**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO**

# OpenFHE-Embedded: Criptografia Homomórfica para Sistemas Embarcados

**Gabriel de Oliveira Rosa Mariano Madeira**

**Projeto Final de Graduação**

**Centro Técnico Científico - CTC**

**Departamento de Informática**

Curso de Graduação em Ciência da Computação

Rio de Janeiro, Dezembro de 2025


**Gabriel de Oliveira Rosa Mariano Madeira**

# OpenFHE-Embedded: Criptografia Homomórfica para Sistemas Embarcados

Proposta de Projeto Final, apresentado ao programa de Graduação do Departamento de Informática da PUC-Rio como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Anderson Oliveira da Silva

Rio de Janeiro, Dezembro de 2025

## Resumo

de Oliveira Rosa Mariano Madeira, Gabriel. Oliveira da Silva, Anderson. OpenFHE-Embedded. Rio de Janeiro, 2025. 22 p. Proposta de Projeto Final II - Departamento de Informática. Pontifícia Universidade Católica do Rio de Janeiro.

A OpenFHE-Embedded é a versão da biblioteca OpenFHE de criptografia homomórfica portada e otimizada para o micro-controlador nRF52840DK de acordo com suas limitações de recursos computacionais permitindo apenas a cifragem de dados. O micro-controlador deve receber dados a partir do seu sensor, realizar a cifragem dos mesmo, e quando necessário, enviá-los para um outro computador poder realizar computações nos dados difrados. Dito isso, o foco desse projeto encontra-se em capturar os dados e cifrá-los. A solução consta da utilização do RTOS Zephyr e do emulador Renode para facilitar e auxiliar no desenvolvimento do projeto.



**Palavras-chave:** Criptografia Homomórfica, Criptografia, nRF52840DK, Zephyr, Renode, IoT



## Abstract

de Oliveira Rosa Mariano Madeira, Gabriel. Oliveira da Silva, Anderson. Rio de Janeiro, 2025. 22 p. Final Project II - Department of Informatics. Pontifícia Universidade Católica do Rio de Janeiro.

OpenFHE-Embedded is the version of the OpenFHE homomorphic encryption library ported and optimized for the nRF52840DK microcontroller, according to its computational resource limitations, allowing only data encryption. The microcontroller must receive data from its sensor, perform the encryption of this data, and when necessary, send it to another computer to perform computations on the encrypted data. That said, the focus of this project is on capturing and encrypting the data. The solution involves the use of the Zephyr RTOS and the Renode emulator to facilitate and assist in the project's development.



**Keywords:** Homomorphic Encryption, Cryptography, nRF52840DK, Zephyr, Renode, IoT

# Sumário

# 1 Introdução

A progressiva digitalização dos processos industriais, impulsionada pela Indústria 4.0, tem levado a uma proliferação de sensores e dispositivos conectados à Internet das Coisas (IoT), gerando um volume exponencial de dados em ambientes como digital twins. Nesse contexto, a segurança da informação e a privacidade dos dados coletados tornam-se imperativas, não apenas para garantir a integridade operacional e proteger ativos intelectuais críticos, mas também para mitigar riscos associados a falhas de segurança, como interrupção da produção. A análise e o processamento desses dados são cruciais para otimizar a produção e habilitar funcionalidades avançadas, contudo, essa etapa frequentemente ocorre em infraestruturas de nuvem. Apesar de suas vantagens em escalabilidade e custo, o uso de plataformas de nuvem pode introduzir vulnerabilidades significativas, pois a decriptação de dados para processamento em tais ambientes expõe informações sensíveis a potenciais ameaças, comprometendo a confidencialidade, especialmente quando a confiabilidade da infraestrutura de nuvem não pode ser plenamente assegurada.

Nesse cenário de crescente preocupação com a segurança de dados em ambientes de processamento distribuído, a criptografia homomórfica surge como uma solução tecnológica promissora. A criptografia homomórfica permite que operações computacionais sejam realizadas diretamente sobre dados cifrados, eliminando a necessidade de decriptografia prévia e, consequentemente, mantendo a confidencialidade dos dados mesmo quando processados por terceiros ou em ambientes sobre os quais não se tem total controle de segurança.

Considerando o potencial da criptografia homomórfica para o contexto industrial, este projeto foca no desenvolvimento de uma versão da biblioteca de criptografia homomórfica OpenFHE especificamente adaptada para micro-controladores. O objetivo central é capacitar dispositivos de baixa capacidade computacional, como sensores em plantas industriais ou componentes de digital twins, a realizar a encriptação homomórfica dos dados diretamente no ponto de coleta. É fundamental ressaltar que o escopo desta implementação se concentra exclusivamente na funcionalidade de cifragem. Desta forma, os dados já saem criptografados do dispositivo sensor, permitindo que sejam transmitidos de forma segura para sistemas mais potentes, como plataformas em nuvem, onde as operações homomórficas sobre os dados cifrados poderão ser executadas. Esta abordagem visa mitigar os

riscos associados à confiabilidade da infraestrutura de nuvem e à potencial existência de ameaças, garantindo que o dado permaneça seguro mesmo durante sua modificação em ambientes externos.

