  uncall next_try(try)
34              until try = 0
35          else
36              try -= fact[i - 1]
37          fi (fact[i - 1] * fact[i - 1]) < fact[i]
38      fi num = 1
39
40      call zeroi(i, fact)
41
42      delocal int z = 0
43      delocal int i = 0
44      delocal int try = 0
45
46  procedure zeroi(int i, int fact[])
47      from fact[i + 1] = 0
48      do skip
49      loop
50          i -= 1
51      until i = 0
52
53  procedure next_try(int try)
54      try += 2
55      if try = 4 then
56          try -= 1
57      fi try = 3
```

Procedimento 21: Continuação da função de fatoração proposta.

O método de fatoração proposto e implementado por Lutz e Derby [41] para a Janus86 (ou JanusB) foi adaptado para a JanusP, a linguagem utilizada ao longo desse trabalho, e pode ser visualizada de forma completa nos Procedimentos 20 e 21.

3.3.2 Função de raiz quadrada

Outra função que não existe *built-in* e é extremamente básica é a função de raiz quadrada. Apesar de Lutz e Derby [41] terem demonstrado essa função em seu *paper* inicial, ainda hoje a função não foi incluída na linguagem, talvez pela falta de suporte a módulos ou importações. No Procedimento 22 é apresentada a adaptação feita para JanusP baseando-se no código da JanusB escrito por Lutz e Derby.

```
1  procedure sqrt(int num, int root)
2      local int bit = 1
3
4      from bit = 1
5      do skip
6      loop
7          call double_bit(bit)
8      until (bit * bit) > num
9      from (bit * bit) > num
10     do
11         uncall double_bit(bit)
12         if ((root + bit) * (root + bit)) <= num then
13             root += bit
14         fi ((root / bit) % 2) != 0
15     until bit = 1
16     num -= root * root
17
18     delocal int bit = 1
19
20 procedure double_bit(int b)
21     local int z = b
22     b += z
23     z -= b / 2
24     delocal int z = 0
```

Procedimento 22: Função de raiz quadrada adaptada a partir de Lutz e Derby.

3.3.3 Função de logaritmo base N

A função genérica de logaritmo, ou base N, dá a oportunidade de outras bases, como 8 ou 16, serem utilizadas sem grandes complicações. Algo relativamente importante nos casos em que a estrutura utilizada necessita utilizar uma base diferente para realizar operações. Note também que na implementação proposta não há a possibilidade de obter logaritmos fracionários ou inferiores a zero, visto que a linguagem não possui ponto flutuante. A implementação desta função pode ser visualizada no Procedimento 23.

```
1  procedure logn(int x, int n, int bits, stack s)
2      local int i = x
3      local int p = 0
4
5      if x <= 0 then
6          print ("undefined.")
7      else
8          from i = x do
9              p += i / n
10             i <=> p
11             push(p, s)
12         loop
13             bits += 1
14         until i < 1
15
16         from i < 1 do
17             pop(p, s)
18             i <=> p
19             p -= i / n
20         until i = x
21     fi x <= 0
22
23     delocal int p = 0
24     delocal int i = x
```

Procedimento 23: Função de logaritmo genérica, base n, proposta.

Mas outras possibilidades existem como, por exemplo, se quisermos descobrir a quantidade de bits que seriam necessários para alocar um determinado número inteiro, ou então, determinar a quantidade de dígitos que um número inteiro possui. Esses dois exemplos podem ser calculados através da utilização função genérica de logaritmo supracitada e sua implementação visualizada no Procedimento 24.

```
1  procedure log2(int x, int bits, stack s)
2      call logn(x, 2, bits, s)
3
4  procedure log10(int x, int bits, stack s)
5      call logn(x, 10, bits, s)
```

Procedimento 24: Função de logaritmo base 2 e 10 utilizando a função genérica.

3.3.4 Algoritmo de Euclides estendido

Em sua versão mais simples o algoritmo de Euclides baseia-se na ideia de que dois números quaisquer a e b podem ser expressos através de um fator multiplicativo em comum g, o máximo divisor comum (MDC), e que esse fator é irredutível para qualquer relação entre a e b. Dado que conhecemos o valor de g, pode-se afirmar que o máximo divisor comum entre a e $b-a$, até que $a = b$, sempre se manterá o mesmo. Por ser um algoritmo simples que realiza consecutivas divisões até que o resto seja igual a zero, este consegue convergir para o máximo divisor comum em poucas operações. A partir da ideia de que a diferença entre os dois números irá resultar no MDC, podemos realizar as seguintes operações de divisão e usar o resto da divisão para agilizar o processo. Fazendo trocas a cada iteração do valor de a por b e o de b por a mod b, teremos para $a = 42$ e $b = 9$:

$$\begin{array}{rcrcr} \text{mdc}(42,\,9) & = & a \,/\, b & , & a \bmod b \\ & & 42 \,/\, 9 = 4 & , & 42 \bmod 9 = 6 \\ & & 9 \,/\, 6 = 1 & , & 9 \bmod 6 = 3 \\ & & 6 \,/\, 3 = 2 & , & 6 \bmod 3 = 0 \\ & & 3 & & \end{array}$$

O resultado do $mdc(42{,}9) = 3$ se encontra na última linha, ou na penúltima quando o resto é igual a zero, informando que o máximo divisor comum foi encontrado. Mas note que o algoritmo utiliza as divisões e seus restos para executar o próximo passo; após a iteração esse conteúdo é descartado. Na versão estendida do algoritmo essas informações serão aproveitadas para determinar os coeficientes da identidade de Bézout e, assim, poderemos escrever o MDC da seguinte forma:

$$mdc(a, b) = a \times \alpha + b \times \beta$$

A estratégia utilizada é começar com os valores a e b como sendo o resto da divisão e os coeficientes com 1 e 0. Recomenda-se utilizar $a > b$, mas isso não irá interferir no resultado, apenas acarretará mais uma iteração. Seguindo o mesmo exemplo acima, verifique como poderíamos montar a tabela para o algoritmo de Euclides estendido.

As duas primeiras linhas dizem respeito à inicialização do algoritmo, onde os coeficientes de a e b devem ser invertidos. Na linha 3, ou iteração 2, faremos a mesma divisão que anteriormente, porém iremos calcular o coeficiente α_i utilizando os valores das duas iterações anteriores fazendo $\alpha_i = \alpha_{i-2} - q_i \times \alpha_{i-1}$. De forma análoga iremos calcular o coeficiente de b fazendo $\beta_i = \beta_{i-2} - q_i \times \beta_{i-1}$. A última linha apresenta os valores de a e b divididos pelo maior fator multiplicativo em comum. O máximo divisor comum é infor-

iteração	quociente	resto	α	β
0	–	42	1	0
1	–	9	0	1
2	$42/9 = 4$	$42 \bmod 9 = 6$	$1 - 4 \times 0 = 1$	$0 - 4 \times 1 = -4$
3	$9/6 = 1$	$9 \bmod 6 = 3$	$0 - 1 \times 1 = -1$	$1 - 1 \times -4 = 5$
4	$6/3 = 2$	$6 \bmod 3 = 0$	$1 - 2 \times -1 = 3$	$-4 - 2 \times 5 = -14$

mado na penúltima linha, ou iteração 3, na coluna do resto. Enquanto que os coeficientes são informados na mesma linha, porém nas colunas dos coeficientes α e β. Dessa forma, podemos verificar que:

$$mdc(42, 9) = 42 \times -1 + 9 \times 5 = 3$$

A implementação desse algoritmo foi feita utilizando uma pilha para guardar os valores anteriores de forma que ao final da execução deste método a pilha seja zerada. Ao menos 3 iterações são envolvidas no processo, por isso se justifica o uso de 3 conjuntos de variáveis representando o resto r, e os coeficientes sa e tb. A implementação proposta é apresentada nos Procedimentos 25, 26 e 27 devido ao seu tamanho.

```
1  // gcd(a, b) = sa * a + tb * b
2  procedure ext_gcd(int a, int b, int gcd, int sa, int tb, stack s)
3      local int r1 = 0
4      local int s1 = 1
5      local int t1 = 0
6
7      local int r2 = 0
8      local int s2 = 0
9      local int t2 = 1
10
11     local int q3 = 0
12     local int r3 = 0
13     local int s3 = 0
14     local int t3 = 0
15
16     if a = 0 || b = 0 then
17         print ("*a* and *b* must be different than zero.")
18     else
19         r1 += a
20         r2 += b
```

Procedimento 25: Implementação proposta do algoritmo de Euclides estendido.

Inicialmente os valores de entrada a e b são copiados para os valores de resto r_1 e r_2, como vimos anteriormente. Logo após essa operação, veja as linhas 22 e 23 no Procedimento 26, o valor do quociente é calculado e armazenado em q_3, assim como o resto em r_3. No *loop* os valores de s_3 e t_3 serão calculados a fim de determinar os valores previamente apresentados como α e β, respectivamente. A cada iteração os valores dos restos r_1, r_2 e r_3, do quociente q_3, valores de α (s_1, s_2 e s_3) e β (t_1, t_2 e t_3) sofrem uma modificação copiando os valores das duas últimas iterações. Por exemplo, nas linhas 28 a 32 do Procedimento 26, são zerados os valores dos restos e uma troca ocorre entre r_1 e r_2, e depois entre r_2 e r_3, resultando no último zerado. Quando r_3 for igual a zero, encontramos o máximo divisor comum e os valores dos coeficientes α e β. O mesmo procedimento de trocas também ocorre para os valores intermediários de α e β. Vale ressaltar que q_3 é zerado a cada iteração, porém os valores intermediários não são necessários.

```
21          from r2 = b do
22              q3 += r1 / r2
23              r3 += r1 - q3 * r2
24          loop
25              s3 += s1 - q3 * s2
26              t3 += t1 - q3 * t2
27
28              push(r1, s)             // r1 = 0
29              r1 += r2                // r1 += r2
30              push(r2, s)             // r2 = 0
31              r2 += r3                // r2 += r3
32              push(r3, s)             // r3 = 0
33              push(q3, s)
34              push(s1, s)
35              s1 += s2
36              push(s2, s)
37              s2 += s3
38              push(s3, s)
39              push(t1, s)
40              t1 += t2
41              push(t2, s)
42              t2 += t3
43              push(t3, s)
44          until r3 = 0
45
46          gcd += r2
47          sa += s2
48          sb += t2
```

Procedimento 26: Parte 2 da implementação proposta do Euclides estendido.

```
49          from r3 = 0 do
50              r3 -= r1 - q3 * r2
51              q3 -= r1 / r2
52          loop
53              pop(t3, s)
54              t2 -= t3
55              pop(t2, s)
56              t1 -= t2
57              pop(t1, s)
58              pop(s3, s)
59              s2 -= s3
60              pop(s2, s)
61              s1 -= s2
62              pop(s1, s)
63              pop(q3, s)
64              pop(r3, s)
65              r2 -= r3
66              pop(r2, s)
67              r1 -= r2
68              pop(r1, s)
69              t3 -= t1 - q3 * t2
70              s3 -= s1 - q3 * s2
71          until r2 = b
72          r1 -= a
73          r2 -= b
74      fi a = 0 || b = 0
75
76      delocal int t3 = 0
77      delocal int s3 = 0
78      delocal int r3 = 0
79      delocal int q3 = 0
80
81      delocal int t2 = 1
82      delocal int s2 = 0
83      delocal int r2 = 0
84
85      delocal int t1 = 0
86      delocal int s1 = 1
87      delocal int r1 = 0
```

Procedimento 27: Parte 3 da implementação proposta do Euclides estendido.

3.3.5 Função de valor absoluto

Em alguns casos é necessário determinar o valor de um número desconsiderando o sinal; para isso deve-se utilizar a função de valor absoluto. Em muitos casos essa função, que

geralmente possui assinatura **abs(x)**, acompanha as mais diversas linguagens de programação em suas bibliotecas matemáticas. Nas linguagens clássicas a função possui apenas um parâmetro e retorna o valor absoluto respectivo, mas como a linguagem Janus não possui suporte a retorno, acrescentaremos o retorno em um parâmetro adicional. A implementação proposta é apresentada no Procedimento 28.

```
1 procedure abs(int a, int b)
2     if a < 0 then
3         b -= a
4     else
5         b += a
6     fi a < 0
```

Procedimento 28: Função de valor absoluto proposta.

3.3.6 Função teto (ceil)

A função teto tem como objetivo retornar o maior valor possível dada uma divisão. Por exemplo, ao dividirmos 5 por 2, obteremos 2,5. Porém, ao aplicarmos a função teto, espera-se obter 3, visto que esse é o valor arredondado para cima. Como a linguagem Janus apenas possui inteiros, essa abordagem de divisão é necessária para que os valores não sejam perdidos.

Uma implementação proposta para resolver esse problema é descrita no Procedimento 29. Assim como na divisão, se faz necessário utilizar um parâmetro de saída c para a operação se tornar reversível.

```
1 procedure ceil(int a, int b, int c)
2     c += a / b
3
4     if a % b != 0 then
5         c += 1
6     fi a % b != 0
```

Procedimento 29: Função matemática teto proposta.

3.3.7 Função piso (floor)

A função piso, que retorna o inteiro igual ou o menor mais próximo do resultado da divisão de a por b, ou seja realiza o arredondamento para baixo, é implementada diretamente

pois a linguagem automaticamente remove qualquer dígito decimal. Logo, a abordagem utilizada se traduz no Procedimento 30.

```
1 procedure floor(int a, int b, int c)
2     c += a / b
```

Procedimento 30: Função matemática piso proposta.

3.4 Operações sobre vetores (arrays)

Atualmente nas principais linguagens de programação existem estruturas denominadas vetores, ou *arrays* caso consideremos a nomenclatura em inglês, que tem como objetivo prover um agrupamento de tipos primitivos. Na Janus, como existe apenas o tipo primitivo *integer*, os vetores são constituídos apenas por inteiros, porém isso não impede que algumas operações sobre vetores sejam implementadas a fim de auxiliar o programador através de uma interface mais intuitiva. Assim, foram propostas as funções de cópia e rotações à esquerda e direita. As rotações funcionam de forma similar, porém mais simples que as vistas anteriormente.

3.4.1 Cópia de vetores

Essa função tem como objetivo duplicar a informação de um vetor em outro, seja por motivos de guardar a informação, manter um estado, ou realizar alguma outra operação posteriormente. A implementação proposta considera que o vetor que irá receber a informação possui, no mínimo, o tamanho do vetor de entrada.

O algoritmo inicialmente verifica se a cópia é possível checando se o tamanho de entrada é menor ou igual ao tamanho de saída, depois realiza de fato a cópia. O procedimento é bastante simples devido a sua baixa complexidade. A operação de soma poderia ser substituída pela operação de XOR, porém essas operações, geralmente quando feitas nas arquiteturas atuais, utilizam a mesma quantidade de ciclos, não havendo assim ganho em eficiência. A escolha da operação de soma se justifica ainda pela legibilidade e clareza no código.

```
1  procedure copy_array(int in[], int out[])
2      local int in_size = size(in)
3      local int out_size = size(out)
4      local int i = 0
5      if in_size <= out_size then
6          from i = 0
7          do
8              out[i] += in[i]
9          loop
10             i += 1
11         until i = in_size - 1
12     fi in_size <= out_size
13     delocal int i = in_size - 1
14     delocal int out_size = size(out)
15     delocal int in_size = size(in)
```

Procedimento 31: Função de cópia de vetores.

3.4.2 Rotação para esquerda e direita

Assim como a rotação de *bits* descrita anteriormente na seção 3.2.5, a operação também se aplica a vetores. No caso de vetores, menos trabalho precisa ser feito, devido as posições já estarem pré-definidas em memória, e podemos remanejar os dados efetuando simples trocas. As implementações propostas de rotação à esquerda e direita podem ser visualizadas, respectivamente, nos Procedimentos 32 e 33.

```
1  procedure rotate_left_array(int a[])
2      local int s = size(a)
3      local int i = 0
4
5      if s > 1 then
6          from i = 0
7          do
8              a[i] <=> a[i + 1]
9          loop
10             i += 1
11         until i = s - 2
12
13         i -= s - 2
14     fi s > 1
15
16     delocal int i = 0
17     delocal int s = size(a)
```

Procedimento 32: Função de rotação para esquerda sobre vetores.

A versão inversa, rotação à direita, também pode utilizar a função de rotação à esquerda utilizando *uncall*. Portanto, a função de rotação irá reverter uma rotação no sentido contrário. Essa abordagem é vista no Procedimento 33.

```
1  procedure rotate_right_array(int a[])
2      uncall rotate_left_array(a)
```

Procedimento 33: Função de rotação à direita em vetores.

Isso significa que ao fazermos a chamada *uncall* usando a função de rotação à direita, na verdade estaremos fazendo uma rotação à esquerda. Utilizando uma frase bastante conhecida pode-se entender como "o verso do inverso é o próprio verso".

3.5 Pseudorandom number generator (PRNG)

Diversos algoritmos foram propostos para geração de números pseudoaleatórios ao longo do tempo, dentre os mais conhecidos e comuns nas linguagens, temos: Linear Congruential Generator (LCG) [54], Linear-feedback shift register (LFSR) [58] e *Mersenne Twister* (MT) [44]. Alguns outros também bastante conhecidos derivam desses e possuem uma qualidade igual ou superior, como os derivados de LCG denominados Inversive congruential generator, Permuted congruential generator [47] e Wichmann–Hill, e os que se baseiam na ideia do LFSR como Xorshift e Xoroshiro128+. Para medir a qualidade e eficiência destes algoritmos ou abordagens, diversos testes também foram criados a fim de avaliar a segurança embutida, visto que quase todos algoritmos criptográficos de chave pública dependem de geração de números pseudoaleatórios. Os testes mais conhecidos são o Diehard [43] e TestU01 [39].

Os LCGs são os mais comuns ainda hoje devido a sua grande facilidade de implementação e baixo custo computacional, porém não são seguros sob a ótica de criptografia e escolhas ruins podem levar a péssimos resultados [20]. O algoritmo é suficiente no caso de aplicações comuns em que não há uma grande necessidade de segurança, ou de forma mais clara, nas quais os valores gerados não são determinantes para "quebrar" o sistema. Caso contrário, algumas análises devem ser feitas acerca dos números gerados, do tamanho da sequência e de quantos *bits* devem ser descartados da LSB (*Least Significant Bit*), ou de forma mais simples dos bits menos significativos, para que exista uma distribuição equiprovável. É derivado de um dos primeiros algoritmos para geração de número pseudoaleatórios denominado Lehmer Random Number Generator [40, 48] e que realiza

a seguinte multiplicação modular, onde n é o módulo, g é o multiplicador, X_0 é o valor inicial da sequência e X_k é o k-ésimo elemento gerado na sequência:

$$X_{k+1} = g \cdot X_k \mod n.$$

O caso generalizado, conhecido como LCG, adiciona uma constante c que serve como incremento na equação. Portanto, o caso descrito acima, Lehmer RNG (ou *Multiplicative Congruential Generator*), ocorrerá quando $c = 0$. O outro caso denominado *Mixed Congruential Generator* acontecerá quando $c \neq 0$ e serve para adicionar, visualmente, um grau de aleatoriedade à sequência gerada. É recomendado [48] que, g seja uma raiz primitiva (ou elemento gerador) e o valor inicial X_0 seja coprimo com n.

$$X_{k+1} = (g \cdot X_k + c) \mod n$$

Para que o período da sequência seja n é necessário que a escolha dos parâmetros g, c e n seja feita seguindo o teorema de Hull-Dobell [33], que diz:

1. n e c devem ser coprimos, ou seja, o máximo divisor comum entre eles é igual a 1.
2. $g - 1$ deve ser divisível por todos os fatores de n.
3. $g - 1$ é divisível por 4 se n é divisível por 4.

Além disso é recomendado pelos autores do teorema acima que o valor de n seja potência de 2 em máquinas binárias, ou potência de 10 em máquinas decimais, a fim de facilitar a operação de módulo que possui divisões de forma implícita. De forma mais clara, o teorema diz que se n for potência de 2, em máquinas binárias, teremos c ímpar e $g \equiv 1 \pmod 4$. Enquanto que em máquinas decimais, teremos n como potência de 10, c não divisível por 2 ou 5, e $g \equiv \pmod{20}$. A escolha mais comum para o parâmetro c é 1, devido a existir, no mínimo, uma combinação equivalente que irá gerar a mesma sequência [36]. A reversibilidade presente nessa abordagem permite que o inverso modular de g, chamado aqui de g_{inv}, seja utilizado para cálculo dos números anteriores, fazendo:

$$X_k = g_{inv} \cdot (X_{k+1} - c) \mod n$$

A segunda abordagem para geração de números pseudoaleatórios denominada LFSR utiliza um polinômio gerado a partir de uma relação binária sequencial e recursiva. As relações binárias são geradas utilizando uma função $f(x)$ primitiva sobre GF(2) e a partir

de uma entrada de n-tuplas da forma $(c_1, c_2, c_3, \cdots, c_n)$, onde c possui valor 0 ou 1 e c_n necessariamente é igual a 1. Essa relação, juntamente com a n-tupla, resulta em um polinômio de grau n primitivo sobre GF(2) e que possui um tamanho de sequência (ou período) máximo igual a $2^n - 1$.

$$f(x) = 1 + c_1x + c_2x^2 + c_3x^3 + \cdots + c_{n-1}x^{n-1} + x^n$$

Dessa forma, a formação de um conjunto de *taps*, que são as posições c_k da n-tupla e que possuem valor igual a 1, influencia na saída gerada pela função $f(x)$ e determina, consequentemente, a sequência na qual os números serão gerados e o tamanho da mesma. As condições necessárias para obter o tamanho de sequência máxima são:

1. O polinômio $f(x)$ deve ser primitivo, ou seja, irredutível e tal que o menor m para o qual $f(x)$ divide $x^m - 1$ é $m = p^n - 1$, onde n é o grau do polinômio.

2. A quantidade de *taps*deve ser par.

3. As *taps* devem ser coprimos dois-a-dois, ou seja, o MDC entre as *taps* é igual a 1.

A última condição diz, de forma mais clara, que todas as *taps* não devem possuir fatores em comum. Por exemplo, a *tap* [2, 4, 7] fere a terceira condição, porém a *tap* [4, 7, 9] atende à condição imposta apesar da primeira e da última *tap* não ser um número primo. A validação desse requisito pode ser feita utilizando o algoritmo de máximo divisor comum usando como parâmetro as posições c_k e verificando se o resultado é igual a 1. O tamanho da sequência máxima S_{max} gerada quando estes requisitos são atendidos podem ser calculados a partir da quantidade de bits m utilizados na representação do polinômio da seguinte forma:

$$S_{max} = 2^m - 1$$

Apesar do espaço utilizado nesse algoritmo ser um pouco superior ao anterior, LCG, o grande benefício é que podemos derivar outra sequência dada uma sequência que atende a esses requisitos supracitados. Isso pode ser entendido devido as sequências poderem ser espelhadas, visto que podemos "ler" o polinômio em qualquer direção. Portanto uma sequência que possui 20 bits poderia utilizar as *taps* escritas como [20, 17, 0] e ser equivalente a uma outra, também de 20 bits, que possui as *taps* [20, 3, 0]. Logo, uma função reversível poderia utilizar as *taps* para gerar a sequência em qualquer direção tendo conhecimento das *taps* iniciais e calcular a sequência inversa respectiva fazendo

$[m, m-k_1, m-k_2, ..., m-k_i]$, onde i representa a quantidade de *taps*. Portanto a geração de um número pseudoaleatório depende apenas do estado inicial *state* e da posição das *taps*. Uma implementação em C, seguindo o exemplo do polinômio $x^{20} + x^{17} + 1$, pode ser compreendida como:

