ESTUDO E IMPLEMENTAÇÃO EFICIENTE DE ALGORITMOS CRIPTOGRÁFICOS

Victor Cracel Messner

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Claudio Luis de Amorim

Rio de Janeiro
Março de 2018

ESTUDO E IMPLEMENTAÇÃO EFICIENTE DE ALGORITMOS CRIPTOGRÁFICOS

Victor Cracel Messner

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

Prof. Claudio Luis de Amorim, Ph.D.

Prof. Luis Felipe Magalhães de Moraes, Ph.D.

Prof. Alexandre Sztajnberg, D.Sc.

RIO DE JANEIRO, RJ – BRASIL
MARÇO DE 2018

Messner, Victor Cracel

Estudo e implementação eficiente de algoritmos criptográficos/Victor Cracel Messner. – Rio de Janeiro: UFRJ/COPPE, 2018.

XI, 70 p.: il.; 29, 7cm.

Orientador: Claudio Luis de Amorim

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 66 – 70.

1. ICN. 2. IOT. 3. Criptografia. I. Amorim, Claudio Luis de. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Aos meus pais

Agradecimentos

Agradeço primeiramente o apoio dado por toda minha família por nunca me deixarem desistir do meu sonho, mesmo nos tempos mais escuros.

Agradeço também aos meus professores, em especial ao meu orientador que sempre se mostrou paciente e me ajudou quando possível, que tornaram minha caminhada no mestrado mais interessante e por terem me proporcionado contato com conteúdos que não possuía sequer conhecimento durante a minha graduação, aos meus amigos Anderson Zudio, Ronaldo Ferreira, Hugo Neves, Thais Roberta e também aos meus colegas que passaram pelas mesmos desafios na universidade: Victor Cruz, Caio Bonfatti e Felipe Pollola.

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

ESTUDO E IMPLEMENTAÇÃO EFICIENTE DE ALGORITMOS CRIPTOGRÁFICOS

Victor Cracel Messner

Março/2018

Orientador: Claudio Luis de Amorim

Programa: Engenharia de Sistemas e Computação

Devido a grande quantidade de dispositivos sendo utilizados no dia a dia das pessoas formas de diminuir o consumo energético vêm sendo cada vez mais alvo de estudo e pesquisa.

Este trabalho visa comparar diferentes formas de criptografia a fim de tornar elegível àquela que possua um baixo consumo energético com uma segurança adequada. Será também alvo deste trabalho discorrer sobre os conceitos emergentes como as redes ICN e sobre a IoT.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

EFFICIENT STUDY AND IMPLEMENTATIONS OF CRIPTOGRAPHIC ALGORITHMS

Victor Cracel Messner

March/2018

Advisor: Claudio Luis de Amorim

Department: Systems Engineering and Computer Science

Due to the great amount of devices being used in the daily life of people there has been studied increasingly ways to reduce the energetic consumption.

This work aims to compare different forms of encryption in order to make it eligible to the one that has a low energy consumption with adequate security. It will also be the aim of this work to discuss emerging concepts such as the textit ICN networks and the textit IoT.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Tratar de criptografia não é algo novo, o desejo de transmitir informações entre indivíduos sem a compreensão de terceiros é uma prática antiga, sendo possível destacar existência de mensagens codificadas até mesmo no Egito antigo [1]. Na atualidade, o uso de e-mails, a realização de transações bancárias e a frequente utilização de aplicativos de redes sociais tornou essencial a codificação de mensagens e a garantia da segurança das informações codificadas.

Com o avanço das tecnologias móveis foi estimado que em 2020 o número de dispositivos conectados à rede sejam de 30 a 50 bilhões [2] [3], chegando ao valor de 75 bilhões em 2025 [2].

Com o aumento no número de dispositivos e a grande presença que os mesmos ocupam no dia a dia dos indivíduos, formou-se um cenário onde figuram sensores dos mais variados tipos, smartphones e até mesmo eletrodomésticos, todos capazes de serem reconhecidos e controlados pela internet.

Com base na definição proposta pelo conselho nacional de inteligência americano a Internet das coisas (IoT) é uma ideia geral de coisas, especialmente objetos do dia a dia, que são capazes de serem lidos, reconhecidos, encontrados, endereçados e controlados pela internet[3].

A partir dos parágrafos anteriores, podemos concluir que a internet das coisas vem cada vez mais tendo participação no dia a dia dos indivíduos.

Dentro do contexto da IoT podemos destacar como desafios as formas de interação entre dispositivos de grande heterogeneidade em sistemas altamente distribuídos. Tais desafios, podem ser solucionados através de um modelo de rede capaz de adaptar-se a muitos dispositivos com alta heterogeneidade, fornecendo rápido acesso e distribuição, segurança e escalabilidade. Essas características estão presentes em uma abordagem diferente que tem como proposta mudar a forma *host-centric* da internet como utilizamos hoje, a *Information-centric network(ICN).*

Como evidenciado em [4], a *ICN* é um paradigma emergente para a Internet do Futuro que inicialmente focava ganhos de largura de banda em conexões cabeadas,

mas revelou também um suporte significativo para conexões wireless, tornando-se de grande relevância em aplicações do dia a dia. Como ela *"ainda está em fase de debate"* [5], torna-se uma tarefa menos árdua o desenvolvimento de modelos que preservem as características básicas desse paradigma tais como: a qualidade do serviço, a escalabilidade, a segurança, a mobilidade e a heterogeneidade.

As *ICNs* possuem uma abordagem de obtenção de dados através de pedidos de objetos que possuem nomes únicos, chamados de *named objects*. Seus nomes devem ser capazes de identificar semanticamente um único objeto, isto é, em dado modelo um indivíduo é capaz de compreender e interpretar os nomes únicos atribuídos a esses objetos, nomes que atendam tal característica são ditos expressivos. A literatura considera principalmente a utilização de nomes hierárquicos a fim de identificar os componentes de uma aplicação IoT [5] bem como, documentos, páginas da internet, vídeos, músicas, entre outros, de maneira independente da localização de um servidor, aplicação ou canal de comunicação.

Em grande parte dos modelos de *ICN* utiliza-se a ideia *publish/subscribe* na qual se faz necessário um identificador único, com suporte para verificação de integridade, independente de seu dono *(owner)* e ainda um conjunto de metadados vinculados a esse objeto.

Devido a obtenção de conteúdo ser de forma independente da localização do servidor, foram desenvolvidos mecanismos de *caching* em nós operadores, em usuários e em terminais móveis. Foi também desenvolvido o conceito de segurança baseada no conteúdo de internet, e não no canal de comunicação como é feito no protocolo de comunicação TCP-IP.

A obtenção de informações se dá através de uma *API* capaz de realizar as operações de *request* e *delivery named data objects (NDOs)* e capazes de rotear e encaminhar esses objetos.

Diversas arquiteturas *ICN* foram propostas, com diversos designs de protocolos diferentes, conforme aquelas exibidas em [5], mas sempre de modo a possuírem um conjunto de características de *cashing*, objetos e *API* conforme o descrito para toda *ICN* [5] [6].

Conforme [5] a *ICN* possui uma série de desafios, contudo para o escopo deste trabalho serão apenas considerados como relevantes:

- Eficiência energética:Devido a sua grande quantidade de nós, sendo muitos deles móveis e com hardware limitado, a análise de custo energético torna-se um fator de grande relevância do ponto de vista financeiro, devido a economia de consumo de dispositivos, assim como para obtenção de dados de dispositivos que trafegam na rede, por exemplo, os logs de equipamentos. A eficiência energética também influencia diretamente no cotidiano do usuário, visto que, smartphones possuem uso de bateria limitado, dessa forma, uma

melhor eficiência energética garantiria também uma maior vida útil da bateria.

- Segurança do conteúdo:Como na *ICN* a segurança é baseada no conteúdo, faz-se necessária a utilização de mecanismos de criptografia a fim de garantir a segurança e a privacidade dos usuários que fazem parte da rede.

Devido à alta difusão do uso da internet, melhores soluções de segurança nas redes tornaram-se alvo frequente de pesquisa, nesse contexto, algoritmos de encriptação passaram a desempenhar um papel fundamental na proteção do conteúdo e na garantia da privacidade do usuário. Contudo, esses algoritmos consomem certa quantidade de recursos, tais como: tempo de processamento de dados, memória e bateria do dispositivo. Apesar do aumento visível nas frequências dos *clocks* é possível notar o *battery gap* criado, ou seja, as frequências dos *clocks* em dispositivos móveis estão crescendo de forma muito mais rápida do que a tecnologia de bateria utilizada. Por esse motivo a economia de consumo energético por meio de algoritmos tornou-se um fator importante para o desenvolvimento de aplicações eficientes e novas tecnologias.

É importante ressaltar que algoritmos utilizados para criptografia, que descreveremos em detalhes posteriormente, possuem instruções muito simples e são repetidos diversas vezes. Por exemplo, para criptografar um áudio *mp3* de 5 a 10 minutos, são necessárias cerca de 10^7 repetições da etapa de encriptação de um algoritmo. Devido a esse alto número de repetições, pequenas alterações no código podem gerar uma grande diminuição tanto no tempo de processamento, quanto no consumo energético do algoritmo.

Esta dissertação lida com o problema de criptografar e descriptografar dados de um usuário a fim de garantir a privacidade sobre o conteúdo; levando em consideração a baixa capacidade de processamento dos nós, o tempo de espera requerido para as operações de criptografar e descriptografar as mensagens e, por fim, o consumo energético requerido por essas operações.

É importante ressaltar que um sistema possui um nível adequado de segurança se atender pelo menos uma das duas condições [7]:

- O custo de quebrar a criptografia é maior que o valor da informação a ser obtida.

- O tempo de quebrar a cifra excede a vida útil da informação a ser obtida.

Dessa forma, cabe ao programador averiguar qual algoritmo possui o nível de segurança adequado e, se necessário, realizar alterações para que o mesmo se adeque. Todos os testes realizados neste trabalho fazem uso de algoritmos diferentes

com graus diferentes de segurança, mas considerados seguros, ou seja, não possuem criptoanálise capaz de obter a mensagem original sem ter em posse a chave privada. Tais conceitos serão mais bem explicados posteriormente neste trabalho.

1.1 Motivação

Existem muitas perspectivas quanto ao método de criptografia a ser empregado podendo variar entre modelos de criptografia simétrica e assimétrica [5], porém apesar da grande quantidade de algoritmos que poderiam ser empregados, os trabalhos, em geral, não mostram o impacto de sua implementação em dispositivos móveis, tampouco os códigos utilizados. Como dito anteriormente, pequenas modificações em um código podem gerar grandes diferenças de consumo energético em uma entrada grande o suficiente, e por esse motivo, faz-se necessária a explicação detalhada da implementação do algoritmo, e idealmente a disponibilização de seus códigos fonte, bem como a avaliação de seu desempenho em diversas arquiteturas de processadores a fim de evitar qualquer favoritismo e garantir resultados mais próximos da realidade.

Os algoritmos a serem apresentados deverão possuir as seguintes características:

- *Portabilidade:* A facilidade de reutilizar os códigos em diferentes sistemas operacionais e arquiteturas de processadores sempre foi um fator determinante para a escolha de código por um desenvolvedor. Um argumento que justifica de maneira clara essa tendência é o custo de desenvolvimento de uma implementação de um algoritmo para diferentes arquiteturas e sistemas operacionais.

  Essa dissertação lidará com códigos capazes de serem executados nas mais diversas arquiteturas sem alteração em qualquer linha, portanto se faz necessária a utilização de uma linguagem de programação que funcione em todas as arquiteturas e sistemas operacionais utilizados em larga escala, e que possua uma alta performance. Dadas as características supracitadas foi escolhida a linguagem de programação C++, sem a utilização de qualquer biblioteca externa para os códigos a serem propostos, garantindo dessa forma a portabilidade para sistemas móveis como, por exemplo, o Android, através da *native Lib*, como para sistemas Windows e Linux clássicos.

- *Eficiência energética:* Tanto usuários quanto desenvolvedores devem se preocupar com esse fator, uma vez que as baterias de dispositivos móveis possuem uma capacidade de utilização energética pequena. O aumento da utilização de algoritmos criptográficos em dispositivos móveis causa um impacto negativo no consumo energético, visto que esses algoritmos demandam tempo de

processamento e consequentemente consumo energético. É esperado que em um cenário de grande escala como em uma rede *ICN-IoT*, desenvolvedores aproveitem melhor as variações nos níveis de segurança de aplicações a fim de obter um menor consumo energético em suas aplicações.

É importante também ressaltar que: com o aumento dos números de dispositivos móveis, o consumo energético por parte de prestadores de serviços também vem aumentando e, portanto, faz-se necessário a utilização de algoritmos de baixo consumo energético a fim de diminuir custos de empresas.

Nesta dissertação serão apresentados exemplos de algoritmos, destacaremos sua segurança esperada e realizaremos comparações de eficiência energética dos mesmos. Através dos resultados obtidos por este trabalho, é esperado que desenvolvedores sejam capazes de avaliar qual algoritmo melhor atende suas expectativas.

- *Evolução* Com o passar do tempo pode haver a necessidade de modificações nos algoritmos de criptografia, devido a avanços em métodos de criptanálise, ou, de uma demanda por maior segurança em alguns tipos de aplicações, portanto, apresentaremos algoritmos que possuam uma grande perspectiva de utilização em função do tempo, longevidade e uma alta adaptabilidade. É importante ainda ressaltar que apesar desses dois casos serem os mais comuns, existem informações cujo valor deixam de ser relevantes muito rapidamente, e, portanto, não se faz necessária a utilização de um algoritmo de alto grau de segurança, mas sim com uma boa eficiência energética.

1.2 Objetivo

- Apresentar resultados sobre o comportamento dos algoritmos escolhidos para o escopo do trabalho com relação ao seu consumo energético, tempo de inicialização, tempo de encriptação, número de instruções executadas e o número de ciclos de clock.

- Propor mudanças em algoritmos apresentados pela biblioteca escolhida, gerando novos códigos fonte e avaliando o desempenho dos mesmos.

- Comparar o desempenho dos algoritmos propostos e os implementados com a biblioteca escolhida e avaliar possíveis ganhos de desempenho.

- Discutir a aplicabilidade dos algoritmos nas próximas tecnologias.

1.3 Contribuições

Esta dissertação contribui para os seguintes aspectos:

- *Segurança:* Apresentação de diversos algoritmos (ChaCha, RC4,RC5,RC6, Blowfish e Trivium) que devido ao quesito evolução, anteriormente apresentado, podem adequar-se aos mais diversos requisitos de segurança. Tal característica proporciona uma vasta gama de escolha por parte de desenvolvedores.

- *Portabilidade:* Todos os códigos de implementações propostos neste trabalho para processadores AMD e Intel encontram-se escritos na linguagem C++, fazendo uso somente de suas bibliotecas padrão. Os códigos para dispositivos móveis Android fazem uso da *NDK* a fim de manter a utilização de seus códigos C++. Todos os códigos podem ser recompilados em diversas plataformas desde que as bibliotecas padrões do Android e da linguagem C++ sejam aceitas.

- *Evolução* Todos os códigos desse trabalho serão disponibilizados para utilização e, portanto, caso exista necessidade de modificações para melhorias de desempenho ou testes em outras arquiteturas a utilização dos mesmos códigos torna-se possível.

- *Eficiência energética:* Cada um dos códigos será averiguado em consumo número de instruções utilizadas e número de ciclos necessários para sua execução. Serão também averiguados os tempos de execução em processos de inicialização e em processos de encriptação, a fim de obtermos resultados mais detalhados sobre o comportamento das implementações dos algoritmos, cujas alterações em comparação com a biblioteca serão descritas no próximo capítulo. Em dispositivos móveis, essas implementações irão medir o consumo em Joules da execução de cada um dos algoritmos.

  As implementações dos algoritmos propostas por esta dissertação possuem ganhos significativos de desempenho em 5 dos 6 algoritmos escolhidos em comparação com os resultados encontrados utilizando a biblioteca Cripto++. Foram comparados seus consumos de clock, seus números de instruções utilizadas e seus tempos de execução.

- *Relevância:* A fim de filtrar a quantidade de algoritmos escolhidos, essa dissertação escolherá algoritmos de acordo com testes de desempenho e pesquisas realizadas por outros pesquisadores. Para cada algoritmo considerado relevante, foi justificado o porquê se sua escolha ou não escolha, no escopo desta dissertação.

- *Aprendizado:* Cada um dos algoritmos escolhidos neste trabalho possui uma explicação detalhada de seu funcionamento e deixando clara a forma de como implementá-lo. Por este motivo um leitor pode aprender, não somente sobre os resultados da implementação do algoritmo, mas também sobre funcionamento de cada um deles caso deseje desenvolver uma versão alternativa à proposta dessa dissertação.

[7] e [8] são considerados como fundamentais para a compreensão dos fundamentos da criptografia, bem como da importância da mesma, contendo definições básicas sobre segurança e cifras, como cifras de bloco e cifras de *stream* e os conceitos matemáticos para a compreensão de diversos algoritmos.

Em [8] é possível obter muito mais detalhamento sendo discorrido com muito mais detalhes protocolos, sobre criptanálise, e avaliados características práticas como tempo necessário esperado para realizar a força bruta em um algoritmo para um certo tamanho de chave, entre outros. São também abordados muito mais algoritmos e técnicas para garantir a segurança envolvendo múltiplos algoritmos do que em [7].

Em [7] está presente o algoritmo *AES* que será utilizado para o este trabalho, tal algoritmo não estava presente em [8]. Também é explicitado que os problemas na segurança do *RC4* na verdade ocorreram por uma falha de segurança no protocolo e tal argumento, juntamente com os resultados obtidos por SINGHAL e J.P.S.RAINA [9] tornaram suficiente a inclusão desse algoritmo nesse trabalho.

[4], [5] e [6], discorrem sobre as características das arquiteturas ICN e apresentam os avanços recentes bem como seus problemas em aberto.

[10] discorre sobre métricas para força de algoritmos.

[11], [12], [13] e [14] apresentam análises sobre diversos algoritmos que possuem um desempenho pior que o padrão internacional (AES), já em [15], [16] e [17] são apresentadas vulnerabilidades e criptanálise de diversos outros. Os resultados obtidos por esses trabalhos serão utilizados para eliminar tais algoritmos e serão posteriormente.

[18] e [19] são importantes referências a fim de destacar que os resultados podem ser diferentes dependendo da integração com softwares de terceiros, como navegadores, e caso o processador possua facilitadores nativos para algum sistema de criptografia.

[20] e [21] são utilizados para justificar a escolha do algoritmo *ChaCha* e não da versão mais utilizada *Salsa* e é dito também que tal algoritmo possui melhor performance do que o padrão internacional com um alto grau de segurança.

[22] e [23] apresentam resultados a respeito dos algoritmos Mars e Serpent respectivamente. E no segundo o baixo desempenho do algoritmo Serpent em implementações de software serviu como argumento para sua não utilização no escopo deste trabalho.

[24] apresenta comparativos de consumo energéticos de diferentes protocolos, possibilitando obter um consumo estimado da combinação do consumo do algoritmo e do protocolo a ser utilizado.

[25] explica um algoritmo para a obtenção dígitos hexadecimais de π. Tais dígitos são necessários para a utilização do algoritmo blowfish.

[26] define conceitos como confusão e difusão utilizados na compreensão das melhorias entre o algoritmo *ChaCha* e *Salsa*. Esses conceitos também são fundamentais para um melhor entendimento do funcionamento dos algoritmos de criptografia.

[27] [28] demonstram a diferença entre a performance do RSA ao ser comparado com o AES, apresentando comparativos entre as seguranças, sendo necessário para o RSA um tamanho de chave cerca de vinte vezes maior para atingir o mesmo grau de segurança do AES. Tais resultados justificam a não inclusão desse algoritmo no escopo desse trabalho.

[29] explica o algoritmo MCbits, contudo em seu algoritmo é utilizada a criptografia através do algoritmo Salsa, que originou o ChaCha, e, portanto, o desempenho do algoritmo obtido é inferior ao do *ChaCha*. É importante também ressaltar que não é disponibilizado código oficial e não possuem fontes confiáveis para garantir a corretude da implementação, nem mesmo test vectors. Portanto, apesar de promissor, este algoritmo não foi incluído no escopo deste trabalho.

[30], [31], [32], [33], [34], [35], [36] e [37] discorrem sobre as complexidades de segurança dos algoritmos escolhidos.

[38], [39], [40] e [41] apresentam resultados de implementação em hardware, sobretudo em FPGAs, de alguns dos algoritmos escolhidos, possibilitando dessa forma observar claramente o *trade-off* das implementações hardware/software e estimar o ganho de desempenho das implementações propostas caso sejam implementadas em software.

[42], [43] e [35] discorrem sobre os algoritmos RC5, RC6 e Blowfish descrevendo-os e dando detalhes sobre como realizar suas implementações.

[44] compara os custos de energia do envio de pacotes de diferentes tamanhos utilizando o algoritmo RC4 e o algoritmo AES.

[45] compara os algoritmos RC6, AES, Serpent, Twofish e XTea.

