CRIPTOGRAFIA: A SEGURANÇA DAS INFORMAÇÕES

1 Introdução

A criptografia é uma técnica utilizada para transformar dados legíveis em dados ininteligíveis, garantindo a confidencialidade e integridade das informações. Está presente em diversos aspectos da vida digital, como transações financeiras, comunicações e autenticações. Os métodos criptográficos evoluíram significativamente, desde algoritmos simples de substituição até técnicas complexas, como criptografia simétrica e assimétrica. A criptografia simétrica utiliza uma única chave para encriptar e desencriptar dados, enquanto a criptografia assimétrica envolve um par de chaves (pública e privada) para aumentar a segurança em sistemas de comunicação.

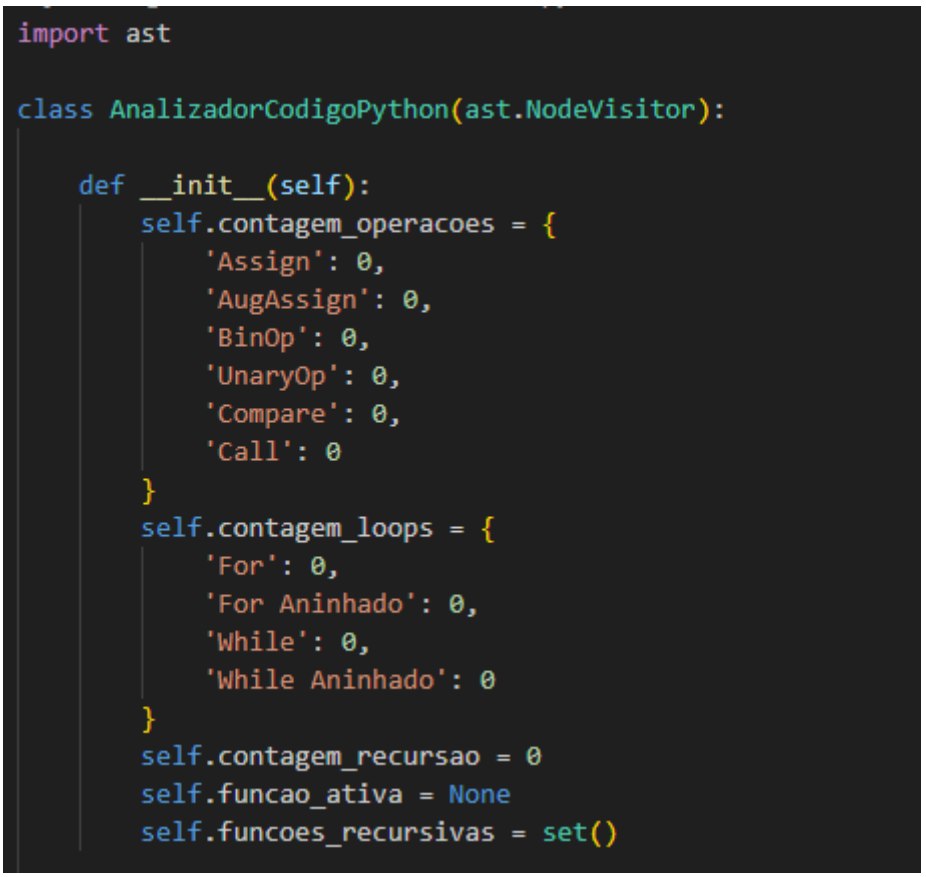
Uma das formas mais simples e antigas de criptografia é a cifra de César, que consiste em trocar cada letra do texto por outra correspondente de acordo com um alfabeto deslocado. Por exemplo:

ABCDEFGHIJKLMNOPQRSTUVWXYZ (texto plano)
DEFGHIJKLMNOPQRSTUVWXYZABC (alfabeto de ciframento)
A palavra ANALISE será representada pela cifra DQDOLVH

Atualmente, algoritmos como AES (Advanced Encryption Standard) e RSA são amplamente utilizados para proteger dados em redes, dispositivos e serviços online. O AES é um tipo de criptografia que usa uma única chave para codificar e decodificar informações rapidamente, sendo ideal para grandes volumes de dados. Já o RSA utiliza duas chaves, uma pública e uma privada, para garantir a segurança na troca de informações, especialmente em comunicações.