```
1 int gen_next_random(int state)
2 {
3     int bit  = (state ^ (state >> 3)) & 1;
4     state = (state >> 1) | (bit << 19);
5     return state;
6 }
```

Procedimento 34: Implementação em C para o exemplo da *tap* [20, 17, 0].

A abordagem feita pelo algoritmo MT possui como vantagens o tamanho do período gerado, de $2^{19937} - 1$ para o *MT-19937*, e algumas otimizações são capazes de superar a velocidade na geração de números até mesmo que implementações feitas em *hardware* [55]. Porém, em contraponto, requer uma quantidade de memória superior e falha em alguns casos de teste [47]. Além disso, a aleatoriedade proveniente da inicialização da matriz de estados não é suficiente, ou seja, a sequência gerada será muito semelhante caso os estados iniciais sejam similares. Por estes motivos esse algoritmo será descartado como possível implementação nesta biblioteca que visa operações eficientes em termos de espaço e quantidade de operações necessárias para cumprir o objetivo. A não implementação desta abordagem se justifica ainda pelo fato de que caso o usuário fosse utilizar tal função, esse precisaria informar como parâmetro o vetor de inicialização, algo relativamente trabalhoso para obter um ganho de desempenho ínfimo. Uma alternativa para contornar essa situação seria utilizar uma função que derivaria os valores utilizados na matriz a partir de uma entrada, mas que provavelmente recairia sobre o problema inicial da aleatoriedade.

Em relação a abordagem feita pela família de algoritmos Xorshift e Xoroshiro, apesar de ser eficiente em espaço e quantidade de operações, há o descarte das informações ao longo de sua execução. Como as informações são sobrescritas, e não há como recuperá-las, essa opção de implementação também será descartada. Note que poderíamos guardar os valores intermediários, caso houvesse uma modificação no algoritmo, porém tal mudança acabaria por impactar no espaço utilizado.

Um estudo recente [26] mostra que alguns dos algoritmos supracitados passam nos testes propostos pelo NIST (National Institute of Standards and Technology), apesar de não serem seguros para aplicações criptográficas, e possuem uma boa qualidade. Enquanto que em outros testes como TestU01 que substituiu o antigo Diehard, e também

o DieHarder [8], por razões de ampliar o escopo de testes e permitir que outras análises independentes fossem feitas, reprovam alguns desses. Por exemplo, apesar de utilizado em linguagens como Python [50] e C++ [10], o sofisticado Mersenne Twister é reprovado como mostra O'Neill [47], enquanto que o *naïve* LCG ainda consegue obter bom desempenho e passar nos testes, caso a quantidade de *bits* seja maior que 88 *bits* e sejam utilizados 32 *bits* para representar os números gerados. Essa quantidade de *bits* é relativa à bateria de testes utilizada, ou seja, um conjunto de testes mais exigente poderá precisar descartar mais *bits* a fim de eliminar a correlação existente entre os números gerados. O MT-19937, por exemplo, apresenta uma falha de previsibilidade a partir de 624 amostras, onde é possível determinar seu estado interno e prever a sequência de números pseudoaleatórios.

Dito isso, a abordagem escolhida é baseada no LCG por sua simplicidade e eficiência, assim como sua capacidade de reversibilidade devido a ser possível calcularmos o inverso modular de n. A implementação utilizada possui valores fixos, mas que podem ser modificados caso assim o usuário deseje. Primeiramente será apresentada a versão com valores fixos denominada **rand(x)** e depois a versão que aceita parâmetros variados **random(x, a, i, n)**, onde o parâmetro i se refere ao inverso multiplicativo modular de a e que pode ser encontrado utilizando o algoritmo de Euclides estendido apresentado na seção 3.3.4. Além disso, uma versão genérica que adota o nome de **randx(x, a, ainv, n)**, e que pode ser visualizada no Procedimento 35, foi feita de forma que parte do código pode ser reutilizado por ambas as funções.

```
1  procedure randx(int x, int a, int ainv, int n)
2      local int c = 1
3      local int t = 0
4
5      // nextx = (a * x + c) mod n
6      t += (a * x + c) % n
7
8      //prevx = (ainverse * (x - c)) mod n
9      x -= ainv * (t - c) % n
10     x <=> t
11
12     delocal int t = 0
13     delocal int c = 1
```

Procedimento 35: Implementação da função LCG reversível *randx*.

A implementação da função mais simples **rand(x)**, visualizada no Procedimento 36, recebe como parâmetro o valor x [1], que servirá para gerar os números seguintes, e utiliza

[1] o primeiro valor, denominado *seed*, deve ser escolhido pelo usuário de forma arbitrária.

os demais valores pré-fixados. A cada chamada da função o valor de x será substituído pelo próximo da sequência sem perda de informação, economizando assim espaço e facilitando seu uso. Os valores escolhidos tendem a maximizar o período gerado e seguem as informações obtidas no relatório de L'Ecuyer [42], que realizou um estudo com o objetivo de verificar quais os melhores parâmetros de a dado o valor de n.

```
1  procedure rand(int x)
2      local int a = 3935559000370003845
3      local int n = 18446744073709551616
4      local int ainv = -4746243923404857011
5
6      call randx(x, a, ainv, n)
7
8      delocal int ainv = -4746243923404857011
9      delocal int n = 18446744073709551616
10     delocal int a = 3935559000370003845
```

Procedimento 36: Parâmetros propostos por L'Ecuyer [42] para $n = 2^{64}$.

A versão onde o usuário pode escolher os valores desejados, se assim possibilitarem a geração dos números pseudoaleatórios, é apresentada no Procedimento 37. Note que os parâmetros escolhidos devem satisfazer, no mínimo, a condição do inverso modular multiplicativo. As demais condições, que possuem como objetivo verificar se o número irá gerar o período máximo, não são verificadas a fim de dar mais liberdade para o usuário.

```
1  procedure random(int x, int a, int i, int n)
2      local int test = (a * i) % n
3
4      if test = 1 then
5          call randx(x, a, i, n)
6      else
7          print ("*i* is not modular multiplicative inverse of *a*.")
8      fi test = 1
9
10     delocal int test = (a * i) % n
```

Procedimento 37: Implementação da função genérica *random*.

Vale ressaltar que o valor de c não é modificado em nenhuma das funções, e possui o valor constante de 1, pois como dito anteriormente, não influi no tamanho da sequência gerada nem modifica qualquer outro aspecto como eficiência, segurança ou reversibilidade.

Capítulo 4

Implementações criptográficas

Neste capítulo são descritos os resultados obtidos utilizando algumas funções da biblioteca proposta assim como a implementação de dois algoritmos de chave simétrica: AES e Cha-Cha. O primeiro e bastante conhecido, até mesmo por quem não possui alto conhecimento em criptografia, possui a sua implementação feita na JanusP adotando uma postura de minimizar as operações, e os custos respectivos, e, quando não possível, tentar equilibrar espaço e complexidade algorítmica. Na segunda implementação a abordagem de manter o espaço de entrada e saída constante foi utilizado, assim como também minimizar a quantidade de operações feitas.

Apesar de ambos algoritmos serem classificados como criptografia de chave simétrica, são usados para finalidades diferentes, devido a diferenças em termos de complexidade, espaço utilizado e reversibilidade. Mas antes precisamos entender o que é criptografia e quais são os tipos de criptografia mais básicos existentes. Analisando a morfologia da palavra criptografia temos em sua origem grega *kryptós* que significa escondida e *graphein* sendo escrita, portanto podemos concluir que criptografia tem seu significado como "escrita escondida". Mas, como podemos imaginar, existem diversas formas de comunicação, sendo a escrita apenas uma delas, logo a palavra criptografia possui um sentido muito mais amplo que confere o sentido, nos dias atuais, de "comunicação escondida". Ao longo do tempo uma variedade de sistemas criptográficos baseados em problemas matemáticos, perda de informação e textos gerados a partir de uma chave secreta foram idealizados. Os que possuem perda de informação (e não são reversíveis) adotaram o nome de funções de dispersão (ou em inglês, *hash function*), os que usam segredos distintos para cifrar e decifrar possuem o nome de criptografia de chave pública (ou chave assimétrica), e o último, que será o nosso foco neste capítulo, a criptografia de chave simétrica que foi construída a partir da ideia de um segredo mútuo compartilhado.

Falando primeiramente sobre as funções *hash*, que perdem informação, a ideia central consiste de uma entrada E e uma saída S que é mapeada através de uma função irreversível que dissipa a informação gerando como saída um resumo criptográfico da entrada. Na computação convencional esse tipo de criptografia serve, dentre as inúmeras possibilidades, para armazenar senhas em bancos de dados, prover uma assinatura digital e verificar a integridade de dados. Mas note que outras aplicações como, por exemplo, gerar uma sequência pseudoaleatória, contadores de integridade e encontrar duplicidade de arquivos também são válidas. Vale ressaltar que não há qualquer garantia de que essas funções irão funcionar em todos os casos, pois devido a possuírem uma saída (possivelmente) menor do que a entrada acabam sendo geradas colisões nos resumos obtidos. Ou seja, a função para duas entradas E_1 e E_2 distintas pode ter a mesma saída S, o que é chamado de colisão. Esse fato é fácil de ser entendido pensando que se temos um domínio infinito e um contra-domínio finito, logo haverá alguma repetição em um certo momento. Quando as colisões destas funções *hash* começam a aparecer, ou surgem algoritmos capazes de encontrar colisões em um curto espaço de tempo, outros sistemas criptográficos são propostos, validados e recomendados por serem mais seguros sob a ótica de criptografia.

A respeito dos outros dois tipos de criptografia, temos uma abordagem que utiliza a mesma chave para esconder (cifrar) e recuperar (decifrar) os dados, e outra que utiliza duas chaves distintas para realizar as mesmas operações porém que dependem do objetivo desejado. A criptografia que possui duas chaves é denominada de criptografia assimétrica, e a que utiliza a mesma chave é reconhecida por criptografia simétrica.

A criptografia de chave simétrica adota operações computacionais de difusão e confusão para esconder os dados e, geralmente, possui diversas rodadas em que são repetidas essas operações com o objetivo de embaralhar e dificultar a recuperação da informação. Na criptografia assimétrica os algoritmos são baseados em problemas matemáticos que possuem uma função matemática simples para calcular quando todos os dados de entrada são conhecidos e extremamente difícil caso contrário. Como, por exemplo, a função de logaritmo que possui o problema do logaritmo discreto onde é fácil calcularmos o valor de $K = \log_b a \mod N$ porém, quando possuímos conhecimento apenas de K, b e N, o cálculo de a se torna computacionalmente difícil. O leitor pode pensar que é um problema relativamente simples devido a podermos montar uma lista com todos os números possíveis $[0..N)$ e depois realizar uma varredura até que o valor de K seja encontrado, porém note que a complexidade é linear em relação a b e exponencial em relação a N. Nenhum algoritmo clássico consegue resolver esse problema de forma eficiente, porém algoritmos quânticos como Shor conseguem encontrar essas soluções em tempo polinomial. Em rela-

ção às duas chaves geradas, há uma divisão onde uma pode ser distribuída amplamente chamada de chave pública e outra que deve ser mantida em segredo e recebe o nome de chave privada. Dependendo da ocasião, a encriptação utiliza a chave privada ou a chave pública. Caso o objetivo seja enviar uma informação de maneira segura pela rede (por exemplo, uma chave secreta para utilização de uma criptografia de chave simétrica), será utilizada a chave pública do destino para encriptar os dados. Como apenas o destino conseguirá recuperar a informação utilizando a sua chave privada, podemos dizer que a informação foi decodificada unicamente pelo destino. Mas podemos também ter o caso em que queremos garantir que uma informação seja distribuída para todos interessados e que estes possam recuperar a informação inicial a partir de uma informação pública (chave pública da origem). Além disso, a garantia de que aquela informação foi enviada pela origem ocorre pois a chave privada é utilizada para encriptar a mensagem enviada. De maneira geral, podemos dizer que, se todos devem possuir acesso a informação, será utilizada a chave privada da origem para esconder os dados e a chave pública da origem para recuperá-los. E no caso de apenas o destino precisar conhecer o conteúdo da informação, a chave pública desse será utilizada e, quando a mensagem for recebida, o mesmo irá utilizar a sua chave privada para recuperar a informação.

4.1 Introdução a chaves simétricas

Como dito anteriormente a criptografia que utiliza apenas uma chave para encriptar e decriptar os dados é denominada de criptografia de chave simétrica. O leitor pode se perguntar se essa escolha possui um impacto negativo em relação à criptografia assimétrica, visto que a última faz uso de duas chaves e portanto deve ser mais segura. Porém note que a segurança dos procedimentos está intrinsecamente ligada a problemas matemáticos, alguns dos quais podem ser resolvidos eficientemente por computadores quânticos. Vale notar que não há qualquer interferência negativa nos sistemas de criptografia simétrica (seguros), seja por computadores quânticos, e seus algoritmos, ou por ataques em computadores clássicos.

Grande parte dos sistemas computacionais atuais, assim como a Internet, fazem uso de tais criptografias para transmissão de dados entre agentes desconhecidos. A comunicação é feita de forma segura pois devido a chave secreta entre os participantes ser combinada anteriormente através de dispositivos *offline* (pendrives, disco rígido externos e afins) ou *online* utilizando criptografia de chave assimétrica. Esse segredo compartilhado pode ser combinado usando, por exemplo, o sistema criptográfico Diffie-Hellman que realiza uma

troca de mensagens através de chave públicas e privadas, combinando o segredo entre as partes sem que nenhuma das partes possua o segredo previamente. Assim, mesmo que um intruso esteja visualizando as trocas de mensagens, de ambos os lados, não será capaz de descobrir o segredo compartilhado. Dito isso, podemos perceber que uma das maiores desvantagens da criptografia de chave simétrica é a escolha da chave secreta. Porém, em contra-partida, após ambas as partes entrarem em consenso são muito mais rápidas e usam menos recursos computacionais para serem executadas. Além de serem flexíveis em relação ao modo de encriptação, que pode ocorrer em blocos, que é útil quando a mensagem está completa, ou em fluxo que aprimora a segurança nos casos em que apenas parte da mensagem é conhecida.

Considerando que os computadores estão ficando cada vez mais rápidos e eficientes, é natural que esses possam realizar mais operações por segundo, e consequentemente é recomendado que os intervalos de tempo entre as trocas de segredo sejam reduzidos. A constante troca da chave compartilhada se justifica devido a ser possível realizar um ataque analisando uma grande quantidade de mensagens em certos criptossistemas. Mas essa constante modificação nas chaves não é algo restrito dessa abordagem de apenas uma chave. As chaves públicas e privadas também passam por processo semelhante, a cada período de 6 a 12 meses, onde são geradas novas e as antigas são descartadas. Assim também acontece com certificados digitais que possuem a recomendação de serem revogados anualmente. Vale ressaltar que essas recomendações se aplicam a sistemas *online*. Sistemas não conectados com o meio externo, como por exemplo, um sistema de *backup* local não necessariamente devem seguir essas diretrizes, e sim priorizar senhas "seguras" para longo prazo.

Na última década surgiram alguns processadores que possuíam otimizações para o algoritmo AES, devido a ser um dos mais amplamente difundidos, onde um conjunto de instruções denominado AES-NI [21] foi um dos precursores desse tipo. Porém com o surgimento, e enaltecimento, de sistemas criptográficos seguros baseados em operações ARX (Add-Rotate-Xor), como o Salsa e ChaCha, fizeram com que essas abordagens de otimização não possuam grande diferencial, visto que a segurança provida sob a ótica de criptografia é semelhante. Isso acontece porque uma arquitetura otimizada para executar o algoritmo AES acaba possuindo o mesmo desempenho [11, 14], em ciclos por byte, que uma não otimizada para o ChaCha. Apesar disso, empresas como Intel e AMD ainda investem em processadores que beneficiem o AES. Talvez por questões de tecnologias legadas que ainda fazem uso do AES, ou simplesmente por razão da forte utilização desse algoritmo em ampla escala por fatores de resistência a ataques. Se olharmos sob

o ponto de visto de ataques, podemos perceber que o AES foi testado por muito mais tempo que o ChaCha. Porém devemos levar em consideração também que a utilização de recursos computacionais, eficiência algorítmica e velocidade de encriptação [13] beneficiam o ChaCha. Também vale ressaltar que a utilização do ChaCha20-Poly1305 vem crescendo enquanto o ECDSA-AES-GCM-SHA256 decai [12]. Isto ocorre, em grande parte, devido a adoção do ChaCha20 no TLS em grandes serviços como Google [27], SSH e devido a um desejo de padronização [34] dos protocolos utilizados.

4.2 Caso 1 - AES

Nessa seção é descrito o algoritmo do AES e sua respectiva implementação na linguagem JanusP, assim como uma comparação com o circuito reversível proposto por Grassl et al. [28]. O circuito proposto apresenta um bom método de comparação devido a possuir como objetivo a minimização de recursos computacionais para realizar a encriptação. A comparação é feita através de um paralelo entre baixo e alto nível com os circuitos idealizados por Kowada [37] e Vedral [61] que apresentam todas as operações básicas necessárias. As operações mais complexas são quebradas em operações mais simples para que a comparação ainda assim seja possível. Vale ressaltar que a implementação não tem como objetivo superar a versão em circuitos, mas sim avaliar a eficiência da mesma.

4.2.1 Algoritmo do AES

O algoritmo Rjindael [15], mais conhecido como AES (Advanced Encryption Standard) por ter ganho a competição feita pelo NIST (National Institute of Standards and Technology), foi idealizado por dois criptógrafos belgas, Vincent Rijmen e Joan Daemen, em 1998. De fato, o AES é um caso particular do algoritmo idealizado, já que o Rjindael não possui o tamanho de blocos fixo em 128 *bits*. A competição elaborada pelo NIST tinha como objetivo encontrar um algoritmo para substituir o padrão anterior denominado DES (Data Encryption Standard), que havia sido "quebrado" pela máquina DES Cracker construído pela EFF (Electronic Frontier Foundation), e proteger os arquivos do governo estadounidense de forma segura. O processo de escolha do novo padrão criptográfico durou 5 anos e diversos projetos concorrentes foram apresentados, dentre eles os mais famosos foram: Serpent, Twofish, MARS e RC6. Os três primeiros ofereciam uma maior segurança, porém eram mais complexos de serem implementados em arquiteturas diversas e, nos casos do Serpent e Twofish128, apresentavam uma velocidade de execução menor. O

outro algoritmo finalista, RC6, que possui como um dos idealizadores Rivest [1] foi descartado devido a quantidade de votos negativos na competição e por possuir uma qualidade inferior aos demais, porém que ainda apresentava uma boa estruturação e resistência a ataques, e que também é derivado do seu antecessor RC5.

Como dito anteriormente, algumas empresas recentemente também estão direcionando esforços para a otimização do algoritmo Rjindael, fazendo com que os demais concorrentes apresentem ainda piores resultados quando comparados em termos de velocidade. Sabendo destes esforços algumas pessoas mal intencionadas investem na indústria de resgate de dados digitais usando *ransomwares*, *softwares* que criptografam os dados contidos nos computadores e pedem uma recompensa em Bitcoins para liberar os arquivos originais, enquanto que outras preferem utilizar tecnologias de encriptação simétrica, como Twofish e Serpent, aliadas com outras de *secure deletion* para dificultar o trabalho de recuperação de dados por parte de terceiros.

A definição do algoritmo AES consiste de diversos passos para gerar difusão da informação e confusão dificultando que um agente externo saiba diferenciar uma entrada real de outra gerada de forma aleatória. Além disso, algumas tabelas contendo constantes criptográficas são utilizadas, como: SBOX e RCON. A Tabela SBOX 4.1, ou tabela de substituição, possui uma inversa conhecida como Tabela SBOX-Inv 4.2 que serve para encriptar e decriptar os dados, respectivamente. A Tabela RCON 4.3, ou tabela de constantes de *round*, armazena as constantes que serão usadas a cada iteração do algoritmo. Ambas as tabelas, SBOX e RCON, possuem o objetivo de criar confusão ao longo da execução dos procedimentos SubBytes e AddRoundKey. A técnica de difusão é introduzida pelos procedimentos ShiftRows e MixColumns que, respectivamente, embaralham as linhas e colunas dos vetores que contêm os dados. O procedimento MixColumns acaba também gerando confusão devido a realizar operações sobre o corpo finito de Galois que resultam na substituição, de forma reversível, dos dados.

O procedimento SubBytes, que utiliza a Tabela SBOX 4.1, e em sua forma reversa a Tabela SBOX-Inv 4.2, realiza duas operações para que a substituição dos dados seja feita. Primeiramente uma modificação através do inverso multiplicativo no corpo finito de Galois $GF(2^8)$ é aplicada ao valor inicial e logo após uma transformação Afim é feita no resultado anterior. Esse procedimento tem como objetivo adicionar uma resistência a ataques algébricos simples. O resultado dessas operações é mapeado na Tabela 4.1, onde são mostradas as substituições necessárias para cada *byte* da matriz de entrada e

[1] Um dos criadores do famoso algoritmo RSA de chave pública ainda utilizado hoje em dia.

Tabela 4.1: Tabela de substituição SBOX utilizada no procedimento SubBytes.

	0	1	2	3	4	5	6	7	8	9	*A*	*B*	*C*	*D*	*E*	*F*
00	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
10	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
20	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
30	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
40	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
50	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
60	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
70	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
80	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
90	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A0	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B0	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C0	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D0	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E0	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F0	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

seu resultado equivalente, onde o dígito hexadecimal mais significativo indica a linha e o menos significativo indica a coluna. De forma mais clara podemos considerar a entrada hexadecimal 42 e, usando a tabela referida anteriormente, chegarmos ao resultado $2C$.

Tabela 4.2: Tabela SBOX-Inv utilizada no procedimento inverso da SubBytes.

	0	1	2	3	4	5	6	7	8	9	*A*	*B*	*C*	*D*	*E*	*F*
00	52	09	6A	D5	30	36	A5	38	BF	40	A3	9E	81	F3	D7	FB
10	7C	E3	39	82	9B	2F	FF	87	34	8E	43	44	C4	DE	E9	CB
20	54	7B	94	32	A6	C2	23	3D	EE	4C	95	0B	42	FA	C3	4E
30	08	2E	A1	66	28	D9	24	B2	76	5B	A2	49	6D	8B	D1	25
40	72	F8	F6	64	86	68	98	16	D4	A4	5C	CC	5D	65	B6	92
50	6C	70	48	50	FD	ED	B9	DA	5E	15	46	57	A7	8D	9D	84
60	90	D8	AB	00	8C	BC	D3	0A	F7	E4	58	05	B8	B3	45	06
70	D0	2C	1E	8F	CA	3F	0F	02	C1	AF	BD	03	01	13	8A	6B
80	3A	91	11	41	4F	67	DC	EA	97	F2	CF	CE	F0	B4	E6	73
90	96	AC	74	22	E7	AD	35	85	E2	F9	37	E8	1C	75	DF	6E
A0	47	F1	1A	71	1D	29	C5	89	6F	B7	62	0E	AA	18	BE	1B
B0	FC	56	3E	4B	C6	D2	79	20	9A	DB	C0	FE	78	CD	5A	F4
C0	1F	DD	A8	33	88	07	C7	31	B1	12	10	59	27	80	EC	5F
D0	60	51	7F	A9	19	B5	4A	0D	2D	E5	7A	9F	93	C9	9C	EF
E0	A0	E0	3B	4D	AE	2A	F5	B0	C8	EB	BB	3C	83	53	99	61
F0	17	2B	04	7E	BA	77	D6	26	E1	69	14	63	55	21	0C	7D

De maneira análoga a Tabela SBOX-Inv 4.2 realiza as substituições usando como entrada o resultado encontrado acima e tem como objetivo recuperar o valor original.

Continuando o exemplo anterior, o resultado encontrado foi $2C$, o qual seria revertido olhando a posição referente na Tabela 4.2, que nos retornaria 42 novamente. Note porém que os valores precisam ser interpretados utilizando sua representação hexadecimal.

Tabela 4.3: Tabela de constantes utilizada no procedimento AddRoundKey.

0x8d	**0x01**	**0x02**	**0x04**	**0x08**	**0x10**	**0x20**	**0x40**
0x80	**0x1b**	**0x36**	**0x6c**	**0xd8**	**0xab**	0x4d	0x9a

A Tabela RCON 4.3, ou tabela de constantes de *round*, não é completamente utilizada na versão atual do AES e depende da quantidade de iterações que serão feitas, ou seja, quanto maior a quantidade de expansões feitas mais posições desta tabela serão utilizadas. Como as versões aprovadas pelo NIST são de 128, 192 e 256 *bits* e com o tamanho de bloco fixo em 128 *bits*, a quantidade máxima de posições que serão utilizadas na tabela é de 13, que é a quantidade máxima de rodadas decrescida de uma unidade. Os valores contidos nessa tabela também são gerados a partir de uma operação sobre o corpo finito de Galois $GF(2^8)$ e os que são utilizados ao longo da execução do algoritmo padrão podem ser visualizados em negrito na Tabela 4.3. A tabela completa é demonstrada no Apêndice A.1.

A expansão de chaves utiliza como entrada o tamanho da chave que está sendo utilizada e que pode possuir 128, 192 ou 256 *bits*. Nessa fase a chave inicial é expandida para um conjunto de chaves de rodada que contém, respectivamente, 176, 208 ou 240 *bits*. Podemos interpretar a chave expandida como um vetor de inteiros de 4 *bytes*, ou 32 *bits*, de forma que seja calculada da seguinte forma:

Passo 1. As 4 primeiras colunas da *expanded key* serão compostas pela própria chave de entrada.

Passo 2. Enquanto a quantidade de *bytes* da *expanded key* for menor do que a quantidade necessária de $(rodadas + 1) \times 16$, faremos os ciclos abaixo para expandir a chave, onde a quantidade de rodadas necessárias para uma chave de 128, 192 e 256 são, respectivamente, 10, 12 e 14 [2].

Passo 2.1. No primeiro procedimento do ciclo iremos aplicar um *left-rotate* de 1 *byte* na *word*, que é uma parte de 4 *bytes* da chave expandida.

Passo 2.2. Então iremos aplicar o procedimento de *SubBytes* para cada *byte* da *word* gerada no passo anterior.

[2] A quantidade de rodadas necessárias pode ser calculada através da expressão $R = T/32 + 6$, onde R é quantidade de rodadas e T o tamanho da chave em *bits*.

Passo 2.3. Daí faremos um XOR do *byte* mais a esquerda (o primeiro byte da *word*) com a posição i (identificador do ciclo) da Tabela RCON 4.3.

Passo 2.4. Iremos repetir este passo quatro vezes a fim de misturar os 4 *bytes* da *word* fazendo um XOR com os n últimos *bytes* da *expanded key*, ou de forma mais clara, com os *bytes* da última sub-chave gerada. O resultado deste passo será concatenado à chave expandida.

Passo 2.5 Neste ponto iremos checar se a chave expandida possui 176, 208 ou 240 bits, dependendo da quantidade de bits da chave (128, 192 ou 256 bits, respectivamente). Se a quantidade já foi atingida, iremos para o passo 3. Senão, iremos para o passo 2.6.

Passo 2.6 Este passo é especial e apenas precisa ser executado caso a chave possua 256 bits. Iremos aplicar a S-Box na *word* e depois aplicar um XOR do resultado com os bytes da última sub-chave gerada. Concatene o resultado deste passo à chave expandida.

Passo 2.7 Caso a chave possua 192 ou 256 bits, este passo também deverá ser executado. Iremos realizar um procedimento parecido com o passo 2.4, porém aqui iremos repetir o procedimento duas vezes no caso da chave possuir 192 bits e três vezes no caso da chave possuir 256 bits. Iremos concatenar o resultado deste passo à *expanded key*.

Passo 3. No último ciclo iremos expandir sem utilizar os casos especiais 2.6 e 2.7 (no caso das chaves de 192 e 256 bits).

Passo 4. Ao final deste processo iremos ter, finalmente, a *expanded key* com 176, 208 ou 240 bits.

O procedimento AddRoundKey receberá como entrada a chave expandida e o estado atual do bloco, onde será feita a aplicação de XOR para cada posição do bloco na respectiva posição da chave. Por essa função utilizar uma operação reversível não existirá sua inversa, como ocorreu no caso da SubBytes, por exemplo. O que irá diferenciar um processo de encriptação de um de decriptação será a ordem no qual identificador de rodada é informado. Caso as rodadas sejam crescentes, uma encriptação ocorre e, caso contrário, um processo para recuperar a informação está sendo executado.

O procedimento de deslocamento de linhas ShiftRows realizará um deslocamento à esquerda de $l - 1$, onde l representa a linha que está deslocada. De forma mais clara, não teremos um deslocamento na primeira linha, na segunda linha haverá uma rotação à esquerda, na terceira duas rotações e na última linha três rotações à esquerda, ou uma à direita. Enquanto que no processo inverso, para decriptar, as rotações serão feitas no sentido inverso, ou seja, para a direita, e com a mesma quantidade de rotações.

O último procedimento, MixColumns, que modifica o bloco de estado através de multiplicações sobre o corpo finito de Galois consiste em tratar cada entrada da coluna de estado como um elemento do corpo e realizar uma multiplicação modular $x^4 + 1$ com o polinômio fixo $3x^3 + x^2 + x + 2$. Esse processo pode ser visto como uma multiplicação matricial sobre o corpo finito de Galois. O processo inverso denominado MixColumnsInv apenas substitui o polinômio utilizado por $11x^3 + 13x^2 + 9x + 14$ e realiza as operações de forma natural. De forma geral, utiliza-se uma tabela pré-calculada para realizar esse procedimento a fim de evitar que a complexidade da implementação seja elevada e, assim, os custos computacionais sejam reduzidos. O custo dessa redução de complexidade é uma elevação na quantidade de memória, justificando assim que dispositivos que possuem pouca capacidade de espaço façam essas operações e elevem seu consumo energético. As tabelas utilizadas nesse procedimento serão mostradas no Apêndice A.2.

O procedimento de encriptação e decriptação utilizará os procedimentos AddRoundKey, SubBytes, ShiftRows e MixColumns. A inicialização do processo ocorre através da expansão de chaves descrita anteriormente, que gerará a chave expandida e, logo após a aplicação do procedimento AddRoundKey, que modificará a entrada inicial realizando o XOR com a chave expandida. Depois será executada uma sequência de passos que compreendem os procedimentos SubBytes que modificará o estado, e, posteriormente, os procedimentos para criar difusão ShiftRows e MixColumns. Ao final de cada iteração intermediária o AddRoundKey deverá ser chamado de forma que a próxima iteração seja preparada. A rodada final executará todos os procedimentos exceto o MixColumns. No processo para recuperar a informação, ou seja, a decriptação, as funções serão utilizadas na mesma ordem, porém com suas inversas respectivas, se assim lhe for aplicável.