# 2 Situação Atual

## 2.1 Glossário

Esse projeto utiliza os conceitos e a terminologia baseada nos seguintes termos:

- Criptografia Homomórfica (CH): Tipo de criptografia utilizada nesse projeto a qual não requer a decriptografia do dado para que o mesmo seja modificado;
- Digital Twin (DT): Gêmeo Virtual de um ambiente ou pessoa, no contexto deste trabalho, trata-se de um digital twin de uma indústria;
- RTOS: Real-Time Operating System é um sistema operacional focado em executar tarefas de prazos previsíveis, garantindo respostas rápidas a eventos.

## 2.2 Soluções Existentes

A aplicação de técnicas de criptografia homomórfica em ambientes com recursos computacionais restritos, como micro-controladores, é um campo de pesquisa e desenvolvimento emergente, mas de crescente importância. Embora a maioria das bibliotecas de criptografia homomórfica tenha sido historicamente projetada para servidores ou sistemas com maior capacidade de processamento, esforços recentes têm buscado viabilizar seu uso em dispositivos de borda. Nesse contexto, um trabalho de destaque é o SEAL-Embedded, desenvolvido pela Microsoft Research [1]. Trata-se de uma versão da biblioteca de criptografia homomórfica da Microsoft, a Simple Encrypted Arithmetic Library (SEAL), que foi especificamente adaptada e otimizada para operar em sistemas embarcados e micro-controladores. O objetivo principal do SEAL-Embedded é estender as capacidades da criptografia homomórfica para dispositivos com limitações de processamento e memória, como os encontrados em sensores e atuadores na Internet das Coisas (IoT). Este trabalho da Microsoft [1] serve como um importante precedente e ponto de referência, ilustrando abordagens e desafios na portabilidade de bibliotecas de criptografia homomórfica para o domínio embarcado. A existência de soluções como o SEAL-Embedded reforça a relevância da proposta deste projeto, que busca um objetivo

similar utilizando, contudo, a biblioteca OpenFHE como base para o desenvolvimento de um módulo de criptografia homomórfica para micro-controladores.

# 3 Objetivos do Trabalho

## 3.1 Descrição da Solução Proposta

Este trabalho propõe a estratégia do desenvolvimento da OpenFHE-Embedded, uma biblioteca de criptografia homomórfica otimizada e adaptada a partir da OpenFHE, para sistemas embarcados e micro-controladores com recursos computacionais limitados, tornando possível apenas a cifragem de dados. A solução será composta apenas por essa biblioteca a ser desenvolvida utilizando ferramentas apropriadas para o ambiente IoT.

A estatégia é direcionada ao micro-controlador nRF52840DK para cifrar os dados e posteriormente enviá-los para outro computador poder realizar as computações necessárias nas cifras.

Para facilitar o desenvolvimento, será utilizado o RTOS Zephyr.

Além do Zephyr, será utilizado também o emulador Renode para emular o micro-controlador nRF52840DK e poder testar o código em desenvolvimento. Isso facilitará a validação da biblioteca sem ser necessário instalá-la no micro-controlador físico a cada mudança realizada.

## 3.2 Objetivos Específicos

Dada a descrição do problema e solução proposta, os objetivos do projeto incluem:

- Configuração do micro-controlador nRF52840DK virtualizado com Renode;
- Configuração do ambiente de desenvolvimento utilizando o Zephyr;
- Estudo de soluções similares já existentes
- Estudo das otimizações possíveis a serem realizadas na versão original da biblioteca de criptografia;
- Elaboração de estratégias de otimização para a versão Embedded da biblioteca;

### 3.2.1 Escopo do Sistema Desejado

O projeto abrange o desenvolvimento de uma biblioteca portada e otimizada para o micro-controlador nRF52840DK permitindo a cifragem de dados, incluindo:

- Algoritmos de criptografia homomórfica otimizados para ambientes com recursos computacionais limitados;

**Usuários/Programadores e Situações que se Deseja Apoiar**

O sistema se destina a:

- Indivíduos ou instituições desenvolvendo aplicações para o mesmo micro-controlador e buscam implementar segurança dos dados a partir do mesmo tipo de criptografia;
- Sistemas que desejam realizar computação em dados mesmo enquanto cifrados.

