



UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

Isac Martins

Dartcryptosec: Desenvolvimento de uma Biblioteca Criptográfica para Dart

Florianópolis
2026

Isac Martins

Dartcryptosec: Desenvolvimento de uma Biblioteca Criptográfica para Dart

Trabalho de Conclusão de Curso de Graduação em Ciências da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para a obtenção do título de bacharel em Ciências da Computação.

Orientadora: Thaís Bardini Idalino, Dra.

Coorientador: Prof. Martín Augusto Gagliotti Vigil, Dr.

Florianópolis

2026

Ficha catalográfica gerada por meio de sistema automatizado gerenciado pela BU/UFSC.
Dados inseridos pelo próprio autor.

Martins, Isac

Dartcryptosec : desenvolvimento de uma biblioteca
criptográfica para Dart / Isac Martins ; orientadora, Thaís
Bardini Idalino, coorientador, Martín Augusto Gagliotti
Vigil, 2026.

141 p.

Trabalho de Conclusão de Curso (graduação) -
Universidade Federal de Santa Catarina, Centro Tecnológico,
Graduação em Ciências da Computação, Florianópolis, 2026.

Inclui referências.

1. Ciências da Computação. 2. Infraestrutura de Chaves
Públicas. 3. Criptografia. 4. Dart. 5. OpenSSL. I.
Idalino, Thaís Bardini. II. Vigil, Martín Augusto Gagliotti.
III. Universidade Federal de Santa Catarina. Graduação em
Ciências da Computação. IV. Título.

Isac Martins

Dartcryptosec: Desenvolvimento de uma Biblioteca Criptográfica para Dart

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de “bacharel em Ciências da Computação” e aprovado em sua forma final pelo Curso de Graduação em Ciências da Computação.

Florianópolis, 8 de julho de 2026.

Banca Examinadora:

Profa. Thaís Bardini Idalino, Dra.
Universidade Federal de Santa Catarina

Prof. Martín Augusto Gagliotti Vigil, Dr.
Universidade Federal de Santa Catarina

Prof. Jean Everson Martina, Dr.
Universidade Federal de Santa Catarina

Anthony Bernardo Kamers
Universidade Federal de Santa Catarina

À minha mãe, que me ensinou a lutar pelo que quero. Saudades.

AGRADECIMENTOS

À minha mãe, que partiu em 25 de janeiro de 2025 sem que eu pudesse mostrar-lhe o resultado deste trabalho: obrigado por cada palavra de incentivo, por me ensinar a lutar pelo que quero e por nunca ter deixado de acreditar em mim. Sua ausência pesa, mas sua presença permanece em tudo que construo.

À minha namorada Emanuelle, pelo suporte constante, pela paciência e pelo cuidado durante todo o período de desenvolvimento deste trabalho. Sua companhia fez a diferença.

À minha família, por nunca ter saído do meu lado. A segurança que sempre me ofereceram foi o alicerce que tornou esta jornada possível.

RESUMO

Este trabalho apresenta o desenvolvimento de uma biblioteca criptográfica multiplataforma em Dart, compatível com Linux e Android, com suporte básico de compilação para iOS e Windows, com o objetivo de facilitar a criação de aplicações que demandem interoperabilidade e funcionalidades criptográficas avançadas. A biblioteca integra FFI (*Foreign Function Interface*) com OpenSSL 4.0.0, oferecendo rotinas simplificadas que abstraem a complexidade inerente do OpenSSL. Com essa abordagem, desenvolvedores conseguem implementar operações criptográficas focadas na Infraestrutura de Chaves Públicas (ICP) sem a necessidade de lidar com detalhes específicos de baixo nível ou dependentes de cada plataforma.

A interface em Dart oferece uma camada unificada para o acesso às operações criptográficas implementadas. Inspirado na biblioteca libcryptosec, desenvolvida pelo Laboratório de Segurança em Computação (LabSEC) da Universidade Federal de Santa Catarina, o projeto busca facilitar a integração dessas operações em aplicações de alto nível. O protótipo inicial foi desenvolvido no LabSEC como parte do projeto Medida Inteligente, em parceria com o Instituto Nacional de Metrologia, Qualidade e Tecnologia (Inmetro). Neste trabalho, a implementação foi validada em Linux e Android.

A compatibilidade com o framework Flutter confere flexibilidade, permitindo que desenvolvedores criem aplicações híbridas e multiplataforma utilizando uma solução criptográfica unificada. A biblioteca foi comparada com soluções consagradas como Bouncy Castle e OpenSSL, evidenciando sua adequação para cenários que demandam ICP e interoperabilidade. Este trabalho contribui para o desenvolvimento de software seguro, ao fornecer uma ferramenta que simplifica a implementação de criptografia em diversos sistemas móveis.

Palavras-chave: Infraestrutura de Chaves Públicas (ICP); criptografia; Dart; Flutter; OpenSSL; FFI; multiplataforma; libcryptosec

ABSTRACT

This work presents the development of a cross-platform cryptographic library in Dart, compatible with Linux and Android, with basic compilation support for iOS and Windows, aiming to facilitate the creation of applications that demand interoperability and advanced cryptographic functionalities. The library integrates FFI (Foreign Function Interface) with OpenSSL 4.0.0, offering simplified routines that abstract the inherent complexity of OpenSSL. With this approach, developers can implement cryptographic operations focused on Public Key Infrastructure (PKI) without the need to deal with low-level or platform-specific details.

The Dart interface offers a unified layer that makes the use of cryptographic operations consistent and equivalent across different operating systems. Inspired by the libcryptosec library, developed by the Laboratory of Computer Security (LabSEC) at the Federal University of Santa Catarina (UFSC), the project adopts design and functionality principles that aim to facilitate the implementation of cryptography in high-level applications. The initial prototype of the library was developed at LabSEC as part of the Medida Inteligente project, in partnership with the National Institute of Metrology, Quality and Technology (Inmetro). In this work, the implementation was validated on Linux and Android.

Compatibility with the Flutter framework provides flexibility, allowing developers to create hybrid and cross-platform applications using a unified cryptographic solution. The library was compared with established solutions such as Bouncy Castle and OpenSSL, demonstrating its suitability for scenarios requiring PKI and interoperability. This work contributes to the development of secure software by providing a tool that simplifies the implementation of cryptography in various mobile systems.

Keywords: Public Key Infrastructure (PKI); cryptography; Dart; Flutter; OpenSSL; FFI; cross-platform; libcryptosec.

LISTA DE FIGURAS

Figura 1 – Exemplo da estrutura de um certificado digital X.509 v3.	31
Figura 2 – Funcionamento básico de uma ICP tradicional.	34
Figura 3 – Fluxograma de validação de cadeia de certificados X.509.	36
Figura 4 – Anatomia de uma estrutura CMS/CAdES (PKCS#7 SignedData), com destaque para os atributos assinados do perfil CAdES.	41
Figura 5 – Fluxo conceitual de uma chamada FFI entre Dart e uma biblioteca C nativa.	46
Figura 6 – Fluxo principal da arquitetura em sete camadas da dartcryptosec . Os componentes transversais de métricas e utilitários não são representados.	66
Figura 7 – Sequência de chamadas para a operação AES-256-GCM, atravessando as camadas <i>Facade</i> , <i>Operations</i> , FFI e <i>Models</i> , com as funções da API EVP do OpenSSL invocadas na camada FFI.	72
Figura 8 – Sequência de operações para validação de cadeia X.509 via <code>X509_verify_cert()</code> do OpenSSL.	78
Figura 9 – Consumo de memória RSS ao longo de 5 pontos de amostragem durante testes de stress.	80
Figura 10 – Latência média por categoria criptográfica, Linux x86_64, OpenSSL 4.0.0.	84
Figura 11 – Simulação de <i>handshake</i> TLS 1.3 com <i>cipher suite</i> TLS_AES_128_GCM_SHA256 (total: 0,848 ms).	88
Figura 12 – Mapa estrutural da superfície de ataque com anotações STRIDE nas arestas de fluxo de dados.	120
Figura 13 – Diagrama de Fluxo de Dados — AES-256-GCM Encrypt com AAD (9 processos).	121

LISTA DE TABELAS

Tabela 2	– Conjuntos de parâmetros do ML-KEM (FIPS 203).	44
Tabela 3	– Conjuntos de parâmetros do ML-DSA (FIPS 204).	45
Tabela 4	– Comparação de tamanhos: algoritmos clássicos <i>vs.</i> pós-quânticos.	45
Tabela 5	– Ferramentas e versões utilizadas no desenvolvimento e validação	50
Tabela 6	– Comparação unificada de funcionalidades entre bibliotecas criptográficas de referência e bibliotecas Dart.	58
Tabela 7	– Principais grupos funcionais vinculados à API do OpenSSL.	62
Tabela 8	– Ameaças STRIDE de prioridade elevada na <code>dartcryptosec</code> .	68
Tabela 9	– Hierarquia de subtipos de <code>CryptoError</code> .	69
Tabela 10	– Resumo geral dos testes executados.	74
Tabela 11	– Configuração dos ambientes de teste.	75
Tabela 12	– Distribuição dos testes Linux por domínio funcional.	76
Tabela 13	– Distribuição dos testes Android por arquivo de integração.	76
Tabela 14	– Resultados dos testes de isolamento (<i>stress</i>).	77
Tabela 15	– Paridade funcional Linux <i>vs.</i> Android por domínio criptográfico.	78
Tabela 16	– Resultados dos testes de <i>soak</i> (4 operações, 5 minutos cada).	79
Tabela 17	– Cobertura de código por camada arquitetural (<code>lcov</code>).	81
Tabela 18	– Latência das operações criptográficas, Linux x86_64, OpenSSL 4.0.0.	83
Tabela 19	– <i>Throughput</i> por primitiva, Linux x86_64, 1 MB (exceto onde indicado).	85
Tabela 20	– <i>Throughput</i> em operações por segundo, Linux x86_64, OpenSSL 4.0.0.	85
Tabela 21	– Comparação de desempenho Linux (x86_64) <i>vs.</i> Android (arm64-v8a).	86
Tabela 22	– <i>Ranking</i> de <i>cipher suites</i> simétricas, Linux x86_64, 1 MB.	89
Tabela 23	– Validações de aleatoriedade do CSPRNG.	89
Tabela 24	– Limitações conhecidas de segurança.	92
Tabela 25	– Comparação entre bibliotecas criptográficas para Dart/Flutter.	93
Tabela 26	– Latência da <code>dartcryptosec</code> (FFI) <i>vs.</i> OpenSSL em C (Linux x86_64).	93
Tabela 27	– Sumário dos resultados experimentais.	96
Tabela 28	– Cumprimento dos objetivos específicos da pesquisa.	103
Tabela 29	– Síntese quantitativa dos resultados experimentais.	104
Tabela 30	– Categorias STRIDE e propriedades de segurança violadas.	117
Tabela 31	– Matriz completa de 15 ameaças STRIDE com pontuação de risco.	118
Tabela 32	– Vetores relacionados à substituição da biblioteca nativa.	118
Tabela 33	– Vetores relacionados à exposição de material sensível na memória.	119
Tabela 34	– Vetores relacionados à corrupção de memória na fronteira FFI.	119
Tabela 35	– Vetores relacionados à exaustão de recursos.	119
Tabela 36	– Elementos da fronteira FFI e riscos associados.	120
Tabela 37	– Cobertura de contramedidas por categoria STRIDE.	120

Tabela 38 – Anotações de segurança do pipeline AES-256-GCM.	121
Tabela 39 – Matriz de cobertura funcional — API pública (métodos 1–25).	123
Tabela 40 – Matriz de cobertura funcional — métodos internos (26–36).	124
Tabela 41 – Completude por critério de avaliação (36 métodos $\times$ 13 critérios). . . .	124

LISTA DE ABREVIATURAS E SIGLAS

ABNT	Associação Brasileira de Normas Técnicas
AC	Autoridade Certificadora
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
CAdES	CMS Advanced Electronic Signatures
CMS	Cryptographic Message Syntax
CSR	Certificate Signing Request
DER	Distinguished Encoding Rules
DES	Data Encryption Standard
DIP	Dependency Inversion Principle
DSA	Digital Signature Algorithm
DSS	Digital Signature Standard
ECDSA	Elliptic Curve Digital Signature Algorithm
ETSI	European Telecommunications Standards Institute
FFI	Foreign Function Interface
FIPS	Federal Information Processing Standards
GPL	General Public License
HMAC	Hash-based Message Authentication Code
HTTPS	HyperText Transfer Protocol Secure
ICP	Infraestrutura de Chaves Públicas
ICP-Brasil	Infraestrutura de Chaves Públicas Brasileira
IP	Internet Protocol
ISP	Interface Segregation Principle
ITI	Instituto Nacional de Tecnologia da Informação
JCA	Java Cryptography Architecture
JCE	Java Cryptography Extension
JDK	Java Development Kit
KAT	Known Answer Test
KDF	Key Derivation Function
LCR	Listas de Certificados Revogados
LDAP	Lightweight Directory Access Protocol
LSP	Liskov Substitution Principle
MAC	Message Authentication Code
MIT	Massachusetts Institute of Technology

ML-DSA	Module-Lattice-based Digital Signature Algorithm
ML-KEM	Module-Lattice-based Key-Encapsulation Mechanism
NIST	National Institute of Standards and Technology
NSA	National Security Agency
OCP	Open/Closed Principle
OCSF	Online Certificate Status Protocol
PKCS	Public Key Cryptography Standards
PQC	Post-Quantum Cryptography
PRNG	Pseudo-Random Number Generator
RFC	Request for Comments
RSA	Rivest-Shamir-Adleman
SHA	Secure Hash Algorithm
SNTP	Simple Network Time Protocol
SRP	Single Responsibility Principle
URL	Uniform Resource Locator

SUMÁRIO

	Lista de abreviaturas e siglas	11
1	INTRODUÇÃO	17
1.1	CONTEXTUALIZAÇÃO	18
1.2	PROBLEMA FUNDAMENTAL	18
1.3	JUSTIFICATIVA	19
1.4	OBJETIVOS	20
1.4.1	Objetivo Geral	20
1.4.2	Objetivos Específicos	20
1.5	ESCOPO DO TRABALHO	21
1.6	ORGANIZAÇÃO DO TRABALHO	21
2	FUNDAMENTAÇÃO TEÓRICA	22
2.1	FUNÇÕES HASH CRIPTOGRÁFICAS	22
2.1.1	Códigos de Autenticação de Mensagens (MAC)	23
2.2	CRIPTOGRAFIA SIMÉTRICA	23
2.2.1	Modos de Operação de Cifras de Bloco	24
2.3	CRIPTOGRAFIA ASSIMÉTRICA	25
2.3.1	Assinaturas Digitais	26
2.4	INFRAESTRUTURA DE CHAVES PÚBLICAS (ICP)	29
2.4.1	Certificados Digitais	29
2.4.2	Estrutura de um Certificado X.509	29
2.4.3	Componentes da ICP	32
2.4.4	Gerenciamento de Certificados	34
2.4.5	Validação de Certificados	35
2.4.6	Mecanismos de Revogação de Certificados	36
2.4.7	Carimbos de Tempo	37
2.5	PADRÕES E PROTOCOLOS RELEVANTES	37
2.5.1	Extensões do padrão X.509	38
2.5.2	Normas	39
2.5.3	Formatos de Assinatura e ASN.1	40
2.6	ALGORITMOS PÓS-QUÂNTICOS	41
2.6.1	O Processo de Padronização do NIST	42
2.6.2	Fundamentos de Criptografia Baseada em Reticulados	43
2.6.3	ML-KEM: Mecanismo de Encapsulamento de Chaves	43
2.6.4	ML-DSA: Algoritmo de Assinatura Digital	44
2.7	FOREIGN FUNCTION INTERFACE (FFI)	46
2.8	OPENSSL E A API EVP	46
3	METODOLOGIA	48

3.1	CLASSIFICAÇÃO DA PESQUISA	48
3.2	ETAPAS DA PESQUISA	49
3.2.1	Revisão Bibliográfica e Análise do Estado da Arte	49
3.2.2	Análise Comparativa de Bibliotecas Existentes	49
3.2.3	Desenvolvimento Iterativo	49
3.2.4	Testes e Validação	49
3.3	FERRAMENTAS E AMBIENTE DE DESENVOLVIMENTO	50
3.4	MÉTRICAS E CRITÉRIOS DE AVALIAÇÃO	51
3.4.1	Cobertura de Código	51
3.4.2	Taxa de Sucesso dos Testes	51
3.4.3	Desempenho Computacional	51
4	IMPLEMENTAÇÃO	52
4.1	INTRODUÇÃO	52
4.2	BIBLIOTECAS CRIPTOGRÁFICAS DE REFERÊNCIA	52
4.2.1	Bouncy Castle	52
4.2.2	OpenSSL	53
4.3	BIBLIOTECAS CRIPTOGRÁFICAS EM DART	55
4.3.1	Pointycastle	55
4.3.2	Cryptography	56
4.3.3	Crypto	57
4.4	COMPARAÇÃO DE BIBLIOTECAS	58
4.5	<i>FOREIGN FUNCTION INTERFACE</i> (FFI)	59
4.5.1	Funcionamento e Arquitetura	59
4.5.2	Declaração de <i>Bindings</i> : Padrão Triplo	60
4.5.3	<i>Bindings</i> por Subsistema	61
4.5.4	Carregamento Multiplataforma	63
4.5.5	Gerenciamento de Memória Nativa	63
4.6	ARQUITETURA DA BIBLIOTECA PROPOSTA	63
4.6.1	Arquitetura Geral	63
4.6.2	Padrões de Projeto	64
4.6.3	Considerações de Segurança	67
4.6.4	Modelo de Ameaças STRIDE	67
4.6.5	Hierarquia de Erros Criptográficos	68
4.7	INTEGRAÇÃO DOS ALGORITMOS PÓS-QUÂNTICOS	70
4.7.1	Integração Arquitetural	70
4.7.2	ML-DSA e o <i>Digest</i> Nulo	70
4.8	DESAFIOS TÉCNICOS	71
4.8.1	Tratamento da <i>Tag</i> GCM	71
4.8.2	<i>Parsing</i> de ASN1_TIME	71

4.8.3	Pilhas de Certificados em Operações CMS	72
4.8.4	Compilação e Carregamento do Provedor FIPS	72
4.8.5	Limitações da Implementação e da Validação	73
5	RESULTADOS E VALIDAÇÃO	74
5.1	EXECUÇÃO E ANÁLISE DOS TESTES	74
5.1.1	Distribuição dos Testes Linux	75
5.1.2	Distribuição dos Testes Android	75
5.1.3	Testes de Isolamento	75
5.1.4	Paridade Funcional entre Plataformas	77
5.1.5	Testes de Longa Duração (<i>Soak</i>)	78
5.2	COBERTURA DE CÓDIGO	80
5.2.1	Cobertura por Camada	80
5.2.2	Análise da Cobertura por Camada	81
5.2.3	Limitações da Métrica de Cobertura	82
5.3	DESEMPENHO	82
5.3.1	Latência por Categoria de Operação	82
5.3.2	Análise de Variabilidade	84
5.3.3	Throughput Agregado	84
5.3.4	Aceleração por Hardware	86
5.3.5	Comparação Linux vs. Android	86
5.3.6	Simulação TLS 1.3	87
5.3.7	Throughput em Operações por Segundo	88
5.3.8	Ranking de <i>Cipher Suites</i> Simétricas	88
5.4	SEGURANÇA DA IMPLEMENTAÇÃO	89
5.4.1	Validações de Aleatoriedade	89
5.4.2	Unicidade de Chaves e Vetores de Inicialização	90
5.4.3	Curvas Seguras	90
5.4.4	Autenticação GCM	90
5.4.5	Teste NIST SP800-22	90
5.4.6	Testes Adversariais e de Memória	91
5.4.7	Limitações Conhecidas de Segurança	91
5.5	COMPARAÇÃO COM O ESTADO DA ARTE	91
5.5.1	Vantagens da <i>dartcryptosec</i>	94
5.5.2	Desvantagens e Limitações	95
5.5.3	Posicionamento	95
5.6	ANÁLISE DOS RESULTADOS EXPERIMENTAIS	95
5.6.1	Latência de FFI para Operações Simétricas	95
5.6.2	Overhead de FFI para Operações Assimétricas	96
5.6.3	Cobertura FFI e Detecção de Regressões em ICP	96

6	CONCLUSÃO	98
6.1	RETOMADA DOS OBJETIVOS	98
6.2	SÍNTESE DOS PRINCIPAIS RESULTADOS	100
6.3	CONTRIBUIÇÕES DO TRABALHO	101
6.4	LIMITAÇÕES E DIFICULDADES ENCONTRADAS	101
6.5	TRABALHOS FUTUROS	102
6.6	CUMPRIMENTO DOS OBJETIVOS	103
6.7	SÍNTESE QUANTITATIVA DOS RESULTADOS	104
	Referências	105
7	GLOSSÁRIO	111
	APÊNDICE A – REGISTROS DE DECISÃO ARQUITETU-	
	RAL (ADR)	115
A.1	ADR-001: MECANISMO DE INTEROPERABILIDADE — FFI . . .	115
A.2	ADR-002: ESCOLHA DO BACKEND CRIPTOGRÁFICO — OPENSSL 4.0.0	115
A.3	ADR-003: BINDINGS MANUAIS VS. AUTOMÁTICOS (FFIGEN) . .	115
A.4	ADR-004: ARQUITETURA EM 7 CAMADAS	116
A.5	ADR-005: PADRÃO SINGLETON PARA A FACADE	116
A.6	ADR-006: ORGANIZAÇÃO DOS CASOS DE USO NA CAMADA DE FLOWS	116
A.7	ADR-007: EMPACOTAMENTO DE BIBLIOTECAS NATIVAS . . .	116
	APÊNDICE B – MODELO DE AMEAÇAS DETALHADO .	117
B.1	MATRIZ COMPLETA DE AMEAÇAS (T1–T15)	117
B.2	ANÁLISE DOS VETORES DE AMEAÇA	118
B.2.1	Substituição da Biblioteca Nativa	118
B.2.2	Exposição de Material Sensível na Memória	119
B.2.3	Corrupção de Memória na Fronteira FFI	119
B.2.4	Exaustão de Recursos	119
B.3	MAPA DE SUPERFÍCIE DE ATAQUE	119
B.4	DFD: AES-256-GCM COM AAD	121
B.5	MATRIZ DE COBERTURA FUNCIONAL (36 × 13)	122
	APÊNDICE C – CÓDIGO-FONTE DA DARTCRYPTOSEC	126
C.1	ACESSO E UTILIZAÇÃO DO CÓDIGO	126
C.1.1	Uso básico e validação	127
	APÊNDICE D – ARTIGO NO FORMATO SBC	128

1 INTRODUÇÃO

No cenário contemporâneo do desenvolvimento de software, a segurança da informação tornou-se um componente indispensável, especialmente no contexto de aplicações distribuídas e móveis multiplataforma. A ampla adoção de dispositivos inteligentes, serviços digitais e aplicações conectadas expandiu consideravelmente a superfície de ataque dos sistemas. Aos desenvolvedores, impõem-se desafios crescentes na proteção de dados, na garantia da integridade das informações e na autenticação segura de usuários, serviços e dispositivos (STALLINGS, 2017).

A Infraestrutura de Chaves Públicas (ICP), equivalente ao termo em inglês *Public Key Infrastructure* (PKI), constitui um dos pilares fundamentais para assegurar autenticidade, integridade e confidencialidade nas comunicações digitais, por meio do uso de criptografia assimétrica, certificados digitais e cadeias de confiança (ADAMS; LLOYD, 2002). Entre seus principais elementos estão certificados X.509, autoridades certificadoras, mecanismos de validação de cadeia, listas de revogação e formatos padronizados de assinatura digital.

No ecossistema de desenvolvimento móvel, destaca-se o Flutter, *framework* de interface de usuário mantido pelo Google, amplamente utilizado para a construção de aplicações multiplataforma. Flutter utiliza a linguagem Dart para a implementação da lógica de negócio, integração com serviços externos e construção das camadas internas da aplicação.

Dart, entretanto, não se confunde com Flutter. Enquanto Flutter é um *framework* voltado à construção de interfaces, Dart é uma linguagem de programação independente, com compilação nativa para múltiplas plataformas, suporte a *Foreign Function Interface* (FFI) para integração com código C e um ecossistema de bibliotecas em expansão. Essas características tornam Dart uma alternativa relevante para o desenvolvimento de componentes de software voltados à segurança e à interoperabilidade criptográfica (TEAM, D., 2024b).

Apesar disso, o ecossistema Dart apresenta lacunas relevantes no suporte a recursos avançados de criptografia e ICP. Atualmente, não há uma biblioteca que ofereça suporte completo a operações como validação de cadeias de certificados, manipulação de certificados X.509, verificação de revogação, processamento de assinaturas digitais e uso de formatos padronizados como CMS e CAdES. O CMS (*Cryptographic Message Syntax*) define um contêiner interoperável para mensagens assinadas, cifradas ou autenticadas, enquanto o CAdES (*CMS Advanced Electronic Signatures*) estende esse formato com atributos voltados a assinaturas eletrônicas avançadas. Essa limitação dificulta a construção de aplicações Dart que dependam de conformidade criptográfica, validação documental ou integração com infraestruturas de certificação digital.

Diante desse problema, este trabalho apresenta a `dartcryptosec`¹, uma biblioteca desenvolvida com o objetivo de preencher parte dessa lacuna no ecossistema Dart, oferecendo suporte a operações relacionadas à Infraestrutura de Chaves Públicas, conforme detalhado na Seção 1.2.

1.1 CONTEXTUALIZAÇÃO

Durante o desenvolvimento do projeto *Medida Inteligente*, conduzido pelo Laboratório de Segurança em Computação (LabSEC) da UFSC em parceria com o Instituto Nacional de Metrologia, Qualidade e Tecnologia (Inmetro), identificou-se a ausência de bibliotecas criptográficas adequadas para operar com Infraestrutura de Chaves Públicas (ICP) no ambiente Dart (IDALINO *et al.*, 2022). Para contornar essa carência, tomou-se como inspiração a *libcryptosec*, um *wrapper* orientado a objetos para a biblioteca *OpenSSL libcrypto*. Essa escolha forneceu uma base sólida e uma direção clara para a implementação das funcionalidades criptográficas necessárias.

Com base no código da *libcryptosec*, foi possível iniciar e planejar o desenvolvimento de uma biblioteca para Dart que atendesse às necessidades específicas do projeto. A implementação inicial concentrou-se em resolver casos de uso diretamente relacionados ao *Medida Inteligente*, estabelecendo uma base funcional adaptada a essas demandas.

Esse histórico, fundamentado pela *libcryptosec* e pelo modelo de biblioteca criptográfica em que se baseia, serviu como alicerce para a estruturação do *PluginCrypto*, um plug-in criptográfico para Dart desenvolvido como prova de conceito durante o projeto Medida Inteligente. A experiência adquirida com o *PluginCrypto* e as dificuldades técnicas observadas no contexto do *Medida Inteligente* motivaram a expansão do escopo da biblioteca, contemplando recursos e cenários além dos casos de uso específicos do projeto originário, dando origem ao conceito do *dartcryptosec*.

1.2 PROBLEMA FUNDAMENTAL

O ecossistema Dart carece de bibliotecas que ofereçam suporte a funcionalidades de Infraestrutura de Chaves Públicas (ICP) em sua integralidade: APIs para manipulação de certificados X.509, verificação de assinaturas digitais, validação de cadeias de confiança e suporte a formatos de assinatura padronizados, como CMS e CAdES. Embora a biblioteca Pointycastle ofereça primitivas criptográficas básicas, como funções de *hash*, cifradores simétricos e geração de chaves RSA e ECDSA, trata-se de uma implementação voltada às operações de baixo nível, que não se estende à camada de infraestrutura. Permanecem descobertas, portanto, operações centrais como o parsing de certificados, a construção e validação de cadeias de confiança e os mecanismos de verificação de revogação.

¹ Repositório institucional do projeto: <https://codigos.ufsc.br/isac.martins/dartcryptosec>.

Essa lacuna impõe limitações concretas ao desenvolvimento de aplicações seguras em Dart. Os desenvolvedores veem-se forçados a manter múltiplas bases de código, recorrendo a bibliotecas e APIs específicas de cada plataforma, como o Bouncy Castle em Java/Kotlin para Android e a combinação de Common Crypto com o Security framework em iOS, o que eleva a complexidade de manutenção e o risco de inconsistências de comportamento e segurança entre plataformas (CASTLE, 2024; COOPER *et al.*, 2008; STANDARDS; TECHNOLOGY, 2001).

1.3 JUSTIFICATIVA

A crescente adoção de aplicações móveis multiplataforma amplia a necessidade de proteger dados e comunicações em diferentes sistemas operacionais e ambientes de execução (LLC, 2024). No ecossistema Dart, entretanto, ainda são limitadas as soluções que oferecem acesso integrado a operações criptográficas avançadas e aos padrões empregados pela Infraestrutura de Chaves Públicas Brasileira. Tal restrição dificulta o desenvolvimento de aplicações que dependem de certificação digital, assinaturas e validação de cadeias de confiança.

O trabalho também se justifica pelo desafio técnico de reaproveitar, a partir de Dart, uma biblioteca criptográfica madura escrita em C. O OpenSSL reúne décadas de implementação, auditoria e adoção em sistemas reais, mas sua API é nativa e foi projetada para programas que manipulam diretamente ponteiros, estruturas opacas e buffers de memória. Para expor essas rotinas a aplicações Dart, utiliza-se a *Foreign Function Interface* (FFI), mecanismo que permite carregar bibliotecas C e invocar suas funções sem reimplementar os algoritmos criptográficos. Essa arquitetura preserva o uso de primitivas consolidadas, mas exige cuidados com gerenciamento de memória, conversão de tipos, tratamento de erros e compatibilidade entre plataformas. A documentação sobre esse processo no contexto de Dart ainda é limitada, sobretudo quando aplicada a operações de ICP (TEAM, D., 2024c).

A integração com o OpenSSL possibilita o uso de operações criptográficas compatíveis com padrões internacionais e reduz a necessidade de implementar diretamente algoritmos sensíveis (OPENSSL PROJECT AUTHORS, 2026a; COOPER *et al.*, 2008; STANDARDS; TECHNOLOGY, 2001). Ao organizar essas funcionalidades em uma interface própria para Dart, a biblioteca busca reduzir a complexidade enfrentada pelos desenvolvedores e diminuir a possibilidade de erros durante o uso dos recursos criptográficos (FOUNDATION, 2024).

Assim, este estudo contribui para atender a uma necessidade do ecossistema Dart, favorecendo o desenvolvimento de aplicações multiplataforma que precisam lidar com proteção de dados, certificação digital e assinaturas digitais.

1.4 OBJETIVOS

1.4.1 Objetivo Geral

Desenvolver uma biblioteca criptográfica para Dart que forneça suporte abrangente a criptografia e a recursos de Infraestrutura de Chaves Públicas (ICP), suprimindo as deficiências identificadas no ecossistema atual. A biblioteca deverá oferecer ferramentas para verificação de assinaturas digitais, validação de cadeias de certificados, manipulação de certificados X.509 e tratamento dos formatos de assinatura CMS e CAdES, facilitando a implementação de práticas de segurança em aplicações multiplataforma desenvolvidas em Dart.

1.4.2 Objetivos Específicos

Para alcançar o objetivo geral, foram definidos os seguintes objetivos específicos:

- **Analisar o estado da arte das bibliotecas criptográficas em Dart:** Realizar um estudo detalhado das bibliotecas existentes, como *PointyCastle*, *Dart Cryptography* e outras, identificando suas funcionalidades, limitações e lacunas, especialmente no contexto de ICP e gerenciamento de certificados (CONTRIBUTORS, 2024; TEAM, D. C., 2024).
- **Projetar uma arquitetura modular para integração criptográfica nativa:** Definir uma arquitetura em camadas que organize os componentes criptográficos, com comunicação direta entre Dart e a biblioteca OpenSSL por meio de *Foreign Function Interface* (FFI), eliminando a necessidade de código intermediário e permitindo expansões futuras.
- **Assegurar compatibilidade multiplataforma:** Garantir que a biblioteca funcione em pelo menos duas plataformas distintas, como Linux (x86_64) e Android (arm64-v8a, armeabi-v7a, x86_64), com infraestrutura de build preparada para iOS (podspec, FFI path, build script), sem testes funcionais executados, e suporte básico de compilação para Windows, permitindo seu uso em diferentes tipos de aplicações móveis e de desktop (LLC, 2024).
- **Implementar funcionalidades essenciais de ICP:** Desenvolver componentes para geração e verificação de assinaturas digitais, validação de cadeias de certificados e manipulação de certificados X.509, incluindo criação, análise e gerenciamento de extensões (COOPER *et al.*, 2008).
- **Implementar mecanismos de verificação de revogação de certificados:** Incorporar suporte a Listas de Revogação de Certificados (CRLs) e ao Protocolo de Status de Certificado Online (OCSP) (COOPER *et al.*, 2008; SANTESSON *et al.*, 2013).

- **Implementar manipulação de Certificate Signing Requests (CSRs):** Desenvolver funcionalidades para criação e gerenciamento de CSRs (NYSTROM; KALISKI, 2000).
- **Suporte a primitivas criptográficas:** Incluir implementações de primitivas criptográficas fundamentais, como cifradores simétricos e funções de *hash*.
- **Desenvolver documentação abrangente:** Produzir documentação detalhada e exemplos de uso que facilitem a adoção da biblioteca pela comunidade de desenvolvedores Dart.
- **Integrar a biblioteca a um *backend* criptográfico consolidado:** Utilizar a *Foreign Function Interface* (FFI) para integrar a biblioteca Dart com o OpenSSL, garantindo conformidade com padrões internacionais (TEAM, D., 2024c; OPENSSL PROJECT AUTHORS, 2026a).

1.5 ESCOPO DO TRABALHO

Este trabalho abrange o projeto, a implementação e a avaliação de uma biblioteca Dart que acessa recursos criptográficos do OpenSSL por meio de FFI. O recorte concentra-se na arquitetura dessa integração e em sua aplicação a operações de ICP, sem pretender oferecer cobertura integral da API do OpenSSL.

A avaliação limita-se ao funcionamento da biblioteca nos ambientes Linux e Android considerados durante o desenvolvimento. A preparação de arquivos de compilação para outras plataformas não caracteriza suporte validado. Também não fazem parte do estudo a operação de serviços externos de certificação, a auditoria formal de segurança da implementação nativa e a análise de resistência a canais laterais.

1.6 ORGANIZAÇÃO DO TRABALHO

Este documento está organizado em seis capítulos. O Capítulo 2 apresenta os fundamentos de criptografia, ICP, formatos de assinatura, algoritmos pós-quânticos, FFI e OpenSSL. O Capítulo 3 descreve a classificação da pesquisa, as etapas executadas, o ambiente utilizado e os critérios de avaliação. O Capítulo 4 detalha a análise das bibliotecas existentes, a arquitetura da `dartcryptosec`, a estratégia de integração via FFI e os desafios técnicos encontrados. O Capítulo 5 apresenta os resultados de testes, cobertura, desempenho e segurança. Por fim, o Capítulo 6 retoma os objetivos, consolida as contribuições e indica limitações e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

A criptografia é um dos pilares da segurança digital, dedicada à proteção de dados por meio de técnicas que garantem confidencialidade, integridade e autenticidade (STALLINGS, 2017). Ela é tradicionalmente dividida em criptografia simétrica e assimétrica, cada uma com características e aplicações específicas. Este capítulo apresenta os fundamentos teóricos necessários à compreensão do trabalho: inicialmente, são abordadas as funções *hash* e as duas modalidades de criptografia; em seguida, discutem-se assinaturas digitais, certificados, infraestrutura de chaves públicas e os principais padrões e formatos que garantem a interoperabilidade entre sistemas criptográficos.

2.1 FUNÇÕES HASH CRIPTOGRÁFICAS

Funções *hash* criptográficas transformam dados de tamanho arbitrário em valores de comprimento fixo, chamados de resumos ou *hashes* (MENEZES, A. J.; OORSCHOT, P. C. van; VANSTONE, S. A., 2018). Denota-se por h uma função *hash* criptográfica. Para que h seja considerada criptograficamente segura, ela deve possuir três propriedades:

- **Resistência à Pré-imagem:** Dado um *hash* y , é computacionalmente inviável encontrar uma mensagem x tal que $h(x) = y$.
- **Resistência à Segunda Pré-imagem:** Para uma mensagem x específica, é difícil encontrar outra mensagem $x' \neq x$ que produza o mesmo *hash*, ou seja, $h(x') = h(x)$.
- **Resistência a Colisões:** É difícil encontrar quaisquer duas mensagens distintas x e x' que resultem no mesmo *hash*, $h(x) = h(x')$.

Algoritmos que atendem a esses critérios incluem SHA-256 e SHA-3, padronizados pelo NIST (DANG; ZHAO, 2015). Em contrapartida, algoritmos como o MD5 e o SHA-1 já não são considerados seguros, pois pesquisadores demonstraram ataques práticos de colisão que permitem gerar dois documentos distintos com o mesmo resumo criptográfico (STALLINGS, 2017). Por essa razão, normas atuais recomendam o uso de funções da família SHA-2 ou SHA-3 para novas aplicações.

