Crypgas: Biblioteca de Criptografia em C para Sistemas Bare-Metal

Gabriel Norberto Oka Lôbo - 13680205
Gastón Enrique Christianini Buela - 13682792
Marcelo Takayama Russo - 13680164

Introdução

No projeto da disciplina de Laboratório de Processadores, o grupo decidiu desenvolver uma biblioteca de criptografia em linguagem C para sistemas bare-metal. A escolha surgiu a partir dos conteúdos trabalhados no laboratório e do interesse em aprofundar o uso da Raspberry Pi 2. Durante as atividades, estudamos conceitos como montadores, GCC e o fluxo de compilação para sistemas embarcados, mas a implementação prática acabou se limitando a um algoritmo simples para acender um LED a partir de um evento de entrada. Querendo ir além, optamos por criar uma biblioteca criptográfica que colocasse em prática de forma mais ampla os conceitos vistos em aula.

Além do desenvolvimento da biblioteca, realizamos um estudo de benchmark para avaliar o desempenho dos algoritmos implementados e do processador ARM Cortex-A7 presente na Raspberry Pi 2. Foram utilizados diferentes padrões de entrada para medir a eficiência de cada algoritmo e entender como o hardware se comporta em diferentes cenários de processamento. Esse estudo permitiu não apenas analisar a performance da plataforma, mas também comparar o custo computacional entre algoritmos simples, como a cifra de César, e outros mais complexos, como AES e Blowfish.

Cifra de César

A **Cifra de César** recebeu esse nome por ter sido utilizada pelo imperador romano **Júlio César** no século I a.C. para proteger mensagens militares. Segundo relatos históricos, César empregava um deslocamento fixo — geralmente de três posições no alfabeto latino — para cifrar ordens enviadas a seus generais, garantindo que, caso fossem interceptadas, não pudessem ser facilmente compreendidas. Na época, a simplicidade do método era suficiente para oferecer um nível básico de sigilo, já que poucos tinham conhecimento formal de técnicas de criptografia. Com o

avanço da escrita e da criptoanálise, a Cifra de César perdeu seu valor estratégico, mas permaneceu como um marco inicial no desenvolvimento histórico da criptografia.

Optamos por implementar a Cifra de César como um dos algoritmos por ser praticamente uma “porta de entrada” para os algoritmos que depois surgiram e são que também são mais complexos. Por ser um algoritmo simples, sua implementação foi mais fácil, permitindo realizar alguns testes de outras partes do projeto, como comunicação serial, benchmarking etc.

A Cifra de César é um método de criptografia simétrica por substituição, considerado um dos mais antigos da história. Seu funcionamento consiste em deslocar cada letra do texto original um número fixo de posições no alfabeto, definido por uma chave. Esse deslocamento é realizado de forma cíclica: ao atingir o final do alfabeto, a contagem recomeça do início. O processo de descriptografia é o inverso, deslocando as letras o mesmo número de posições na direção oposta.

Por utilizar sempre o mesmo deslocamento para todas as letras, trata-se de uma cifra monoalfabética. O espaço de chaves é reduzido — apenas 25 valores possíveis no alfabeto latino —, o que a torna extremamente vulnerável a ataques de força bruta. Além disso, a frequência das letras é preservada, permitindo que seja quebrada por análise de frequência. Apesar de não oferecer segurança prática nos padrões atuais, a Cifra de César ainda é amplamente utilizada para fins didáticos, servindo como exemplo introdutório de conceitos básicos de criptografia, como substituição, modularidade e uso de chave simétrica.

AES

O AES, ou Advanced Encryption Standard, é um algoritmo de criptografia simétrica estabelecido como padrão pelo governo dos Estados Unidos em 2001, após um processo público de seleção conduzido pelo NIST (National Institute of Standards and Technology). Ele foi desenvolvido pelo belga Vincent Rijmen e Joan Daemen, sob o nome original Rijndael, e escolhido para substituir o DES, que já apresentava vulnerabilidades frente ao aumento do poder computacional. Desde então, o AES

tornou-se o padrão mundial para criptografia simétrica, sendo amplamente utilizado em comunicações seguras, armazenamento de dados e protocolos como TLS, VPNs e criptografia de disco.

Optamos por implementar o AES no projeto por ser o algoritmo de criptografia simétrica mais utilizado atualmente, oferecendo segurança robusta e desempenho eficiente em diferentes plataformas. Sua implementação envolve lidar com conceitos como substituições não lineares, permutações, operações sobre campos finitos ($GF(2^8)$) e múltiplas rodadas de transformação, proporcionando um aprendizado mais profundo sobre técnicas de criptografia moderna. Além disso, sua padronização e relevância prática permitem testar o sistema com um algoritmo amplamente reconhecido e confiável.

O AES é uma cifra de bloco simétrica que opera com blocos de 128 bits e permite chaves de 128, 192 ou 256 bits, o que define a quantidade de rodadas de processamento: 10, 12 ou 14, respectivamente. Diferente da estrutura de Feistel usada no Blowfish, o AES utiliza uma arquitetura conhecida como rede de substituição-permutação (SPN), na qual cada rodada aplica operações como SubBytes (substituições não lineares), ShiftRows (deslocamentos de linhas), MixColumns (mistura linear de colunas) e AddRoundKey (adição de subchaves derivadas da chave original). O processo de expansão de chave gera subchaves únicas para cada rodada, aumentando a complexidade e a resistência contra ataques criptoanalíticos.

Com um espaço de chaves extremamente grande e resistência comprovada contra ataques de força bruta e criptoanálise diferencial e linear, o AES é considerado seguro para uso em aplicações críticas. Sua eficiência em hardware e software, combinada à ampla aceitação e ausência de patentes, garantiu sua adoção global como o principal padrão de criptografia simétrica. Além de sua importância prática, o AES é um excelente exemplo de algoritmo moderno, ilustrando conceitos de redes SPN, expansão de chave e segurança baseada em operações sobre campos finitos.

Blowfish

O Blowfish é um algoritmo de criptografia simétrica desenvolvido por Bruce Schneier em 1993, com o objetivo de oferecer uma alternativa pública, rápida e livre de patentes aos padrões proprietários da época, como o DES. Criado em um período em que havia grande demanda por algoritmos eficientes e seguros para aplicações comerciais, o Blowfish rapidamente ganhou popularidade devido à sua alta velocidade e flexibilidade. Sua concepção buscava equilibrar segurança, desempenho e liberdade de uso, tornando-o uma das escolhas preferidas em softwares e sistemas até a adoção mais ampla do AES.

Optamos por implementar o Blowfish no projeto por se tratar de um algoritmo historicamente importante e muito mais robusto que cifras clássicas como a de César. Sua implementação proporcionou a oportunidade de explorar aspectos mais complexos da criptografia moderna. Além disso, seu desempenho relativamente alto permitiu testar de forma eficiente recursos como benchmarking e comunicação entre dispositivos.

O Blowfish é um algoritmo de cifra de bloco simétrica, operando sobre blocos de 64 bits e utilizando chaves variáveis de até 448 bits. Seu funcionamento é baseado na estrutura de rede de Feistel, composta por 16 rodadas de processamento que envolvem substituições não lineares (S-boxes) e permutações dependentes da chave. Antes da criptografia propriamente dita, o algoritmo executa uma fase intensiva de expansão de chave, na qual os dados da chave são usados para inicializar e embaralhar grandes tabelas internas, aumentando a resistência a ataques de criptoanálise.