2 Analisador

Ao desenvolver um programa, é necessário pensar muito além de simplesmente programar; a maneira como estruturamos o código é crucial para a eficiência e a manutenção do sistema. Não se trata apenas de fazê-lo funcionar, mas de garantir que ele seja otimizado, fácil de entender e com desempenho adequado. Devemos sempre buscar um código que ofereça o melhor equilíbrio entre custo e qualidade, levando em conta não apenas a clareza, mas também o tempo de execução e o uso de recursos. Nesse sentido, é essencial utilizar a análise assintótica para avaliar a complexidade de um algoritmo, o que nos permite prever seu comportamento em diferentes cenários e escalas. A Notação Big(O) é uma ferramenta fundamental nesse processo, pois oferece uma forma matemática de medir a eficiência de um algoritmo em termos de tempo e espaço, possibilitando comparações e escolhas mais informadas sobre a melhor abordagem a ser adotada.



```python
import ast

class AnalizadorCodigoPython(ast.NodeVisitor):

    def __init__(self):
        self.contagem_operacoes = {
            'Assign': 0,
            'AugAssign': 0,
            'BinOp': 0,
            'UnaryOp': 0,
            'Compare': 0,
            'Call': 0
        }
        self.contagem_loops = {
            'For': 0,
            'For Aninhado': 0,
            'While': 0,
            'While Aninhado': 0
        }
        self.contagem_recursao = 0
        self.funcao_ativa = None
        self.funcoes_recursivas = set()
```

Figura 1.

Na Figura 1, importamos a biblioteca ast, que permite analisar o código Python como uma Árvore de Sintaxe Abstrata (AST), onde cada elemento do código é representado como um nó da árvore. A seguir, temos um código responsável por percorrer essa árvore para realizar uma análise detalhada, contando operações como atribuições, comparações, chamadas de funções, e loops, sejam aninhados ou não. Além disso, o código identifica chamadas recursivas em funções, permitindo uma visão estruturada da complexidade do código.

```python
    def visit(self, node):

        if isinstance(node, ast.FunctionDef):
            self.funcao_ativa = node.name
            self.generic_visit(node)
            self.funcao_ativa = None
        elif isinstance(node, ast.For):
            self.contagem_loops['For'] += 1
            nivel_aninhado = contar_loops_aninhados(node)
            if nivel_aninhado > self.contagem_loops['For Aninhado']:
                self.contagem_loops['For Aninhado'] = nivel_aninhado
            self.generic_visit(node)
        elif isinstance(node, ast.While):
            self.contagem_loops['While'] += 1
            nivel_aninhado = contar_loops_aninhados(node)
            if nivel_aninhado > self.contagem_loops['While Aninhado']:
                self.contagem_loops['While Aninhado'] = nivel_aninhado
            self.generic_visit(node)
        elif isinstance(node, ast.Call) and isinstance(node.func, ast.Name):
            if node.func.id == self.funcao_ativa:
                # Olha a recursão
                self.contagem_recursao += 1
                self.funcoes_recursivas.add(self.funcao_ativa)
        else:
            tipo_node = type(node).__name__
            if tipo_node in self.contagem_operacoes:
                self.contagem_operacoes[tipo_node] += 1
            self.generic_visit(node)
```

Figura 2.

O método visit é uma parte fundamental da análise de código, herdado da classe ast.NodeVisitor, que é usada para percorrer os elementos do código representados na árvore de sintaxe abstrata (AST). Ele visita cada nó da árvore e executa uma ação dependendo do tipo de nó encontrado, como a contagem de operações (atribuições, comparações), identificação de loops (aninhados ou não) e detecção de recursividade. O principal objetivo desse método é garantir que todos os nós da árvore, ou seja, todo o código, sejam percorridos e analisados de forma sistemática.

```python
def contar_loops_aninhados(node, profundidade=1):
    ## profundidade dos loops
    max_profundidade = profundidade
    for child in ast.iter_child_nodes(node):
        if isinstance(child, (ast.For, ast.While)):
            max_profundidade = max(max_profundidade, contar_loops_aninhados(child, profundidade + 1))
    return max_profundidade

def analisar_arquivo_codigo(caminho_arquivo):
    # realização da análise
    with open(caminho_arquivo, 'r') as file:
        codigo = file.read()
    arvore = ast.parse(codigo)
    analisador = AnalizadorCodigoPython()
    analisador.visit(arvore)
    return analisador.contagem_operacoes, analisador.contagem_loops, analisador.contagem_recursao, analisador.funcoes_recursivas
```

Figura 3.