Espera-se ainda que tais funções sejam computacionalmente eficientes, capazes de processar rapidamente grandes volumes de dados sem comprometer a segurança. A resistência a ataques como o do aniversário reforça a necessidade de resumos de comprimento adequado. Esse ataque explora o fato probabilístico de que, para uma função com saída de n bits, espera-se encontrar uma colisão após aproximadamente $2^{n/2}$ tentativas, e não 2^n . Por isso, um resumo de 256 bits oferece cerca de 128 bits de segurança contra colisões (STALLINGS, 2017). Essas propriedades sustentam a integridade das aplicações que utilizam funções *hash*, como assinaturas digitais, autenticação de mensagens, sistemas de login e verificação de integridade de dados. No contexto das assinaturas digitais, o

papel do *hash* é retomado na Seção 2.3.1, pois a combinação entre função de resumo e algoritmo de assinatura depende do esquema utilizado.

2.1.1 Códigos de Autenticação de Mensagens (MAC)

Um Código de Autenticação de Mensagens (MAC, *Message Authentication Code*) é um mecanismo simétrico que utiliza uma chave secreta compartilhada para verificar a integridade e a origem de uma mensagem. Diferentemente de um *hash* sem chave, o valor de autenticação somente pode ser reproduzido por quem possui o segredo compartilhado. O HMAC (*Hash-based Message Authentication Code*) é uma construção específica de MAC baseada em funções *hash* criptográficas (KRAWCZYK; BELLARE; CANETTI, 1997).

Formalmente, o cálculo do HMAC pode ser representado pela seguinte equação:

$$\text{HMAC}(K, m) = H((K' \oplus \text{opad}) \parallel H((K' \oplus \text{ipad}) \parallel m)) \quad (1)$$

em que K' representa a chave ajustada ao tamanho do bloco da função *hash*, m representa a mensagem, *ipad* e *opad* são constantes de preenchimento interno e externo, respectivamente, e $\parallel$ denota concatenação.

Essa construção é utilizada em protocolos de autenticação e na proteção de mensagens entre partes que já compartilham uma chave. Ela também ajuda a distinguir autenticação simétrica, baseada em segredo compartilhado, de assinaturas digitais, baseadas em pares de chaves assimétricas.

2.2 CRIPTOGRAFIA SIMÉTRICA

A criptografia simétrica emprega uma única chave tanto para criptografar quanto para descriptografar informações (STINSON, 2006). A segurança do esquema depende de que essa chave permaneça secreta entre as partes comunicantes (MENEZES, A. J.; OORSCHOT, P. C. van; VANSTONE, S. A., 2018). Por essa razão, o modelo simétrico mostra-se particularmente eficiente em cenários nos quais as partes podem estabelecer um segredo compartilhado previamente, como na proteção de dados em repouso e na comunicação entre servidores de um mesmo domínio administrativo.

Definição 1 (Criptografia Simétrica). *Criptografia simétrica é um esquema composto por um par de funções (E, D) e um conjunto de chaves $\mathcal{K}$. A função E é uma função de cifragem que utiliza a chave k , e D é uma função de decifragem que utiliza a mesma chave k . Para qualquer chave $k \in \mathcal{K}$ e mensagem m , a relação abaixo deve ser válida:*

$$D_k(E_k(m)) = m. \quad (2)$$

Do ponto de vista computacional, esse tipo de criptografia é eficiente e consome menos recursos que operações assimétricas típicas, o que a torna adequada para comunicações

de alto volume e proteção de dados armazenados (MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A., 1996). Contudo, sua segurança depende da distribuição segura da chave compartilhada. Esse problema é normalmente resolvido por protocolos de troca de chaves e por Infraestruturas de Chave Pública, que empregam criptografia assimétrica e serão discutidos na Seção 2.3.

2.2.1 Modos de Operação de Cifras de Bloco

O *Advanced Encryption Standard* (AES), padronizado pelo NIST como FIPS 197, é o cifrador simétrico de bloco de referência na criptografia simétrica, tendo substituído o *Data Encryption Standard* (DES) por oferecer segurança superior e melhor desempenho (DAEMEN; RIJMEN, 2002; STANDARDS; TECHNOLOGY, 2001). O AES opera com blocos de 128 bits e suporta chaves de 128, 192 e 256 bits. Na prática, porém, mensagens reais raramente possuem exatamente esse tamanho. Por isso, os **modos de operação** definem como uma cifra de bloco deve ser aplicada repetidamente para processar mensagens de comprimento arbitrário, estabelecendo como cada bloco de texto claro se relaciona com os demais e com o processo de cifragem (STALLINGS, 2017).

O modo mais simples, **ECB (Electronic Codebook)**, cifra cada bloco independentemente com a mesma chave. Essa independência, no entanto, constitui sua principal fragilidade: blocos idênticos de texto claro produzem blocos idênticos de texto cifrado, revelando padrões da mensagem original. Por essa razão, o ECB é considerado inadequado para a maioria das aplicações criptográficas reais (STALLINGS, 2017).

Para superar essa limitação, modos como **CBC (Cipher Block Chaining)** introduzem um vetor de inicialização (*IV*) e encadeiam cada bloco cifrado com o bloco de texto claro seguinte por meio de uma operação XOR, garantindo que mensagens idênticas produzam textos cifrados distintos (MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A., 1996). O **CTR (Counter)** transforma a cifra de bloco em um gerador de fluxo de chave, cifrando valores sucessivos de um contador e combinando o resultado com o texto claro via XOR, o que permite acesso aleatório e paralelização eficiente.

Por fim, modos como **GCM (Galois/Counter Mode)** combinam a estrutura do CTR com um código de autenticação de mensagem baseado em aritmética de campos de Galois, provendo simultaneamente confidencialidade e autenticidade, propriedade conhecida como *Authenticated Encryption with Associated Data* (AEAD) (MENEZES, A. J.; OORSCHOT, P. C. van; VANSTONE, S. A., 2018; STANDARDS; TECHNOLOGY, 2001). O GCM é recomendado por normas atuais como NIST SP 800-38D devido à sua eficiência e às garantias combinadas de sigilo e integridade que oferece em uma única operação. No contexto deste trabalho, os modos CBC e GCM são particularmente relevantes: o CBC é amplamente utilizado para cifragem de envelopes CMS, enquanto o GCM, por reunir confidencialidade e autenticidade em uma única operação (AEAD), é o modo recomendado

para novas aplicações (STANDARDS; TECHNOLOGY, 2001).

2.3 CRIPTOGRAFIA ASSIMÉTRICA

A criptografia assimétrica fundamenta-se em um par de chaves matematicamente relacionadas: a **chave pública** e a **chave privada**. A chave pública pode ser livremente divulgada, enquanto a chave privada deve permanecer sob confidencialidade do emissor (MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A., 1996). A segurança desse sistema decorre da dificuldade computacional associada à inversão do processo de geração das chaves, propriedade formalizada pelo conceito de **função unidirecional**: uma função fácil de computar, mas para a qual não existe algoritmo eficiente conhecido que compute sua inversa (MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A., 1996). Embora seja simples derivar a chave pública a partir da chave privada, o caminho inverso é considerado inviável na prática.

Definição 2 (Criptografia Assimétrica). *Criptografia assimétrica é um esquema no qual as operações criptográficas utilizam um par de chaves (pk, sk) , em que pk representa a chave pública (public key) e sk representa a chave privada ou secreta (secret key). A chave pública pode ser divulgada, enquanto a chave privada deve permanecer sob controle exclusivo de seu titular. Em um esquema de cifração assimétrica, para uma mensagem m , uma função de cifração E e uma função de decifração D , deve-se satisfazer:*

$$D_{sk}(E_{pk}(m)) = m. \quad (3)$$

Nem todo esquema assimétrico, contudo, serve simultaneamente para cifração e assinatura. RSA possui variantes para ambos os usos, mas ECDSA é um algoritmo de assinatura, e ML-KEM é um mecanismo de encapsulamento de chaves, não uma cifra genérica. Tampouco todo esquema segue o modelo de derivação direta entre chaves: no ECDSA, por exemplo, ambas as chaves são geradas a partir de parâmetros de curva, e a relação entre elas é mediada pelo Problema do Logaritmo Discreto, sem que uma seja diretamente derivável da outra por uma função simples (JOHNSON; MENEZES, A.; VANSTONE, S., 2001).

Desse modelo decorrem duas funcionalidades essenciais: (i) criptografia de mensagens, assegurando a confidencialidade, e (ii) verificação de assinaturas digitais, garantindo autenticidade, integridade e não repúdio. Seja pk a chave pública, sk a chave privada, m a mensagem em claro (*plaintext*) e c o texto cifrado (*ciphertext*):

$$E(pk, m) = c \quad \text{e} \quad D(sk, c) = m, \quad (4)$$

assegurando a propriedade de correção:

$$D(sk, E(pk, m)) = m. \quad (5)$$

Na prática, as duas modalidades costumam ser combinadas, resultando na chamada **criptografia híbrida**. Nessa abordagem, a criptografia assimétrica protege a chave simétrica de sessão, enquanto a simétrica cifra os grandes volumes de dados, unindo a facilidade de distribuição de chaves de uma ao desempenho superior da outra (STALLINGS, 2017). Esse modelo híbrido é adotado por praticamente todos os protocolos de segurança modernos, como TLS, SSH e PGP, constituindo a estratégia predominante para conciliar eficiência computacional com gerenciamento prático de chaves em sistemas reais.

Para resolver o problema da distribuição de chaves simétricas em redes abertas, são utilizados protocolos de troca de chaves baseados em criptografia assimétrica, como o Diffie-Hellman, que permite que duas partes estabeleçam um segredo compartilhado sem precisar de um canal prévio seguro (DIFFIE; HELLMAN, 1976). Infraestruturas de Chave Pública (ICP) complementam essa abordagem ao gerenciar e distribuir chaves públicas de forma autenticada, oferecendo mecanismos de verificação de identidade e revogação (ADAMS; LLOYD, 2002). A combinação entre protocolos de troca de chaves e ICP possibilita que sistemas criptográficos simétricos operem com segurança mesmo em redes distribuídas e abertas.

2.3.1 Assinaturas Digitais

As assinaturas digitais proveem três propriedades fundamentais para a segurança das comunicações eletrônicas. A **integridade** assegura que o conteúdo não foi alterado durante a transmissão ou o armazenamento, pois qualquer modificação na mensagem original produz um resumo criptográfico distinto (STALLINGS, 2017). A **autenticidade** permite verificar a origem da mensagem, confirmando a identidade do emissor por meio de sua chave pública (MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A., 1996). O **não repúdio** fornece evidência técnica de autoria quando a chave privada permanece sob controle do signatário e quando o certificado correspondente pode ser associado a uma identidade. Essa propriedade depende também de fatores externos ao algoritmo, como proteção da chave, políticas de certificação e validade jurídica do certificado.

Definição 3 (Assinatura Digital). *Assinatura digital é um esquema criptográfico composto por três algoritmos: **KeyGen**, que gera um par de chaves (pk, sk) ; **Sign**, que utiliza sk para produzir uma assinatura σ sobre uma mensagem m ; e **Verify**, que utiliza pk para verificar se σ é válida para m . A função de verificação retorna **true** quando a assinatura é aceita e **false** quando é rejeitada. Para qualquer par de chaves válido e mensagem m , deve-se satisfazer:*

$$\text{Verify}_{pk}(\text{Sign}_{sk}(m), m) = \text{true}. \quad (6)$$

O processo prático de assinatura de uma mensagem m frequentemente envolve o cálculo do resumo criptográfico $H(m)$ e a aplicação da função de assinatura com a chave

privada do remetente, gerando a assinatura σ :

$$\sigma = \text{Sign}_{sk}(H(m)). \quad (7)$$

O destinatário, ao receber o par (m, σ) , recalcula $H(m)$ e verifica a assinatura com a chave pública do emissor:

$$\text{Verify}_{pk}(\sigma, H(m)) = \text{true}. \quad (8)$$

Se a verificação for bem-sucedida, confirmam-se simultaneamente a integridade da mensagem e a autenticidade criptográfica do emissor. Alguns esquemas, como Ed25519 e Ed448, podem receber a mensagem completa e realizar internamente o processamento necessário; por isso, $H(m)$ deve ser entendido como uma representação conceitual do material assinado, não como uma etapa obrigatoriamente externa em todos os algoritmos.

Diversos esquemas criptográficos implementam os conceitos de criptografia assimétrica e assinaturas digitais. Entre os mais relevantes para este trabalho estão o RSA e os algoritmos baseados em Curvas Elípticas (ECC), descritos a seguir (RIVEST; SHAMIR; ADLEMAN, 1978; MILLER, 1985).

RSA: O RSA (Rivest-Shamir-Adleman) é um dos esquemas de criptografia assimétrica mais estabelecidos e adotados em protocolos como TLS, SSH e S/MIME. Sua segurança baseia-se na dificuldade de fatorar grandes números inteiros que são produtos de dois primos, um problema para o qual não se conhecem algoritmos clássicos eficientes quando os fatores são suficientemente grandes (RIVEST; SHAMIR; ADLEMAN, 1978). Em termos gerais, o algoritmo escolhe dois primos grandes p e q , calcula $n = pq$, define expoentes e e d relacionados pelo módulo $\varphi(n)$, publica (n, e) e mantém d em segredo. Sua presença estende-se do TLS (Transport Layer Security), que protege a comunicação na internet, aos sistemas de assinatura digital, nos quais garante a autenticidade e a integridade das mensagens (STALLINGS, 2017).

Em uma representação simplificada da assinatura RSA, o signatário calcula um resumo da mensagem e aplica o expoente privado, obtendo $s = H(m)^d \bmod n$. O verificador aplica o expoente público e compara $s^e \bmod n$ com o valor codificado de $H(m)$. Implementações reais não assinam o resumo diretamente: utilizam esquemas de codificação seguros, como RSA-PSS ou RSASSA-PKCS1-v1_5, definidos na RFC 8017 (MORIARTY *et al.*, 2016).

Criptografia de Curvas Elípticas (ECC): A Criptografia de Curvas Elípticas (ECC, do inglês *Elliptic Curve Cryptography*) constrói esquemas assimétricos a partir das propriedades matemáticas das curvas elípticas sobre campos finitos. Diferentemente de sistemas como o RSA, a ECC proporciona um nível de segurança equivalente com tamanhos de chave significativamente menores, o que resulta em maior eficiência computacional e menor consumo de recursos (MILLER, 1985).

A segurança da ECC baseia-se na dificuldade do Problema do Logaritmo Discreto em Curvas Elípticas (ECDLP, *Elliptic Curve Discrete Logarithm Problem*), que consiste

em determinar um inteiro k tal que $Q = kP$, onde P e Q são pontos na curva elíptica. Este problema é considerado intratável para curvas seguras (MILLER, 1985). Devido à sua eficiência, a ECC mostra-se especialmente adequada a dispositivos com recursos limitados, como smartphones, dispositivos IoT (Internet das Coisas) e sistemas embarcados, onde o desempenho e a economia de energia são críticos (GUPTA; PRAKASH, 2024).

ECDSA: O ECDSA (Elliptic Curve Digital Signature Algorithm) é uma variante do DSA (Digital Signature Algorithm) que utiliza curvas elípticas para gerar assinaturas digitais. Comparado ao RSA, o ECDSA oferece o mesmo nível de segurança com chaves consideravelmente menores. Por exemplo, uma chave ECDSA de 256 bits proporciona segurança equivalente a uma chave RSA de 3072 bits (SANTOS; OLIVEIRA, 2017). Essa redução no tamanho das chaves resulta em assinaturas digitais menores, o que é vantajoso para armazenamento e transmissão, além de melhorar o desempenho em termos de velocidade de geração e verificação de assinaturas (SHOUP, 2005).

A eficiência do ECDSA o torna ideal para cenários com restrições computacionais, como dispositivos móveis e sistemas de IoT, onde recursos de processamento e memória são limitados. Ademais, a adoção do ECDSA tem crescido significativamente em protocolos modernos, como o TLS 1.3, que prioriza algoritmos de curvas elípticas para o estabelecimento seguro de conexões (RESCORLA, 2018).

Comparação entre RSA e ECDSA: Enquanto o RSA com 3072 bits é considerado seguro para os padrões atuais, o ECDSA atinge segurança equivalente com chaves de apenas 256 bits, o que reduz de forma expressiva o tamanho das assinaturas e melhora o desempenho, exigindo menos recursos de armazenamento e processamento. O RSA predomina em sistemas legados e em protocolos consolidados como TLS e S/MIME devido à sua adoção histórica, ao passo que o ECDSA destaca-se por sua eficiência e menor tamanho de chave, sendo ideal para dispositivos com recursos limitados.

Além do RSA e dos esquemas baseados em curvas elípticas, outro algoritmo historicamente relevante é o ElGamal, cuja segurança também se fundamenta no Problema do Logaritmo Discreto (MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A., 1996). Ele é citado aqui apenas como referência conceitual para a evolução dos esquemas assimétricos; a implementação analisada neste trabalho concentra-se em RSA, ECDSA e algoritmos pós-quânticos.

Apesar de sua consolidação, os esquemas assimétricos apresentados têm sua segurança ameaçada pelo advento da computação quântica. O algoritmo de Shor resolve eficientemente tanto o problema da fatoração de inteiros, fundamento do RSA, quanto o do logaritmo discreto, base da ECC e do ElGamal, o que tornaria esses esquemas inseguros em um cenário de computadores quânticos de larga escala (SHOR, 1994). Essa limitação motiva o desenvolvimento de alternativas resistentes a ataques quânticos, abordadas na Seção 2.6.

Um desafio fundamental da criptografia assimétrica é garantir que uma chave

pública realmente pertence à entidade que afirma possuí-la. Qualquer parte pode gerar um par de chaves e divulgar a chave pública, mas sem um mecanismo de verificação, um adversário poderia substituir a chave pública legítima pela sua própria, conduzindo ataques de interceptação (*man-in-the-middle*) (MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A., 1996). A solução consiste em vincular formalmente a chave pública a uma identidade por meio de um documento assinado por uma entidade confiável: o **certificado digital**. Esse mecanismo, juntamente com a infraestrutura necessária para emitir, distribuir e revogar certificados, é abordado na seção a seguir.

2.4 INFRAESTRUTURA DE CHAVES PÚBLICAS (ICP)

A Infraestrutura de Chaves Públicas (ICP) sustenta a segurança das comunicações digitais ao reunir, de forma integrada, tecnologias, políticas e procedimentos destinados a gerenciar com segurança as chaves criptográficas e os certificados digitais (ADAMS; LLOYD, 2002). Nela se estabelece um ambiente confiável para a emissão, distribuição, gerenciamento, revogação e validação de certificados digitais, permitindo a autenticação, integridade e confidencialidade nas interações eletrônicas. Essa infraestrutura associa chaves públicas a identidades por meio de certificados digitais emitidos por Autoridades Certificadoras (ACs) confiáveis, formando a base para a segurança em diversos contextos digitais (ADAMS; LLOYD, 2002).

2.4.1 Certificados Digitais

Os certificados digitais são documentos eletrônicos que vinculam uma chave pública a uma entidade específica, seja uma pessoa, organização ou dispositivo (ADAMS; LLOYD, 2002). Esses certificados asseguram a autenticidade da chave pública, permitindo que terceiros confiem na identidade da entidade titular do certificado. O padrão predominante para a criação e gerenciamento de certificados digitais é o X.509 (UNION, 2016), que define a estrutura, os campos obrigatórios e opcionais, bem como as extensões dos certificados.

2.4.2 Estrutura de um Certificado X.509

O padrão X.509 define com precisão a estrutura do certificado, de modo a garantir sua funcionalidade e segurança. Cada certificado reúne diversos campos, descritos a seguir:

- **Versão:** Indica a versão do padrão X.509 utilizada, sendo as mais comuns v1, v2 e v3. Cada versão adiciona ou modifica campos para atender a requisitos de segurança e funcionalidade.
- **Número de Série:** Um identificador único emitido pela AC para cada certificado. Este número é essencial para o rastreamento e revogação do certificado, garantindo que cada certificado possa ser individualmente identificado.

- **Algoritmo de Assinatura:** Especifica o método criptográfico utilizado para assinar o certificado, assegurando sua integridade e autenticidade. Algoritmos comuns incluem RSA, ECDSA e SHA-2.
- **Emissor:** Contém a identificação da AC que emitiu o certificado, incluindo seu nome e outras informações relevantes. Este campo estabelece a origem de confiança do certificado.
- **Validade:** Define o período durante o qual o certificado é considerado válido, especificando uma data de início e uma data de expiração. A validade temporal assegura que certificados desatualizados ou comprometidos não sejam utilizados indevidamente.
- **Sujeito:** Identifica a entidade titular do certificado, podendo incluir detalhes como nome, organização e endereço. Este campo estabelece a relação entre a chave pública e a identidade da entidade.
- **Chave Pública do Sujeito:** Contém a chave pública associada à entidade titular, utilizada para operações de criptografia e verificação de assinaturas digitais.
- **Extensões:** Campos adicionais da versão X.509 v3 que restringem o uso da chave, identificam nomes alternativos e fornecem informações necessárias à construção e à validação da cadeia. As principais extensões são detalhadas na Seção 2.5.1.

Essa estrutura permite que os certificados X.509 sejam utilizados em uma ampla variedade de aplicações, desde a segurança de websites até a assinatura de documentos eletrônicos e códigos, além da autenticação de e-mails, proporcionando flexibilidade e segurança adequadas às necessidades modernas. A adoção universal do padrão X.509 deve-se à sua capacidade de representar diferentes políticas de segurança e níveis de garantia por meio de extensões padronizadas, permitindo que cada domínio de aplicação selecione as extensões pertinentes ao seu cenário. A interoperabilidade oferecida pelo X.509 constitui, ainda, a base para a confiança em escala global na Internet: é sobre ela que protocolos como TLS, S/MIME e IPsec constroem suas garantias de segurança.

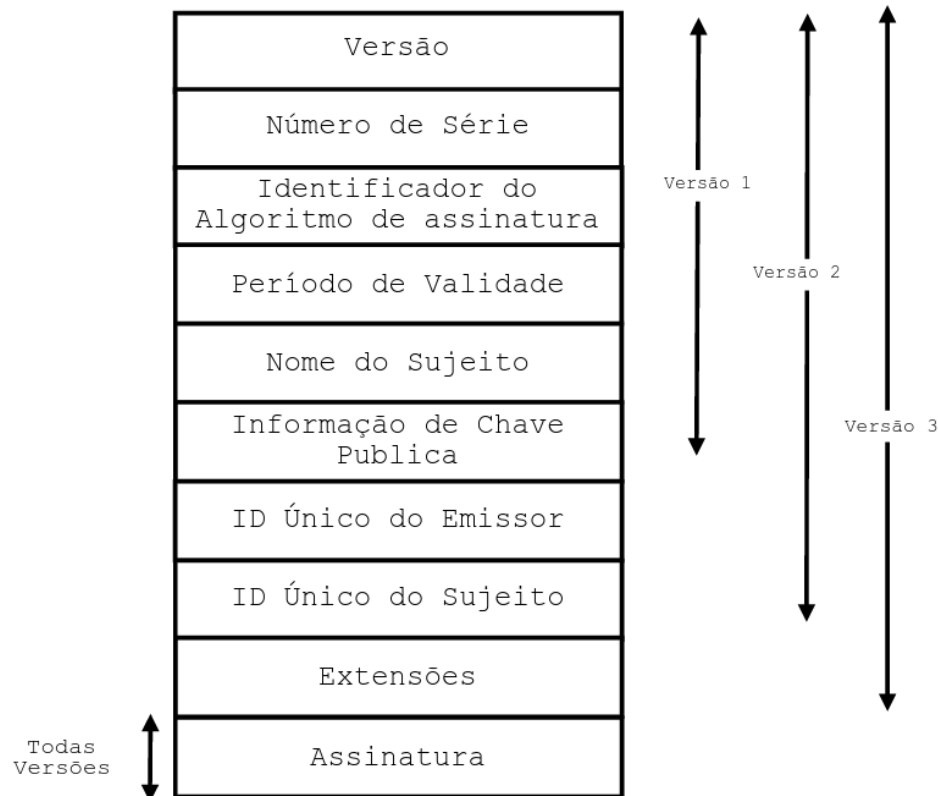


Figura 1 – Exemplo da estrutura de um certificado digital X.509 v3.

Fonte: Adaptado de (AL-JANABI; HAMAD, 2019).

Os certificados digitais apresentam variações significativas conforme o grau de verificação realizado e os objetivos específicos de aplicação. Entre os tipos mais comuns, destacam-se:

- **Certificados de Validação de Domínio (DV):** Validam exclusivamente a posse do domínio, sendo de obtenção rápida e simples. São adequados para aplicações que requerem uma verificação básica, como a criação de sites pessoais ou blogs.
- **Certificados de Validação de Organização (OV):** Além da validação do domínio, verificam a existência legal da organização. Este nível adicional de verificação proporciona maior confiança aos usuários, sendo frequentemente utilizados por empresas e entidades que buscam demonstrar sua legitimidade.
- **Certificados de Validação Estendida (EV):** Representam o nível mais elevado de verificação, exigindo uma rigorosa validação da organização. Os certificados EV aumentam a confiança dos usuários e são comumente empregados em sites que demandam alta segurança, como instituições financeiras e plataformas de comércio eletrônico.
- **Certificados de Assinatura de Código:** Utilizados para assinar digitalmente softwares, asseguram a integridade e autenticidade do código, protegendo con-

tra a distribuição de software malicioso e garantindo que o software não foi adulterado.

- **Certificados de Cliente:** Destinados a identificar usuários individuais, permitem autenticação segura em sistemas e serviços. Fortalecem o controle de acesso em ambientes corporativos e governamentais.

Cada tipo de certificado atende a necessidades específicas, oferecendo diferentes níveis de segurança e confiabilidade conforme o contexto de uso. Os certificados DV, por exemplo, são adequados para cenários que demandam implantação rápida e custo reduzido, enquanto os certificados EV são reservados para aplicações em que a confiança do usuário final é crítica, como nas transações bancárias e no comércio eletrônico. A escolha do tipo de certificado deve, portanto, ser orientada pelo perfil de risco da aplicação e pelos requisitos regulatórios do setor em que será empregado.

2.4.3 Componentes da ICP

Múltiplos elementos compõem a ICP e, em conjunto, asseguram a segurança do ambiente digital. Esses componentes formam uma **Cadeia de Confiança**, na qual a credibilidade de um certificado final de usuário deriva da confiança em uma Autoridade Certificadora Raiz. As raízes de confiança podem delegar a emissão de certificados a **Autoridades Certificadoras Intermediárias**, que operam sob a autoridade da AC raiz. Cada certificado é digitalmente assinado pela autoridade certificadora que o emitiu, garantindo sua autenticidade e integridade. Assim, um certificado final é reconhecido como confiável porque sua cadeia de assinaturas remonta a uma autoridade raiz em quem todos confiam.

O uso de ACs intermediárias é necessário por razões operacionais e de segurança. A chave privada de uma AC raiz deve permanecer altamente protegida e ser utilizada raramente; por isso, a emissão cotidiana de certificados costuma ser delegada a autoridades intermediárias. Essa separação reduz o impacto de comprometimentos, permite aplicar políticas específicas por domínio de uso e facilita a substituição de uma autoridade intermediária sem trocar a raiz de confiança instalada nos clientes.

Os principais componentes da ICP são:

- **Autoridades Certificadoras (ACs):** São entidades confiáveis responsáveis por emitir, renovar e revogar certificados digitais, além de assinar digitalmente cada certificado emitido para garantir sua autenticidade. As ACs também publicam as Listas de Certificados Revogados (LCRs) e disponibilizam serviços de verificação de status, como o OCSP, assegurando que as partes confiem nas chaves públicas associadas às identidades certificadas (ADAMS; LLOYD, 2002).
 - **Autoridades Certificadoras Raiz:** Emitem certificados para si próprias e para as ACs intermediárias. São a base da cadeia de confiança.

- **Autoridades Certificadoras Intermediárias:** Emitem certificados para entidades finais, operando sob a autoridade das ACs raiz. Todas as emissões de certificados são acompanhadas de assinaturas digitais, que asseguram a validade e a origem dos certificados emitidos.
- **Autoridades de Registro (ARs):** Entidades intermediárias responsáveis por verificar a identidade e as credenciais das entidades solicitantes antes que uma AC emita um certificado. O processo típico de registro envolve a validação de documentos, a confirmação de posse de domínio e, em níveis mais elevados de garantia, a verificação presencial da organização solicitante (COOPER *et al.*, 2008). Após a validação bem-sucedida, a AR encaminha a solicitação à AC para emissão do certificado, separando assim as funções de verificação de identidade das funções de emissão criptográfica.
- **Repositórios de Certificados:** Infraestruturas onde os certificados digitais e as Listas de Revogação de Certificados (LCRs) são armazenados e disponibilizados para consulta pública, permitindo que usuários e sistemas verifiquem a validade dos certificados.
- **Políticas de Certificação:** Conjunto de regras e diretrizes que orientam os procedimentos para emissão, gerenciamento e revogação de certificados, assegurando conformidade e segurança em todas as etapas do ciclo de vida dos certificados.
- **Mecanismos de Validação:** Ferramentas e protocolos utilizados para verificar a autenticidade e a validade dos certificados digitais durante as transações, garantindo que apenas certificados legítimos e não revogados sejam reconhecidos como válidos.

A Figura 2 ilustra o funcionamento básico desse processo, desde a geração do par de chaves por uma entidade até o uso do certificado emitido pela AC para estabelecer comunicação segura.

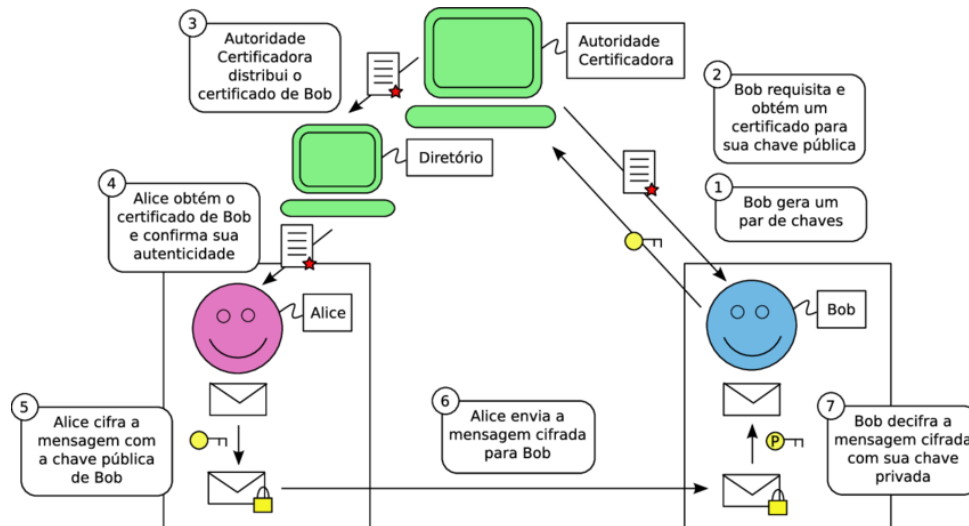


Figura 2 – Funcionamento básico de uma ICP tradicional.

Fonte: Adaptado de (ARANHA, 2007).

Definição 4 (Certificado Digital Válido). *Um certificado digital é considerado válido quando satisfaz simultaneamente três condições: (i) sua assinatura digital foi emitida por uma Autoridade Certificadora reconhecida e pode ser verificada com a chave pública dessa AC; (ii) a data atual encontra-se dentro do período de validade especificado nos campos *notBefore* e *notAfter*; e (iii) o certificado não consta em nenhuma Lista de Certificados Revogados (LCR) vigente nem foi revogado segundo consulta ao protocolo OCSP (COOPER et al., 2008).*

2.4.4 Gerenciamento de Certificados

Manter a segurança e a validade dos certificados digitais exige gerenciamento ao longo de todo o seu ciclo de vida. Esse acompanhamento abrange desde a emissão até a revogação e a validação, garantindo que apenas certificados válidos e atualizados sejam utilizados.

O ciclo de vida de um certificado digital compreende diversas fases que asseguram sua integridade:

1. **Emissão:** A Autoridade Certificadora (AC) emite o certificado digital após verificar a identidade da entidade solicitante. Este processo assegura que apenas entidades legítimas obtenham certificados válidos, estabelecendo uma base de confiança.
2. **Distribuição:** O certificado é disponibilizado ao titular e, quando necessário, a outras partes interessadas. Esta etapa garante que o certificado possa ser utilizado corretamente nas comunicações seguras.
3. **Renovação:** Antes da expiração, o certificado pode ser renovado para manter sua validade contínua. A renovação garante que a entidade continue a ser

reconhecida como legítima, sem interrupções na segurança.

4. **Revogação:** Em situações onde o certificado é comprometido ou as informações do titular são alteradas, a AC pode revogar o certificado, invalidando-o antes de sua data de expiração. A revogação é a medida para responder a incidentes de segurança e manter a integridade da ICP.
5. **Expiração:** O certificado deixa de ser válido automaticamente ao final do período estipulado, exigindo renovação para continuar sendo utilizado. A expiração temporal impede o uso de certificados desatualizados ou potencialmente comprometidos.

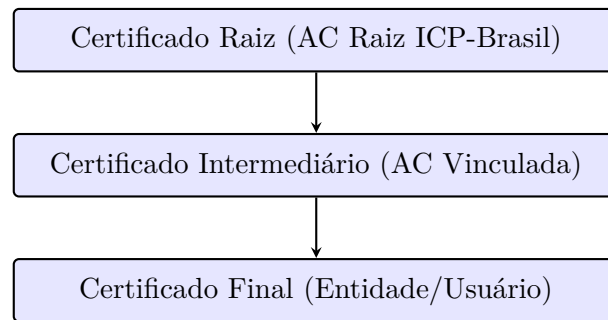
Em assinaturas digitais de documentos, a expiração posterior do certificado não invalida automaticamente uma assinatura produzida quando o certificado ainda era válido. Para sustentar essa validade ao longo do tempo, usam-se carimbos de tempo emitidos por Autoridades de Carimbo do Tempo (ACT), que atestam que a assinatura existia em determinado instante. Essa evidência temporal permite distinguir uma assinatura realizada durante a vigência do certificado de uma assinatura produzida após sua expiração ou revogação.

2.4.5 Validação de Certificados

Validar um certificado é indispensável para assegurar sua autenticidade e confiabilidade, o que envolve as seguintes etapas:

- **Verificação da Assinatura:** Confirma-se se o certificado foi assinado por uma AC confiável, utilizando a chave pública da AC.
- **Cheragem da Cadeia de Certificação:** Assegura-se que existe uma cadeia válida de confiança desde o certificado até uma AC raiz confiável.
- **Verificação de Revogação:** Utiliza-se mecanismos como Listas de Revogação de Certificados (LCRs) ou o Protocolo de Status de Certificado Online (OCSP) para determinar se o certificado foi revogado antes da data de expiração.
- **Validade Temporal:** Confirma-se que o certificado está dentro do seu período de validade.
- **Políticas de Certificado:** Verificam-se se o certificado atende aos requisitos específicos da aplicação ou serviço em questão.

A Figura 3 ilustra a estrutura hierárquica da cadeia de certificados X.509 e as etapas de validação percorridas desde o certificado final até a AC raiz.

**Etapas de Validação de Cadeia X.509:**

1. Verificar a assinatura de cada certificado com a chave pública do emissor
2. Verificar o período de validade (*notBefore/notAfter*)
3. Verificar o status de revogação (CRL/OCSP)
4. Verificar restrições básicas (*BasicConstraints, KeyUsage*)
5. Confirmar que a cadeia termina em âncora de confiança (*trust anchor*)

Figura 3 – Fluxograma de validação de cadeia de certificados X.509.

Fonte: Elaborado pelo autor.

2.4.6 Mecanismos de Revogação de Certificados

Revogar um certificado é a medida de segurança que o invalida quando ele não deve mais ser considerado confiável. A validação de revogação ocorre depois que a cadeia de confiança foi construída: o sistema identifica o emissor do certificado, consulta as informações de distribuição presentes nas extensões X.509 e verifica se há uma LCR vigente ou uma resposta OCSP assinada por entidade autorizada. Existem dois mecanismos principais para a revogação de certificados:

- **Listas de Revogação de Certificados (LCRs):** São listas emitidas periodicamente pelas Autoridades Certificadoras (ACs) contendo os números de série dos certificados que foram revogados antes do término de sua validade (COOPER *et al.*, 2008). As LCRs são assinadas digitalmente pela AC, garantindo a autenticidade e integridade das informações contidas. A periodicidade de atualização das LCRs depende do tipo de AC: ACs subordinadas podem emitir LCRs diariamente, enquanto uma AC raiz, de natureza mais estática, tende a atualizá-las com frequência menor, em geral mensal. Ainda assim, uma LCR pode ser atualizada a qualquer momento diante de uma situação de emergência. Embora sejam simples de implementar e não exijam comunicação em tempo real, as LCRs podem apresentar problemas de escalabilidade e atraso na atualização das informações, especialmente em ambientes com um grande número de certificados.
- **Protocolo de Status de Certificado Online (OCSP):** Permite a verificação em tempo real do status de revogação de um certificado (SANTESSON *et al.*,

2013). Diferentemente das LCRs, o OCSP oferece respostas imediatas sobre o status do certificado (válido, revogado ou desconhecido), melhorando a eficiência e reduzindo a latência na validação.