```
r ← 0;
gere a chave expandida;
add_round_key();
enquanto r < R faça
    sub_bytes();
    shift_rows();
    mix_columns();
    add_round_key();
    r ← r + 1;
fim
sub_bytes();
shift_rows();
add_round_key();
```

Procedimento 38: Processo de encriptação do algoritmo AES.

4.2.2 Implementação do AES

A implementação do algoritmo descrito anteriormente pode ser visualizado abaixo. Cada passo será dividido em uma subseção para efeitos de melhor legibilidade, onde também serão explicadas as decisões tomadas e como o projeto foi idealizado. O código e alguns comentários serão apresentados em cada sub-seção, detalhando um pouco mais do algoritmo e como a implementação lidou com as limitações técnicas da linguagem. Isso também é feito de forma a auxiliar pessoas que desconheçam o tema e possam vir a sentir dificuldade ao entender a implementação. É importante ressaltar que nenhum cálculo em relação às tabelas apresentadas na seção anterior foi feito, de forma que essas sejam utilizadas diretamente e pré-calculadas. Isso se justifica pela quantidade de operações que seriam acrescidas e que possivelmente teriam um impacto negativo no compilador *online*. Além disso, a complexidade da implementação é reduzida, assim como as funções que seriam necessárias para realizar as operações sobre o corpo finito.

4.2.2.1 Inicialização do AES

A função de inicialização do AES pode ser definida da seguinte forma: o tamanho da chave que será utilizada (128, 192 ou 256 bits, onde nesta implementação está disponível apenas a versão de 128 bits), o modo de operação (0 para encriptação ou 1 para decriptação), a chave criptográfica e a mensagem. As constantes definidas inicialmente são o tamanho do bloco com 16 *bytes*, a quantidade de *rounds* definida através da fórmula [3] $AES_SIZE/32 + 6$, a quantidade necessária de *bytes* para armazenar a chave expandida calculada através da fórmula [4] $16 \times (ROUNDS + 1)$ devido ser a chave inicial somada aos outros *rounds* necessários, a inicialização dos vetores que irão armazenar as tabelas pré-calculadas de substituição SBOX e SBOX-Inv, corpo finito de Galois e as constantes de rounds contidas na RCON.

Os primeiros passos, mostrados no Procedimento 39, a serem feitos durante a inicialização são o preenchimento das Tabelas SBOX 4.1, SBOX-Inv 4.2 e RCON 4.3 que servem para o passo de expansão de chaves, ou seja, gerar a chave expandida. Logo após são preenchidas as tabelas do corpo de Galois que serão utilizadas no procedimento do **MixColumns**, como vimos anteriormente. Daí será checado através da *flag* **mode** se devemos encriptar ou decriptar a mensagem. Caso **mode** seja 0 faremos a encriptação da mensagem, e decriptaremos caso contrário. As duas funções recebem os mesmos pa-

[3] 128 *bits* = 10, 192 *bits* = 12 e 256 *bits* = 14 *rounds*.
[4] 128 *bits* = 176, 192 *bits* = 208 e 256 *bits* = 240 *bytes*.

râmetros e são uma a inversa da outra, ou seja, são apenas para facilitar o entendimento. De forma mais clara, poderíamos fazer uma chamada da função de encriptar utilizando **call init_aes()** e para decriptar **uncall init_aes()**, e ainda teríamos uma economia em termos de espaço, visto que não precisaríamos mais da *flag* **mode**.

```
1  procedure init_aes(int aes_size, int mode, int key[], int block[])
2      local int block_size = 16
3      local int rounds_required = aes_size / 32 + 6
4      local int expanded_key[(rounds_required + 1) * 16]
5      local int SBOX[256]
6      local int SBOXINV[256]
7      local int GF02[256]
8      local int GF03[256]
9      local int GF09[256]
10     local int GF11[256]
11     local int GF13[256]
12     local int GF14[256]
13     local int RCON[11]
14     call fill_sbox(SBOX)
15     call fill_sbox_inv(SBOXINV)
16     call fill_rcon(RCON)
17     call expand_key(RCON, SBOX, SBOXINV, key, expanded_key)
18     call fill_gf02(GF02)
19     call fill_gf03(GF03)
20     call fill_gf09(GF09)
21     call fill_gf11(GF11)
22     call fill_gf13(GF13)
23     call fill_gf14(GF14)
24     if mode = 0 then
25         call encrypt_block(block, expanded_key, rounds_required,
26             SBOX, SBOXINV, GF02, GF03, GF09, GF11, GF13, GF14)
27     else
28         call decrypt_block(block, expanded_key, rounds_required,
29             SBOX, SBOXINV, GF02, GF03, GF09, GF11, GF13, GF14)
30     fi mode = 0
```

Procedimento 39: Proposta de função de inicialização para o algoritmo do AES.

A continuação do Procedimento 39 mostrada no Procedimento 40, por razões de espaço, apenas tem como objetivo zerar as matrizes do corpo de Galois, realizar o processo inverso de expansão de chaves e zerar as demais Tabelas SBOX 4.1, SBOX-Inv 4.2 e RCON 4.3. Quaisquer erros decorrentes da expansão de chaves ou preenchimento das tabelas acarretaria um erro no processo de desalocação do espaço utilizado pela função.

Apenas para fins didáticos e que seja mais claro, verifique o Procedimento 41 que

```
31      uncall fill_gf14(GF14)
32      uncall fill_gf13(GF13)
33      uncall fill_gf11(GF11)
34      uncall fill_gf09(GF09)
35      uncall fill_gf03(GF03)
36      uncall fill_gf02(GF02)
37      uncall expand_key(RCON, SBOX, SBOXINV, key, expanded_key)
38      uncall fill_rcon(RCON)
39      uncall fill_sbox_inv(SBOXINV)
40      uncall fill_sbox(SBOX)
41      delocal int RCON[11]
42      delocal int GF14[256]
43      delocal int GF13[256]
44      delocal int GF11[256]
45      delocal int GF09[256]
46      delocal int GF03[256]
47      delocal int GF02[256]
48      delocal int SBOXINV[256]
49      delocal int SBOX[256]
50      delocal int expanded_key[(rounds_required + 1) * 16]
51      delocal int rounds_required = aes_size / 32 + 6
52      delocal int block_size = 16
```

Procedimento 40: Continuação da proposta de inicialização do algoritmo AES.

tem como objetivo "configurar" as entradas para que a função descrita acima seja inicializada corretamente. Note que como a linguagem não possui suporte a Strings é necessário que os valores de entrada, tanto da chave quanto da mensagem, sejam inteiros. O bloco de entrada *block* se traduz como "helloworldjanusp" e a chave secreta *key* como "ultrasafepassword".

```
1  procedure main()
2      int aes_size
3      int mode
4      int block[16] = {0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x77, 0x6f,
5              0x72, 0x6c, 0x64, 0x6a, 0x61, 0x6e, 0x75, 0x73, 0x70}
6      int key[16] = {0x75, 0x6c, 0x74, 0x72, 0x61, 0x73, 0x61,
7              0x66, 0x65, 0x70, 0x61, 0x73, 0x73, 0x77, 0x6f, 0x72}
8      aes_size += 128
9      call init_aes(aes_size, mode, key, block)
```

Procedimento 41: Função de exemplo para utilização da proposta do AES.

Como o valor padrão para inteiros é zero, a operação de encriptação será feita no Procedimento 41. Caso a operação de decriptação fosse desejada, apenas o valor de *mode*

deveria ser alterado para 1. De maneira similar, poderíamos manter o valor da variável **mode**, e termos feito um **uncall** ao invés de **call** na linha 9, com isso obteríamos o mesmo resultado. Como o espaço alocado para a *flag* que possibilita essa abordagem é ínfima, e traz legibilidade para quem vê, optamos por mantê-la.

4.2.2.2 Encriptação e decriptação

Na função principal de encriptação do algoritmo, mostrada no Procedimento 42, temos que todas as tabelas e valores já estão consistentes e prontas para uso. Por questões de organização foi escolhido calcular as tabelas do corpo de Galois anteriormente, porém essas poderiam ser calculadas nesta função. O espaço utilizado será o mesmo, porém a assinatura do procedimento seria menor. Em compensação, a complexidade da implementação seria acrescida e o tempo gasto seria ligeiramente superior.

```
1  procedure encrypt_block(int block[], int expanded_key[],
2    int rounds_required, int SBOX[], int SBOXINV[], int GF02[],
3    int GF03[], int GF09[], int GF11[], int GF13[], int GF14[])
4      local int round = 0
5      call add_round_key(expanded_key, block, round)
6
7      round += 1
8      from round = 1
9      do
10         call sub_bytes(block, SBOX, SBOXINV)
11         call shift_rows(block)
12         call mix_columns(block,GF02,GF03,GF09,GF11,GF13,GF14)
13         call add_round_key(expanded_key, block, round)
14     loop
15         round += 1
16     until round = rounds_required - 1
17     round += 1
18
19     call sub_bytes(block, SBOX, SBOXINV)
20     call shift_rows(block)
21     // não precisa chamar o mix columns no último round
22     call add_round_key(expanded_key, block, round)
23
24     delocal int round = rounds_required
```

Procedimento 42: Função de encriptação proposta para o AES.

A função inversa da encriptação, visualizada no Procedimento 43, pode ser entendida como simplesmente começar a contagem de *rounds* de maneira inversa. Note que

```
1  procedure decrypt_block(int block[], int expanded_key[],
2    int rounds_required, int SBOX[], int SBOXINV[], int GF02[],
3    int GF03[], int GF09[], int GF11[], int GF13[], int GF14[])
4      local int round = rounds_required
5      call add_round_key_inv(expanded_key, block, round)
6
7      round -= 1
8      from round = rounds_required - 1
9      do
10         call shift_rows_inv(block)
11         call sub_bytes_inv(block, SBOX, SBOXINV)
12         call add_round_key_inv(expanded_key, block, round)
13         call mix_columns_inv(block,GF02,GF03,GF09,GF11,GF13,GF14)
14     loop
15         round -= 1
16     until round = 1
17     round -= 1
18
19     call shift_rows_inv(block)
20     call sub_bytes_inv(block, SBOX, SBOXINV)
21     call add_round_key_inv(expanded_key, block, round)
22     // não precisa chamar o mix columns no último round
23
24     delocal int round = 0
```

Procedimento 43: Função de decriptação proposta para o AES.

podemos até inverter a ordem das operações *shift_rows* e *sub_bytes*, ou *add_round_key* e *mix_columns*, visto que as operações feitas nesses pares não afetam o resultado. Por exemplo, fazer a substituição e depois realizar o deslocamento ou deslocar e depois substituir resultará na mesma saída. Mas como estamos utilizando uma linguagem reversível, poderíamos utilizar apenas uma das funções e nos beneficiar de **call** e **uncall**.

4.2.2.3 Expansão de chaves

O procedimento para realizar a expansão de chaves foi idealizado de modo a generalizar as versões do AES, ou seja, 128 bits, 192 bits e 256 bits. Porém, por dificuldades técnicas, apenas a versão de 128 bits foi implementada. Nas demais versões, de 192 e 256 bits, o *"round"* termina antes de realizar um passo completo de expansão, devido a isso é necessário checar a cada iteração se a chave foi expandida completamente. Ao colocarmos essa condição através de uma estrutura condicional **IF**, o compilador, por algum motivo, não gera nenhum tipo de saída, impossibilitando assim a implementação. Além disso foi pen-

sado em uma forma de retornar a execução para algum ponto através de um **RETURN**, ou **GOTO**, porém nenhuma dessas está disponível na linguagem. Nos Procedimentos 44 e 45 se encontra a implementação proposta para esse procedimento, onde as variáveis são mapeadas da seguinte forma: $R[]$ contem os valores da RCON, $SB[]$ armazena os valores da SBOX, $SBI[]$ da SBOX inversa, $k[]$ é o vetor utilizado como entrada padrão e que contém a chave criptográfica, $expk[]$ irá armazenar o resultado da chave expandida e que será utilizada para realizar os demais passos.