[46] faz utilização de três algoritmos não utilizados nesta dissertação e a comparação de desempenho entre eles. A implementação destes três algoritmos e a comparação com os resultados obtidos neste trabalho tornam-se interessantes para um trabalho futuro.

1.4 Metodologia

Para exibição dos resultados obtidos em desktop, serão exibidas comparações dos tempos de inicialização e encriptação, dos consumos de ciclos e do número de instruções executadas entre a versão proposta, sem a utilização de bibliotecas externas à linguagem, e uma versão utilizando a biblioteca Cripto++. Essa biblioteca é amplamente utilizada na prática e conhecida na literatura, sendo utilizada como base para comparação e averiguação do desempenho obtido pelas implementações desenvolvidas para este trabalho. A biblioteca possui a versão mais recente de acordo com a data de início dessa dissertação.

Em desktops, com processador AMD (FX8300), ou Intel (Core I5 4200U), foram realizadas mil medições para cada algoritmo testado. Em ambas arquiteturas, foi utilizado o sistema operacional Ubuntu 16.04. Em um smartphone Sony XperiaJ com um processador (Qualcomm MSM7227A Snapdragon) e sistema operacional Android 4.1.2 foram realizadas 10 medições para cada algoritmo testado. Foram disponibilizadas as médias dos resultados das medições e seus desvios padrões quando maiores que 10^{-5}. Os dados de output das medições serão disponibilizados juntamente com o código.

Para a comparação entre os algoritmos, serão contabilizados a quantidade de ciclos de clock, medidos através da biblioteca PERF do Linux, bem como, o seu tempo, contabilizado através da biblioteca chrono disponível na biblioteca padrão do C++11. Em sistemas Android, será medido o consumo energético através da aplicação Power Tutor e o tempo de execução será utilizada a biblioteca padrão do Android.

Através dos dados obtidos será possível extrair resultados importantes, tais como, tempos de inicialização, os tempos de encriptação e paralelismo à nível de cpu de cada um dos códigos, e em seguida, destacar resultados promissores em cada uma das implementações dos algoritmos e suas equivalentes na biblioteca Cripto++. Seremos capazes de observar também o consumo energético de cada uma das implementações propostas em um dispositivo móvel real e obter assim dados relevantes para uma utilização em larga escala.

1.5 Estrutura

A dissertação consiste de 4 capítulos:

- Capítulo 2 - Algoritmos Simétricos: Este capítulo explica os conceitos básicos da criptografia e seus tipos. Discorre também sobre quais algoritmos serão escolhidos e porque outros serão omitidos. Este capítulo também descreve

algoritmos anteriores ao período do AES e posteriores ao mesmo que possuem um potencial de aplicação para criptografia. Para cada abordagem são apresentadas de forma sintetizada as ideias necessárias para a construção do algoritmo. São também apresentados os algoritmos em pseudocódigo e sua implementação em *C++*, relacionando o programa com o pseudocódigo apresentado e a ideia explicada no trabalho original.

- Capítulo 3 - Resultados: Apresenta as limitações de cada implementação e os resultados dos testes práticos de performance dos algoritmos apresentados.

- Capítulo 4 - Conclusão: Consiste na conclusão da dissertação, com as observações do autor e possíveis trabalhos futuros.

Capítulo 2

Algoritmos Simétricos

Neste capítulo explicaremos o funcionamento da criptografia simétrica e suas subdivisões. Exibiremos o funcionamento de diversos algoritmos desse tipo de criptografia e as razões pelas quais alguns algoritmos conhecidos não serão avaliados por este trabalho.

É importante destacar que os finalistas da competição AES, promovida pelo NIST em 1997 a fim de eleger qual algoritmo de chave simétrica iria ser utilizado para proteger as informações do governo federal americano, serão implementados fazendo uso da biblioteca *Cripto++*, e o funcionamento dos mesmos serão explicados passo a passo.

Como desejamos verificar os ganhos de desempenho dos demais algoritmos com relação ao padrão internacional AES foi utilizada uma implementação que faz uso da biblioteca *Cripto++*, que é amplamente utilizada na prática e na literatura, a fim de garantir que os dados comparativos serão os mais próximos daqueles utilizados na prática. A biblioteca faz também uso de diversas instruções Assembly e diversas melhorias em baixo nível usufruindo de características do processador em uso para obtenção de melhor desempenho.

Para os resultados presentes no próximo capítulo serão alterados os inputs e outputs no próprio código a fim de evitar tempos de espera desnecessários.

2.1 Criptografia Simétrica

A criptografia simétrica, também conhecida como criptografia convencional, ou ainda, criptografia de chave privada, é uma forma de criptografia onde é utilizada somente uma chave tanto para o processo de encriptação, quanto para o processo de desencriptação. Essa chave deverá ser distribuída para ambas as partes antes que ocorra a transmissão.

A segurança de um sistema de criptografia simétrica depende de duas coisas: A força do algoritmo aplicado e o tamanho da chave. Em [10] é melhor abordado o

conceito de força de um algoritmo e definidas métricas para este conceito. Conforme ilustrado em [8] se assumirmos que a força de um algoritmo é perfeita a melhor forma de quebrar um sistema de criptografia é através de força bruta, tentando cada possível chave.

2.1.1 Técnicas de criptografia

Dentre as mais variadas técnicas de criptografia algoritmos simétricos, de maneira geral, utilizam como base duas técnicas:

- *Substituição* Modelo que consiste em um mapeamento do texto original e a troca de caracteres por outros. Para uma melhor compreensão do funcionamento desse modelo podemos apresentar a cifra de César. Considere o mecanismo de criptografia a seguir: Tome uma mensagem composta de letras do nosso alfabeto, o texto que desejamos criptografar e um número qualquer, 9 por exemplo. Se uma letra ocupa a posição x em nosso dicionário, ela ocupará a posição $(x + 9) \mod 26$, pois possuímos 26 letras no nosso alfabeto. Facilmente percebemos que essa cifra é vulnerável, ou seja, dado o texto criptografado existem formas de obtenção da informação original. Tal vulnerabilidade se dá do fato em que que mesmo que a pessoa não perceba o deslocamento das letras no alfabeto, essa cifra não muda a frequência que letras e palavras ocorrem no texto.

- *Transposição* Modelo que consiste em mudar letras do texto original de lugar seguindo determinado padrão para que assim seja feito um texto criptografado. Suponha agora o seguinte cenário: Tome uma mensagem composta pelos caracteres a1...a28, e um número que divida 28, digamos 7. Elaboraremos então uma matriz com 4X7 onde cada linha da matriz contém os caracteres da mensagem sete a sete. Então definiremos uma chave, que será uma permutação dos algarismos de 1 a 7. Agora o texto criptografado será a nova matriz lida coluna a coluna, na ordem da permutação, isto é, se a permutação for *3214567*, a terceira coluna seria o início da nova mensagem.

2.1.2 Tipos de criptografia simétrica

Os algoritmos de chave simétrica podem ainda ser classificados em dois grupos:

- *Cifras de Fluxo (Stream cipher)* Consistem de utilizar a chave bit a bit, em geral, de modo a gerar um texto criptografado de forma pseudorandômica. É importante que a sequência de encriptação tenha um longo período a fim de evitar repetições. Deve-se possuir aproximadamente o mesmo número de zeros e uns.

- *Cifras de bloco (Block cipher)* Consistem na divisão do texto original em blocos, usualmente maiores ou iguais a 64 bits, onde cada bloco é tratado como um todo e gera um texto criptografado de mesmo tamanho. Existem formas de transformar esses tipos de cifras em cifras de fluxo, como é mostrado em [7].

2.2 Algoritmos não avaliados

O *IDEA* e o *CAST* possuem um desempenho pior do que o padrão internacional *AES* [11], assim como o *DES*, *3DES* e o *RC2* [12] e o *XOR* [13].

É importante ressaltar que os testes aplicados indicam os possíveis candidatos, mas isso não significa que o algoritmo seja o melhor e deva ser utilizado incondicionalmente. Em alguns casos resultados das aplicações de criptografia em navegadores podem produzir mais tempo ocioso em software devido a forma como o sistema e as suas demais aplicações se integram [18]. Deve-se também levar em consideração que o *AES* possui facilitadores a nível de hardware em algumas ARM-CPU, em alguns hardwares de aparelhos médicos existem facilitadores para o *ECC* e para o *RSA*, podendo haver variações com relação aos resultados obtidos, uma vez que os ganhos da implementação em hardware de diversos algoritmos apresentam *speedup* [19].

As criptografias *Cast256*, *Deal* e *Safer+* possuem vulnerabilidades claras, já em *CRYPTON* torna-se possível a descoberta de 92 bits da chave e em *DFC* e *E2* é possível descobrir em poucos passos cerca da metade dos bits da chave [15].

O modelo *Loki97* é fraco a ataques lineares [16], e a criptoanalise do *MAGENTA* é explicada [17], sendo esses um dos motivos pelos quais foram eliminados na primeira fase da competição.

O *Mars* foi uma das cifras submetidas pela *IBM* para a competição *AES*. Essa cifra faz uso de operações de computadores modernos e faz um *tradeoff* de performance para um maior ganho de segurança [22]. Constatou-se que esse cifra é inferior em desempenho quando comparado ao AES [14].

Serpent foi mais um candidato ao *AES* sendo inclusive um dos finalistas competindo com o *Rijindel* e o *Twofish*, contudo exibido nos relatórios gerais o algoritmo era cerca de três vezes mais lento do que os outros dois em software [23] e portanto não será abordado neste trabalho, apesar de sua grande relevância e segurança.

Os algoritmos CIPHERUNICORN-A, Bear and lion, Akelarre, CS Ciphe são também relevantes, mas não propostos neste trabalho.

2.3 Algoritmos Avaliados

Segue uma tabela definindo os algoritmos apresentados destacando se são algoritmos de cifra de bloco ou de fluxo.

Tabela 2.1: Algoritmos Avaliados

Algoritmo	Tipo de Cifra
AES	Bloco
RC5	Bloco
RC6	Bloco
Blowfish	Bloco
RC4	Fluxo
ChaCha	Fluxo
Trivium	Fluxo

2.3.1 AES

Também conhecido como Rijindel antes de vencer a competição de cifras AES. Esse algoritmo possui uma bom desempenho tanto em software quanto em hardware, inclusive em *CPU*s de 8 *bits*, sendo o modelo de criptografia eletrônica estabelecido pelo *National Institute of Standards and Technology (NIST)*. Essa também é a primeira cifra aprovada para arquivos considerados como *top secret* pelo *NSA* que é disponibilizada publicamente.

De acordo com [30] a complexidade do *AES128* quando foi apresentada era $2^{126.1}$ e graças aos *biblique attacks* sua complexidade foi diminuída para $2^{124.97}$.

Algoritmo

Todas as explicações feitas realizadas nesta sessão nessa área baseiam-se em [7]. O algoritmo é constituído de 6 etapas, sendo uma delas a expansão da chave. A seguir explicaremos o funcionamento deste algoritmo.

Primeiramente a chave dada ao algoritmo será transformada em quarenta e quatro novas palavras de 32 *bits* cada, cada quatro palavras geram uma nova chave de 128 *bits* que será uma chave de round (*round-key*). Para isso será utilizada uma matriz especial chamada *State-box*, ou *S-box*.

Nesse algoritmo, uma *S-box* pode ser criada através de uma multiplicação especial de matrizes onde não são somados os valores como na multiplicação de matrizes normais, mas feito o *XOR* de seus valores. Após a multiplicação faz-se um último *XOR* com o valor 63 para que finalmente seja encontrado seu correspondente.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Tomemos o número 95 como exemplo, sabemos que seu binário é 10010101. Devemos descobrir seu inverso multiplicativo em GF(2^8), ou seja, um número cujo produto com 10010101 tenha resto 1 na divisão por 000100011011. O resultado encontrado é 10001010.

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \oplus \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Uma vez que já sabemos seu inverso multiplicativo em GF(2^8) e dada a matriz base, basta operarmos com a multiplicação de matrizes especial anteriormente descrita e operarmos o resultado com *xor bit* a *bit* com o binário dado. Encontraremos assim o valor 2A que deverá ser inserido na linha 9 e coluna 5 de nossa *S-box*.

Para a transformação inversa é utilizada uma *S-Box* inversa que é construída aplicando a transformação inversa da descrita na *S-Box* original.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

A expansão é feita através de um *xor* com o valor de quatro posições atrás caso o índice não seja múltiplo de 4. Caso ele seja múltiplo de 4, deve-se rotacionar seus

bits de forma circular e realizar uma substituição de cada byte usando a*S-Box* para que então seja feito o *xor* com o valor de 4 posições atrás. Cada chave de round é utilizada pela etapa de adição de chave do round, *add round key*.

A etapa de substituição de bytes, *substitute bytes*, faz uso da *S-box* para a realização de um simples *look-up*, que contém permutações de todos os possíveis 256 valores de 8 *bits*. Cada *byte* do estado é mapeado fazendo uso dos 4 *bits* mais a esquerda para o índice da linha e os 4 *bits* mais a direita para o índice da coluna, conforme foi visto durante a explicação das *S-Box*.

A etapa de troca de linhas, *Shift Rows*, é uma simples permutação circular da linha. Cada linha é deslocada n-1 vezes para a esquerda, com a contagem comece em 1.

A etapa de mixagem de colunas,*Mix columns*, faz uso da aritmética no GF(2^8) que também é utilizada nas *S-box* para realizar substituições utilizando as regras dispostas:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{0,1} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{0,2} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{0,3} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{0,1} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{0,2} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{0,3} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

É fácil percebermos que podemos generalizar para cada coluna, para possuirmos fórmulas mais simpáticas de serem implementadas, apesar de logicamente não fazer nenhuma diferença a mudança de representação. Observe a generalização:

$s'_{0,j} = (2 \bullet s_{0,j}) \oplus (3 \bullet s_{1,j}) \oplus (s_{2,j}) \oplus (s_{3,j})$
$s'_{1,j} = (s_{0,j}) \oplus (2 \bullet s_{1,j}) \oplus (3 \bullet s_{2,j}) \oplus s_{3,j}$
$s'_{2,j} = (s_{0,j}) \oplus (s_{1,j}) \oplus (2 \bullet s_{2,j}) \oplus (3 \bullet s_{3,j})$
$s'_{3,j} = (3 \bullet s_{0,j}) \oplus (s_{1,j}) \oplus (s_{2,j}) \oplus (2 \bullet s_{3,j})$

A transformação inversa:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{0,1} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{0,2} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{0,3} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{0,1} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{0,2} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{0,3} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

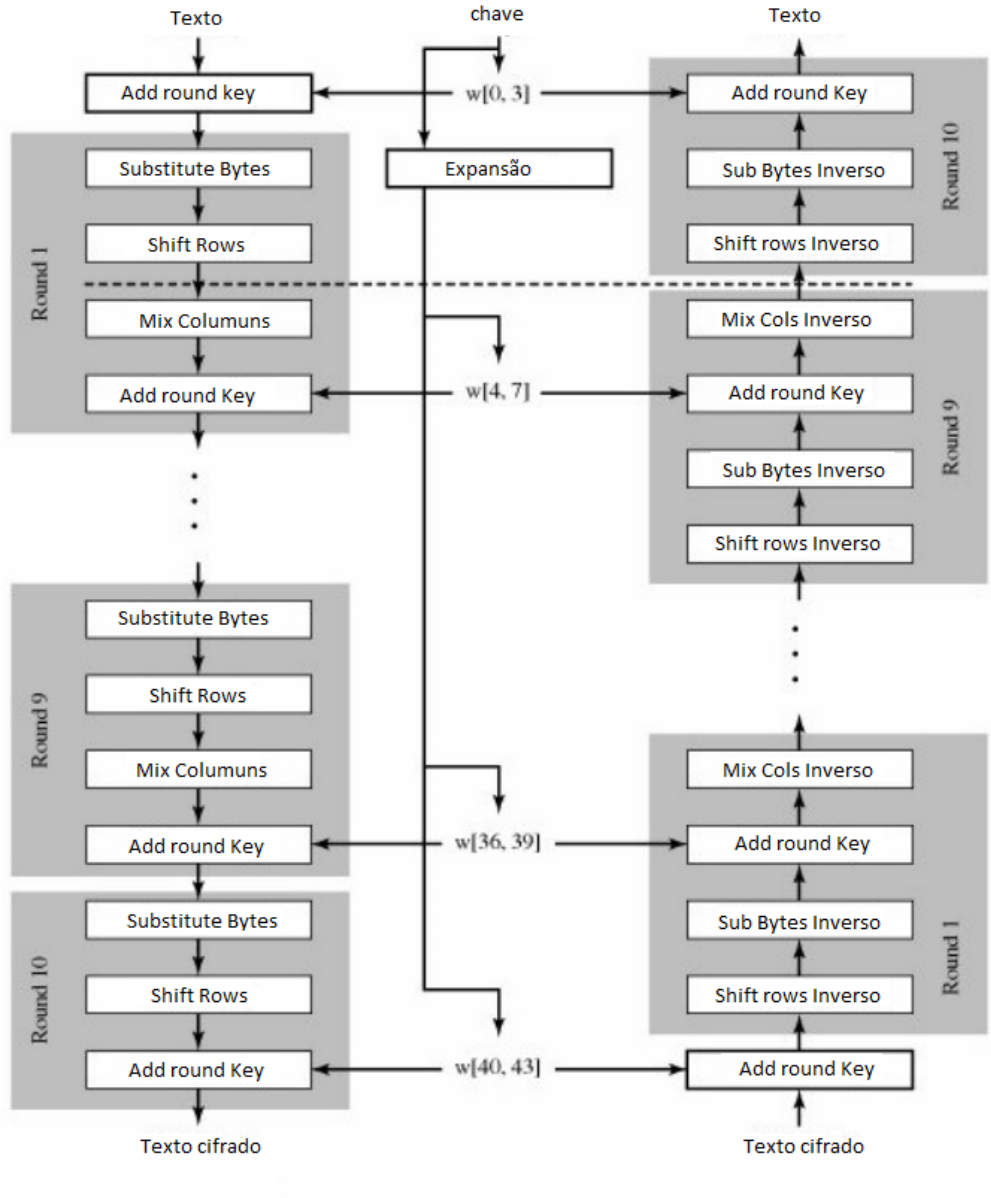
A etapa de adição de chave do round,*Add round key*, é um simples *XOR* do bloco atual com a chave do round.

Como sabemos, todos os estágios são reversíveis, e portanto, podemos obter novamente o texto original, assim sendo eles não acrescentam nenhuma segurança a não ser que façam uso da chave, sendo por isso que a primeira etapa do algoritmo é

um *add round key*.

Agora que temos todas as ideias do algoritmo em mente podemos pensar nele de uma forma mais sintetizada como na figura 3.1:

Figura 2.1: Algoritmo AES



Implementação

Para a implementação desse código será utilizada a biblioteca cripto++.

2.3.2 RC5

O *RC5* foi desenvolvido com o objetivo de ser uma cifra de bloco de implementação simples, com baixos requisitos de memória e de alto desempenho em hardware e

software [42]. A proposta dos criadores era a formulação de um algoritmo que fosse adaptável a *cpus* com diferentes números de palavras e a diferentes graus de segurança, incluindo assim tamanhos de chaves diferentes e números de rounds diferentes. Devido a sua simplicidade a implementação em *FPGA* do algoritmo *RC5* mostrou um aumento de 80% no *throughput*, ou seja um aumento de 80

São necessários 2^{44} textos conhecidos para violar a segurança desse algoritmo utilizando blocos de 64 bits, ou seja fazendo uso de seus dois blocos de 32 bits [32], e devido a sua alta velocidade mais rodadas podem ser utilizadas para aumentar dessa forma a segurança do sistema. Mas é concluído que com 16 rounds e um bloco de 64 *bits* o algoritmo apresenta uma segurança razoável contra os ataques, para maiores informações sobre o *trade-off* segurança/desempenho consultar o artigo supracitado.

Algoritmo

Começaremos definindo as constantes mágicas do *RC5* que são requisitos para a implementação do algoritmo. São elas:

- $P_w = Odd((e-2) * 2^w)$
- $Q_w = Odd((\phi - 1) * 2^w)$

Onde:

- Odd(x) Representa o número impar mais próximo do resultado
- e É o logaritmo natural (2.718281828459...)
- ϕ É a razão áurea (1.628033988749...)

Substituindo para 32, temos: P_w = odd(3084996962,542487076864) = 3084996963. Em hexadecimal teremos B7E15163. É fácil obtermos constantes para qualquer valor de w desejado.

Primeiramente devemos expandir nossa chave através de uma conversão de *bytes* para palavras, para tanto seja K nossa chave composta de b *bytes*, e seja L um *array* com $\lceil 8b/w \rceil$ *bytes*. É fácil vermos que quanto maior nosso w, consequentemente maior nossas palavras mágicas, menor será o *array* L.

Nos casos em que b=c=0, c é iniciado como 1 e L[0]=0. Para os demais casos usa-se u chaves consecutivas de k para preencher cada palavra em L. Qualquer posição não preenchida é deixada como 0.