Diferente de cifras simples, o Blowfish apresenta um espaço de chaves extremamente grande, tornando ataques de força bruta impraticáveis com recursos convencionais. Sua segurança é reforçada pelo uso de operações bit a bit e funções não lineares, o que dificulta ataques baseados na análise estatística dos dados cifrados. No entanto, por operar em blocos de 64 bits, versões mais modernas de protocolos de segurança têm substituído o Blowfish por algoritmos com blocos maiores, como o

AES, para prevenir ataques de repetição em volumes muito grandes de dados. Ainda assim, o Blowfish permanece como um marco no avanço da criptografia simétrica, servindo tanto para aplicações práticas quanto para fins didáticos, expansão de chave e segurança baseada em tamanho de chave elevado.

Implementação na Raspberry Pi 2

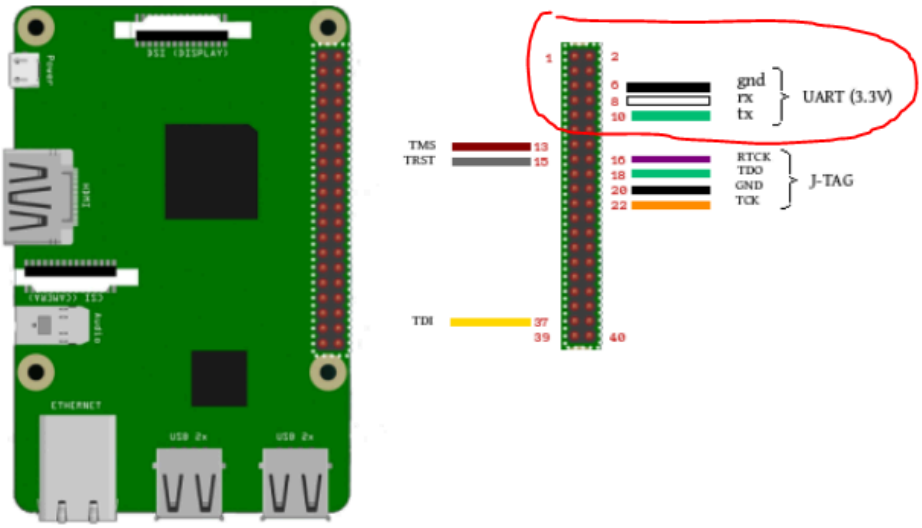
Comunicação Serial

Cabo UART

A ideia do projeto é utilizar o processador da Raspberry para executar os algoritmos de criptografia/descriptografia. Para isso, é necessário enviar os dados que queremos que sejam criptografados e é necessário recebê-los também. Por isso, a comunicação entre a Raspberry Pi 2 (rodando em bare-metal) e o computador foi feita por meio de **UART** (*Universal Asynchronous Receiver/Transmitter*) utilizando um cabo conversor USB–Serial, que no PC é reconhecido como a porta **/dev/ttyUSB0**.

O UART é um protocolo de comunicação **assíncrono** ponto a ponto, no qual os dados são transmitidos byte a byte sem a necessidade de um sinal de clock dedicado. Em vez disso, transmissor e receptor combinam previamente parâmetros como **taxa de transmissão** (*baud rate*), número de bits de dados, uso ou não de bit de paridade e quantidade de bits de parada. No nosso caso, usamos configuração padrão de 8 bits de dados, sem paridade e 1 bit de parada (8N1).

Por meio do guia presente no fórum, realizamos a conexão do cabo à raspberry.



UART.C

O arquivo uart.c foi necessário para implementar toda a lógica de comunicação serial do projeto diretamente sobre o hardware da Raspberry Pi 2, sem uso de sistema operacional.

Nele estão as funções que inicializam e controlam o periférico UART0 por meio de registradores mapeados em memória, usando o endereço base 0x3F201000. A função uart_init() configura o módulo, definindo parâmetros como taxa de transmissão (baud rate), formato de dados e habilitação do transmissor e receptor. Já uart_send() e uart_get() permitem enviar e receber um caractere, enquanto uart_puts() envia strings inteiras e uart_put_uint() converte números inteiros para texto antes de transmiti-los.

Com esse arquivo, foi possível padronizar e centralizar o controle da comunicação serial, permitindo que outras partes do código apenas chamassem funções simples para enviar ou receber dados, sem precisar lidar diretamente com os detalhes de configuração e operação do hardware.

```
void uart_init(void) {
    UART0_CR = 0x00000000;

    UART0_ICR = 0x7FF;

    UART0_IBRD = 312;
```

```
        UART0_FBRD = 32;

        UART0_LCRH = (3 << 5);
        UART0_CR = 0x301;
    }
```

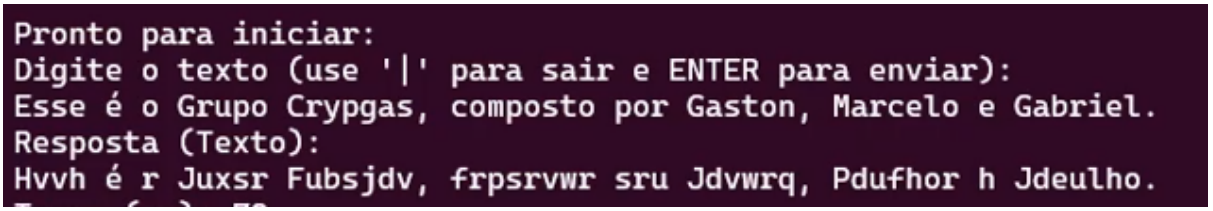
Cifra de César

Com tudo preparado, partiu-se para a implementação dos algoritmos de criptografia. Começamos pela Cifra de César, pela sua simplicidade.

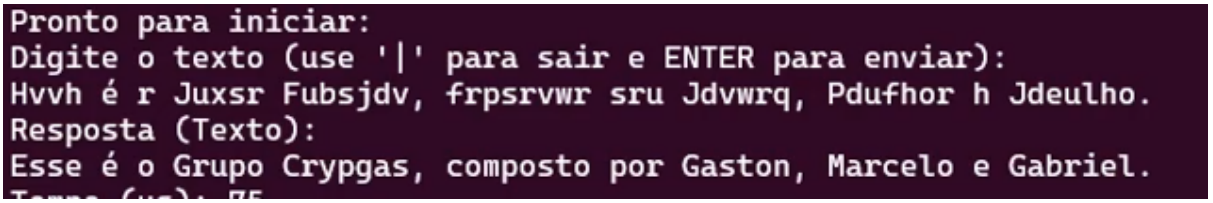
```
char cifra_cesar(char c, int chave) {
    chave %= 26;
    if (chave < 0) chave += 26;

    if (c >= 'A' && c <= 'Z') {
        return ((c - 'A' + chave) % 26) + 'A';
    } else if (c >= 'a' && c <= 'z') {
        return ((c - 'a' + chave) % 26) + 'a';
    } else if (c >= '0' && c <= '9') {
        // Para números, usa módulo 10
        int chaveNum = (chave % 10 + 10) % 10;
        return ((c - '0' + chaveNum) % 10) + '0';
    }
    return c;
}
```

Como explicado anteriormente, basta apenas fazer um deslocamento de caracteres a partir da chave definida (utilizamos 3 como default). Isso faz com que para a palavra “Bruno”, tenhamos “Evxqr”.