O funcionamento do OCSP envolve uma interação entre o cliente, o servidor OCSP e a Autoridade Certificadora (AC). Quando um cliente necessita verificar o status de um certificado, ele envia uma requisição OCSP para o servidor OCSP designado pela AC que emitiu o certificado. Essa requisição contém o identificador do certificado a ser verificado. O servidor OCSP, por sua vez, consulta sua base de dados para determinar o status atual do certificado e retorna uma resposta assinada digitalmente pela AC responsável.

O **OCSP Stapling**, por sua vez, é uma extensão que permite ao servidor web anexar a resposta OCSP diretamente ao certificado durante o estabelecimento da conexão SSL/TLS (EASTLAKE, 2011). Esta abordagem reduz a carga nos servidores OCSP, melhora o desempenho ao evitar consultas adicionais e aumenta a privacidade, uma vez que o cliente não precisa contatar diretamente o servidor OCSP. Por essas razões, o OCSP Stapling é adotado em servidores web de alto desempenho e recomendado por diretrizes recentes de segurança como alternativa eficiente à consulta direta ao respondedor OCSP.

2.4.7 Carimbos de Tempo

Carimbos de tempo vinculam uma evidência criptográfica, como o resumo de um documento assinado, a uma data e hora confiáveis. Em uma ICP, essa função é desempenhada por uma Autoridade de Carimbo do Tempo (ACT), que assina uma resposta contendo o instante declarado e o valor associado ao documento. O carimbo de tempo é especialmente importante em assinaturas eletrônicas de longa duração, pois permite demonstrar que a assinatura foi criada enquanto o certificado ainda estava válido e antes de eventual revogação. Perfis avançados de assinatura, como CAdES, podem incorporar essa evidência temporal juntamente com certificados e informações de revogação, preservando a verificabilidade da assinatura mesmo anos após a expiração do certificado original.

2.5 PADRÕES E PROTOCOLOS RELEVANTES

A conformidade com padrões internacionais garante a interoperabilidade e a segurança nas implementações de Infraestruturas de Chaves Públicas (ICP). Tais padrões estabelecem diretrizes e especificações que orientam a gestão de chaves públicas e certificados digitais, assegurando que as soluções adotadas sejam robustas e compatíveis com as melhores práticas globais. Sem essa padronização, cada fornecedor de software precisaria implementar formatos proprietários, o que inviabilizaria a comunicação segura entre sistemas de diferentes origens e comprometeria a confiança nas comunicações da Internet.

2.5.1 Extensões do padrão X.509

O padrão X.509 constitui a base dos certificados digitais utilizados em diversas aplicações, incluindo SSL/TLS para segurança na web, assinaturas digitais e autenticação de e-mails (UNION, 2016). O padrão define a estrutura dos certificados digitais, as Listas de Revogação de Certificados (LCRs) e os algoritmos de autenticação, oferecendo fundamento sólido para a implementação de soluções de segurança. A longevidade do X.509, cuja primeira versão data de 1988, atesta sua eficácia como modelo de confiança, tendo evoluído ao longo de três versões para acomodar novos requisitos de segurança sem perder a compatibilidade retroativa.

Além dos campos básicos que compõem a estrutura de um certificado X.509, as extensões permitem uma adaptação flexível dos certificados a diferentes cenários e necessidades. As principais extensões incluem:

- **Uso da Chave (Key Usage):** Especifica as finalidades permitidas para a chave pública, como assinatura digital, criptografia de dados ou autenticação de servidor.
- **Uso Estendido da Chave (Extended Key Usage):** Define usos adicionais para a chave pública, como autenticação de cliente, proteção de e-mail ou estabelecimento de túneis VPN.
- **Nomes Alternativos do Sujeito (Subject Alternative Name):** Permite associar múltiplos nomes, como DNS, IP ou e-mail, a um único certificado.
- **Políticas de Certificado (Certificate Policies):** Definem regras e requisitos associados ao certificado, como as práticas de emissão e os níveis de segurança exigidos. Esta extensão especifica as políticas sob as quais o certificado foi emitido, permitindo que os usuários confiem nas condições de emissão estabelecidas pela Autoridade Certificadora (AC).
- **Basic Constraints:** Indica se o certificado pertence a uma AC e especifica a profundidade máxima de certificação na hierarquia de certificados.
- **CRL Distribution Points:** Fornece URLs onde as Listas de Revogação de Certificados (CRLs) podem ser obtidas, facilitando a verificação da revogação de certificados.
- **Authority Information Access (AIA):** Especifica métodos para acessar informações adicionais sobre a autoridade certificadora, como pontos de distribuição para certificados de AC intermediárias.
- **Subject Key Identifier:** Fornece um identificador único para a chave pública do sujeito, facilitando a correspondência entre certificados e chaves.
- **Authority Key Identifier:** Fornece um identificador para a chave pública da autoridade certificadora que emitiu o certificado, auxiliando na validação da

cadeia de certificação.

- **Informações Adicionais (Additional Information):** Inclui dados complementares, como a localização geográfica do titular ou informações de contato, enriquecendo os certificados com atributos relevantes para contextos específicos.

2.5.2 Normas

A padronização de práticas criptográficas é conduzida por diferentes organismos internacionais, cada qual atuando em seu respectivo domínio geográfico e normativo. Nos Estados Unidos, o *National Institute of Standards and Technology* (NIST) publica diretrizes como a *SP 800-57*, que orienta a gestão de chaves criptográficas, incluindo geração, distribuição, armazenamento, uso e destruição, sendo essas recomendações seguidas por sistemas governamentais e comerciais nos Estados Unidos (STANDARDS; TECHNOLOGY, 2006). Na Europa, o *European Telecommunications Standards Institute* (ETSI) define especificações técnicas para assinaturas eletrônicas avançadas, enquanto no Brasil a Infraestrutura de Chaves Públicas Brasileira (ICP-Brasil), coordenada pelo Instituto Nacional de Tecnologia da Informação (ITI), estabelece as políticas e os padrões nacionais para certificação digital (ADAMS; LLOYD, 2002). Em conjunto, essas iniciativas asseguram a interoperabilidade entre sistemas de diferentes jurisdições.

O protocolo TLS, padronizado pelo IETF, é o principal exemplo prático de convergência desses padrões: ele utiliza certificados X.509 para autenticação de servidores, negocia algoritmos de cifra simétrica e assimétrica durante o *handshake* inicial, e garante a confidencialidade e a integridade dos dados transmitidos entre cliente e servidor (RESCORLA, 2018). Dessa forma, o TLS sintetiza os conceitos de ICP, criptografia híbrida e formatos de assinatura, constituindo a base da segurança na web moderna. A versão mais recente do protocolo, TLS 1.3, simplificou o *handshake* e removeu algoritmos considerados obsoletos, reforçando a tendência de adoção de criptografia de curvas elípticas e modos de operação autenticados.

Outras normas relevantes incluem:

- **RFC 5652:** Define o *Cryptographic Message Syntax (CMS)*, um padrão para criptografia e assinatura de mensagens digitais (HOUSLEY, 2009).
- **ETSI TS 101 733:** Estabelece os requisitos para *CAdES (CMS Advanced Electronic Signatures)*, garantindo conformidade legal e interoperabilidade (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2013).
- **RFC 8017:** Especifica o *PKCS#1*, que define o padrão para criptografia e assinaturas digitais RSA (MORIARTY *et al.*, 2016).

Essas normas complementam as diretrizes do NIST e RFC 5280, fornecendo especificações detalhadas para a implementação de funcionalidades de ICP e criptografia. A adesão a tais normas não apenas assegura a compatibilidade técnica entre sistemas

heterogêneos, mas também constitui requisito obrigatório em processos de certificação e auditoria de segurança, especialmente em setores regulados como o financeiro e o governamental. Desse modo, qualquer biblioteca criptográfica que almeje aplicação em ambientes de produção deve implementar esses padrões em conformidade com suas especificações mais recentes.

2.5.3 Formatos de Assinatura e ASN.1

Para que assinaturas digitais e mensagens criptografadas possam ser interpretadas corretamente por sistemas heterogêneos, é necessário o uso de formatos padronizados de representação. Uma assinatura puramente binária, frequentemente denominada assinatura *raw*, consiste em uma sequência de bytes sem qualquer metainformação associada, o que inviabiliza a identificação automática do algoritmo utilizado, do certificado do signatário ou do formato dos dados originais (HOUSLEY, 2009). Para contornar essa limitação, foram desenvolvidos formatos que encapsulam essas informações de maneira estruturada e autodescritiva.

O **Cryptographic Message Syntax (CMS)**, definido pela RFC 5652, é um formato derivado do PKCS#7 que especifica como dados criptografados, assinados ou autenticados devem ser encapsulados para transmissão ou armazenamento (HOUSLEY, 2009). O CMS permite a inclusão de certificados X.509, listas de revogação e múltiplos signatários em uma única estrutura, garantindo que qualquer implementação compatível consiga interpretar corretamente o conteúdo. O **CAdES (CMS Advanced Electronic Signatures)**, padronizado pelo ETSI na TS 101 733, estende o CMS com elementos voltados à conformidade legal, como carimbos de tempo, referências a políticas de assinatura e informações de validação de longo prazo (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2013). Essas extensões são necessárias para que assinaturas eletrônicas mantenham validade jurídica por períodos prolongados, mesmo após a expiração dos certificados originais ou a obsolescência dos algoritmos criptográficos utilizados.

A mesma filosofia foi estendida pelo ETSI a outros tipos de contêiner: o **PAdES (PDF Advanced Electronic Signatures)** aplica esses princípios a documentos PDF (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2009), o **XAdES (XML Advanced Electronic Signatures)** a dados em formato XML (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2010), e o **JAdES (JSON Advanced Electronic Signatures)** a estruturas JSON (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2021), formando uma família coerente de perfis de assinatura avançada adaptados a diferentes formatos de dados.

Neste contexto, o termo “suporte completo” a CMS/CAdES significa que uma biblioteca é capaz não apenas de gerar e verificar assinaturas nesses formatos, mas também de construir as estruturas de encapsulamento, incluir cadeias de certificados, incorporar atributos assinados exigidos por perfil CAdES e realizar a validação completa conforme

as regras definidas nas especificações (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2013). Sem esse suporte, a interoperabilidade com sistemas que exigem assinaturas padronizadas, como portais governamentais, sistemas judiciais eletrônicos e plataformas de contratação pública, fica comprometida. Portanto, para que uma biblioteca criptográfica atenda às demandas de aplicações reais de ICP, é indispensável que ela implemente tanto as primitivas criptográficas básicas quanto as camadas de formatação e os perfis normativos que garantem a comunicação entre sistemas heterogêneos. A Figura 4 detalha a anatomia de uma estrutura CMS/CAdES, evidenciando os campos que compõem o **SignedData** e os atributos assinados adicionados pelo perfil CAdES.

- **ContentInfo**

- **contentType**: id-signedData
- **content**: **SignedData**
 - * **version**: 1/3
 - * **digestAlgorithms**: [SHA-256, SHA-512, ...]
 - * **encapContentInfo**: eContentType, eContent (opcional *detached*)
 - * **certificates**: [Cadeia X.509] (opcional)
 - * **crls**: [Listas de Revogação] (opcional)
 - * **signerInfos**: **SignerInfo**
 - **version**, **sid** (IssuerAndSerialNumber), **digestAlgorithm**
 - **signedAttrs**: **contentType**, **messageDigest**, **signingTime** (CAdES), **signingCertificate** (CAdES), **signaturePolicyIdentifier** (CAdES), ...
 - **signatureAlgorithm**: RSA-2048 / ECDSA P-256
 - **signature**: [valor da assinatura]

Figura 4 – Anatomia de uma estrutura CMS/CAdES (PKCS#7 SignedData), com destaque para os atributos assinados do perfil CAdES.

Fonte: Elaborado pelo autor.

A base sintática comum a CMS, CAdES e aos certificados X.509 é a **Abstract Syntax Notation One (ASN.1)**, uma linguagem de descrição de estruturas de dados independente de plataforma, padronizada pela ITU-T (UNION, 2016). ASN.1 define regras formais para especificar tipos de dados, enquanto seus esquemas de codificação, como DER (*Distinguished Encoding Rules*) e BER (*Basic Encoding Rules*), determinam a representação binária desses dados. Dominar ASN.1 e seus esquemas de codificação torna-se, assim, requisito para qualquer biblioteca que pretenda interoperar com o ecossistema de ICP global.

2.6 ALGORITMOS PÓS-QUÂNTICOS

Este trabalho não desenvolve novas construções matemáticas pós-quânticas nem reimplementa internamente os algoritmos padronizados. A fundamentação a seguir delimita os conceitos necessários para compreender ML-KEM e ML-DSA; no artefato desenvolvido,

suas implementações são fornecidas pelo OpenSSL e acessadas pela camada Dart por meio da API EVP e de *bindings* FFI.

A computação quântica representa uma ameaça concreta à segurança dos esquemas criptográficos assimétricos atualmente em uso. Em 1994, Peter Shor demonstrou que um computador quântico com capacidade suficiente poderia fatorar números inteiros e computar logaritmos discretos em tempo polinomial, o que comprometeria a segurança do RSA, do Diffie-Hellman e dos esquemas baseados em curvas elípticas, como o ECDSA (SHOR, 1994). Em paralelo, o algoritmo de Grover (1996) provê uma aceleração quadrática para buscas em espaços não estruturados, reduzindo pela metade a segurança efetiva de cifradores simétricos e funções de *hash*. Assim, um AES-128, por exemplo, ofereceria apenas 64 bits de segurança contra um adversário quântico (BERNSTEIN, 2009).

Embora computadores quânticos com poder de processamento suficiente para essas aplicações ainda não existam, a evolução da tecnologia tem avançado rapidamente, e a comunidade criptográfica internacional mobilizou-se para padronizar algoritmos resistentes a ataques quânticos, denominados algoritmos pós-quânticos (*Post-Quantum Cryptography* — PQC). Diferentemente da criptografia quântica, que utiliza propriedades da física quântica para distribuição de chaves, a PQC baseia-se em problemas matemáticos clássicos para os quais não se conhecem algoritmos quânticos eficientes (BERNSTEIN, 2009; CHEN; AL., 2024).

2.6.1 O Processo de Padronização do NIST

Em 2016, o NIST iniciou um processo competitivo e público de padronização de algoritmos pós-quânticos (STANDARDS; TECHNOLOGY, 2020). O processo foi estruturado em múltiplas rodadas de avaliação, nas quais as candidaturas foram analisadas quanto à segurança, ao desempenho e à adequação para implementação em hardware e software:

- **Primeira rodada (2017–2019):** Das 69 candidaturas submetidas, 64 foram aceitas como completas, abrangendo cinco famílias matemáticas distintas: reticulados, códigos corretores de erros, sistemas multivariados, funções de *hash* e isogenias de curvas elípticas supersingulares (STANDARDS; TECHNOLOGY, 2020);
- **Segunda rodada (2019–2020):** 26 algoritmos avançaram, com predominância de esquemas baseados em reticulados;
- **Terceira rodada (2020–2022):** 7 finalistas e 8 candidatos alternativos foram selecionados. Em julho de 2022, o NIST anunciou a escolha do **CRYSTALS-Kyber** para encapsulamento de chaves e do **CRYSTALS-Dilithium** para assinaturas digitais, além do **Falcon** e do **SPHINCS⁺** como alternativas (DUCAS *et al.*, 2018; CHEN; AL., 2024);

- **Padronização (2024):** Em agosto de 2024, o NIST publicou os três primeiros padrões federais: FIPS 203 (ML-KEM, evolução do CRYSTALS-Kyber), FIPS 204 (ML-DSA, evolução do CRYSTALS-Dilithium) e FIPS 205 (SLH-DSA, baseado no SPHINCS⁺) (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b, 2024a).

2.6.2 Fundamentos de Criptografia Baseada em Reticulados

Os dois padrões de maior relevância para este trabalho, ML-KEM e ML-DSA, pertencem à família dos reticulados (*lattices*), que constitui a abordagem predominante na PQC contemporânea. Um reticulado é um conjunto discreto de pontos em um espaço n -dimensional, gerado por todas as combinações lineares inteiras de um conjunto de vetores linearmente independentes denominados base (REGEV, 2009). A segurança dos esquemas baseados em reticulados fundamenta-se na dificuldade computacional de resolver certos problemas geométricos nessa estrutura, como encontrar o vetor mais curto (*Shortest Vector Problem*, SVP) ou o vetor mais próximo (*Closest Vector Problem*, CVP), para os quais não se conhecem algoritmos eficientes, nem clássicos nem quânticos (REGEV, 2009; BERNSTEIN, 2009).

O problema específico que sustenta o ML-KEM e o ML-DSA é o *Module Learning With Errors* (M-LWE), uma generalização estruturada do problema *Learning With Errors* (LWE) introduzido por Regev em 2005 (REGEV, 2009). Informalmente, o LWE consiste em recuperar um vetor secreto $\mathbf{s}$ a partir de um conjunto de equações lineares ruidosas da forma $\langle \mathbf{a}_i, \mathbf{s} \rangle + e_i = b_i$, em que $\mathbf{a}_i$ são vetores públicos, b_i são os valores observados e e_i são pequenos erros aleatórios. A presença do erro torna o problema computacionalmente difícil, e demonstrou-se que resolvê-lo no caso médio é tão difícil quanto resolver problemas de reticulado no pior caso (REGEV, 2009). O M-LWE estende essa formulação para matrizes de polinômios sobre anéis, oferecendo um compromisso entre a flexibilidade do LWE genérico e a eficiência das variantes estruturadas.

2.6.3 ML-KEM: Mecanismo de Encapsulamento de Chaves

Um *Key-Encapsulation Mechanism* (KEM) é um esquema criptográfico assimétrico projetado especificamente para o estabelecimento seguro de chaves simétricas compartilhadas entre duas partes que não possuem um canal prévio de comunicação segura (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b). Diferentemente de um esquema de cifração genérico, em que uma das partes escolhe a mensagem a ser transmitida, o KEM gera internamente o segredo compartilhado e o encapsula criptograficamente. O processo compõe-se de três algoritmos:

1. **KeyGen()** $\rightarrow (pk, sk)$: gera um par de chaves pública e privada;

2. $\text{Encaps}(pk) \rightarrow (ct, ss)$: a partir da chave pública, produz um texto cifrado ct e um segredo compartilhado ss ;
3. $\text{Decaps}(sk, ct) \rightarrow ss$: a partir da chave privada e do texto cifrado, recupera o segredo compartilhado.

Definição 5 (ML-KEM: Mecanismo de Encapsulamento de Chaves Baseado em Reticulados Modulares). O **ML-KEM** (*Module-Lattice-based Key-Encapsulation Mechanism*), padronizado pelo NIST como FIPS 203 em 2024, é um mecanismo de encapsulamento de chaves pós-quântico derivado do algoritmo **CRYSTALS-Kyber** (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b). Sua segurança fundamenta-se no problema *Module Learning With Errors* (MLWE), uma generalização estruturada do LWE para reticulados modulares, para a qual não se conhecem algoritmos eficientes de resolução, nem clássicos nem quânticos. O esquema define três operações, geração de chaves (**KeyGen**), encapsulamento (**Encaps**) e decapsulamento (**Decaps**), e três conjuntos de parâmetros progressivos de segurança: **ML-KEM-512** (nível NIST 1), **ML-KEM-768** (nível NIST 3) e **ML-KEM-1024** (nível NIST 5).

O ML-KEM define três conjuntos de parâmetros que oferecem níveis crescentes de segurança, com a contrapartida de tamanhos maiores de chave e texto cifrado, conforme apresentado na Tabela 2 (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b):

Tabela 2 – Conjuntos de parâmetros do ML-KEM (FIPS 203).

Variante	Nível NIST	pk (bytes)	sk (bytes)	ct (bytes)	Equivalente clássico
ML-KEM-512	1	800	1.632	768	AES-128
ML-KEM-768	3	1.184	2.400	1.088	AES-192
ML-KEM-1024	5	1.568	3.168	1.568	AES-256

Fonte: Adaptado de (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b).

Observa-se que mesmo a variante mais leve (ML-KEM-512) produz chaves públicas de 800 bytes, mais de uma ordem de grandeza acima das chaves públicas ECDSA de 64 bytes discutidas na Seção 2.3. Esse aumento de tamanho constitui um dos principais desafios de engenharia para a adoção prática de algoritmos pós-quânticos, com implicações para o armazenamento de certificados e para a largura de banda de protocolos como o TLS.

2.6.4 ML-DSA: Algoritmo de Assinatura Digital

O **ML-DSA** (*Module-Lattice-based Digital Signature Algorithm*), padronizado pelo NIST como FIPS 204 em 2024, é um algoritmo de assinatura digital pós-quântico derivado do esquema **CRYSTALS-Dilithium** (NATIONAL INSTITUTE OF STANDARDS AND

TECHNOLOGY, 2024a; DUCAS *et al.*, 2018). À semelhança dos esquemas de assinatura clássicos discutidos na Seção 2.3, o ML-DSA provê as propriedades fundamentais de autenticidade, integridade e não repúdio, com a distinção de que sua segurança é considerada robusta mesmo contra adversários com acesso a computadores quânticos de grande escala. O esquema apoia-se na dificuldade combinada de dois problemas computacionais sobre reticulados: o *Module Learning With Errors* (M-LWE) e o *Module Short Integer Solution* (M-SIS) (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024a; DUCAS *et al.*, 2018). O esquema define três operações: geração de chaves (**KeyGen**), assinatura (**Sign**) e verificação (**Verify**). Também define três conjuntos de parâmetros com níveis progressivos de segurança: ML-DSA-44 (nível NIST 2), ML-DSA-65 (nível NIST 3) e ML-DSA-87 (nível NIST 5).

As três variantes padronizadas do ML-DSA são (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024a):

Tabela 3 – Conjuntos de parâmetros do ML-DSA (FIPS 204).

Variante	Nível NIST	pk (bytes)	sk (bytes)	Assinatura (bytes)
ML-DSA-44	2	1.312	2.560	2.420
ML-DSA-65	3	1.952	4.032	3.309
ML-DSA-87	5	2.592	4.896	4.627

Fonte: Adaptado de (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024a).

Em comparação com o ECDSA sobre a curva P-256, cujas chaves públicas ocupam 64 bytes e cujas assinaturas ocupam tipicamente entre 70 e 72 bytes, as assinaturas do ML-DSA são de 34 a 66 vezes maiores, o que reitera o impacto da transição pós-quântica sobre os custos de armazenamento e transmissão.

Tabela 4 – Comparação de tamanhos: algoritmos clássicos *vs.* pós-quânticos.

Algoritmo	Chave Pública	Assinatura/Texto Cifrado
RSA-2048	256 B	256 B (assinatura)
ECDSA P-256	64 B	70–72 B (assinatura)
ML-KEM-768	1.184 B	1.088 B (texto cifrado)
ML-DSA-44	1.312 B	2.420 B (assinatura)

Fonte: Elaborado pelo autor com base nos padrões FIPS 203 e FIPS 204.

A Tabela 4 ilustra que a segurança pós-quântica tem um custo em termos de tamanho: a chave pública do ML-KEM-768 é cerca de $4,6\times$ maior que a do RSA-2048, e a assinatura do ML-DSA-44 é aproximadamente $34\times$ maior que a do ECDSA P-256. Na prática, isso significa *handshakes* TLS mais longos, cadeias de certificados maiores e maior consumo de largura de banda, impactos que a comunidade internacional ainda está endereçando por meio de propostas em discussão no IETF e no ETSI (CHEN; AL., 2024).

2.7 FOREIGN FUNCTION INTERFACE (FFI)

Uma *Foreign Function Interface* (FFI) permite que programas escritos em uma linguagem invoquem funções compiladas em outra, geralmente C. No Dart, o pacote `dart:ffi` possibilita o carregamento de bibliotecas nativas e o mapeamento de funções, estruturas e ponteiros para tipos acessíveis pela aplicação (TEAM, D., 2024c).

Como as chamadas ocorrem no mesmo processo, a FFI evita a serialização exigida por mecanismos de troca de mensagens. Em contrapartida, transfere para a aplicação responsabilidades relacionadas à correspondência de tipos, ao tempo de vida dos objetos nativos e à liberação de memória. Erros nesses pontos podem causar vazamentos, acessos inválidos e falhas de execução.

Esse mecanismo é recorrente em aplicações que precisam reutilizar bibliotecas nativas já consolidadas ou obter desempenho próximo ao código compilado. Em aplicações móveis, FFI permite acessar bibliotecas C distribuídas junto ao aplicativo; em aplicações desktop, permite consumir APIs de sistema operacional ou bibliotecas científicas; em software criptográfico, permite reaproveitar implementações avaliadas e otimizadas, como OpenSSL, sem reescrever primitivas sensíveis em uma linguagem de alto nível.

No caso de uma chamada FFI, a aplicação Dart carrega uma biblioteca dinâmica, resolve o símbolo nativo desejado e declara duas assinaturas: uma assinatura C, baseada nos tipos nativos, e uma assinatura Dart, usada pelo restante do programa. Ponteiros opacos, como estruturas internas do OpenSSL, são normalmente representados como `Pointer<Void>`, pois a aplicação não interpreta sua memória diretamente; ela apenas transporta esses ponteiros entre funções nativas responsáveis por criar, usar e liberar os recursos.

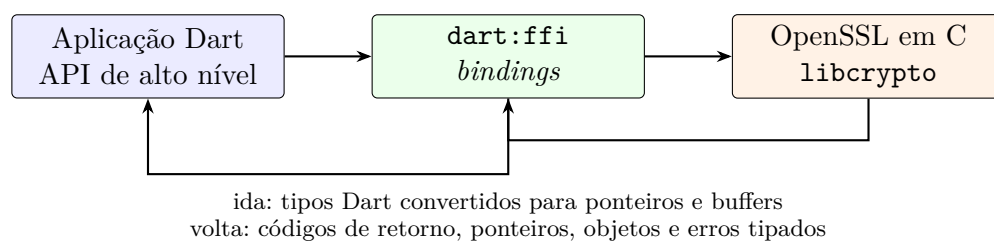


Figura 5 – Fluxo conceitual de uma chamada FFI entre Dart e uma biblioteca C nativa.

Fonte: Elaborado pelo autor.

Neste trabalho, a FFI estabelece a comunicação entre Dart e OpenSSL. A organização dos *bindings*, o carregamento das bibliotecas e o gerenciamento de memória são apresentados no Capítulo 4.

2.8 OPENSSL E A API EVP

O **OpenSSL** é uma biblioteca criptográfica de código aberto, adotada em servidores Web, clientes de e-mail, VPNs e sistemas operacionais (OPENSSL PROJECT AUTHORS,

2026a). Sua trajetória, iniciada em 1998 como uma ramificação da SSLeay, reflete a evolução dos padrões de segurança na Internet: da introdução do suporte a TLS 1.3 na série 1.1, passando pelo redesenho da arquitetura de provedores (*Provider Architecture*) na série 3.0, até a reestruturação profunda concretizada na versão 4.0, que removeu APIs legadas e consolidou a **API EVP** (*Envelope*) como único ponto de acesso às operações criptográficas (OPENSSL PROJECT AUTHORS, 2026a, 2026b).

A API EVP abstrai os detalhes de implementação de cada algoritmo por meio de uma interface uniforme de alto nível. Em vez de invocar diretamente funções específicas de baixo nível, como `RSA_sign()` ou `AES_cbc_encrypt()`, o desenvolvedor utiliza contextos genéricos (`EVP_MD_CTX` para *hash*, `EVP_CIPHER_CTX` para cifras, `EVP_PKEY_CTX` para chaves assimétricas) que encapsulam a seleção e a configuração do algoritmo subjacente (OPENSSL PROJECT AUTHORS, 2026a). Essa arquitetura oferece três benefícios centrais para o desenvolvimento de bibliotecas criptográficas:

1. **Independência de algoritmo:** a troca de um algoritmo de *hash* ou de um cifrador simétrico requer apenas a substituição do identificador do algoritmo (por exemplo, de `EVP_sha256()` para `EVP_sha3_256()`), sem alterações na lógica de controle;
2. **Suporte a *providers* carregáveis:** a arquitetura de provedores introduzida no OpenSSL 3.0 e consolidada no 4.0 permite que implementações externas, incluindo algoritmos pós-quânticos, módulos de hardware (*engines*) e políticas de conformidade FIPS, sejam carregadas dinamicamente, sem recompilação do núcleo;
3. **Sustentabilidade:** a remoção das APIs de baixo nível na versão 4.0 força o uso exclusivo da EVP, garantindo que o código permaneça compatível com versões futuras e eliminando a dependência acidental de funções obsoletas ou com falhas de segurança conhecidas (OPENSSL PROJECT AUTHORS, 2026b).

A escolha do OpenSSL como *backend* nativo para a `dartcryptosec`, em detrimento de alternativas como o LibreSSL ou o BoringSSL, fundamenta-se em três fatores: a maturidade do ecossistema (o OpenSSL é a biblioteca criptográfica mais utilizada em servidores e dispositivos embarcados), a antecipação da adoção de algoritmos pós-quânticos via provedores (ML-KEM e ML-DSA figuram como provedores de primeira classe na versão 4.0) e a disponibilidade de documentação de migração estruturada, que permite auditar e manter os bindings FFI com confiabilidade (OPENSSL PROJECT AUTHORS, 2026b). As decisões de engenharia decorrentes dessa escolha, incluindo a organização dos *bindings*, a estratégia de alocação de memória e a abordagem de zeroização, são detalhadas no Capítulo 4.

3 METODOLOGIA

A metodologia adotada neste trabalho estruturou-se em etapas que buscaram, em primeiro lugar, mapear o estado atual das bibliotecas de criptografia disponíveis no ecossistema Dart, compreendendo sua abrangência, limitações e carências funcionais. Esse diagnóstico foi efetuado por meio de uma análise exploratória de material técnico, documentação oficial e códigos-fonte. Complementarmente, bibliotecas como Bouncy Castle e OpenSSL foram comparadas de forma sistemática com soluções já existentes em Dart, notadamente Pointycastle, Crypto e Cryptography. Ao longo dessa fase, foram identificadas lacunas no suporte criptográfico e em funcionalidades de ICP e confirmou-se a relevância desses recursos para o desenvolvimento seguro de aplicações multiplataforma.

Essa análise orientou o desenvolvimento incremental da biblioteca e a definição dos critérios empregados em sua avaliação. O procedimento combinou revisão bibliográfica, comparação de bibliotecas, implementação iterativa e testes automatizados nos ambientes Linux e Android. Os fundamentos teóricos que embasam essas decisões arquiteturais e metodológicas são apresentados no Capítulo 2, iniciando pelos princípios de criptografia que constituem o núcleo conceitual da biblioteca.

3.1 CLASSIFICAÇÃO DA PESQUISA

Esta pesquisa classifica-se, quanto à natureza, como **pesquisa aplicada**, pois envolve o desenvolvimento de um artefato de *software* destinado à integração entre Dart e OpenSSL.

No que se refere aos objetivos, caracteriza-se como **exploratória e descritiva**. A dimensão exploratória corresponde à investigação do uso de `dart:ffi` na integração com uma biblioteca criptográfica nativa. A dimensão descritiva refere-se ao registro das decisões de projeto, dos procedimentos de desenvolvimento e das condições de avaliação do artefato.

Em relação à abordagem, a pesquisa adota o método **quali-quantitativo**. A análise qualitativa foi empregada na comparação das bibliotecas e no exame das decisões arquiteturais. A análise quantitativa foi utilizada na avaliação dos testes, da cobertura de código e do desempenho das operações.

Quanto aos procedimentos técnicos, a pesquisa combina três estratégias complementares:

- a) **Pesquisa bibliográfica**: consulta à literatura e à documentação técnica relacionada aos conceitos e às tecnologias adotadas;
- b) **Estudo de caso**: desenvolvimento da biblioteca como caso de integração entre uma linguagem gerenciada e uma biblioteca nativa;
- c) **Pesquisa experimental**: execução controlada de testes automatizados, com

registro do ambiente, das versões utilizadas e das métricas avaliadas.

3.2 ETAPAS DA PESQUISA

A execução da pesquisa organizou-se em quatro etapas sequenciais, descritas a seguir.

3.2.1 Revisão Bibliográfica e Análise do Estado da Arte

A etapa inicial consistiu na consulta à literatura acadêmica, aos padrões técnicos e à documentação oficial das tecnologias utilizadas. O levantamento concentrou-se em ICP, formatos de assinatura digital, criptografia pós-quântica, FFI e OpenSSL. Os conceitos selecionados para sustentar o trabalho são apresentados no Capítulo 2.

3.2.2 Análise Comparativa de Bibliotecas Existentes

A segunda etapa comparou bibliotecas criptográficas do ecossistema Dart com soluções consolidadas em outras plataformas. A comparação considerou disponibilidade de operações de ICP, suporte multiplataforma, integração com código nativo, manutenção e documentação. As bibliotecas examinadas e os resultados dessa análise encontram-se no Capítulo 4.

3.2.3 Desenvolvimento Iterativo

A terceira etapa correspondeu ao desenvolvimento incremental da biblioteca, organizado em ciclos de implementação, teste e revisão. Em cada ciclo, um conjunto de operações era integrado ao OpenSSL, encapsulado pela API Dart e submetido a testes antes da inclusão de novas funcionalidades. Esse procedimento permitiu tratar separadamente problemas nos *bindings* FFI, gerenciamento de memória e organização da API. A arquitetura resultante e as decisões adotadas são detalhadas no Capítulo 4.

3.2.4 Testes e Validação

A quarta etapa avaliou o artefato por meio de testes unitários e de integração executados em Linux x86_64 e em dispositivo Android ARM64. Os testes foram organizados por domínio funcional e incluíram casos válidos, entradas inválidas e cenários de integração. Também foram coletadas métricas de cobertura e desempenho.

A avaliação de segurança incluiu a modelagem de ameaças pela metodologia STRIDE. O procedimento delimitou a superfície de ataque entre a API Dart, a fronteira FFI, os objetos nativos do OpenSSL e as bibliotecas carregadas em tempo de execução; em seguida, classificou as ameaças nas seis categorias do método e atribuiu pontuações

de probabilidade e impacto para priorizar seu tratamento. Os resultados dessa análise são apresentados na Seção 4.6.4 e detalhados no Apêndice B.

3.3 FERRAMENTAS E AMBIENTE DE DESENVOLVIMENTO

O desenvolvimento e a validação da **dartcryptosec** apoiaram-se em um conjunto de ferramentas e ambientes cuja configuração precisa é condição necessária para a reprodutibilidade dos resultados. O ambiente de desenvolvimento foi estabelecido sobre **Ubuntu 22.04** executado via *Windows Subsystem for Linux* (WSL1) sobre Windows 11, em um computador Avell A52i com processador Intel Core i7-13620H, 10 núcleos físicos, 16 núcleos lógicos, frequência nominal máxima de 2,4 GHz reportada pelo sistema e 32 GB de RAM.

As versões específicas das ferramentas empregadas estão reunidas no Quadro 5.

Tabela 5 – Ferramentas e versões utilizadas no desenvolvimento e validação

Ferramenta	Versão	Propósito
Dart	3.x (<i>stable</i>)	Linguagem de programação da camada pública do plugin
Flutter	3.24+ (<i>stable</i>)	<i>Framework</i> de desenvolvimento e execução de testes
OpenSSL	4.0.0	Motor criptográfico nativo, utilizado como <i>backend</i>
GCC	13.x	Compilador C para as bibliotecas nativas Linux
Android NDK	28.2.13676358	Compilação cruzada para Android (ARM64, ARMv7, x86_64)
Android OS	15 (API 35) + HyperOS	Sistema operacional do dispositivo de teste
<code>dart:ffi</code> (pacote <code>ffi</code>)	^2.1.4	<i>Helpers</i> para <i>bindings</i> FFI (<code>calloc</code> , <code>Utf8</code>)
<code>flutter_test</code>	SDK	<i>Framework</i> de testes unitários e de integração
<code>lcov</code>	–	Geração e processamento de relatórios de cobertura de código
<code>glados</code>	^1.1.7	Testes baseados em propriedades (<i>property-based testing</i>)
<code>ffigen</code>	^20.1.1	Referência para geração automática de <i>bindings</i> (não utilizado em produção)

Fonte: Elaborado pelo autor (2026).

O OpenSSL 4.0.0 foi utilizado como biblioteca criptográfica nativa e acessado por meio de *bindings* FFI escritos em Dart. As decisões relacionadas à escolha do OpenSSL e à estratégia adotada para esses *bindings* são discutidas no Capítulo 4.

O ambiente de testes utilizou `flutter test` como executor e `lcov` para geração dos relatórios de cobertura. A suíte foi organizada por etiquetas que permitem executar separadamente testes demorados, concorrentes, de estresse e de medição.

3.4 MÉTRICAS E CRITÉRIOS DE AVALIAÇÃO

A avaliação foi organizada em três categorias: correção funcional, cobertura de código e desempenho. Os valores obtidos e sua análise constam do Capítulo 5.

3.4.1 Cobertura de Código

A cobertura de código foi mensurada com `lcov` sobre os testes unitários executados em Linux. O critério de aceitação adotado foi cobertura global mínima de 60%. Ramos específicos de outras plataformas foram desconsiderados na interpretação dos resultados quando não podiam ser exercitados nesse ambiente.