Algorithm 1 Primeiro Passo da Expansão de Chave RC5

```
c=⌈max(b, 1)/u⌉
for i=(b-1) até 0 do
    L[i/u] = (L[i/u] <<<8) + K[i]
end for
```

O simbolo (<<<) é utilizado para representar a rotação no sentido para esquerda.

O segundo passo do algoritmo de expansão de chave é inicializar um vetor S de tamanho igual a $(2*r+1)$ onde r é o número de rounds seguindo a seguinte recursão:

- S[0] = P_w)
- S[i] S[i-1] + Q_w

Algorithm 2 Segundo Passo da Expansão de Chave RC5

```
S[0]=P_w
for i=0 até (t-1) do
    S[i] = S[i-1]+ Q_w
end for
```

O ultimo passo diz respeito a mixar nossa chave secreta fazendo uso dos *arrays* S e L. A mixagem é um algoritmo bem direto.

Algorithm 3 Segundo Passo da Expansão de Chave RC5

```
i=0,j=0,A=0,B=0
for i=0 até 3*max(t,c) do
    A=S[i] = (S[i]+ Q_wA + B) <<<3
    B=L[j] = (L[j]+ Q_wA + B) <<<(A+B)
    i = i + 1 mod t
    j = j + 1 mod c
end for
```

Feito isso, já temos a nossa fase de expansão da chave completa, preenchendo assim o Vetor S e abrindo caminho para que sejam explicadas as ideias referentes à encriptação e desencriptação usando esse algoritmo.

Suponha que tenhamos dois registradores de w *bits* A e B iniciados com os valores que desejamos criptografar respectivamente, antes de tudo, devemos somar a esses dois registradores com valores iguais a S[0] e S[1] respectivamente, então o processo de encriptação será fazer A se tornar o valor contido na posição do vetor S de índice igual a duas vezes o número do round somado com o valor deslocado b vezes para a esquerda de A *xor* B. B deverá se tornar o valor o valor contido em S[2*i+1] somado com o valor deslocado A vezes para a esquerda de B *xor* A.

Esse processo deverá ser feito por todos os r rounds escolhidos. Ao final dos r rounds obteremos os *bits* criptografados como desejamos.

Algorithm 4 Encriptação RC5

```
B = B + S[1]
A= A + S[0]
for i=1 até r do
    A = ((A ⊕B) <<<B) +(S[2*i])
    B = ((B ⊕A) <<<A) +(S[2*i+1])
end for
```

O processo de desencriptação pode ser escrito como a aplicação das operações em ordem inversa.

Algorithm 5 Decriptação RC5

```
for i=r até 1 do
    B = ((B - S[2*i+1]) >>>A) ⊕A
    A = ((A - S[2*i]) >>>B) ⊕B
end for
B = B - S[1]
A= A - S[0]
```

Implementação

A implementação do RC5 se dá seguindo os passos conforme descrito pelo algoritmo, contudo sua haverá uma modificação em sua função de rotação quando comparada com a versão da biblioteca. Para a esquerda será realizada através da operação ROTL(v,n) (((v)<<(n))|((v)>>(32-(n)))). Analogamente a rotação para a direita é feita através da operação ROTR(v,n) (((v)>>(n))|((v)<<(32-(n)))).

Para melhor entendermos devemos pensar no binário de um número. Observemos que (v) <<(n)) descartará os n primeiros números do binário de v e acrescentará n zeros ao seu final. Já (v) >>(32-(n)) irá gerar a diferença entre o número máximo de bits, 32 e o número de bits que desejamos rotacionar, obtendo assim todos os demais bits do número e removendo-os acrescentando então os zeros a sua esquerda. Dessa forma o ou (OR) fará a adição das duas partes obtendo o número rotacionado de n bits. A interpretação da outra rotação se dá de forma análoga.

2.3.3 RC6

O *RC6* é uma cifra apresentada durante a competição *AES* como a sucessora do *RC5*. Desenvolvido para atender os requerimentos da competição, o algoritmo *RC6* possui uma melhoria significativa na difusão por round, melhorando assim a segurança, devido ao uso da multiplicação. Assim como o *RC5* o *RC6* faz uso de rotações de dados.

No *RC6* são utilizados quatro registradores ao invés de dois, essa medida foi tomada pois a arquitetura e as linguagens apresentadas pela competição ainda não suportavam operações de 64 *bits* de maneira clara e eficiente, contudo existem formas de adaptar o algoritmo para a utilização de dois registradores, como abordado em [43]. A utilização de quatro registradores permite que sejam feitas duas rotações por round ao invés de uma rotação por meio round como é feito no *RC5*.

Assim como o *RC5* o *RC6* continua a utilizar técnicas que apresentem uma alta segurança e difusão e que sejam implementadas eficientemente em processadores modernos, como a rotação e a multiplicação de inteiros.

É estimado que com a execução de 12 rounds o *RC6* precise de 2^{83} textos conhecidos, e com 16 rounds esse número suba para 2^{119} [33]. Atualmente o valor mais utilizado é 20 rounds com segurança estimada de 2^{155}, porém essa complexidade foi diminuída através da criptanalise X^2 sendo averiguado que com 16 rounds ainda são necessários $2^{127.20}$ textos na teoria [34]. É ainda citada uma forma de proteger o algoritmo contra esse tipo de ataque, a ideia proposta será apresentada nos algoritmos, mas não será implementada.

Algoritmo

Assim como no *RC5*, $>>>$ representa uma rotação de bits para a direita, e $<<<$ representa uma rotação de bits para esquerda. O simbolo $\oplus$ significa o *xor* bit a bit. Assim como no *RC5* as constantes mágicas estão presentes e são calculadas da mesma forma.

A expansão da chave ocorre de forma muito parecida no *RC5* e no *RC6*, tendo diferença somente na constante t, que é o tamanho do vetor S, que deixa de ser $2 * (r + 1)$ e passa a ser $2 * (r + 2)$ e no preenchimento do vetor L.

O processo de encriptação e desencriptação do texto sofre mudanças por se tratar e 4 registradores de 32 *bits* conforme mostram os algoritmos a seguir:

Algorithm 6 Encriptação RC6

B = B + S[0]
D = D + S[1]
for i=1 até r **do**
    t = (B X (2B+1)<<<lg w
    u = (D X (2D+1)<<<lg w
    A = $A\oplus$t)<<<u) + S[2*i]
    C = $C\oplus$u)<<<t) + S[2*i+1]
    (A,B,C,D) = (B,C,D,A)
end for
A= A + S[2*r+2]
C= C + S[2*r+3]

Algorithm 7 Desencriptação RC6

A= A - S[2*r+2]
C= C - S[2*r+3]
for i=r até 1 **do**
    (A,B,C,D) = (D,A,B,C)
    u = (D X (2D+1))<<<lg w
    t = (B X (2B+1))<<<lg w
    C = ((C - S[2*i+1])>>>t) $\oplus$ u
    A = ((A - S[2*i])>>>u) $\oplus$ t
end for
D= D - S[1]
B= B - S[0]

O algoritmo para melhoria da segurança [34] consiste em avaliar o números de "1"s no binário do número. Caso esse valor seja par deve-se manter o binário como está.Caso ele seja ímpar deve-se trocar as duas metades do binário. É fácil percebermos que a implementação de troca pode ser feita com a mesma função de rotação. A descoberta do *hamming weight* pode ser feita com o uso de um *bitset* em C++, ou ainda checando cada um dos *bits*. Entretanto muitos processadores dão suporte a esse tipo de chamada. Suponha que T seja minha função que verifica e faz a troca dos *bits* se necessário. A encriptação ficaria:

Algorithm 8 Hamming RC6

```
B = B + S[0]
D = D + S[1]
for i=1 até r do
    B=T(B)
    D=T(D)
    t = (B X (2B+1)<<<lg w
    u = (D X (2D+1)<<<lg w
    A = A⊕t)<<<u) + S[2*i]
    C = C⊕u)<<<t) + S[2*i+1]
    (A,B,C,D) = (B,C,D,A)
end for
A= A + S[2*r+2]
C= C + S[2*r+3]
```

Implementação

Por se tratar de uma derivação direta do algoritmo RC5, as melhorias na implementação seguem as derivações diretamente.

2.3.4 Blowfish

O *blowfish* é um algoritmo simétrico desenvolvido por Schneier que faz uso de blocos de 64 *bits* e chaves que podem variar até 448 *bits*. Sua fase de inicialização é bastante complicada, porém a fase de encriptação de dados e decriptação é bastante eficiente [35]. Esse algoritmo também foi desenvolvido para possuir uma fácil implementação em hardware.

Algoritmo

Antes de começarmos a falar propriamente do algoritmo vamos falar sobre seus requisitos de funcionamento. O *blowfish* precisa de um *array* P contendo 18 subchaves e 4 S-boxes de 32 bits, com 256 entradas em cada.

O *array* P é composto de dígitos hexadecimais de Pi (com exceção do inicial), eles podem ser aproximados pela formula BBP [25], contudo não é necessário o cálculo uma vez que esses valores já são fáceis de serem encontrados. Usaremos a *string*: 243F6A8885A308D313198A2E03707344A4093822299F31D0082EFA98EC4E6C8945 2821E638D01377BE5466CF34E90C6CC0AC29B7C97C50DD3F84D5B5B547091792 16D5D98979FB1B.

Dessa forma P1 = 0x243f6a88 P2 = 0x85a308d3 P3 = 0x13198a2e P4 = 0x03707344 ... P18 = 0x8979FB1B.

As quatro *S-boxes* deverão ser preenchidas da mesma forma, utilizando os dígitos hexadecimais de PI. Continuando a sequencia de dígitos até que todas estejam preenchidas.

Com todas as dependências cumpridas podemos analisar o algoritmo que pode ser dividido em duas partes: a expansão da chave e a encriptação dos dados.

A expansão transforma a chave em palavras de 32 *bits*, usando 4 caracteres, e guarda em P[i] o valor de P[i] *xor* o valor da chave de 32 *bits*. Isso é feito até completar as 18 posições de P[i]. A chave original deverá ser percorrida circularmente.

Agora devemos encriptar os registradores que começam todos como zero durante 9 passos e salvá-los 2 a 2 no vetor P[i]. Por fim devemos encriptar cada uma das posições das 4 *S-boxes* usando os mesmos registradores obtidos da encriptação do vetor P.

Eis o algoritmo de inicialização completo:

Algorithm 9 Inicialização Blowfish

```
for i=0 até 18 do
    palavra = 0
    for k=0 até 4 do
        palavra = (palavra¡¡8) + chave[j]
        j = (j+1) mod (tamanho-da-chave)
    end for
    P[i] = P[i] ⊕ palavra
end for
A = 0
B = 0
for i=0 até r/2; do
    Encripta(A,B)
    P[2*i]=A
    P[2*i+1]=B
end for
for i=0 até 4; do
    for j=0 até 127; do
        Encripta(A,B)
        S[i*2][j]=A
        S[i*2+1][j]]=B
    end for
end for
```

Antes de entrarmos no processo de encriptação e desencriptação devemos falar sobre a função não reversível F que garante o melhor efeito avalanche possível para uma rede de feistel [35]. Essa função recebe uma palavra de 32 bits, XL e a divide em 4 palavras de 8 bits, a,b,c,d, e aplica a seguinte equação: F(XL) = ((S[1][a] + S[2][b] $\mod 2^{32}$) $\oplus$ $S[3][c]$)+S[4][d] $\mod 2^{32}$

Algorithm 10 F Blowfish

```
d = XL & 31
XL=XL>>8;
c = XL & 31
XL=XL>>8;
b = XL & 31
XL=XL>>8;
a = XL & 31
XL=XL>8;
retornar (((((S[0][a] + S[1][b]) ⊕ S[2][c]) + S[3[d]]))
```

Devemos analisar agora os processos de encriptação e desencriptação. Nessas fases todas as operações podem ser resumidas a *xor* e adições.

Suponha que nosso *input* sejam duas palavras de 32 *bits*, A e B. Faremos durante 16 *rounds* o valor de A se tornar *xor* de A com p[i], com i indicando o índice do *round*, para depois fazermos b se tornar o *xor* de F(A) com B. e por fim um *swap* de A com B. Após os 16 *rounds* faça mais um *swap* de A com B. e faça o *xor* com P[17] e P[18] e você obterá os textos criptografados.

Algorithm 11 Encripta Blowfish

```
for i=0 até 15 do
    A=A⊕P[i]
    B=F(A)⊕B
    troca(A,B)
end for
troca(A,B)
B = B⊕P[16]
A = A ⊕ P[17]
```

A decriptação ocorre na ordem revesa porém usando todo o mesmo conteúdo.

Algorithm 12 Decripta Blowfish

```
for i=17 até 2 do
    A=A⊕P[i]
    B=F(A)⊕B
    troca(A,B)
end for
troca(A,B)
B = B⊕P[1]
A = A ⊕ P[0]
```

Implementação

A implementação segue o algoritmo e sua compreensão é bem simples, contudo foram feitas alterações para a obtenção de um aumento de performance.

Primeiramente durante a encriptação e desencriptação nos rounds as palavras operam entre si com um simples swap. Percebemos então que F(A) será aplicada aos termos ímpares enquanto F(B) será aplicada aos termos pares. Dessa forma podemos simplificar os rounds caso façamos o *unroll* dos mesmos.

A chave utiliza permutação circular, a ideia é evitar o uso da divisão. Para tanto sabemos que a chave será iterada por 72 índices e podemos executar sua expansão até 72 com antecedência poupando assim tempo de processamento.

A última característica importante a ser citada é que o round de expansão apesar de também possuir o *merge* de 4 chaves para uma palavra, não será de uma forma ingênua. Isso se deve ao fato de que a essa fosse a escolha tomada ao calcularmos F deveríamos reparticionar toda a palavra em 4 partes novamente. A ideia para evitar esse custo extra de computação é usar a estrutura *Union*.

Para fins de legibilidade as 4 *S-boxs* serão adicionadas ao apêndice.

2.3.5 Twofish

O *Twofish* foi um dos finalistas da competição *AES* juntamente com o *Rijindel* e com o *Serpent*. Diferentemente do *Rijindel* o *Twofish* não possuía uma boa implementação em hardware possuindo um baixo *speedup* quando comparado aos demais competidores [40] e apresenta um alto consumo de área em *FPGAs*. Maiores detalhes de como implementar o *twofish* em *FPGAs* podem ser encontrados em [47].

Esse algoritmo foi desenvolvido com a finalidade de ser um algoritmo de chave simétrica que utiliza chaves de 128,192 e 256 *bits*, sem a existência de chaves fracas e que seja eficiente, flexível e simples. Para chaves que não estejam no tamanho especificado coloca-se zeros a direita até que esteja em um dos tamanhos definidos.

Assim como o *blowfish*, o *twofish* utiliza o conceito de quatro S-Boxes, contudo nesse algoritmo as S-boxes possuem as características de serem bijetivas. São ainda necessárias operações em GF(2^8) assim como no *Rijindel* porém sua aplicação é utilizada para as matrizes MDS.

Sem a técnica de *whitening*, que nada mais é do que um *xor* na chave antes do primeiro round e após o ultimo, não existem ataques que requeiram menos de 2^{80} textos [48].

Algoritmo

Antes de falarmos sobre os métodos de encriptação e descriptação devemos falar sobre as dependências desses algoritmos. Para tanto começaremos exibindo a matriz constante MDS, a Matriz constante RS.

Neste algoritmo será necessária a apresentação de requisitos básicos para o funcionamento do algoritmo antes de apresentarmos de fato o processo de expansão de chaves e de encriptação.

Começaremos apresentadas as matrizes MDS e RS.

$$MDS = \begin{bmatrix} 01 & EF & 5B & 5B \\ 5B & EF & EF & 01 \\ EF & 5B & 01 & EF \\ EF & 01 & EF & 5B \end{bmatrix}$$

$$RS = \begin{bmatrix} 01 & A4 & 55 & 87 & 5A & 58 & DB & 9E \\ A4 & 56 & 82 & F3 & 1E & C6 & 68 & E5 \\ 02 & A1 & FC & C1 & 47 & AE & 3D & 19 \\ A4 & 55 & 87 & 5A & 58 & DB & 9E & 03 \end{bmatrix}$$

Agora devemos exibir como realizar as permutações q0 e q1 necessárias para o processo de expansão de chave. Essas permutações podem ser geradas pelo algoritmo a seguir

Algorithm 13 Twofish Permutação q0

t0 = [8 1 7 D 6 F 3 2 0 B 5 9 E C A 4]
t1 = [E C B 8 1 2 3 5 F 4 A 6 7 0 9 D]
t2 = [B A 5 E 6 D 9 0 C 8 F 3 2 4 7 1]
t3 = [D 7 F 4 1 2 6 E 9 B 3 0 8 5 C A]
$a_0, b_0 = x/16, x \bmod 16$
$a_1 = a_0 \oplus b_0$
$b_1 = a_0 ROR_4(b_0, 1) \oplus 8a_0 \bmod 16$
$a_2, b_2 = t_0[a_1]t_1[b_1]$
$a_3 = a_2 \oplus b_2$
$b_3 = a_2 ROR_4(b_2, 1) \oplus 8a_2 \bmod 16$
$a_4, b_4 = t_2[a_2]t_3[b_3]$
$y = 16b_4 + a_4$

Para a permutação q1 o mesmo processo é feito com exceção dos vetores utilizados. Em q1:

Algorithm 14 Twofish Permutação q1

t0 = [2 8 B D F 7 6 E 3 1 9 4 0 A C 5]
t1 = [1 E 2 B 4 C 3 7 6 D A 5 F 9 0 8]
t2 = [4 C 7 5 1 6 9 A 0 E D 8 2 B 3 F]
t3 = [B 9 5 1 C 3 D E 6 4 7 F 2 0 8 A]

Agora devemos descrever o processo de expansão de chave. Esse método deve produzir 40 chaves e 4 *S-boxes* que são dependentes da chave, que serão utilizadas na função g explicada posteriormente.

O primeiro passo da expansão de chaves consiste em dividir a chave de 8k *bytes* em 2k palavras de 32 bits cada e separá-las em dois *arrays* $M_e = (M_0, M2, ..., M_{2k-2})$ e $M_o = (M_1, M_3, ..., M_{2k-1})$ cada um de tamanho k. Sendo M_i o i-ésimo *byte* da chave. Um terceiro vetor, S, de tamanho k é derivado da chave através da separação dos *bytes* da chave em grupos de 8 e interpretando-os como um vetor sobre GF(2^8) e multiplicando-o pela matriz RS. Cada resultado de 4 *bytes* é interpretado como uma palavra de 32 *bits*.

A função h consiste em dois *inputs* X, uma palavra de 32 *bits*, e L, uma lista de palavras de 32 *bits* de tamanho k. Essa função retorna uma palavra. Durante k estágios 4 *bytes* passam por uma *S-box* fixa e é feito um *xor* com um *byte* da lista. Por fim os *bytes* passam mais uma vez por uma *S-box* fixa e os 4 *bytes* são multiplicados pela matriz MDS.

Podemos descrever o algoritmo dado por h da seguinte forma:

Algorithm 15 Twofish Função h

if k=4 **then**

$y_{3,0} = q_1[y_{4,0}] \oplus l_{3,0}$

$y_{3,1} = q_0[y_{4,1}] \oplus l_{3,1}$

$y_{3,2} = q_0[y_{4,2}] \oplus l_{3,2}$

$y_{3,3} = q_1[y_{4,3}] \oplus l_{3,3}$

end if

if (k$\geq$ 3) **then**

$y_{2,0} = q_1[y_{3,0}] \oplus l_{2,0}$

$y_{2,1} = q_1[y_{3,1}] \oplus l_{2,1}$

$y_{2,2} = q_0[y_{3,2}] \oplus l_{2,2}$

$y_{2,3} = q_0[y_{3,3}] \oplus l_{2,3}$

end if

$y_0 = q_1[q_0[q_0[y_{2,0}] \oplus l_{1,0}] \oplus l_{0,0}]$

$y_1 = q_0[q_0[q_1[y_{2,1}] \oplus l_{1,1}] \oplus l_{0,1}]$

$y_2 = q_1[q_1[q_0[y_{2,2}] \oplus l_{1,2}] \oplus l_{0,2}]$

$y_3 = q_0[q_1[q_0[y_{2,3}] \oplus l_{1,3}] \oplus l_{0,3}]$

$A = [y_0, y_1, y_2, y_3]$

retornar A X MDS

Dessa forma podemos gerar nossas *key dependent S-boxes* através da chamada h(X,S), ou seja para cada s_i pode ser gerado pelo mapeamento de x_i em y_i na função h, onde L é o vetor S derivado da chave.

As palavras da chave expandida são definidas através das seguintes regras:

- $= \rho = 2^{24} + 2^{16} + 2^8 + 2^0$
- A_i = h(2iρ,M_e)
- B_i = ROL(h((2i+1)ρ,M_o),8)
- $K_{2i} = (A_i + B_i) \bmod 2^{32}$
- $K_{2i+1} = ROL((A_i + B_i) \bmod 2^{32}, 9)$

A função g recebe como parâmetro uma palavra X e a separa em 4 *bytes*, que rodarão cada um em sua própria *S-box*. As *S-boxes* são bijetivas, ou seja, recebem 8 bits de *input* e retornam 8 *bits* de *output*, que serão interpretados como um vetor de tamanho 4 sobre o GF(2^8) e multiplicado pela matriz MDS, resultando num vetor que deverá ser interpretado como uma palavra de 32 *bits* que é o resultado da função G.