```
 1  procedure expand_key(int R[],int SB[],int SBI[],int k[],int expk[])
 2      local int word[4]
 3      local int rounds_required = (size(expk) / 16) - 1
 4      local int rcon_iteration = 1
 5      local int bytes_generated = 0
 6      local int j = 0
 7      local int z = 0
 8
 9      call copy_array(k, bytes_generated, expk)
10      bytes_generated += size(k)
11
12      word[0] ^= expk[bytes_generated - 4]
13      word[1] ^= expk[bytes_generated - 3]
14      word[2] ^= expk[bytes_generated - 2]
15      word[3] ^= expk[bytes_generated - 1]
```

Procedimento 44: Proposta para função de expansão de chaves do AES.

Nas linhas de 2 a 7 do Procedimento 44 a inicialização do procedimento é feita, onde é calculada a quantidade de iterações necessárias para a expansão completa da chave. Logo em seguida, nas linhas 9 e 10, a primeira expansão ocorre simplesmente copiando a chave inicial para a chave expandida e incrementando o contador de *bytes* que servirá para controlar o tamanho da chave expandida gerada. O ciclo de expansão inicia utilizando a *word*, uma variável temporária de 4 *bytes*, para realizar as etapas de expansão em cada ciclo. Note que a variável é criada inicialmente fazendo uso de **XOR** para evitar que operações de somas, que são mais custosas, sejam realizadas.

A cada iteração mostrada no Procedimento 45 a *word* será rotacionada para a esquerda e aplicada uma função denominada **SubWord**, que possui o mesmo funcionamento da **SubBytes**, e que modifica um *array* de 4 posições. Logo após é aplicado um **XOR** na primeira posição do vetor e a expansão da *word* interna se inicia. Nas linhas 22 a 37 a *word* temporária é modificada e expandida, onde a cada iteração é aplicado um **XOR** com a última *word* gerada. As *words* intermediárias são concatenadas à chave expandida

```
16      from rcon_iteration = 1
17      do
18          call rotate_left_array(word)
19          call sub_bytes(word, SB, SBI)
20          word[0] ^= R[rcon_iteration]
21
22          from z = 0
23          do
24              from j = 0
25              do
26                  word[j] ^= expk[bytes_generated - size(k) + j]
27              loop
28                  j += 1
29              until j = 3
30              j -= 3
31
32              call copy_array(word, bytes_generated, expk)
33              bytes_generated += size(word)
34          loop
35              z += 1
36          until z = 3
37          z -= 3
38      loop
39          rcon_iteration += 1
40      until rcon_iteration = rounds_required
41
42      word[0] -= expk[bytes_generated - 4]
43      word[1] -= expk[bytes_generated - 3]
44      word[2] -= expk[bytes_generated - 2]
45      word[3] -= expk[bytes_generated - 1]
46
47      delocal int z = 0
48      delocal int j = 0
49      delocal int bytes_generated = size(expk)
50      delocal int rcon_iteration = rounds_required
51      delocal int rounds_required = size(expk) / 16 - 1
52      delocal int word[4]
```

Procedimento 45: Continuação da proposta para expansão de chaves do AES.

final e a variável de contador é incrementada para que seja possível detectar quando a expansão foi concluída.

Após o ciclo de iterações terminar a variável temporária é zerada, ou seja, volta ao estado inicial. Neste ponto a expansão de chaves foi realizada com sucesso e todas as variáveis que tenham sido inicializadas podem ser recuperadas realizando a operação

inversa. Lembre-se que a *keyword* **delocal** equivale à operação de inicialização **local**, porém inversa, a fim de recuperar o estado inicial.

4.2.2.4 AddRoundKey

Como dito anteriormente a etapa de adicionar as chaves de *round*, ou **AddRoundKey**, não possui inversa, porém para facilitar a compreensão foi criada uma função denominada *add_round_key_inv()* que executa uma chamada para a função "normal". Essa etapa é feita múltiplas vezes modificando o parâmetro de *round* e *block[]*. A implementação desse procedimento pode ser visualizada no Procedimento 46.

```
1  procedure add_round_key(int expanded_key[], int block[], int round)
2      local int offset = round * 16
3      local int i = 0
4
5      from i = 0
6      do
7          block[i] ^= expanded_key[offset + i]
8      loop
9          i += 1
10     until i = 15
11
12     delocal int i = 15
13     delocal int offset = round * 16
14
15 procedure add_round_key_inv(int expanded[], int block[], int round)
16     call add_round_key(expanded, block, round)
```

Procedimento 46: Implementação da função AddRoundKey e sua inversa.

No inicio do procedimento é calculado o *offset* que será utilizado para adicionar a chave de *round*. Em cada iteração é feita a operação de **XOR** do resultado parcial com a *expanded_key*. Após feito isso a variável que contém o *offset* é desalocada.

4.2.2.5 SubBytes e SubBytes-Inv

De modo similar, apesar da etapa de substituição utilizada para criar confusão não possuir inversa propriamente dita, uma outra tabela denominada SBOX-Inv, e que pode ser vista na Tabela 4.2, é utilizada para realizar a recuperação dos dados criptografados. Sabendo disso, a função inversa apenas inverte os parâmetros a fim de que a função que utiliza a SBOX possa ser reutilizada. Confira no Procedimento 47 a implementação feita.

```
1  procedure sub_bytes(int block[], int SBOX[], int SBOXINV[])
2      local int bsize = size(block)
3      local int b[bsize]
4      local int i = 0
5
6      from i = 0
7      do
8          b[i] += SBOX[block[i]]
9      loop
10         i += 1
11     until i = bsize - 1
12
13     from i = bsize - 1
14     do
15         block[i] -= SBOXINV[b[i]]
16     loop
17         i -= 1
18     until i = 0
19
20     b <=> block
21
22     delocal int i = 0
23     delocal int b[bsize]
24     delocal int bsize = size(block)
25
26 procedure sub_bytes_inv(int block[], int SBOX[], int SBOXINV[])
27     call sub_bytes(block, SBOXINV, SBOX)
```

Procedimento 47: Implementação da função SubBytes e sua inversa.

Inicialmente é identificado o tamanho do bloco no qual estamos processando. Apesar de o AES padrão utilizar sempre tamanho 16, outras implementações podem variar o tamanho do bloco. Depois é alocado um vetor de mesmo tamanho que o bloco e que será responsável por guardar a informação de forma intermediária. A aplicação da SBOX é feita armazenando as informações no vetor temporário, enquanto que o vetor original é zerado através da aplicação da SBOX-Inv. Para retornarmos a informação correta fazemos um *swap* entre os dois vetores. Ao final dessas operações teremos que o vetor de entrada terá a informação após a aplicação da SBOX e o vetor temporário estará zerado. Analogamente, a função inversa irá aplicar primeiramente a SBOX-Inv e depois a SBOX.

4.2.2.6 ShiftRows e ShiftRows-Inv

Essa etapa alinhada com a função de MixColumns serve para criar difusão, porém essa tem a vantagem de que pode ser feita apenas com *swaps* e por isso pode ser implementada de forma mais simples. A função inversa ainda pode reutilizar a função "normal" apenas fazendo o procedimento de trás para frente. Verifique no Procedimento 48 a implementação idealizada.

```
1  procedure shift_rows(int b[])
2      b[01] <=> b[05]
3      b[05] <=> b[09]
4      b[09] <=> b[13]
5
6      b[02] <=> b[10]
7      b[06] <=> b[14]
8
9      b[03] <=> b[15]
10     b[07] <=> b[11]
11     b[15] <=> b[07]
12
13 procedure shift_rows_inv(int b[])
14     uncall shift_rows(b)
```

Procedimento 48: Implementação das funções ShiftRows e ShiftRowsInv.

A implementação que é bastante trivial utiliza apenas trocas de posições, de forma que a informação não seja perdida ou sequer gere novos *bits*. Note que ao contrário das demais implementações supracitadas o procedimento *shift_rows_inv* realiza um **uncall**, ou seja, realiza a função de forma inversa, iniciando nos últimos passos e terminando nos primeiros passos, em relação à função "normal". De forma mais sucinta, em vez de serem realizadas 3 rotações no mesmo sentido, apenas uma rotação no sentido oposto é feita.

4.2.2.7 MixColumns e MixColumns-Inv

Esse procedimento foi implementado de forma similar ao **SubBytes**, utilizando uma variável temporária e recuperando o valor através das propriedades do algoritmo criptográfico. Apesar de não ser totalmente eficiente, reduz a quantidade de espaço utilizado e facilita a compreensão do algoritmo. A função "normal", **MixColumns**, utiliza as matrizes de corpo de Galois 2 e 3, enquanto que a inversa faz uso das matrizes 9, 11, 13 e 14. Porém, pela forma na qual foi implementada, os parâmetros são os mesmos em ambas. Pode-se imaginar que isso acarretará em uma maior quantidade de espaço utili-

zado, mas lembre-se de que o espaço já está sendo ocupado devido essas matrizes serem pré-calculadas. Poder-se-ia ter uma economia de espaço, e na assinatura do método, caso as matrizes fossem calculadas sob demanda, porém isso acarretaria em diversos cálculos sendo feitos a cada iteração. Como a linguagem Janus não possui suporte a paralelismo e mecanismos de medição de tempo, essa ideia foi descartada por acharmos que o impacto seria relativamente alto. O Procedimento 49 mostra a implementação do MixColumns enquanto que o Procedimento 50 mostra sua versão inversa denominada MixColumnsInv.

```
1  procedure mix_columns(int bk[], int G2[], int G3[], int G9[],
2                        int G11[], int G13[], int G14[])
3      local int b[16]
4      local int c = 0
5
6      from c = 0
7      do
8          b[c+0] += G2[bk[c+0]] ^ bk[c+3] ^ bk[c+2] ^ G3[bk[c+1]]
9          b[c+1] += G2[bk[c+1]] ^ bk[c+0] ^ bk[c+3] ^ G3[bk[c+2]]
10         b[c+2] += G2[bk[c+2]] ^ bk[c+1] ^ bk[c+0] ^ G3[bk[c+3]]
11         b[c+3] += G2[bk[c+3]] ^ bk[c+2] ^ bk[c+1] ^ G3[bk[c+0]]
12         c += 4
13     until c = 16
14     c -= 16
15
16     from c = 0
17     do
18         bk[c+0] -= G14[b[c+0]]^G9[b[c+3]]^G13[b[c+2]]^G11[b[c+1]]
19         bk[c+1] -= G14[b[c+1]]^G9[b[c+0]]^G13[b[c+3]]^G11[b[c+2]]
20         bk[c+2] -= G14[b[c+2]]^G9[b[c+1]]^G13[b[c+0]]^G11[b[c+3]]
21         bk[c+3] -= G14[b[c+3]]^G9[b[c+2]]^G13[b[c+1]]^G11[b[c+0]]
22         c += 4
23     until c = 16
24     c -= 16
25
26     b <=> bk
27
28     delocal int c = 0
29     delocal int b[16]
```

Procedimento 49: Implementação proposta para a função MixColumns do AES.

Esse procedimento apesar de ser maior que o **SubBytes** devido a não ser possível simplesmente inverter as matrizes, visto que a quantidade de matrizes utilizadas nos métodos é diferente. Porém, percebemos que existe um padrão, logo há a possibilidade de otimização. Note também que as operações podem ser feitas em paralelo. Por exemplo,