Na Figura 3, a função contar_loops_aninhados determina a profundidade dos loops aninhados de forma recursiva, visitando cada nó da árvore de sintaxe. Já a função analisar_arquivo_codigo utiliza a classe AnalizadorCodigoPython para transformar o código em uma árvore de sintaxe abstrata (AST) e percorrê-la, coletando informações sobre operações, loops e recursividade.

```python
def exibir_notacao_assintotica(contagem_loops, contagem_recursao, funcoes_recursivas):
    max_for_aninhado = contagem_loops.get('For Aninhado', 0)
    max_while_aninhado = contagem_loops.get('While Aninhado', 0)

    if contagem_recursao > 0 and len(funcoes_recursivas) > 0:
        # Identificação de casos
        print("O(n log n) devido à recursão")
    elif max_for_aninhado == 0 and max_while_aninhado == 0:

        print("O(1)")
    elif max_for_aninhado == 0:
        print(f"O(n^{max_while_aninhado})")
    elif max_while_aninhado == 0:
        print(f"O(n^{max_for_aninhado})")
    else:
        print(f"O(n^{max_for_aninhado + max_while_aninhado})")

if __name__ == "__main__":
    caminho_arquivo = "copaAmerica.py"  # caminho do arquivo
    operacoes, loops, contagem_recursao, funcoes_recursivas = analisar_arquivo_codigo(caminho_arquivo)

    print("Contagem de Operações:")
    for operacao, contagem in operacoes.items():
        print(f"{operacao}: {contagem}")

    print("\nContagem de Loops:")
    for tipo_loop, contagem in loops.items():
        print(f"{tipo_loop}: {contagem}")

    print("\nTotal de Chamadas Recursivas:")
    print(f"Recursivas: {contagem_recursao} (funções recursivas: {', '.join(funcoes_recursivas)})")

    print("\nNotação Assintótica:")
    exibir_notacao_assintotica(loops, contagem_recursao, funcoes_recursivas)
```

Figura 4.

Na Figura 4, o código tem o objetivo de, através de condicionais, determinar a notação assintótica do código analisado e a quantidade de chamadas recursivas presentes no arquivo. Após essa análise, ele imprime os resultados detalhadamente, incluindo a contagem de operações, loops, chamadas recursivas e a complexidade computacional do código.

3 Conclusão

Concluímos que o estudo da complexidade de algoritmos é um tema de suma importância, pois permite avaliar a eficiência de um programa e seu comportamento em diferentes escalas de dados. Esse projeto permitiu automatizar essa análise, proporcionando uma visão mais clara sobre possíveis melhorias de performance e eficiência. Com a identificação automática de loops, recursividade e operações complexas, torna-se mais fácil tomar decisões informadas sobre otimizações, garantindo que o código seja mais eficiente e capaz de lidar com volumes maiores de dados de forma otimizada.

4 Referências

Crypto ID. A história da criptografia. CryptoID. Disponível em: https://cryptoid.com.br/identidade-digital-destaques/a-historia-da-criptografia-2/. Acesso em: 11 out. 2024.

Python Software Foundation. ast — Abstract Syntax Trees. Disponível em: https://docs.python.org/pt-br/3.7/library/ast.html. Acesso em: 12 out. 2024.

FreeCodeCamp. O que é a notação Big-O: complexidade de tempo e de espaço. FreeCodeCamp Brasil. Disponível em: https://www.freecodecamp.org/portuguese/news/o-que-e-a-notacao-big-o-complexidade-de-tempo-e-de-espaco/. Acesso em: 12 out. 2024.

Veritas. "Padrão de criptografia avançada: o guia definitivo para criptografia AES." Veritas. Disponível em: https://www.veritas.com/pt/br/information-center/aes-encryption. Acesso em: 12 out. 2024.