3.4.2 Taxa de Sucesso dos Testes

O critério de correção funcional exigiu a aprovação integral da suíte padrão nas plataformas validadas. Testes ignorados por restrições de plataforma ou dependências externas foram registrados separadamente e não considerados como aprovações ou falhas.

3.4.3 Desempenho Computacional

O desempenho foi medido em Linux x86_64 com aquecimento prévio e repetições de cada operação. Na suíte principal, o medidor executou 75 chamadas de aquecimento e, conforme o custo da operação, 25 ou 75 chamadas cronometradas; as gerações de chaves, mais custosas, utilizaram de 3 a 5 chamadas de aquecimento e de 3 a 10 medições. Na comparação Linux/Android, foram usadas 5 iterações de aquecimento e 25 medições efetivas por operação. Os valores reportados correspondem à média aritmética das medições; a distribuição e a mediana foram calculadas para as duas operações cujas amostras individuais foram preservadas. Também foram registrados o *throughput* e, quando disponível, o desvio-padrão.

A implementação decorrente desta metodologia é descrita no Capítulo 4, que detalha a arquitetura, as decisões técnicas adotadas e os padrões de projeto aplicados no desenvolvimento da `dartcryptosec`.

4 IMPLEMENTAÇÃO

4.1 INTRODUÇÃO

Este capítulo apresenta as decisões de implementação da `dartcryptosec`. Inicialmente, são comparadas bibliotecas criptográficas de referência e soluções disponíveis no ecossistema Dart. Em seguida, são descritas a integração com o OpenSSL por meio de FFI, a arquitetura da biblioteca e a implementação dos algoritmos pós-quânticos. O capítulo também registra desafios técnicos encontrados durante o desenvolvimento e limitações que permanecem abertas.

A análise aqui conduzida toma como base a documentação oficial, os códigos-fonte públicos e a literatura técnica de cada biblioteca, com ênfase nas funcionalidades essenciais para Infraestrutura de Chaves Públicas (ICP): manipulação de certificados X.509, verificação de revogação por LCR e OCSP, suporte a formatos de assinatura CMS e CAdES, e geração de chaves criptográficas. A metodologia que orientou essa análise comparativa, incluindo os critérios de seleção das bibliotecas e os procedimentos de avaliação, foi detalhada no Capítulo 3. Os resultados dessa análise fundamentam as decisões arquiteturais discutidas na Seção 4.6 e a estratégia de integração via FFI apresentada na Seção 4.5.

4.2 BIBLIOTECAS CRIPTOGRÁFICAS DE REFERÊNCIA

Como parâmetro de comparação para as bibliotecas disponíveis no ecossistema Dart, apresentam-se a seguir duas bibliotecas criptográficas consagradas em sistemas de produção: Bouncy Castle e OpenSSL. Ambas oferecem implementações maduras, auditadas e em conformidade com os principais padrões internacionais de segurança, constituindo a referência de abrangência e robustez para qualquer biblioteca que almeje aplicação em cenários reais de ICP (CASTLE, 2024; OPENSSL PROJECT AUTHORS, 2026a). A seleção dessas duas bibliotecas como parâmetro decorre de sua presença ubíqua em sistemas corporativos, governamentais e de código aberto, bem como da extensa documentação e suporte comunitário de que dispõem.

4.2.1 Bouncy Castle

Bouncy Castle é uma das bibliotecas de criptografia mais completas e amplamente utilizadas nos ecossistemas Java e C# (CASTLE, 2024). Ela oferece uma vasta gama de funcionalidades que abrangem praticamente todos os aspectos da criptografia moderna, incluindo:

- **Cifradores Simétricos e Assimétricos:** Suporte extensivo a AES, DES, 3DES, RSA, ElGamal, ECDSA, entre outros, permitindo a implementação de uma ampla variedade de protocolos de segurança.

- **Algoritmos Pós-Quânticos:** Implementação dos padrões pós-quânticos do NIST, incluindo ML-KEM (FIPS 203), ML-DSA (FIPS 204) e SLH-DSA (FIPS 205). A biblioteca também inclui esquemas alternativos, como Falcon (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b, 2024a).
- **Modos de Operação:** Implementação completa de modos como CBC, ECB, CFB, OFB, CTR, GCM, CCM, garantindo flexibilidade na aplicação de cifradores simétricos.
- **Funções de *Hash* e MACs:** Suporte a MD2, MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-3, HMAC, CMAC, Poly1305, proporcionando ferramentas para criação de resumos criptográficos e autenticação de mensagens.
- **Derivação de Chaves (KDFs):** Implementações de PBKDF2, scrypt, HKDF, Argon2, permitindo a geração segura de chaves a partir de senhas ou outros materiais de entrada.
- **Certificados X.509 e ICP:** Implementação completa para criação, *parsing*, manipulação e verificação de certificados X.509, suporte a LCR e OCSP, permitindo a construção e o gerenciamento do ciclo de vida completo de certificados, emissão, renovação e revogação, em uma Infraestrutura de Chave Pública.
- **CMS/CAdES:** Suporte completo para *Cryptographic Message Syntax* (CMS) e assinaturas eletrônicas avançadas (CAdES), incluindo a construção de estruturas de encapsulamento, a inclusão de cadeias de certificados, a incorporação de atributos assinados exigidos por perfil CAdES e a validação completa conforme as regras definidas nas especificações ETSI TS 101 733 (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2013).
- **Geradores de Números Aleatórios:** Implementações seguras, essenciais para a geração de chaves criptográficas e outros componentes sensíveis.
- **ASN.1 *Parsing* e *Encoding*:** Ferramentas completas para manipulação de estruturas ASN.1, nas quais *parsing* designa a interpretação de dados codificados em ASN.1 para estruturas utilizáveis e *encoding*, o processo inverso de serialização de estruturas para o formato binário padronizado, garantindo a interoperabilidade com outros sistemas e bibliotecas.

4.2.2 OpenSSL

OpenSSL é uma biblioteca amplamente adotada para a implementação de protocolos de segurança como TLS/SSL, além de fornecer uma vasta gama de funcionalidades criptográficas (OPENSSL PROJECT AUTHORS, 2026a). Ela é essencial em ambientes

que requerem comunicação segura e é utilizada em diversas linguagens de programação e plataformas. As principais funcionalidades incluem:

- **Cifradores Simétricos e Assimétricos:** Suporte a AES, DES, 3DES, RSA, ECDSA, entre outros, base para a implementação de protocolos de segurança variados.
- **Algoritmos Pós-Quânticos:** Suporte aos padrões pós-quânticos do NIST, ML-KEM (FIPS 203) e ML-DSA (FIPS 204), disponibilizados a partir da versão 3.5 como provedores (*providers*) carregáveis pela arquitetura modular da biblioteca, sem necessidade de recompilação do núcleo (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b, 2024a).
- **Modos de Operação:** Suporte a CBC, ECB, CFB, OFB, CTR, GCM, o que confere flexibilidade ao uso de cifradores simétricos.
- **Funções de *Hash* e MACs:** Implementações de MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, HMAC, CMAC, que atendem à criação de resumos criptográficos e à autenticação de mensagens.
- **Derivação de Chaves (KDFs):** Implementações de PBKDF2, HKDF, scrypt, voltadas à derivação segura de chaves a partir de senhas ou outros materiais de entrada.
- **Certificados X.509 e ICP:** Conjunto completo para criação, *parsing*, manipulação e verificação de certificados X.509, com suporte a LCR e OCSP, o que viabiliza o gerenciamento do ciclo de vida de certificados em uma Infraestrutura de Chave Pública.
- **CMS/CAdES:** Suporte completo para CMS e CAdES, abrangendo a geração, verificação, construção de estruturas de encapsulamento, inclusão de cadeias de certificados e validação conforme perfil CAdES (EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE, 2013).
- **Geradores de Números Aleatórios Seguros:** Fontes de aleatoriedade indispensáveis à geração de chaves criptográficas e de outros componentes sensíveis.
- **ASN.1 *Parsing* e *Encoding*:** Ferramentas completas para interpretação e serialização de estruturas ASN.1, nas quais *parsing* refere-se à decodificação de dados ASN.1 e *encoding*, à codificação de estruturas para o formato binário DER/BER.
- **Protocolos de Segurança:** Implementação completa de TLS/SSL, DTLS e outros protocolos, permitindo a criação de comunicações seguras entre clientes e servidores.

4.3 BIBLIOTECAS CRIPTOGRÁFICAS EM DART

Nesta seção, são analisadas as capacidades e limitações das principais bibliotecas de criptografia disponíveis no ecossistema Dart (CONTRIBUTORS, 2024; TEAM, D. C., 2024; TEAM, D., 2024a). As três bibliotecas examinadas, Pointycastle, Cryptography e Crypto, representam as alternativas mais relevantes para o desenvolvimento de aplicações criptográficas na linguagem, cada qual com diferentes níveis de abrangência e aderência a padrões de segurança. A análise considera tanto funcionalidades criptográficas fundamentais, como cifradores e funções de *hash*, quanto capacidades avançadas de ICP, incluindo manipulação de certificados X.509, verificação de revogação e suporte a formatos de assinatura padronizados (CASTLE, 2024; OPENSOURCE PROJECT AUTHORS, 2026a).

4.3.1 Pointycastle

Pointycastle é uma biblioteca de criptografia para Dart, inspirada na biblioteca Bouncy Castle para Java. Ela fornece implementações de diversos algoritmos, incluindo cifradores simétricos e assimétricos, funções de *hash*, MACs, derivação de chaves (KDFs) e geradores de números aleatórios seguros. A biblioteca destaca-se por sua abrangência e confiabilidade no ecossistema Dart (CONTRIBUTORS, 2024).

Funcionalidades Implementadas:

- **Cifradores Simétricos:** Implementações de AES, DES, 3DES, RC2, Salsa20 e ChaCha20, permitindo a criptografia e descriptografia de dados em diferentes cifras.
- **Modos de Operação:** Suporte a modos de operação como CBC, ECB, CFB, OFB, CTR, GCM, CCM e IGE.
- **Cifradores Assimétricos:** Implementações de RSA (com padding PKCS1 e OAEP), ElGamal e algoritmos de Curvas Elípticas (EC).
- **Assinaturas Digitais:** Suporte para RSA, ECDSA e DSA, permitindo a criação e verificação de assinaturas digitais.
- **Funções de Hash:** Implementações de MD2, MD4, MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-3, RIPEMD, Whirlpool, Tiger e SM3, usados para a criação de resumos criptográficos e verificação de integridade.
- **MACs:** Suporte para HMAC, CMAC e Poly1305, fornecendo mecanismos para garantir a autenticidade e integridade das mensagens.
- **Derivação de Chaves (KDFs):** Implementações de PBKDF2, scrypt, HKDF e Argon2, permitindo a geração segura de chaves a partir de senhas.
- **Geradores de Chave:** Geração de pares de chaves RSA e EC, usado para a criação de chaves públicas e privadas para criptografia assimétrica e assinaturas digitais.

- **Geradores de Números Aleatórios Seguros:** Implementações de Fortuna e Block CTR Random, essenciais para a geração de números aleatórios criptograficamente seguros.
- **Parsing e Geração de ASN.1:** Implementação básica para manipulação de estruturas ASN.1, permitindo a interpretação (*parsing*) e a serialização (*encoding*) de dados estruturados conforme os padrões internacionais.

Funcionalidades de ICP:

- **Certificados X.509:** Apresenta suporte limitado para o processamento e manipulação de certificados X.509. A biblioteca é capaz de interpretar estruturas ASN.1 e gerenciar chaves públicas, entretanto, suas funcionalidades se restringem a esses aspectos.
- **Verificação de Assinaturas:** Implementa a verificação de assinaturas digitais utilizando os algoritmos RSA e ECDSA.
- **CMS/CADES:** Não dispõe de implementações diretas para os padrões CMS (*Cryptographic Message Syntax*) ou CAdES (*CMS Advanced Electronic Signatures*), limitando sua aplicabilidade em contextos que requerem assinaturas e criptografia de mensagens conforme especificações padronizadas.
- **Verificação de Revogação:** Não implementa mecanismos de verificação de revogação por meio de Listas de Certificados Revogados (LCR) ou do Protocolo de Status de Certificados Online (OCSP).

4.3.2 Cryptography

A biblioteca **Cryptography** é uma ferramenta moderna que visa fornecer uma API intuitiva e segura para criptografia em Dart (TEAM, D. C., 2024). Projetada para ser simples de usar sem abrir mão da segurança, ela oferece suporte a algoritmos criptográficos atualizados e considerados seguros. Fácil de integrar em projetos Dart, permite implementar funcionalidades de criptografia sem exigir do desenvolvedor uma compreensão profunda dos algoritmos subjacentes.

Funcionalidades Implementadas:

- **Cifradores Simétricos:** Implementações de AES (CBC, CTR, GCM), ChaCha20 e XChaCha20 para a criptografia de dados em diferentes cifras.
- **Assinaturas Digitais:** Suporte para Ed25519, ECDSA (curvas P-256, P-384, P-521), RSA-PSS e RSA-SSA-PKCS1v15.
- **Funções de Hash:** Implementações de SHA-1, SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512), Blake2b e Blake2s, aplicáveis à criação de resumos criptográficos e à verificação de integridade.

- **MACs:** Suporte para HMAC e Poly1305, que asseguram a autenticidade e a integridade das mensagens.
- **Derivação de Chaves (KDFs):** Implementações de PBKDF2, HKDF e Argon2id, voltadas à geração segura de chaves a partir de senhas ou outros materiais de entrada.
- **Troca de Chaves (*Key Exchange*):** Suporte para ECDH (*Elliptic Curve Diffie-Hellman*) e X25519.
- **Manipulação de Chaves:** Suporte para chaves públicas e privadas, incluindo geração de pares de chaves, armazenamento seguro e gerenciamento de chaves.

Funcionalidades de ICP:

- **Certificados X.509:** Não possui suporte nativo para *parsing* ou manipulação de certificados X.509, limitando sua aplicação em cenários que requerem manipulação direta de certificados.
- **Verificação de Assinaturas:** Suporta verificação de assinaturas digitais usando Ed25519, ECDSA e RSA, permitindo a validação da autenticidade de mensagens e documentos.
- **CMS/CAdES:** Não possui implementação direta para CMS ou CAdES, restringindo sua aplicação em cenários que exigem assinatura e criptografia de mensagens padronizadas.
- **Verificação de Revogação:** Não implementa verificação de revogação via LCR ou OCSP.

4.3.3 Crypto

A biblioteca **Crypto** é uma coleção de funções de *hash* e HMAC para Dart (TEAM, D., 2024a). Ela fornece implementações básicas de algoritmos de *hash*, sendo adequada para necessidades simples de *hashing* e HMAC. Leve e fácil de integrar, atende projetos Dart que precisam apenas de resumos criptográficos e autenticação de mensagens.

Funcionalidades Implementadas:

- **Funções de *Hash*:** Implementações de MD5, SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 e SHA-512/256, permitindo a criação de resumos criptográficos de diferentes tamanhos e níveis de segurança.
- **HMAC:** Suporte para HMAC com os algoritmos de *hash* disponíveis, o que garante autenticidade e integridade às mensagens.

Funcionalidades de ICP:

- **Certificados X.509:** Não possui suporte para certificados, o que a torna inadequada para cenários que envolvem manipulação ou verificação de certificados digitais.

- **Verificação de Assinaturas:** Não implementa mecanismos de assinatura ou verificação, restringindo sua utilização a apenas funções de *hash* e autenticação de mensagens.
- **CMS/CAdES:** Não possui implementação, o que impede seu uso em cenários que requerem criptografia ou assinatura de mensagens padronizadas.
- **Verificação de Revogação:** Não implementa verificação de revogação via LCR ou OCSP.

Diante desse quadro, evidencia-se a necessidade de uma análise comparativa sistemática que quantifique as lacunas identificadas e oriente as decisões de projeto da nova biblioteca, tarefa conduzida na seção a seguir.

4.4 COMPARAÇÃO DE BIBLIOTECAS

A Tabela 6 apresenta uma comparação unificada das funcionalidades oferecidas pelas bibliotecas de criptografia analisadas. As cinco bibliotecas são dispostas lado a lado, permitindo uma visualização imediata das lacunas existentes nas bibliotecas Dart em relação às bibliotecas de referência, especialmente nas áreas de ICP, verificação de revogação e formatos de assinatura padronizados. Utilizam-se os símbolos ✓ para funcionalidade presente, × para ausente e ~ para suporte parcial.

Tabela 6 – Comparação unificada de funcionalidades entre bibliotecas criptográficas de referência e bibliotecas Dart.

Funcionalidade	Bouncy Castle	OpenSSL	PointyCastle	Cryptography	Crypto
Cifradores Simétricos	✓	✓	✓	✓	×
Cifradores Assimétricos	✓	✓	✓	✓	×
Algoritmos Pós-Quânticos (ML-KEM/ML-DSA)	✓	✓	×	×	×
Modos de Operação	✓	✓	✓	✓	×
Assinaturas Digitais	✓	✓	✓	✓	×
Funções de <i>Hash</i>	✓	✓	✓	✓	✓
MACs (HMAC, CMAC e outros)	✓	✓	✓	✓	✓
Derivação de Chaves (KDFs)	✓	✓	✓	✓	×
Troca de Chaves	✓	✓	✓	✓	×
PRNGs Seguros	✓	✓	✓	×	×
<i>Parsing</i> /Geração ASN.1	✓	✓	~	×	×
Certificados X.509	✓	✓	~	×	×
Verificação de Certificados	✓	✓	×	×	×
Verificação de Revogação (LCR/OCSP)	✓	✓	×	×	×
CMS	✓	✓	×	×	×
CAdES	✓	✓	×	×	×
Protocolos de Segurança (TLS)	✓	✓	×	×	×
Certificados Autoassinados	✓	✓	×	×	×
Extensões X.509	✓	✓	×	×	×
Suporte a CSRs	✓	✓	×	×	×

✓ = presente × = ausente ~ = suporte parcial

Fonte: Elaborado pelo autor.

A análise comparativa evidencia três conclusões principais. Em primeiro lugar, as bibliotecas de referência Bouncy Castle e OpenSSL oferecem cobertura completa de todas

as funcionalidades avaliadas, incluindo suporte a certificados X.509, verificação de revogação, CMS/CAdES e protocolos de segurança. Trata-se de capacidades indispensáveis para aplicações que demandam conformidade com padrões internacionais de ICP. Em segundo lugar, as bibliotecas Dart existentes suprem adequadamente as primitivas criptográficas fundamentais, cifradores, funções de *hash* e assinaturas digitais, mas apresentam lacunas críticas justamente nas funcionalidades de ICP: nenhuma delas implementa verificação de revogação (LCR/OCSP), suporte a CMS/CAdES ou validação completa de cadeias de certificados.

Por fim, mesmo a Pointycastle, que é a mais abrangente das bibliotecas Dart, oferece apenas suporte parcial a ASN.1 e certificados X.509, sem contemplar os mecanismos de validação e os formatos de interoperabilidade exigidos por sistemas reais. O contraste se repete no campo pós-quântico: enquanto Bouncy Castle e OpenSSL já incorporam os algoritmos padronizados pelo NIST (ML-KEM e ML-DSA), nenhuma das três bibliotecas Dart avaliadas oferece qualquer suporte a essa categoria, o que amplia a defasagem do ecossistema diante da transição para a criptografia resistente a ataques quânticos.

Essa lacuna justifica a necessidade de uma nova biblioteca que combine a abrangência funcional de Bouncy Castle e OpenSSL com a facilidade de uso do ecossistema Dart. A estratégia adotada para alcançar esse objetivo, a integração via FFI com código nativo, constitui o tópico da seção a seguir. Essa abordagem elimina a necessidade de reimplementar algoritmos criptográficos em Dart, herdando diretamente a maturidade, a auditabilidade e a cobertura algorítmica do OpenSSL.

4.5 FOREIGN FUNCTION INTERFACE (FFI)

A fundamentação teórica da Foreign Function Interface foi apresentada na Seção 2.7. Esta seção concentra-se nos aspectos de implementação específicos da integração entre Dart e o OpenSSL no contexto da `dartcryptosec`. São abordados o funcionamento da arquitetura FFI e seu impacto no desempenho, o padrão de declaração dos *bindings*, o carregamento multiplataforma e o gerenciamento de memória nativa, elementos que, em conjunto, constituem o núcleo técnico da biblioteca.

4.5.1 Funcionamento e Arquitetura

O subsistema `dart:ffi` na `dartcryptosec` viabiliza a invocação de funções nativas do OpenSSL diretamente a partir de código Dart, estruturando-se em três elementos arquiteturais interdependentes. O primeiro elemento consiste nas funções nativas expostas em bibliotecas dinâmicas (`.so` no Linux/Android, `.dylib` no macOS/iOS, `.dll` no Windows), referenciadas pelo Dart por meio da classe `DynamicLibrary`, que realiza o carregamento em tempo de execução. O segundo elemento diz respeito à tipagem de cada função nativa no lado Dart, por meio de assinaturas que mapeiam tipos C, como `Int32` e

`Pointer<Utf8>`, para seus equivalentes Dart, garantindo segurança de tipos na fronteira entre as duas linguagens. O terceiro elemento refere-se à decisão de empregar *bindings* escritos manualmente, em detrimento da ferramenta `dart:ffigen`, conforme detalhado na Seção 4.5.2.

A arquitetura adotada neste trabalho estabelece sete camadas, descritas em detalhe na Seção 4.6, estendendo-se desde a biblioteca nativa OpenSSL até a aplicação de demonstração, com a FFI realizando chamadas diretas ao OpenSSL sem código nativo intermediário adicional.

No tocante ao desempenho, o uso de FFI introduz uma sobrecarga de transição entre a máquina virtual do Dart e o código nativo, além do custo de conversão ou cópia dos dados utilizados nas chamadas. Em operações criptográficas intensivas em processamento, essa sobrecarga representa apenas uma parte do tempo total. O impacto observado nas medições do projeto é analisado no Capítulo 5, distinguindo operações de baixa latência daquelas dominadas pelo processamento nativo.

4.5.2 Declaração de *Bindings*: Padrão Triplo

Cada função do OpenSSL exposta à camada Dart por meio do subsistema `dart:ffi` segue um padrão triplo de declaração, utilizado para organizar o mapeamento entre as assinaturas C e Dart. O padrão compõe-se de três artefatos para cada *binding*:

1. **Typedef Native (C)**: declara a assinatura da função exatamente como ela aparece no cabeçalho C do OpenSSL, utilizando os tipos nativos mapeados pelo `dart:ffi`, `Int32` para `int`, `Pointer<Int8>` para `char*`, `Pointer<Void>` para ponteiros opacos e `Size` para `size_t`. Esse *typedef* é anotado com o tipo `NativeFunction` e a convenção de chamada padrão da plataforma.
2. **Typedef Dart**: declara a assinatura da função como ela será invocada pelo código Dart. A diferença em relação ao *typedef* nativo reside na conversão de `Pointer<NativeType>` para tipos mais expressivos quando apropriado, por exemplo, `Pointer<Utf8>` em lugar de `Pointer<Int8>` para parâmetros que representam cadeias de caracteres.
3. **late final binding**: a variável que armazenará a referência à função nativa, inicializada tardiamente via `dynamicLibrary.lookup()`, que recebe o nome da função como argumento. O modificador `late final` assegura que a resolução do *binding* ocorra uma única vez, no primeiro acesso, e que o resultado seja imutável durante o restante do ciclo de vida da aplicação.

A adoção sistemática desse padrão para os *bindings* facilita a identificação de discrepâncias entre as assinaturas Dart e C antes da execução das operações criptográficas, reduzindo o risco de erros difíceis de diagnosticar na fronteira FFI.

O arquivo de integração contém 189 *bindings* escritos manualmente, e não gerados pela ferramenta `dart:ffigen`. Essa escolha não decorre de uma limitação funcional da ferramenta, mas do recorte do projeto: a API pública da `dartcryptosec` utiliza apenas um subconjunto pequeno e controlado do OpenSSL, enquanto os cabeçalhos oficiais expõem milhares de símbolos, macros e estruturas condicionais. A geração automática exigiria filtros extensos para evitar a inclusão de símbolos não utilizados e ainda demandaria revisão manual das regras de propriedade de memória, especialmente em estruturas opacas como `EVP_PKEY`, `X509` e `CMS_ContentInfo`. Ao escrever os *bindings* manualmente, cada função incluída permanece associada a uma operação efetivamente exposta pela biblioteca, e sua assinatura pode ser auditada junto ao código que aloca, usa e libera os recursos nativos.

Como contrapartida, a inclusão ou atualização de funções exige a revisão manual dos tipos nativos, das conversões e das regras de liberação de memória. Por essa razão, a avaliação futura do `ffigen` permanece relevante, desde que acompanhada de filtros de geração e de testes de equivalência em relação aos *bindings* atuais.

4.5.3 *Bindings* por Subsistema

Os *bindings* definidos em `openssl_bindings.dart` são agrupados por domínio funcional. A Tabela 7 apresenta sua distribuição revisada. Como algumas funções auxiliares são utilizadas por mais de um domínio, os grupos representam uma classificação funcional e não uma partição estritamente disjunta do arquivo.

O tipo `EVP_PKEY` é o contêiner opaco e genérico empregado pelo OpenSSL para representar chaves públicas e privadas de diferentes algoritmos. A aplicação não acessa seus campos internos; ela mantém apenas um ponteiro e utiliza as funções da API EVP para gerar, importar, usar e liberar a chave. Assim, referências a *PKEY* na tabela designam o ciclo de vida desses objetos `EVP_PKEY`.

Tabela 7 – Principais grupos funcionais vinculados à API do OpenSSL.

Grupo	Finalidade	Funções
Inicialização e diagnóstico	Versão, provedor, geração aleatória, tratamento de erros e liberação genérica	8
Hash e cifra simétrica	Funções EVP para resumo criptográfico e operações AES	23
Chaves assimétricas	Ciclo de vida de <code>EVP_PKEY</code> e geração de chaves	10
Assinatura e cifração assimétrica	Assinatura, verificação, RSA-OAEP e ML-KEM, incluindo inicialização	13
Entrada, saída e codificação	BIO, BIGNUM, PEM, DER, ASN.1 e objetos	28
Certificados X.509	Criação, leitura, repositórios de confiança e interpretação de extensões	50
CMS e CAdES	Assinatura, verificação, cifração e manipulação de atributos assinados	12
LCR e requisições de certificação	Manipulação de LCR e requisições PKCS#10	25
OCSP	Construção, serialização e verificação de requisições e respostas OCSP	18
Total de <i>bindings</i> no arquivo	Contagem direta de declarações late final	189

Fonte: Elaborado pelo autor.

A classificação funcional contabiliza 187 *bindings* atribuídos aos grupos da tabela. Duas funções genéricas de pilha, compartilhadas entre diferentes domínios, não são atribuídas a um grupo exclusivo. A contagem direta do arquivo permanece em 189 *bindings*. A revisão também evidencia que certificados X.509, LCR, OCSP e requisições de certificação representam parcela significativa da integração, coerente com o foco da biblioteca em operações de ICP.

A camada FFI não procura reproduzir toda a API do OpenSSL. Foram vinculadas apenas as funções necessárias às operações oferecidas pela biblioteca, priorizando a API EVP e interfaces de alto nível, como X.509, CMS e OCSP. Esse recorte evita dependência de funções legadas ou de baixo nível e torna o código de integração mais simples de revisar e manter (OPENSSL PROJECT AUTHORS, 2026b).

Muitas estruturas internas do OpenSSL são tratadas pelo Dart apenas como ponteiros opacos. Um ponteiro opaco é uma referência a uma estrutura de dados cuja organização interna não é exposta ao código que a utiliza: conhece-se apenas o endereço de memória do objeto, e não a disposição de seus campos. Esse mecanismo é idiomático no próprio OpenSSL, que oculta deliberadamente a definição de tipos como `EVP_PKEY` e `X509` atrás de declarações incompletas, obrigando o consumidor a manipulá-los exclusivamente por meio das funções da API.

Na camada Dart, esses tipos são representados de maneira uniforme por `Pointer<Void>` e transportados de uma chamada a outra sem qualquer interpretação de seu conteúdo.

O arquivo de *bindings* declara 42 tipos dessa natureza, entre eles contextos EVP, chaves, certificados e estruturas CMS. Essa abordagem permite criar, utilizar e liberar tais objetos sem expor sua representação interna em C, preservando o encapsulamento e reduzindo o acoplamento entre o código Dart e os detalhes de implementação da biblioteca nativa. A organização dos *bindings* por grupos funcionais, por sua vez, facilita a localização das declarações e a inclusão de novas operações.

4.5.4 Carregamento Multiplataforma

Antes de executar uma operação criptográfica, a biblioteca precisa localizar e carregar a versão adequada do OpenSSL. Esse processo varia entre sistemas operacionais: no Linux, a biblioteca nativa é carregada a partir dos arquivos instalados no sistema; no Android, ela é distribuída junto ao aplicativo em versões compiladas para cada arquitetura de processador.

Essa diferença é tratada pelo arquivo `native_loader.dart`, que concentra a lógica específica de cada plataforma. As demais camadas utilizam a mesma interface independentemente do ambiente de execução. Com isso, mudanças no processo de carregamento permanecem isoladas e não afetam as operações criptográficas.

4.5.5 Gerenciamento de Memória Nativa

Dart gerencia automaticamente a memória de seus próprios objetos, mas não controla os recursos criados pelo OpenSSL. Quando uma função nativa cria um contexto, uma chave ou um certificado, a camada FFI também precisa garantir sua liberação após o uso. Caso contrário, a aplicação pode acumular memória não utilizada ou acessar recursos que já foram liberados.

Para reduzir esse risco, as operações mantêm a alocação e a liberação dos recursos próximas no código. Blocos `try/finally` asseguram que a limpeza seja executada mesmo quando ocorre uma falha durante a operação criptográfica. Funções auxiliares também centralizam padrões recorrentes de conversão e liberação, reduzindo a repetição dessa lógica nas camadas superiores. Esse cuidado permite que a API pública apresente objetos comuns do Dart enquanto os detalhes de memória nativa permanecem restritos à camada de integração.

4.6 ARQUITETURA DA BIBLIOTECA PROPOSTA

4.6.1 Arquitetura Geral

A arquitetura da biblioteca `dartcryptosec` é organizada em sete camadas, que se estendem desde a biblioteca nativa OpenSSL até a aplicação de demonstração. A comunicação entre camadas segue uma direção bem definida: `OpenSSL` → `FFI` → `Operations` → `Models` → `Flows` → `Facade` → `App`.

1. **OpenSSL**, biblioteca nativa em C que provê as primitivas criptográficas e é carregada conforme a plataforma de execução.
2. **FFI**, camada de *Foreign Function Interface* composta pelos *bindings* manuais que representam funções e estruturas opacas do OpenSSL no lado Dart.
3. **Operations**, classes que encapsulam as chamadas FFI por domínio criptográfico, convertem resultados nativos em tipos Dart e gerenciam o ciclo de vida dos recursos utilizados.
4. **Models**, classes de domínio que representam os resultados das operações criptográficas: `CryptoResult<T>` (tipo selado para sucesso ou falha tipada), `KeyPair` (par de chaves pública e privada) e `CertificateData` (dados extraídos de certificados X.509).
5. **Flows**, camada que organiza casos de uso de maior nível, como geração de chaves, assinatura de arquivos, criação de certificados e validação de cadeias. Esses fluxos coordenam operações das camadas inferiores e oferecem interfaces específicas para cada domínio.
6. **Facade**, classe `PluginCryptoAPI`, que expõe uma API unificada ao desenvolvedor e oculta a coordenação das camadas inferiores.
7. **App**, aplicação de demonstração `tcc_test_app`, utilizada na validação manual e nos testes de integração.

Essa organização mantém as particularidades da integração nativa nas camadas inferiores e permite testar separadamente operações e casos de uso. A `dartcryptosec` realiza chamadas FFI diretamente ao OpenSSL, sem código nativo intermediário adicional. Essa decisão reduz o número de componentes mantidos no projeto, mas transfere para os *bindings* Dart a responsabilidade pelo mapeamento de tipos e pelo gerenciamento de memória.

Além dessas camadas, o projeto contém componentes transversais de métricas e utilitários. Os módulos de métricas apoiam a avaliação de desempenho, concorrência e comportamento da memória, sem participar da API pública. Os utilitários concentram tarefas recorrentes de serialização, conversão e tratamento de erros utilizadas por diferentes operações.

4.6.2 Padrões de Projeto

O design da `dartcryptosec` emprega um conjunto de padrões de projeto cuja escolha é motivada por necessidades concretas da arquitetura em Dart. A classe `PluginCryptoAPI` materializa o padrão **Facade**, simplificando operações complexas, como a validação completa de uma cadeia de certificados, em uma única chamada de método. Essa abordagem oculta a complexidade interna das camadas de `Operations` e `Flows` e reduz a curva de aprendizado para o desenvolvedor.

Na camada de *Flows*, cada classe representa um caso de uso da biblioteca e coordena as operações necessárias para executá-lo. A geração de uma chave, por exemplo, seleciona a implementação adequada e aciona as funções criptográficas correspondentes. Já a validação de uma cadeia reúne a preparação do repositório de confiança e a verificação dos certificados. Essa organização evita que a API pública precise conhecer os detalhes de cada etapa e permite testar os fluxos de forma isolada.

Para a criação de objetos criptográficos (chaves, certificados, assinadores) a partir de dados brutos, emprega-se o padrão **Factory** na camada de Operations, desacoplando o código cliente dos detalhes de construção de cada tipo criptográfico. Essa centralização é particularmente importante no contexto de ASN.1, em que um mesmo *blob* binário pode corresponder a diferentes estruturas (um certificado, uma LCR, uma chave privada), e a decisão sobre qual objeto instanciar deve ser tomada com base na interpretação do conteúdo. Sem o Factory, cada ponto de código que recebe dados criptográficos precisaria conter lógica de decisão duplicada, aumentando a probabilidade de erros e dificultando a manutenção.

A intercambialidade dos algoritmos de assinatura, hashing e criptografia é obtida por meio do padrão **Strategy**. Ao definir uma interface comum para cada família de algoritmos na camada de Operations, o sistema permite que o algoritmo concreto seja selecionado em tempo de execução, por exemplo, alternando entre RSA e ECDSA para assinatura sem alterar o código que solicita a operação. Essa flexibilidade é indispensável em uma biblioteca que almeja conformidade com padrões internacionais, pois diferentes perfis de certificação exigem diferentes combinações de algoritmos e tamanhos de chave.

Em pontos específicos da camada de Operations, aplica-se o padrão **Singleton** para garantir que a inicialização do OpenSSL ocorra uma única vez e de forma controlada. Esse cuidado evita condições de corrida na configuração dos *providers* criptográficos e assegura que todos os componentes compartilhem o mesmo estado de inicialização.

Alguns princípios SOLID orientam essa organização. A separação das operações por domínio limita a responsabilidade de cada componente, enquanto interfaces como **CryptoContext**, **KeyCreator** e **ChainVerifier** reduzem a dependência direta dos casos de uso em relação às implementações concretas. A fábrica de criadores de chave também permite incluir novas especificações sem alterar a API utilizada pelas camadas superiores.

O padrão **Builder** é empregado na classe **CertificateBuilder**. A criação programática de certificados X.509 envolve campos interdependentes, como emissor, sujeito, validade, chave pública e extensões. A interface encadeável organiza essa configuração antes de delegar a criação do certificado ao OpenSSL.

As **Classes Seladas** (*sealed classes*) são utilizadas em **CryptoResult<T>** e **KeySpec**. O primeiro representa resultados de sucesso ou falha tipada; o segundo representa as especificações aceitas pela geração de chaves. Em expressões **switch**, essa modelagem permite ao compilador verificar se todos os subtipos conhecidos foram considerados.

As camadas de Operations e Flows interagem conforme a necessidade de cada caso de uso. A geração de pares de chaves, por exemplo, aciona a operação assimétrica correspondente e retorna um `CryptoResult<KeyPair>`. A verificação de certificados pode envolver operações encadeadas de leitura, consulta de revogação e validação de assinatura. A Figura 6 apresenta o fluxo principal entre as sete camadas.