Exemplo de Criptografia com chave = 3



Exemplo de Descriptografia com chave = 3



AES

O AES implementado no projeto segue o padrão AES-128, operando com blocos de 16 bytes e chave de 128 bits, executando 10 rodadas de processamento. O algoritmo trabalha sobre uma matriz lógica 4×4 bytes chamada *state*, onde os dados do bloco são carregados e transformados a cada etapa. Antes da cifragem, ocorre a expansão de chave (*Key Expansion*), que gera 176 bytes de subchaves a partir da chave original. Esse processo copia os 16 bytes iniciais da chave e, para cada nova palavra de 4 bytes, aplica rotações de byte (*RotWord*), substituição não linear via tabela S-Box (*SubWord*), e a adição de constantes Rcon, combinando o resultado com a palavra localizada 16 bytes antes no vetor de subchaves.

A cifragem começa com a operação *AddRoundKey*, que realiza um XOR do *state* com a primeira subchave. Em seguida, são executadas nove rodadas contendo, nesta ordem, as operações *SubBytes* (substituição não linear de cada byte pela S-Box), *ShiftRows* (deslocamento circular das linhas do *state*), *MixColumns* (mistura linear das colunas no campo GF(2^8)), e nova aplicação de *AddRoundKey* com a subchave da rodada. A décima e última rodada omite o *MixColumns*, conforme a especificação do padrão.

A decifragem reverte esse processo aplicando as transformações inversas (*InvShiftRows*, *InvSubBytes* e *InvMixColumns*), percorrendo as subchaves na ordem

inversa. Além do modo básico de bloco (ECB), a implementação inclui o modo CBC com *padding* PKCS#7, permitindo processar mensagens de tamanho arbitrário. No modo CBC, cada bloco de texto plano é XORado com o bloco cifrado anterior (ou com o IV no caso do primeiro bloco) antes de passar pela cifragem AES, e na decifragem esse XOR é aplicado após o bloco ser decifrado, removendo-se o *padding* ao final.

Exemplo de Criptografia com chave = "aeskey128bit0001" e iv = 0

Exemplo de Descriptografia com chave = "aeskey128bit0001" e iv = 0

Pode-se notar que a saída está em hexadecimal. Isso foi proposital por nos dar uma possibilidade de analisar a saída criptografada, uma vez que a saída criptografada sem uma conversão para hexa não é interpretável em terminal.

Blowfish

Já no Blowfish, começamos carregando no contexto as constantes públicas (P_init e S_init) e, em seguida, "misturamos" a chave do usuário no P-array: percorremos o P em palavras de 32 bits e fazemos XOR com a chave (repetindo os bytes da chave se necessário). Depois disso, derivamos todas as subchaves cifrando repetidamente o par (L=0, R=0): a cada cifragem, substituímos dois elementos do P-array pelo resultado; ao terminar o P, repetimos o processo para preencher, em pares, todas as entradas das quatro S-boxes. Com P e S prontos, ciframos dados de 64 bits por uma rede de Feistel com 16 rodadas: em cada rodada, fazemos L ^= P[i], calculamos R ^= F(L) (onde F consulta as S-boxes com os quatro bytes de L e combina os resultados com somas e XOR), e trocamos L e R; ao final, desfazemos a troca e aplicamos R ^= P[16] e L ^= P[17] para obter o bloco cifrado. No modo CBC,

antes de cifrar cada bloco de 8 bytes fazemos um XOR com o IV (no primeiro bloco) ou com o bloco cifrado anterior; para mensagens não múltiplas de 8, aplicamos padding PKCS#7 no último bloco. Na decifragem, seguimos o caminho inverso: percorremos o P-array ao contrário na Feistel, desfazemos o encadeamento do CBC com XOR do IV/bloco anterior e removemos o padding no final.

Exemplo de Criptografia com chave = "blowfishkey" e iv = 0

Exemplo de Descriptografia com chave = "blowfishkey" e iv = 0

Benchmark

Para obter uma avaliação completa do desempenho dos algoritmos de criptografia implementados, optamos por utilizar mais de um tipo de benchmark. Essa decisão se baseia no fato de que o padrão dos dados de entrada pode influenciar diretamente o tempo de execução e o throughput alcançado. Vale ressaltar que, para realização do Benchmark, optamos por usar os parâmetros Tempo de Execução e Throughput.

Algoritmos criptográficos podem apresentar comportamentos distintos dependendo do nível de repetição e variação dos dados processados. Por exemplo, entradas extremamente repetitivas tendem a ser processadas de forma mais previsível, enquanto dados mais variados exigem um número maior de operações de substituição, mistura e transformação, o que pode impactar negativamente o desempenho.

Assim, foram definidos diferentes padrões de teste, que variam desde massas uniformes até textos mais próximos de informações reais. Essa abordagem permite observar o comportamento de cada algoritmo em cenários distintos, proporcionando

uma análise mais realista e abrangente do desempenho, além de identificar possíveis pontos fortes e limitações em situações de uso prático.

Tipo 1 – Massa uniforme: todo o buffer é preenchido com o mesmo caractere ('A'). Esse caso representa dados extremamente repetitivos e previsíveis, úteis para avaliar como o algoritmo lida com padrões de baixa entropia.

Tipo 2 – Padrão curto repetitivo: o buffer é preenchido repetindo o padrão fixo "ABC123". Apesar de ainda ser repetitivo, esse padrão é mais variado que o do tipo 1, e ajuda a verificar se pequenas variações no conteúdo influenciam no desempenho.

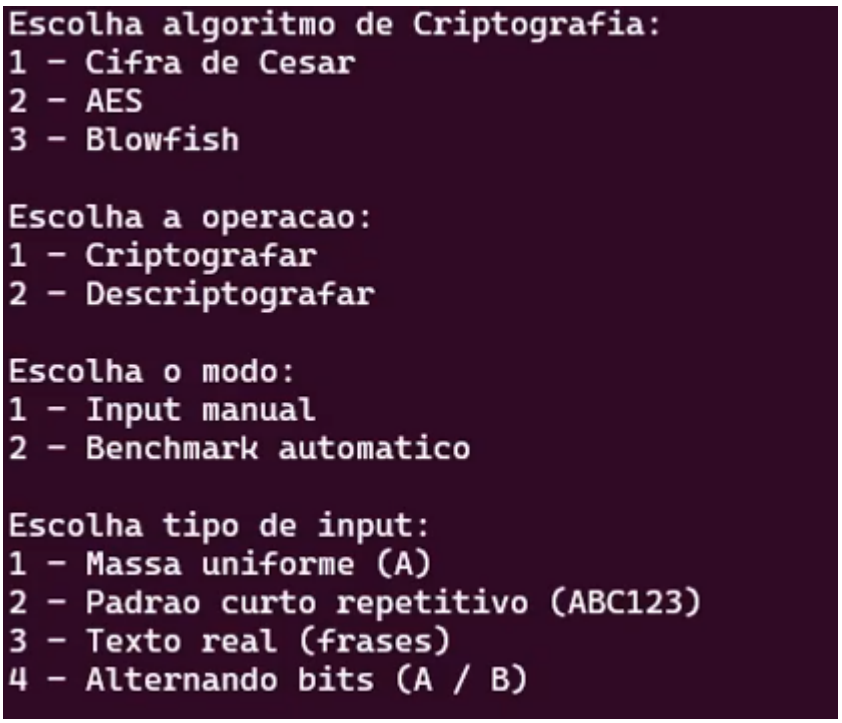
Tipo 3 – Texto real: o buffer é preenchido repetindo a frase "O rápido marrom raposo salta sobre o cachorro preguiçoso. ". Essa entrada simula um texto natural, mais próximo de dados reais, com variação de letras, espaços e pontuação.