**Diferenças ou Avanços em Relação a Soluções Existentes**

Em comparação a outras bibliotecas de criptografia homomórfica, o projeto oferece avanços significativos dado que a biblioteca escolhida, possui a capacidade de Criptografia por Reencriptação de Proxy (PRE). Onde diferente de apenas cifrar dados para uma única chave de encriptação, a PRE permite que um texto cifrado, originalmente cifrado sob uma chave pública, seja transformado por uma entidade intermediária (o proxy) em um novo texto cifrado do mesmo dado, porém sob uma chave pública diferente. Lembrando que isso é realizado a partir do proxy sem a capacidade de decriptar o dado original, preservando sua confidencialidade.
No contexto industrial proposto, onde dados de sensores podem precisar ser acessados por diferentes sistemas ou departamentos, essa capacidade é particularmente vantajosa. Permite, por exemplo, que um sensor cifre dados com uma chave geral e, posteriormente, um gateway ou um serviço na nuvem (atuando como proxy) cifre de novo de forma seletiva esses dados para destinatários autorizados, cada um possuindo sua própria chave de decriptação.

**Itens a Serem Implementados**

- Aplicação de recebimento de valores no micro-controlador;
- Biblioteca de criptografia homomórfica otimizada para cifrar dados recebidos pelo micro-controlador nRF52840DK;
- Testes realizados e documentados no emulador do dispositivo e no dispositivo físico.

# 4 Plano de Ação

## 4.1 Estudos a serem realizados

Os seguintes estudos serão realizados ao longo do desenvolvimento do projeto:

- Solução desenvolvida em [1] analisando similaridades e diferenças
- Algoritmos de encriptação utilizados;
- Estratégias de otimização do algoritmo de encriptação;

## 4.2 Atividades a serem realizadas

As principais atividades a serem realizadas incluem:

- Aplicação das estratégias de otimização usadas em [1] para a biblioteca OpenFHE;
- Desenvolvimento do código para interpretação de dados do sensor do micro-controlador;
- Escrita, edição e entrega da proposta do Projeto Final 1;
- Testes da biblioteca desenvolvida em emulador e micro-controlador físico;
- Redação dos relatórios dos Projetos finais 1 e 2.

## 4.3 Produtos que serão criados

Os seguintes produtos serão gerados como resultado deste projeto:

- Artigo detalhando o estudo completo sobre a abordagem do desenvolvimento da biblioteca para o micro-controlador
- Relatório do Projeto Final 1;
- Relatório do Projeto Final 2;
- Apresentação final do projeto.

## 4.4 Cronograma Projeto Final 1

1. Estudo da estrutura da bibioteca OpenFHE;
   (a) Enumeração das bibliotecas utilizadas dentro dessa biblioteca *e.g.* bibliotecas que auxiliam em algoritmos matemáticos existentes em criptografia.
   (b) Compreensão das dependências da funcionalidade de encriptação.
2. Configuração do ambiente de desenvolvimento utilizando Zephyr;

3. Configuração do emulador com Renode para auxílio na validação da biblioteca sendo desenvolvida;
4. Leitura, estudo e compreensão do trabalho realizado em [1];
5. Escrita de rascunho da proposta;
6. Edição do rascunho da proposta;
7. Escrita do relatório do Projeto Final 1.

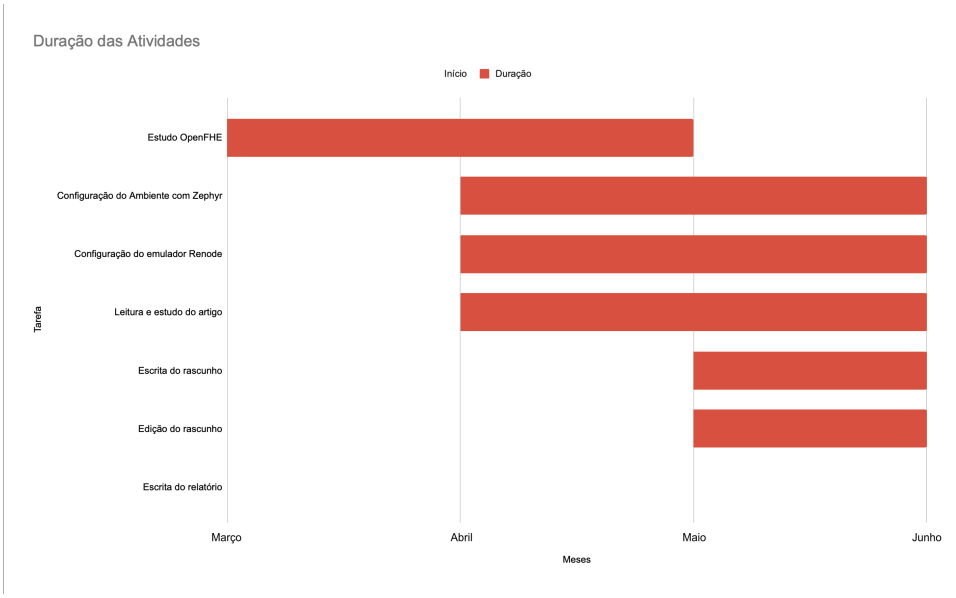


Figura 1: Cronograma do Projeto Final 1.

## 4.5 Cronograma Projeto Final 2

1. Estruturação da biblioteca otimizada;
2. Desenvolvimento da biblioteca otimizada para o micro-controlador nRF52840DK;
3. Escrita do relatório final e apresentação.