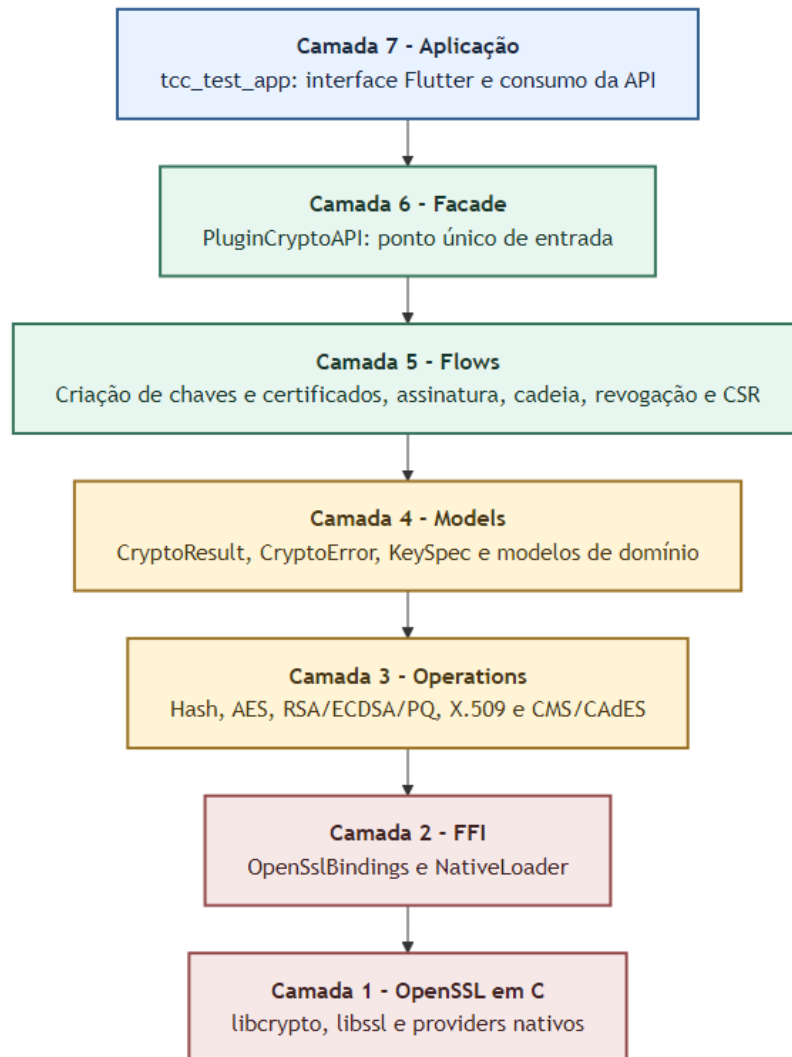


Figura 6 – Fluxo principal da arquitetura em sete camadas da `dartcryptosec`. Os componentes transversais de métricas e utilitários não são representados.

Fonte: Elaborado pelo autor.

Na implementação, a classe `PluginCryptoAPI` atua como ponto de entrada da biblioteca e delega o trabalho às classes da camada de operações. Processos que envolvem várias etapas são organizados na camada de *Flows*. É o caso da criação de certificados, da assinatura de arquivos e da validação de cadeias. Os resultados e erros retornados por essas operações são representados pelos modelos de domínio, evitando que ponteiros e códigos de erro do OpenSSL sejam expostos ao código cliente.

4.6.3 Considerações de Segurança

A `dartcryptosec` adota cuidados específicos no tratamento de dados criptográficos. A liberação dos recursos nativos é realizada de forma explícita, e as operações expostas priorizam algoritmos e parâmetros compatíveis com os padrões considerados no trabalho. Os erros retornados pelo OpenSSL são convertidos em falhas compreensíveis pela camada Dart, o que facilita sua identificação e evita que retornos inválidos sejam tratados como resultados bem-sucedidos. Esses mecanismos reduzem riscos de implementação, mas não substituem auditorias de segurança nem comprovam resistência a ataques de canais laterais.

4.6.4 Modelo de Ameaças STRIDE

A análise de segurança da `dartcryptosec` foi conduzida por meio da metodologia STRIDE (Kohnfelder & Garg, 1999; Microsoft SDL, Howard & Lipner, 2006), que organiza ameaças em seis categorias relacionadas a propriedades de segurança: **S**poofing (autenticidade), **T**ampering (integridade), **R**epudiation (não repúdio), **I**nformation Disclosure (confidencialidade), **D**enial of Service (disponibilidade) e **E**levation of Privilege (autorização).

A aplicação do STRIDE seguiu três passos. Primeiro, delimitou-se a superfície de ataque da biblioteca, separando a API Dart pública, a fronteira FFI, os objetos nativos do OpenSSL e os artefatos externos carregados em tempo de execução, como `libcrypto.so` e provedores. Em seguida, cada fluxo relevante foi examinado segundo as seis categorias STRIDE, identificando ameaças associadas a substituição de bibliotecas, entradas ASN.1/CMS malformadas, persistência de material sensível em memória, omissão de consultas de revogação e variações temporais observáveis. Por fim, cada ameaça recebeu pontuações de probabilidade e impacto, permitindo priorizar mitigação, monitoramento ou aceitação documentada.

A aplicação da metodologia ao artefato produziu um catálogo de 15 ameaças (T1–T15), cada qual avaliada quanto à probabilidade (P, escala 1–5), ao impacto (I, escala 1–5) e ao risco resultante ($R = P \times I$, escala 1–25). As ameaças foram classificadas em quatro níveis de prioridade: P1 (risco ≥ 20 , ação imediata), P2 (risco 12–19, mitigação no próximo ciclo), P3 (risco 6–11, monitoramento) e P4 (risco 1–5, risco aceito).

A Tabela 8 apresenta as ameaças de prioridade P2 e P3, as mais relevantes para o estágio atual de maturidade da biblioteca.

Tabela 8 – Ameaças STRIDE de prioridade elevada (P2–P3) identificadas na `dartcryptosec`.

ID	Ameaça	Categoria	P	I	R
T2	Crash via CMS/PKCS#7 malformado	DoS	3	4	12
T4	Persistência de chaves em memória	Info. Disclosure	3	4	12
T15	Omissão da consulta de revogação pela aplicação consumidora	Spoofing	3	4	12
T1	Substituição de <code>libcrypto.so</code>	Spoofing	2	5	10
T7	Vazamento temporal (timing)	Info. Disclosure	2	5	10
T10	Buffer overflow na fronteira FFI	Tampering	2	5	10
T3	Spoofing de certificado X.509	Spoofing	3	3	9
T6	Ausência de auditoria	Repudiation	3	3	9

Fonte: Elaborado pelo autor.

Cada ameaça foi analisada quanto ao vetor de ataque, às pré-condições necessárias para exploração e às contramedidas existentes ou planejadas. Como resultado, as ameaças de maior prioridade orientaram decisões concretas da implementação: limpeza explícita da fila de erros do OpenSSL, tratamento tipado de falhas, validação de ponteiros nulos, separação entre carregamento nativo e operações criptográficas, testes adversariais com entradas malformadas e registro explícito das limitações de consulta de revogação. O modelo completo, incluindo as 15 ameaças com justificativas detalhadas, encontra-se documentado no Apêndice B.

4.6.5 Hierarquia de Erros Criptográficos

A biblioteca adota uma hierarquia selada de `CryptoError`, permitindo que diferentes condições de falha sejam tratadas de forma tipada pelo código cliente, sem depender da comparação de cadeias de caracteres ou códigos numéricos. A Tabela 9 apresenta os principais subtipos utilizados nas operações abordadas neste trabalho.

Tabela 9 – Hierarquia de subtipos de `CryptoError`.

Subtipo	Condição de Falha	Origem
<code>KeygenError</code>	Falha na geração de par de chaves	Falha em <code>EVP_PKEY_keygen</code>
<code>CertificateError</code>	Falha na criação de certificado X.509	Falha no construtor ou nas funções X.509
<code>FileSigningError</code>	Falha na assinatura de arquivo (CMS/CAdES)	Falha em <code>CMS_sign</code> com BIO
<code>ValidationError</code>	Parâmetro de entrada inválido	Validação de argumentos na camada Dart
<code>ChainValidation-Error</code>	Falha na validação de cadeia X.509	Falha em <code>X509_verify_cert</code>
<code>CrlError</code>	Falha em operação de LCR	Falha na leitura ou verificação da LCR
<code>X509ExtensionError</code>	Falha na leitura de extensão X.509 v3	Extensão desconhecida ou malformada
<code>OcspError</code>	Falha em operação de OCSP	Falha na construção ou verificação OCSP
<code>Asn1Error</code>	Falha na leitura de ASN.1 (DER/PEM)	Dados malformados ou corrompidos
<code>AesGcmAuthFailure</code>	Falha na autenticação GCM	Texto cifrado adulterado ou chave incorreta
<code>CsrError</code>	Falha na geração de CSR (PKCS#10)	Falha nas funções <code>X509_REQ_*</code>
<code>TimestampError</code>	Falha em operação de marca temporal	Falha nas funções <code>TS_REQ_*</code>

Fonte: Elaborado pelo autor.

Cada subtipo de `CryptoError` encapsula a mensagem de erro original do OpenSSL, obtida por meio de `ERR_error_string()`, e o código de erro numérico, preservando a rastreabilidade completa da falha. O uso de classes seladas (*sealed classes*) do Dart 3 garante que o compilador verifique exaustivamente todos os casos em expressões `switch`, eliminando a possibilidade de condições de erro não tratadas em tempo de compilação.

A função de tradução `mapExceptionToCryptoError()` atua como ponte entre o modelo de exceções do Dart e os códigos de erro do OpenSSL, assegurando que toda falha na fronteira FFI seja convertida para o subtipo apropriado. A chamada sistemática de `ERR_clear_error()` antes de cada novo contexto criptográfico, aplicada em 12 pontos críticos do código, previne a corrupção da fila de erros do OpenSSL e garante que o diagnóstico de cada operação seja preciso e independente de operações anteriores.

Parte das operações de baixo nível ainda sinaliza falhas por meio de exceções, enquanto os casos de uso mais recentes retornam `CryptoResult<T>`. A função `mapExceptionToCryptoError()` permite a convivência entre essas abordagens ao converter exceções em erros tipados. Essa solução preserva a compatibilidade da implementação atual, mas também evidencia uma transição arquitetural ainda não concluída.

4.7 INTEGRAÇÃO DOS ALGORITMOS PÓS-QUÂNTICOS

A fundamentação do ML-KEM e do ML-DSA foi apresentada na Seção 2.6. Aqui são descritas apenas as alterações necessárias para disponibilizar esses algoritmos por meio da API Dart.

4.7.1 Integração Arquitetural

A integração aproveitou as interfaces genéricas da API EVP e exigiu alterações em três partes da biblioteca:

1. **FFI**: inclusão das funções de encapsulamento e desencapsulamento utilizadas pelo ML-KEM. O ML-DSA reutiliza as funções genéricas de assinatura e verificação da API EVP.
2. **Operations**: inclusão das operações de geração de chaves, encapsulamento, desencapsulamento, assinatura e verificação.
3. **Models**: criação de especificações de chave próprias para selecionar as variantes do ML-KEM e do ML-DSA.

As camadas superiores continuaram utilizando os mesmos fluxos de geração de chaves e assinatura. As particularidades dos novos algoritmos permaneceram concentradas nas camadas de integração e operações.

As operações de encapsulamento e desencapsulamento do ML-KEM seguem o padrão de duas chamadas adotado por APIs nativas que produzem saídas de tamanho variável. A primeira chamada consulta os tamanhos necessários; em seguida, a camada Dart aloca os *buffers* e repete a operação para preenchê-los. Essa estratégia evita assumir tamanhos fixos na fronteira FFI.

4.7.2 ML-DSA e o *Digest* Nulo

A integração do ML-DSA revelou uma peculiaridade da API EVP que exigiu tratamento específico. Diferentemente dos esquemas de assinatura clássicos, como RSA e ECDSA, nos quais a função de *hash* é configurada separadamente no contexto de assinatura, o ML-DSA realiza internamente o processamento definido pelo FIPS 204 (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024a). Por esse motivo, a inicialização da assinatura não recebe um *digest* separado. Essa diferença é tratada na camada de operações, evitando que a particularidade da API nativa seja exposta às camadas superiores.

As variantes disponíveis são representadas por valores tipados na API Dart e traduzidas internamente para os identificadores esperados pelo OpenSSL. Os tamanhos e níveis de segurança correspondentes foram apresentados na fundamentação teórica e não são repetidos nesta seção.

4.8 DESAFIOS TÉCNICOS

O desenvolvimento de uma biblioteca que integra Dart e C por meio do OpenSSL apresentou desafios relacionados à memória nativa, à declaração de *bindings* e à configuração do ambiente criptográfico. Esta seção descreve os principais problemas encontrados e as soluções adotadas durante a implementação.

4.8.1 Tratamento da *Tag* GCM

O modo GCM do AES provê *Authenticated Encryption with Associated Data* (AEAD), combinando confidencialidade e autenticidade em uma única operação. A autenticidade é garantida por uma *tag* de autenticação, tipicamente de 128 bits, que o OpenSSL calcula durante a cifração e verifica durante a decifração por meio da função `EVP_CIPHER_CTX_ctrl`.

Na integração via FFI, a *tag* precisa ser configurada no contexto nativo antes da finalização da decifração. O GCM admite *tags* com tamanhos variáveis, mas a implementação deste trabalho adota 16 bytes (128 bits), tamanho recomendado e mais comum para preservar a margem máxima de autenticação. A restrição, portanto, é uma decisão da API da `dartcryptosec` sobre o modo como expõe o OpenSSL, não uma impossibilidade absoluta do algoritmo. A biblioteca transfere o valor ao OpenSSL por meio de `EVP_CIPHER_CTX_ctrl` e considera a operação válida somente quando `EVP_DecryptFinal_ex` confirma a autenticação. Esse tratamento impede que uma falha de autenticação seja confundida com uma decifração bem-sucedida e mantém a validação concentrada na camada de operações.

A Figura 7 resume a sequência de chamadas dessa operação de cifração AES-256-GCM, evidenciando como a requisição percorre as camadas *Facade*, *Operations* e FFI até as funções da API EVP do OpenSSL, retornando o *ciphertext* e a *authentication tag* encapsulados no modelo `AesGcmResult`.

4.8.2 *Parsing* de `ASN1_TIME`

O padrão X.509 admite formatos distintos para a representação das datas de validade dos certificados (COOPER *et al.*, 2008). Para evitar que a camada Dart dependa da estrutura interna de `ASN1_TIME`, a implementação solicita ao OpenSSL sua representação textual por meio de `ASN1_TIME_print` e converte o resultado para `DateTime`. Essa solução preserva o uso de ponteiros opacos, mas cria dependência do formato textual produzido pela biblioteca nativa. Por isso, entradas inesperadas resultam em valor nulo, e a conversão precisa ser exercitada nos testes de certificados.

Passo	Camada	API / Método	Descrição
1	Facade	PluginCryptoAPI.aes256GcmEncrypt()	Ponto de entrada da operação
2	Operations	AesOperations._gcmCipherOp()	Lógica de negócio
3	FFI	EVP_CIPHER_CTX_new()	Aloca contexto EVP
4	FFI	EVP_EncryptInit_ex()	Inicializa AES-256-GCM, chave e IV
5	FFI	EVP_EncryptUpdate() (AAD)	Processa dados autenticados adicionais
6	FFI	EVP_EncryptUpdate() (dados)	Cifra <i>plaintext</i> em <i>ciphertext</i>
7	FFI	EVP_EncryptFinal_ex()	Finaliza a cifração
8	FFI	EVP_CIPHER_CTX_ctrl()	Obtém <i>authentication tag</i> (16 bytes)
9	FFI	EVP_CIPHER_CTX_free()	Libera contexto (par com passo 3)
10	Models	AesGcmResult	Retorna <i>ciphertext</i> e <i>tag</i>

Figura 7 – Sequência de chamadas para a operação AES-256-GCM, atravessando as camadas *Facade*, *Operations*, FFI e *Models*, com as funções da API EVP do OpenSSL invocadas na camada FFI.

Fonte: Elaborado pelo autor.

4.8.3 Pilhas de Certificados em Operações CMS

Algumas operações CMS recebem conjuntos de certificados representados pelo OpenSSL como pilhas de ponteiros. A camada Dart precisa construir essas pilhas, inserir cada certificado e liberar corretamente tanto o contêiner quanto os demais recursos envolvidos. O ponto mais delicado é a propriedade dos objetos: após a inserção, o ciclo de vida do certificado pode diferir do ciclo de vida da pilha que o referencia. A implementação concentra essa montagem em funções auxiliares e mantém explícitas as regras de liberação, reduzindo o risco de vazamentos e liberações duplicadas.

4.8.4 Compilação e Carregamento do Provedor FIPS

O suporte aos algoritmos pós-quânticos exigiu mais do que a inclusão de novos *bindings*. O OpenSSL precisava ser compilado com o provedor FIPS disponível e configurado para carregá-lo durante a execução. Sem essa configuração, os identificadores de ML-KEM e ML-DSA não eram resolvidos, embora as funções correspondentes estivessem presentes na API EVP.

A preparação desse ambiente apresentou diferenças entre Linux e Android. No Linux, o provedor e seus arquivos de configuração podiam ser instalados junto à biblioteca. No Android, esses artefatos precisaram ser empacotados com a aplicação e localizados em tempo de execução, respeitando a arquitetura do dispositivo. A solução adotada separa o carregamento do provedor das operações criptográficas, permitindo verificar sua

disponibilidade durante a inicialização e informar uma falha antes da execução de uma operação pós-quântica.

4.8.5 Limitações da Implementação e da Validação

A principal limitação arquitetural decorre do uso direto da interface C do OpenSSL. Embora essa escolha permita reutilizar suas implementações criptográficas, ela mantém a camada FFI dependente das assinaturas, da organização e do comportamento da versão nativa utilizada. Uma atualização do OpenSSL pode exigir a revisão dos *bindings*, das funções auxiliares e das regras de gerenciamento de memória.

Outra limitação está na fronteira entre a memória gerenciada pelo Dart e a memória nativa. Os padrões de liberação adotados reduzem o risco de vazamentos e acessos inválidos, mas não oferecem as mesmas garantias fornecidas pelo sistema de tipos e pelo coletor de lixo para objetos inteiramente gerenciados. A correção dessa camada depende de revisão do código e de testes que exercitem os diferentes caminhos de execução. Por isso, os resultados de memória devem ser interpretados como evidência prática de estabilidade no ambiente avaliado, não como prova exaustiva de ausência de vazamentos.

Por fim, a validação funcional foi realizada em Linux e Android. Os resultados obtidos não permitem afirmar que o comportamento será idêntico em iOS, macOS ou Windows, pois diferenças no carregamento de bibliotecas, na compilação do OpenSSL e no ambiente de execução podem revelar problemas específicos dessas plataformas. A ampliação dessa validação é tratada como trabalho futuro no Capítulo 6.

5 RESULTADOS E VALIDAÇÃO

A apresentação dos resultados está organizada em cinco seções: a Seção 5.1 detalha os resultados dos testes unitários e de integração; a Seção 5.2 discute a cobertura de código obtida; a Seção 5.3 apresenta os *benchmarks* de desempenho das operações criptográficas; a Seção 5.4 analisa aspectos de segurança da implementação; e a Seção 5.5 estabelece uma comparação com o estado da arte em bibliotecas criptográficas para Dart.

5.1 EXECUÇÃO E ANÁLISE DOS TESTES

A suíte de testes da biblioteca `dartcryptosec` totaliza **692 testes**, distribuídos entre 515 testes unitários executados em ambiente Linux¹ e 177 testes de integração executados em dispositivo Android. Não houve falha em nenhuma das plataformas: os 515 testes de Linux e os 177 de Android foram todos aprovados, como sintetiza a Tabela 10.

Tabela 10 – Resumo geral dos testes executados.

Plataforma	Executados	Falhas
Linux (x86_64)	515	0
Android (arm64-v8a)	177	0
Total	692	0

Fonte: Elaborado pelo autor.

Os testes unitários de Linux representam 74,4% do total e os de integração de Android, os 25,6% restantes. A diferença de cerca de 3:1 entre Linux (515) e Android (177) não indica menor cobertura, mas a natureza distinta de cada suíte: em Linux, os testes são **unitários**, com granularidade fina (um caso por função ou parâmetro); em Android, são de **integração** (*end-to-end*), em que cada teste exercita um fluxo completo em dispositivo físico, agrupando operações que em Linux seriam fragmentadas em dezenas de testes. Algumas categorias, como *fuzzing*, análise estatística de aleatoriedade e medições de *timing*, permanecem restritas ao Linux por restrições de tempo, recursos ou precisão de medição.

Os testes foram conduzidos em dois ambientes controlados, cujas configurações são detalhadas na Tabela 11.

¹ O valor de 515 corresponde ao número de testes efetivamente executados pelo `runner flutter test`, fonte canônica adotada neste trabalho. A contagem estática das declarações `test()` restrita aos arquivos de zona e de suporte resulta em 495; a diferença decorre de arquivos de métricas e temporização que o `runner` também carrega e executa por não possuírem a marcação de suíte `@Tags(['metrics'])`, compensada pelos casos das zonas individualmente ignorados por marcações internas (por exemplo, `slow`).

Tabela 11 – Configuração dos ambientes de teste.

Parâmetro	Linux	Android
Sistema Operacional	Ubuntu 22.04 (WSL1)	Android 15 + HyperOS
Arquitetura	x86_64	arm64-v8a
Hardware	Avell A52i, Intel Core i7-13620H, 10 núcleos físicos, 16 lógicos, 32 GB RAM	Xiaomi 2412DPC0AI, 12 GB RAM
OpenSSL	4.0.0 (<code>enable-fips</code>)	4.0.0 (cross-compilado NDK 28)
Dispositivo	Host (WSL1)	Dispositivo físico via ADB
Framework	Flutter 3.24+	Flutter 3.24+
<i>Runner</i>	<code>flutter test</code>	<code>flutter test</code>
Conexão	—	<code>integration_test</code> WiFi ADB (192.168.1.56:5555)

Fonte: Elaborado pelo autor.

5.1.1 Distribuição dos Testes Linux

A suíte Linux é composta por 49 arquivos de teste, organizados em 40 zonas (**zone01** a **zone40**) que cobrem desde o carregador nativo FFI até testes de interoperabilidade e estabilidade de longa duração. A Tabela 12 discrimina essa distribuição por domínio funcional.

5.1.2 Distribuição dos Testes Android

Os 177 testes de integração Android estão organizados em 8 arquivos, executados diretamente no dispositivo Xiaomi 2412DPC0AI. A Tabela 13 detalha a distribuição por arquivo.

O tempo total de execução dos testes Android foi de 467 segundos (aproximadamente 7 minutos e 47 segundos), com média de 2,64 segundos por teste. O *overhead* de *build* do APK (cerca de 30 segundos) e de instalação via ADB (aproximadamente 5 segundos) por arquivo de teste representa 60% do ciclo total de testes de integração Android, conforme documentado no relatório de benchmarks formais do projeto.

5.1.3 Testes de Isolamento

No modelo de concorrência do Dart, um *isolate* é uma unidade de execução independente, com seu próprio *heap* de memória e laço de eventos, que não compartilha estado com os demais e se comunica exclusivamente por troca de mensagens. Esse isolamento elimina condições de corrida sobre memória compartilhada, mas torna a paralelização efetiva dependente da capacidade de distribuir a carga entre múltiplos *isolates*. Para avaliar a escalabilidade da biblioteca sob concorrência, testes foram conduzidos com números cres-

Tabela 12 – Distribuição dos testes Linux por domínio funcional.

Zonas	Domínio	Testes
01	Carregador nativo (FFI)	8
02–03	<i>Hash</i> (SHA-256, SHA-512, SHA3-256, SHA3-512) e geração de números aleatórios	17
04–05	AES-CBC e AES-GCM	21
06–07	RSA e ECDSA	17
08–09	X.509 (<i>parsing</i> de certificados) e CMS	14
10–12	Tratamento de erros	35
13–18	Casos de borda (AES, <i>random</i> , <i>hash</i> , RSA, ECDSA, X.509)	56
19–22	Fluxos de criação (chaves, certificados, assinatura de arquivos, PQ)	76
23–29	ICP (validação de cadeia, LCR, OCSP, CSR, extensões X.509, CAdES, ASN.1)	103
30–33	Propriedades, <i>fuzzing</i> , NIST SP800-22, NIST SP800-90B	23
34–39	<i>Timing</i> RSA, FIPS 186-4/5, diferencial CLI, <i>soak</i> , interoperabilidade, combinatorial	47
40	ICP da API pública (conformidade de interface: CAdES, LCR, OCSP, CSR e <i>timestamp</i> via <i>facade</i>)	39
Total		456

Fonte: Elaborado pelo autor.

Tabela 13 – Distribuição dos testes Android por arquivo de integração.

Arquivo de Integração	Testes	Tempo (s)	Categoria
crypto_integration_test.dart	52	66	Integração geral
crypto_flows_integration_test.dart	21	57	Fluxos completos
crypto_e2e_adversarial_test.dart	31	60	Testes adversariais
crypto_e2e_hash_pipeline_test.dart	14	51	<i>Pipeline</i> de <i>hash</i>
crypto_e2e_pipeline_test.dart	18	76	<i>Pipeline</i> criptográfico
crypto_e2e_pki_pipeline_test.dart	17	58	<i>Pipeline</i> ICP
crypto_e2e_pq_test.dart	22	52	<i>Pipeline</i> pós-quântico
crypto_e2e_pq_fips_test.dart	2	47	Testes FIPS (PQ)
Total	177	467	–

Fonte: Elaborado pelo autor.

centes de *isolates* Dart. Cada configuração cumpriu 20 iterações de uma carga de trabalho sintética, medindo a eficiência de *scaling* à medida que o número de *isolates* aumenta, e não a correção criptográfica das operações, que é coberta pelos testes funcionais. Nenhuma configuração registrou falha, como registra a Tabela 14.

Tabela 14 – Resultados dos testes de isolamento (*stress*).

Isolates	Iterações	Resultado
1	20	Aprovado
2	20	Aprovado
4	20	Aprovado
8	20	Aprovado

Fonte: Elaborado pelo autor.

Em complemento, testes combinatórios exaustivos no Linux percorreram 63 combinações de parâmetros de forma sequencial, todas aprovadas (C1: 28/28, C2: 20/20, C3: 6/6, C4: 9/9), avaliando sistematicamente as combinações de algoritmos, tamanhos de chave e funções de *hash*. Em Android, o teste de estresse H3 executou 100 gerações rápidas de chaves sem nenhuma falha; esse volume foi escolhido para exercitar repetidamente a fronteira FFI e o gerador de material criptográfico sem tornar o teste de integração proibitivamente longo no dispositivo físico. O monitoramento de uso de memória residente (RSS) é conduzido separadamente pelos testes de estresse do grupo 11 (`crypto_integration_test.dart`).

A independência do material criptográfico entre gerações foi verificada estaticamente: 100 pares RSA-2048 produzidos em sequência resultaram em 100 *fingerprints* SHA-256 distintos, e 1000 chamadas SHA-256 com entradas aleatórias distintas produziram 1000 *digests* únicos. Nos casos avaliados, portanto, as gerações produziram saídas sempre distintas.

5.1.4 Paridade Funcional entre Plataformas

A Tabela 15 sintetiza a paridade funcional observada entre Linux e Android.

Entre as operações de ICP validadas, a verificação de cadeias de certificados X.509 é uma das mais críticas. A Figura 8 detalha a sequência de operações dessa validação, conduzida pela função `X509_verify_cert()` do OpenSSL, desde a montagem do repositório de confiança até a verificação encadeada das assinaturas até a AC raiz.

Tabela 15 – Paridade funcional Linux vs. Android por domínio criptográfico.

Domínio	Linux	Android
<i>Hash</i> (SHA-256, SHA-512, SHA3)	Aprovado	Aprovado (14/14)
AES-CBC / AES-GCM	Aprovado	Aprovado
RSA (geração, assinatura, verificação)	Aprovado	Aprovado
ECDSA (P-256, P-384, P-521)	Aprovado	Aprovado
X.509 / ICP	Aprovado	Aprovado (17/17)
CMS / CAdES	Aprovado	Aprovado
ML-KEM / ML-DSA (pós-quântico)	Aprovado	Aprovado (22 testes PQ + 2 FIPS)

Fonte: Elaborado pelo autor. Dados extraídos da execução da suíte de testes no ambiente descrito no Capítulo 3.

Passo	Operação	API OpenSSL	Descrição
1	<i>Parse</i> cert. folha	d2i_X509()	Deserializa certificado DER
2	<i>Parse</i> cert. intermediário	d2i_X509()	Deserializa certificado DER
3	<i>Parse</i> cert. raiz	d2i_X509()	Deserializa certificado DER
4	Criar <i>trusted store</i>	X509_STORE_new()	Inicializa loja de confiança
5	Adicionar âncora de confiança	X509_STORE_add_cert()	Registra certificado raiz
6	Verificar cadeia completa	X509_verify_cert()	Validação criptográfica e temporal
7	Liberar certificados	X509_free()	Libera memória alocada
8	Retornar resultado	—	CryptoSuccess

Erros possíveis: certificado expirado (notBefore/notAfter); assinatura inválida; cadeia incompleta (raiz não confiável); revogação detectada (CRL).

Figura 8 – Sequência de operações para validação de cadeia X.509 via `X509_verify_cert()` do OpenSSL.

Fonte: Elaborado pelo autor.

5.1.5 Testes de Longa Duração (*Soak*)

Para verificar a estabilidade da biblioteca sob carga contínua, a suíte incluiu testes de longa duração (*soak tests*) compreendendo 4 operações criptográficas distintas, cada uma executada por 5 minutos ininterruptos em ambiente Linux x86_64. O protocolo de teste monitorou três propriedades: estabilidade de memória (crescimento de RSS inferior a 10 MB), corretude funcional (zero falhas) e unicidade de material criptográfico (ausência

de colisões de chaves e IVs).

O limiar de 10 MB foi adotado como critério operacional conservador para detectar crescimento progressivo incompatível com o tamanho dos buffers manipulados nas operações avaliadas; não corresponde a um limite normativo externo, mas a uma regra prática de triagem para o ambiente experimental. A memória foi monitorada por amostragem de RSS (*Resident Set Size*) via `ProcessInfo.currentRss`, disponível exclusivamente em Linux; as primeiras duas amostras de cada cenário foram descartadas para eliminar o viés de aquecimento da JIT e dos alocadores do OpenSSL, e o crescimento foi calculado a partir da terceira amostra como linha de base.

Cada cenário foi considerado aprovado apenas quando satisfizesse, de forma contínua, o conjunto de condições associadas a essas três propriedades. Nos cenários de SHA-256, AES-256-GCM e assinatura RSA, executados durante os cinco minutos completos, exigiu-se que toda operação produzisse o resultado esperado, que o crescimento de RSS permanecesse abaixo de 10 MB e que nenhuma exceção fosse lançada. Conforme a operação avaliada, a corretude funcional foi verificada pela obtenção de um resumo SHA-256 de 32 bytes, pela geração de uma *tag* AES-256-GCM de 16 bytes e pela recuperação exata do texto original após a decifração, ou pela verificação bem-sucedida da assinatura RSA. O cenário de geração de chaves obedeceu a um protocolo próprio: em vez de duração fixa, executou 30 gerações sucessivas de pares RSA-2048. Sua aprovação dependeu da validade de cada par PEM e da distinção entre todas as chaves privadas, evidência de que o material criptográfico não se repete entre execuções. Em todos os casos, o resultado “Aprovado” registrado na Tabela 16, que condensa o desfecho dos quatro cenários, corresponde à satisfação integral dessas condições ao longo de toda a execução.

Tabela 16 – Resultados dos testes de *soak* (4 operações, 5 minutos cada).

Operação	Duração	Volume Aproximado	Resultado
SHA-256 (1 KB)	5 min	~85,5 milhões de <i>hashes</i>	Aprovado
AES-256-GCM <i>round-trip</i> (1 KB)	5 min	~54 milhões de <i>round-trips</i>	Aprovado
RSA-2048 <i>keygen</i> (30 pares)	30 iterações	30 pares de chaves	Aprovado
RSA-2048 <i>sign + verify</i> (1 KB)	5 min	~390 mil assinaturas	Aprovado

Fonte: Elaborado pelo autor.

O consumo de memória manteve-se estável durante as operações observadas, com crescimento de RSS inferior a 10 MB por operação e o indicador `leak_detected` reportando `false`, valor confirmado também de forma global ao final de toda a suíte de *timing*, sob um limiar mais conservador de 50 MB adotado para acomodar buffers de trabalho legítimos do OpenSSL. Nenhum *crash*, exceção não tratada ou resultado incorreto foi registrado nas execuções realizadas. A taxa de unicidade das chaves RSA foi de 100% na amostra de 30 pares, assim como a dos vetores de inicialização AES-GCM na amostra de 10.000 valores. Esses resultados fornecem evidência de estabilidade no ambiente e no período avaliados,

mas não comprovam ausência de vazamentos ou falhas em todos os cenários de execução.

A Figura 9 ilustra a evolução do consumo de memória ao longo de cinco pontos de amostragem durante os testes de estresse. O RSS (*Resident Set Size*) corresponde à quantidade de memória física efetivamente ocupada pelo processo na memória principal, sendo a métrica usual para acompanhar o consumo real de uma aplicação. Cada ponto representa uma fase da execução: T0 é a linha de base, medida antes do carregamento das bibliotecas; T1 registra o consumo após o carregamento do OpenSSL; T2 marca o pico atingido durante a carga de estresse; T3 corresponde ao estado imediatamente posterior; e T4, à medição final. A coluna *Delta* expressa a variação acumulada de cada ponto em relação à linha de base T0, o que permite distinguir um crescimento permanente, indício de vazamento, de oscilações transitórias.

No experimento, o crescimento máximo foi de 17,5 MB, atribuível ao carregamento único das bibliotecas nativas e das estruturas internas do OpenSSL, e não a um vazamento progressivo: o valor estabiliza após T1 e o indicador `leak_detected` permanece `false`.

Ponto	Fase	RSS (KB)	Delta (KB)
T0	Baseline	146.124	0
T1	Após carregamento	157.644	+11.520
T2	Pico durante stress	164.060	+17.936
T3	Pós-stress	162.908	+16.784
T4	Final	164.060	+17.936

`leak_detected = false` Crescimento máximo: 17,5 MB 140M operações

Figura 9 – Consumo de memória RSS ao longo de 5 pontos de amostragem durante testes de stress. O crescimento de 17,5 MB entre *baseline* e pico é atribuível ao carregamento das bibliotecas nativas e estruturas internas do OpenSSL. O indicador `leak_detected = false` confirma ausência de vazamento de memória. Dados extraídos de `tcc_metrics_report.json:920-926`.

Fonte: Elaborado pelo autor.

5.2 COBERTURA DE CÓDIGO

A cobertura de código foi gerada via `flutter test -coverage` e processada com a ferramenta `lcov` (formato `lcov.info`). O resultado agregado é de **68,4%** de cobertura global, com 2.871 linhas executadas de um total de 4.195 linhas instrumentadas em 60 arquivos analisados.

5.2.1 Cobertura por Camada

A cobertura discriminada por camada arquitetural da biblioteca consta na Tabela 17.

Tabela 17 – Cobertura de código por camada arquitetural (1cov).

Camada	Cobertura (%)	Linhas cobertas/total
FFI (<code>openssl_bindings.dart</code>)	96,8	298/308
<i>Facade</i> (<code>PluginCryptoAPI</code>)	88,9	8/9
<i>Utils</i> (<i>serializers</i> , <i>parsers</i>)	80,5	215/267
<i>Operations</i> (EVP-backed)	74,8	572/765
<i>Metrics</i>	66,1	966/1.462
<i>Models</i> (DTOs + Erros)	61,4	124/202
<i>Flows</i>	58,3	684/1.174
<i>Platform Interface</i>	50,0	4/8
Total Geral	68,4	2.871/4.195

Fonte: Elaborado pelo autor.

5.2.2 Análise da Cobertura por Camada

A camada FFI (`openssl_bindings.dart`) apresenta a maior cobertura, 96,8% (298/308 linhas); os 3,2% não cobertos correspondem a caminhos de erro de falhas catastróficas do OpenSSL, de difícil reprodução controlada. A camada *Facade* atinge 88,9% (8/9 linhas) e a camada *Utils*, que reúne *serializers* e *parsers*, chega a 80,5% (215/267 linhas), com as linhas restantes concentradas em variantes de formato pouco frequentes.

As três camadas intermediárias permanecem acima de 60%, e as linhas que deixam de ser cobertas têm explicação clara em cada caso. Em *Operations* (74,8%, 572/765), o que não é exercitado são os trechos de tratamento de erro que protegem as chamadas ao OpenSSL; esses trechos só executam quando uma chamada falha, o que não ocorre durante a operação normal da biblioteca. Em *Metrics* (66,1%, 966/1.462), grande parte do código serve à coleta de métricas e só roda quando os testes correspondentes são ativados ou quando certas variáveis de ambiente estão definidas, condições ausentes na execução padrão da suíte. Em *Models* (61,4%, 124/202), as linhas não cobertas são formatos de serialização e mensagens de erro que os testes raramente acionam.

As duas camadas com menor cobertura, *Flows* (58,3%, 684/1.174) e *Platform Interface* (50,0%, 4/8), ficam baixas por motivos diferentes. A *Platform Interface* tem apenas 8 linhas, e as 4 não cobertas são exatamente o trecho que carrega a biblioteca nativa no Android (`DynamicLibrary.open('libcrypto.so')`). Esse trecho não roda em Linux, ambiente em que a cobertura foi medida; se a medição fosse feita no Android, a camada chegaria a 100%. Seus 50%, portanto, não significam código sem teste, mas um caminho específico de uma plataforma que não foi exercido no ambiente medido.

A camada *Flows* é a verdadeira lacuna de cobertura. Como ela coordena as operações mais complexas da biblioteca, possui muitos caminhos alternativos e tratamentos de erro, e os testes cobrem sobretudo o caminho principal de cada operação. Os caminhos secundários ficaram de fora não por limitação do ambiente, mas porque o esforço de teste foi concentrado nos cenários mais críticos. Ampliar a cobertura dessa camada é o principal

alvo dos trabalhos futuros.