Tipo 4 – Alternando bits: o buffer alterna entre 'A' e 'B' a cada caractere. Esse padrão cria uma alternância binária simples, permitindo observar como os algoritmos se comportam com mudanças previsíveis de um byte para outro.

Benchmark na Raspberry Pi 2

Como estamos usando Bare-Metal, foi necessário alocar os registradores da Raspberry para realizar essa tarefa. Por conta disso, criou-se um arquivo **benchmark.c**, que possui funções para realização do benchmark dos algoritmos. O tempo é medido utilizando o **System Timer** do processador, um contador de 64 bits que é incrementado automaticamente a cada microssegundo desde que a Raspberry Pi é ligada. Dessa forma, marca-se o tempo de início e fim da criptografia/descriptografia, tendo o tempo de execução. O throughput também é encontrado usando esse parâmetro, mas relacionado ao número de caracteres criptografados.

Além disso, criamos um menu para escolha do tipo de benchmark diretamente pelo terminal. Ao escolher um benchmark específico, o computador envia automaticamente os inputs para realização das medidas.



Benchmark para Computador pessoal

Foram usados 2 computadores pessoais com as seguintes especificações:

Processador 11th Gen Intel(R) Core(TM) i5-11400H @ 2.70GHz (2.69 GHz)
RAM instalada 8,00 GB (utilizável: 7,73 GB)

Processador 11th Gen Intel(R) Core(TM) i5-11300H @ 3.10GHz (3.11 GHz)
RAM instalada 8,00 GB (utilizável: 7,79 GB)

Para a coleta das medições de tempo de execução e throughput, adotou-se o uso de makefiles como ferramenta de automação do processo de compilação e execução. Essa abordagem permitiu padronizar a execução das diferentes implementações — Cifra de César, AES e Blowfish — garantindo que todas fossem submetidas às mesmas condições de teste, eliminando variações causadas por procedimentos manuais.

Os makefiles foram configurados para compilar os códigos-fonte de cada algoritmo com as mesmas opções de otimização, assegurando que o desempenho medido fosse consequência direta da implementação e não de diferenças no processo de compilação.

Essa metodologia não apenas agilizou a execução repetida dos experimentos, como também reduziu a possibilidade de erros humanos, garantindo maior confiabilidade aos resultados obtidos. Com isso, foi possível comparar de forma mais justa o desempenho relativo das três cifras em termos de tempo de processamento e throughput, fornecendo dados consistentes para a análise apresentada nas seções seguintes.

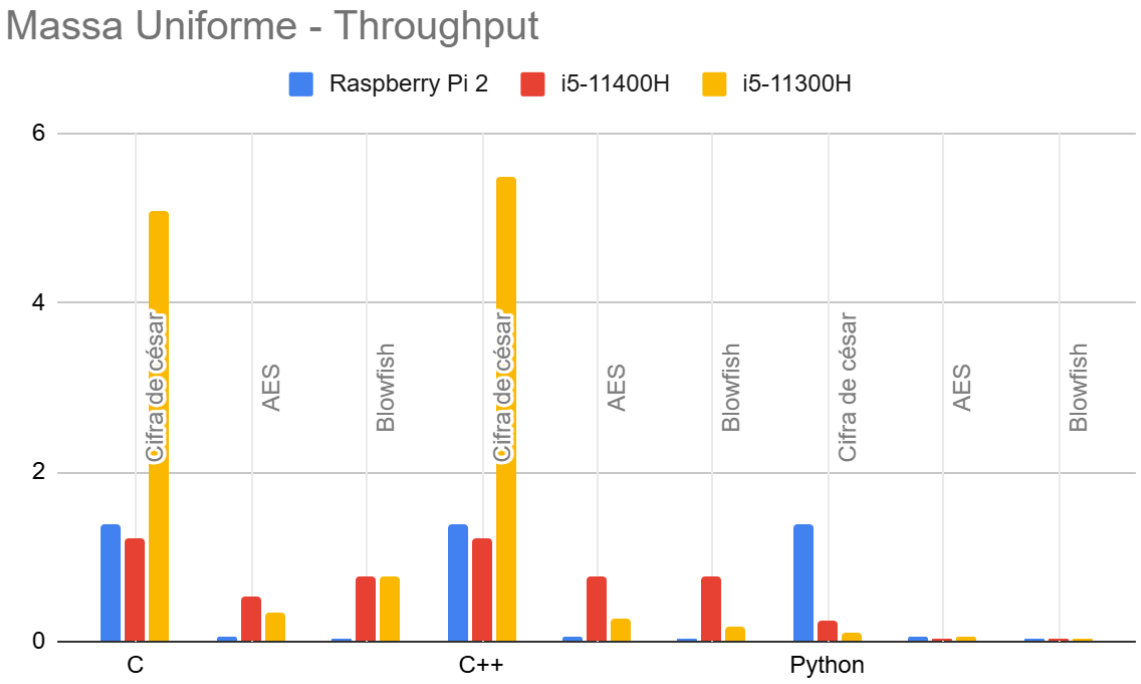
Análise - Massa Uniforme

A análise dos valores médios de tempo de execução obtidos a partir das medições realizadas evidencia diferenças expressivas no desempenho dos algoritmos e linguagens nas três plataformas avaliadas. No Raspberry Pi 2, a Cifra de César apresentou tempos médios significativamente baixos (5,76 ms), enquanto o AES e o Blowfish registraram médias muito mais elevadas (142 ms e 212 ms, respectivamente), refletindo o custo computacional inerente a algoritmos criptográficos mais complexos e as limitações de processamento desse hardware.

Nos processadores Intel i5 de 11ª geração (11400H e 11300H), os tempos médios de execução para AES e Blowfish em C e C++ caíram drasticamente, chegando a cerca de 10 ms em vários casos, evidenciando o impacto positivo de arquiteturas mais modernas e maior capacidade de processamento. A Cifra de César manteve tempos médios bastante reduzidos em todas as plataformas, confirmando seu baixo custo computacional.

Ao comparar linguagens, C e C++ apresentaram tempos médios praticamente equivalentes nas três plataformas, enquanto Python apresentou médias significativamente superiores, especialmente para AES e Blowfish, onde os tempos

chegaram a ser dezenas de vezes maiores em relação às versões compiladas. Essa diferença reforça o efeito da sobrecarga de execução de linguagens interpretadas em tarefas intensivas em processamento.