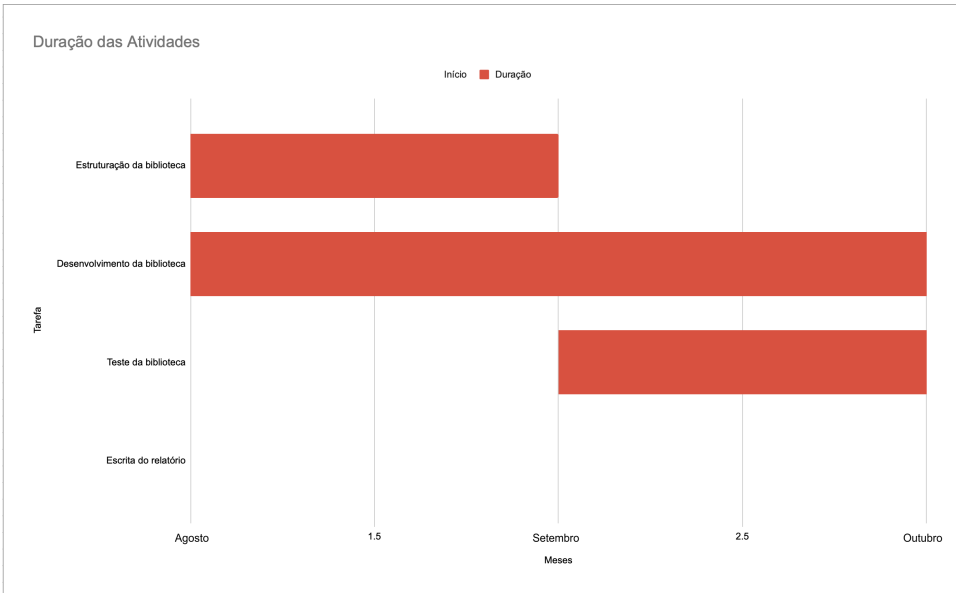


Figura 2: Cronograma do Projeto Final 2.

# 5 Desafio do micro-controlador

Um desafio decorre da disparidade entre os recursos exigidos pelas bibliotecas de HE (Criptografia Homomórfica) e os recursos disponíveis em micro-controladores. A OpenFHE é uma biblioteca extensa, complexa e com alto consumo de memória. Em contrapartida, o micro-controlador deste projeto, o nRF52840DK, possui recursos altamente limitados. Ele conta com um processador ARM Cortex-M4 com apenas 256KB de RAM e 1MB de memória flash. A migração direta da biblioteca OpenFHE **completa** é, sem dúvidas, impossível.

O objetivo específico deste projeto é desenvolver o OpenFHE-Embedded, uma adaptação da OpenFHE projetada para ser executada no nRF52840 sob o sistema operacional Zephyr RTOS. O escopo limita-se estritamente à funcionalidade de criptografia. Isso viabiliza um modelo seguro, onde o microcontrolador criptografa os dados no ponto de coleta e todas as computações homomórficas subsequentes são delegadas a um servidor confiável.

# 6 Arquitetura do projeto

Este projeto é definido por um conjunto de restrições que, coletivamente, formam um 'grande filtro', ditando cada decisão arquitetural subsequente:

- Criptografia CKKS Exclusiva: A biblioteca embarcada (OpenFHE-Embedded) implementará apenas a função de criptografia para o esquema CKKS. Todas

as outras funcionalidades (geração de chaves, descriptografia, operações homomórficas) serão removidas;

- Serialização ”cereal”: O projeto deve utilizar a biblioteca C++ cereal para a serialização de dados. Este é o mecanismo utilizado pela biblioteca OpenFHE completa;
- Headers C++ Idênticos: Para alcançar a serialização baseada em templates do cereal, o projeto deve ser compilado utilizando os headers C++ (.h) exatos e não modificados da biblioteca OpenFHE original para CryptoContext, PublicKey, Ciphertext e todas as suas dependências;
- Interoperabilidade: Uma PublicKey serializada pela biblioteca OpenFHE completa deve ser desserializada e utilizada pelo dispositivo embarcado. Um Ciphertext serializado pelo dispositivo embarcado deve ser desserializado e descriptografado com sucesso pela biblioteca OpenFHE completa.