5.2.3 Limitações da Métrica de Cobertura

A métrica de 68,4% reflete exclusivamente os testes unitários Dart executados em Linux. Os 177 testes de integração Android não contribuem para esta métrica porque o *framework* Flutter não gera `lcov.info` para execuções em `integration_test` via dispositivo remoto. Projeta-se que a cobertura efetiva, considerando ambos os ambientes, atinja ao menos 75%, uma vez que os 177 testes de integração Android exercitam caminhos de código adicionais não contabilizados pelo `lcov` em Linux.

Uma limitação adicional, e mais relevante para a interpretação dos resultados, é que a métrica contabiliza apenas os arquivos alcançados por ao menos um teste. Módulos que não são importados por nenhum teste não chegam a ser instrumentados pelo `lcov` e, por consequência, não figuram na Tabela 17 nem no denominador da cobertura global. A porcentagem reportada refere-se, portanto, ao subconjunto de código efetivamente exercitado pela suíte, e não à totalidade do código-fonte da biblioteca. Algumas funcionalidades implementadas em estágio mais avançado do desenvolvimento foram validadas apenas parcialmente ou por inspeção manual, sem testes automatizados dedicados; a ampliação dessa cobertura, sobretudo nas camadas de orquestração, é identificada como trabalho futuro prioritário.

5.3 DESEMPENHO

Os *benchmarks* de desempenho foram extraídos dos artefatos `tcc_metrics_report.json`² (schema 1.2.0, 47 KB, 63 operações distintas) e `tcc_pq_metrics.json`³. A coleta foi executada em Linux x86_64. O medidor principal realizou 75 chamadas de aquecimento antes da cronometragem; nas operações configuradas com 100 chamadas, 25 foram medidas, e nas configuradas com 150 chamadas, 75 foram medidas. Para as gerações de chaves, mais custosas, foram usadas de 3 a 5 chamadas de aquecimento e de 3 a 10 medições. Os tempos médios foram calculados sobre as chamadas cronometradas com `Stopwatch` Dart, em resolução de microssegundos.

5.3.1 Latência por Categoria de Operação

A Tabela 18 reúne a latência média das principais operações criptográficas, agrupadas por categoria.

As operações pós-quânticas merecem destaque. A geração de chaves do ML-KEM-768 (0,073 ms) e do ML-DSA-44 (0,121 ms) é sub-milissegundo e, portanto, ordens de

² Relatório de métricas gerais disponível no repositório do projeto: https://codigos.ufsc.br/isac.martins/dartcryptosec/-/blob/main/plugin_crypto/tcc_metrics_report.json. Acesso em: 11 jul. 2026.

³ Relatório de métricas pós-quânticas disponível em: https://codigos.ufsc.br/isac.martins/dartcryptosec/-/blob/main/plugin_crypto/tcc_pq_metrics.json. Acesso em: 11 jul. 2026.

Tabela 18 – Latência das operações criptográficas, Linux x86_64, OpenSSL 4.0.0.

Categoria	Operação	Tamanho	Latência (ms)
<i>Hash</i>	SHA-256	1 MB	0,538
<i>Hash</i>	SHA-512	1 MB	1,162
<i>Hash</i>	SHA3-256	1 MB	2,029
<i>Hash</i>	SHA3-512	1 MB	3,900
<i>Cipher</i>	AES-256-GCM <i>encrypt</i>	64 KB	0,026
<i>KeyGen</i>	RSA-2048	–	31,776
<i>KeyGen</i>	RSA-4096	–	514,012
<i>KeyGen</i>	ECDSA P-256	–	0,183
<i>KeyGen</i>	ECDSA P-384	–	0,618
<i>KeyGen</i>	ECDSA P-521	–	1,691
<i>Sign</i>	RSA SHA-256	32 B	0,766
<i>Sign</i>	ECDSA P-256	32 B	0,062
<i>Sign</i>	CMS ECDSA	256 B	0,095
<i>Sign</i>	CMS RSA	256 B	0,923
<i>Verify</i>	RSA SHA-256	32 B	0,032
<i>Verify</i>	ECDSA P-256	32 B	0,072
<i>Verify</i>	CMS ECDSA	256 B	0,105
<i>Verify</i>	CMS RSA	256 B	0,111
<i>Asymmetric</i>	RSA <i>encrypt</i>	32 B	0,032
<i>Asymmetric</i>	RSA <i>decrypt</i>	256 B	0,858
<i>Pós-Quântico</i>	ML-KEM-768 <i>keygen</i>	–	0,073
<i>Pós-Quântico</i>	ML-KEM-768 <i>encaps</i>	–	0,043
<i>Pós-Quântico</i>	ML-KEM-768 <i>decaps</i>	–	0,070
<i>Pós-Quântico</i>	ML-DSA-44 <i>keygen</i>	–	0,121
<i>Pós-Quântico</i>	ML-DSA-44 <i>sign</i>	32 B	0,418
<i>Pós-Quântico</i>	ML-DSA-44 <i>verify</i>	32 B	0,103

Fonte: Elaborado pelo autor com base nos artefatos de métricas do projeto.

grandeza mais rápida que a geração de chaves RSA-2048 (31,8 ms), uma vez que os esquemas baseados em reticulados não precisam buscar números primos grandes. A operação pós-quântica mais custosa é a assinatura ML-DSA, que também apresenta a maior variabilidade entre todas as primitivas medidas, comportamento atribuível à amostragem por rejeição empregada pelo Dilithium; o encapsulamento e o decapsulamento do ML-KEM, em contraste, permanecem bastante estáveis.

A Figura 10 consolida as latências das primitivas clássicas em uma visão por categoria, com os valores expressos em microssegundos (μ s), o que facilita a comparação direta do custo dentro de cada grupo funcional.

Operação	Latência (µs)
<i>Hash</i>	
SHA-256 (1 KB)	2,8
SHA-256 (1 MB)	538,4
SHA-512 (1 MB)	1.161,8
SHA3-256 (1 MB)	2.029,5
SHA3-512 (1 MB)	3.900,0
<i>Geração de Chaves</i>	
RSA-2048	31.776,0
RSA-4096	514.012,0
EC P-256	183,2
EC P-384	617,7
EC P-521	1.691,2

Operação	Latência (µs)
<i>Cifra de Bloco (CBC)</i>	
AES-128-CBC (1 KB)	2,5
AES-256-CBC (1 KB)	1,7
AES-128-CBC (1 MB)	1.611,0*
AES-256-CBC (1 MB)	2.165,0*
<i>Cifra Autenticada (GCM)</i>	
AES-256-GCM (1 KB)	5,7
AES-256-GCM (64 KB)	26,2
<i>Assinatura / Verificação</i>	
RSA-2048 Sign	765,7
RSA-2048 Verify	32,1
EC P-256 Sign	62,1
EC P-256 Verify	71,7

* Para as cifras com entrada de 1 MB, o microbenchmark aquecido produziu tempos pequenos demais para serem reais, levando o medidor a registrar uma vazão saturada (teto de 10.000 MB/s). Nesses dois casos, em vez do tempo cronometrado, usam-se medições mais confiáveis já apresentadas neste capítulo. O valor de 1.611 µs do AES-128-CBC é a média das execuções repetidas relatada na análise de variabilidade (Seção 5.3.2). O valor de 2.165 µs do AES-256-CBC é o tempo necessário para processar 1 MB à sua vazão sustentada de 461,8 MB/s (Tabela 22), isto é, $1 \text{ MB} \div 461,8 \text{ MB/s}$.

Figura 10 – Latência média por categoria criptográfica, Linux x86_64, OpenSSL 4.0.0.

Fonte: Elaborado pelo autor.

5.3.2 Análise de Variabilidade

A suíte de temporização cronometrou todas as operações apresentadas na Tabela 18, mas, para a maioria, reteve apenas valores agregados, como média, mínimo, máximo e *throughput*. Duas operações foram medidas em mais detalhe, preservando-se o tempo de cada iteração individual, o que permite examinar como os tempos se distribuem e não apenas sua média. São essas duas que viabilizam a análise de dispersão a seguir:

- **SHA-256 (1 MB):** coeficiente de variação (CV) de 4,46%, com média de 0,556 ms, mediana de 0,549 ms, desvio-padrão de 0,025 ms e percentil 99 de 0,607 ms. A baixa dispersão indica operação altamente determinística e ambiente de execução estável para esta primitiva.
- **AES-128-CBC *encrypt* (1 MB):** CV de 28,57%, com média de 1,611 ms, mediana de 1,385 ms, desvio-padrão de 0,460 ms e percentil 99 de 2,998 ms. A média superior à mediana e o percentil 99 mais elevado indicam a ocorrência de algumas medições mais lentas.

As amostras individuais foram preservadas para essas duas operações; para as demais, o relatório reteve os valores agregados apresentados na Tabela 18.

5.3.3 Throughput Agregado

O *throughput* agregado por primitiva criptográfica é resumido na Tabela 19.

Tabela 19 – *Throughput* por primitiva, Linux x86_64, 1 MB (exceto onde indicado).

Primitiva	<i>Throughput</i> (MB/s)
SHA-256	1.857,5
SHA-512	860,7
SHA3-256	492,7
SHA3-512	256,4
AES-256-GCM <i>encrypt</i> (64 KB)	2.387,9
AES-128-CBC <i>encrypt</i> (64 KB+)	10.000,0*

Fonte: Elaborado pelo autor.

Entre as funções de *hash*, a Tabela 19 mostra o SHA-256 como o mais rápido (1.857,5 MB/s), seguido pelo SHA-512 (860,7 MB/s), pelo SHA3-256 (492,7 MB/s) e pelo SHA3-512 (256,4 MB/s). No *payload* de 1 MB, o SHA-256 processa os dados cerca de 2,2 vezes mais rápido que o SHA-512 e 7,2 vezes mais rápido que o SHA3-512. Essa diferença decorre da maturidade das implementações otimizadas do SHA-2 frente ao custo da permutação Keccak, base da família SHA-3. As cifras simétricas, por sua vez, alcançam vazão superior à de qualquer função de *hash*: o AES-128-CBC ultrapassa 10 GB/s e o AES-256-GCM atinge 2.387,9 MB/s, graças à aceleração por *hardware* AES-NI discutida adiante.

O *throughput* expresso em megabytes por segundo, contudo, favorece operações sobre grandes volumes de dados. Em aplicações que processam muitas mensagens pequenas, como a assinatura de *tokens* ou o resumo de registros curtos, é mais informativo medir o número de operações concluídas por segundo. A Tabela 20 adota essa perspectiva para *payloads* representativos.

Tabela 20 – *Throughput* em operações por segundo, Linux x86_64, OpenSSL 4.0.0.

Operação	<i>Throughput</i> (ops/s)
SHA-256 (1 KB)	357.000
AES-256-GCM <i>encrypt</i> (1 KB)	180.000
AES-256-GCM <i>encrypt</i> (1 MB)	2.388*
SHA-256 (1 MB)	1.857

* A medição do AES-256-GCM em 1 MB sofreu a mesma saturação das demais cifras em buffers grandes, e o tempo cronometrado não é confiável. O valor de 2.388 ops/s parte da vazão de 2.387,9 MB/s medida em 64 KB (Tabela 19), a maior medição não saturada dessa operação: como cada operação cifra cerca de 1 MB, a taxa em operações por segundo equivale numericamente à vazão em MB/s.

Fonte: Elaborado pelo autor.

Os dados da Tabela 20 evidenciam a relação inversa entre o tamanho do *payload* e o número de operações por segundo: o SHA-256 sobre 1 KB é aproximadamente 192× mais rápido, em operações por segundo, que o mesmo algoritmo sobre 1 MB. Esse contraste mostra que, para entradas pequenas, o custo é dominado pela sobrecarga fixa de cada chamada, e não pelo processamento dos dados em si.

Para a `dartcryptosec`, esses números têm uma leitura prática direta. Como a biblioteca delega o processamento ao OpenSSL por meio da FFI, sua vazão acompanha a da implementação nativa, e a camada de integração não se torna um gargalo em cargas reais. Vazões na casa de gigabytes por segundo para cifra e de centenas de megabytes por segundo para *hash* são suficientes para os cenários a que a biblioteca se destina, como cifração de arquivos, transferência de dados em sessões seguras e assinatura de documentos em volume. Em fluxos com grande quantidade de operações sobre entradas pequenas, entretanto, o custo de cada chamada passa a predominar. Nesses casos, agrupar as operações tende a ser mais eficiente do que invocá-las isoladamente.

5.3.4 Aceleração por Hardware

O OpenSSL 4.0.0 compilado para `x86_64` utiliza aceleração por *hardware* AES-NI (Intel Advanced Encryption Standard New Instructions) para operações de cifra simétrica, resultando em latências inferiores a $10\ \mu\text{s}$ para *payloads* de 1 KB. O *overhead* da camada FFI (`dart:ffi`) é estimado em $1\text{--}2\ \mu\text{s}$ por chamada, representando fração relevante apenas para operações de latência muito baixa (abaixo de $10\ \mu\text{s}$). Para operações assimétricas como geração de chaves RSA-2048 (31,8 ms), o *overhead* de FFI é inferior a 0,01% do tempo total.

5.3.5 Comparação Linux vs. Android

A Tabela 21 confronta a latência (*warm*) das duas plataformas, medida com a mesma metodologia (5 iterações de aquecimento e 25 medidas, via Stopwatch) em ambos os ambientes. Os dados Android foram extraídos do artefato `android_benchmark.json`.⁴

Tabela 21 – Comparação de desempenho Linux (`x86_64`) vs. Android (`arm64-v8a`).

Operação	Linux	Android	Fator
SHA-256 (1 KB)	$2,8\ \mu\text{s}$	$36,2\ \mu\text{s}$	$12,9\times$
AES-256-GCM (1 KB)	$5,7\ \mu\text{s}$	$16,0\ \mu\text{s}$	$2,8\times$
RSA-2048 <i>keygen</i>	31,8 ms	223,4 ms	$7,0\times$
RSA-2048 <i>sign</i>	0,766 ms	4,0 ms	$5,2\times$
ECDSA P-256 <i>sign</i>	0,062 ms	1,0 ms	$16,2\times$
ML-KEM-768 <i>keygen</i>	0,073 ms	0,154 ms	$2,1\times$
ML-DSA-44 <i>sign</i>	0,418 ms	0,957 ms	$2,3\times$

Fonte: Elaborado pelo autor com base nos artefatos de métricas do projeto.

O Android é mais lento em todas as operações, mas o fator varia muito conforme a primitiva. As operações pós-quânticas apresentam a menor diferença entre plataformas (ML-KEM-768 *keygen* $2,1\times$, ML-DSA-44 *sign* $2,3\times$), justamente por não se beneficiarem

⁴ Relatório do *benchmark* Android disponível em: https://codigos.ufsc.br/isac.martins/dartcryptosec/-/blob/main/tcc_test_app/android_benchmark.json. Acesso em: 11 jul. 2026.

de instruções dedicadas no x86_64. Em contraste, SHA-256 (12,9×) e ECDSA P-256 (16,2×) exibem os maiores fatores, pois as extensões SHA-NI e AES-NI do x86_64 não têm equivalente explorado no ARM64. Contribuem ainda para a diferença o ambiente de execução Dart em modo *debug* (sem AOT) no Android e o carregamento dinâmico com verificações de segurança adicionais; uma compilação em modo *release* tenderia a reduzir esses fatores.

5.3.6 Simulação TLS 1.3

Para validar a aplicabilidade da biblioteca em cenários reais de segurança de rede, foi conduzida uma simulação completa de sessão TLS 1.3 com a *cipher suite* TLS_AES_128_GCM_SHA256, percorrendo todos os estágios desde o *handshake* inicial até a transferência de dados em massa. O *handshake* completo, composto por *key exchange* ECDHE (0,549 ms, 64,7%), *certificate verify* ECDSA P-256 (0,287 ms, 33,8%) e derivação HKDF (0,012 ms, 1,4%), totaliza **0,848 ms**. Esse valor sub-milissegundo demonstra a viabilidade da *dartcryptosec* como *backend* criptográfico para servidores HTTPS e sistemas de mensageria segura.

Na fase de transferência de dados em massa, o *throughput* de cifração atinge 922,8 MB/s e o de decifração, 994,3 MB/s (10 transferências de 1 MB cada). A sessão completa, incluindo *handshake*, transferência de 10 MB unidirecionais e *overhead* de *kernel* e *framing*, totaliza 39,586 ms, resultando em um *throughput* efetivo de 252,6 MB/s (unidirecional). A assimetria entre cifração e decifração é de +7,7% (decifração mais rápida), consistente com o comportamento esperado para AES-GCM, em que a verificação da *authentication tag* é ligeiramente mais rápida que sua geração.

A Figura 11 resume essa simulação, reunindo a troca de mensagens entre cliente e servidor e a decomposição temporal do *handshake*. A sequência segue o fluxo padrão do TLS 1.3: o *ClientHello* inicia a negociação, o *ServerHello* fixa a *cipher suite* escolhida, o servidor envia seu certificado X.509 (ECDSA P-256), e as mensagens *CertificateVerify* e *Finished* concluem o estabelecimento da sessão, após o qual os dados de aplicação trafegam cifrados. A decomposição dos tempos mostra que o custo do *handshake* está quase todo nas operações assimétricas: a troca de chaves ECDHE e a verificação do certificado somam 98,5% do total, enquanto a derivação das chaves de sessão é praticamente desprezível. O conjunto confirma que, mesmo concentrando o esforço nas primitivas assimétricas, o estabelecimento completo da sessão permanece na casa de um milissegundo, compatível com o uso da biblioteca em servidores que abrem muitas conexões.

Sequência de Mensagens TLS 1.3

Cliente (Dart)	Dir.	Servidor (OpenSSL)
ClientHello	→	
	←	ServerHello (TLS_AES_128_GCM_SHA256)
	←	Certificate (X.509, ECDSA P-256)
CertificateVerify	→	
	←	Finished
Finished	→	
↔ Application Data Cifração: 922,8 MB/s Decifração: 994,3 MB/s		

Fases do Handshake (total: 0,848 ms)

Fase	Tempo	%
Key Exchange ECDHE	0,549 ms	64,7%
Certificate Verify	0,287 ms	33,8%
HMAC Deriv.†	0,012 ms	1,4%
Total	0,848 ms	100%

Cipher Suite: TLS_AES_128_GCM_SHA256 *Key Exchange*: ECDHE secp256r1 *Sessão total*: 39,586 ms

† Aproximação do custo do HKDF via $4 \times$ SHA-256 (`security_benchmark.dart:790-795`); não é uma execução real de HMAC.

Figura 11 – Simulação de *handshake* TLS 1.3 com *cipher suite* TLS_AES_128_GCM_SHA256 (total: 0,848 ms).

Fonte: Elaborado pelo autor.

5.3.7 Throughput em Operações por Segundo

Em termos de *throughput*, o SHA-256 processa aproximadamente 357.000 operações por segundo (ops/s) para *payloads* de 1 KB, enquanto o AES-256-GCM *encrypt* atinge cerca de 180.000 ops/s. Para *payloads* de 1 MB, o *throughput* reduz-se para 1.857 ops/s (SHA-256) e 2.388 ops/s (AES-256-GCM), refletindo o custo proporcional ao tamanho dos dados processados. No domínio assimétrico, o RSA-2048 *verify* atinge aproximadamente 31.000 ops/s, enquanto o ML-DSA-44 *verify* alcança cerca de 9.700 ops/s (diferença de fator $\sim 3,2\times$). Já na assinatura o quadro se inverte: o ML-DSA-44 *sign* (cerca de 2.400 ops/s) é competitivo com o RSA-2048 *sign* (cerca de 1.300 ops/s), evidenciando que os esquemas pós-quânticos baseados em reticulados não implicam, em geral, penalidade de desempenho frente aos esquemas clássicos.

5.3.8 Ranking de *Cipher Suites* Simétricas

A Tabela 22 ordena as quatro combinações de *cipher suites* simétricas avaliadas por *throughput* de cifração para *payloads* de 1 MB.

Tabela 22 – *Ranking* de *cipher suites* simétricas, Linux x86_64, 1 MB.

Rank	<i>Cipher Suite</i>	<i>Encrypt</i> (MB/s)	<i>Decrypt</i> (MB/s)
#1	AES-256-GCM	718,4	795,5
#2	AES-128-GCM	624,9	786,2
#3	AES-128-CBC	495,4	720,4
#4	AES-256-CBC	461,8	766,9

Fonte: Elaborado pelo autor.

A AES-256-GCM lidera o *ranking* com 718,4 MB/s de *throughput* de cifração, superando a segunda colocada AES-128-GCM em 15,0%. Esse resultado, aparentemente contraintuitivo (mais *rounds* (14 vs. 10) resultam em maior *throughput*), é explicado por três fatores: (i) a diferença de *rounds* é amortizada sobre *payloads* grandes (65.536 blocos de 16 bytes para 1 MB); (ii) o OpenSSL 4.0.0 utiliza AES-NI, onde a diferença entre AES-128 e AES-256 é marginal (1–2 ciclos por *round*); e (iii) o *pipelining* de instruções do processador sobrepõe *rounds* de diferentes blocos (GUPTA; PRAKASH, 2024). O modo GCM supera o CBC em aproximadamente 40% na cifração devido ao paralelismo inerente ao CTR (modo subjacente) e à execução paralela do GHASH via instrução PCLMULQDQ. A decifração é consistentemente mais rápida que a cifração em todas as *cipher suites* (razão *decrypt/encrypt* de 1,11 a 1,66), e o AES-256-GCM exibe a menor assimetria (1,11×). Esse conjunto de resultados o consolida como a recomendação primária para todos os cenários de uso da biblioteca.

5.4 SEGURANÇA DA IMPLEMENTAÇÃO

A segurança da implementação foi verificada por meio de 63 verificações automatizadas registradas no relatório de métricas gerais já identificado nesta seção, cobrindo entropia, aleatoriedade, unicidade de chaves, autenticação GCM, curvas seguras e testes de resposta conhecida (*Known Answer Tests* – KAT). O sumário do relatório indica: “***All security checks passed.***”

5.4.1 Validações de Aleatoriedade

Os resultados das validações de qualidade do gerador de números aleatórios (CS-PRNG) constam na Tabela 23.

Tabela 23 – Validações de aleatoriedade do CSPRNG.

Métrica	Valor	Limiar	Resultado
Entropia Shannon (1.024 B)	7,981 bits/byte	$\geq 7,950$	Aprovado
χ^2 (chi-quadrado)	254,39	–	–
p-valor (χ^2)	0,511	$\geq 0,01$	Aprovado

Fonte: Elaborado pelo autor.

A entropia de 7,981 bits/byte está dentro do esperado para um CSPRNG (*Cryptographically Secure Pseudo-Random Number Generator*). O p-valor de 0,511 no teste qui-quadrado indica que a distribuição dos bytes não se desvia da distribuição uniforme ao nível de significância de 0,01. Esses dois indicadores, combinados, permitem afirmar que o gerador de números aleatórios do OpenSSL atende aos requisitos de qualidade exigidos para aplicações criptográficas.

5.4.2 Unicidade de Chaves e Vetores de Inicialização

Foram gerados 100 pares de chaves RSA e 100 pares de chaves EC, além de 100 vetores de inicialização (IV) para AES-GCM. Em todos os casos, a taxa de unicidade foi de 100%, confirmando que o gerador de chaves do OpenSSL não produz colisões. Esse resultado é consistente com a alta entropia do CSPRNG documentada na subseção anterior e reforça a confiabilidade do sistema para cenários em que a não-repetição de material criptográfico é crítica.

5.4.3 Curvas Seguras

Três curvas elípticas NIST foram validadas contra o *checklist* de curvas seguras (Bernstein/Lange SafeCurves): `prime256v1`, `secp384r1` e `secp521r1`. Todas as três passaram em todos os critérios: ordem prima (imune a ataques Pohlig-Hellman), cofator 1 (sem subgrupos não-triviais), *embedding degree* astronomicamente grande (imune a ataques de emparelhamento) e *twist* seguro (imune a ataques de curva inválida).

5.4.4 Autenticação GCM

O AES-GCM está configurado com verificação de *tag* de autenticação e *binding* de AAD (*Additional Authenticated Data*). A rejeição de chave cruzada foi confirmada: uma *tag* gerada com uma chave é rejeitada quando verificada com outra chave distinta.

5.4.5 Teste NIST SP800-22

O teste Serial Test (T7) da bateria NIST SP800-22 apresentou uma ocorrência isolada de p-valor abaixo do limiar de 0,01 ($p = 0,00418$) em uma das execuções. O comportamento é intrinsecamente estatístico: mesmo um gerador perfeitamente funcional produzirá p-valores abaixo de 0,01 em aproximadamente 1% das execuções. A re-execução com semente diferente confirmou que 15 dos 16 testes NIST passam consistentemente. O incidente está documentado nos registros de limitações conhecidas com severidade *Low* e classificação de *flake* estatístico, não defeito de código.

5.4.6 Testes Adversariais e de Memória

Os testes adversariais (`crypto_adversarial_test.dart`) submeteram a biblioteca a 31 cenários de entradas maliciosas ou malformadas no Android, todos aprovados. Os testes de memória (`crypto_memory_test.dart`) verificaram a ausência de vazamento de memória. O delta de RSS (*Resident Set Size*) entre a linha de base e o pico foi de 17.936 KB (17,5 MB), valor atribuível ao carregamento das bibliotecas nativas e à alocação de estruturas internas do OpenSSL. O campo `leak_detected` no relatório JSON reporta `false`.

5.4.7 Limitações Conhecidas de Segurança

Quatro limitações ativas de segurança estão documentadas nos registros de limitações conhecidas do projeto, sumarizadas na Tabela 24.

A modelagem de ameaças STRIDE da `dartcryptosec`, apresentada na Seção 4.6.4 no contexto da segurança arquitetural, identificou 15 ameaças (T1–T15), das quais 8 foram classificadas como prioridade P2–P3 e constituem o principal foco de mitigação. O catálogo completo encontra-se documentado no Apêndice B.

5.5 COMPARAÇÃO COM O ESTADO DA ARTE

Esta seção posiciona a biblioteca `dartcryptosec` em relação às principais alternativas disponíveis no ecossistema Dart/Flutter: `pointycastle` (versão 3.x) e `cryptography` (package:cryptography), ambas implementações *pure Dart*. A Tabela 25 organiza a comparação segundo critérios relevantes para aplicações que demandam segurança criptográfica, incluindo um panorama das vantagens que as alternativas oferecem sobre a `dartcryptosec`. A tabela concentra-se nos critérios que *diferenciam* as bibliotecas. As primitivas criptográficas básicas, como funções de *hash*, MACs, cifras simétricas (AES) e os esquemas assimétricos clássicos (RSA e curvas elípticas), estão presentes nas três bibliotecas e, por isso, foram omitidas: por serem comuns a todas, não auxiliam na distinção entre elas.

Tabela 24 – Limitações conhecidas de segurança.

Issue	Severidade	Descrição
<i>Fuzzing</i> de corrupção CMS	Média	<code>StateError</code> não capturado em <code>cms_operations.dart:325</code> quando o OpenSSL reporta erro de <i>parsing</i> PEM em entrada malformada. A correção planejada envolve encapsular <code>PEM_read_bio_X509</code> em bloco <code>try/catch</code> .
NIST SP800-22 <i>Serial Test</i> (T7)	Baixa	<i>Flake</i> estatístico: p-valor ocasionalmente abaixo de 0,01 por definição do teste. Resolve-se com re-execução.
Limiares de <i>constant-time</i> no WSL1	Baixa	As execuções no WSL1 apresentaram variabilidade temporal suficiente para exigir o relaxamento do limiar <code>maxMinRatio</code> de 30,0 para 100,0. O experimento não isolou a origem dessa variação nem realizou comparação controlada com Linux nativo.
Flutuação de RSS no WSL1	Baixa	Foram observadas flutuações nas leituras de RSS obtidas por <code>/proc/self/statm</code> durante as execuções no WSL1. Como não houve comparação controlada entre ambientes, esses valores são tratados como característica observada da medição, sem atribuição causal exclusiva ao WSL1.

Fonte: Elaborado pelo autor.

Tabela 25 – Comparação entre bibliotecas criptográficas para Dart/Flutter.

Critério	dartcryptosec	pointycastle 3.x	cryptography
<i>Infraestrutura de Chaves Públicas</i>			
X.509 — <i>parsing</i> e geração de certificados	Sim	Parcial [†]	Não
Validação de cadeia de certificação	Sim	Não	Não
LCR — <i>parsing</i> e validação	Sim	Não	Não
OCSF — verificação de revogação	Sim	Não	Não
CSR PKCS#10	Sim	Não	Não
CMS — assinatura e cifração	Sim	Não	Não
CAdES	Sim	Não	Não
<i>Outros recursos</i>			
Pós-Quântico (ML-KEM / ML-DSA)	Sim	Não	Não
FIPS 140-3	Sim	Não	Não
<i>Vantagens das alternativas sobre a dartcryptosec</i>			
Pure Dart (sem dependências nativas)	Não	Sim	Sim
Instalação sem etapa de compilação nativa	Não	Sim	Sim

Fonte: Elaborado pelo autor.

Nota: [†] pointycastle oferece *parsing* básico de X.509, sem validação de cadeia completa nem extensões críticas.

Os valores de referência do OpenSSL foram extraídos do artefato `tcc_openssl_cli_metrics.json`.⁵

Tabela 26 – Latência da dartcryptosec (FFI) vs. OpenSSL em C (Linux x86_64).

Operação	dartcryptosec	OpenSSL C	Eficiência
SHA-256 (1 KB)	2,8 μ s	0,54 μ s	19%
AES-256-GCM <i>encrypt</i> (1 KB)	5,7 μ s	0,29 μ s	5%
RSA-2048 <i>keygen</i>	31,8 ms	62–153 ms [†]	—
ECDSA P-256 <i>sign</i>	0,062 ms	51–141 ms [†]	—
ML-KEM-768 <i>keygen</i>	0,073 ms	47–139 ms [†]	—
ML-DSA-44 <i>sign</i>	0,418 ms	48–200 ms [†]	—

Fonte: Elaborado pelo autor com base nos artefatos de métricas do projeto.

[†] Valor obtido via *openssl* CLI, que inclui ~17 ms de criação de processo por chamada; não representa a latência bruta da biblioteca C, mas a invocação por linha de comando. Para SHA-256 e AES-256-GCM, a coluna usa a latência bruta de *openssl speed*.

Os dados revelam dois regimes distintos. Para operações na escala de microssegundos (SHA-256 e AES-256-GCM), o tempo bruto da biblioteca C é de apenas 0,3–0,5 μ s, de modo que o *overhead* fixo da fronteira FFI (alocação de ponteiros, travessia do `dart:ffi` e cronometragem), da ordem de 2–5 μ s por chamada, domina o custo total e reduz a eficiên-

⁵ Relatório de métricas do OpenSSL nativo e da ferramenta de linha de comando disponível em: https://codigos.ufsc.br/isac.martins/dartcryptosec/-/blob/main/plugin_crypto/tcc_openssl_cli_metrics.json. Acesso em: 11 jul. 2026.

cia aparente a 5–19%. Esse *overhead* é por chamada, e não por byte: em *payloads* de 1 MB, a vazão atinge a casa de gigabytes por segundo (Tabela 19), o que dilui completamente o custo de travessia. Para operações na escala de milissegundos (geração de chaves RSA e operações pós-quânticas), o *overhead* de FFI torna-se imperceptível e a `dartcryptosec` reproduz a latência nativa.

Já a comparação direta com o `openssl` CLI é dominada pelo custo de criação de processo (~ 17 ms por invocação), o que torna a API FFI duas a três ordens de grandeza mais rápida que invocar o executável a cada operação. Resta, por fim, a ausência de alternativas para algoritmos pós-quânticos no ecossistema Dart, aspecto em que a biblioteca permanece sem equivalentes.

5.5.1 Vantagens da `dartcryptosec`

A `dartcryptosec` ocupa uma posição singular no ecossistema Dart/Flutter: é a única biblioteca que combina *bindings* FFI nativos de alto desempenho com OpenSSL 4.0.0 em modo FIPS e suporte a algoritmos pós-quânticos. Essa combinação de características confere à biblioteca vantagens competitivas substanciais em relação às alternativas *pure Dart*, as quais são enumeradas a seguir:

1. **Desempenho próximo ao nativo:** para operações assimétricas e pós-quânticas, o *overhead* de FFI ($1\text{--}2\ \mu\text{s}$ por chamada) é inferior a 0,01% do tempo total, e a `dartcryptosec` reproduz a latência do OpenSSL nativo em C (Tabela 26). Em primitivas simétricas muito rápidas, esse *overhead* fixo passa a dominar o custo por chamada, mas é diluído em *payloads* grandes, nos quais a vazão alcança a casa de gigabytes por segundo.
2. **Conformidade FIPS 140-3:** herdada do OpenSSL 4.0.0 compilado com `enable-fips`. Além do elevado desempenho, essa conformidade constitui um segundo diferencial crítico, uma vez que nenhuma biblioteca *pure Dart* oferece compatibilidade com FIPS.
3. **Suporte a algoritmos pós-quânticos:** ML-KEM-768 (FIPS 203) e ML-DSA-44 (FIPS 204) estão implementados e testados em Linux e Android, incluindo 22 testes de integração específicos para PQ. Outro diferencial relevante é que a biblioteca antecipa a transição para criptografia pós-quântica, funcionalidade indisponível nas alternativas *pure Dart*.
4. **Suporte a operações de ICP:** a biblioteca integra em uma única dependência Dart um subconjunto relevante de operações de Infraestrutura de Chaves Públicas — geração e verificação de certificados X.509, validação de cadeias de certificação, *parsing* de LCR, verificação de revogação via OCSP, criação de CSRs PKCS#10 e CMS (assinatura, verificação, cifração e decifração) —,

evitando a necessidade de combinar múltiplas bibliotecas para essas operações específicas.

5.5.2 Desvantagens e Limitações

1. **Complexidade de *build*:** requer compilação nativa do OpenSSL para cada plataforma-alvo, enquanto bibliotecas *pure Dart* funcionam imediatamente.
2. **Dependência externa:** o uso do OpenSSL introduz riscos de compatibilidade com atualizações de sistema operacional e complexidade de depuração de problemas nativos (ex.: corrupção da fila de erros no Android, documentada no *commit b22631f*). Associada à complexidade de *build*, essa dependência externa impõe um custo de manutenção continuada que não existe em bibliotecas *pure Dart*.
3. **Tamanho do binário:** as bibliotecas compartilhadas nativas adicionam aproximadamente 2–5 MB ao tamanho do APK/*app*. Adicionalmente, o impacto desse acréscimo pode ser relevante em contextos com restrições severas de espaço de armazenamento, como dispositivos embarcados.

5.5.3 Posicionamento

Para aplicações que demandam conformidade regulatória (FIPS 140-3), alto *throughput* criptográfico ou algoritmos pós-quânticos, a **dartcryptosec** representa a escolha recomendada no ecossistema Dart/Flutter. Para prototipagem rápida ou aplicações sem requisitos rigorosos de segurança, bibliotecas *pure Dart* como **pointycastle** oferecem menor complexidade de configuração inicial. A decisão entre as alternativas deve ponderar, portanto, o custo de configuração da infraestrutura nativa em relação aos ganhos de desempenho, conformidade e cobertura de funcionalidades criptográficas.

5.6 ANÁLISE DOS RESULTADOS EXPERIMENTAIS

Esta seção consolida a análise dos principais indicadores de desempenho e cobertura medidos nos experimentos, utilizando os resultados descritos nas seções anteriores.

5.6.1 Latência de FFI para Operações Simétricas

Os resultados experimentais mostram que a latência das operações simétricas em Linux permanece muito abaixo de 10 μ s para *payloads* de 1 KB: o SHA-256 (1 KB) apresenta latência média de 2,8 μ s e o AES-256-GCM *encrypt* (1 KB), de 5,7 μ s. Na prática, uma latência de 2,8 μ s por operação SHA-256 significa que uma *pipeline* de assinatura digital de 10.000 documentos consome menos de 30 ms em *overhead* criptográfico, viabilizando

processamento em lote sem degradação perceptível do tempo de resposta em aplicações interativas.

5.6.2 Overhead de FFI para Operações Assimétricas

O *overhead* estimado de FFI ($\sim 1\text{--}2\ \mu\text{s}$ por chamada) é negligível para operações dominadas por processamento nativo pesado: representa menos de 0,01% da latência do RSA-2048 *keygen* (31,8 ms), em que o tempo é consumido pela geração de primos. Para operações pós-quânticas muito rápidas, como o ML-KEM-768 *keygen* (0,073 ms), o mesmo *overhead* fixo passa a representar uma fração pequena, porém não desprezível, do tempo total, já que a aritmética de reticulados é concluída em dezenas de microssegundos. Em todos os casos, o resultado sustenta empiricamente a decisão arquitetural de usar FFI em vez de *Method Channels*: a sobrecarga adicional de serialização de *Method Channels* ($\sim 100\text{--}200\ \mu\text{s}$), embora não crítica isoladamente, torna-se cumulativa em *pipelines* com múltiplas operações sequenciais.