```
1  procedure mix_columns_inv(int bk[], int G2[], int G3[], int G9[],
2                            int G11[], int G13[], int G14[])
3      local int b[16]
4      local int c = 0
5
6      from c = 0
7      do
8         b[c+0] += G14[bk[c+0]]^G9[bk[c+3]]^G13[bk[c+2]]^G11[bk[c+1]]
9         b[c+1] += G14[bk[c+1]]^G9[bk[c+0]]^G13[bk[c+3]]^G11[bk[c+2]]
10        b[c+2] += G14[bk[c+2]]^G9[bk[c+1]]^G13[bk[c+0]]^G11[bk[c+3]]
11        b[c+3] += G14[bk[c+3]]^G9[bk[c+2]]^G13[bk[c+1]]^G11[bk[c+0]]
12        c += 4
13     until c = 16
14     c -= 16
15
16     from c = 0
17     do
18        bk[c+0] -= G2[b[c+0]] ^ b[c+3] ^ b[c+2] ^ G3[b[c+1]]
19        bk[c+1] -= G2[b[c+1]] ^ b[c+0] ^ b[c+3] ^ G3[b[c+2]]
20        bk[c+2] -= G2[b[c+2]] ^ b[c+1] ^ b[c+0] ^ G3[b[c+3]]
21        bk[c+3] -= G2[b[c+3]] ^ b[c+2] ^ b[c+1] ^ G3[b[c+0]]
22        c += 4
23     until c = 16
24     c -= 16
25
26     b <=> bk
27
28     delocal int c = 0
29     delocal int b[16]
```

Procedimento 50: Implementação proposta para a função MixColumnsInv.

na linha 18, poderiamos calcular $G14[b[c+0]] \oplus G9[b[c+3]]$, $G13[b[c+2]] \oplus G11[b[c+1]]$ e depois fazermos o **XOR** dos resultados obtidos por essas operações. Ou até mesmo calcularmos linhas diferentes, como por exemplo, $G14[bk[c+0]]$, $G14[bk[c+1]]$, $G14[bk[c+2]]$, $G14[bk[c+3]]$ e $G14[bk[c+4]]$ e utilizarmos esses resultados posteriormente.

4.2.3 Comparação com a versão em circuitos

Para efeitos de comparação e eficiência utilizaremos o circuito apresentado por Grassl et al. [28] que tem como objetivo estimar os recursos quânticos necessários para realizar a encriptação e decriptação do algoritmo supracitado e, além disso, utiliza o algoritmo de Grover [30] para analisar os recursos necessários para realização da criptanálise do

AES, ou seja, a simulação de um ataque que tem como objetivo recuperar informações da entrada ou extrair informações sensíveis sobre a criptografia. Porém não iremos entrar em detalhes sobre a criptanálise em si, apenas iremos analisar as operações criptográficas para "esconder" (encriptar) a mensagem, o que também se aplica, indiretamente, a sua função inversa para decriptar.

A versão em circuito tenta minimizar as operações, seja em quantidade de *qubits*, seja em *gates* lógicos utilizados, além de apresentar alternativas intermediárias que "equilibram" ambas. Como a quantidade de bits e *qubits* é relativa, não iremos analisar sob este aspecto, de fato iremos fazê-la mapeando a quantidade de *gates* e operações necessárias para realizar uma determinada função (ou parte do algoritmo). Por exemplo, um *SWAP* feito em alto nível utilizará 3 operações por bit, pois o equivalente em circuitos seriam 3 *CNOTs*. De modo similar, diversos algoritmos (e circuitos) existem para realização das operações de soma/subtração, multiplicação, divisão, exponenciação e outras. E como nossa implementação em alto nível depende de somas, multiplicações, divisões e *swaps* algumas decisões focadas em reduzir a quantidade de operações foram tomadas. Abaixo serão descritas a quantidade de operações necessárias para cada uma dessas funções essenciais.

Tabela 4.4: Quantidade de operações necessárias utilizando 8 bits para cada função.

Circuito	**Qtd. Operações**
ADDR [37]	80
ADDR [61]	76
C-ADDR [61]	92
MULT1 [37]	736
MULT2 [37]	1960
DIV [37]	1224
SWAP	24

Analisando a Tabela 4.4 a decisão óbvia seria utilizar as funções com menor quantidade de operações, visto que desejamos minimizá-las. Porém adotamos algumas soluções não ortodoxas como, por exemplo, utilizar o circuito MULT2 em vez do MULT1. Por mais que o primeiro circuito seja muito mais eficiente que o outro, em termos de quantidade de operações, essa decisão foi tomada baseando-se no conceito de que a operação de multiplicação em uma linguagem em alto nível seja provavelmente eficiente em termos de espaço. Portanto, enquanto o circuito MULT1 realiza a operação de multiplicação da forma $(a, b, 0) \rightarrow (a, b, ab)$, ou seja, guardando o resultado no terceiro operando, o circuito MULT2 utiliza uma forma de substituição $(a, b) \rightarrow (a, ab)$ e portanto economiza

espaço a longo prazo. Note que o MULT1 poderia ser utilizado sem problemas, porém precisaríamos chegar ao final da multiplicação para que o valor fosse revertido e o espaço pudesse ser recuperado. Ambos os circuitos foram propostos por Kowada [37].

Caso o primeiro circuito fosse utilizado poderíamos economizar 13464 operações em relação à segunda abordagem, visto que são feitas 11 multiplicações ao longo da implementação. Mas precisaríamos guardar todos os resultados intermediários feitos na multiplicação, impactando negativamente no desempenho e estruturação geral da implementação.

No caso da operação de soma acabamos optando pelos circuitos apresentados por Vedral [61] ao invés do ADDR proposto por Kowada [37], mesmo que a diferença na quantidade de operações seja pequena. Isso se deve ao fato de que a implementação proposta do algoritmo AES utiliza muitas operações de soma, então o impacto será acrescido em 4 operações em cada utilização. Como são feitas 3174 operações de soma, apenas para efeitos de comparação, o circuito proposto por Vedral utilizará, no total, 241224 operações enquanto que o outro circuito irá precisar realizar 253920 operações. Desse modo podemos economizar, aproximadamente, 13 mil operações apenas utilizando um circuito mais complexo, porém mais eficiente em termos de quantidade de operações.

Em relação a quantidade de operações necessárias feitas na função de divisão e SWAP são considerados os valores padrão da Tabela 4.4, ou seja, 3 CNOTs para cada bit no SWAP e o circuito de divisão proposto por Kowada [37] que utiliza para cada bit 2 ADDRs de 8 bits e 1 XOR.

Se formos contabilizar todas as operações, dividindo-as em funções, teríamos algo parecido com a Tabela 4.5, apresentada abaixo, onde as funções *fill_sbox, fill_sbox_inv, fill_gf02, fill_gf03, fill_gf09, fill_gf11, fill_gf13, fill_gf14* e *fill_rcon* são constantes e portanto, se formos mapeá-las para circuitos, não haverá acréscimo na quantidade de operações. Para facilitar a compreensão e reduzir o espaço utilizado na tabela essas funções foram agrupadas em apenas uma denominada *fill_boxes*. As colunas da tabela abaixo são respectivamente: nome da função, quantidade de atribuições feitas na função, quantidade de somas, quantidade multiplicações, divisões, *swaps*, *XORs* e a quantidade total de operações mapeadas para circuito. As duas primeiras linhas da tabela, excluindo o cabeçalho, são a quantidade total de operações feitas ao longo de toda a execução enquanto que as demais são a quantidade de operações feitas por chamada de função.

A implementação utiliza uma abordagem com funções pequenas, e reversíveis, para que o espaço utilizado durante a execução seja mínimo, porém tabelas de *look-up* pré-

Tabela 4.5: Quantidade total de operações utilizadas em cada função do AES baseando-se na implementação ao longo desta dissertação.

Função	**#Atr**	**#Soma**	**#Mult**	**#Div**	**#Swap**	**#XOR**	**#Ops**
fill_boxes	2058	0	0	0	0	0	2058
copy_array	2	176	0	0	0	0	13378
rotate_left_array	2	4	0	0	0	0	306
sub_bytes(word)	6	16	0	0	4	0	1318
sub_bytes(block)	18	64	0	0	16	0	5266
add_round_key	2	32	1	0	0	16	4522
shift_rows	0	0	0	0	8	0	192
mix_columns	17	130	0	0	16	96	11049
expand_key	11	1002	0	1	0	174	78779
encrypt_block	1	2172	11	0	384	1040	204169
init_aes	2237	3174	11	1	384	1214	285173

calculadas são utilizadas reduzindo drasticamente a quantidade de operações feitas. Quase 60% das operações são feitas na parte de encriptar a mensagem de entrada, o restante é feito no processo de expansão de chaves. O processo de expansão de chaves é impactado negativamente, em termos de espaço, devido a Janus não possuir vetores dinâmicos, daí é necessário copiar os valores para cada rodada de expansão. Porém a quantidade de operações se mantém constante, em 176 como mostra a segunda linha da Tabela 4.5, pois o vetor é alocado inicialmente com este tamanho.

Apesar do circuito utilizar uma abordagem na qual todas as tabelas são calculadas ao longo da execução, e sob demanda, tentaremos traçar um paralelo com a nossa implementação em alto nível. Para facilitar o entendimento serão utilizados os mesmos nomes das funções, apesar do circuito não ser dividido desta forma. Vale ressaltar que o *gates* lógicos utilizados são reversíveis e quânticos, porém a quantidade de operações ainda pode ser calculada de maneira clássica. Para a construção completa do circuito são utilizados os *gates*: T-, *Clifford*, NOT, $CNOT$ e *Toffoli*. O conjunto de *gates* lógicos *Clifford*+T resulta em um circuito tolerante a falhas [22, 23, 51, 56], e além disso os *Clifford* são, geralmente, mais baratos em relação aos T-. Por razões do algoritmo em questão ser clássico, e qualquer erro nas operações ser catastrófica para a execução, o circuito utilizará para cada Toffoli 7 T-, apesar de existirem soluções probabilísticas [35] que fazem uso de 4 T-.

Em relação às operações, o circuito adota uma abordagem que minimiza o espaço e a profundidade, maximizando assim a eficiência dos recursos, algo necessário nestes casos mas não tão necessário em aplicações de alto nível. A *SBOX* e *SBOX-Inv*, por exemplo, utilizam 3584 T- e 4569 *Clifford*, o que se traduz em 473 $CNOT$ e 512 *Toffoli*. Os cálculos

Tabela 4.6: Quantidade total de operações necessárias realizadas pelo circuito proposto por Grassl et al. [28] mapeadas para implementação desenvolvida nesta dissertação.

Função	**#T-**	**#Clifford**	**#NOT**	**#CNOT**	**#Toffoli**	**#Ops**
fill_sbox	3584	4569	0	473	512	985
fill_sbox_inv	3584	4569	0	473	512	985
sub_bytes(word)	1792	2132	0	84	256	340
sub_bytes(block)	448	533	0	21	64	85
add_round_key	0	128	0	128	0	128
shift_rows	0	0	0	0	0	0
mix_columns	0	277	0	277	0	277
expand_key	143360	185288	176	21448	20480	42104
encrypt_block	71680	147310	0	65390	10240	75630
init_aes	1060864	1380420	0	168004	151552	319556

das demais tabelas, corpo de Galois e constantes de *round*, são feitas diretamente nas funções que as utilizam. A *SubWord*, que é a aplicação da função *SubBytes* na expansão de chaves, por sua vez, utiliza 84 *CNOT* e 256 *Toffoli*, ou 1792 *T*- e 2132 *Clifford*.

Como a função *SubBytes* irá precisar reutilizar os valores gerados, todas as saídas geradas pela *SubWord* são armazenadas até o final da execução de todos os processos de substituição necessários para encriptar a mensagem. Com isso, há um gasto elevado na *sub_bytes(word)* e reduzido em *sub_bytes(block)*, já que os valores foram armazenados anteriormente, algo que não acontece na implementação em alto nível. Essa economia nas operações reflete positivamente, e diretamente, na quantidade total de operações necessárias, visto que a geração da *SBOX* é feita de maneira sequencial, ou seja, para gerar o valor i é necessário conhecer o(s) valor(es) anterior(es). Caso esses valores não sejam guardados, uma abordagem alternativa seria calcular os valores utilizando a ideia proposta por Sebastian e Markus [18], que consiste em utilizar um grupo de permutações a fim de minimizar a quantidade de dados necessários ao custo de um aumento na quantidade de operações feitas. Neste caso a quantidade de *gates* lógicos seria aumentada para 9695 T- e 12631 Clifford, ao passo que a quantidade de *qubits* necessários reduziria de 40 para 9.

A operação de adicionar as constantes de round, e que utiliza a *RCON*, pode ser feita utilizando 128 *CNOT*. Em ambos os casos, Janus e circuito, as operações podem ser feitas em paralelo, apesar da linguagem Janus não suportar paralelismo. Caso o paralelismo seja realmente necessário há a opção de gerar um código *C++* através do compilador online disponibilizado pela Universidade de Copenhague e modificá-lo, externamente, a fim de torná-lo mais eficiente. Note que a utilização de bibliotecas externas, não-reversíveis, pode fazer com que a aplicação se torne instável e, provavelmente, irreversível.

Como a função *ShiftRows* consiste apenas em reposicionar os bits, ou seja realizar uma permutação nos mesmos, não há gasto nenhum visto que o circuito precisa apenas conectar a(s) saída(s) com a(s) entrada(s) de forma correta para o próximo passo.

O *MixColumns* também utiliza de forma similar ao *SubBytes* um processo de multiplicação de matrizes, porém mais trivial e em que não há dependência sequencial de valores. Uma decomposição LUP (ou decomposição LU -*lower up*- com pivoteamento parcial) é realizada e cada coluna da matriz, que possui dimensões 32x32, utiliza 277 CNOT por vez.

A expansão de chaves, como dito anteriormente, utiliza a *SubWords*, porém nem todas as *words* geradas que irão compor a chave expandida a utilizam. Assim as *words* são divididas em dois grupos: as que usam a *SubBytes* e as que não usam. Fazendo isso, segundo Grassl et al. [28], há uma economia de 75% no espaço utilizado. Mas mesmo assim há um custo elevado na quantidade de operações para que a geração de todas as *words*, visto que serão necessárias posteriormente. Apenas para fins de esclarecimento, as *words* que não fazem uso da *SubBytes* são geradas a partir das *words* anteriores através de utilização de *XOR*.

Em relação ao mecanismo utilizado no circuito para controlar os *rounds* e seus respectivos procedimentos necessários para a encriptação da mensagem, Grassl faz com que o primeiro *round* seja simplificado de modo que a saída do passo anterior (entrada da mensagem e expansão da chave) seja direcionada para um *XOR* e depois diretamente para a entrada do passo seguinte, economizando assim a "chamada" de um passo do *AddRoundKey* e utilizando 64 *NOT*. Isso reflete também na economia de 128 *qubits*. Caso o circuito executasse o procedimento *AddRoundKey*, seriam necessários 128 *qubits* e 128 *CNOT*. Em cada *round* são necessários 16 "chamadas" do *SubBytes*, o que se traduz em 384 bits auxiliares. Se o espaço for limitado há a possibilidade de utilizar apenas 24 *qubits* auxiliares além dos 128 *qubits* necessários para armazenar o resultado, ao custo de um aumento na profundidade equivalente a 8 ciclos do *SubBytes*. Como a encriptação através do AES-128 utiliza um total de 10 *rounds* são necessários 536 *qubits* (sendo 512 de armazenamento e 24 auxiliares) para a execução destes, somados aos 320 *qubits* necessários para a expansão das chaves e os 128 bits da entrada inicial, totalizando 984 *qubits*.

Uma comparação clara, direta e sucinta pode ser visualizada na Tabela 4.7, onde são mostradas as quantidades de operações em cada abordagem. O espaço utilizado não é levado em consideração, apesar do foco do circuito proposto por Grassl et al [28] ser a economia dos recursos computacionais necessários. As operações *copy_array* e *ro-*

tate_left_array foram suprimidas da tabela e incluídas na linha referente à *expand_key*. Note também que apesar da implementação proposta nesta dissertação possuir uma menor quantidade de operações, a versão em circuitos acaba sendo melhor. Isso se deve ao fato de que a abordagem feita por Grassl et al [28] calcula todos os valores em contra ponto que uma tabela de *look-up* é utilizada na versão em alto nível.

Tabela 4.7: Tabela comparativa para ilustrar a quantidade de operações feitas em cada abordagem. A versão em circuitos possui foco em reduzir a quantidade de recursos utilizados, enquanto a versão Janus tem como objetivo reduzir a quantidade de operações.

Função	**#Ops Circuito**	**#Ops Janus**
fill_boxes	1970	2058
sub_bytes(word)	340	1318
sub_bytes(block)	85	5266
add_round_key	128	4522
shift_rows	0	192
mix_columns	277	11049
expand_key	42104	92463
encrypt_block	75630	204169
init_aes	319556	285173

4.3 Caso 2 - ChaCha20

Uma outra implementação feita é a do algoritmo ChaCha [5] com 20 rodadas, ou ChaCha20, que como dito anteriormente apresenta uma eficiência maior em relação aos recursos computacionais e uma complexidade menor em relação ao AES. Esse algoritmo tem como base outro também idealizado Bernstein denominado de Salsa [6]. O nome de ambos algoritmos vem de danças latinas que possuem um ritmo acelerado e exigem um fluxo constante de movimentos, no qual os parceiros estão em constante troca de mensagens corporais. Não surpreendentemente as duas cifras são de fluxo e passos especiais são feitos a cada 4 "batidas" (ou rodadas). Na música três passos são feitos e fazem com que o compasso seja mantido, onde um novo conjunto de passos é reproduzido a cada 8 batidas. No algoritmo acontece algo similar, porém são feitos 4 passos diretamente a cada 8 batidas que se complementam de uma forma única através de combinações de colunas e diagonais.

Assim como no caso anterior uma comparação é feita, porém desta vez com um exemplo existente no compilador online da linguagem JanusP. Esta outra implementação, deste algoritmo, foi disponibilizada recentemente por um aluno de mestrado [57] e servirá para

efeitos de comparação. Ela está disponível em `https://topps.diku.dk/pirc/janus-playground/#e` e também pode ser visualizada no Apêndice B.1. Como as duas implementações estão em alto nível, e são escritas na mesma linguagem, o método utilizado verifica se a saída apresenta discrepância e avalia a eficiência em termos de operações de *swap*, somas, exponenciações, divisões, XOR, AND e OR.

4.3.1 Algoritmo do ChaCha

A base deste algoritmo é o Salsa [6] que tem como objetivo prover uma função criptográfica simples e eficiente computacionalmente para ser utilizada em grande escala nos mais variados dispositivos. Segundo Bernstein, grandes volumes de dados são criptografados diariamente para simples envio através de comunicações seguras e, por isso, uma cifra de fluxo eficiente deve ser capaz de criptografar, e descriptografar, em um curto intervalo de tempo. Os destinatários também devem ser capazes de diferenciar uma mensagem falsa de uma legitima de maneira eficaz, visto que esses também podem receber um grande volume de informações forjadas. A segurança também é algo que o preocupa porque os usuários leigos acreditam estar seguros por simplesmente usarem conexões criptografadas ou sites assinados digitalmente por terceiros confiáveis, o que geralmente não é verdade devido aos atacantes possuírem um grande poder computacional. Diversos casos em que governos, ou grandes redes de criminosos, praticavam quebra de sigilo ou impersonificação foram relatados ao longo dos anos. O mais famoso talvez seja a da agência americana NSA que espionou os próprios cidadãos (e alguns outros governos) durante décadas extraindo informações sigilosas e pessoais.

A idealização do Salsa então foi constituída através de três operações simples que usam *words* de 32 *bits*. A primeira operação é uma simples adição modular $a+b \mod 2^{32}$, onde a e b representam *words* de 32 *bits* cada. A segunda operação realiza um XOR entre duas *words* de 32 *bits*. A terceira, e última operação, realiza a rotação de b *bits* em uma *word* para a esquerda. O autor do algoritmo justifica que essas operações são extremamente eficientes em computadores de mesa, *smartphones* e dispositivos embarcados e, por este motivo, devem ser explorados ao máximo nas mais diversas CPUs. Operações com 64 *bits* poderiam ser utilizadas também, visto que a grande maioria possui suporte, porém as multiplicações e somas seriam mais complexas acarretando em muitos ciclos por instrução o que, consequentemente, elevaria o tempo gasto por operação. O algoritmo também não utiliza tabelas de substituição, como o AES, devido essas dificultarem a implementação em dispositivos com limitações de espaço, memória e poder computacional, assim como

também criarem o problema de serem vulneráveis a ataques de canal lateral e não proverem nenhum *speed-up* que justifique a utilização exaustiva de cache L1.

Além da preocupação em fazer o algoritmo eficiente, Bernstein utilizou rotações ao invés de *shifts*, o que facilita a implementação reversível deste algoritmo. A justificava pelo uso de rotações se dá porque a difusão gerada é a mesma por uma quantidade de tempo sempre menor ou igual do que a combinação *shift-shift-xor-xor*. O algoritmo expande a chave de 256 *bits* e o *nonce* [5] de 64 *bits* em um fluxo de 2^{70} *bytes*. O texto claro é encriptado realizando um XOR com os primeiros b *bytes* do fluxo e descartando o resto. O fluxo gerado é dividido em blocos de 64 *bytes*, onde cada bloco é independente e não possui qualquer encadeamento com os blocos seguintes. Assim os blocos podem ser gerados de forma paralela, e em qualquer ordem, descartando qualquer correlação existente entre os blocos ou o fluxo. Ao contrário dos outros algoritmos não há um pré-processamento das chaves para que os blocos possam ser gerados, o que elimina possíveis ataques em memória para recuperar a chave expandida. O método utilizado para a geração de fluxo também é um simples XOR que, na opinião do autor, provê a mesma segurança que métodos mais robustos como o encadeamento de blocos CBC (Cipher Block Chaining). O fluxo também não possui correlação direta com o texto claro pois o fluxo é dependente apenas da chave secreta e o do *nonce*. Isso faz com que ataques de negação de serviço sejam mitigados mais cedo do que se a mensagem fosse adicionada ao fluxo. Como o destinatário não saberia a mensagem que seria enviada, este precisaria descriptografar o conteúdo para verificar a validade da informação.

Tabela 4.8: Tabela ilustrativa do conteúdo do vetor no algoritmo Salsa.

C1	K1	K2	K3
K4	C2	N1	N2
P1	P2	C3	K5
K6	K7	K8	C4

O *core* do algoritmo Salsa possui um vetor de 16 posições, onde cada posição contém 32 *bits*, e é organizado da seguinte forma: constante $C1 = 0x61707865$, quatro espaços para os primeiros 128 *bits* da chave, constante $C2 = 0x3320646e$, dois espaços para o *nonce*, dois espaços para o contador, constante $C3 = 0x79622d32$, os 128 *bits* restantes da chave secreta e a constante $C4 = 0x6b206574$. Uma exemplificação dessa estrutura pode ser vista na Tabela 4.8. As constantes possuem o significado literal de "expand 32-byte k".

As três operações enunciadas anteriormente são utilizadas nesse ponto através de uma

[5] número único que identifica uma mensagem

função denominada *quarter-round* que consiste em receber quatro parâmetros a, b, c e d, contendo 32 *bits* cada, e calcular os novos valores para cada variável, onde o símbolo <<< se refere à rotação a esquerda:

```
b ← b ⊕ ((a + d) <<< 7);
c ← c ⊕ ((b + a) <<< 9);
d ← d ⊕ ((c + b) <<< 13);
a ← a ⊕ ((d + c) <<< 18);
```

Procedimento 51: Algoritmo da função *quarter-round* do Salsa.

A função *core* utilizará essa função *quarter-round* para executar todas as R rodadas com os seguintes parâmetros:

```
r ← 0;
copie o vetor de entrada in[] para out[];
enquanto r < R faça
    quarter_round(out[00], out[04], out[08], out[12]);
    quarter_round(out[05], out[09], out[13], out[01]);
    quarter_round(out[10], out[14], out[02], out[06]);
    quarter_round(out[15], out[03], out[07], out[11]);
    quarter_round(out[00], out[01], out[02], out[03]);
    quarter_round(out[05], out[06], out[07], out[04]);
    quarter_round(out[10], out[11], out[08], out[09]);
    quarter_round(out[15], out[12], out[13], out[14]);
    r ← r + 2;
fim
some o vetor out[] com o de entrada in[];
```

Procedimento 52: Algoritmo base do Salsa.

As primeiras quatro linhas do Algoritmo 52 irão modificar as colunas da matriz e representam as rodadas ímpares, enquanto que as outras linhas representam as rodadas pares e modificam as diagonais. Como os dados são colocados no mesmo vetor, após algumas rodadas as informações de entrada são indistinguíveis. A adição final apenas previne que um criptoanalista inverta a computação a fim de obter o resultado de entrada. O algoritmo principal é simples de entender, devido suas funções serem triviais e curtas, e cumprem com as premissas impostas anteriormente. Um ponto positivo de manter o código enxuto é que *cache* podem ser feitos e mantidos durante um intervalo de tempo maior em memória, se estiverem sendo utilizados frequentemente. A difusão criada entre as linhas e diagonais, e a confusão criada pela função *quarter-round* são suficientes e não criam uma correlação entre si até a terceira rodada, segundo o autor.

A versão aprimorada do Salsa [6] e que recebe o nome de ChaCha [5] tem como

melhoria principal a difusão por rodada. Uma reestruturação também é feita no vetor principal, porém os tamanhos da *word* assim como os valores das constantes são mantidos. Ao contrário do que se possa imaginar não há custo adicional nenhum em relação ao Salsa, ainda são realizadas 16 adições, rotações e XOR por rodada. Além disso, o nível de paralelismo é o mesmo e utiliza, segundo o autor, um registrador a menos. A função *quarter-round* sofre uma ligeira modificação e é demonstrada no Algoritmo 53.

$a \leftarrow a + b;\ d \leftarrow d \oplus a;\ d \leftarrow d <<< 16;$
$c \leftarrow c + d;\ b \leftarrow b \oplus c;\ b \leftarrow b <<< 12;$
$a \leftarrow a + b;\ d \leftarrow d \oplus a;\ d \leftarrow d <<< 8;$
$c \leftarrow c + d;\ b \leftarrow b \oplus c;\ b \leftarrow b <<< 7;$

Procedimento 53: Algoritmo da função *quarter-round* do ChaCha.

Essa modificação faz com que cada *word* seja modificada duas vezes ao invés de apenas uma. Isso acarreta uma possível correlação com a entrada porém faz com que a difusão ocorra de forma mais rápida. Segundo Bernstein a cada *quarter-round* são modificados em média 12,5 *bits* enquanto no Salsa há a mudança de 8 *bits* de saída. Note também que a quantidade de rotações foram mudadas mas não apresentam nenhuma melhora em relação ao desempenho ou segurança.

Em relação à modificação feita no vetor principal, as posições das variáveis e constante foram modificadas de forma que as constantes ocupem a primeira linha, a chave ocupe a segunda e terceira, e as posições juntamente com o *nonce* ocupe a última. Uma ilustração da reestruturação é visualizada na Tabela 4.9.

Tabela 4.9: Tabela ilustrativa da reestruturação feita no algoritmo ChaCha.

C1	C2	C3	C4
K1	K2	K3	K4
K5	K6	K7	K8
P1	P2	N1	N2

A função base do ChaCha também utiliza a função *quarter-round*, semelhante ao Salsa, porém as modificações são feitas na mesma ordem nas colunas e depois nas diagonais. O autor especula que essa abordagem de varredura melhora a difusão enquanto que a mudança feita na estrutura do vetor não tenha impacto significativo em termos de criptoanálise. A função *core* é apresentada abaixo no Algoritmo 54.

```
r ← 0;
copie o vetor de entrada in[] para out[];
enquanto r < R faça
    quarter_round(out[00], out[04], out[08], out[12]);
    quarter_round(out[01], out[05], out[09], out[13]);
    quarter_round(out[02], out[06], out[10], out[14]);
    quarter_round(out[03], out[07], out[11], out[15]);
    quarter_round(out[00], out[05], out[10], out[15]);
    quarter_round(out[01], out[06], out[11], out[12]);
    quarter_round(out[02], out[07], out[08], out[13]);
    quarter_round(out[03], out[04], out[09], out[14]);
    r ← r + 2;
fim
some o vetor out[] com o de entrada in[];
```

Procedimento 54: Algoritmo base do Salsa.

4.3.2 Implementação do ChaCha

O algoritmo descrito anteriormente segue a mesma abordagem utilizada do Caso 1 (AES) apresentado na Seção 4.2. O código adiciona algumas funções ao algoritmo de modo que a cifra de fluxo possa ser utilizada de maneira prática. Vale ressaltar que certas escolhas refletem que o código apresentado tem como objetivo demonstrar uma implementação e não um programa viável. Por exemplo, o contador utilizado inicializa em 0, apesar disso representar um parâmetro inseguro em aplicações reais. O programa também não é completo devido a não cifrar a mensagem de fato, visto que não há a entrada da mensagem em si. Nas cifras de fluxo a cifragem da mensagem é feita através da aplicação de um XOR da saída gerada com a mensagem que deseja ser enviada. Em outras palavras, o usuário poderá utilizar a implementação proposta para obter a matriz de estados e aplicar um XOR com a mensagem que deve ser transmitida a fim de realizar a cifragem.

4.3.2.1 Inicialização do ChaCha

Os parâmetros de entrada no algoritmo ChaCha compreendem a chave secreta k, o *nonce* n e a posição inicial p que por motivos de testes foi mantida com valor zero. Assim como o *setup* que tem como objetivo montar o vetor utilizado ao longo do algoritmo. O código apresentado no Procedimento 55 considera que o usuário escolheu seus parâmetros sem interferência externa, ou de forma aleatória, a fim de que não haja nenhuma tendência probabilística para algum resultado.

Essa função de *setup* será utilizada de forma direta tanto na encriptação quanto na

```
1  procedure chacha_setup(int s[], int k[], int n[], int p)
2      // sigma "expand 32-byte k" em hexadecimal
3      s[00] += 0x61707865 // primeiros 4 bytes de sigma
4      s[01] += 0x3320646e // próximos  4 bytes de sigma
5      s[02] += 0x79622d32 //
6      s[03] += 0x6b206574 //
7
8      s[04] += k[0] // primeiros 4 bytes da chave
9      s[05] += k[1] // próximos  4 bytes da chave
10     s[06] += k[2] //
11     s[07] += k[3] //
12     s[08] += k[4] //
13     s[09] += k[5] //
14     s[10] += k[6] //
15     s[11] += k[7] //
16
17     s[12] += p[0] // posição inicial do contador
18     s[13] += p[1] // posição inicial do "overflow" do contador
19
20     s[14] += n[0]  // primeiros 4 bytes do nonce
21     s[15] += n[1]  // próximos  4 bytes do nonce
```

Procedimento 55: Proposta de implementação para *setup* do ChaCha20.

decriptação, pois reflete apenas a inicialização da matriz base do algoritmo. Para uma exemplificação melhor de como utilizar a função acima, veja o Procedimento 56, que faz uso dessa e tenta simplificar o entendimento do algoritmo centralizando as funções em somente uma. Note que a mensagem a ser criptografada não é levada em consideração nessa abordagem, mas pode ser feita de maneira simples através da aplicação da operação XOR da saída *output* com a mensagem de entrada. Após a inicialização da matriz de estados, na linha 4, a variável *output* sofrerá modificação e guardará a mensagem criptografada. Ao final do Procedimento 56, na linha 6, a matriz de estados será zerada.