# 7 Tese de desenvolvimento

Este artigo demonstrará que essas restrições criam um conflito entre as abstrações C++ impostas e o limite de 256KB de RAM, tornando impossível importar uma PublicKey com segurança padrão de 128 bits. Propomos, então, uma arquitetura de port que respeita todas as restrições, mantendo-se dentro do limite de 256KB de RAM. As principais contribuições deste artigo constituem uma estratégia de arquitetura e refatoração em três partes: Uma arquitetura de ”Contexto Mínimo e Mantendo Interoperabilidade”que redefine a fronteira do sistema, provando que parâmetros de segurança de 128 bits são inviáveis e propondo uma alternativa viável de parâmetros mínimos (N=4096). Uma estratégia de gerenciamento de memória determinístico que domina a complexidade dos headers C++ do OpenFHE ao habilitar o suporte total à biblioteca padrão C++ do Zephyr e redirecionar todas as alocações de heap da STL (ex: std::vector) para um heap de kernel do Zephyr dedicado e estático (k_heap) através da sobrescrita do operator **new** global. Um plano de aceleração de hardware que identifica o acelerador correto no Cortex-M4, DSP e não FPU, e mapeia a aritmética RNS de inteiros de 64 bits central do OpenFHE diretamente para as instruções SMLAL da CPU.

# 8 Visão geral da biblioteca OpenFHE

A OpenFHE é uma biblioteca C++ polimórfica e intensiva em templates. Seu design depende fortemente de recursos modernos do C++, incluindo a Standard Template Library (STL), exceções e Run-Time Type Information (RTTI).

**O CryptoContext**: O objeto central na OpenFHE é o CryptoContext. Ele possui todos os parâmetros criptográficos, tabelas pré-computadas para operações de Residue Number System (RNS) e Number Theoretic Transform (NTT), e serve para gerar todos os objetos associados (chaves, cifras). Objetos criados a partir de um contexto não podem ser usados em outro.

**Estruturas de Dados Principais**: As estruturas de dados primárias para CKKS são:

- `lbcrypto::DCRTPoly`: Representa um polinômio em forma RNS. Internamente, esta classe é construída sobre `std::vector`;
- `lbcrypto::PublicKey`: Chave pública na OpenFHE consiste em dois elementos DCRTPoly, convencionalmente chamados de $a$ e $b$;
- `lbcrypto::Ciphertext`: Um texto cifrado CKKS é estruturalmente idêntico a uma chave pública, consistindo também de dois elementos DCRTPoly.

**Serialização com cereal**: A OpenFHE usa explicitamente a cereal, uma biblioteca C++ baseada apenas em headers para toda a serialização. Isso é alcançado incluindo headers específicos (ex: cryptocontext-ser.h, ciphertext-ser.h) que registram as classes polimórficas da OpenFHE no framework de serialização da cereal. Essa integração exige que qualquer sistema interoperável não apenas use a cereal, mas também tenha acesso às definições exatas de classe (os arquivos .h) para interpretar corretamente os dados serializados.

# 9 Abordagem da SEAL-Embedded

O projeto é análogo ao trabalho realizado para criar o SEAL-Embedded a partir da biblioteca SEAL da Microsoft. Uma análise profunda da metodologia do SEAL-Embedded, no entanto, revela uma estratégia incompatível com as restrições deste projeto. A estratégia do SEAL-Embedded foi eliminar a complexidade do C++ para alcançar uma ocupação reduzida:

**Funcionalidade Removida**: É uma biblioteca apenas de criptografia, removendo a descriptografia, geração de chaves e todas as operações homomórficas;

**Linguagem**: Foi significativamente refatorada para ser ”inteiramente em C”para máxima portabilidade, com apenas wrappers em C++;
**Remoção de Recursos C++**: Esta refatoração centrada em C removeu todas as dependências da STL do C++, exceções e RTTI;
**Gerenciamento de Memória**: Abandonou completamente a alocação dinâmica de memória (ex: `new`, `malloc`) em favor de pools de memória estática configurados pelo desenvolvedor e computação sob demanda (on-the-fly) (ex: para tabelas NTT) para gerenciar o tradeoff entre RAM e flash.
**O problema dessa arquitetura**: A equipe do SEAL-Embedded alcançou seu objetivo sacrificando a compatibilidade ao nível de headers C++. Nosso projeto, em contraste, exige essa compatibilidade através das restrições de headers idênticos e do cereal. Portanto, não podemos seguir a abordagem do SEAL-Embedded. Devemos inventar uma nova estratégia de port que domine a complexidade dos headers C++ da OpenFHE, em vez de eliminá-los.

# 10 Analisando a factibilidade

Esta seção analisa a ocupação de RAM imposta pelas restrições do projeto e prova que o suporte a parâmetros de segurança padrão de 128 bits é impossível no dispositivo desejado.

## 10.1 Análise de RAM necessário

As restrições de headers idênticos e do cereal forçam o dispositivo embarcado a instanciar classes C++ (como PublicKey) com o exato mesmo layout de dados que o servidor. Conforme estabelecido, os objetos PublicKey e Ciphertext são compostos primariamente por elementos DCRTPoly, que são construídos sobre std::vector. Portanto, o tamanho da RAM desses objetos é uma função direta dos parâmetros criptográficos escolhidos para o CryptoContext:
$N$: A dimensão do Ring Dimension;
$k$: O número de módulos RNS, que é igual a $MultiplicativeDepth + 1$.