5.6.3 Cobertura FFI e Detecção de Regressões em ICP

A cobertura FFI observada em Linux foi de 68,4%, exercitando os fluxos críticos de ICP: *parsing* de certificados X.509, validação de cadeias de confiança, consultas OCSP/LCR, encapsulamento CMS, geração de CSRs PKCS#10 e assinatura digital com RSA e ECDSA. A evidência de suficiência dessa cobertura é a ausência de falhas nos 177 testes de integração Android sobre os mesmos subsistemas: nenhum subsistema aprovado em Linux revelou regressão no Android. Em nível de componente, o módulo `openssl_bindings.dart` atinge 96,8% (298/308 linhas); os 3,2% restantes correspondem a caminhos de erro de falhas catastróficas do OpenSSL (ex.: ponteiro NULL em retorno de `EVP_MD_CTX_new`), de difícil simulação controlada. A cobertura do `native_loader.dart` é de 44,4%, pois o caminho de carregamento Android (`DynamicLibrary.open('libcrypto.so')`) não é exercitável no Linux.

A Tabela 27 sumariza os principais indicadores obtidos.

Tabela 27 – Sumário dos resultados experimentais.

Indicador	Resultado	Evidência Principal
Latência simétrica (1 KB)	$< 10\ \mu\text{s}$	SHA-256: $2,8\ \mu\text{s}$; AES-GCM: $5,7\ \mu\text{s}$
Overhead FFI assimétrico	Desprezível	$< 0,01\%$ para RSA-2048
Diferença Android / Linux	$2,1\text{--}16,2\times$	Menor em PQ ($2,1\times$); maior em ECDSA ($16,2\times$)
Cobertura ICP Linux	68,4%	0 falhas em 177 testes de integração Android

Fonte: Elaborado pelo autor.

Os indicadores experimentais confirmam a adequação da arquitetura FFI adotada: latência simétrica na casa de microssegundos, *overhead* assimétrico negligível, paridade

funcional entre plataformas e cobertura de código suficiente para detecção de regressões nas operações de ICP.

Os resultados apresentados neste capítulo fornecem a base empírica para a validação das hipóteses de pesquisa e fundamentam as conclusões e recomendações de trabalhos futuros discutidas no Capítulo 6.

6 CONCLUSÃO

O presente trabalho documentou o processo de concepção, arquitetura e implementação da `dartcryptosec`, uma biblioteca criptográfica para a linguagem Dart. A investigação partiu da constatação, obtida durante o desenvolvimento do projeto *Medida Inteligente* no Laboratório de Segurança em Computação (LabSEC) em parceria com o Inmetro, de que aplicações Dart com requisitos de criptografia assimétrica, certificados digitais e formatos de assinatura encontravam limitações nas bibliotecas disponíveis (IDALINO *et al.*, 2022). Ao longo do trabalho, foram apresentados os fundamentos teóricos relacionados à ICP (Infraestrutura de Chaves Públicas), a análise das bibliotecas existentes, a arquitetura proposta e sua validação em Linux e Android. A execução de 692 testes unitários e de integração nessas plataformas fornece evidência de que a integração direta com o OpenSSL por meio de FFI (Foreign Function Interface) permite reunir essas operações em uma API Dart sem reimplementar as primitivas criptográficas.

Este capítulo final retoma os objetivos que orientaram a pesquisa e sintetiza os principais resultados alcançados. Em seguida, delimita as contribuições do trabalho para o ecossistema Dart, para a comunidade acadêmica e para o LabSEC/UFSC. Expõe, ainda, as limitações e dificuldades encontradas durante o desenvolvimento e, por fim, aponta direções para trabalhos futuros que possam estender e consolidar a biblioteca.

6.1 RETOMADA DOS OBJETIVOS

O objetivo geral estabelecido na Seção 1.4.1, desenvolver uma biblioteca criptográfica para Dart que fornecesse suporte abrangente a funcionalidades de ICP, suprimindo as deficiências identificadas no ecossistema atual, foi alcançado em sua essência. A `dartcryptosec` materializa-se como uma biblioteca funcional, com arquitetura documentada e verificada por bateria de testes. Ela oferece à comunidade Dart um conjunto de APIs até então indisponível, cobrindo desde a manipulação de certificados até algoritmos pós-quânticos. Os parágrafos a seguir retomam cada objetivo específico enunciado na Seção 1.4.2 e avaliam seu grau de consecução, com respaldo nas evidências reunidas ao longo do desenvolvimento.

Analisar o estado da arte das bibliotecas criptográficas em Dart. A análise de `Pointycastle`, `Cryptography` e `Crypto`, comparadas às bibliotecas de referência `Bouncy Castle` e `OpenSSL`, cumpriu este objetivo (CONTRIBUTORS, 2024; TEAM, D. C., 2024; TEAM, D., 2024a; CASTLE, 2024). A Tabela 6 mostra que as soluções Dart analisadas não reúnem verificação de revogação, suporte a CMS (Cryptographic Message Syntax)/CAAdES (CMS Advanced Electronic Signatures) e validação completa de cadeias de certificados. Essas lacunas, estabelecidas na Seção 1.2, justificaram o desenvolvimento da `dartcryptosec`.

Projetar uma arquitetura modular para integração criptográfica nativa.

A arquitetura em sete camadas foi concebida com separação de responsabilidades entre a integração nativa, as operações criptográficas, os modelos, os casos de uso e a API pública. A inclusão de ML-KEM e ML-DSA exigiu alterações nas camadas de integração, operações e modelos, sem modificar a forma de acesso utilizada pelos casos de uso.

Assegurar compatibilidade multiplataforma. No escopo definido para o trabalho, o objetivo foi alcançado. A biblioteca foi compilada e testada em Linux x86_64 e Android nas arquiteturas contempladas pela validação, demonstrando seu funcionamento em ambientes de execução distintos. Como iOS, macOS e Windows não foram submetidos a testes de integração, não há evidência suficiente para declarar suporte validado a essas plataformas, cuja inclusão permanece como possibilidade de expansão.

Implementar funcionalidades essenciais de ICP. A implementação atendeu a este objetivo. A `dartcryptosec` oferece geração e verificação de assinaturas digitais com RSA (Rivest-Shamir-Adleman) e ECDSA (Elliptic Curve Digital Signature Algorithm), além de incorporar dois algoritmos pós-quânticos: o ML-KEM, destinado ao estabelecimento de segredos compartilhados por encapsulamento e desencapsulamento, e o ML-DSA, empregado na geração e na verificação de assinaturas digitais. Esses algoritmos seguem, respectivamente, os FIPS 203 e FIPS 204 (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b, 2024a). A biblioteca provê, ainda, validação de cadeias de certificados X.509, incluindo a verificação de assinaturas, a validação de período de vigência e a aplicação de restrições básicas, bem como o gerenciamento de *trust anchors* por meio de repositórios de autoridades certificadoras confiáveis (COOPER *et al.*, 2008).

Implementar mecanismos de verificação de revogação de certificados. O suporte incorporado à consulta e ao processamento de LCR (Listas de Certificados Revogados) e ao OSCP (Online Certificate Status Protocol) responde a este objetivo, permitindo a verificação do status de revogação de certificados por ambos os mecanismos definidos na RFC (Request for Comments) 5280 e na RFC 6960 (COOPER *et al.*, 2008; SANTESSON *et al.*, 2013). A implementação abrange a validação da assinatura da LCR, a verificação de que o certificado consultado não consta da lista e a interpretação das respostas OSCP conforme os códigos de status padronizados.

Implementar manipulação de Certificate Signing Requests (CSRs). A funcionalidade correspondente encontra-se implementada e validada. O módulo de CSRs permite criar requisições no formato PKCS#10 e assiná-las com a chave privada do solicitante, conforme a RFC 2986 (NYSTROM; KALISKI, 2000).

Suporte a primitivas criptográficas. A disponibilização de funções de *hash* e de cifradores AES nos modos CBC e GCM, por meio da API EVP do OpenSSL, concretizou este objetivo (STANDARDS; TECHNOLOGY, 2001; OPENSSL PROJECT AUTHORS, 2026a).

Desenvolver documentação abrangente. A produção de documentação técnica voltada ao desenvolvedor deu cumprimento a este objetivo. O conjunto elaborado inclui

referência detalhada da API, descrição da arquitetura, orientações de integração, exemplos de uso e comentários no código-fonte. Esses materiais documentam tanto as operações criptográficas expostas quanto as decisões arquiteturais e os procedimentos necessários para compreender, utilizar e estender a biblioteca.

Integrar com bibliotecas criptográficas consolidadas. A integração com o OpenSSL 4.0.0 foi realizada por meio de *bindings* manuais via `dart:ffi`, abrangendo os subsistemas necessários às operações implementadas. A inclusão dos algoritmos pós-quânticos ML-KEM (FIPS 203) e ML-DSA (FIPS 204), inicialmente considerada uma possibilidade de expansão, foi incorporada durante o desenvolvimento (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b, 2024a).

6.2 SÍNTESE DOS PRINCIPAIS RESULTADOS

O desenvolvimento da `dartcryptosec` (MARTINS, 2026) produziu uma base de código Dart que reúne operações de ICP por meio de integração via FFI com o OpenSSL nativo. Esta seção sintetiza os resultados nas dimensões de arquitetura, implementação e validação.

No plano arquitetural, a estrutura em sete camadas (Seção 4.6) permitiu concentrar as particularidades do OpenSSL nas camadas inferiores. A inclusão do ML-KEM e do ML-DSA exigiu alterações na integração FFI, nas operações assimétricas e nos modelos de chave, sem alterar a forma como os casos de uso acessam essas operações.

No plano de implementação, a biblioteca disponibiliza geração de chaves, assinatura e verificação, cifração simétrica, manipulação de certificados X.509, validação de cadeias, verificação de revogação, geração de requisições PKCS#10 e operações CMS/CAdES. Também foram integrados ML-KEM e ML-DSA conforme as interfaces definidas nos FIPS 203 e FIPS 204 (NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY, 2024b, 2024a), por meio da API EVP e de forma concentrada nas camadas inferiores da arquitetura.

No plano de validação, a suíte compreende 692 testes unitários e de integração, distribuídos em 49 arquivos de teste Linux e 8 arquivos de integração Android, organizados em 40 zonas de teste. A execução ocorreu com zero falhas e 68,4% de cobertura de código em Linux x86_64 e Android (arm64-v8a, armeabi-v7a, x86_64). As zonas de teste cobrem os principais subsistemas: geração de chaves, operações de hash, cifração simétrica, assinatura e verificação, manipulação de certificados X.509, validação de cadeias, verificação de revogação, geração de CSRs, operações CMS e algoritmos pós-quânticos.

A estratégia combinou testes unitários, que aferem o comportamento isolado de cada componente, e testes de integração, que verificam a interação entre as camadas, desde a invocação Dart até a execução nativa no OpenSSL. Esse arranjo assegura que os resultados criptográficos sejam consistentes entre as plataformas testadas.

Como contribuição técnica, a biblioteca demonstra a viabilidade de disponibilizar

operações de ICP e algoritmos pós-quânticos em Dart por meio de uma integração nativa comum. O uso do OpenSSL evita a reimplementação das primitivas criptográficas em Dart, embora a segurança do conjunto continue dependente da configuração da biblioteca nativa e da correção da camada FFI.

6.3 CONTRIBUIÇÕES DO TRABALHO

Para o LabSEC e a UFSC, a **dartcryptosec** representa um ativo tecnológico que poderá ser empregado em projetos de pesquisa e desenvolvimento que demandem operações criptográficas avançadas no ecossistema Dart/Flutter. A continuidade do projeto *Medida Inteligente*, que originou a demanda pela biblioteca, e de outras iniciativas do laboratório que envolvem dispositivos móveis e comunicação segura poderá se beneficiar diretamente das funcionalidades implementadas.

No âmbito da formação de recursos humanos, o desenvolvimento da biblioteca proporcionou ao autor deste trabalho uma imersão abrangente em temas como criptografia aplicada, engenharia de software e interoperabilidade entre linguagens de programação. Essas competências dialogam diretamente com as demandas do mercado de segurança digital e fortalecem a capacidade do laboratório de formar profissionais qualificados nessa área. Ademais, a disponibilização da biblioteca como código aberto sob licença permissiva permite que outros pesquisadores do laboratório a utilizem e a estendam em projetos futuros, ampliando o retorno do investimento institucional realizado.

6.4 LIMITAÇÕES E DIFICULDADES ENCONTRADAS

O desenvolvimento da **dartcryptosec** apresentou obstáculos técnicos relacionados à integração entre Dart e OpenSSL. Os principais desafios e as soluções adotadas foram documentados no Capítulo 4. Entre eles estiveram o tratamento da autenticação no modo GCM, a interpretação de datas ASN.1 e o gerenciamento de pilhas de certificados utilizadas nas operações CMS. A disponibilização dos algoritmos pós-quânticos também exigiu a compilação, o empacotamento e o carregamento do provedor utilizado nos ambientes Linux e Android.

No plano da integração via FFI, dois desafios foram recorrentes. O primeiro foi o mapeamento manual de tipos entre C e Dart: tipos como `unsigned char*`, `size_t` e ponteiros opacos exigem assinaturas equivalentes dos dois lados da fronteira. Uma diferença de largura, nulabilidade ou propriedade do ponteiro pode resultar em falhas difíceis de diagnosticar. O segundo desafio foi o gerenciamento da memória nativa. Como o coletor de lixo do Dart não conhece os recursos criados pelo OpenSSL, cada caminho de execução precisa associar a alocação à função de liberação correta e evitar tanto vazamentos quanto liberações duplicadas.

A curva de aprendizado do OpenSSL constituiu uma dificuldade transversal a todo o desenvolvimento. A documentação da biblioteca, embora extensa, apresenta fragmentação: funções correlatas estão dispersas entre páginas de manual independentes, e exemplos de uso são escassos para cenários avançados, como a construção programática de cadeias de certificados e a manipulação de estruturas CMS. A transição da série 3.x para a 4.0.0 introduziu mudanças na API que tornaram obsoletos diversos tutoriais e referências disponíveis na literatura técnica (OPENSSL PROJECT AUTHORS, 2026b). Consequentemente, a migração forçada para a API EVP, embora arquiteturalmente benéfica a longo prazo, exigiu a reescrita de componentes que, em versões anteriores, utilizavam funções de baixo nível mais intuitivas para o desenvolvedor iniciante.

Quanto à validação, os testes de integração foram conduzidos em Linux x86_64 e Android. Assim, não há evidência empírica suficiente para afirmar que o comportamento será equivalente em iOS, macOS e Windows. Essa limitação decorre das diferenças entre as cadeias de compilação, o carregamento das bibliotecas nativas e os ambientes de execução de cada plataforma.

A cobertura de código também constitui uma limitação reconhecida. A medição realizada em Linux foi de 68,4% e reflete apenas o subconjunto de código exercitado pela suíte unitária, pois não contabiliza módulos que não são importados pelos testes. As principais lacunas de cobertura concentram-se na camada de orquestração (*Flows*), conforme analisado na Seção 5.2. Algumas funcionalidades implementadas em estágio mais avançado do desenvolvimento foram validadas apenas parcialmente ou por inspeção manual, e a ampliação dessa cobertura é tratada entre os trabalhos futuros (Seção 6.5).

6.5 TRABALHOS FUTUROS

A `dartcryptosec` encontra-se funcional e validada em Linux e Android. Os trabalhos futuros incluem a ampliação dessa validação, novas operações criptográficas e atividades necessárias para facilitar a manutenção e a adoção da biblioteca.

- **Expansão da validação multiplataforma.** A execução da suíte de testes em iOS, macOS e Windows permitirá identificar diferenças relacionadas à compilação do OpenSSL, ao carregamento das bibliotecas nativas e ao comportamento da FFI em cada ambiente.
- **Ampliação da cobertura de testes.** A cobertura de código medida em Linux, de 68,4%, concentra suas lacunas na camada de orquestração (*Flows*, 58,3%), cujos caminhos alternativos e ramos de erro foram exercitados apenas parcialmente, conforme discutido na Seção 5.2. Trabalhos futuros devem priorizar a criação de testes dedicados a esses fluxos secundários e a módulos ainda não instrumentados pela suíte, elevando a cobertura efetiva e reduzindo o risco de regressões em caminhos pouco exercitados.

- **Suporte a HMAC.** A inclusão de operações de autenticação de mensagens baseadas em chave compartilhada ampliaria o uso da biblioteca em protocolos e aplicações que não dependem de certificados ou assinaturas assimétricas. A implementação pode utilizar a API EVP do OpenSSL e seguir a mesma organização adotada pelas demais operações criptográficas.
- **Avaliação do uso de *ffigen*.** A geração assistida dos *bindings* pode reduzir divergências em relação aos cabeçalhos oficiais do OpenSSL. Sua adoção deve ser avaliada com filtros que preservem apenas o subconjunto necessário e com testes que comparem o comportamento dos *bindings* gerados aos atuais (TEAM, D., 2024c).
- **Publicação no *pub.dev*.** Após a consolidação das validações prioritárias e a preparação do pacote para distribuição, a publicação facilitará o acesso à documentação existente e o uso da biblioteca por projetos externos.

Essas atividades concentram-se na redução das limitações identificadas durante o desenvolvimento e na preparação da biblioteca para uso fora do ambiente em que foi validada.

6.6 CUMPRIMENTO DOS OBJETIVOS

A Tabela 28 sumariza o grau de cumprimento de cada objetivo específico estabelecido no Capítulo 1.

Tabela 28 – Cumprimento dos objetivos específicos da pesquisa.

Objetivo Específico	Evidência de Cumprimento	Status
Analisar o estado da arte	Revisão de 3 bibliotecas Dart e 2 referências (Seção 3.2.2)	Cumprido
Projetar arquitetura modular	Arquitetura em 7 camadas (Seção 4.6)	Cumprido
Assegurar compatibilidade multiplataforma	Validação em Linux e Android (Seção 5.1)	Cumprido
Implementar funcionalidades ICP	36 métodos públicos: X.509, cadeias, assinaturas (Capítulo 4)	Cumprido
Implementar verificação de revogação	Suporte a CRL e OCSP (Capítulo 4)	Cumprido
Implementar CSRs	Criação e gerenciamento de CSRs PKCS#10 (Capítulo 4)	Cumprido
Suporte a primitivas criptográficas	SHA-256/512, SHA-3, AES-CBC/GCM (Seção 5.1)	Cumprido
Integrar com bibliotecas consolidadas	189 <i>bindings</i> FFI para o OpenSSL (Seção 4.5)	Cumprido
Desenvolver documentação abrangente	Referência da API, documentação arquitetural, exemplos de uso e comentários no código-fonte	Cumprido

Fonte: Elaborado pelo autor.

6.7 SÍNTESE QUANTITATIVA DOS RESULTADOS

A Tabela 29 consolida os principais indicadores quantitativos obtidos nos experimentos deste trabalho, organizados pelas dimensões de desempenho, cobertura e posicionamento competitivo.

Tabela 29 – Síntese quantitativa dos resultados experimentais.

Dimensão Analisada	Evidência Quantitativa
Latência simétrica (1 KB)	SHA-256: 2,8 μ s; AES-GCM: 5,7 μ s
Latência assimétrica + PQC	RSA-2048 <i>keygen</i> : 31,8 ms; ML-KEM-768 <i>keygen</i> : 0,07 ms
<i>Throughput</i> simétrico	SHA-256: 357k ops/s; AES-GCM: 180k ops/s
Diferença Android / Linux	2,1 $\times$ (PQ) a 16,2 $\times$ (ECDSA)
<i>Overhead</i> FFI (simétrico)	< 10 μ s por chamada
<i>Overhead</i> FFI (assimétrico)	< 0,01% do tempo total (RSA-2048)
Cobertura de código (Linux)	68,4%

Fonte: Elaborado pelo autor.

REFERÊNCIAS

- ADAMS, Carlisle; LLOYD, Steve. **Understanding PKI: Concepts, Standards, and Deployment Considerations**. Second. Boston, MA: Addison-Wesley Professional, 2002. ISBN 978-0321743091.
- ARANHA, Diego de Freitas. **Serviço de nomes e roteamento para redes de anonimização de tráfego**. 2007. Tese de Mestrado – Instituto de Computação, UNICAMP. Financiado pelo CNPq, processo número 131820/2005-2.
- BERNSTEIN, Daniel J. Introduction to Post-Quantum Cryptography. *In*: BERNSTEIN, Daniel J.; BUCHMANN, Johannes; DAHMEN, Erik (Ed.). **Post-Quantum Cryptography**. [S.l.]: Springer, 2009. P. 1–14.
- CASTLE, Bouncy. **Bouncy Castle Cryptography APIs**: Documentação Técnica. Cambridge, 2024. Disponível em: <https://www.bouncycastle.org/documentation.html>. Acesso em: 7 nov. 2024.
- CHEN, L.; AL., et. **Report on Post-Quantum Cryptography**: NISTIR 8105. Gaithersburg, 2024. Disponível em: <https://nvlpubs.nist.gov/nistpubs/ir/2024/NIST.IR.8105.pdf>. Acesso em: 7 nov. 2024.
- CONTRIBUTORS, PointyCastle. **PointyCastle: A Dart Library for Cryptography**: Documentação Técnica. Cambridge, 2024. Disponível em: <https://pub.dev/packages/pointycastle>. Acesso em: 7 nov. 2024.
- COOPER, D.; SANTESSON, S.; FARRELL, S.; BOEYEN, S.; HOUSLEY, R.; POLK, W. **RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile**. [S.l.], 2008. Disponível em: <https://datatracker.ietf.org/doc/html/rfc5280>. Acesso em: 7 nov. 2024.
- DAEMEN, J.; RIJMEN, V. **The Design of Rijndael: AES - The Advanced Encryption Standard**. [S.l.]: Springer, 2002.
- DANG, T. A.; ZHAO, B. **SHA-3: The New SHA Standard**. [S.l.: s.n.], 2015.
- DIFFIE, W.; HELLMAN, M. E. New Directions in Cryptography. **IEEE Transactions on Information Theory**, v. 22, n. 6, p. 644–654, 1976.

DUCAS, Léo; KILTZ, Eike; LEPOINT, Tancrede; LYUBASHEVSKY, Vadim; SCHWABE, Peter; SEILER, Gregor; STEHLÉ, Damien. CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme. *In*: 1. IACR Transactions on Cryptographic Hardware and Embedded Systems. [S.l.: s.n.], 2018. v. 2018, p. 238–268.

EASTLAKE, D. **RFC 6066: Transport Layer Security (TLS) Extensions: Extension Definitions**. [S.l.], 2011. Disponível em: <https://www.rfc-editor.org/rfc/rfc6066.html>. Acesso em: 11 jul. 2026.

EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. **ETSI TS 101 733 V2.2.1: Electronic Signatures and Infrastructures (ESI); CMS Advanced Electronic Signatures (CAAdES)**. Sophia Antipolis, 2013. Disponível em: https://www.etsi.org/deliver/etsi_TS/101700_101799/101733/02.02.01_60/ts_101733v020201p.pdf. Acesso em: 11 jul. 2026.

EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. **ETSI TS 101 903 V1.4.2: Electronic Signatures and Infrastructures (ESI); XML Advanced Electronic Signatures (XAdES)**. Sophia Antipolis, 2010. Disponível em: https://www.etsi.org/deliver/etsi_TS/101900_101999/101903/01.04.02_60/ts_101903v010402p.pdf. Acesso em: 11 jul. 2026.

EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. **ETSI TS 102 778-1 V1.1.1: Electronic Signatures and Infrastructures (ESI); PDF Advanced Electronic Signature Profiles; Part 1: PAdES Overview**. Sophia Antipolis, 2009. Disponível em: https://www.etsi.org/deliver/etsi_TS/102700_102799/10277801/01.01.01_60/ts_10277801v010101p.pdf. Acesso em: 11 jul. 2026.

EUROPEAN TELECOMMUNICATIONS STANDARDS INSTITUTE. **ETSI TS 119 182-1 V1.1.1: Electronic Signatures and Infrastructures (ESI); JAdES Digital Signatures; Part 1: Building Blocks and JAdES Baseline Signatures**. Sophia Antipolis, 2021. Disponível em: https://www.etsi.org/deliver/etsi_TS/119100_119199/11918201/01.01.01_60/ts_11918201v010101p.pdf. Acesso em: 11 jul. 2026.

FOUNDATION, OWASP. **Cryptography Best Practices: Guia de Segurança**. Salt Lake City, 2024. Disponível em: <https://owasp.org/www-project-cryptography>. Acesso em: 7 nov. 2024.

GUPTA, S.; PRAKASH, S. Performance analysis of cryptographic algorithms in mobile applications. **International Journal of Computer Applications**, v. 184, n. 5, p. 1–6, 2024.

HOUSLEY, R. **RFC 5652: Cryptographic Message Syntax (CMS)**. [S.l.], 2009. Disponível em: <https://www.rfc-editor.org/rfc/rfc5652.html>. Acesso em: 11 jul. 2026.

IDALINO, Thaís Bardini; VIGIL, Martin; OLIVEIRA, Gabriel Estevam de; ABEL, Paola de Oliveira; NA, Erik Gabriel; GOMES, Carlos Eduardo Vitorino; HARDT, Victor Henrique; TOMASZEWSK, Marcos; LOLI, Gabriel Bristot. **Estudo de Usabilidade de Assinaturas Digitais em Objetos Metrológicos**. Florianópolis, 2022. Relatório público do projeto Medida Inteligente, desenvolvido em parceria com o Inmetro. Disponível em: <https://repositorio.ufsc.br/handle/123456789/241804>. Acesso em: 11 jul. 2026.

AL-JANABI, Sufyan T. Faraj; HAMAD, Noor. A Framework for I-Voting based on Helios and Public-Key Certificates. **REVISTA AUS**, v. 26, n. 2, p. 234–243, set. 2019. Publicado em setembro de 2019.

JOHNSON, D.; MENEZES, A.; VANSTONE, S. **Elliptic Curve Cryptography**. [S.l.]: Springer, 2001.

KRAWCZYK, H.; BELLARE, M.; CANETTI, R. **RFC 2104: HMAC: Keyed-Hashing for Message Authentication**. [S.l.], 1997. Disponível em: <https://www.rfc-editor.org/rfc/rfc2104.html>. Acesso em: 11 jul. 2026.

LLC, Google. **Flutter Documentation: Guia de Desenvolvimento**. Cambridge, 2024. Disponível em: <https://flutter.dev/docs>. Acesso em: 7 nov. 2024.

MARTINS, Isac. **dartcryptosec: Biblioteca Criptográfica para Dart**. [S.l.: s.n.], 2026. Repositório institucional de código-fonte da UFSC. Disponível em: <https://codigos.ufsc.br/isac.martins/dartcryptosec>. Acesso em: 11 jul. 2026.

MENEZES, A. J.; OORSCHOT, P. C. van; VANSTONE, S. A. **Handbook of Applied Cryptography**. [S.l.]: CRC Press, 2018.

MENEZES, Alfred J.; OORSCHOT, Paul C. van; VANSTONE, Scott A. **Handbook of Applied Cryptography**. 1st. [S.l.]: CRC Press, 1996. Available at <http://cacr.uwaterloo.ca/hac/>.

- MILLER, Victor S. Use of Elliptic Curves in Cryptography. *In*: SPRINGER. **ADVANCES in Cryptology — CRYPTO’85**. [S.l.: s.n.], 1985. P. 417–426.
- MORIARTY, K.; KALISKI, B.; JONSSON, J.; RUSCH, A. **RFC 8017: PKCS #1: RSA Cryptography Specifications Version 2.2**. [S.l.], 2016. Disponível em: <https://www.rfc-editor.org/rfc/rfc8017.html>. Acesso em: 11 jul. 2026.
- NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. **Module-Lattice-based Digital Signature Standard**. [S.l.], 2024. Disponível em: <https://csrc.nist.gov/pubs/fips/204/final>.
- NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. **Module-Lattice-based Key-Encapsulation Mechanism Standard**. [S.l.], 2024. Disponível em: <https://csrc.nist.gov/pubs/fips/203/final>.
- NYSTROM, M.; KALISKI, B. **RFC 2986: PKCS #10: Certification Request Syntax Specification Version 1.7**. [S.l.], 2000. Disponível em: <https://www.rfc-editor.org/rfc/rfc2986.html>. Acesso em: 11 jul. 2026.
- OPENSSL PROJECT AUTHORS. **OpenSSL Documentation: Commands, Libraries, File Formats and Overviews**. [S.l.], 2026. Disponível em: <https://docs.openssl.org/master/>. Acesso em: 11 jul. 2026.
- OPENSSL PROJECT AUTHORS. **OpenSSL Guide: Migrating from Older OpenSSL Versions: openssl-guide-migration**. [S.l.], 2026. Disponível em: <https://docs.openssl.org/master/man7/openssl-guide-migration/>. Acesso em: 11 jul. 2026.
- REGEV, Oded. On Lattice-Based Cryptography. *In*: POST-QUANTUM Cryptography. [S.l.]: Springer, 2009. P. 1–20.
- RESCORLA, E. **RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3**. [S.l.], 2018. Disponível em: <https://www.rfc-editor.org/rfc/rfc8446.html>. Acesso em: 11 jul. 2026.
- RIVEST, Ronald L.; SHAMIR, Adi; ADLEMAN, Leonard. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. **Communications of the ACM**, ACM, v. 21, n. 2, p. 120–126, 1978.

SANTESSON, S.; MYERS, M.; ANKNEY, R.; MALPANI, A.; GALPERIN, S.; ADAMS, C. **RFC 6960: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP**. [S.l.], 2013. Disponível em: <https://www.rfc-editor.org/rfc/rfc6960.html>. Acesso em: 11 jul. 2026.

SANTOS, João Miguel Oliveira; OLIVEIRA, Alberto M. O. de. **Elliptic Curve Cryptography**. Berlin: Springer, 2017.

SHOR, Peter W. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In: IEEE. PROCEEDINGS 35th Annual Symposium on Foundations of Computer Science. [S.l.: s.n.], 1994. P. 124–134.

SHOUP, Victor. Efficiency Considerations for Cryptosystems Based on Discrete Logarithms and RSA. **Journal of Cryptology**, v. 18, n. 3, p. 237–286, 2005.

STALLINGS, W. **Cryptography and Network Security: Principles and Practice**. 7. ed. [S.l.]: Pearson, 2017.

STANDARDS, National Institute of; TECHNOLOGY. **FIPS 197: Advanced Encryption Standard (AES)**. Gaithersburg, 2001. Disponível em: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>. Acesso em: 7 nov. 2024.

STANDARDS, National Institute of; TECHNOLOGY. **Post-Quantum Cryptography**. [S.l.: s.n.], 2020. <https://csrc.nist.gov/Projects/post-quantum-cryptography>. Accessed: 2024-11-07.

STANDARDS, National Institute of; TECHNOLOGY. **Recommendation for Key Management Part 1: General**: NIST Special Publication 800-57. Gaithersburg, 2006. Disponível em: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf>. Acesso em: 7 nov. 2024.

STINSON, Douglas R. **Cryptography: Theory and Practice**. 3. ed. Boca Raton: CRC Press, 2006.

TEAM, Dart. **Dart Packages: crypto**: Package Documentation. Cambridge, 2024. Disponível em: <https://pub.dev/packages/crypto>. Acesso em: 7 nov. 2024.

TEAM, Dart. **Dart Programming Language**: Guia de Referência. Cambridge, 2024. Disponível em: <https://dart.dev>. Acesso em: 7 nov. 2024.

TEAM, Dart. **Using FFI in Dart**: Guia de Interoperabilidade. Cambridge, 2024. Disponível em: <https://dart.dev/guides/libraries/c-interop>. Acesso em: 7 nov. 2024.

TEAM, Dart Cryptography. **Dart Cryptography Package**: Documentação Técnica. Cambridge, 2024. Disponível em: <https://pub.dev/packages/cryptography>. Acesso em: 7 nov. 2024.

UNION, International Telecommunication. **X.509 Information Framework**: ITU-T Recommendation X.509. Geneva, 2016. Disponível em: <https://www.itu.int/rec/T-REC-X.509>. Acesso em: 7 nov. 2024.

7 GLOSSÁRIO

Este glossário reúne definições dos principais termos criptográficos empregados ao longo deste trabalho, organizados em ordem alfabética. As definições baseiam-se nas referências técnicas citadas nos capítulos anteriores e nos padrões internacionais adotados pela biblioteca `dartcryptosec`.

AEAD *Authenticated Encryption with Associated Data*. Modalidade de cifração que provê confidencialidade, autenticidade e integridade simultaneamente, vinculando dados autenticados adicionais (AAD) ao texto cifrado. O AES-GCM é um exemplo proeminente de AEAD.

AES *Advanced Encryption Standard*. Cifrador simétrico de bloco padronizado pelo NIST no documento FIPS 197, operando com blocos de 128 bits e chaves de 128, 192 ou 256 bits. É o algoritmo simétrico primário da `dartcryptosec`.

AES-GCM *Galois/Counter Mode* do AES. Modo de operação que combina o contador (CTR) para confidencialidade com a função de *hash* GHASH para autenticação, provendo AEAD. Requer um vetor de inicialização (IV) único por operação e gera uma *tag* de autenticação de 128 bits.

ASN.1 *Abstract Syntax Notation One*. Notação padrão internacional (ITU-T X.680) para descrição de estruturas de dados independentes de plataforma, amplamente utilizada em criptografia para codificação de certificados X.509, LCR, CSRs e mensagens CMS no formato DER.

CAdES *CMS Advanced Electronic Signatures*. Conjunto de perfis definidos pelo ETSI que estendem o CMS com atributos assinados, políticas de assinatura e informações de validação voltadas a assinaturas eletrônicas avançadas, conforme especificado no padrão ETSI TS 101 733.

CMS *Cryptographic Message Syntax*. Padrão definido na RFC 5652 que especifica formatos para assinatura digital, cifração e autenticação de mensagens, sendo a base para o CAdES e para a interoperabilidade em Infraestruturas de Chaves Públicas.

CSR *Certificate Signing Request*. Requisição de certificação no formato PKCS#10, contendo a chave pública do requerente e atributos de identidade, assinada com a chave privada correspondente e submetida a uma Autoridade Certificadora para emissão de certificado X.509.

CSPRNG *Cryptographically Secure Pseudo-Random Number Generator*. Gerador de números pseudoaleatórios cuja saída é computacionalmente indistinguível de aleatoriedade verdadeira, atendendo aos requisitos de imprevisibilidade exigidos para geração de chaves criptográficas e vetores de inicialização.

- ECDHE** *Elliptic Curve Diffie-Hellman Ephemeral*. Mecanismo de acordo de chaves baseado em curvas elípticas em que ambas as partes geram pares de chaves efêmeros para cada sessão, provendo *forward secrecy*. Utilizado no *handshake* TLS 1.3 como método primário de troca de chaves.
- ECDSA** *Elliptic Curve Digital Signature Algorithm*. Algoritmo de assinatura digital baseado na criptografia de curvas elípticas, padronizado pelo NIST no documento FIPS 186-5. A `dartcryptosec` suporta as curvas NIST P-256, P-384 e P-521.
- EVP** Acrônimo de *Envelope*. API de alto nível do OpenSSL que provê uma interface uniforme para operações criptográficas como *hashing*, cifração, assinatura e verificação, independentemente do algoritmo subjacente. A `dartcryptosec` utiliza a EVP em conjunto com APIs de alto nível para certificados X.509, CMS, OCSP, LCR e requisições de certificação.
- FFI** *Foreign Function Interface*. Mecanismo de interoperabilidade entre linguagens de programação que permite a invocação de funções compiladas em código nativo a partir de código gerenciado. No ecossistema Dart, o subsistema `dart:ffi` viabiliza a chamada direta a funções C do OpenSSL sem camada intermediária.
- FIPS** *Federal Information Processing Standards*. Conjunto de padrões publicados pelo NIST que estabelecem requisitos de segurança para sistemas criptográficos. O FIPS 140-3 define níveis de segurança para módulos criptográficos; o FIPS 203 e o FIPS 204 especificam os algoritmos pós-quânticos ML-KEM e ML-DSA, respectivamente.
- ICP** Infraestrutura de Chaves Públicas. Conjunto de políticas, procedimentos, *hardware*, *software* e pessoas necessários para criar, gerenciar, distribuir, armazenar e revogar certificados digitais baseados em criptografia de chave pública, estabelecendo uma cadeia de confiança entre entidades.
- IV** *Initialization Vector*. Vetor de inicialização. Valor aleatório ou pseudoaleatório utilizado como entrada adicional em modos de operação de cifradores simétricos para garantir que o mesmo texto claro produza textos cifrados distintos em cada operação. Deve ser único por chave no AES-GCM.
- LCR** Lista de Certificados Revogados. Estrutura de dados assinada pela Autoridade Certificadora que enumera os certificados cuja validade foi revogada antes do término de seu período de vigência, permitindo que entidades confiáveis verifiquem o *status* de revogação de um certificado.
- ML-KEM** *Module-Lattice-based Key-Encapsulation Mechanism*. Algoritmo pós-quântico de encapsulamento de chaves padronizado pelo NIST no documento FIPS 203, baseado no problema *Module Learning With Errors* (MLWE) sobre reticulados. A

`dartcryptosec` implementa as variantes ML-KEM-512, ML-KEM-768 e ML-KEM-1024.

ML-DSA *Module-Lattice-based Digital Signature Algorithm*. Algoritmo pós-quântico de assinatura digital padronizado pelo NIST no documento FIPS 204, baseado em problemas computacionais sobre reticulados. Na API EVP, sua utilização dispensa a configuração de um *digest* separado.