```
1  procedure chacha20(int output[16], int k[8], int n[2], int p)
2      local int state[16]
3
4      call chacha_setup(state, k, n, p)
5      call chacha20_core(state, output)
6      uncall chacha_setup(state, k, n, p)
7
8      delocal int state[16]
```

Procedimento 56: Abordagem proposta para simplificar funções do ChaCha20.

A função básica do ChaCha, *chacha20()*, por sua vez poderá ser utilizada de maneira

simples através da chamada da função apresentada no Procedimento 57, onde os parâmetros são passados para função base onde a criação da matriz e geração da saída será feita. Essa abordagem possibilita a utilização nos casos em que a mensagem é gerada sob demanda, de forma que um XOR possa ser feito em partes, até que os parâmetros mudem novamente, ou seja, será feita a chamada da função, os parâmetros sofrerão as modificações necessárias (incrementar posição, por exemplo) e outra chamada será feita logo em seguida.

```
1  procedure main()
2      int k[8]
3      int n[2]
4      int p[2]
5      int out[16]
6      // key 256 bits
7      k[0] += 0x34875461
8      k[1] += 0x47984269
9      k[2] += 0x52624319
10     k[3] += 0x11736182
11     k[4] += 0x22342753
12     k[5] += 0x33985433
13     k[6] += 0x75742354
14     k[7] += 0x98982563
15     // nonce 64 bits
16     n[0] += 0x97115834
17     n[1] += 0x84952347
18     // posicao inicial (contador de 32 bits)
19     p[0] += 0x00000000
20     p[1] += 0x00000000
21
22     call chacha20(out, k, n, p)
```

Procedimento 57: Função de exemplo para utilização da proposta do ChaCha20.

4.3.2.2 Core e Quarter Round

O "núcleo" do ChaCha que utiliza a função *quarter-round* para gerar o conteúdo criptografado utiliza a função de rotação à esquerda apresentada na Seção 3.2.5 e as operações de soma modular da Seção 3.1.1. A função XOR é a padrão da linguagem pois atende aos requisitos de operações com 32 *bits* por padrão. Primeiramente será apresentada a função **chacha20_core()**, que pode ser visualizada no Procedimento 58, e que recebe como parâmetros apenas os vetores de entrada e saída. Inicialmente a função calcula o valor do módulo que será utilizado $n = 2^{32}$ e copia o valor do vetor de entrada para o

vetor de saída. Logo após os *rounds* iniciam, onde são necessárias 4 etapas para cada, e ao final o vetor de saída é novamente somado (usando uma operação modular) com o vetor de entrada. Note que das linhas 13 a 30 são feitas 10 iterações, ao invés de 20 *rounds*. Isso se justifica devido a cada iteração serem executados dois *rounds* de maneira direta, sendo o primeiro para as colunas e o segundo para as diagonais.

```
1  procedure chacha20_core(int s_in[], int s_out[])
2      local int i = 0
3      local int n = 1 << 32
4
5      from i = 0
6      do
7          s_out[i] += s_in[i]
8      loop
9          i += 1
10     until i = 15
11     i -= 15
12
13     i += 1
14     from i = 1
15     do
16         call quarter_round(n,s_out[0],s_out[4],s_out[ 8],s_out[12])
17         call quarter_round(n,s_out[1],s_out[5],s_out[ 9],s_out[13])
18         call quarter_round(n,s_out[2],s_out[6],s_out[10],s_out[14])
19         call quarter_round(n,s_out[3],s_out[7],s_out[11],s_out[15])
20         call quarter_round(n,s_out[0],s_out[5],s_out[10],s_out[15])
21         call quarter_round(n,s_out[1],s_out[6],s_out[11],s_out[12])
22         call quarter_round(n,s_out[2],s_out[7],s_out[ 8],s_out[13])
23         call quarter_round(n,s_out[3],s_out[4],s_out[ 9],s_out[14])
24     loop
25         i += 2
26     until i = 19
27     i -= 19
28
29     from i = 0
30     do
31         call mod_add(s_out[i], s_in[i], n)
32     loop
33         i += 1
34     until i = 15
35     i -= 15
36
37     delocal int n = 1 << 32
38     delocal int i = 0
```

Procedimento 58: Implementação proposta para a função *core* do ChaCha20.

A outra função, *quarter-round*, é bastante trivial como foi vista anteriormente e não apresentou dificuldades em sua implementação como pode ser vista no Procedimento 59. O único fato a que devemos nos atentar é que todas as operações feitas no algoritmo são modulares de forma que as somas não devem ultrapassar os 32 *bits* pré-estabelecidos.

```
1  procedure quarter_round(int n, int a, int b, int c, int d)
2      call mod_add(a, b, n)
3      d ^= a
4      call left_rotate(d, 32, 16)
5
6      call mod_add(c, d, n)
7      b ^= c
8      call left_rotate(b, 32, 12)
9
10     call mod_add(a, b, n)
11     d ^= a
12     call left_rotate(d, 32, 8)
13
14     call mod_add(c, d, n)
15     b ^= c
16     call left_rotate(b, 32, 7)
```

Procedimento 59: Implementação da função *quarter-round* do ChaCha20.

4.3.2.3 Encriptação e decriptação

A encriptação, ou decriptação, ocorrerá através da realização da operação de XOR da mensagem com o *keystream* de 512 *bits* gerado utilizando a função **chacha20(out, k, n, p)**. Ao contrário do caso anterior, não há necessidade de expandir as chaves para realizar a encriptação. Em contraponto, os *nonces* e a posição que serão utilizadas devem ser controladas por outra parte do algoritmo ou por uma função externa. Vale ressaltar ainda que essa abordagem é tradicional em um sistema de comunicação que possui os pacotes enumerados, para fins de determinar se uma mensagem foi perdida ao longo do caminho ou não. Não entraremos em detalhes de como gerar a posição inicial, *nonces* e realizar a comunicação de forma segura pois foge do escopo desta implementação. Mas o leitor pode imaginar que os parâmetros são gerados de forma aleatória e a comunicação ocorre quando ambas as partes chegam a um consenso.

No processo de comunicação os envolvidos geram o *keystream* através dos parâmetros pré-estabelecidos e iniciam as trocas de mensagens. Quando mais de 512 *bits* de informação forem trocados, um novo *keystream* é gerado. Os programadores devem então

se atentar que antes do *keystream* "terminar" novos parâmetros para o *nonce* devem ser trocados. Geralmente, para reduzir o tráfego na rede, os *nonces* são gerados de forma determinística. Dito isso, a função de geração de números pseudoaleatórios apresentada na seção 3.5 poderá ser utilizada sem que haja essa preocupação constante com a quantidade de *bits* enviados ao longo da comunicação. A quantidade máxima de blocos é proporcional à quantidade de *bits* do contador (ou posição). Na versão proposta a quantidade máxima de mensagens é igual a 2^{64}, onde cada uma possui o tamanho de 512 *bits*. Como esse valor de tráfego é exorbitante, cerca de 1 *zettabyte*, para uma única comunicação algumas abordagens aumentam o tamanho do *nonce* de 64 para 96 *bits* e reduzem o contador para 32 *bits*, fazendo com que a quantidade máxima de informação transferida diminua para cerca de 256 *gigabytes*. Em aplicações locais, ou que lidam com um volume maior de dados, a abordagem que possui o contador com mais *bits* é preferível.

Para recuperar as mensagens iniciais, se assim for necessário, as posições deverão ser decrementadas assim como os *nonces* gerados serão utilizados na forma reversa. Se uma função PRNG reversível for utilizada, essa informação será conhecida de forma natural e nenhum outro parâmetro adicional será necessário. A mensagem em texto claro m também será recuperada de forma reversível a partir da cifrada c, visto que a operação XOR quando aplicada duas vezes retorna o valor inicial, ou seja, $m = m \oplus c \oplus c$.

4.3.3 Comparação com implementação DIKU

Para efeitos de comparação foi utilizada a implementação DIKU disponível na página do compilador *online* e feita por um estudante de mestrado [57] da mesma instituição. Vale ressaltar, porém, que a implementação feita por esse aluno não segue a risca os procedimentos definidos pelo algoritmo, misturando o algoritmo ChaCha [5] com o Salsa [6] em certas partes e não fazendo uso de operações de adição modular. Como a linguagem de programação é a mesma, iremos considerar que o desempenho apresentado nas operações será o mesmo em quaisquer eventuais compiladores que venham a surgir. Iremos considerar também que as operações mais simples usam menos ciclos que instruções mais complexas, e portanto oferecem um menor tempo e resultam em uma maior eficiência. A classificação das operações, que foram utilizadas, das mais básicas para as mais complexas foi feita da seguinte forma:

1. XOR, AND, OR, SHIFT, somas e subtrações.
2. SWAP.

3. Multiplicações, divisões e módulo.

4. Exponenciações.

```
1  procedure Chacha20_k32(int k0[4], int k1[4], int n[4], int out[16])
2  local int sigma[4] = {1634760805,857760878,2036477234,1797285236}
3  local int seq[16]
4  call Chacha20_setup_seq(k0, k1, n, sigma, seq)
5  call Chacha20(seq, out)
6  uncall Chacha20_setup_seq(k0, k1, n, sigma, seq)
7  delocal int seq[16]
8  delocal int sigma[4] = {1634760805,857760878,2036477234,1797285236}
```

Procedimento 60: Função da implementação DIKU que centraliza as funções.

Iniciamos a comparação seguindo o fluxo de execução. No final, é apresentada a comparação feita em sua totalidade. A primeira função **ChaCha20_k32()**, mostrada no Procedimento 60, que foi mapeada para a **chacha20()** segue a mesma estrutura e não apresenta grandes diferenças. O espaço utilizado é maior porque as constantes são inicializadas neste ponto para serem utilizadas como parâmetro para inicialização. A chave é dividida em dois vetores $k0$ e $k1$, e o vetor n guarda informações sobre o *nonce* misturadas com as posições iniciais. A eficiência das operações é idêntica e não apresenta discrepância em relação à implementação proposta nesta dissertação.

```
1  procedure Chacha20_setup_seq(int k0[4], int k1[4], int filler[4],
2                               int n[4], int seq[16])
3      seq[0] += filler[0]
4      iterate int i = 0 by 1 to 3
5      seq[i+1] += k0[i]
6      end
7      seq[5] += filler[1]
8      iterate int i = 0 by 1 to 3
9      seq[i+6] += n[i]
10     end
11     seq[10] += filler[2]
12     iterate int i = 0 by 1 to 3
13     seq[i+11] += k1[i]
14     end
15     seq[15] += filler[3]
```

Procedimento 61: Função da implementação DIKU que executa o *setup*.

A segunda função **Chacha20_setup_seq()**, apresentada no Procedimento 61, organiza a matriz seguindo a definição do Salsa, que difere do algoritmo ChaCha como visto

anteriormente na Seção 4.3.1. Essa função foi mapeada para **chacha_setup()** onde os valores de *sigma* são inicializados e ocupam o espaço temporário de 256 *bits*. Ambas as implementações fazem uso de 16 somas, e portanto apresentam o mesmo desempenho.

```
1  procedure Chacha20(int in[16], int out[16])
2      iterate int i = 0 by 1 to 15
3          out[i] += in[i]
4      end
5      iterate int i = 0 by 1 to 9
6          call doubleround(out)
7      end
8      iterate int i = 0 by 1 to 15
9          out[i] += in[i]
10     end
```

Procedimento 62: Função *core* da implementação DIKU.

As funções **chacha20_core()** e **quarter_round()** apresentadas anteriormente são equivalentes às três funções da implementação DIKU denominadas **Chacha20()**, **doubleround()** e **quarterround()**. Vale relembrar que o *core* da implementação DIKU, mostrado no Procedimento 62, não realiza as operações modulares após a execução dos *rounds*, o que resulta em uma modificação do algoritmo. As modificações sobre as linhas e colunas são aninhadas na função **doubleround()** de forma que a função principal apenas executa 10 iterações e soma os valores de entrada e saída. A implementação proposta nesta dissertação realiza os 20 *rounds* fazendo 10 iterações considerando as linhas e diagonais, que é o equivalente a 10 *double rounds*, e ao fim das iterações a soma modular é realizada. Antes das iterações não é necessário utilizarmos a soma modular pois estamos considerando que os valores de entrada são válidos e portanto atendem aos requisitos da função.

A função **doubleround()**, mostrada nos Procedimentos 63 e 64, executa 64 *swaps* que podem ser simplificados para uma chamada de função como mostrada na implementação proposta, enquanto que a função **quarterround()**, da implementação DIKU e mostrada no Procedimento 65, se assemelha à função **quarter_round()** da implementação proposta. A principal diferença é que as operações modulares também não são feitas, e divergem da abordagem explicitada no algoritmo, devido as rotações serem feitas seguindo a lógica do ChaCha, porém utilizando a estrutura do algoritmo Salsa. Outra diferença existente entre as duas funções é na assinatura do procedimento, que não influencia no desempenho; enquanto a função proposta nesta dissertação usa quatro variáveis a sua equivalente (implementação DIKU) usa diretamente o vetor de estados. Apesar da

```
1  procedure doubleround(int seq[16])
2      local int tmp_seq[4]
3      tmp_seq[0] <=> seq[0]
4      tmp_seq[1] <=> seq[4]
5      tmp_seq[2] <=> seq[8]
6      tmp_seq[3] <=> seq[12]
7      call quarterround(tmp_seq)
8      tmp_seq[0] <=> seq[0]
9      tmp_seq[1] <=> seq[4]
10     tmp_seq[2] <=> seq[8]
11     tmp_seq[3] <=> seq[12]
12     tmp_seq[0] <=> seq[1]
13     tmp_seq[1] <=> seq[5]
14     tmp_seq[2] <=> seq[9]
15     tmp_seq[3] <=> seq[13]
16     call quarterround(tmp_seq)
17     tmp_seq[0] <=> seq[1]
18     tmp_seq[1] <=> seq[5]
19     tmp_seq[2] <=> seq[9]
20     tmp_seq[3] <=> seq[13]
21     tmp_seq[0] <=> seq[2]
22     tmp_seq[1] <=> seq[6]
23     tmp_seq[2] <=> seq[10]
24     tmp_seq[3] <=> seq[14]
25     call quarterround(tmp_seq)
26     tmp_seq[0] <=> seq[2]
27     tmp_seq[1] <=> seq[6]
28     tmp_seq[2] <=> seq[10]
29     tmp_seq[3] <=> seq[14]
30     tmp_seq[0] <=> seq[3]
31     tmp_seq[1] <=> seq[7]
32     tmp_seq[2] <=> seq[11]
33     tmp_seq[3] <=> seq[15]
34     call quarterround(tmp_seq)
35     tmp_seq[0] <=> seq[3]
36     tmp_seq[1] <=> seq[7]
37     tmp_seq[2] <=> seq[11]
38     tmp_seq[3] <=> seq[15]
```

Procedimento 63: Função *double-round* da implementação DIKU.

função implementada nesta dissertação usar mais parâmetros, esses são os valores diretos dos vetores e que sendo assim não ocupam um espaço a mais na memória se forem utilizados por referência em vez dos valores serem copiados, modificados e substituídos.

Sendo assim, o *core* da implementação DIKU realiza 74 somas, sendo 32 para mo-

```
39      tmp_seq[0] <=> seq[0]
40      tmp_seq[1] <=> seq[5]
41      tmp_seq[2] <=> seq[10]
42      tmp_seq[3] <=> seq[15]
43      call quarterround(tmp_seq)
44      tmp_seq[0] <=> seq[0]
45      tmp_seq[1] <=> seq[5]
46      tmp_seq[2] <=> seq[10]
47      tmp_seq[3] <=> seq[15]
48      tmp_seq[0] <=> seq[1]
49      tmp_seq[1] <=> seq[6]
50      tmp_seq[2] <=> seq[11]
51      tmp_seq[3] <=> seq[12]
52      call quarterround(tmp_seq)
53      tmp_seq[0] <=> seq[1]
54      tmp_seq[1] <=> seq[6]
55      tmp_seq[2] <=> seq[11]
56      tmp_seq[3] <=> seq[12]
57      tmp_seq[0] <=> seq[2]
58      tmp_seq[1] <=> seq[7]
59      tmp_seq[2] <=> seq[8]
60      tmp_seq[3] <=> seq[13]
61      call quarterround(tmp_seq)
62      tmp_seq[0] <=> seq[2]
63      tmp_seq[1] <=> seq[7]
64      tmp_seq[2] <=> seq[8]
65      tmp_seq[3] <=> seq[13]
66      tmp_seq[0] <=> seq[3]
67      tmp_seq[1] <=> seq[4]
68      tmp_seq[2] <=> seq[9]
69      tmp_seq[3] <=> seq[14]
70      call quarterround(tmp_seq)
71      tmp_seq[0] <=> seq[3]
72      tmp_seq[1] <=> seq[4]
73      tmp_seq[2] <=> seq[9]
74      tmp_seq[3] <=> seq[14]
75      delocal int tmp_seq[4]
```

Procedimento 64: Continuação da função *double-round* da implementação DIKU .

dificar os vetores de entrada, 10 para contagem das rodadas e mais 32 para adicionar o resultado ao vetor de saída. Além disso são utilizados 64 *swaps* na função **double-round()** somadas às oito chamadas de função **quarterround()**. Como as operações do *quarter round* envolvem a operação de rotação, que analisaremos mais à frente, iremos dizer neste momento que a soma modular feita na implementação proposta possui, po-

```
1  procedure quarterround(int seq[4])
2      seq[0] += seq[1]
3      seq[3] ^= seq[0]
4      call rotate_left(seq[3], 16)
5      seq[2] += seq[3]
6      seq[1] ^= seq[2]
7      call rotate_left(seq[1], 12)
8      seq[0] += seq[1]
9      seq[3] ^= seq[0]
10     call rotate_left(seq[3], 8)
11     seq[2] += seq[3]
12     seq[1] ^= seq[2]
13     call rotate_left(seq[1], 7)
```

Procedimento 65: Função *quarter-round* da implementação DIKU.

tencialmente, um dobro na quantidade de operações feitas, enquanto que a quantidade de XORs e rotações se mantém a mesma. Como são feitas 4 operações de soma, a implementação proposta usa cerca de 8 operações enquanto a de exemplo (DIKU) usa a metade, ao custo da modificação do algoritmo. A implementação proposta também utiliza 2 *shifts* que servem para realizar a exponenciação 2^{32} e fazer as operações modulares no *core* e no *quarter round*.

Em relação à função de rotação, a implementação genérica proposta pode ser vista na Seção 3.2.5 e faz uso de 14 somas, 8 subtrações, 5 multiplicações, 4 divisões, 10 *shifts*, 1 *swap* e 1 módulo, enquanto a implementação apresentada no código de exemplo (DIKU), que pode ser visualizada no Procedimento 66, usa 4 somas, 13 subtrações, 4 multiplicações, 2 divisões, 2 módulos, 4 *ANDs*, 2 *ORs* e 10 exponenciações.