A função F é uma permutação dependente da chave em valores de 64 *bits*. Ela recebe duas palavras R_0 e R_1 e o número de *rounds* r utilizado para selecionar a sub-chave apropriada. Podemos definir a função F com o seguinte algoritmo:

Algorithm 16 Twofish Função F

$T_0 = g(R_0)$
$T_1 = g(ROL(R_1, 8))$
$F_0 = T_0 + T_1 + K_{2r+8} \bmod s^{32}$ // Pseudo hadamart transform (PTH)
$F_1 = T_0 + 2T_1 + K_{2r+9} \bmod s^{32}$ // Pseudo hadamart transform (PTH)
Retornar (F_0, F_1)

Sejam P_i e K_i valores do texto e da chave respectivamente. Podemos definir $R_{0,i}$ como o resultado do *whitening step*, ou seja o *xor* de uma palavra do texto (32 *bits*), com uma chave estendida.

Dessa forma, depois do *whitening step*, executaremos o seguinte algoritmo de encriptação em cada um dos 16 rounds:

Algorithm 17 Twofish Round

$(F_{r,0}, F_{r,1}) = F(R_{r,0}, R_{r,1}, r)$
$R_{r+1,0} = ROR(R_{r,2} \oplus F_{r,0}, 1)$
$R_{r+1,1} = ROL(R_{r,3}, 1) \oplus F_{r,1}$
$R_{r+1,2} = R_{r,0}$
$R_{r+1,3} = R_{r,1}$

Agora devemos desfazer o *whitening Step* para obtermos o texto criptografado. Esse texto deverá ser escrito em 16 bytes.

Implementação

Para a implementação do algoritmo Twofish foi utilizada a biblioteca Cripto++ pois, assim como o AES, este algoritmo possui diversas otimizações em Assembly para diferentes modelos de processadores.

2.3.6 RC4

O *RC4* é um algoritmo de cifragem de fluxo desenvolvido em 1987 e baseia-se em permutações randômicas exigindo de 8 a 16 operações para cada *byte* de *output* e é esperado que a cifra possua um alto desempenho a nível de software [7]. Em [9] é demonstrado que esse algoritmo possui melhor desempenho em tamanhos pequenos de dados.

O *RC4* foi amplamente utilizado nos padrões do *SSL/TLS*, de navegadores WEB, nos protocolos WEP e até mesmo nos recentes protocolos WPA, que são parte do IEEE 802.11. No passado esse algoritmo gerou uma certa insegurança entre os desenvolvedores devido a uma possível vulnerabilidade encontrada nesse algoritmo. Contudo tal vulnerabilidade foi era em virtude do protocolo WEP utilizado para gerar as chaves. [7] O *RC4* vem também sendo testado com implementações de hardware devido a sua simplicidade, com destaque para as novas implementações eficientes do código visando um menor consumo energético em *FPGAs* como proposto em [38] .

Em [31] é evidenciado que são necessários $6 * 2^{30}$ análises de encriptação de textos conhecidos para descobrir uma mensagem usando a mesma chave.

Algoritmo

O algoritmo consiste em usar uma chave para iniciar um *array* de estados que contém uma permutação de todos os números de 8 bits de 0 a 255. Ao array de estados daremos o nome de S.

Para a inicialização do *array* S, será necessário outro *array* de 256 posições, que nomearemos como T, que é um *array* que possui a chave expandida através de uma permutação circular até 256 *bytes*.

Algorithm 18 Inicialização S RC4

```
for i=0 até 255 do
    S[i]=i
    T[i]=K[i mod tamanho-da-chave]
end for
```

Agora todos os *bytes* de S deverão ser trocados, um a um, com o índice da troca anterior; zero caso seja a primeira, somado com o *byte* da chave e com o valor da posição atual mod 256. Dessa forma o vetor S estará iniciado. Segue o algoritmo:

Algorithm 19 Primeira Permutação S RC4

```
j=0
for i=0 até 255 do
    j = (j + S[i] + T[i]) mod256
    Swap(S[i],S[j])
end for
```

Agora, para a geração da *stream*, o vetor deve ser varrido posição a posição circularmente e posição atual trocada com a nova posição, que nada mais é que

soma (mod256) da posição nova anterior , sendo esta 0 caso seja a primeira, com o valor encontrado na posição atual.

Para criptografar faça a soma do valor encontrado na nova posição com o valor encontrado na posição atual mod256. Busque no vetor S o valor encontrado no índice que é resultado da soma das suas posições. Faça um *XOR* com o *byte* do texto a ser criptografado. Para descriptografar, basta aplicar o *XOR* novamente.

Algorithm 20 Geração Stream RC4

```
i,j=0
while 1 do
    i = (i+1) mod256
    j = (j+S[i]) mod256
    Swap(S[i],S[j])
    k = S[(S[i]+S[j])mod256]
    Texto-Criptografado[i]= Texto-Original[i] ⊕ k
end while
```

Implementação

Neste algoritmo serão propostas mudanças na forma como é comumente implementado. São elas:

- Remoção do *array* T, uma vez que o mesmo não se faz necessário já que o conteúdo armazenado será utilizado e uma única vez. Para isso basta realizarmos a permutação circular da chave fazendo uso de seu índice mod o tamanho da chave.

- sabemos que a divisão no quesito desempenho deixa muito a desejar, portanto é proposta uma forma de preencher o vetor S através da operação E bit a bit para a expansão da chave.

2.3.7 ChaCha

ChaCha é uma cifra de *stream* de 256 bits baseada na cifra *Salsa20* [21], porém sendo desenhada para possuir uma melhor difusão[26], por round, melhor resistência a criptanálise e, em algumas arquiteturas, um melhor desempenho por round [20]. É importante ressaltar que a difusão adicional não vem ao custo de operações adicionais, sendo inclusive capaz de usar um registrador a menos do que a cifra *Salsa20*.

O autor dessa cifra afirma [49] que uso de *S-boxes* causa pressão na cache L1 do processador forçando assim diferentes técnicas de implementação para *CPUs* de menor porte e que ainda, na maioria das plataformas, *S-boxes* são vulneráveis a

timming attacks, o que contradiz o que foi afirmado pelo *NIST*; ainda afirma que é muito difícil contornar o problema sem abrir mão de muita performance. Também é explicado sobre o impacto das rotações no desempenho, bem como todas as tomadas de decisões sobre como a cifra seria desenhada.

A cifra ChaCha possui um ganho de desempenho em todas as arquiteturas testadas em comparação com a *Salsa20* exceto a *x86 Athlon 622* e a *x86 Pentium 4 f12* [20]. Essa cifra possui ainda uma complexidade de tempo de $2^2 48$ e $2^2 7$ de complexidade de dados [36]

Vale destacar que Bernstein melhorou o *AES*[50] e afirmou que a cifra *Salsa20* é consistentemente mais rápida que o *AES* [20]

Algoritmo

Antes de analisarmos o funcionamento dos rounds de encriptação e desencriptação devemos entender como é o funcionamento da expansão das chaves, que nesse algoritmo é utilizada uma matriz contendo 4 chaves, uma palavra, 4 constantes e 4 *bytes* de *input*.

Dessa forma o algoritmo para preenchimento da matriz seria:

Algorithm 21 ChaCha Init

if **then**$tamanho = 32$
constante = "expand 32-byte k"
end if
if (**then**tamanho != 32)
constante = "expand 16-byte k"
end if
// LE pega os 4 bits a partir da posição demarcada
$M[0] = LE(constante + 0)$
$M[1] = LE(constante + 4)$
$M[2] = LE(constante + 8)$
$M[3] = LE(constante + 12)$
// Array key já está em palavras
$M[4] = key[0]$
$M[5] = key[1]$
$M[6] = key[2]$
$M[7] = key[3]$
$M[8] = key[16 \bmod tam]$
$M[9] = key[20 \bmod tam]$
$M[10] = key[24 \bmod tam]$
$M[11] = key[28 \bmod tam]$
$M[12] = 0$
$M[13] = 0$
$M[14] = LE(nonce + 0)$
$M[15] = LE(nonce + 4)$

Agora que temos a matriz pronta, devemos entender como funciona a função de *round*. Para tanto definamos um *Quarterround* que utiliza um vetor x externo.

Algorithm 22 ChaCha QuarterRound

QUARTERROUND(a,b,c,d)
$x[a] += x[b]; x[d] = x[d] \oplus x[a] <<< 16);$
$x[c] += x[d]; x[b] = x[b] \oplus x[c] <<< 12);$
$x[a] += x[b]; x[d] = x[d] \oplus x[a] <<< 8);$
$x[c] += x[d]; x[b] = x[b] \oplus x[c] <<< 7);$

Agora que compreendemos um *round* do *ChaCha* com base na ideia do que é um *Quarterround*

Algorithm 23 ChaCha Round

```
for (i de 0 a 16) do
    x[i] = input[i] // copiando para o array x
end for
for (i de 8 a 1; i-=2) do
    QUARTERROUND( 0, 4, 8,12)
    QUARTERROUND( 1, 5, 9,13)
    QUARTERROUND( 2, 6,10,14)
    QUARTERROUND( 3, 7,11,15)
    QUARTERROUND( 0, 5,10,15)
    QUARTERROUND( 1, 6,11,12)
    QUARTERROUND( 2, 7, 8,13)
    QUARTERROUND( 3, 4, 9,14)
end for
for (i de 0 a 16) do
    x[i] += input[i] // copiando para o array x
end for
for (i de 0 a 16) do
    U32TO8_LITTLE(output + 4 * i,x[i]); // convertendo para 8 bits o conteúdo
de x
end for
```

Para desencriptar basta rodar a encriptação com o texto cifrado.

Implementação

A implementação do ChaCha se deu de forma quase idêntica ao algoritmo proposto, porém sua a inicialização na posição 64 no lugar da inicialização como 0 como proposto no algoritmo, dessa forma são poupadas comparações durante toda a execução do algoritmo.

2.3.8 Trivium

Trivium é um algoritmo de cifragem em *stream* que foi desenvolvido com o propósito de descobrir o quanto um algoritmo desse tipo pode ser simplificado sem sacrificar a segurança, desempenho e flexibilidade. Apesar de seu uso não ser recomendado pelos autores dada a simplicidade, argumenta-se que se não forem desenvolvidos ataques, modelos simples inspiram mais confiança do que modelos complexos.

Trivium assim como o ChaCha foi desenhado para gerar até 2^{64} bits de *stream*, porém esse algoritmo utiliza chaves de 80 *bits* e *IVs* de 80 *bits*.

Como demonstrado pelos autores, sua segurança é questionável, tornando-se ainda mais fina após a publicação dos *cube attacks* [37], o que resultou em um aumento no número de *rounds* para garantir a segurança da cifra. Mas o Trivium possui um alto *throughput* e um baixo consumo de área sendo ideal para implementações em hardware [41].

Devido sua simplicidade estima-se que em software ele também obtenha bons resultados.

Algoritmo

O algoritmo pode ser dividido em duas partes a inicialização de chaves e valores iniciais, *Initial Values (Ivs)*, e a geração de *stream*.

O algoritmo para a geração de chaves consiste basicamente de carregar a chave e os *Ivs* preenchendo com zeros os espaços entre eles e todos os demais índices do vetor S com exceção das 3 últimas posições que serão sempre 1. Após isso são calculados 3 números com base no preenchimento de S e realizada quatro rotações completas no vetor. Note que S e T são dados em *bits*, logo as operações de soma e multiplicação são em GF(2), ou seja, *XOR* e *AND* respectivamente.

O algoritmo pode ser visto a seguir:

Algorithm 24 Trivium Inicialização

for i de 0 a 80 **do** // 80 da chave
s[i] = chave[i]
end for
for i de 81 a 93 **do**
s[i] = 0
end for
for i de 94 a 173 **do** // 80 dos IVs
s[i]=IV[i]
end for
for i de 174 a 177 **do**
s[i] = 0
end for
for i de 178 a 285 **do**
s[i] = 0
end for
for i de 286 a 288 **do**
s[i] = 1
end for
for i de 1 a 1152 **do** //(4*288)
$t_1 = s_{66} + s_{91} * s_{92} + s_{93} + s_{171}$
$t_2 = s_{162} + s_{175} * s_{176} + s_{177} + s_{264}$
$t_3 = s_{243} + s_{286} * s_{287} + s_{288} + s_{69}$
for i de 93 a 2 **do**
s[i] = S[i-1]
end fors[1]=t3
for i de 177 a 95 **do**
s[i] = S[i-1]
end fors[94]=t1
for i de 288 a 179 **do**
s[i] = S[i-1]
end fors[178]=t2
end for

O algoritmo para a geração da *stream* faz uso do S gerado na inicialização para calcular 3 variáveis t1,t2 e t3 cuja soma produz um bit da *stream*. Após esse cálculo são calculadas novas variáveis com base nas anteriores e realizada uma nova rotação. Esse processo se repete até que tenham sido calculados todos os *bits* da *stream* ou que tenha chegado ao limite de 2^64 *bits* calculados.

Segue o algoritmo:

Algorithm 25 Trivium Key Stream

for i de 1 a N **do** // tamanho da stream

$t_1 = s_{66} + s_{93}$

$t_2 = s_{162} + s_{177}$

$t_3 = s_{243} + s_{288}$

$z_i = t_1 + t_2 + t_3$

$t_1 = t_1 + s_{91} * s_{92} + s_{171}$

$t_2 = t_2 + s_{175} * s_{176} + s_{264}$

$t_3 = t_3 + s_{286} * s_{287} + s_{69}$

for i de 93 a 2 **do**

s[i] = S[i-1]

end fors[1]=t3

for i de 177 a 95 **do**

s[i] = S[i-1]

end fors[94]=t1

for i de 288 a 179 **do**

s[i] = S[i-1]

end fors[178]=t2

end for

Implementação

É simples percebermos que 288 *bits* são 36 posições de 1 *byte*, um *char*, e que 80 *bits* são na verdade 10 posições de 1 *byte*, dessa forma, nosso vetor S consiste de 36 posições e os vetores chave e IV possuem 10 posições cada.

Para essa implementação foi utilizada a estrutura de dados *bitset* permitindo assim o acesso aos *bits* em tempo O(1) e rápido deslocamento de dados para rotação e fácil leitura da implementação.

Capítulo 3

Resultados e Discussões

Neste capítulo estaremos exibindo os comparativos de desempenho dos algoritmos implementados no capítulo anterior em processadores Intel, AMD e em dispositivo móvel. Serão também comparados os resultados obtidos nos computadores com uma versão que utiliza a biblioteca Cripto++ a fim de comparação de desempenho e consequentemente eficiência energética.

Os testes em desktop foram realizados em um processador AMD FX8300 e em um processador Intel core i5 4200U ambos com o sistema operacional Ubuntu 16.04. Já para os testes em dispositivo móvel foi utilizado um Sony Xperia J, com um Qualcomm MSM7227A Snapdragon e sistema operacional Android 4.1.2.

As medições de tempo foram feitas através da biblioteca Chrono para C++11 a fim de captar mesmo as pequenas variações nos computadores desktops. Sistemas Android já são capazes de captar pequenas variações de tempo.

As medições de ciclos de clock em sistemas Linux foram feitas através da biblioteca perf para esse sistema operacional.

Medições de consumo energético em sistemas Android foram feitas através do Power Tutor [51] que é capaz de medir consumo energético de aplicações em Joules.

Em desktops foram realizadas 1000 medições e em dispositivos móveis foram realizadas 10 medições. A escolha de realização de 10 medições em sistemas Android, se deve ao fato de necessitar de interferência humana para interação com o aviso de que o aplicativo está demorando para responder, caso contrário parte da medição seria contabilizada como pertencente ao consumo do sistema operacional.

As implementações não puderam ser comparadas com a biblioteca Cripto++ em dispositivos móveis por limitações dos aparelhos disponíveis. Para a execução da biblioteca em aparelhos celulares faz-se necessária uma *API level* 21 no mínimo, o aparelho utilizado possui uma *API level* 16.

3.1 Avaliações dos resultados obtidos

Para que possamos fazer uma comparação entre os algoritmos que fazem uso das mudanças propostas no capítulo anterior e aqueles que utilizam a biblioteca Cripto++ serão comparadas as razões entre o maior e menor valor encontrado em cada um dos testes realizados.

Aqueles resultados que possuírem valores maiores do que os encontrados utilizando a biblioteca, seja por tempo ou consumo de ciclos, serão representados com um valor negativo e orientação para baixo no gráfico; aqueles que possuírem valores menores, serão representados por valores positivos e orientação para cima.

Serão também comparados os valores dos resultados obtidos com as medições realizadas com as cifras AES e o Twofish, a fim de averiguar os desempenhos das cifras implementadas em relação ao padrão internacional e com um competidor com alto desempenho em software.

Em cada teste realizado será comparado o seu resultado em relação a variações do tamanho da chave, podendo ter tamanhos de 2,4,8,16,32,64,128 e 256. Cada uma das chaves foi escolhida aleatoriamente entre o conjunto das possíveis chaves daquele tamanho e utilizada em todos os algoritmos. Foram descartadas chaves de tamanhos não suportados pela biblioteca Cripto++ nos gráficos, uma vez que não seria possível realizar uma comparação entre os resultados obtidos e os resultados da biblioteca para estes tamanhos de chave.

Nas medições em desktops não serão incluídos gráficos ou tabelas sobre os desvios padrão sobre as medições de ciclos, nem as diferenças entre as variâncias de tempo porque a primeira possui uma variação máxima inferior a 0,1%, já a segunda possui variâncias da ordem de 10^{-12}.

Para medições que fazem uso de sistemas Android, devido ao baixo número de medições, será apresentado tanto o desvio padrão quanto o intervalo de confiança das medições realizadas.

Será também exibido o consumo de energia separadamente entre CPU e LCD a fim de determinar o possível impacto de novas tecnologias de display no consumo total. Essa distinção traz uma maior precisão aos dados obtidos pelos resultados, uma vez que podemos averiguar quanto efetivamente um algoritmo consome e quanto do seu consumo é tomado pelo LCD.

Outro dado a ser avaliado é o intervalo de confiança dos valores encontrados. Tal parâmetro torna-se importante nas avaliações em dispositivos móveis devido a baixa quantidade de medições. O Intervalo de confiança é o intervalo estimado de um parâmetro de interesse utilizado para indicar a confiabilidade de uma estimativa, logo um intervalo de confiança pode ser usado para descrever o quanto os resultados de uma pesquisa são confiáveis. Temos como ideia, construir um intervalo de

confiança para um determinado parâmetro com uma probabilidade $1 - \alpha$ de que o intervalo contenha o verdadeiro parâmetro. Neste trabalho utilizaremos um intervalo de confiança de 95%, ou seja, um alfa de 0,05, e nosso parâmetro será a média. O cálculo do intervalo de confiança realizado faz uso de uma distribuição normal.

3.2 Maiores informações

Todos os códigos fontes executados bem como as saídas dos mesmos, como desvios padrões e variâncias podem ser encontrados em:

https://www.dropbox.com/sh/dg3o3mqnjzjawza/AABkVO8zuHdaycUH8Sbputea?dl=0

3.3 Resultados AMD

Nesta seção serão apresentados os resultados dos testes em um desktop de arquitetura AMD, sistema operacional Ubuntu 16.04, 8 GB RAM DDR3, processador FX8300.

Como podemos observar nas figuras 3.1 e 3.2, com exceção da implementação do Trivium com utilização de bitset, todos os demais algoritmos obtiveram substancial diminuição no número de ciclos e instruções utilizados; Apesar disso apresentar um forte indicativo de ganho de desempenho, somente esse dado é insuficiente para uma análise completa do desempenho, portanto foram realizados testes comparativos de tempo sobre as etapas de inicialização e encriptação, figuras 3.3 e 3.4 respectivamente, para que seja mais claro onde cada uma das implementações possui um maior destaque.

Como pode ser percebido os tempos de inicialização, figura 3.3, da biblioteca possuem, em geral, um ganho de desempenho em relação às implementações propostas. Tal ganho pode ser devido à utilização de bibliotecas de alocação rápida de dados, o que não é utilizado nas versões propostas, uma vez que, o enfoque é manter o código portável entre sistemas fazendo assim uso somente das bibliotecas padrão da linguagem C++.

Devemos também ressaltar que, de maneira geral, os tempos de encriptação das versões propostas possuem um *speedup* em relação àqueles que utilizam a biblioteca e, dessa forma, torna-se possível afirmar que para uma entrada grande o suficiente, as perdas de performance nos custos de inicialização serão supridas pelo ganho de desempenho das encriptações.

Uma vez comparados os ganhos de desempenho de cada um de seus equivalentes na biblioteca, vamos compará-los com o AES, algoritmo que se tornou o padrão internacional, e o Twofish, algoritmo que possui um alto desempenho em software

e que foi o segundo colocado na competição que elegeu o AES como o padrão internacional.