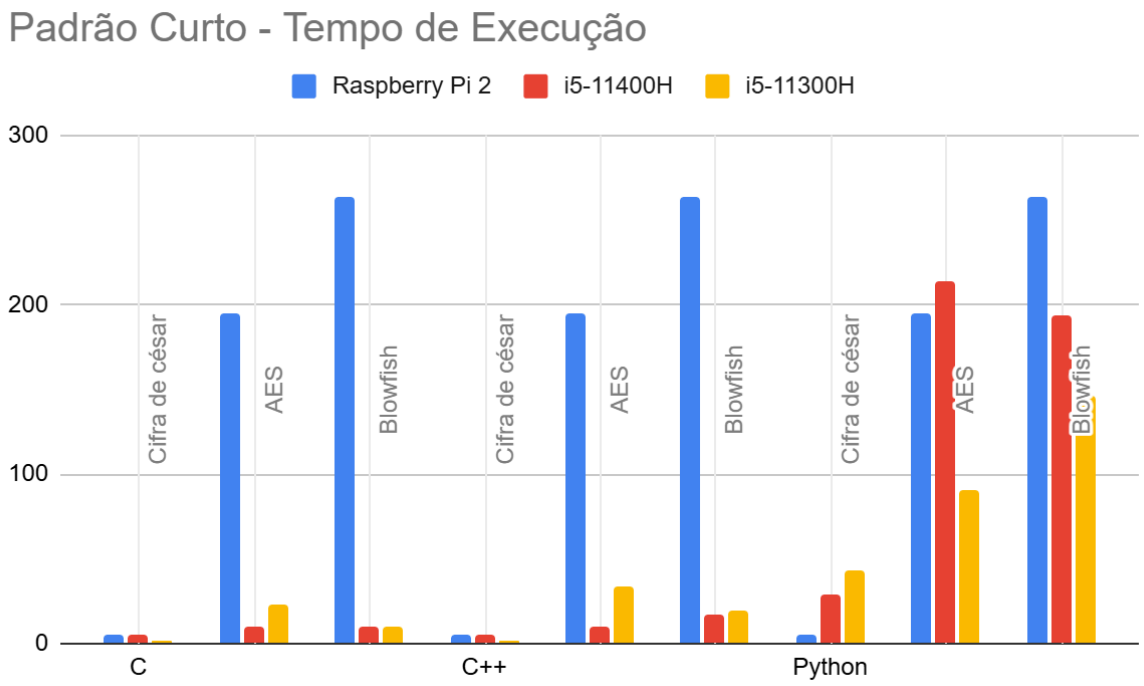
No Raspberry Pi 2, a Cifra de César apresentou os maiores valores médios de throughput (1,39 MB/s), enquanto AES e Blowfish ficaram muito abaixo (0,05 MB/s e 0,038 MB/s), refletindo o custo computacional das cifras modernas em hardware limitado.

Nos processadores i5, especialmente o i5-11300H, o throughput da Cifra de César aumentou consideravelmente (5,08 MB/s em C), demonstrando a influência direta da capacidade de processamento do hardware. AES e Blowfish melhoraram em comparação ao Raspberry Pi, mas ainda apresentaram valores menores que a cifra simples.

Comparando linguagens, Python exibiu throughput muito inferior para todos os algoritmos, destacando a sobrecarga da interpretação em operações intensivas. C e

C++ tiveram desempenho muito similar, reforçando que a linguagem compilada mantém vantagem em eficiência de processamento.

Análise - Padrão Curto

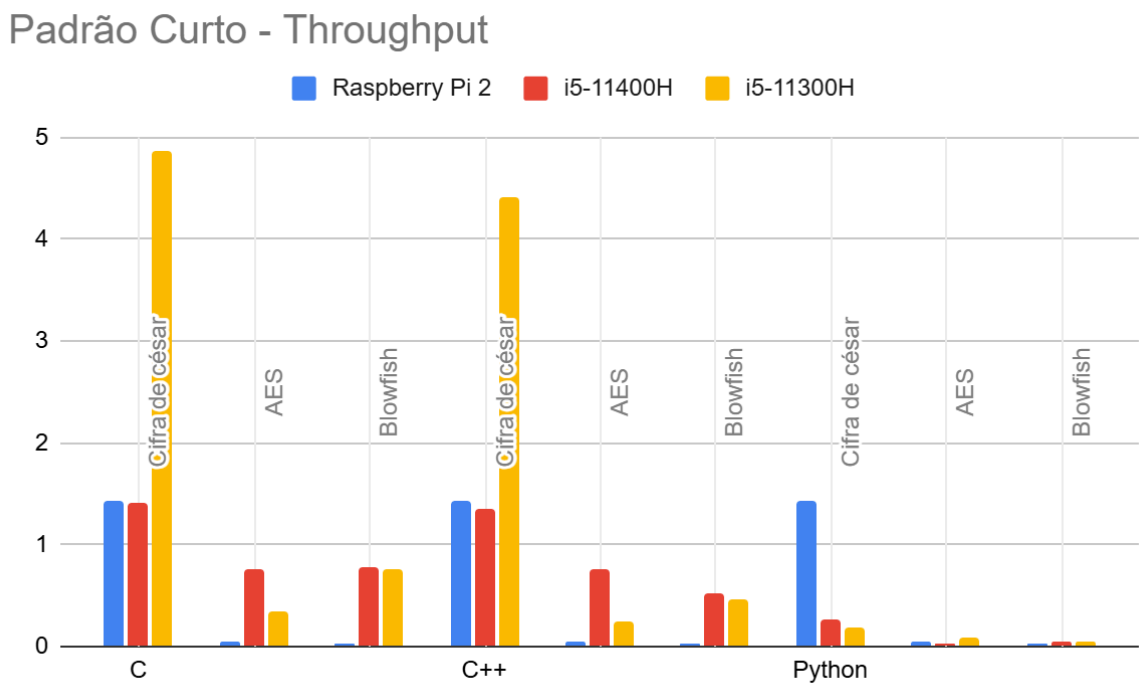


A análise dos valores médios de tempo de execução obtidos no benchmark com padrão curto — no qual o buffer foi preenchido repetindo o padrão fixo "ABC123" — tem como objetivo verificar se uma entrada com maior variação em relação ao padrão tipo massa uniforme influencia o desempenho das cifras. Apesar de ainda ser repetitivo, o padrão curto contém maior diversidade de caracteres, o que pode afetar algoritmos mais sensíveis à distribuição dos dados de entrada.

No Raspberry Pi 2, a Cifra de César manteve tempos médios muito baixos (5,57 ms), enquanto o AES e o Blowfish apresentaram tempos médios mais elevados (194,88 ms e 263,10 ms, respectivamente). Em comparação ao padrão de massa uniforme, esses valores são ligeiramente superiores, indicando que mesmo pequenas variações no conteúdo podem aumentar o custo computacional em hardware mais limitado.

Nos processadores Intel i5 de 11ª geração (11400H e 11300H), a diferença em relação ao padrão de massa uniforme foi menos perceptível para implementações em C e C++, com o AES e o Blowfish mantendo tempos médios próximos de 10 s em vários casos. Entretanto, há exceções, como no i5-11300H, onde o AES em C++ aumentou para 33,33 ms e o Blowfish para 20 ms, sugerindo que a variação do padrão pode interagir com aspectos internos da implementação e do compilador.