```
1  procedure rotate_left(int x, int r)
2      local int rm = r % 32
3      local int temp_hi = x & (((2**rm)-1)*(2**(32-rm)))
4      local int temp_lo = x & ((2**(32-rm))-1)
5
6      x -= (temp_hi | temp_lo)
7      x += ((temp_hi / (2**(32-rm))) | (temp_lo * (2 ** rm)))
8
9      delocal int temp_lo = (x & (((2**(32-rm))-1)*(2**rm)))/(2**rm)
10     delocal int temp_hi = (x & ((2**rm)-1))*(2**(32-rm))
11     delocal int rm = r % 32
```

Procedimento 66: Função de rotação à esquerda da implementação DIKU.

De conhecimento destes fatos temos que, somando todas as operações e agrupando-as

por categoria como mostra a Tabela 4.10, podemos ter uma visão mais clara do impacto que a função de rotação tem sobre ambas implementações. Como a função de rotação da implementação DIKU utiliza diversas exponenciações e operadores binários (AND e OR), essa acaba tendo um desempenho considerado pior usando como base as informações introduzidas no inicio desta seção. Analisando as informações abaixo podemos perceber que a implementação proposta nesta dissertação apresenta um desempenho melhor, possui um código mais enxuto e claro. Grande parte do desempenho superior se deve à quantidade de exponenciações feitas na função de rotação da implementação DIKU. Note também que a abordagem de *swaps* que a implementação DIKU utiliza interfere pouco, porém representa 640 operações que poderiam ser economizadas através do uso de funções ou reutilização do próprio vetor de estados.

Tabela 4.10: Comparação da quantidade de operações feitas na implementação proposta e na DIKU.

	Proposta				Implementação DIKU				
	Setup	Core	Quarter	Rotate	Setup	Core	DR	QR	Rotate
Soma e subtração	32	7771	96	22	32	5834	576	72	17
Multiplicação e divisão	0	2880	36	9	0	1920	192	24	6
Exponenciação	0	0	0	0	0	3200	320	40	10
Módulo	0	320	4	1	0	640	64	8	2
Shift	0	3202	40	10	0	0	0	0	0
Swap	0	320	4	1	0	640	64	0	0
AND	0	0	0	0	0	1280	128	16	4
OR	0	0	0	0	0	640	64	8	2
XOR	0	320	4	0	0	320	32	4	0

Vale ressaltar ainda que as operações modulares não adicionaram muitas operações ao código geral sendo 336 de 7771, representando cerca de 4%, das operações de soma e subtração totais. A quantidade de operações de *shift* utilizadas para obter o valor do módulo, que é utilizado na operação de soma modular, foi de apenas 2 de 3202. As 3200 operações de *shift* restante foram devido as rotações utilizadas. A quantidade de XOR foi a mesmo em ambos os casos, visto que não há como reduzi-las e não são utilizadas em nenhuma outra parte do código. Note que a função de *setup* apresenta 32 somas nas duas abordagens analisadas. Isso ocorre porque são consideradas as duas chamadas da função (*call* e *uncall*), resultando em duas operações que gastam 16 somas cada.

Capítulo 5

Conclusão

A biblioteca proposta apresenta diversas funções básicas implementadas de forma clara e intuitiva a fim de auxiliar novos programadores e pesquisadores que se interessem por essa área de computação reversível. As implementações criptográficas mostram que diversos algoritmos simétricos podem ser implementados de forma reversível sem grandes complicações, ao contrário do que se possa imaginar. Porém vale ressaltar que funções *hash*, por exemplo, sofrem perda de informação e são irreversíveis por natureza. Uma análise melhor acerca desse tipo de função deverá ser feita no futuro, assim como o estudo de criptografia de chave pública. Os procedimentos apresentados ao longo desta dissertação podem auxiliar a implementação do protocolo Diffie-Hellman. Em uma versão futura, testes de primalidade e fatoração mais eficientes serão implementados a fim de possibilitar a implementação também de outros algoritmos de chave pública como RSA.

Apesar da linguagem de programação utilizada nesta dissertação ser antiga, e em tempos recentes estar sendo "resgatada" pela Universidade de Copenhague, existem pouquíssimas implementações e notas a respeito da mesma. O material escasso pode ter desencorajado pesquisadores ao longo do tempo, e encorajado outros que buscam expandir o conhecimento e ajudar a área a se desenvolver. Com essa dissertação esperamos que possamos contribuir, mesmo que minimamente, com o avanço dessa linguagem e novas pessoas venham a se interessar por computação reversível. Em trabalhos futuros queremos ainda modificar a linguagem para que essa possua suporte à rede, disco, importação de módulos e consiga lidar com estruturas mais complexas como listas e árvores binárias. Pretendemos também adicionar outros tipos primitivos como *strings* (variáveis de texto) e *float* (variáveis de ponto flutuante).

Ainda sobre trabalhos futuros podemos analisar o impacto causado se computadores totalmente reversíveis fossem utilizados em sistemas que recompensam os usuários pelo

trabalho feito ou que gastam muita energia. Em outras palavras, uma análise deve ser feita acerca dos sistemas descentralizados de criptomoedas (rede Bitcoin, por exemplo), sistemas de alto desempenho e o custo energético no gerenciamento, e manutenção, de servidores *high-end*.

Todo o conteúdo desenvolvido ao longo deste trabalho está disponibilizado no GitHub no link `https://github.com/raphaelbernardino/Kairos` e pode ser acessado de forma gratuita. Espera-se que com o auxilio da comunidade a biblioteca seja aprimorada e otimizada.

Referências

[1] Axelsen, H. B. Clean translation of an imperative reversible programming language. In *International Conference on Compiler Construction* (2011), Springer, pp. 144–163.

[2] Axelsen, H. B.; Glück, R. Reversible representation and manipulation of constructor terms in the heap. In *International Conference on Reversible Computation* (2013), Springer, pp. 96–109.

[3] Baker, H. G. Nreversal of fortune—the thermodynamics of garbage collection. In *Memory management*. Springer, 1992, pp. 507–524.

[4] Bennett, C. H. Logical reversibility of computation. *IBM journal of Research and Development 17*, 6 (1973), 525–532.

[5] Bernstein, D. J. Chacha, a variant of salsa20. In *Workshop Record of SASC* (2008), vol. 8, pp. 3–5.

[6] Bernstein, D. J. The salsa20 family of stream ciphers. In *New stream cipher designs*. Springer, 2008, pp. 84–97.

[7] Bérut, A.; Arakelyan, A.; Petrosyan, A.; Ciliberto, S.; Dillenschneider, R.; Lutz, E. Experimental verification of landauer's principle linking information and thermodynamics. *Nature 483*, 7388 (2012), 187.

[8] Brown, R. Dieharder test suite, 2009.

[9] Brum, V. d. C. Compilador para linguagem reversível Janus. trabalho de conclusão de curso de bacharelado. Instituto de Computação, Universidade Federal Fluminense., 2017.

[10] C++. Documentação da biblioteca random do c++, 2018. Disponível em `http://www.cplusplus.com/reference/random/mersenne_twister_engine/`. Consultado em 28/01/2019.

[11] Cloudflare. Why some cryptographic keys are much smaller than others, 2013. Disponível em `https://blog.cloudflare.com/why-are-some-keys-small/`. Consultado em 28/01/2019.

[12] Cloudflare. Do the chacha: better mobile performance with cryptography, 2015. Disponível em `https://blog.cloudflare.com/do-the-chacha-better-mobile-performance-with-cryptography/`. Consultado em 28/01/2019.

[13] Crypto++. Crypto++ 5.6.0 benchmarks for amd processors, 2009. Disponível em https://www.cryptopp.com/benchmarks-amd64.html. Consultado em 28/01/2019.

[14] Crypto++. Crypto++ 5.6.0 benchmarks for intel processors, 2009. Disponível em https://www.cryptopp.com/benchmarks.html. Consultado em 28/01/2019.

[15] Daemen, J.; Rijmen, V. *The Design of Rijndael: AES-The Advanced Encryption Standard*. Springer Science & Business Media, 2002.

[16] de Vries, A. Bitcoin's growing energy problem. *Joule 2*, 5 (2018), 801–805.

[17] Digiconomist. Bitcoin energy consumption index, 2018. Disponível em https://digiconomist.net/bitcoin-energy-consumption. Consultado em 28/01/2019.

[18] Egner, S.; Püschel, M. Solving puzzles related to permutation groups. In *Proceedings of the 1998 international symposium on Symbolic and algebraic computation* (1998), ACM, pp. 186–193.

[19] Elitefixtures. Bitcoin mining costs throughout the world, 2018. Disponível em https://www.elitefixtures.com/blog/post/2683/bitcoin-mining-costs-by-country/. Consultado em 28/01/2019.

[20] Entacher, K. Bad subsequences of well-known linear congruential pseudorandom number generators. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998), 61–70.

[21] Firasta, N.; Buxton, M.; Jinbo, P.; Nasri, K.; Kuo, S. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper* (2008).

[22] Fowler, A. G.; Mariantoni, M.; Martinis, J. M.; Cleland, A. N. Surface codes: Towards practical large-scale quantum computation. *Physical Review A 86*, 3 (2012), 032324.

[23] Fowler, A. G.; Stephens, A. M.; Groszkowski, P. High-threshold universal quantum computation on the surface code. *Physical Review A 80*, 5 (2009), 052312.

[24] Frank, M. P. The R programming language and compiler. Tech. rep., MIT Reversible Computing Project Memo, 1997. Disponível em https://web.archive.org/web/20031021044014/https://www.cise.ufl.edu/~mpf/rc/memos/M08/doc.ps. Consultado em 28/01/2019.

[25] Frank, M. P. Are "reversible"computers more energy efficient, faster?, 2004. Disponível em https://www.eetimes.com/author.asp?section_id=36&doc_id=1265588. Consultado em 28/01/2019.

[26] Gaeini, A.; Mirghadri, A.; Jandaghi, G.; Keshavarzi, B. Comparing some pseudo-random number generators and cryptography algorithms using a general evaluation pattern. *Int. J. Inf. Technol. Comput. Sci.(IJITCS) 8*, 9 (2016), 25–31.

[27] Google. Speeding up and strengthening https connections for chrome on android, 2014. Disponível em https://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html. Consultado em 28/01/2019.

[28] Grassl, M.; Langenberg, B.; Roetteler, M.; Steinwandt, R. Applying grover's algorithm to aes: quantum resource estimates. In *International Workshop on Post-Quantum Cryptography* (2016), Springer, pp. 29–43.

[29] Gries, D. Inverting programs. In *The Science of Programming*. Springer, 1981, pp. 265–274.

[30] Grover, L. K. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing* (1996), ACM, pp. 212–219.

[31] Hall, J. S. A reversible instruction set architecture and algorithms. In *Physics and Computation, 1994. PhysComp'94, Proceedings., Workshop on* (1994), IEEE, pp. 128–134.

[32] Haulund, T. Design and implementation of a reversible object-oriented programming language. *arXiv preprint arXiv:1707.07845* (2017).

[33] Hull, T. E.; Dobell, A. R. Random number generators. *SIAM review 4*, 3 (1962), 230–254.

[34] Internet Research Task Force (IRTF). Chacha20 and poly1305 for ietf protocols, 2015. Disponível em `https://tools.ietf.org/html/rfc7539`. Consultado em 28/01/2019.

[35] Jones, N. C. Logic synthesis for fault-tolerant quantum computers. *arXiv preprint arXiv:1310.7290* (2013).

[36] Knuth, D. E. Seminumerical algorithms. *The Art of Computer Programming. Volume 2* (1989), 213–217.

[37] Kowada, L. A. B. *Construção de algoritmos reversíveis e quânticos*. Tese de Doutorado, UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 2006.

[38] Landauer, R. Irreversibility and heat generation in the computing process. *IBM journal of research and development 5*, 3 (1961), 183–191.

[39] L'Ecuyer, P.; Simard, R. Testu01: Ac library for empirical testing of random number generators. *ACM Transactions on Mathematical Software (TOMS) 33*, 4 (2007), 22.

[40] Lehmer, D. see\proc. of 2nd symp. on large-scale digital calculating machinery, 1951.

[41] Lutz, C.; Derby, H. Janus: a time-reversible language. *Caltech class project* (1982).

[42] L'ecuyer, P. Tables of linear congruential generators of different sizes and good lattice structure. *Mathematics of Computation of the American Mathematical Society 68*, 225 (1999), 249–260.

[43] Marsaglia, G. Diehard, a battery of tests for random number generators. *CD-ROM, Department of Statistics and Supercomputer Computations Research Institute, Florida State University (available at http://stat. fsu. edu/Ägeo)* (1995).

[44] Matsumoto, M.; Nishimura, T. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS) 8*, 1 (1998), 3–30.

[45] Mu, S.-C.; Hu, Z.; Takeichi, M. An algebraic approach to bi-directional updating. In *Asian Symposium on Programming Languages and Systems* (2004), Springer, pp. 2–20.

[46] Mu, S.-C.; Hu, Z.; Takeichi, M. An injective language for reversible computation. In *International Conference on Mathematics of Program Construction* (2004), Springer, pp. 289–313.

[47] O'Neill, M. E. Pcg: A family of simple fast space-efficient statistically good algorithms for random number generation. *ACM Transactions on Mathematical Software* (2014).

[48] Payne, W.; Rabung, J. R.; Bogyo, T. Coding the lehmer pseudo-random number generator. *Communications of the ACM 12*, 2 (1969), 85–86.

[49] Perumalla, K. S. *Introduction to reversible computing.* CRC Press, 2013.

[50] Python 3.7. Documentação da biblioteca random do python versão 3.7, 2018. Disponível em `https://docs.python.org/3/library/random.html`. Consultado em 28/01/2019.

[51] Reichardt, B. W. Quantum universality by state distillation. *arXiv preprint quant-ph/0608085* (2006).

[52] Ressler, A. L. Practical circuits using conservative reversible logic. trabalho de conclusão de curso de bacharelado, 1979.

[53] Ressler, A. L. *The design of a conservative logic computer and a graphical editor simulator.* Tese de Doutorado, Massachusetts Institute of Technology, 1981.

[54] Rotenberg, A. A new pseudo-random number generator. *J. ACM 7*, 1 (Jan. 1960), 75–77.

[55] Route, M. Radio-flaring ultracool dwarf population synthesis. *The Astrophysical Journal 845*, 1 (2017), 66.

[56] Steane, A. M. Overhead and noise threshold of fault-tolerant quantum error correction. *Physical Review A 68*, 4 (2003), 042322.

[57] Táborskỳ, D.; Larsen, K. F.; Thomsen, M. K. Encryption and reversible computations. In *International Conference on Reversible Computation* (2018), Springer, pp. 331–338.

[58] Tausworthe, R. C. Random numbers generated by linear recurrence modulo two. *Mathematics of Computation 19*, 90 (1965), 201–209.

[59] Thomsen, M. K.; Axelsen, H. B.; Glück, R. A reversible processor architecture and its reversible logic design. In *International Workshop on Reversible Computation* (2011), Springer, pp. 30–42.

[60] U.S Energy Information Administration. Consumo energético de 2017 informado pela EIA, 2017. Disponível em `https://www.eia.gov/energyexplained/index.php?page=electricity_use`. Consultado em 28/01/2019.

[61] Vedral, V.; Barenco, A.; Ekert, A. Quantum networks for elementary arithmetic operations. *Physical Review A 54*, 1 (1996), 147.

[62] Vieri, C. J. *Pendulum–a reversible computer architecture*. Tese de Doutorado, Massachusetts Institute of Technology, 1993.

[63] Yokoyama, T.; Axelsen, H. B.; Glück, R. Principles of a reversible programming language. In *Proceedings of the 5th conference on Computing frontiers* (2008), ACM, pp. 43–54.

[64] Yokoyama, T.; Axelsen, H. B.; Glück, R. Towards a reversible functional language. In *International Workshop on Reversible Computation* (2011), Springer, pp. 14–29.

[65] Yokoyama, T.; Glück, R. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2007), ACM, pp. 144–153.

APÊNDICE A - Tabelas AES

Abaixo são mostradas as tabelas do algoritmo AES, mostrado na Seção 4.2, que, por razões estéticas, foram omitidas ao longo da dissertação. As tabelas apresentadas são a tabela RCON, que guardam as constante de *round*, e as do corpo finito de Galois 2, 3, 9, 11, 13 e 14 que são utilizadas no método MixColumns e MixColumnsInv, descritos na mesma seção.

A.1 Tabela RCON completa

A tabela omitida na Seção 4.2.1, por simplicidade, contendo as constantes de *round* é demonstrada abaixo. Essa tabela é utilizada na função AddRoundKey do algoritmo AES, descrita na mesma seção.

Tabela A.1: Tabela completa de constantes de rodada.

0x8d	**0x01**	**0x02**	**0x04**	**0x08**	**0x10**	**0x20**	**0x40**	**0x80**	**0x1b**	**0x36**	**0x6c**	**0xd8**	**0xab**	0x4d	0x9a
0x2f	0x5e	0xbc	0x63	0xc6	0x97	0x35	0x6a	0xd4	0xb3	0x7d	0xfa	0xef	0xc5	0x91	0x39
0x72	0xe4	0xd3	0xbd	0x61	0xc2	0x9f	0x25	0x4a	0x94	0x33	0x66	0xcc	0x83	0x1d	0x3a
0x74	0xe8	0xcb	0x8d	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1b	0x36	0x6c	0xd8
0xab	0x4d	0x9a	0x2f	0x5e	0xbc	0x63	0xc6	0x97	0x35	0x6a	0xd4	0xb3	0x7d	0xfa	0xef
0xc5	0x91	0x39	0x72	0xe4	0xd3	0xbd	0x61	0xc2	0x9f	0x25	0x4a	0x94	0x33	0x66	0xcc
0x83	0x1d	0x3a	0x74	0xe8	0xcb	0x8d	0x01	0x02	0x04	0x08	0x10	0x20	0x40	0x80	0x1b
0x36	0x6c	0xd8	0xab	0x4d	0x9a	0x2f	0x5e	0xbc	0x63	0xc6	0x97	0x35	0x6a	0xd4	0xb3
0x7d	0xfa	0xef	0xc5	0x91	0x39	0x72	0xe4	0xd3	0xbd	0x61	0xc2	0x9f	0x25	0x4a	0x94
0x33	0x66	0xcc	0x83	0x1d	0x3a	0x74	0xe8	0xcb	0x8d	0x01	0x02	0x04	0x08	0x10	0x20
0x40	0x80	0x1b	0x36	0x6c	0xd8	0xab	0x4d	0x9a	0x2f	0x5e	0xbc	0x63	0xc6	0x97	0x35
0x6a	0xd4	0xb3	0x7d	0xfa	0xef	0xc5	0x91	0x39	0x72	0xe4	0xd3	0xbd	0x61	0xc2	0x9f
0x25	0x4a	0x94	0x33	0x66	0xcc	0x83	0x1d	0x3a	0x74	0xe8	0xcb	0x8d	0x01	0x02	0x04
0x08	0x10	0x20	0x40	0x80	0x1b	0x36	0x6c	0xd8	0xab	0x4d	0x9a	0x2f	0x5e	0xbc	0x63
0xc6	0x97	0x35	0x6a	0xd4	0xb3	0x7d	0xfa	0xef	0xc5	0x91	0x39	0x72	0xe4	0xd3	0xbd
0x61	0xc2	0x9f	0x25	0x4a	0x94	0x33	0x66	0xcc	0x83	0x1d	0x3a	0x74	0xe8	0xcb	0x8d

A.2 Tabelas do Corpo Finito de Galois

As seis tabelas do corpo de Galois omitidas na Seção 4.2.1 são demonstradas abaixo. Vale notar que essas tabelas foram pré-calculadas e utilizadas na implementação feita. A implementação pode ser visualizada na Seção 4.2.2.

Tabela A.2: Tabela completa do corpo de Galois GF02.

0x00	0x02	0x04	0x06	0x08	0x0a	0x0c	0x0e	0x10	0x12	0x14	0x16	0x18	0x1a	0x1c	0x1e
0x20	0x22	0x24	0x26	0x28	0x2a	0x2c	0x2e	0x30	0x32	0x34	0x36	0x38	0x3a	0x3c	0x3e
0x40	0x42	0x44	0x46	0x48	0x4a	0x4c	0x4e	0x50	0x52	0x54	0x56	0x58	0x5a	0x5c	0x5e
0x60	0x62	0x64	0x66	0x68	0x6a	0x6c	0x6e	0x70	0x72	0x74	0x76	0x78	0x7a	0x7c	0x7e
0x80	0x82	0x84	0x86	0x88	0x8a	0x8c	0x8e	0x90	0x92	0x94	0x96	0x98	0x9a	0x9c	0x9e
0xa0	0xa2	0xa4	0xa6	0xa8	0xaa	0xac	0xae	0xb0	0xb2	0xb4	0xb6	0xb8	0xba	0xbc	0xbe
0xc0	0xc2	0xc4	0xc6	0xc8	0xca	0xcc	0xce	0xd0	0xd2	0xd4	0xd6	0xd8	0xda	0xdc	0xde
0xe0	0xe2	0xe4	0xe6	0xe8	0xea	0xec	0xee	0xf0	0xf2	0xf4	0xf6	0xf8	0xfa	0xfc	0xfe
0x1b	0x19	0x1f	0x1d	0x13	0x11	0x17	0x15	0x0b	0x09	0x0f	0x0d	0x03	0x01	0x07	0x05
0x3b	0x39	0x3f	0x3d	0x33	0x31	0x37	0x35	0x2b	0x29	0x2f	0x2d	0x23	0x21	0x27	0x25
0x5b	0x59	0x5f	0x5d	0x53	0x51	0x57	0x55	0x4b	0x49	0x4f	0x4d	0x43	0x41	0x47	0x45
0x7b	0x79	0x7f	0x7d	0x73	0x71	0x77	0x75	0x6b	0x69	0x6f	0x6d	0x63	0x61	0x67	0x65
0x9b	0x99	0x9f	0x9d	0x93	0x91	0x97	0x95	0x8b	0x89	0x8f	0x8d	0x83	0x81	0x87	0x85
0xbb	0xb9	0xbf	0xbd	0xb3	0xb1	0xb7	0xb5	0xab	0xa9	0xaf	0xad	0xa3	0xa1	0xa7	0xa5
0xdb	0xd9	0xdf	0xdd	0xd3	0xd1	0xd7	0xd5	0xcb	0xc9	0xcf	0xcd	0xc3	0xc1	0xc7	0xc5
0xfb	0xf9	0xff	0xfd	0xf3	0xf1	0xf7	0xf5	0xeb	0xe9	0xef	0xed	0xe3	0xe1	0xe7	0xe5

Tabela A.3: Tabela completa do corpo de Galois GF03.

0x00	0x03	0x06	0x05	0x0c	0x0f	0x0a	0x09	0x18	0x1b	0x1e	0x1d	0x14	0x17	0x12	0x11
0x30	0x33	0x36	0x35	0x3c	0x3f	0x3a	0x39	0x28	0x2b	0x2e	0x2d	0x24	0x27	0x22	0x21
0x60	0x63	0x66	0x65	0x6c	0x6f	0x6a	0x69	0x78	0x7b	0x7e	0x7d	0x74	0x77	0x72	0x71
0x50	0x53	0x56	0x55	0x5c	0x5f	0x5a	0x59	0x48	0x4b	0x4e	0x4d	0x44	0x47	0x42	0x41
0xc0	0xc3	0xc6	0xc5	0xcc	0xcf	0xca	0xc9	0xd8	0xdb	0xde	0xdd	0xd4	0xd7	0xd2	0xd1
0xf0	0xf3	0xf6	0xf5	0xfc	0xff	0xfa	0xf9	0xe8	0xeb	0xee	0xed	0xe4	0xe7	0xe2	0xe1
0xa0	0xa3	0xa6	0xa5	0xac	0xaf	0xaa	0xa9	0xb8	0xbb	0xbe	0xbd	0xb4	0xb7	0xb2	0xb1
0x90	0x93	0x96	0x95	0x9c	0x9f	0x9a	0x99	0x88	0x8b	0x8e	0x8d	0x84	0x87	0x82	0x81
0x9b	0x98	0x9d	0x9e	0x97	0x94	0x91	0x92	0x83	0x80	0x85	0x86	0x8f	0x8c	0x89	0x8a
0xab	0xa8	0xad	0xae	0xa7	0xa4	0xa1	0xa2	0xb3	0xb0	0xb5	0xb6	0xbf	0xbc	0xb9	0xba
0xfb	0xf8	0xfd	0xfe	0xf7	0xf4	0xf1	0xf2	0xe3	0xe0	0xe5	0xe6	0xef	0xec	0xe9	0xea
0xcb	0xc8	0xcd	0xce	0xc7	0xc4	0xc1	0xc2	0xd3	0xd0	0xd5	0xd6	0xdf	0xdc	0xd9	0xda
0x5b	0x58	0x5d	0x5e	0x57	0x54	0x51	0x52	0x43	0x40	0x45	0x46	0x4f	0x4c	0x49	0x4a
0x6b	0x68	0x6d	0x6e	0x67	0x64	0x61	0x62	0x73	0x70	0x75	0x76	0x7f	0x7c	0x79	0x7a
0x3b	0x38	0x3d	0x3e	0x37	0x34	0x31	0x32	0x23	0x20	0x25	0x26	0x2f	0x2c	0x29	0x2a
0x0b	0x08	0x0d	0x0e	0x07	0x04	0x01	0x02	0x13	0x10	0x15	0x16	0x1f	0x1c	0x19	0x1a

Tabela A.4: Tabela completa do corpo de Galois GF09.

0x00	0x09	0x12	0x1b	0x24	0x2d	0x36	0x3f	0x48	0x41	0x5a	0x53	0x6c	0x65	0x7e	0x77
0x90	0x99	0x82	0x8b	0xb4	0xbd	0xa6	0xaf	0xd8	0xd1	0xca	0xc3	0xfc	0xf5	0xee	0xe7
0x3b	0x32	0x29	0x20	0x1f	0x16	0x0d	0x04	0x73	0x7a	0x61	0x68	0x57	0x5e	0x45	0x4c
0xab	0xa2	0xb9	0xb0	0x8f	0x86	0x9d	0x94	0xe3	0xea	0xf1	0xf8	0xc7	0xce	0xd5	0xdc
0x76	0x7f	0x64	0x6d	0x52	0x5b	0x40	0x49	0x3e	0x37	0x2c	0x25	0x1a	0x13	0x08	0x01
0xe6	0xef	0xf4	0xfd	0xc2	0xcb	0xd0	0xd9	0xae	0xa7	0xbc	0xb5	0x8a	0x83	0x98	0x91
0x4d	0x44	0x5f	0x56	0x69	0x60	0x7b	0x72	0x05	0x0c	0x17	0x1e	0x21	0x28	0x33	0x3a
0xdd	0xd4	0xcf	0xc6	0xf9	0xf0	0xeb	0xe2	0x95	0x9c	0x87	0x8e	0xb1	0xb8	0xa3	0xaa
0xec	0xe5	0xfe	0xf7	0xc8	0xc1	0xda	0xd3	0xa4	0xad	0xb6	0xbf	0x80	0x89	0x92	0x9b
0x7c	0x75	0x6e	0x67	0x58	0x51	0x4a	0x43	0x34	0x3d	0x26	0x2f	0x10	0x19	0x02	0x0b
0xd7	0xde	0xc5	0xcc	0xf3	0xfa	0xe1	0xe8	0x9f	0x96	0x8d	0x84	0xbb	0xb2	0xa9	0xa0
0x47	0x4e	0x55	0x5c	0x63	0x6a	0x71	0x78	0x0f	0x06	0x1d	0x14	0x2b	0x22	0x39	0x30
0x9a	0x93	0x88	0x81	0xbe	0xb7	0xac	0xa5	0xd2	0xdb	0xc0	0xc9	0xf6	0xff	0xe4	0xed
0x0a	0x03	0x18	0x11	0x2e	0x27	0x3c	0x35	0x42	0x4b	0x50	0x59	0x66	0x6f	0x74	0x7d
0xa1	0xa8	0xb3	0xba	0x85	0x8c	0x97	0x9e	0xe9	0xe0	0xfb	0xf2	0xcd	0xc4	0xdf	0xd6
0x31	0x38	0x23	0x2a	0x15	0x1c	0x07	0x0e	0x79	0x70	0x6b	0x62	0x5d	0x54	0x4f	0x46

Tabela A.5: Tabela completa do corpo de Galois GF11.

0x00	0x0b	0x16	0x1d	0x2c	0x27	0x3a	0x31	0x58	0x53	0x4e	0x45	0x74	0x7f	0x62	0x69
0xb0	0xbb	0xa6	0xad	0x9c	0x97	0x8a	0x81	0xe8	0xe3	0xfe	0xf5	0xc4	0xcf	0xd2	0xd9
0x7b	0x70	0x6d	0x66	0x57	0x5c	0x41	0x4a	0x23	0x28	0x35	0x3e	0x0f	0x04	0x19	0x12
0xcb	0xc0	0xdd	0xd6	0xe7	0xec	0xf1	0xfa	0x93	0x98	0x85	0x8e	0xbf	0xb4	0xa9	0xa2
0xf6	0xfd	0xe0	0xeb	0xda	0xd1	0xcc	0xc7	0xae	0xa5	0xb8	0xb3	0x82	0x89	0x94	0x9f
0x46	0x4d	0x50	0x5b	0x6a	0x61	0x7c	0x77	0x1e	0x15	0x08	0x03	0x32	0x39	0x24	0x2f
0x8d	0x86	0x9b	0x90	0xa1	0xaa	0xb7	0xbc	0xd5	0xde	0xc3	0xc8	0xf9	0xf2	0xef	0xe4
0x3d	0x36	0x2b	0x20	0x11	0x1a	0x07	0x0c	0x65	0x6e	0x73	0x78	0x49	0x42	0x5f	0x54
0xf7	0xfc	0xe1	0xea	0xdb	0xd0	0xcd	0xc6	0xaf	0xa4	0xb9	0xb2	0x83	0x88	0x95	0x9e
0x47	0x4c	0x51	0x5a	0x6b	0x60	0x7d	0x76	0x1f	0x14	0x09	0x02	0x33	0x38	0x25	0x2e
0x8c	0x87	0x9a	0x91	0xa0	0xab	0xb6	0xbd	0xd4	0xdf	0xc2	0xc9	0xf8	0xf3	0xee	0xe5
0x3c	0x37	0x2a	0x21	0x10	0x1b	0x06	0x0d	0x64	0x6f	0x72	0x79	0x48	0x43	0x5e	0x55
0x01	0x0a	0x17	0x1c	0x2d	0x26	0x3b	0x30	0x59	0x52	0x4f	0x44	0x75	0x7e	0x63	0x68
0xb1	0xba	0xa7	0xac	0x9d	0x96	0x8b	0x80	0xe9	0xe2	0xff	0xf4	0xc5	0xce	0xd3	0xd8
0x7a	0x71	0x6c	0x67	0x56	0x5d	0x40	0x4b	0x22	0x29	0x34	0x3f	0x0e	0x05	0x18	0x13
0xca	0xc1	0xdc	0xd7	0xe6	0xed	0xf0	0xfb	0x92	0x99	0x84	0x8f	0xbe	0xb5	0xa8	0xa3

Tabela A.6: Tabela completa do corpo de Galois GF13.

0x00	0x0d	0x1a	0x17	0x34	0x39	0x2e	0x23	0x68	0x65	0x72	0x7f	0x5c	0x51	0x46	0x4b
0xd0	0xdd	0xca	0xc7	0xe4	0xe9	0xfe	0xf3	0xb8	0xb5	0xa2	0xaf	0x8c	0x81	0x96	0x9b
0xbb	0xb6	0xa1	0xac	0x8f	0x82	0x95	0x98	0xd3	0xde	0xc9	0xc4	0xe7	0xea	0xfd	0xf0
0x6b	0x66	0x71	0x7c	0x5f	0x52	0x45	0x48	0x03	0x0e	0x19	0x14	0x37	0x3a	0x2d	0x20
0x6d	0x60	0x77	0x7a	0x59	0x54	0x43	0x4e	0x05	0x08	0x1f	0x12	0x31	0x3c	0x2b	0x26
0xbd	0xb0	0xa7	0xaa	0x89	0x84	0x93	0x9e	0xd5	0xd8	0xcf	0xc2	0xe1	0xec	0xfb	0xf6
0xd6	0xdb	0xcc	0xc1	0xe2	0xef	0xf8	0xf5	0xbe	0xb3	0xa4	0xa9	0x8a	0x87	0x90	0x9d
0x06	0x0b	0x1c	0x11	0x32	0x3f	0x28	0x25	0x6e	0x63	0x74	0x79	0x5a	0x57	0x40	0x4d
0xda	0xd7	0xc0	0xcd	0xee	0xe3	0xf4	0xf9	0xb2	0xbf	0xa8	0xa5	0x86	0x8b	0x9c	0x91
0x0a	0x07	0x10	0x1d	0x3e	0x33	0x24	0x29	0x62	0x6f	0x78	0x75	0x56	0x5b	0x4c	0x41
0x61	0x6c	0x7b	0x76	0x55	0x58	0x4f	0x42	0x09	0x04	0x13	0x1e	0x3d	0x30	0x27	0x2a
0xb1	0xbc	0xab	0xa6	0x85	0x88	0x9f	0x92	0xd9	0xd4	0xc3	0xce	0xed	0xe0	0xf7	0xfa
0xb7	0xba	0xad	0xa0	0x83	0x8e	0x99	0x94	0xdf	0xd2	0xc5	0xc8	0xeb	0xe6	0xf1	0xfc
0x67	0x6a	0x7d	0x70	0x53	0x5e	0x49	0x44	0x0f	0x02	0x15	0x18	0x3b	0x36	0x21	0x2c
0x0c	0x01	0x16	0x1b	0x38	0x35	0x22	0x2f	0x64	0x69	0x7e	0x73	0x50	0x5d	0x4a	0x47
0xdc	0xd1	0xc6	0xcb	0xe8	0xe5	0xf2	0xff	0xb4	0xb9	0xae	0xa3	0x80	0x8d	0x9a	0x97

Tabela A.7: Tabela completa do corpo de Galois GF14.

0x00	0x0e	0x1c	0x12	0x38	0x36	0x24	0x2a	0x70	0x7e	0x6c	0x62	0x48	0x46	0x54	0x5a
0xe0	0xee	0xfc	0xf2	0xd8	0xd6	0xc4	0xca	0x90	0x9e	0x8c	0x82	0xa8	0xa6	0xb4	0xba
0xdb	0xd5	0xc7	0xc9	0xe3	0xed	0xff	0xf1	0xab	0xa5	0xb7	0xb9	0x93	0x9d	0x8f	0x81
0x3b	0x35	0x27	0x29	0x03	0x0d	0x1f	0x11	0x4b	0x45	0x57	0x59	0x73	0x7d	0x6f	0x61
0xad	0xa3	0xb1	0xbf	0x95	0x9b	0x89	0x87	0xdd	0xd3	0xc1	0xcf	0xe5	0xeb	0xf9	0xf7
0x4d	0x43	0x51	0x5f	0x75	0x7b	0x69	0x67	0x3d	0x33	0x21	0x2f	0x05	0x0b	0x19	0x17
0x76	0x78	0x6a	0x64	0x4e	0x40	0x52	0x5c	0x06	0x08	0x1a	0x14	0x3e	0x30	0x22	0x2c
0x96	0x98	0x8a	0x84	0xae	0xa0	0xb2	0xbc	0xe6	0xe8	0xfa	0xf4	0xde	0xd0	0xc2	0xcc
0x41	0x4f	0x5d	0x53	0x79	0x77	0x65	0x6b	0x31	0x3f	0x2d	0x23	0x09	0x07	0x15	0x1b
0xa1	0xaf	0xbd	0xb3	0x99	0x97	0x85	0x8b	0xd1	0xdf	0xcd	0xc3	0xe9	0xe7	0xf5	0xfb
0x9a	0x94	0x86	0x88	0xa2	0xac	0xbe	0xb0	0xea	0xe4	0xf6	0xf8	0xd2	0xdc	0xce	0xc0
0x7a	0x74	0x66	0x68	0x42	0x4c	0x5e	0x50	0x0a	0x04	0x16	0x18	0x32	0x3c	0x2e	0x20
0xec	0xe2	0xf0	0xfe	0xd4	0xda	0xc8	0xc6	0x9c	0x92	0x80	0x8e	0xa4	0xaa	0xb8	0xb6
0x0c	0x02	0x10	0x1e	0x34	0x3a	0x28	0x26	0x7c	0x72	0x60	0x6e	0x44	0x4a	0x58	0x56
0x37	0x39	0x2b	0x25	0x0f	0x01	0x13	0x1d	0x47	0x49	0x5b	0x55	0x7f	0x71	0x63	0x6d
0xd7	0xd9	0xcb	0xc5	0xef	0xe1	0xf3	0xfd	0xa7	0xa9	0xbb	0xb5	0x9f	0x91	0x83	0x8d

APÊNDICE B – Exemplo ChaCha da JanusP

B.1 Implementação do ChaCha20

A implementação de exemplo, proposta e disponibilizada na página do compilador da Universidade de Copenhague, se encontra abaixo. Essa mesma implementação foi utilizada na Seção 4.3.3 para avaliar a abordagem proposta e descrita nesta dissertação. O código é disponibilizado abaixo apenas para prevenir que o mesmo se perca, ou seja modificado.