Conforme pode ser observado nas figuras 3.5 e 3.6, todas as implementações possuem diminuição de consumo de ciclos e total de instruções executadas em relação ao AES e ao Twofish, dessa forma havendo um forte indício de melhoria de desempenho em sua execução.

Observemos os testes comparativos de tempo de inicialização e tempo de encriptação de ambos os algoritmos. Como pode ser notado o custo de inicialização, figura 3.7, da maior parte desses algoritmos é maior do que o custo de inicialização das versões do AES e do Twofish disponibilizadas pela biblioteca.

Tanto o uso de bibliotecas alocação rápida quanto o código ser escrito em Assembly, podem estar provocando esse ganho de desempenho, contudo, com exceção do trivium, todos os algoritmos propostos possuem menores tempos de encriptação ao compararmos com todas as versões dos algoritmos AES e Twofish analisadas, como evidenciado na figura 3.8.

Figura 3.1: Comparativo normalizado de ciclos AMD

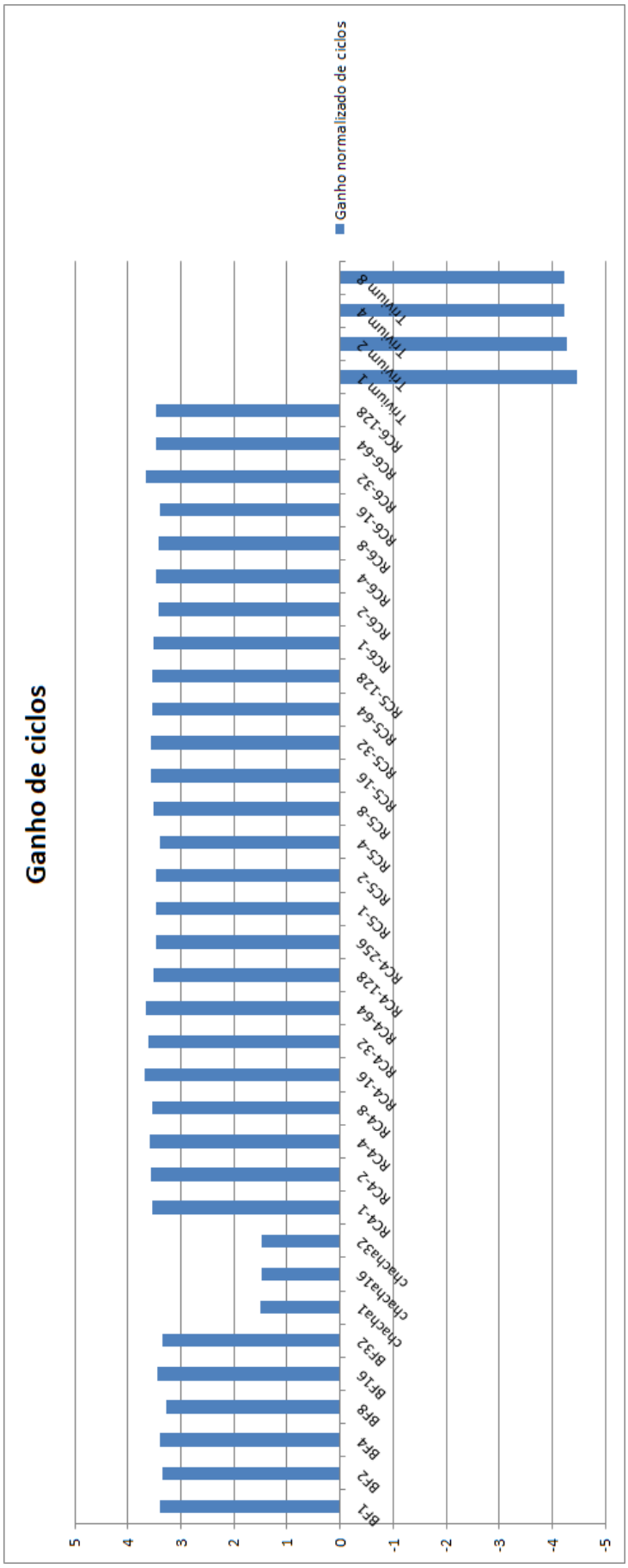


Figura 3.2: Comparativo de número de instruções executadas AMD

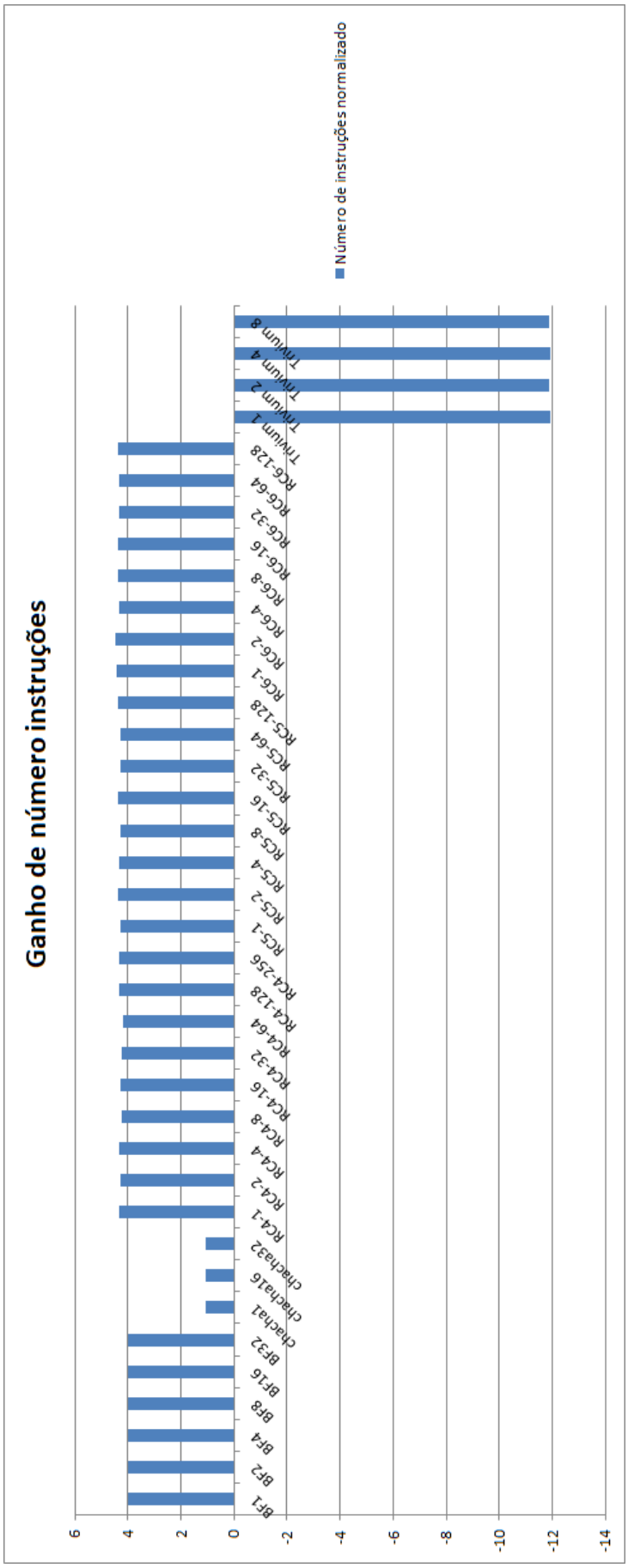


Figura 3.3: Comparativo de tempos de inicialização AMD

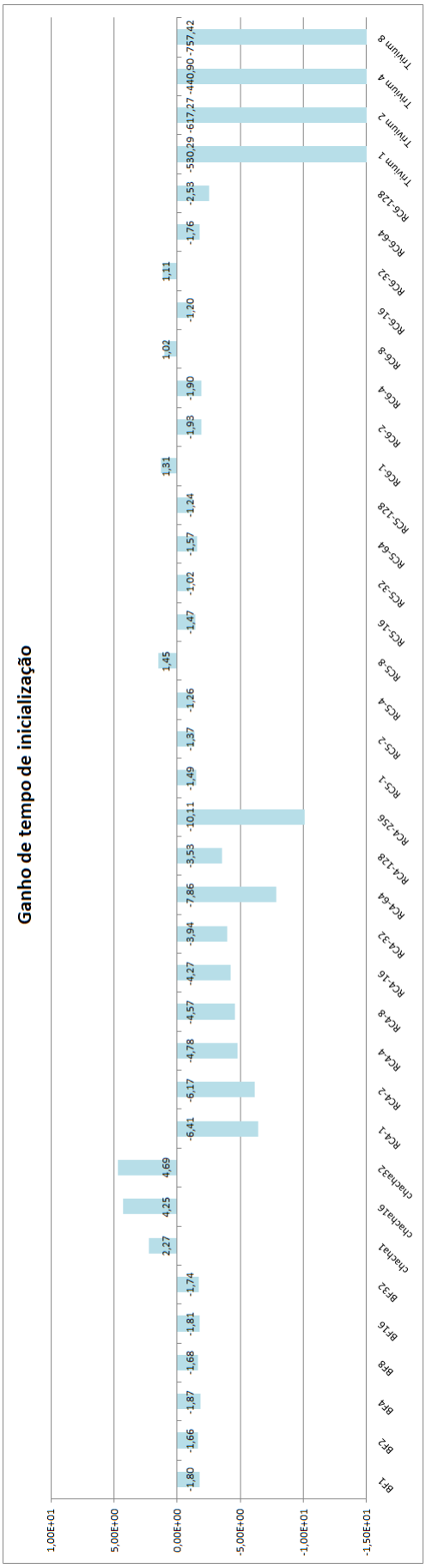


Figura 3.4: Comparativo normalizado de tempos de encriptação AMD

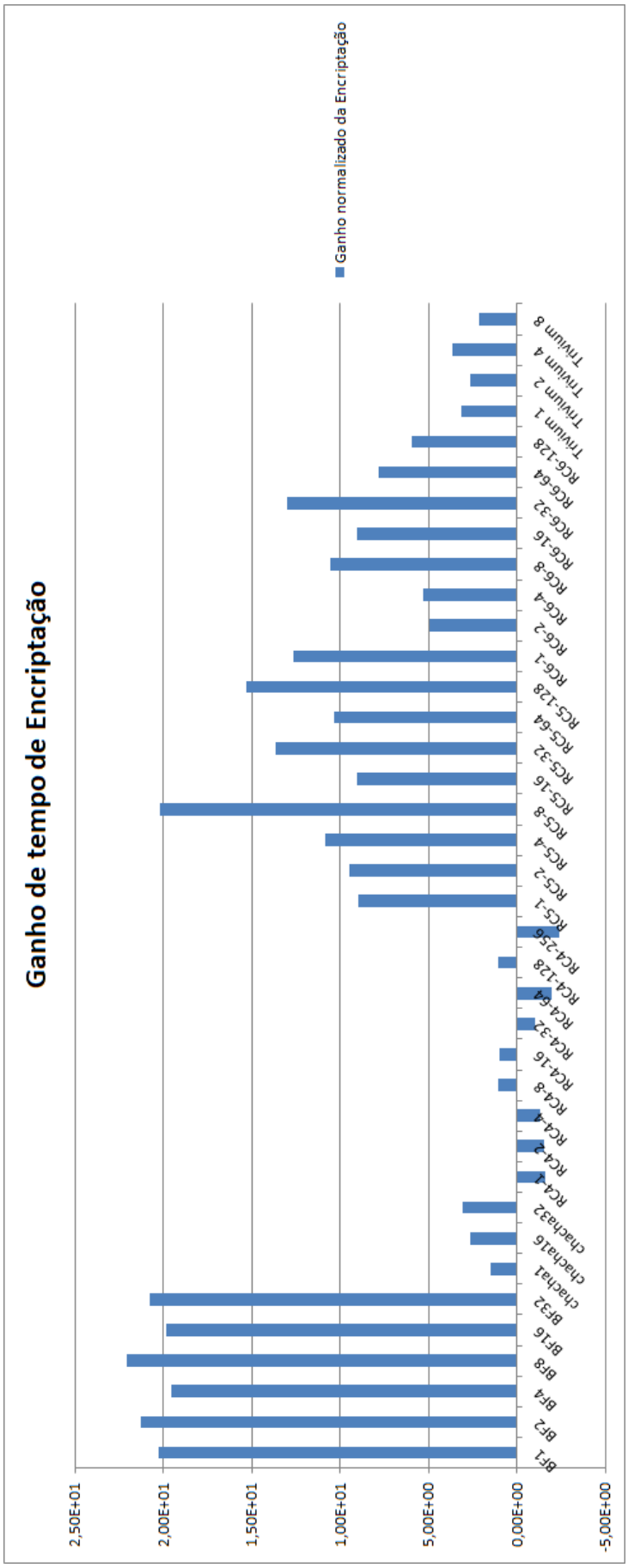


Figura 3.5: Comparativo de ciclos AMD

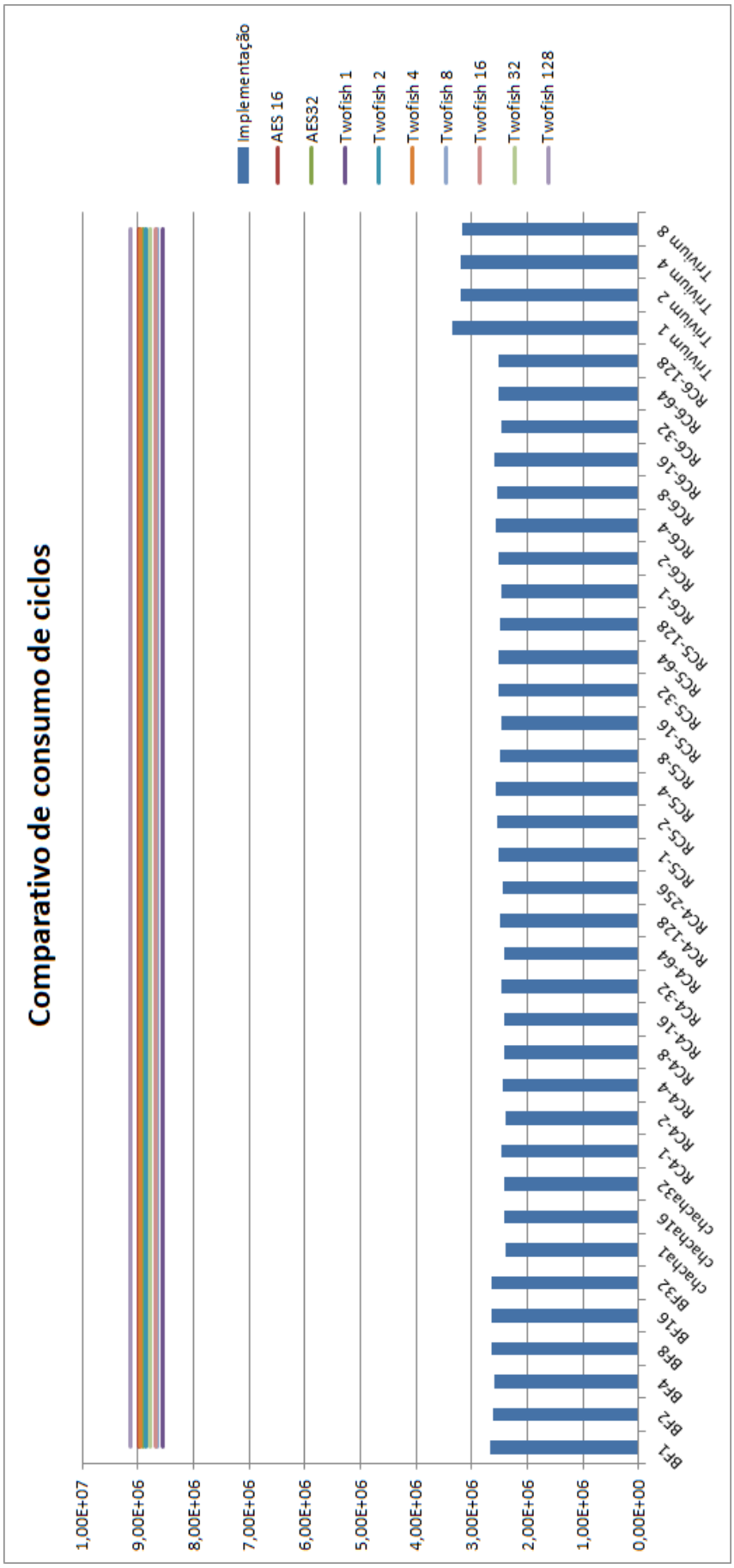


Figura 3.6: Comparativo do número de instruções executadas AMD

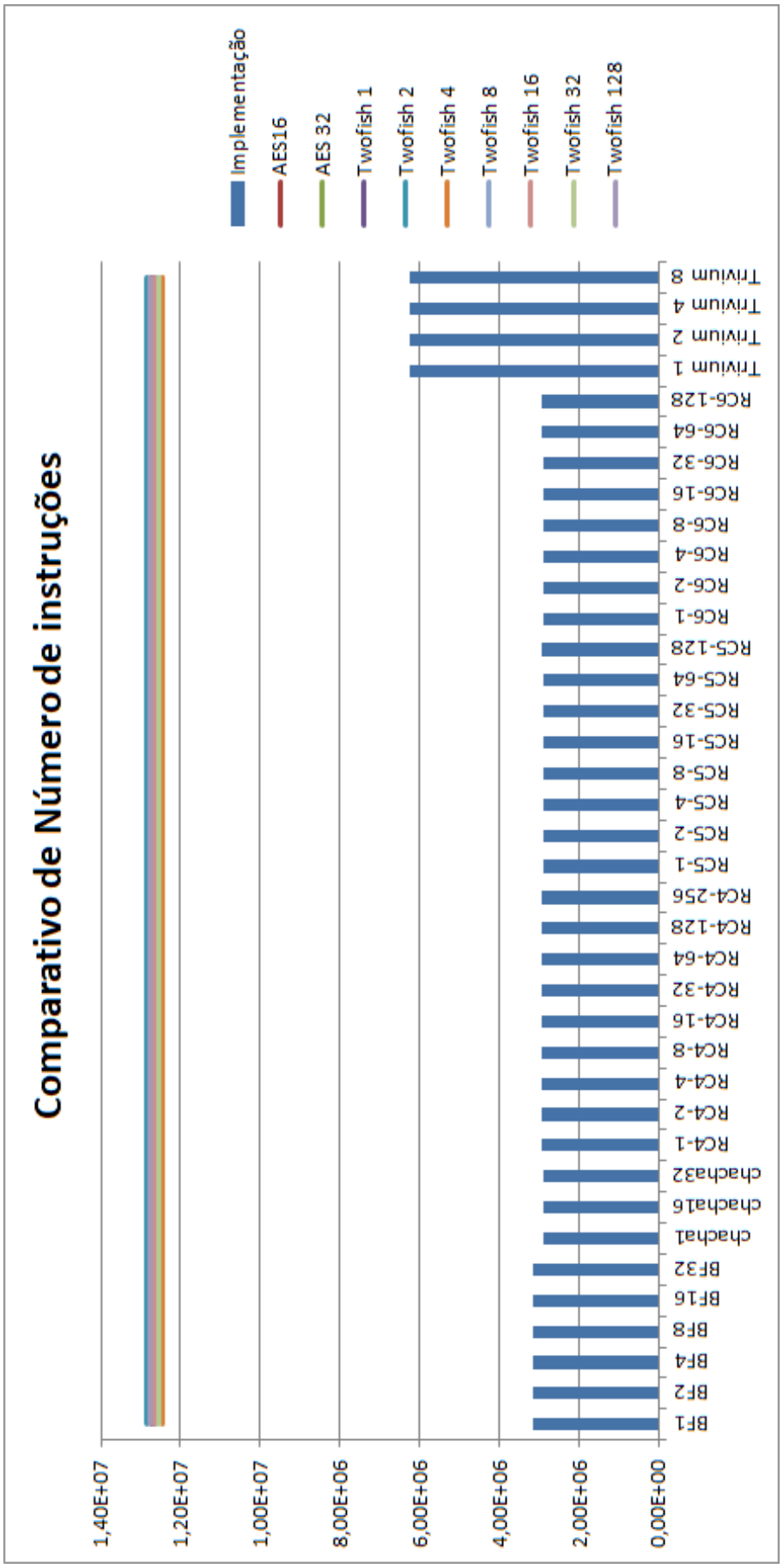


Figura 3.7: Comparativo de Inicialização AMD

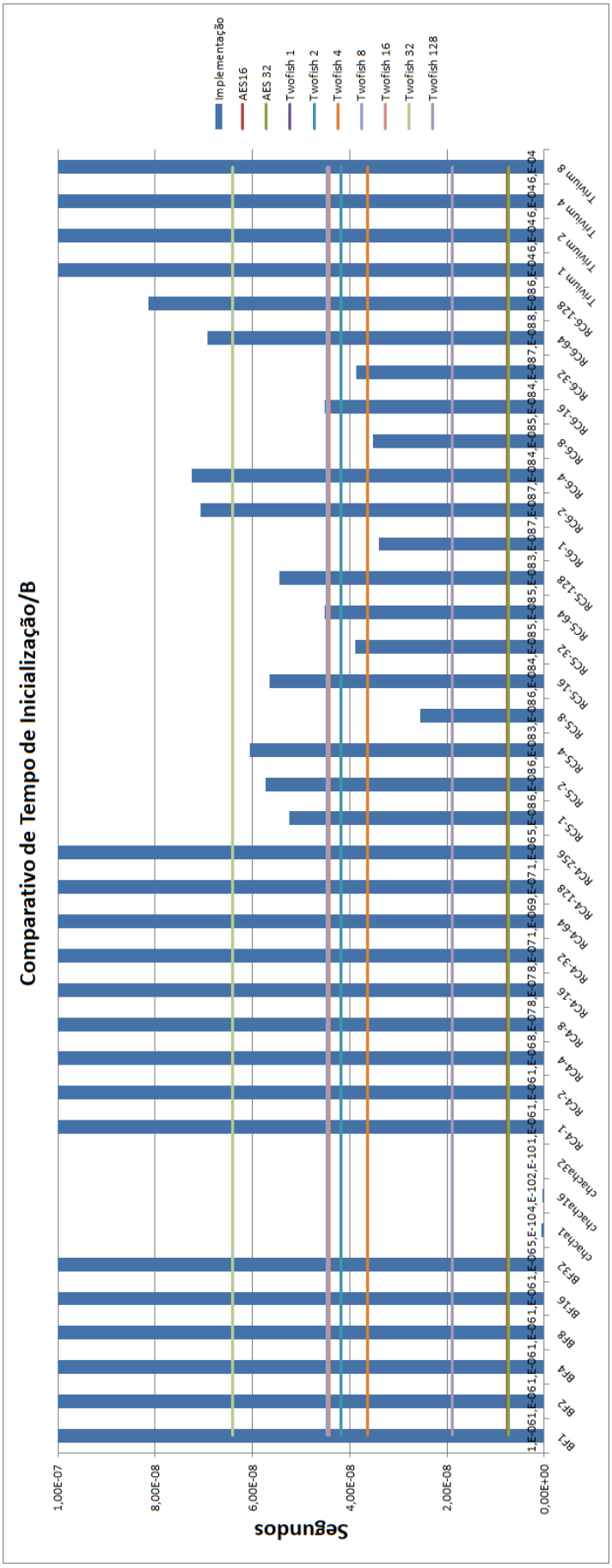
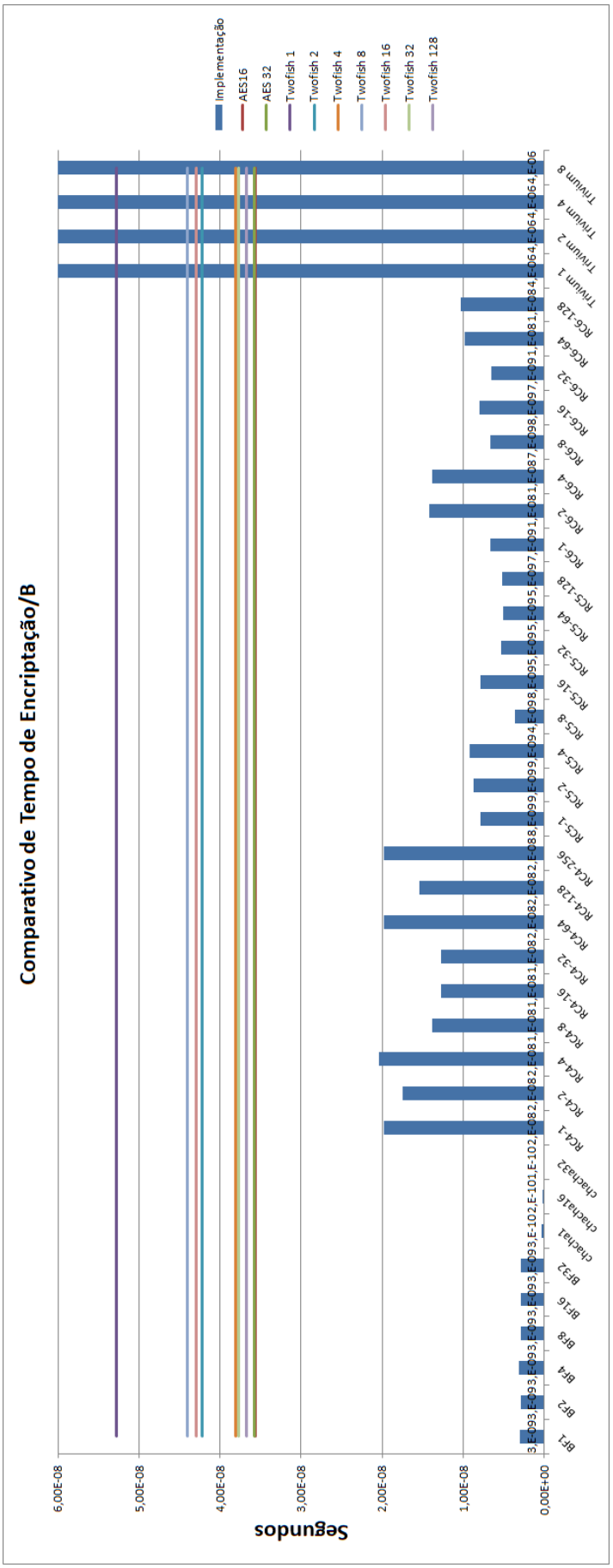


Figura 3.8: Comparativo de tempos de encriptação AMD



3.3.1 Resultados Intel

Após a verificação de desempenho em uma arquitetura de processador, faz-se necessária a verificação em uma arquitetura diferente a fim de garantir que os dados obtidos não estão de alguma forma otimizados para melhor atender a um modelo de processador e não a outros. Por essa forma também foi verificado o desempenho dos algoritmos em um processador Intel core i5 4200U, 4 GB RAM DDR3, sistema operacional Ubuntu 16.04.

Assim como na seção anterior foram medidos os ganhos de desempenho com relação aos ciclos utilizados bem como o número de instruções e realizada as razões entre os maior e o menor valor encontrados entre os algoritmos propostos e seus correspondentes utilizando a biblioteca Cripto++. Tais resultados são evidenciados nas figuras 3.9 e 3.10.

Os ganhos normalizados de desempenho, nesses dois quesitos, das implementações em arquitetura Intel foram ainda maiores do que os obtidos nas arquiteturas AMD. As figuras 3.11 e 3.12 evidenciam tais ganhos de desempenho.

De maneira geral os resultados obtidos pela inicialização continuam sendo inferiores àqueles propostos pela biblioteca e os ganhos de desempenho no processo de encriptação também possuem um melhor desempenho com relação à versão proposta pela biblioteca. Existem dois pontos importantes que merecem destaque na avaliação desses gráficos, o algoritmo ChaCha, com a implementação proposta, possui uma perda de desempenho quando implementado na arquitetura Intel e o algoritmo Trivium possui um slowdown muito maior do que aquele obtido em arquiteturas AMD.

Avaliaremos o ganho de desempenho desses algoritmos em relação ao AES e ao Twofish como feito anteriormente. Como pode ser observado nas figuras 3.13 e 3.14, o número de instruções e ciclos dos algoritmos propostos é menor que as versões do AES e Twofish propostas pela biblioteca.

Agora devemos comparar os tempos de inicialização e encriptação para que dessa forma consigamos resultados mais claros sobre essa comparação entre os algoritmos. Como podemos observar na figura 3.15, os tempos de inicialização são muito maiores que os comparados ao AES e ao Twofish propostos pela biblioteca, contudo os tempos de encriptação, figura 3.16, com exceção do Trivium, são muito menores revelando assim, que para uma entrada grande o suficiente os algoritmos propostos irão possuir resultados melhores que os demais comparados.