## 10.2 Parâmetros Criptográficos para 128-bit Security

A segurança da criptografia homomórfica é padronizada, e a OpenFHE adere aos parâmetros HEStd. Para alcançar a segurança clássica de 128 bits (HEStd_128_clas-

sic), são necessários valores mínimos para $N$ e para o tamanho total do módulo $log(Q)$.

Para o CKKS, a segurança de 128 bits tipicamente requer Ring Dimension $N$≥$16384$. Em algumas configurações de alto risco com módulos muito pequenos, $N = 8192$ pode ser usado.

## 10.3 Cálculo do RAM usado

Podemos agora quantificar a memória necessária para apenas um objeto PublicKey ou Ciphertext.

órmula de Ocupação de Memória: O tamanho de uma PublicKey (ou de um Ciphertext novo) é dominado por seus dois elementos DCRTPoly ($a$ e $b$). Cada DCRTPoly armazena $N$ coeficientes. No RNS, cada coeficiente é representado por um vetor de $k$ inteiros de 64 bits (8 bytes). $Custo\_RAM \approx 2$ (polinômios)$\times N$ (coeficientes)$\times$ $k$ (módulos) $\times 8$ (bytes/módulo). Vamos aplicar esta fórmula aos parâmetros de segurança de 128 bits.

Tabela 1: Ocupação Estimada de RAM de Objetos OpenFHE (Segurança de 128 bits)

Tabela 1: Ocupação Estimada de RAM de Objetos OpenFHE (Segurança de 128 bits)

| Param | $N$ | Profundidade | $k$ | Tamanho do Objeto (Chave/Cifra) | Possível? |
|---|---|---|---|---|---|
| **Padrão** | 16K | 5 | 6 | $2 \times 16384 \times 6 \times 8 =$ **1.57 MB** | **Não** |
| **Mínimo** | 16K | 1 | 2 | $2 \times 16384 \times 2 \times 8 =$ **512 KB** | **Não** |
| **Agressivo** | 8K | 1 | 2 | $2 \times 8192 \times 2 \times 8 =$ **256 KB** | **Não** (100% RAM) |

A análise apresentada na Tabela 1 é definitiva:

Uma PublicKey padrão de servidor (ex: com uma profundidade multiplicativa de 5) requer mais de 1,5 MB de RAM, excedendo em muito nossos 256KB.

Mesmo uma PublicKey mínima de 128 bits (profundidade 1) requer 512KB de RAM.

O conjunto de parâmetros agressivo de 128 bits mínimo absoluto ($N = 8192$, profundidade 1) resulta em uma PublicKey que tem 256KB. Este objeto sozinho consumiria toda a RAM do sistema. Esta análise prova que é **matematicamente impossível** desserializar uma PublicKey padrão de 128 bits no nRF52840.

# 11 Arquitetura ”Contexto Mínimo”

Esta seção propõe a solução do artigo, que resolve o conflito da seção anterior ao fazer um tradeoff crítico que é sacrificar segurança pela viabilidade.

A seção anterior provou que a segurança de 128 bits (que requer $N \geq 8192$) é impossível. Portanto, o único caminho viável a seguir é reduzir a Ring Dimension $N$ para um nível não tão seguro. Seguiremos o precedente estabelecido pelo artigo do SEAL-Embedded, que utilizou $N = 4096$ para seu alvo Cortex-M4.

Este é o tradeoff central: devemos sacrificar explicitamente a segurança de 128 bits para satisfazer todas as outras restrições de interoperabilidade. O artigo deve declarar esta limitação claramente, especificando que o nível de segurança HEStd_NotSet é utilizado.

Esta nova arquitetura abandona a ideia de serializar o CryptoContext e baseia-se em uma configuração de parâmetros mínimos codificados hard-coded.

Lado do Servidor: Um CryptoContext é gerado com parâmetros mínimos: $RingDim = 4096$ e $MultiplicativeDepth = 0$.

Lado do Servidor: A PublicKey (agora muito pequena) é gerada a partir deste contexto, serializada via cereal e transmitida para o dispositivo.

Lado do Dispositivo (Boot): O dispositivo não desserializa um contexto. Ele gera seu próprio CryptoContext chamando GenCryptoContext com parâmetros idênticos e hard-coded ($N = 4096$, $Depth = 0$). Isso cria um objeto de contexto funcional e mínimo na RAM.

Lado do Dispositivo (Runtime): O dispositivo desserializa a PublicKey do servidor usando cereal e os headers OpenFHE idênticos. Esta operação é funciona porque o CryptoContext local do dispositivo corresponde aos parâmetros da chave recebida.

Lado do Dispositivo (Runtime): O dispositivo coleta dados do sensor (ex: um std::vector<double>) e chama a função Encrypt(pk, data). Isso cria um objeto Ciphertext.

Lado do Dispositivo (Runtime): O dispositivo serializa o novo objeto Ciphertext usando cereal e o transmite para o servidor.