Na comparação entre linguagens, C e C++ continuam apresentando tempos muito próximos entre si, enquanto Python mantém tempos significativamente superiores, especialmente para o AES e o Blowfish. Por exemplo, no i5-11400H, o AES em Python atingiu 213,33 ms e o Blowfish 193,33 ms — valores ainda mais altos do que os obtidos no teste de massa uniforme, evidenciando que a linguagem interpretada sofre maior impacto de padrões mais variados.

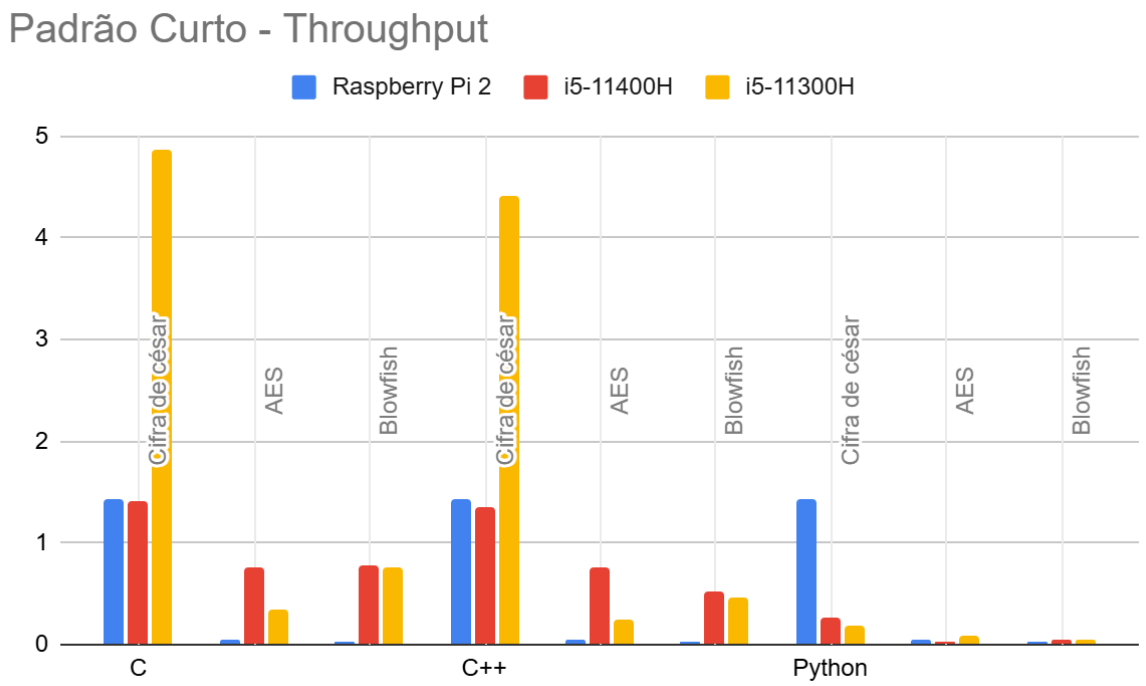


Para o padrão curto, os resultados seguem tendência semelhante: Cifra de César manteve o throughput mais alto em todas as plataformas (1,43 MB/s no Raspberry Pi 2 e 4,87 MB/s no i5-11300H em C).

AES e Blowfish melhoraram levemente em algumas combinações de plataforma e linguagem, mas ainda ficaram abaixo da cifra simples, com destaque para o i5-11400H em C e C++, onde o throughput chegou a 0,77 MB/s.

Python novamente apresentou throughput reduzido (por exemplo, AES 0,036 MB/s no i5-11400H), reforçando que linguagens interpretadas não aproveitam tão bem a capacidade do hardware em operações criptográficas complexas.

Análise - Texto Real

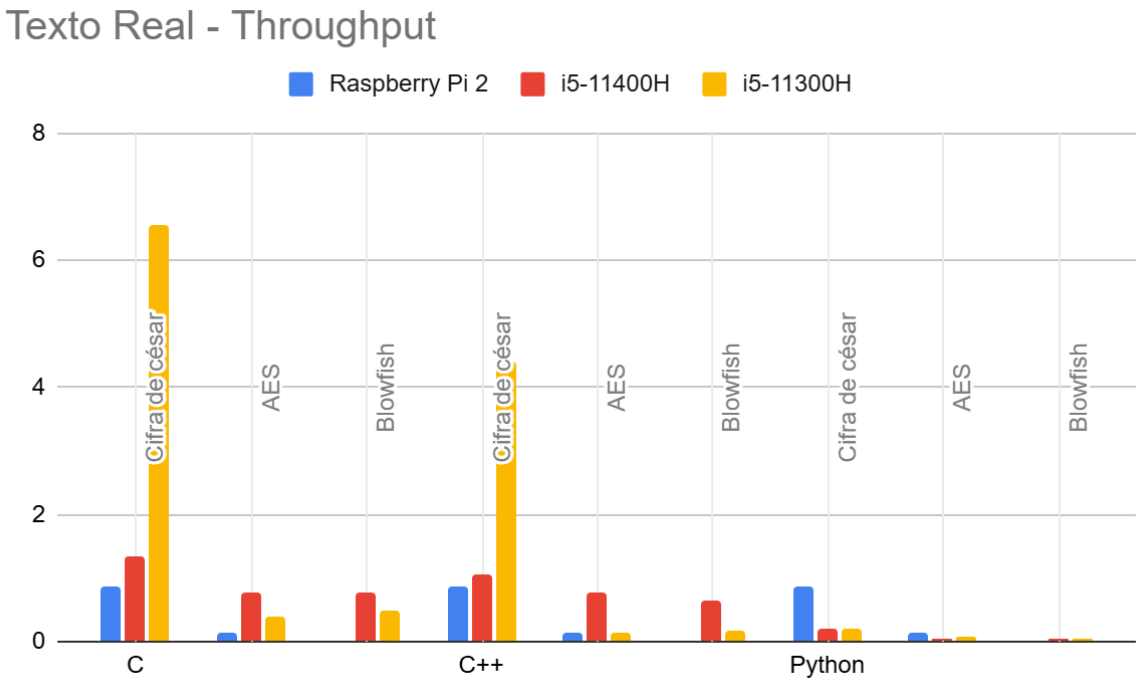


A análise dos valores médios de tempo de execução obtidos no benchmark com texto real — buffer preenchido repetindo a frase “O rápido marrom raposo salta sobre o cachorro preguiçoso.” — busca avaliar o desempenho das cifras diante de dados mais próximos de situações reais, contendo variação de letras, espaços e pontuação. Diferente dos padrões artificiais de massa uniforme e curto, esse cenário simula de forma mais fiel a natureza de textos comuns transmitidos ou armazenados em aplicações.

No Raspberry Pi 2, nota-se um aumento expressivo nos tempos médios em relação aos testes anteriores, principalmente na Cifra de César, que passou de 5,57 ms no padrão curto para 9,26 ms, um acréscimo superior a 65%. O AES e o Blowfish também apresentaram tempos mais elevados (234,95 ms e 333,23 ms, respectivamente), evidenciando que o conteúdo mais variado do texto real impacta significativamente o desempenho, especialmente em hardware de baixo poder de processamento.