Figura 3.9: Comparativo normalizado de ciclos Intel

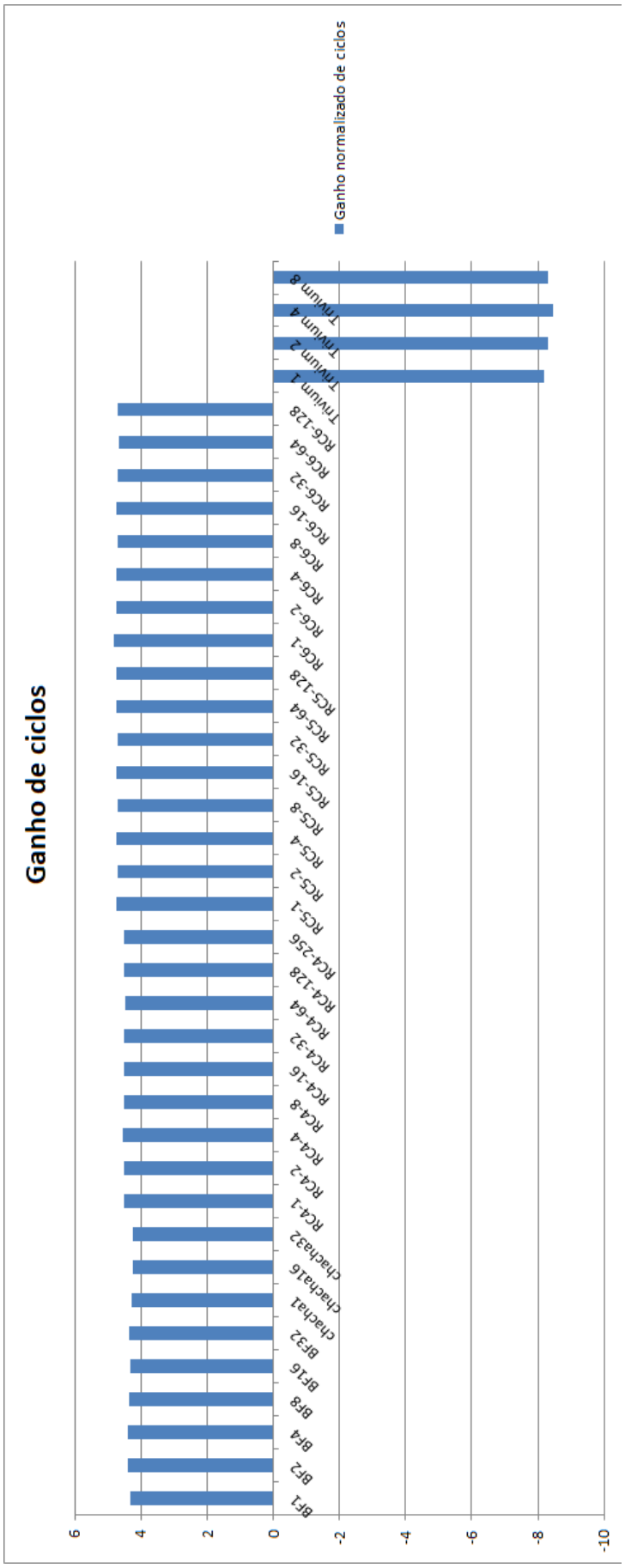


Figura 3.10: Comparativo normalizado do número de instruções executadas Intel

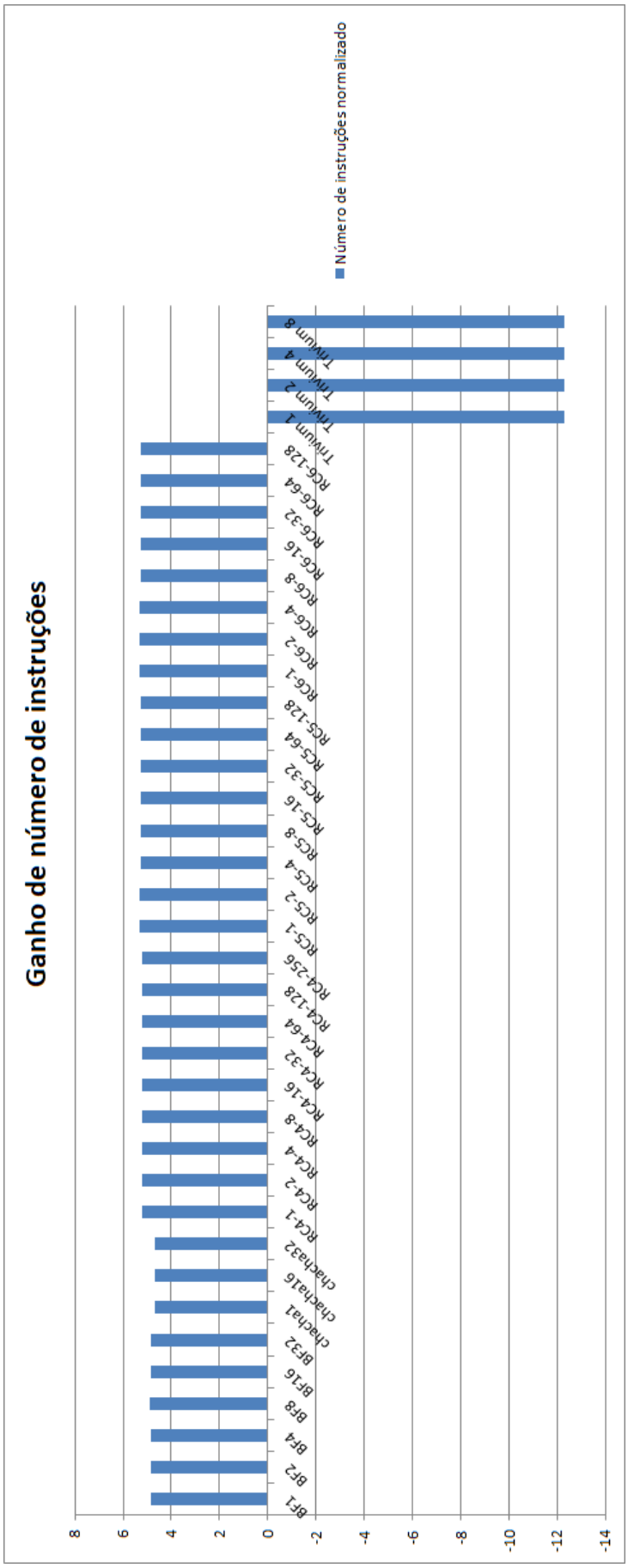


Figura 3.11: Comparativo normalizado do tempo normalizado de inicialização Intel

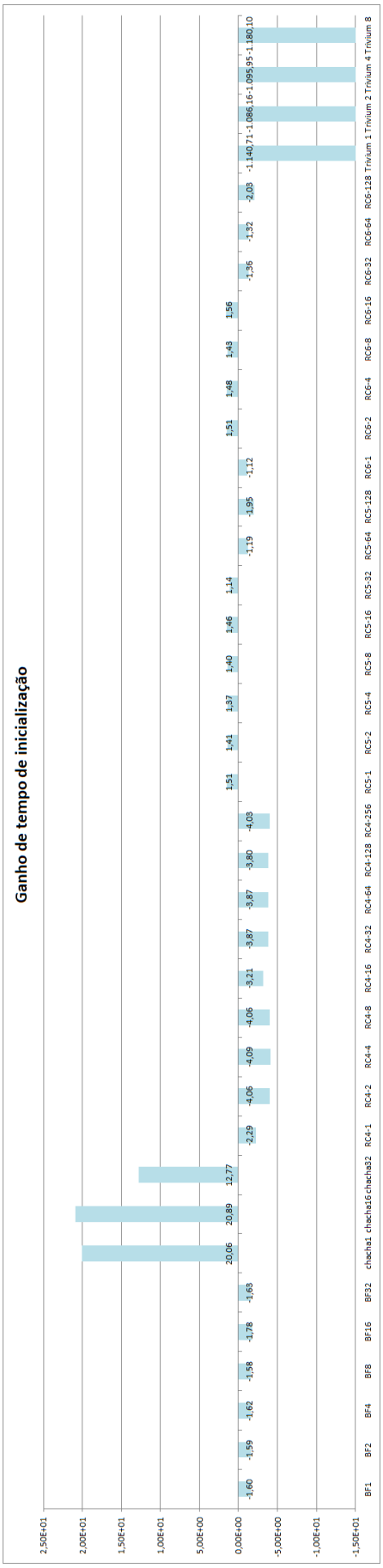


Figura 3.12: Comparativo normalizado do tempo normalizado de encriptação Intel

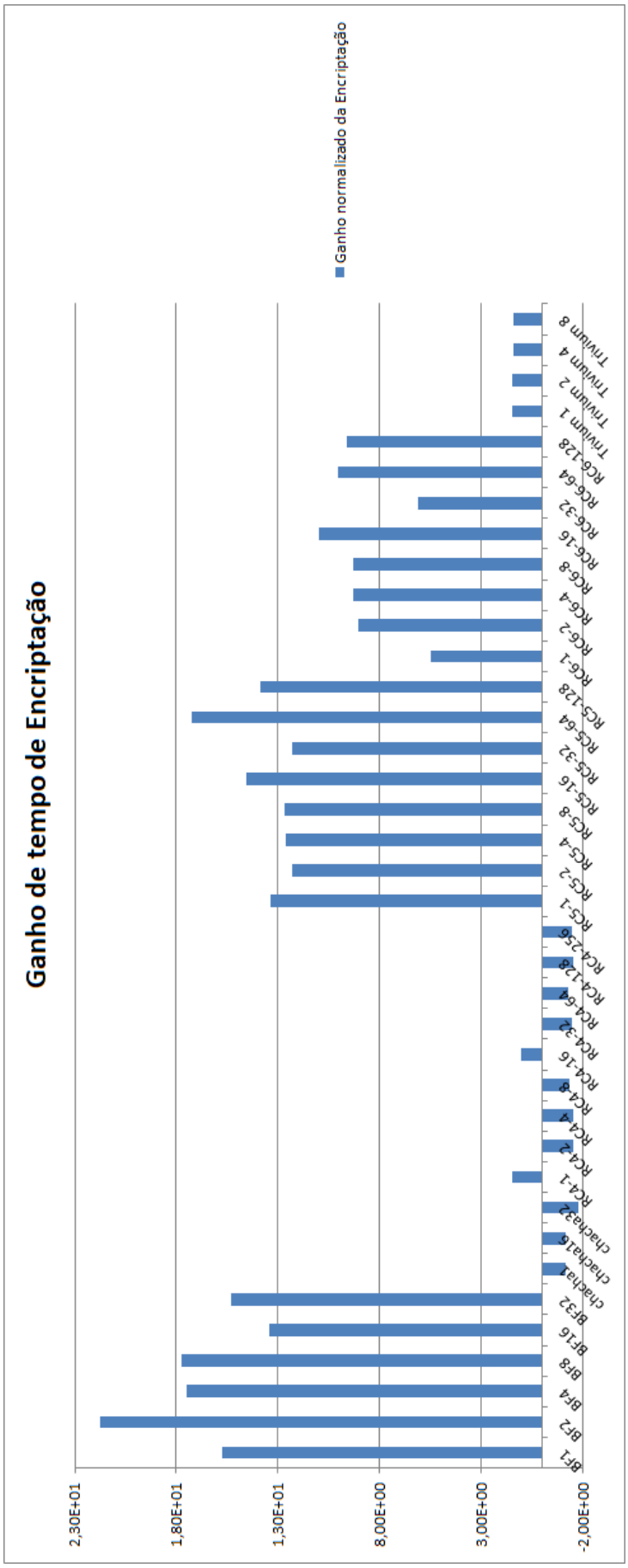


Figura 3.13: Comparativo do número de ciclos totais gastos Intel

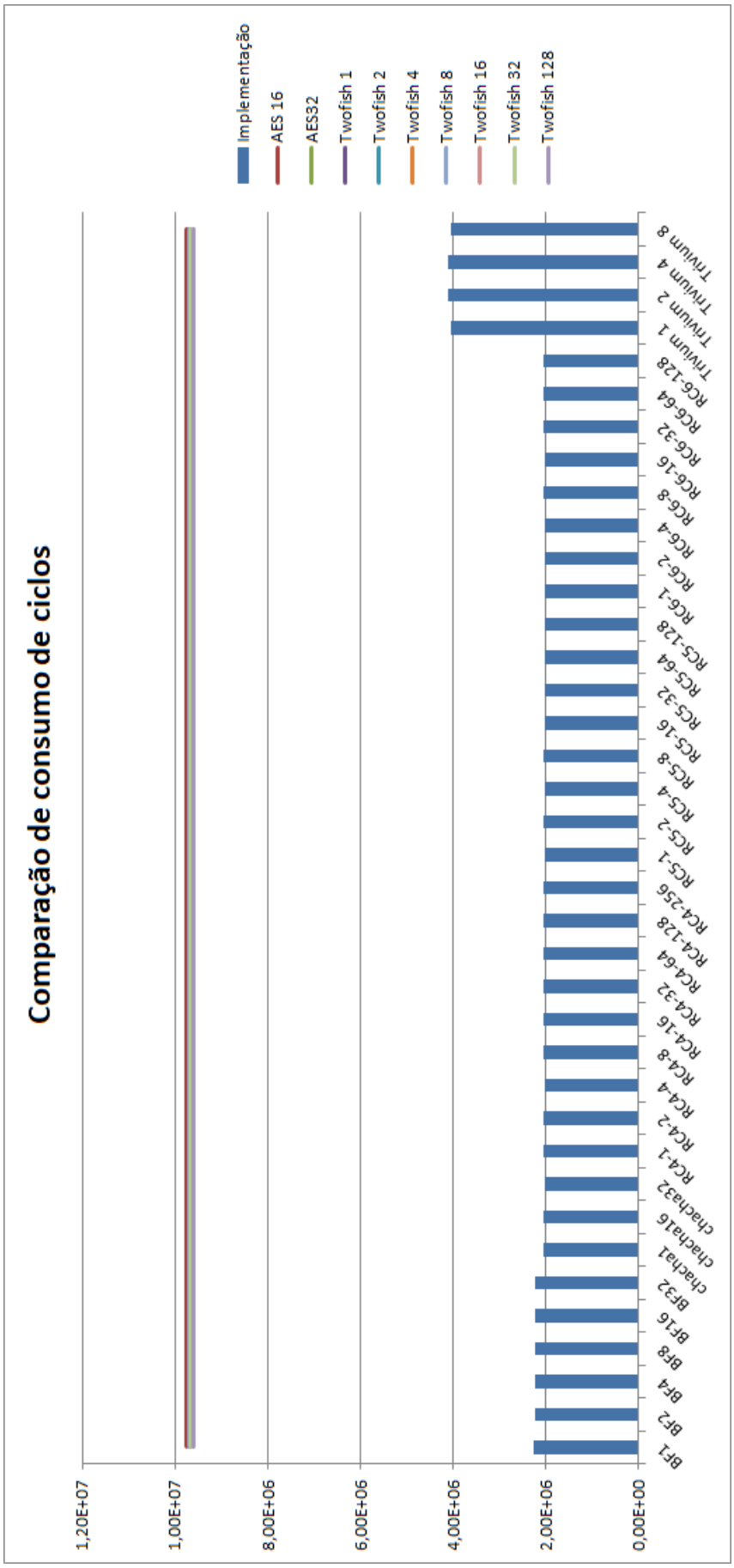


Figura 3.14: Comparativo do número total de instruções executadas Intel

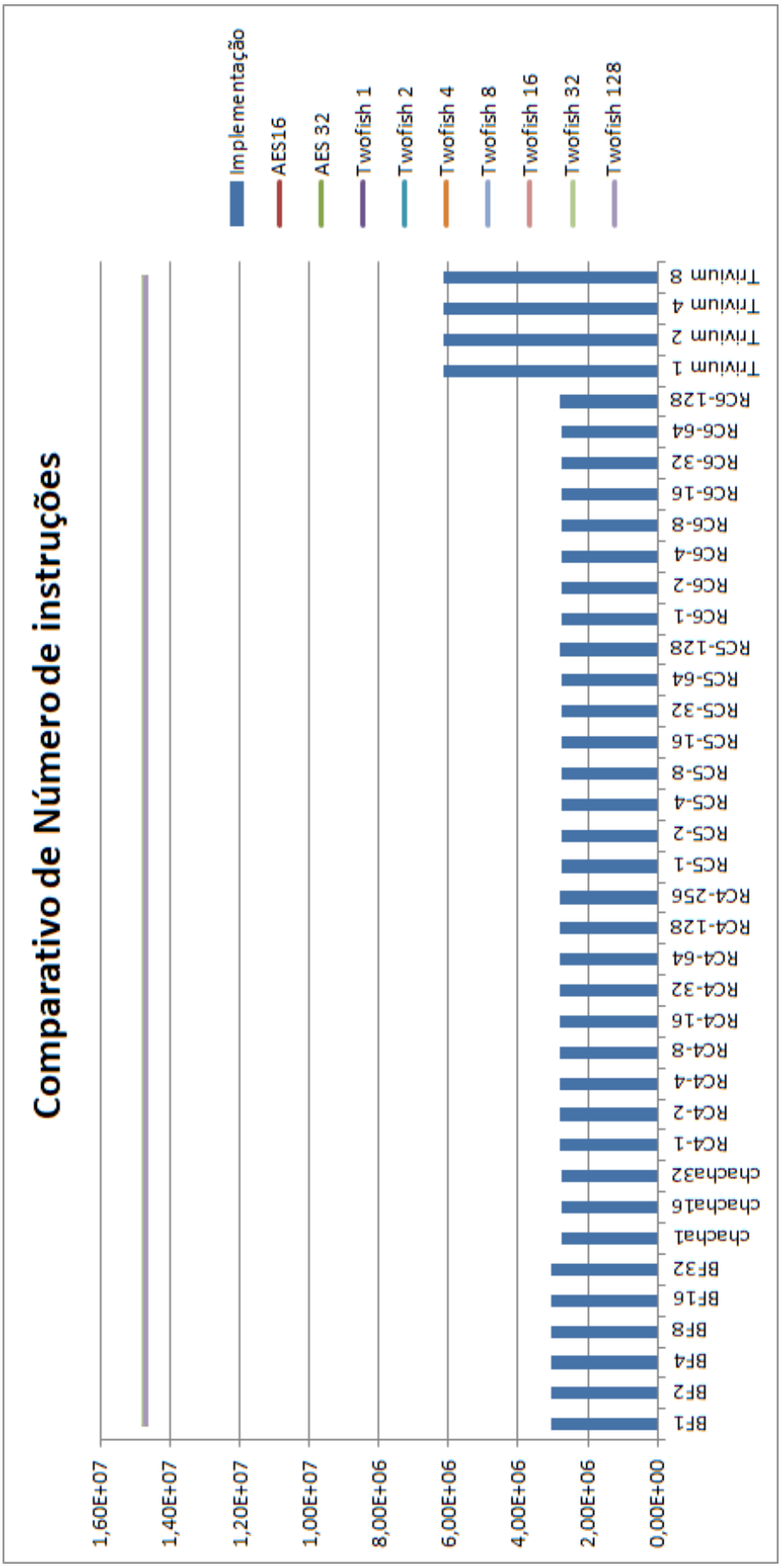


Figura 3.15: Comparativo tempo de inicialização Intel

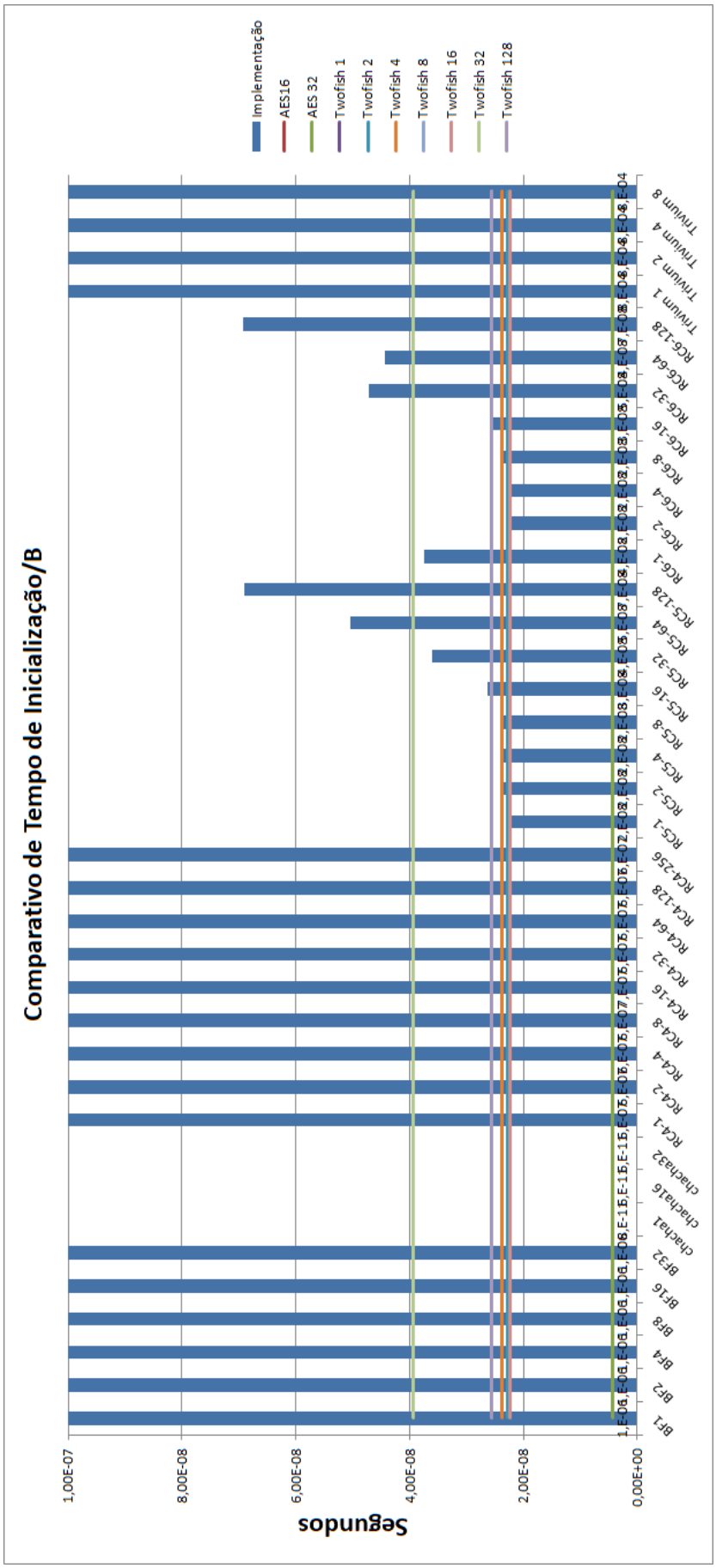
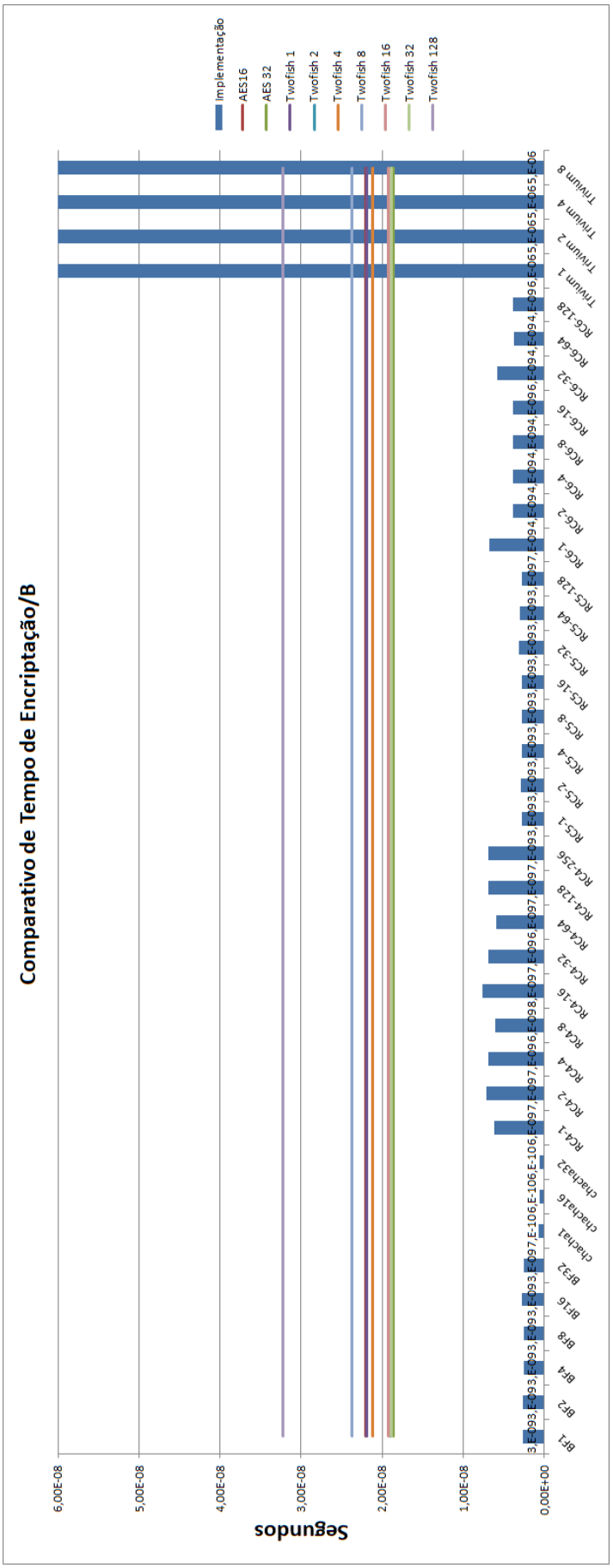


Figura 3.16: Comparativo do tempo de encriptação Intel



3.3.2 Resultados Xperia J

Os resultados em dispositivos moveis são relevantes uma vez que em um cenário ICN-IOT haverá grande número desses tipos de dispositivos. Por esse motivo foi utilizado um Smartphone com o sistema operacional Android que é amplamente utilizado. Para sua a implementação de seus códigos foi utilizado o software Android Studio e para o suporte às bibliotecas do C++ foi utilizada a NDK.

Para este tipo de dispositivo o consumo energético é de extrema importância devido ao "gargalo" ser na bateria e não no processador. Os gráficos 3.17,3.18,3.19 e 3.20 são resultados de 10 medições, seu consumo dado em Joules e seu tempo em segundos. Para o Trivium foram realizadas 10^4 encriptações, para os demais 10^6 encriptações.