```
1  // see https://cr.yp.to/chacha/chacha-20080128.pdf
2
3  procedure rotate_left(int x, int r)
4      local int rm = r % 32
5      local int temp_hi = x & (((2**rm)-1)*(2**(32-rm)))
6      local int temp_lo = x & ((2**(32-rm))-1)
7      x -= (temp_hi | temp_lo)
8      x += ((temp_hi / (2**(32-rm))) | (temp_lo * (2 ** rm)))
9      delocal int temp_lo = (x & (((2**(32-rm))-1)*(2**rm))) / (2**rm)
10     delocal int temp_hi = (x & ((2**rm)-1)) * (2 ** (32-rm))
11     delocal int rm = r % 32
12
13
14 // This and the following procedure differ from Salsa
15 procedure quarterround(int seq[4])
16     seq[0] += seq[1]
17   seq[3] ^= seq[0]
18   call rotate_left(seq[3], 16)
19   seq[2] += seq[3]
20   seq[1] ^= seq[2]
```

```
21    call rotate_left(seq[1], 12)
22    seq[0] += seq[1]
23    seq[3] ^= seq[0]
24    call rotate_left(seq[3], 8)
25    seq[2] += seq[3]
26    seq[1] ^= seq[2]
27    call rotate_left(seq[1], 7)
28
29
30  procedure doubleround(int seq[16])
31    // would be nice if we could get rid of the temporary array, somehow
32    local int tmp_seq[4]
33    // first column
34    tmp_seq[0] <=> seq[0]
35    tmp_seq[1] <=> seq[4]
36    tmp_seq[2] <=> seq[8]
37    tmp_seq[3] <=> seq[12]
38    call quarterround(tmp_seq)
39    tmp_seq[0] <=> seq[0]
40    tmp_seq[1] <=> seq[4]
41    tmp_seq[2] <=> seq[8]
42    tmp_seq[3] <=> seq[12]
43    // second column
44    tmp_seq[0] <=> seq[1]
45    tmp_seq[1] <=> seq[5]
46    tmp_seq[2] <=> seq[9]
47    tmp_seq[3] <=> seq[13]
48    call quarterround(tmp_seq)
49    tmp_seq[0] <=> seq[1]
50    tmp_seq[1] <=> seq[5]
51    tmp_seq[2] <=> seq[9]
52    tmp_seq[3] <=> seq[13]
53    // third column
54    tmp_seq[0] <=> seq[2]
55    tmp_seq[1] <=> seq[6]
```

```
56    tmp_seq[2] <=> seq[10]
57    tmp_seq[3] <=> seq[14]
58    call quarterround(tmp_seq)
59    tmp_seq[0] <=> seq[2]
60    tmp_seq[1] <=> seq[6]
61    tmp_seq[2] <=> seq[10]
62    tmp_seq[3] <=> seq[14]
63    // fourth column
64    tmp_seq[0] <=> seq[3]
65    tmp_seq[1] <=> seq[7]
66    tmp_seq[2] <=> seq[11]
67    tmp_seq[3] <=> seq[15]
68    call quarterround(tmp_seq)
69    tmp_seq[0] <=> seq[3]
70    tmp_seq[1] <=> seq[7]
71    tmp_seq[2] <=> seq[11]
72    tmp_seq[3] <=> seq[15]
73    // first diagonal
74    tmp_seq[0] <=> seq[0]
75    tmp_seq[1] <=> seq[5]
76    tmp_seq[2] <=> seq[10]
77    tmp_seq[3] <=> seq[15]
78    call quarterround(tmp_seq)
79    tmp_seq[0] <=> seq[0]
80    tmp_seq[1] <=> seq[5]
81    tmp_seq[2] <=> seq[10]
82    tmp_seq[3] <=> seq[15]
83    // second diagonal
84    tmp_seq[0] <=> seq[1]
85    tmp_seq[1] <=> seq[6]
86    tmp_seq[2] <=> seq[11]
87    tmp_seq[3] <=> seq[12]
88    call quarterround(tmp_seq)
89    tmp_seq[0] <=> seq[1]
90    tmp_seq[1] <=> seq[6]
```

```
91    tmp_seq[2] <=> seq[11]
92    tmp_seq[3] <=> seq[12]
93    // third diagonal
94    tmp_seq[0] <=> seq[2]
95    tmp_seq[1] <=> seq[7]
96    tmp_seq[2] <=> seq[8]
97    tmp_seq[3] <=> seq[13]
98    call quarterround(tmp_seq)
99    tmp_seq[0] <=> seq[2]
100   tmp_seq[1] <=> seq[7]
101   tmp_seq[2] <=> seq[8]
102   tmp_seq[3] <=> seq[13]
103   // fourth diagonal
104   tmp_seq[0] <=> seq[3]
105   tmp_seq[1] <=> seq[4]
106   tmp_seq[2] <=> seq[9]
107   tmp_seq[3] <=> seq[14]
108   call quarterround(tmp_seq)
109   tmp_seq[0] <=> seq[3]
110   tmp_seq[1] <=> seq[4]
111   tmp_seq[2] <=> seq[9]
112   tmp_seq[3] <=> seq[14]
113   delocal int tmp_seq[4]
114
115
116 procedure Chacha20(int in[16], int out[16])
117   iterate int i = 0 by 1 to 15
118     out[i] += in[i]
119   end
120   iterate int i = 0 by 1 to 9
121     call doubleround(out)
122   end
123   iterate int i = 0 by 1 to 15
124     out[i] += in[i]
125   end
```

```
126
127
128 procedure Chacha20_setup_seq(int k0[4], int k1[4], int filler[4],
129                              int n[4], int seq[16])
130   seq[0] += filler[0]
131   iterate int i = 0 by 1 to 3
132     seq[i+1] += k0[i]
133   end
134   seq[5] += filler[1]
135   iterate int i = 0 by 1 to 3
136     seq[i+6] += n[i]
137   end
138   seq[10] += filler[2]
139   iterate int i = 0 by 1 to 3
140     seq[i+11] += k1[i]
141   end
142   seq[15] += filler[3]
143
144
145 procedure Chacha20_k32(int k0[4], int k1[4], int n[4], int out[16])
146   local int sigma[4] = {1634760805, 857760878, 2036477234, 1797285236}
147   local int seq[16]
148   call Chacha20_setup_seq(k0, k1, n, sigma, seq)
149   call Chacha20(seq, out)
150   uncall Chacha20_setup_seq(k0, k1, n, sigma, seq)
151   delocal int seq[16]
152   delocal int sigma[4] = {1634760805, 857760878, 2036477234, 1797285236}
153
154
155 procedure Chacha20_k16(int k0[4], int n[4], int out[16])
156   local int tau[4] = {1634760805, 824206446, 2036477238, 1797285236}
157   local int seq[16]
158   call Chacha20_setup_seq(k0, k0, n, tau, seq)
159   call Chacha20(seq, out)
160   uncall Chacha20_setup_seq(k0, k0, n, tau, seq)
```

```
161     delocal int seq[16]
162     delocal int tau[4] = {1634760805, 824206446, 2036477238, 1797285236}
163
164   // So, the two procedures above are the minimum for Salsa20/Chacha20
165   // encryption.
166   // The paper splits 'n' into a 64-bit nonce and a 64-bit byte sequence
167   // number (a byte ID from 0 to 2^64-1). Maximum message length is 2^70,
168   // possibly due to some crypto-related reason.
169
170   procedure main()
171     int seq[4] = {1, 0, 0, 0}
172     int k0[4] = {67305985, 134678021, 202050057, 269422093}
173     int k1[4] = {-859059511, -791687475, -724315439, -656943403}
174     int n[4] = {1751606885, 1818978921, 1886350957, 1953722993}
175     int out_s[16]
176     int out_d[16]
177
178     call Chacha20_k16(k0, n, out_s)
```