Nos processadores Intel i5 de 11ª geração, a diferença em relação aos padrões anteriores foi menos uniforme. Em C, tanto o AES quanto o Blowfish mantiveram tempos médios de 10 s no 11400H, sugerindo que o hardware moderno é capaz de lidar com variações no conteúdo sem penalidades expressivas. Já no 11300H, alguns casos apresentaram aumento considerável, como o AES em C++ (66,66 ms) e o Blowfish em C++ (46,66 ms), possivelmente devido a interações entre a implementação, otimizações do compilador e características específicas da arquitetura.

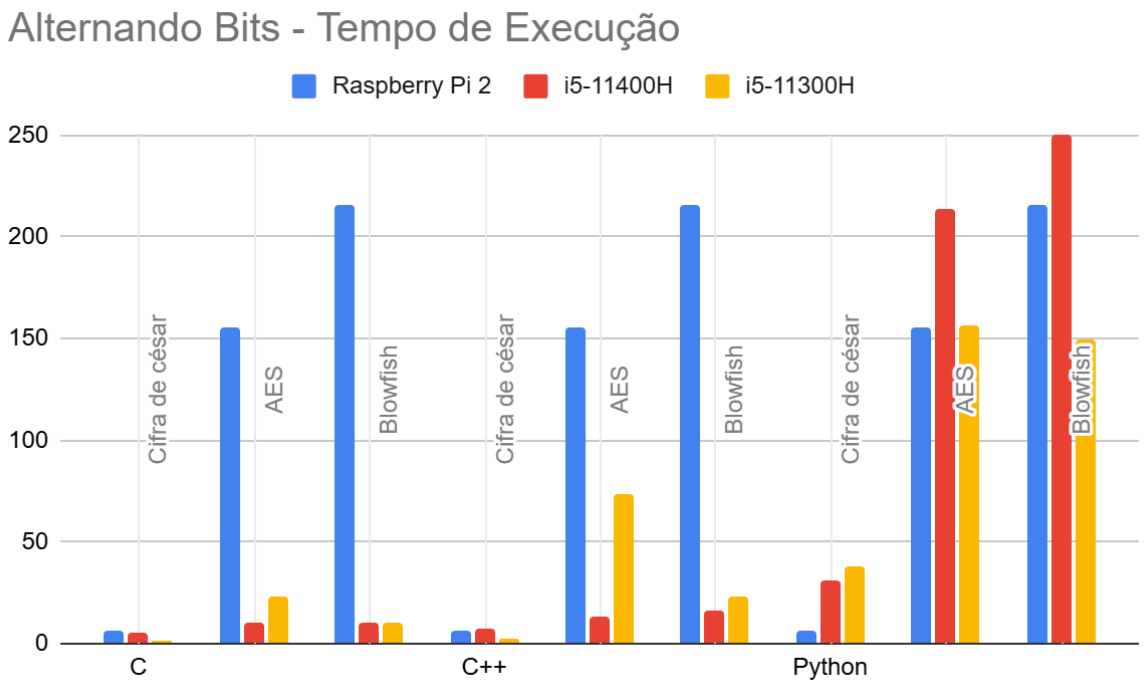
Na comparação entre linguagens, C e C++ continuam entregando tempos muito próximos na maioria dos casos, mas Python se mantém com médias bastante superiores, especialmente em algoritmos mais complexos. No 11400H, o AES e o Blowfish em Python atingiram 190 ms e 170 ms, respectivamente, enquanto no 11300H chegaram a 150 ms e 190 ms, indicando que a sobrecarga da linguagem interpretada é acentuada quando se processa dados mais variados.



Com a entrada de texto natural, observamos mudanças mais significativas. No Raspberry Pi 2, a Cifra de César teve uma queda de throughput (0,86 MB/s), enquanto a AES aumentou levemente (0,146 MB/s). Blowfish manteve valores baixos (0,026 MB/s), evidenciando que textos mais variados reduzem a eficiência de processamento de todos os algoritmos.

Nos computadores de processador Intel i5, a Cifra de César atingiu o maior throughput (6,55 MB/s no i5-11300H em C), refletindo o baixo custo da cifra simples mesmo com entradas mais complexas. AES e Blowfish tiveram aumento marginal em algumas plataformas, mas permanecem muito abaixo da cifra simples. Python manteve throughput baixo em todos os casos, mostrando que textos reais amplificam o impacto da interpretação no desempenho.

Análise - Alternando Bits



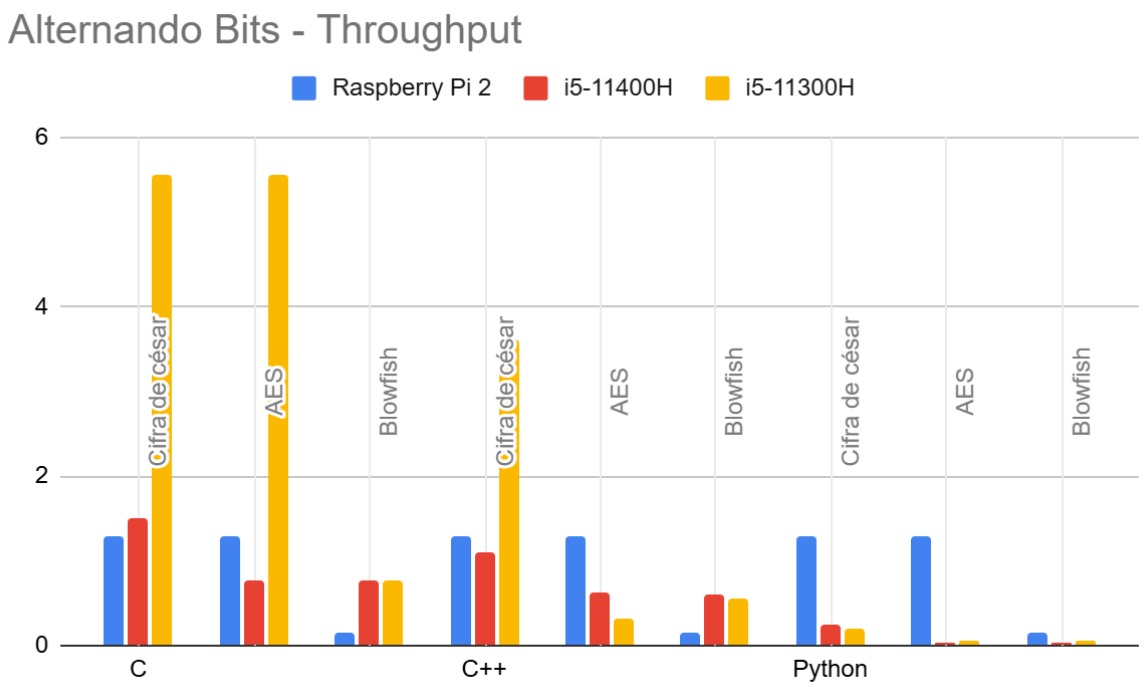
A análise dos valores médios de tempo de execução obtidos no benchmark com alternância de bits — em que o buffer alterna entre os caracteres “A” e “B” a cada posição — busca identificar o impacto de um padrão altamente previsível, mas com mudanças regulares de byte, no desempenho das cifras. Esse tipo de entrada permite avaliar como cada algoritmo responde a uma alternância binária simples, que pode influenciar tabelas de substituição e processos internos de criptografia.