Tais valores levam em consideração o tempo de execução do algoritmo, mas assim como nas versões para Intel e AMD estes valores representam encriptação de um diferente número de bytes. Portanto para uma melhor comparação entre algoritmos será apresentada uma representação de consumo e tempo por byte encriptado.

Devido à baixa quantidade de medições, faz-se necessária a exibição de seus desvios padrões, bem como seu intervalo de confiança a fim de exibir as variações de seus dados. Para seu intervalo de confiança foi utilizado um alpha de 95%.

Tabela 3.1: Desvios Padrão XPeria J

	Chacha	Trivium	Blowfish	Rc4	Rc5	Rc6
Consumo (J)	35,69581519	302,9186985	735,527022	110,441526	61,8802284	37,22587063
Tempo (S)	2,516523	21,72131	11,83497	3,07393	3,517849	5,205957

Tabela 3.2: Confiança 0.05

	Chacha	Trivium	Blowfish	Rc4	Rc5	Rc6
Consumo (J)	22,12408893	187,7475045	455,875994	68,45110918	38,35305816	23,07240969
Tempo (S)	1,559729	13,46276	7,335257	1,905206	2,180345	3,226626

Figura 3.17: Tempo Médio bor byte Encriptação Xperia em Joules

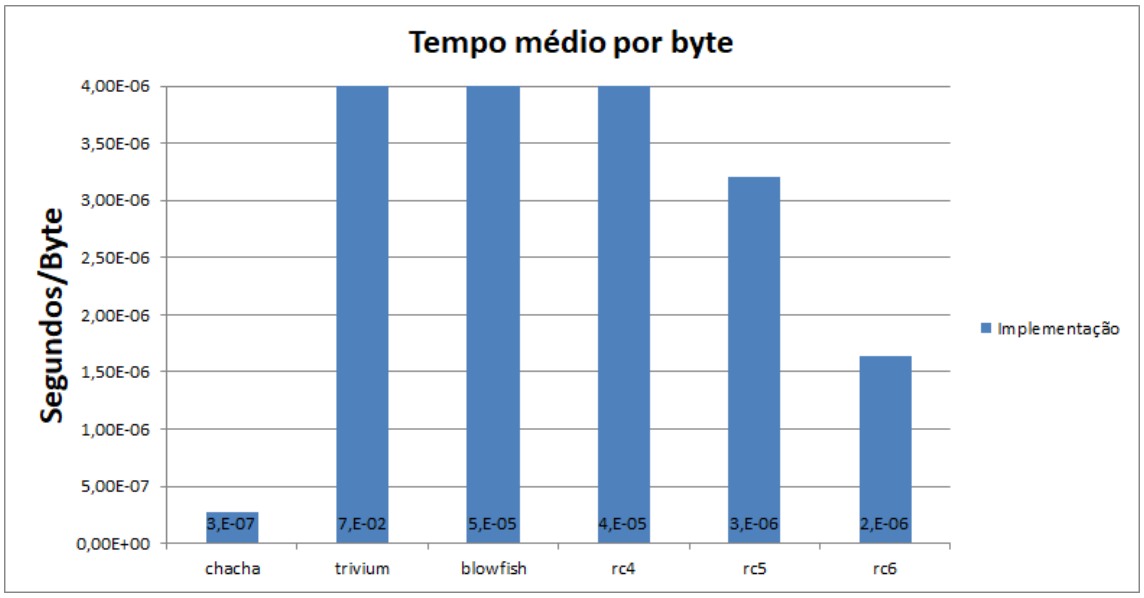


Figura 3.18: Consumo Médio por byte Encriptação Xperia em Joules

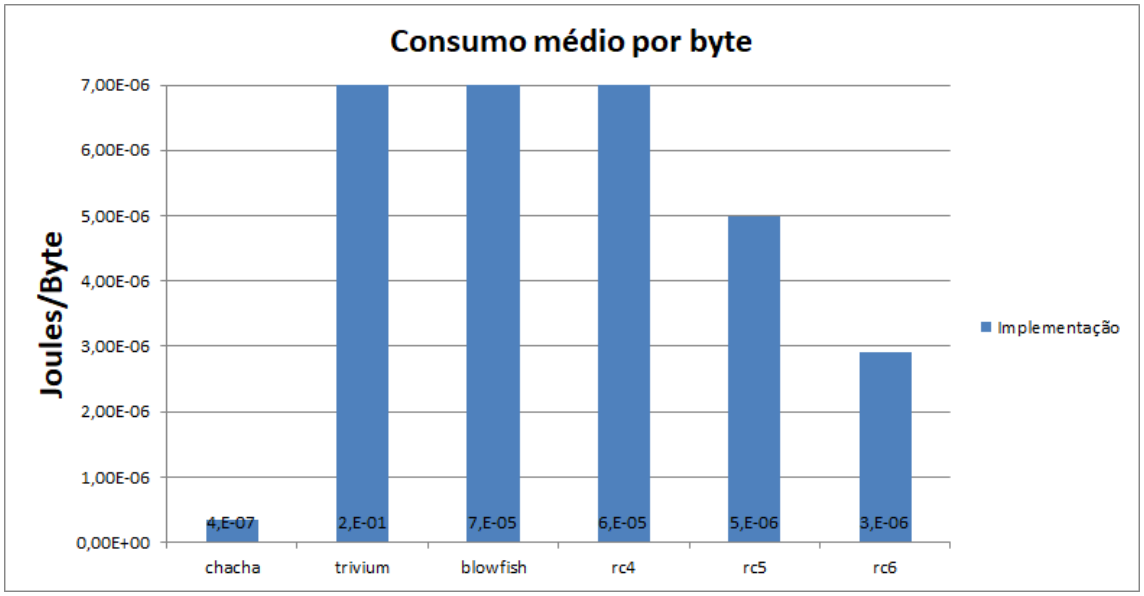


Figura 3.19: Tempo Médio Encriptação Xperia em segundos

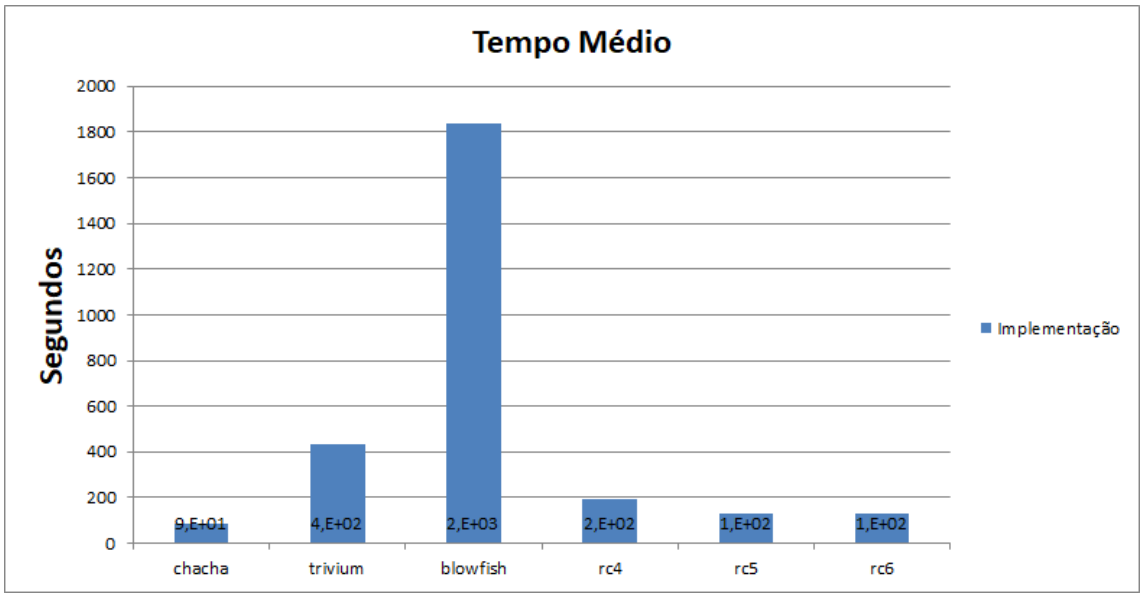
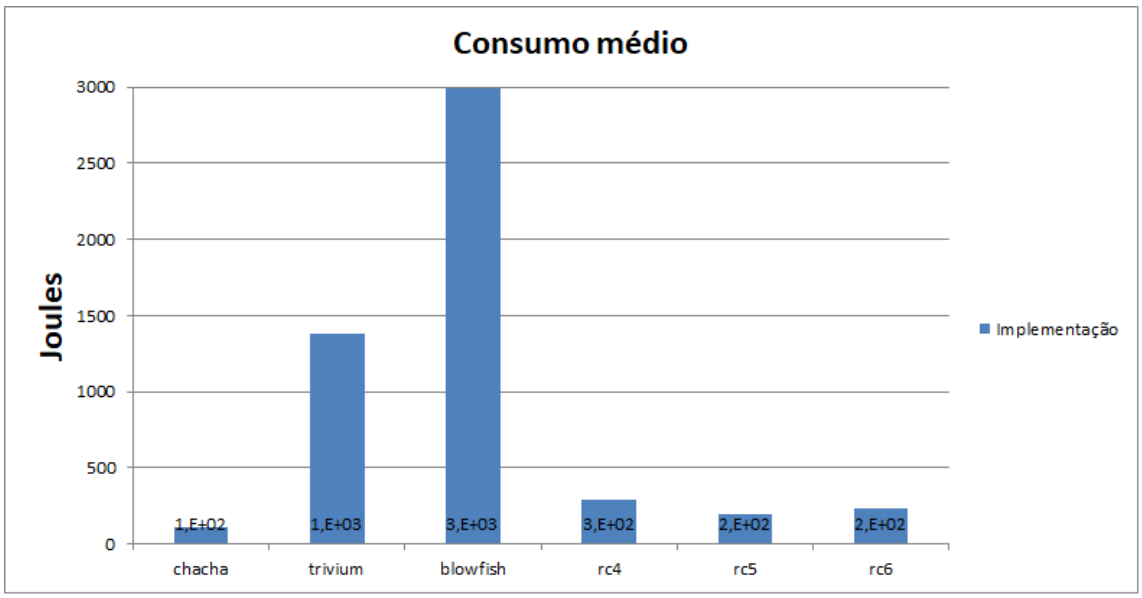


Figura 3.20: Consumo Médio Encriptação Xperia Joule



Devemos também diferenciar o consumo de processamento do consumo do LCD, para que dessa forma, tenhamos resultados sobre o consumo do dispositivo. Para tanto foram realizadas 10 novas medições e salvos seus consumos de LCD e CPU, fazendo a utilização do powertutor. Os valores de consumo são dados em Joule.