Lado do Servidor: O servidor, mantendo o mesmo contexto $N = 4096$, desserializa e descriptografa com sucesso o Ciphertext.

Esta arquitetura alcança todos os objetivos do projeto: criptografia CKKS no dispositivo, uso do cereal, uso de headers idênticos e interoperabilidade total de PublicKey/Ciphertext.

### 11.1 Cálculo de RAM Revisado

Agora analisamos o custo de RAM desta nova arquitetura mínima.

Parâmetros:

- $N = 4096$;
- $MultiplicativeDepth = 0$;
- $k = 1$ (Isto implica apenas um módulo RNS, ex: um único primo de 60 bits).

Cálculo do Tamanho do Objeto:

Tamanho da PublicKey: $2{\times}4096{\times}1{\times}8$ bytes = 65.536 bytes (64 KB)

Tamanho do Ciphertext: $2{\times}4096{\times}1{\times}8$ bytes = 65.536 bytes (64 KB)

Estes objetos são agora pequenos o suficiente para serem gerenciáveis. O pico de memória usada ocorrerá durante a criptografia, quando tanto a PublicKey (64 KB) quanto o Ciphertext (64 KB) devem existir na RAM simultaneamente, resultando em um total de 128 KB.

Tabela 2: Cálculo de RAM viável para OpenFHE-Embedded (N=4096, k=1)

| Componente | Tamanho | Justificativa |
|---|---|---|
| **Zephyr RTOS + Stacks** | 32 KB | Baseline padrão para uma aplicação mínima Zephyr com c++ e kernel. |
| **CryptoContext Estático** | 16 KB | Tamanho estimado para um contexto mínimo com $N = 4096$ apenas com parâmetros e tabela RNS para $k = 1$. |
| **Static C++ Heap (k_heap)** | 132 KB | Uma pool statically-defined reservada exclusivamente para todos os objetos dinâmicos de C++. |
| **Alocação máxima 1: PublicKey** | 64 KB | Desserialização para a k_heap |
| **Alocação máxima 2: Cifra** | 64 KB | Alocado na k_heap durante encriptação. |
| **Total Static Footprint** | 180 KB | 32 (OS) + 16 (Context) + 132 (Heap Pool) = 180 KB |
| **RAM restante** | 76 KB | 256KB (Total) - 180KB (Static) = 76KB |

Esta arquitetura é viável. Ela deixa 76 KB de RAM para a stack do programa, serviços do sistema e outras lógicas de aplicação, encaixando-se com sucesso dentro da restrição de 256KB.

## 12 Estratégia de refatoração I: Ambiente e build C++

Esta seção detalha como configurar o ambiente de compilação do Zephyr para compilar os complexos cabeçalhos C++ da OpenFHE.

### 12.1 Configuração do Zephyr Kconfig (prj.conf)

As seguintes flags do Kconfig devem ser definidas no arquivo prj.conf do projeto para habilitar os recursos de C++ exigidos pelos headers da OpenFHE e pelo ce-

real.

- `CONFIG_CPP=y`: Habilita C++
- `CONFIG_STD_CPP17`=y: OpenFHE e cereal são bibliotecas C++17/C++11, respectivamente
- `CONFIG_LIB_CPLUSPLUS`=y: Essa configuração é essencial. Indica o compilador do Zephyr a linkar com a STL de C++ completa e pré-compilada (ex: `libstdc++`). Isso fornece as implementações para `std::vector`, `std::string`, `std::shared_ptr` e outros componentes da STL dos quais os headers da OpenFHE dependem.
- `CONFIG_CPP_EXCEPTIONS`=y: Tratamento de erros da OpenFHE usa exceções C++ (ex: `throw lbcrypto::OpenFHEException`). A cereal também depende de exceptions. Essa flag deve estar habilitada para que o código funcione corretamente
- `CONFIG_RTTI`=y: A serialização da cereal usa Run-Time Type Information (RTTI) do C++ para identificar e serializar/desserializar corretamente as classes.

## 12.2 CMake do projeto

Para gerenciar a complexidade do projeto e satisfazer a restrição de ”headers idênticos”, este artigo propõe uma estrutura CMake de duas bibliotecas.
Biblioteca 1: openfhe_headers (Biblioteca INTERFACE)
Esta biblioteca será definida no CMake como uma biblioteca INTERFACE.
Uma biblioteca INTERFACE é um alvo ”virtual”que não possui arquivos de código-fonte e não produz nenhum artefato binário. É simplesmente uma coleção de propriedades, como include paths. Este alvo adicionará a biblioteca cereal ao seu include path. Ele também adicionará o subconjunto do diretório src da OpenFHE contendo todos os arquivos .h necessários aos seus target_include_directories como INTERFACE. Este alvo encapsula a restrição de ”headers idênticos”.
Biblioteca 2: openfhe_embedded_impl (Biblioteca STATIC)
Esta será uma biblioteca STATIC padrão. Esta biblioteca conterá os arquivos de código-fonte .cpp mínimos e escritos que fornecem as implementações para as funções declaradas em openfhe_headers. É aqui que 99,9% dos arquivos .cpp originais da OpenFHE são descartados e substituídos por nossas implementações mínimas e otimizadas. Por exemplo, forneceremos uma implementação para

`lbcrypto::PKECKKSRNS::Encrypt`, mas não para `Decrypt` ou `EvalAdd`.