No Raspberry Pi 2, a Cifra de César manteve tempo médio baixo (6,20 ms), ligeiramente acima do registrado no padrão de massa uniforme e próximo ao do padrão curto. AES e Blowfish também apresentaram tempos mais baixos que no teste de texto real, com médias de 155,13 ms e 215,49 ms, respectivamente, sugerindo que a previsibilidade do padrão reduz a carga de processamento em comparação a dados mais variados.

Nos processadores Intel i5 de 11ª geração, a estabilidade nos tempos de execução para implementações em C foi novamente evidente, com AES e Blowfish mantendo aproximadamente 10 ms no 11400H. No entanto, no 11300H e em algumas

execuções em C++, houve aumentos consideráveis — como o AES em C++ (73,33 ms) e o Blowfish em C++ (23,33 ms) — relacionados possivelmente à forma como o compilador otimiza ou processa padrões binários repetitivos.

Quanto às linguagens, C e C++ continuam próximas em desempenho para a maioria dos cenários, mas Python mantém tempos médios significativamente mais altos, especialmente para cifras mais complexas. No 11400H, o AES e o Blowfish em Python atingiram 213,33 ms e 250 ms, respectivamente, enquanto no 11300H chegaram a 156,66 ms e 150 ms. Esses valores mostram que a sobrecarga da linguagem interpretada permanece significativa, mesmo quando o padrão de dados é previsível.



No benchmark de alternância binária, o Raspberry Pi 2 registrou throughput levemente superior ao padrão de texto real para Cifra de César (1,29 MB/s) e AES (1,29 MB/s), sugerindo que padrões previsíveis podem melhorar ligeiramente a eficiência. Blowfish apresentou aumento expressivo em Python (0,146 MB/s), possivelmente devido à simplicidade do padrão.

Já para os computadores Intel I5, os valores foram mais variados: Cifra de César manteve altos valores (5,55 MB/s no i5-11300H em C), enquanto AES e Blowfish apresentaram desempenho heterogêneo, com alguns casos chegando a 5,55 MB/s para AES (i5-11300H em C), mostrando que padrões altamente previsíveis podem favorecer certos algoritmos em hardware moderno. Python continuou com throughput baixo, confirmando seu custo elevado em criptografia interpretada, mesmo com padrões simples.

Comparação dos Benchmarks

Uma análise comparativa entre os benchmarks deixa claro que a complexidade e variabilidade do padrão de dados têm impacto direto no desempenho dos algoritmos criptográficos. No caso do padrão de massa uniforme, a Cifra de César apresentou tempos de execução baixos e throughput elevado, enquanto AES e Blowfish tiveram desempenho mais limitado, especialmente em hardware de menor capacidade, como o Raspberry Pi 2. Pequenas variações no conteúdo, como no padrão curto com o buffer repetindo “ABC123”, resultaram em ligeiro aumento no tempo de execução para algoritmos mais complexos, embora o throughput tenha apresentado uma leve melhoria em processadores mais potentes, indicando que pequenas alterações no padrão podem afetar a carga de processamento em sistemas limitados.

Quando o benchmark utilizou texto real, com maior diversidade de caracteres, espaços e pontuação, observou-se um aumento significativo no tempo médio de execução, principalmente no Raspberry Pi 2, e uma redução expressiva do throughput em todos os algoritmos, com impacto mais acentuado em Python. Esse comportamento demonstra que a variabilidade natural dos dados aumenta o custo computacional, exigindo mais do hardware e das implementações interpretadas. Por outro lado, o benchmark de alternância de bits, em que o buffer alterna entre “A” e “B”, apresentou tempos intermediários e, em alguns casos, ligeira melhora no throughput em comparação ao texto real. Isso indica que padrões previsíveis podem

favorecer a execução, especialmente em hardware moderno, reduzindo parcialmente a penalidade imposta por algoritmos mais complexos.

O hardware utilizado é outro fator determinante para o desempenho. Dispositivos mais limitados, como o Raspberry Pi 2, exibiram os maiores tempos de execução e menor throughput para algoritmos complexos, independentemente do padrão de dados. Já os processadores Intel i5 modernos (11400H e 11300H) reduziram drasticamente os tempos e aumentaram significativamente o throughput, tornando os efeitos da variação de padrão menos perceptíveis, principalmente em implementações compiladas.

Além disso, a linguagem de implementação mostrou-se fundamental na determinação do desempenho. C e C++ apresentaram comportamento consistente e eficiente em todos os benchmarks, aproveitando melhor a capacidade de processamento do hardware. Em contraste, Python apresentou tempos de execução elevados e throughput significativamente reduzido, principalmente em padrões mais variados e para algoritmos mais complexos, evidenciando que a sobrecarga de execução interpretada amplifica o impacto da variabilidade dos dados.

Considerações Finais

O projeto de desenvolvimento da biblioteca de criptografia Crypgas em linguagem C para sistemas bare-metal permitiu uma análise aprofundada do desempenho de diferentes algoritmos em múltiplas plataformas. A avaliação, baseada em tempo de execução e throughput, demonstrou que a eficiência dos algoritmos está intrinsecamente ligada à sua complexidade, ao tipo de dados de entrada e, de forma crucial, às capacidades do hardware e à linguagem de programação utilizada.

Os resultados do benchmark evidenciaram a Cifra de César como o algoritmo mais rápido, com os menores tempos de execução e os maiores valores de throughput em todas as plataformas, especialmente em hardware limitado como o Raspberry Pi 2. Essa superioridade de desempenho reflete sua simplicidade e baixo custo

computacional. Em contrapartida, os algoritmos modernos e mais robustos, AES e Blowfish, apresentaram tempos de execução significativamente maiores e throughput mais baixo no Raspberry Pi 2, ilustrando o peso da sua complexidade em um ambiente de processamento limitado.

A variabilidade dos dados de entrada mostrou-se um fator determinante no desempenho. Enquanto a Cifra de César teve seu desempenho afetado por padrões de texto mais variados, como o "texto real" , o AES e o Blowfish demonstraram maior eficiência com padrões previsíveis, como a "alternância de bits".

A comparação entre as plataformas de hardware ressaltou a importância do processador para o desempenho. O Raspberry Pi 2 (ARM Cortex-A7) mostrou-se mais limitado, com tempos de execução mais longos para algoritmos complexos. Em contraste, os processadores Intel i5 modernos (11400H e 11300H) lidaram com a complexidade do AES e do Blowfish de forma muito mais eficiente, reduzindo drasticamente os tempos de processamento e tornando o impacto da variação dos dados de entrada menos perceptível.

Por fim, a análise das linguagens de programação destacou o papel fundamental das linguagens compiladas (C e C++) em comparação com a linguagem interpretada (Python). As implementações em C e C++ apresentaram desempenho superior e consistente, aproveitando de forma mais eficiente as capacidades do hardware. Já o Python, apesar de sua versatilidade, impôs uma sobrecarga de interpretação que resultou em tempos de execução significativamente mais altos e throughput reduzido, especialmente para os algoritmos mais complexos e com dados de entrada variados.

Em suma, o projeto cumpriu o objetivo de desenvolver uma biblioteca de criptografia e ofereceu uma visão prática sobre os trade-offs entre segurança e desempenho em diferentes contextos computacionais. Ficou claro que, para aplicações críticas de segurança em sistemas embarcados de baixo poder, a escolha do algoritmo e da linguagem de programação é tão relevante quanto a capacidade do hardware.