Tabela 3.3: Consumo LCD - CPU em Joules

	Blowfish	RC4	RC5	RC6	Chacha	Trivium
lcd	1200	149,5	78,9	77,5	54,2	331,3
cpu	2000	197,2	117,2	127,8	129,1	440,8
lcd	1300	164	98,8	97,4	75,5	335,7
cpu	2000	194,7	127,3	128	126,7	447
lcd	1300	157,3	85,8	90,6	78,2	281,3
cpu	2000	198,6	126,2	128,8	128	446,5
lcd	1300	131	89,2	89,9	58,3	290,9
cpu	2000	195,7	128,7	126,8	129,2	448
lcd	1300	124,2	86,5	94	81,6	449,7
cpu	2000	194,7	127,7	128,1	125,5	299,1
lcd	1300	120,7	106,3	124,2	65,2	408,2
cpu	2000	185,6	128,2	129,4	79,9	443,2
lcd	1300	131	129,7	102,9	81,6	302,2
cpu	2000	198	129	123,8	85	450,7
lcd	1300	135,8	145,4	105,2	68,6	305,3
cpu	2000	196,2	130,1	131	85,7	444,8
lcd	1200	154,4	90,6	92,6	88,5	306,8
cpu	2000	196	128,9	129,8	85,9	448,4
lcd	1300	133,8	90,6	109,1	74,1	303,9
cpu	2000	196,3	129,3	128,6	85,6	449,6

Capítulo 4

Conclusões

4.1 Resultados Obtidos

Como previsto nas intenções, esta dissertação avaliou em diferentes arquiteturas, o consumo de processamento, e o consumo energético de diversos algoritmos de criptografia. Para garantir uma comparação com o estado da arte e validar os resultados obtidos nas mudanças propostas em cada um dos algoritmos escolhidos foi utilizada a biblioteca Cripto++. Embora os algoritmos propostos resultem em perdas de desempenho na inicialização, chegando a ser 10X piores os ganhos de desempenho no processo de encriptação podem ser até 20x melhores quando comparados com seus correspondentes utilizando a biblioteca Cripto++ portanto, esses ganhos de desempenho no processo de encriptação podem refletir melhorias globais no algoritmo porque para a criptografar de uma mensagem grande o processo de encriptação é repetido diversas vezes. As perdas de desempenho do processo de inicialização dos algoritmos propostos em relação aos que fazem uso da biblioteca devem decorrer do fato da biblioteca utilizar para a alocação de objetos, o fast aloc. Tais algoritmos não fizeram parte das implementações propostas, dado que, conforme foi descrito na introdução deste trabalho, os códigos devem ser escritos visando a portabilidade e utilizando somente as bibliotecas padrão da linguagem C++. A aplicação ideal seria aquela que fosse capaz de unir os processos de inicialização utilizados pela biblioteca, que possuem melhor desempenho do que àqueles propostos pela dissertação, com as mudanças no processo de encriptação propostas nessa dissertação, obtendo assim melhores desempenhos na inicialização e na encriptação. É também explicitado nesta dissertação a diferença de desempenho nos processos de inicialização e nos processos de encriptação de acordo com a variação do tamanho da chave do algoritmo. A exibição de tais variações torna-se importante para a avaliação dos impactos no desempenho, uma vez que se torna possível avaliar as perdas de desempenho de um algoritmo em decorrência do aumento do tamanho de uma chave

e analisar seu consequentemente aumento de segurança. Resumo dos resultados obtidos De acordo com os resultados obtidos pudemos perceber que a utilização do algoritmo Chacha possui o melhor desempenho em software para aplicações do dia a dia, possuindo boa performance em arquiteturas AMD, Intel e baixo consumo em dispositivos móveis.

4.2 Implicações Práticas

Na introdução foram apresentadas estimativas de uma grande quantidade de dispositivos móveis conectados à rede, bem como tecnologias emergentes como a *IoT* e a *ICN*. Nesta seção discorreremos sobre as implicações práticas dos resultados obtidos e sobre a aplicabilidade dos algoritmos em um cenário de larga escala como os propostos por estas duas tecnologias emergentes. Os resultados obtidos pelos algoritmos propostos revelam um ganho de desempenho e uma consequente redução de consumo em relação aos algoritmos que utilizam a biblioteca. Contudo é importante mostrar um consumo estimado em uma comparação com um objeto do dia a dia, como uma lâmpada. Dessa forma, de acordo com os resultados obtidos, um algoritmo como o RC6 consumiria $3 * 10^{-6}$ J por byte utilizado, ou seja, uma pessoa normal consumindo aproximadamente 10^{10} bytes gastaria $3 * 10^{4} J$, fazendo uso deste algoritmo. Esse consumo é aproximadamente o equivalente a duas lâmpadas fluorescentes de 15w ligadas por 17 minutos. Os resultados obtidos a partir dos algoritmos propostos abrangem qualquer tipo de tecnologia que faça uso de algoritmos de criptografia, contudo em um cenário *ICN*, onde a segurança é no conteúdo, os ganhos de desempenho possuem um alto impacto no consumo total, visto que tais algoritmos serão executados com uma grande frequência. Apesar dessa dissertação apresentar modificações em algoritmos de criptografia e gerar resultados melhores que os encontrados através da utilização da biblioteca Cripto++ os valores encontrados ainda não são os ideais para uma aplicação prática em um cenário amplo como o *ICN-IoT*.

4.3 Considerações Finais

Os resultados obtidos trazem à tona um fator importante muitas vezes não abordado por muitos autores ao decorrer do cenário *ICN-IoT*: O consumo energético. Conforme exibido, os algoritmos propostos nesta dissertação possuem um ganho de desempenho em relação aos da biblioteca cripto++, contudo um número maior de pesquisas sobre métodos de criptografia de baixo consumo energético em dispositivos de hardware limitados, como é o caso de dispositivos móveis fazem-se cada vez mais necessárias.

Referências Bibliográficas

[1] SOUZA, C. C. L. *Um Estudo sobre Criptografia*. Master degree dissertation, Universidade Estadual Paulista Júlio de Mesquita Filho, Avenida 24 A,1515 13506-900 Rio Claro - SP, Brasil, 2013.

[2] STATISTA. "IoT: number of connected devices worldwide 2012-2025". 2018. Disponível em: <https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/>.

[3] SWAN, M. "Sensor Mania! The Internet of Things, Wearable Computing, Objective Metrics, and the Quantified Self 2.0", *Journal of Sensor and Actuator Networks*, v. 1, n. 3, pp. 217–253, dez. 2012.

[4] ANASTASIADES, C., BRAUN, T., SIRIS, V. A. "Information-Centric Networking in Mobile and Opportunistic Networks". In: *Wireless Networking for Moving Objects: Protocols, Architectures, Tools, Services and Applications*, 1 ed., pp. 14–30, Berlin, Heidelberg, Springer International Publishing, 2014.

[5] AMADEO, M., CAMPOLO, C., QUEVEDO, J., et al. "Information-Centric Networking for the Internet of Things: Challenges and Opportunities", *IEEE Network*, v. 30, n. 2, pp. 92–100, mar. 2016.

[6] AHLGREN, B., DANNEWITZ, C., IMBRENDA, C., et al. "A Survey of Information-Centric Networking", *IEEE Communications Magazine*, v. 50, n. 7, pp. 26–36, jul. 2012.

[7] STALLINGS, W. *Cryptography and Network Security principles and practices*. 1 ed. Upper Saddle River, NJ 07458, EUA, Pearson Prentice Hall, 2005.

[8] SCHNEIER, B. *Applied Cryptography*. 2 ed. Minneapolis, MN 55419, EUA, John Wiley and Sons, Inc, 1996.

[9] SINGHAL, N., J.P.S.RAINA. "Comparative Analysis of AES and RC4 Algorithms for Better Utilization", *International Journal of Computer Trends and Technology*, v. 2, n. 6, pp. 177–181, jul. 2011.

[10] JORSTAD, N. D. "CRYPTOGRAPHIC ALGORITHM METRICS". In: *20th National Information Systems Security Conference*, pp. 1–38, Baltimore, Maryland, EUA, jan. 1997. OASD-PA.

[11] HIRANI, S. *Energy Consumption of Encryption Schemes in Wireless Devices*. Master degree dissertation, University of Pittsburgh, 135 North Bellefield Avenue, Pittsburgh, PA 15260, EUA, 2003.

[12] SALAMA, D., KADER, H. A., HADHOUD, M. "Encryption and Power Consumption in Wireless LANs", *International Journal of Computer Theory and Engineering*, v. 1, n. 4, pp. 334–342, out. 2011.

[13] S, R. L., SINGH, K. J. "Efficient Technique for Secure Transmission of Real-Time Video over Internet", *World Applied Sciences Journal*, v. 33, n. 7, pp. 1102–1108, 06 2015.

[14] S., M. H., REDDY, A. R. "Performance Analysis of AES and MARS Encryption Algorithms", *International Journal of Computer Science Issues*, v. 8, n. 4, pp. 363–368, jul. 2011.

[15] BIHAM, E., SHAMIR, A. "Power Analysis of the Key Scheduling of the AES Candidates". In: *Proceedings of the second AES Candidate Conference*, pp. 115–121, Rehovot 76100, Israel, mar. 1999.

[16] KNUDSEN, L. R., RIJMEN, V. "Weaknesses in LOKI97", *URL: http://www.ii.uib.no/larsr/papers/loki97.pdf*, v. 1, n. 1, pp. 168–174, jun. 1999.

[17] BIHAM, E., BIRYUKOV, A., FERGUSON, N., et al. "Cryptanalysis of Magenta". In: *Draft distributed during the First Advanced Encryption Standard Candidate Conference*, pp. 1–3, Rehovot 76100, Israel, ago. 1998.

[18] IDRUS, S. Z. S., ALJUNID, S. A., ASI, S. M., et al. "Performance Analysis of Encryption Algorithms Text Length Size on Web Browsers", *International Journal of Computer Science and Network Security*, v. 8, n. 1, pp. 20–25, jan. 2008.

[19] BURKE, J., MCDONALD, J., AUSTIN, T. "Architectural Support for Fast Symmetric-Key Cryptography", *ACM SIGARCH Computer Architecture News*, v. 28, n. 5, pp. 178–189, nov. 2000.

[20] BERNSTEIN, D. J. "ChaCha, a variant of Salsa20". In: *Workshop Record of SASC*, v. 8, *State of the Art of Stream Ciphers Workshop*, SASC, pp. 3–5, 2008.

[21] BERNSTEIN, D. J. "Salsa20 security", *URL: http://cr.yp.to/snuffle/security.pdf*, v. 1, n. 1, pp. 1–9, 06 2005.

[22] BURWICK, C., COPPERSMITH, D., D'AVIGNON, E., et al. "The MARS Encryption Algorithm", *URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.35.5887&rep=rep1&type=pdf*, v. 1, n. 1, pp. 1–12, ago. 1999.

[23] SCHNEIER, B., KELSEY, J., WHITING, D., et al. "The Twofish Team's Final Comments on AES Selection", *AES round*, v. 2, n. 1, pp. 1–13, maio 2000.

[24] POTLAPALLY, N. R., RAVI, S., RAGHUNATHAN, A., et al. "Analyzing the energy consumption of security protocols". In: *Low Power Electronics and Design, 2003. ISLPED '03. Proceedings of the 2003 International Symposium on*, pp. 30–35, Seoul, South Korea, South Korea, ago. 2003.

[25] BAILEY, D. H. "The BBP Algorithm for Pi", *URL: http://crd-legacy.lbl.gov/ dhbailey/dhbpapers/bbp-alg.pdf*, v. 1, n. 1, pp. 1–8, jul. 2006.

[26] SHANNON, C. E. "Communication Theory of Secrecy Systems", *Bell Labs Technical Journal*, v. 28, n. 4, pp. 656–715, out. 1949.

[27] MALETSKY, K. *RSA vs ECC Comparison for Embedded Systems*. Technical Report TR-8951A, Atmel Corporation, San Jose, CA 95110 USA, 2015.

[28] B. PADMAVATHI, S. R. K. "A Survey on Performance Analysis of DES, AES and RSA Algorithm along with LSB Substitution Technique", *International Journal of Science and Research*, v. 2, n. 4, pp. 170–174, abr. 2013.

[29] BERNSTEIN, D. J., CHOU, T., SCHWABE, P. "McBits:fast constant-time code-based cryptography". In: *International Workshop on Cryptographic Hardware and Embedded Systems*, pp. 250–272, Berlin, Heidelberg, mar. 2013.

[30] BOGDANOV, A., KHOVRATOVICH, D., RECHBERGER, C. "Biclique Cryptanalysis of the Full AES". In: *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 344–371, K.U. Leuven, Belgium and Microsoft Research Redmond, USA and ENS Paris and Chaire France Telecom, France, dez. 2011.

[31] SARKAR, P. G., FITZGERALD, S. *ATTACKS ON SSL A COMPREHENSIVE STUDY OF BEAST, CRIME, TIME, BREACH, LUCKY 13 & RC4 BIASES*. In: Report, iSEC Partners, San Francisco, , CA 94105, EUA, 2013.

[32] KALISKI JR., B. S., YIN, Y. L. *On security os the RC5 Encryption Algorithm.* Technical Report TR - 602, RSA Laboratories, 1998.

[33] RIVEST, R. L., CONTINI, S., ROBSHAW, M., et al. *The Security of the RC6 TM Block Cipher.* In: Report, RSA Laboratories and M.I.T. Laboratory for Computer Science, Bedford, MA 01730, USA and Cambridge, MA 02139, EUA, 1998.

[34] TERADA, R., UEDA, E. T. "A new version of the RC6 algorithm, stronger against X^2 cryptanalysis", *Conferences in Research and Practice in Information Technology*, v. 98, n. 1, pp. 47–52, jan. 2009.

[35] SCHNEIER, B. "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)". In: *Fast Software Encryption, Cambridge Security Workshop, Cambridge, UK, December 9-11, 1993, Proceedings*, v. 809, *Lecture Notes in Computer Science*, Springer, pp. 191–204, 1993.

[36] AUMASSON, J. P., FISCHER, S., KHAZAEI, S., et al. "New Features of Latin Dances:Analysis of Salsa, ChaCha, and Rumba". In: *International Workshop on Fast Software Encryption*, v. 5085, *Lecture Notes in Computer Science*, Springer, pp. 470–488, 2008.

[37] FOUQUE, P. A., VANNET, T. "Improving Key Recovery to 784 and 799 rounds of Trivium using Optimized Cube Attacks". In: *International Workshop on Fast Software Encryption*, pp. 502–517, Berlin, Heidelberg, mar. 2013. Springer. ISBN: 978-3-540-44499-2.

[38] TAQIEDDIN, E., ABU-RJEI, O., MHAIDAT, K., et al. "Efficient FPGA Implementation of the RC4 Stream Cipher using Block RAM and Pipelining", *Procedia Computer Science*, v. 63, n. 6, pp. 8–15, set. 2015.

[39] ELKEELANY, O. "Performance Comparisons, Design, and Implementation of RC5 Symmetric Encryption Core using Reconfigurable Hardware", *Journal of Computers*, v. 3, n. 3, pp. 48–55, mar. 2008.

[40] DANDALIS, A., PRASANNA, V. K., ROLIM, J. D. P. "A comparative study of performance of AES final candidates using FPGAs". In: *Cryptographic Hardware and Embedded Systems — CHES 2000*, pp. 125–140, Berlin, Heidelberg, ago. 2000. Springer Berlin Heidelberg. ISBN: 978-3-540-44499-2.

[41] ROGAWSKI, M. "Hardware evaluation of eSTREAM Candidates:Grain, Lex, Mickey128, Salsa20 and Trivium". In: *Workshop Record of SASC*, v. 25, *State of the Art of Stream Ciphers Workshop*, SASC, pp. 2007–2017.

[42] RIVEST, R. L. "The RC5 Encryption Algorithm". In: *International Workshop on Fast Software Encryption*, pp. 86–96, 545 Technology Square, Cambridge, Mass. 02139, EUA, dez. 1994.

[43] RIVEST, R. L., ROBSHAW, M., SIDNEY, R., et al. "The RC6 Block Cipher". In: *First Advanced Encryption Standard (AES) Conference*, pp. 16–37, M.I.T. Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA and RSA Laboratories, 2955 Campus Drive, Suite 400, San Mateo, CA 94403, USA, dez. 1998.

[44] PRASITHSANGAREE, P., KRISHNAMURTHY, P. "Analysis of energy consumption of RC4 and AES algorithms in wireless LANs". In: *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, pp. 1445–1449 vol.3, San Francisco, CA, USA, dez. 2003. IEEE.

[45] GROSSSCHADL, J., TILLICH, S., RECHBERGER, C., et al. "Energy Evaluation of Software Implementations of Block Ciphers under Memory Constraints". In: *2007 Design, Automation Test in Europe Conference Exhibition*, pp. 1–6, Nice, France, abr. 2007. IEEE.

[46] SEO, H., JEONG, I., LEE, J., et al. "Compact Implementations of ARX-Based Block Ciphers on IoT Processors", *ACM Trans. Embed. Comput. Syst.*, v. 17, n. 3, pp. 60:1–60:16, jun. 2018.

[47] CHODOWIEC, P., GAJ, K. *Implementation of the Twofish Cipher using FPGA Devices*. In: Report, 4400 University Dr, Fairfax, VA 22030, EUA, Washington, D.C., 1999.

[48] SCHNEIER, B., KELSEY, J., WHITING, D., et al. "Twofish: A 128-Bit Block Cipher", *NIST AES Proposal*, v. 15, n. 1, pp. 23–91, jun. 1998.

[49] BERNSTEIN, D. J. "The Salsa20 Family of Stream Ciphers". In: Robshaw, M., Billet, O. (Eds.), *New Stream Cipher Designs: The eSTREAM Finalists*, 1 ed., pp. 84–97, Berlin, Heidelberg, Springer Berlin Heidelberg, 2008.

[50] BERNSTEIN, D. J., SCHWABE, P. "New AES software speed records". In: *Progress in Cryptology - INDOCRYPT 2008*, pp. 322–336, Berlin, Heidelberg, dez. 2008. Springer.

[51] ZHANG, L., TIWANA, B., QIAN, Z., et al. "Accurate Online Power Estimation and Automatic Battery Behavior Based Power Model Generation for Smartphones". In: *Proceedings of the Eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pp. 105–114, New York, NY, USA, out. 2010. ACM.