CMakeLists.txt da Aplicação: O arquivo CMakeLists.txt da aplicação principal então vinculará estes alvos juntos, juntamente com a interface do Zephyr.

# 13 Estratégia de refatoração II: Gerenciamento de Memória

Nessa seção resolvemos o conflito entre as abstrações de alto nível da STL do C++ e a memória limitada de um sistema embarcado.

Começando a partir do problema com o `std::vector` em sistemas embarcados. Os cabeçalhos da OpenFHE, particularmente o DCRTPoly, são construídos sobre o `std::vector`. Quando um objeto como PublicKey é desserializado pelo cereal, os `std::vector`s dentro dele chamarão seus construtores e métodos resize. Estes métodos, por sua vez, chamam o operator new global para alocar memória do heap. No Zephyr, quando CONFIG_LIB_CPLUSPLUS está habilitado, o operator new global é fornecido pela biblioteca padrão, que internamente mapeia para `malloc()`. Este heap padrão da biblioteca C é tipicamente pequeno, compartilhado com todo o sistema e altamente suscetível à fragmentação. Tentar desserializar uma PublicKey de 64 KB com `malloc()` quase certamente falhará, levando a uma exceção `std::bad_alloc` ou a um travamento do sistema. Não podemos confiar neste heap imprevisível.

## 13.1 A Solução K_HEAP

Aproveitaremos o alocador de memória do kernel do Zephyr. Definiremos um pool de memória grande, estático e alinhado por palavra em tempo de compilação, usando o tamanho calculado em nosso cálculo de RAM. Isso é feito em um arquivo .cpp global:

```
#include <zephyr/kernel.h>
// Define um heap estático de 132KB para todas as alocações dinâmicas C++
K\_HEAP\_DEFINE(cpp\_heap, 132 * 1024);
```

Isso reserva 132KB de RAM na seção .bss, gerenciada pelo kernel do Zephyr, especificamente para nossos objetos C++.

### 13.2 Overwrite Global de `new` e `delete`

O passo final é forçar a biblioteca padrão C++ a usar nosso heap em vez de seu heap malloc padrão. Alcançamos isso fornecendo nossas próprias implementações globais do operator new e operator delete, que o linker usará em vez das versões da biblioteca padrão.

Esta técnica viabiliza toda essa arquitetura. Ela redireciona cada alocação de `std::vector` dos headers OpenFHE não modificados para nosso k_heap de 132KB. Podemos agora usar os headers como estão, satisfazendo a restrição de interoperabilidade do cereal sem estourar a memória do sistema.

## 14 Conclusão

Este artigo apresentou uma arquitetura viável, para portar a complexa biblioteca C++ OpenFHE para um microcontrolador com 256KB de RAM. A arquitetura lida com as restrições dos requisitos de interoperabilidade e de headers do projeto.

`Pontos principais`

Viabilidade: Provamos que o gasto de memória de uma PublicKey com segurança padrão de 128 bits (256KB–1.5MB) torna matematicamente impossível desserializá-la em um dispositivo com 256KB de RAM.

Arquitetura Proposta: Propusemos a arquitetura de Contexto Mínimo como a única solução viável. Esta arquitetura troca a segurança de 128 bits por um conjunto de parâmetros viável, com $N = 4096$, e deixa o CryptoContext hard-coded no dispositivo, serializando apenas os objetos PublicKey e Ciphertext de 64KB.

Estratégia de Memória C++: Demonstramos como utilizar headers C++ da OpenFHE habilitando a biblioteca padrão C++ completa do Zephyr (CONFIG_LIB_CPLUSPLUS, CONFIG_CPP_EXCEPTIONS, CONFIG_RTTI) e, em seguida, gerenciando sua memória. Isso é alcançado sobrescrevendo o operator `new` e `delete` globais para redirecionar todas as alocações de std::vector (e outras) para um k_heap grande e estático.

## Referências

[1] K. Micul, H. Chen, K. Laine, and K. Lauter, “SEAL-Embedded: Homomorphic Encryption for Microcontrollers,” *Cryptology ePrint Archive, Report 2020/1441*,

2020.

[2] Z. Temirbekova, A. Pyrkova, Z. Abdiakhmetova, and A. Berdaly, “Library of fully homomorphic encryption on a microcontroller,” in *2022 International Conference on Smart Information Systems and Technologies (SIST)*, pp. 1–5, 2022.

[3] Z. E. Temirbekova, G. Turken, and M. M. Barata, “Using fully homomorphic encryption in iot.,” in *DTESI (workshops, short papers)*, 2023.