OCSP *Online Certificate Status Protocol*. Protocolo definido na RFC 6960 que permite a consulta em tempo real do *status* de revogação de um certificado X.509, oferecendo maior granularidade temporal que as LCRs e reduzindo a largura de banda necessária para a verificação.

PEM *Privacy Enhanced Mail*. Formato de codificação definido na RFC 7468 que representa dados binários — como certificados X.509 e chaves criptográficas — em texto ASCII encapsulado entre marcadores `---BEGIN` e `---END`, codificados em Base64.

PQC *Post-Quantum Cryptography*. Criptografia Pós-Quântica. Ramo da criptografia que desenvolve algoritmos resistentes a ataques de computadores quânticos, baseados em problemas matemáticos para os quais não se conhecem algoritmos quânticos eficientes, como reticulados e códigos corretores de erros.

RFC 3161 *Internet X.509 Public Key Infrastructure Time-Stamp Protocol*. Padrão definido pelo IETF que especifica o protocolo para emissão e verificação de carimbos de tempo criptográficos, utilizado para prover evidência de existência de um dado em determinado instante, essencial para assinaturas de longo prazo e não repúdio.

RAII *Resource Acquisition Is Initialization*. Padrão de projeto originário do C++ que vincula o ciclo de vida de um recurso ao tempo de vida de um objeto, garantindo liberação automática quando o objeto sai de escopo. Adaptado na `dartcryptosec` por meio de blocos `try/finally` para o gerenciamento de memória nativa do OpenSSL.

RSA *Rivest-Shamir-Adleman*. Algoritmo de criptografia assimétrica baseado na dificuldade de fatoração de inteiros, utilizado para cifração, assinatura digital e troca de chaves. A `dartcryptosec` suporta os tamanhos de chave RSA-2048 e RSA-4096.

SHA *Secure Hash Algorithm*. Família de funções de *hash* criptográfico padronizadas pelo NIST no documento FIPS 180-4 e FIPS 202. A `dartcryptosec` implementa SHA-256 e SHA-512 (família SHA-2) e SHA3-256 e SHA3-512 (família SHA-3 baseada em Keccak).

TLS *Transport Layer Security*. Protocolo criptográfico que provê comunicação segura sobre redes de computadores, oferecendo autenticação, confidencialidade e integri-

dade. O TLS 1.3 (RFC 8446) reduz o *handshake* para 1-RTT e remove algoritmos criptograficamente obsoletos.

X.509 Padrão internacional (ITU-T X.509) que define o formato de certificados digitais e listas de revogação. Um certificado X.509 vincula uma chave pública a uma identidade por meio da assinatura digital de uma Autoridade Certificadora, constituindo o elemento central de qualquer Infraestrutura de Chaves Públicas.

APÊNDICE A – REGISTROS DE DECISÃO ARQUITETURAL (ADR)

Este apêndice apresenta de forma sumarizada os sete Registros de Decisão Arquitetural (*Architecture Decision Records* — ADRs) que documentam as decisões de projeto mais significativas da biblioteca `dartcryptosec`.

A.1 ADR-001: MECANISMO DE INTEROPERABILIDADE — FFI (`DART:FFI`)

Status: Aceito e validado.

Decisão: Adotar `dart:ffi` como mecanismo exclusivo de interoperabilidade nativa, rejeitando `MethodChannel` para o caminho crítico de operações criptográficas.

Razões: (1) Eliminação de serialização — opera diretamente sobre ponteiros C; (2) Redução da superfície de ataque — resolução direta dos símbolos pelos *bindings*, sem registro de canal; (3) Desempenho de 80–95% do nativo — *overhead* de FFI $< 1 \mu\text{s}$ para tipos primitivos; (4) Controle síncrono determinístico — essencial para alocação/liberação de memória nativa.

Consequências: Positivas: latência mínima, cobertura de 96,8% nos *bindings*, simplicidade de compilação cruzada. Negativas: gerenciamento manual de memória nativa, ausência de isolamento de processos.

A.2 ADR-002: ESCOLHA DO BACKEND CRIPTOGRÁFICO — OPENSSL 4.0.0

Status: Aceito e validado.

Decisão: Adotar OpenSSL 4.0.0 como *backend* criptográfico exclusivo, compilado com FIPS e com os provedores `default` e `fips`.

Razões: (1) Maturidade do ecossistema — biblioteca criptográfica mais utilizada globalmente; (2) Suporte a algoritmos pós-quânticos via provedores (ML-KEM, ML-DSA); (3) Conformidade FIPS 140-3; (4) Documentação de migração estruturada (série 3.x → 4.x).

A.3 ADR-003: BINDINGS MANUAIS VS. AUTOMÁTICOS (`FFIGEN`)

Status: Aceito e validado.

Decisão: Escrever manualmente 189 *bindings* FFI, rejeitando a geração automática via `ffigen`.

Razões: (1) Superfície de ataque reduzida — apenas funções utilizadas são expostas; (2) Documentação linha a linha — cada *binding* é explícito e auditável; (3) Manutenção localizada — cabeçalhos de seção agrupam funções por subsistema.

A.4 ADR-004: ARQUITETURA EM 7 CAMADAS

Status: Aceito e validado.

Decisão: Organizar a biblioteca em sete camadas sobrepostas com responsabilidades bem definidas: OpenSSL → FFI → Operations → Models → Flows → Facade → API.

Razões: (1) Separação de responsabilidades — cada camada pode ser testada isoladamente; (2) Evolução independente — alterações na camada FFI não afetam a API pública; (3) Testabilidade — 692 testes com 68,4% de cobertura global.

A.5 ADR-005: PADRÃO SINGLETON PARA A FACADE

Status: Aceito e validado.

Decisão: Implementar `PluginCryptoAPI` como *singleton* com inicialização única do OpenSSL.

Razões: (1) Inicialização única do OpenSSL — `OPENSSL_init_crypto()` deve ser chamada uma única vez; (2) Compartilhamento de estado — contextos criptográficos reutilizados entre chamadas; (3) Simplicidade da API pública — ponto de acesso único para 36 métodos.

A.6 ADR-006: ORGANIZAÇÃO DOS CASOS DE USO NA CAMADA DE FLOWS

Status: Adotado.

Decisão: Organizar processos compostos em classes de *Flow*, responsáveis por coordenar operações criptográficas de nível inferior. As dependências necessárias a cada caso de uso são fornecidas por meio de interfaces e do `CryptoContext`.

Razões: (1) Separação de responsabilidades, mantendo a coordenação de processos fora da API pública e das operações de baixo nível; (2) testabilidade, pois as dependências podem ser substituídas durante os testes; (3) manutenção, pois alterações em um caso de uso permanecem concentradas em seu respectivo fluxo.

A.7 ADR-007: EMPACOTAMENTO DE BIBLIOTECAS NATIVAS

Status: Aceito e validado.

Decisão: Empacotar `libcrypto.so.4` e `libssl.so.4` como `jniLibs` no Android e como `.so` no Linux, com compilação cruzada para três ABIs Android (`arm64-v8a`, `armeabi-v7a`, `x86_64`).

Razões: (1) Distribuição autônoma — sem dependência de OpenSSL pré-instalado; (2) Reprodutibilidade — versão controlada do OpenSSL (4.0.0, `enable-fips`); (3) Compilação cruzada unificada — mesma base de código C para todas as plataformas.

APÊNDICE B – MODELO DE AMEAÇAS DETALHADO

Este anexo complementa a Seção 4.6.4 do Capítulo 5, apresentando o catálogo completo das 15 ameaças STRIDE identificadas na `dartcryptosec`, com justificativas detalhadas de pontuação e contramedidas.

B.1 MATRIZ COMPLETA DE AMEAÇAS (T1–T15)

A metodologia STRIDE (Kohnfelder & Garg, 1999; Microsoft SDL) organiza ameaças em seis categorias mutuamente excludentes. A Tabela 30 apresenta a definição de cada categoria aplicada ao contexto da `dartcryptosec`.

Tabela 30 – Categorias STRIDE e propriedades de segurança violadas.

Categoria	Propriedade	Aplicação à <code>dartcryptosec</code>
Spoofing	Autenticidade	Substituição da <code>libcrypto.so</code> no caminho de carregamento; falsificação de certificados X.509
Tampering	Integridade	Corrupção de buffers na fronteira Dart↔C; adulteração de IV em AES-GCM
Repudiation	Não repúdio	Ausência de registro de auditoria ou de evidências suficientes sobre a operação
Information Disclosure	Confidencialidade	Persistência de chaves em memória; vazamento via error oracles
Denial of Service	Disponibilidade	CMS malformado; RSA com chaves excessivas; bombas algorítmicas
Elevation of Privilege	Autorização	Buffer overflow FFI; injeção de biblioteca maliciosa

Fonte: Adaptado de Kohnfelder & Garg (1999) e Microsoft SDL.

A Tabela 31 apresenta o catálogo completo das 15 ameaças identificadas, com pontuação de probabilidade (P), impacto (I), risco ($R = P \times I$) e prioridade de mitigação. Os critérios de pontuação seguem a escala: P de 1 (muito baixa) a 5 (muito alta); I de 1 (muito baixo) a 5 (crítico). O risco é classificado como Baixo (1–5), Médio (6–11), Alto (12–19) ou Crítico (20–25). A prioridade segue: P1 ($R \geq 20$, ação imediata), P2 (R 12–19, próximo ciclo), P3 (R 6–11, monitoramento), P4 (R 1–5, aceitar risco).

Tabela 31 – Matriz completa de 15 ameaças STRIDE com pontuação de risco.

ID	Ameaça	Cat.	P	I	R	Prior.
T1	Substituição de <code>libcrypto.so</code> no caminho de carregamento	S	2	5	10	P3
T2	Falha ou indisponibilidade causada por CMS/PKCS#7 malformado	D	3	4	12	P2
T3	Aceitação de certificado X.509 sem validação de identidade	S	3	3	9	P3
T4	Permanência de material sensível na memória após o uso	I	3	4	12	P2
T5	Vazamento por <i>padding oracle</i> em RSA PKCS#1 v1.5	I	2	4	8	P3
T6	Ausência de registro de auditoria criptográfica	R	3	3	9	P3
T7	Vazamento por canal lateral temporal	I	2	5	10	P3
T8	Exaustão de CPU por geração de chave RSA sem limite adequado	D	2	2	4	P4
T9	Exaustão de recursos em operações de <i>streaming</i> sem limite total	D	2	3	6	P3
T10	Escrita fora dos limites na fronteira FFI	T	2	5	10	P3
T11	Acesso a recurso nativo após sua liberação	T	1	5	5	P4
T12	Desreferenciamento de ponteiro nulo ao resolver algoritmo	T	1	4	4	P4
T13	Elevação de privilégio decorrente de corrupção de memória	E	1	5	5	P4
T14	Carregamento de biblioteca nativa maliciosa no Linux	E	2	5	10	P3
T15	Omissão da consulta de revogação por LCR/OCSP pela aplicação consumidora	S	3	4	12	P2

Fonte: Elaborado pelo autor.

A distribuição de risco resultante é: 3 ameaças de risco alto (20%) — T2, T4, T15; 8 ameaças de risco médio (53%) — T1, T3, T5, T6, T7, T9, T10, T14; e 4 ameaças de risco baixo (27%) — T8, T11, T12, T13. Nenhuma ameaça atinge o nível crítico ($R \geq 20$). Por categoria STRIDE, a distribuição é: Spoofing (3, 20%), Tampering (3, 20%), Repudiation (1, 7%), Information Disclosure (3, 20%), Denial of Service (3, 20%) e Elevation of Privilege (2, 13%). As três ameaças de prioridade P2 — T2, T4 e T15 — constituem o foco prioritário de mitigação.

B.2 ANÁLISE DOS VETORES DE AMEAÇA

Os vetores detalhados a seguir complementam a matriz STRIDE, identificando condições de exploração, impacto e estado das medidas de tratamento consideradas no projeto.

B.2.1 Substituição da Biblioteca Nativa

Tabela 32 – Vetores relacionados à substituição da biblioteca nativa.

ID	Nó Folha	Prob.	Impacto	Mitigação	Status
L1	Substituição da biblioteca no carregamento	Média	Crítico (5)	Empacotamento com checksum (pendente)	Parcial
L2	Library injection Android	Baixa	Crítico (5)	Sandbox + jniLibs isolado	Mitigado
L3	Static linking bypass iOS	Muito baixa	Alto (4)	Code signing + W^X	Mitigado

Fonte: Elaborado pelo autor.

B.2.2 Exposição de Material Sensível na Memória

Tabela 33 – Vetores relacionados à exposição de material sensível na memória.

ID	Nó Folha	Prob.	Impacto	Mitigação	Status
L4	Memory dump (cold boot)	Muito baixa	Crítico (5)	ZeroizationVerifier; sem mlock()	Parcial
L5	Swap file inspection	Média	Alto (4)	RSS baseline 146 MB; sem vazamento	Parcial
L6	/proc/pid/mem reading	Baixa	Alto (4)	Sem proteção Yama/ptrace	Ausente
L7	GC residual data	Média	Médio (3)	ZeroizationVerifier; NativeFinalizer pendente	Parcial
L8	Side-channel timing	Baixa	Médio (3)	OpenSSL constant-time + RFC 6979	Mitigado

Fonte: Elaborado pelo autor.

B.2.3 Corrupção de Memória na Fronteira FFI

Tabela 34 – Vetores relacionados à corrupção de memória na fronteira FFI.

ID	Nó Folha	Prob.	Impacto	Mitigação	Status
L9	Buffer overflow FFI	Baixa	Crítico (5)	Bound check no Dart antes da chamada	Mitigado
L10	Use-after-free FFI	Muito baixa	Crítico (5)	Padrão try/finally; ponteiros não persistidos	Mitigado
L11	Double free FFI	Muito baixa	Crítico (5)	Flags de controle; free único por try/finally	Mitigado
L12	Stack overflow CMS nesting	Baixa	Alto (4)	Validação de profundidade pendente	Ausente

Fonte: Elaborado pelo autor.

B.2.4 Exaustão de Recursos

Tabela 35 – Vetores relacionados à exaustão de recursos.

ID	Nó Folha	Prob.	Impacto	Mitigação	Status
L13	CMS bomb (aninhamento)	Alta	Médio (3)	Validação de profundidade pendente	Ausente
L14	RSA-16384 keygen	Média	Médio (3)	Teto em 8192 bits recomendado	Parcial
L15	Arquivo terabytes streaming	Baixa	Médio (3)	Chunks limitam memória; sem teto total	Parcial
L16	Memory exhaustion (malformed input)	Média	Alto (4)	Wrapper defensivo pendente	Ausente

Fonte: Elaborado pelo autor.

B.3 MAPA DE SUPERFÍCIE DE ATAQUE

A superfície de ataque da `dartcryptosec` decompõe-se em quatro pontos de entrada, quatro elementos da fronteira FFI e quatro zonas de processamento no OpenSSL. A Figura 12 apresenta o diagrama estrutural com anotações STRIDE nas arestas de fluxo de dados.

Ponto de Entrada	Ameaças STRIDE	Fronteira FFI	Processamento (C)
E1: Dados Dart	S, T, I, D, E	F1: Pointer<Uint8>	P1: OpenSSL ops
E2: Arquivos	S, T, D, R	F2: Length param	P2: Error queue
E3: System API	I	F3: Function ptrs	P3: BIO buffers
E4: Carregamento	S, T, E	F4: Memory alloc	P4: Key material

S=Spoofing, T=Tampering, R=Repudiation, I=Information Disclosure, D=Denial of Service, E=Elevation of Privilege.

Figura 12 – Mapa estrutural da superfície de ataque com anotações STRIDE nas arestas de fluxo de dados.

Fonte: Elaborado pelo autor.

A fronteira FFI constitui o ponto de contato de maior criticidade. A Tabela 36 detalha os quatro elementos que a compõem.

Tabela 36 – Elementos da fronteira FFI e riscos associados.

Elemento	Característica	Contramedida
F1: Pointer crossing	Dados trafegam como ponteiro bruto Dart→C	Uint8List imutável; validação antes de malloc
F2: Length parameter	Tamanho passado como int (Dart) → size_t (C)	Validação estrita de chave (16/32 bytes) e IV (16 bytes)
F3: Function pointers	Conversão de closure Dart em ponteiro C	Callbacks de curta duração; sem persistência além do escopo
F4: Memory allocation	malloc/calloc de Dart; free manual	try/finally garante free único; MemoryTracker monitora RSS

Fonte: Elaborado pelo autor.

O resumo de cobertura de contramedidas por categoria STRIDE é apresentado na Tabela 37.

Tabela 37 – Cobertura de contramedidas por categoria STRIDE.

Categoria STRIDE	Ameaças	Mitigadas	Parciais	Ausentes
Spoofing	3	0	3	0
Tampering	3	2	1	0
Repudiation	1	0	0	1
Information Disclosure	3	0	3	0
Denial of Service	3	0	1	2
Elevation of Privilege	2	0	1	1
Total	15	2 (13,3%)	9 (60,0%)	4 (26,7%)

Fonte: Elaborado pelo autor.

A consolidação considera as 15 ameaças T1–T15 da matriz principal, e não os nós folha empregados no detalhamento dos vetores de ataque. Foram classificadas como

mitigadas T11 e T12; como parcialmente mitigadas T1, T3, T4, T5, T7, T9, T10, T13 e T15; e como ausentes T2, T6, T8 e T14. No caso de T15, a biblioteca disponibiliza os mecanismos de consulta e processamento de LCR e OCSP, mas a aplicação consumidora ainda precisa incorporá-los explicitamente ao fluxo de validação para evitar a aceitação de um certificado revogado.

B.4 DFD: AES-256-GCM COM AAD

O Diagrama de Fluxo de Dados (DFD) para a operação de cifração AES-256-GCM com Dados Autenticados Adicionais (AAD) é apresentado na Figura 13. O fluxo modela 9 processos distribuídos em 5 zonas, com a fronteira FFI demarcada entre as zonas de validação Dart e o pipeline OpenSSL.

Entrada Dart	Validação Dart	Fronteira FFI	Pipeline OpenSSL C
P1: Chave (32 B) P2: IV gerado P3: AAD opcional P4: Plaintext	V1: <code>len==32</code> V2: <code>RAND_bytes</code> V3: <code>!=null?</code> V4: <code>malloc(len)</code>	FK: key ptr FI: iv ptr FA: aad ptr FP: plain ptr	P5: <code>EVP_EncryptInit_ex</code> (ctx, AES-256-GCM) P6: <code>EVP_EncryptUpdate</code> (AAD + plaintext) P7: <code>EVP_EncryptFinal_ex</code> P8: <code>EVP_CTRL_GCM_GET_TAG</code> P9: <code>CryptoSuccess</code> (ciphertext + tag)

Figura 13 – Diagrama de Fluxo de Dados — AES-256-GCM Encrypt com AAD (9 processos).

Fonte: Elaborado pelo autor.

A Tabela 38 sumariza as propriedades de segurança garantidas em cada etapa do pipeline.

Tabela 38 – Anotações de segurança do pipeline AES-256-GCM.

Processo	Propriedade	Mecanismo
P1	Confidencialidade da chave	Chave de 256 bits (14 rodadas); ZeroizationVerifier
P2	Unicidade do IV	<code>RAND_bytes</code> com 100% unicidade em 100 amostras
P3	Integridade do AAD	GHASH vincula AAD ao ciphertext
P5	Inicialização segura	<code>EVP_aes_256_gcm()</code> com FIPS 140-3 (enable-fips)
P6–P7	Criptografia autenticada	AEAD: 718,4 MB/s encrypt; 795,5 MB/s decrypt
P8	Tag authentication	128 bits; probabilidade de falsificação: 2^{-128}
P9	Tipos imutáveis	<code>CryptoSuccess<AesGcmResult></code> com sealed classes

Fonte: Elaborado pelo autor.

O gerenciamento de memória do fluxo segue o padrão de alocação explícita com liberação garantida via `try/finally`. As alocações compreendem: `key_ptr` (32 bytes), `iv_ptr` (12 bytes), `aad_ptr` (tamanho variável), `plaintext_ptr` (tamanho do texto claro),

`ciphertext_ptr` (tamanho + 16 bytes para margem GCM), `tag_ptr` (16 bytes) e `ctx` (contexto `EVP_CIPHER_CTX`). A métrica de consumo de memória registra RSS delta máximo de 17.936 KB (17,5 MB) entre baseline e pico, sem vazamento detectado.

O fluxo de descriptografia difere em três aspectos críticos: (1) a tag é configurada antes da operação (`EVP_CTRL_GCM_SET_TAG`), e não extraída após; (2) a verificação da tag ocorre inline em `EVP_DecryptFinal_ex`, eliminando a necessidade de comparação manual (vulnerável a *timing attacks*); (3) a falha de autenticação é sinalizada por meio do subtipo específico `AesGcmAuthFailure` da hierarquia `CryptoError`.

B.5 MATRIZ DE COBERTURA FUNCIONAL (36 × 13)

A matriz de cobertura funcional apresentada na Tabela 39 avalia cada um dos 36 métodos da `dartcryptosec` contra 13 critérios de qualidade e segurança. A legenda de avaliação emprega os símbolos: ✓ (completo/verificado), ≈ (parcial/implementado com ressalvas), × (ausente/não implementado) e — (não aplicável). Métodos 1–25 compõem a API pública; métodos 26–36 são internos/de infraestrutura.

Tabela 39 – Matriz de cobertura funcional — API pública (métodos 1–25).

#	Método	Impl.	Linux	Android	Cob.	Memory
1	getOpenSSLVersion()	✓	✓	✓	85,7%	✓
2	randomBytes(int)	✓	✓	✓	96,8%	✓
3	sha256(UInt8List)	✓	✓	✓	96,8%	✓
4	sha512(UInt8List)	✓	✓	✓	96,8%	✓
5	sha3_256(UInt8List)	✓	✓	✓	96,8%	✓
6	sha3_512(UInt8List)	✓	✓	✓	96,8%	✓
7	aes128CbcEncrypt(...)	✓	✓	✓	96,8%	✓
8	aes128CbcDecrypt(...)	✓	✓	✓	96,8%	✓
9	aes256CbcEncrypt(...)	✓	✓	✓	96,8%	✓
10	aes256CbcDecrypt(...)	✓	✓	✓	96,8%	✓
11	aes128GcmEncrypt(...)	✓	✓	✓	96,8%	✓
12	aes128GcmDecrypt(...)	✓	✓	✓	96,8%	✓
13	aes256GcmEncrypt(...)	✓	✓	✓	96,8%	✓
14	aes256GcmDecrypt(...)	✓	✓	✓	96,8%	✓
15	generateRsaKeyPair(int)	✓	✓	✓	96,8%	✓
16	generateEcKeyPair(String)	✓	✓	✓	96,8%	✓
17	sign(data, key, ...)	✓	✓	✓	66,2%	✓
18	verify(data, key, sig, ...)	✓	✓	✓	66,2%	✓
19	rsaEncrypt(key, data)	✓	✓	✓	66,2%	✓
20	rsaDecrypt(key, data)	✓	✓	✓	66,2%	✓
21	parseX509Certificate(...)	✓	✓	✓	66,2%	✓
22	verifyX509Certificate(...)	✓	✓	✓	66,2%	✓
23	cmsSign(data, cert, key)	✓	✓	✓	66,2%	✓
24	cmsVerify(data, ...)	✓	✓	✓	66,2%	✓
25	cmsEncrypt(data, cert)	✓	✓	✓	66,2%	✓

Fonte: Elaborado pelo autor.

Tabela 40 – Matriz de cobertura funcional — métodos internos (26–36).

#	Método	Impl.	Linux	Android	Cob.	Mem.
26	AesOperations. _validateAesKeyLength(...)	✓	✓	✓	96,8%	✓
27	AsymmetricOperations. _loadPrivateKey(...)	✓	✓	✓	66,2%	✓
28	X509Operations. _parseExtensions(...)	✓	✓	✓	66,2%	✓
29	CmsOperations. _buildSignerCerts(...)	✓	✓	✓	66,2%	✓
30	CmsOperations. _cmsDecryptInternal(...)	✓	✓	✓	66,2%	✓
31	CertificateBuilder. _buildDer()	✓	✓	✓	66,2%	✓
32	KeyCreatorFactory. create(...)	✓	✓	✓	66,2%	✓
33	MemoryTracker. trackAlloc(ptr, size)	✓	✓	≈	66,1%	✓
34	MemoryTracker. trackFree(ptr)	✓	✓	≈	66,1%	✓
35	ZeroizationVerifier. verify(ptr, size)	✓	✓	≈	66,1%	✓
36	NativeLoader. load()	✓	≈	✓	44,4%	✓

Fonte: Elaborado pelo autor.

A Tabela 41 sintetiza a completude agregada por critério de avaliação.

Tabela 41 – Completude por critério de avaliação (36 métodos × 13 critérios).

Critério	✓	≈	×	—	% Completo
Implementado	36	0	0	0	100,0%
Test. Unit. (Linux)	35	1	0	0	97,2%
Test. Integ. (Android)	33	3	0	0	91,7%
Cobertura de Código	35	1	0	0	97,2%
Documentação API	25	0	0	11	100,0%
Validação de Entrada	25	7	0	4	78,1%
Tratamento de Erro	36	0	0	0	100,0%
Memory Safe	36	0	0	0	100,0%
Constant-Time	12	7	0	17	63,2%
FIPS	32	0	0	4	100,0%
PQC	1	3	0	32	25,0%
Benchmark	24	1	0	11	96,0%
Bound Check	24	6	0	6	80,0%

Fonte: Elaborado pelo autor.

As métricas consolidadas da matriz são: cobertura global de código de 68,4% (2.871/4.195 linhas), cobertura FFI de 96,8% (298/308 linhas), cobertura Crypto de 66,2% (1.595/2.408 linhas) e cobertura Metrics de 66,1% (966/1.462 linhas). A bateria de testes totaliza 692 testes (515 unitários Linux + 177 integração Android), todos executados com zero falhas de código. As 63 verificações de segurança documentadas foram integralmente aprovadas. A cobertura de 44,4% do `NativeLoader.load()` reflete exclusivamente o caminho Linux; a ramificação Android é exercitada pelos 177 testes de integração, mas não

gera `lcov.info`. A avaliação “parcial” para *constant-time* em AES-CBC e RSA decorre da dependência da configuração do OpenSSL 4.0.0, que implementa contramedidas de tempo constante para curvas NIST (RFC 6979) e para AES via AES-NI, não de uma deficiência da `dartcryptosec`.

APÊNDICE C – CÓDIGO-FONTE DA DARTCRYPTOSEC

Este apêndice reúne as orientações para acessar, configurar e utilizar o código-fonte da biblioteca `dartcryptosec`. O projeto está disponível no repositório institucional da UFSC em <https://codigos.ufsc.br/isac.martins/dartcryptosec>, que também contém os exemplos de uso e a documentação de referência da API.

C.1 ACESSO E UTILIZAÇÃO DO CÓDIGO

O repositório é público e seu ramo principal contém o pacote Dart/Flutter, a aplicação utilizada nos testes de integração, as bibliotecas nativas e a documentação técnica. Para obter uma cópia local, utiliza-se:

```
git clone https://codigos.ufsc.br/isac.martins/dartcryptosec.git
cd dartcryptosec
```

Os principais diretórios do repositório são:

- `plugin_crypto`: biblioteca Dart/Flutter e binários nativos;
- `tcc_test_app`: aplicação usada na validação em Android;
- `docs`: referência técnica e guias do projeto;
- `tool`: scripts de compilação reproduzível.

O ambiente requer Flutter e Dart, além das dependências nativas indicadas no arquivo `README.md`. As dependências do pacote são instaladas com:

```
cd plugin_crypto
flutter pub get
```

Para consumir a biblioteca a partir de outro projeto localizado ao lado do repositório, acrescenta-se ao arquivo `pubspec.yaml`:

```
dependencies:
  plugin_crypto:
    path: ../dartcryptosec/plugin_crypto
```

C.1.1 Uso básico e validação

Após a resolução das dependências com `flutter pub get`, a fachada da biblioteca pode ser utilizada diretamente. O exemplo a seguir calcula o resumo SHA-256 de uma mensagem:

```
import 'dart:convert';
import 'package:plugin_crypto/plugin_crypto.dart';

void main() {
  final api = PluginCryptoAPI.instance;
  final digest = api.sha256(utf8.encode('mensagem'));
  print(digest);
}
```

No Linux x86_64, a suíte automatizada é executada informando o diretório que contém as bibliotecas compartilhadas do OpenSSL:

```
cd plugin_crypto
LD_LIBRARY_PATH=$PWD/native/linux/x86_64:$LD_LIBRARY_PATH flutter test
```

Para preparar a aplicação Android de validação, executam-se `flutter pub get` e `flutter build apk -debug` no diretório `tcc_test_app`. A documentação complementar encontra-se em:

- `docs/API.md`: referência da API pública;
- `docs/FLUXOS.md`: descrição dos fluxos criptográficos;
- `docs/GUIA_TESTES.md`: execução e organização dos testes;
- `docs/COMPILACAO.md`: compilação das bibliotecas nativas.

Dartcryptosec: Desenvolvimento de uma Biblioteca Criptográfica Multiplataforma para Dart via FFI com OpenSSL

Isac Martins¹, Thaís Bardini Idalino¹, Martín A. G. Vigil¹

¹Departamento de Informática e Estatística
Universidade Federal de Santa Catarina (UFSC) – Florianópolis – SC – Brasil

isacmartins.223@gmail.com, {thais.idalino,martin.vigil}@ufsc.br

Abstract. *This work presents dartcryptosec, a cross-platform cryptographic library for Dart, validated on Linux and Android, that integrates the OpenSSL library through Dart’s Foreign Function Interface (FFI). Inspired by LabSEC’s libcryptosec, the library offers simplified routines focused on Public Key Infrastructure (PKI), abstracting OpenSSL’s low-level complexity, and includes support for the post-quantum algorithms ML-KEM and ML-DSA. The implementation was validated through automated tests, code coverage analysis, performance benchmarks, and comparison with established solutions such as Bouncy Castle, demonstrating its suitability for scenarios requiring PKI and interoperability in mobile applications.*

Resumo. *Este trabalho apresenta a dartcryptosec, uma biblioteca criptográfica multiplataforma para Dart, validada em Linux e Android, que integra a biblioteca OpenSSL por meio da Foreign Function Interface (FFI) do Dart. Inspirada na libcryptosec do LabSEC, a biblioteca oferece rotinas simplificadas focadas em Infraestrutura de Chaves Públicas (ICP), abstraindo a complexidade de baixo nível do OpenSSL, e inclui suporte aos algoritmos pós-quânticos ML-KEM e ML-DSA. A implementação foi validada por testes automatizados, análise de cobertura de código, benchmarks de desempenho e comparação com soluções consagradas como Bouncy Castle, demonstrando sua adequação a cenários que demandam ICP e interoperabilidade em aplicações móveis.*

1. Introdução

No cenário contemporâneo do desenvolvimento de software, a segurança da informação tornou-se componente indispensável, sobretudo em aplicações distribuídas e móveis multiplataforma. A ampla adoção de dispositivos inteligentes e serviços conectados ampliou a superfície de ataque dos sistemas, impondo desafios crescentes à proteção de dados, à garantia de integridade e à autenticação de usuários, serviços e dispositivos [Stallings 2017]. Nesse contexto, a Infraestrutura de Chaves Públicas (ICP) constitui um dos pilares para assegurar autenticidade, integridade e confidencialidade nas comunicações digitais, por meio de criptografia assimétrica, certificados X.509, autoridades certificadoras, cadeias de confiança e mecanismos de revogação [Adams and Lloyd 2002].

No ecossistema móvel, destaca-se o Flutter, *framework* multiplataforma que emprega a linguagem Dart. Dart, contudo, é uma linguagem independente, com compilação nativa para múltiplas plataformas e suporte a *Foreign Function Interface* (FFI)

para integração com código C, características que a tornam relevante para componentes de software voltados à segurança [Dart Team 2024b]. Apesar disso, o ecossistema Dart apresenta lacunas significativas no suporte a funcionalidades avançadas de criptografia e ICP: não há biblioteca que ofereça, de forma integrada, validação de cadeias de certificados, manipulação de certificados X.509, verificação de revogação e suporte a formatos padronizados de assinatura como o *Cryptographic Message Syntax* (CMS) e o *CMS Advanced Electronic Signatures* (CAvES). A biblioteca Pointycastle, por exemplo, limita-se a primitivas de baixo nível, sem alcançar a camada de infraestrutura [Pointycastle Contributors 2024]. Essa carência força desenvolvedores a manter múltiplas bases de código com APIs específicas de cada plataforma, elevando a complexidade de manutenção e o risco de inconsistências de segurança [Bouncy Castle 2024, Cooper et al. 2008].

O problema foi identificado durante o projeto *Medida Inteligente*, conduzido pelo Laboratório de Segurança em Computação (LabSEC) da Universidade Federal de Santa Catarina (UFSC) em parceria com o Inmetro [Idalino et al. 2022]. A experiência com o *PluginCrypto*, prova de conceito desenvolvida durante o projeto e inspirada na *libcryptosec* [Aranha 2007], *wrapper* orientado a objetos do OpenSSL mantido pelo LabSEC, motivou a expansão do escopo para além dos casos de uso originais, dando origem à *dartcryptosec*. O trabalho justifica-se, ainda, pelo desafio técnico de reaproveitar em Dart, via FFI, uma biblioteca criptográfica madura escrita em C: o OpenSSL reúne décadas de implementação e auditoria, mas sua API nativa exige cuidados com gerenciamento de memória, conversão de tipos e compatibilidade entre plataformas, tema sobre o qual a documentação em Dart ainda é escassa [Dart Team 2024c, OpenSSL Project Authors 2026a].

Diante disso, este artigo apresenta a *dartcryptosec*¹ [Martins 2026], biblioteca criptográfica para Dart cujo objetivo geral é fornecer suporte abrangente a criptografia e a funcionalidades de ICP, incluindo verificação de assinaturas digitais, validação de cadeias de certificados, manipulação de certificados X.509 e suporte aos formatos CMS e CAvES. São objetivos específicos: analisar o estado da arte das bibliotecas criptográficas em Dart [Dart Cryptography Team 2024]; projetar uma arquitetura modular com integração direta ao OpenSSL via FFI; assegurar compatibilidade multiplataforma (Linux e Android, com infraestrutura de *build* para iOS e Windows) [Google LLC 2024]; implementar funcionalidades essenciais de ICP, incluindo verificação de revogação por Listas de Revogação de Certificados (CRLs) e pelo protocolo *Online Certificate Status Protocol* (OCSP) [Santesson et al. 2013] e manipulação de *Certificate Signing Requests* (CSRs) [Nystrom and Kaliski 2000]; oferecer primitivas criptográficas fundamentais [National Institute of Standards and Technology 2001]; e produzir documentação que facilite a adoção pela comunidade [OWASP Foundation 2024].

O restante do artigo está organizado da seguinte forma: a Seção 2 apresenta a fundamentação teórica; a Seção 3 descreve a metodologia; a Seção 4 detalha o desenvolvimento da biblioteca; a Seção 5 discute os resultados obtidos; e a Seção 6 apresenta as conclusões e os trabalhos futuros.

¹Código-fonte disponível no repositório institucional da UFSC: <https://codigos.ufsc.br/isac.martins/dartcryptosec>.

2. Fundamentação Teórica

A criptografia moderna organiza-se em torno de três famílias de primitivas. As **funções hash criptográficas** mapeiam dados de tamanho arbitrário para resumos de comprimento fixo e devem oferecer resistência à pré-imagem, à segunda pré-imagem e a colisões [Menezes et al. 2018]; algoritmos das famílias SHA-2 e SHA-3 atendem a esses critérios, enquanto MD5 e SHA-1 possuem ataques práticos de colisão e não são mais recomendados [Stallings 2017, Dang and Zhao 2015]. Construções com chave, como o HMAC (*Hash-based Message Authentication Code*), estendem o *hash* para autenticação de mensagens entre partes que compartilham um segredo [Krawczyk et al. 1997].

A **criptografia simétrica** emprega uma única chave para cifrar e decifrar, sendo eficiente para grandes volumes de dados [Stinson 2006]. O *Advanced Encryption Standard* (AES), padronizado como FIPS 197, é a cifra de bloco de referência, aplicada a mensagens de tamanho arbitrário por meio de modos de operação [National Institute of Standards and Technology 2001]. Destacam-se o CBC (*Cipher Block Chaining*), amplamente usado em envelopes CMS, e o GCM (*Galois/Counter Mode*), que provê confidencialidade e autenticidade em uma única operação, propriedade conhecida como AEAD (*Authenticated Encryption with Associated Data*) e recomendada para novas aplicações [Menezes et al. 2018, National Institute of Standards and Technology 2001].

A **criptografia assimétrica** baseia-se em pares de chaves pública e privada relacionadas por funções unidirecionais [Menezes et al. 1996]. Ela viabiliza a cifração sem segredo prévio, protocolos de acordo de chaves [Diffie and Hellman 1976] e assinaturas digitais, que garantem integridade, autenticidade e não repúdio por meio da tripla (KeyGen, Sign, Verify) aplicada, em geral, ao resumo criptográfico da mensagem [Stallings 2017]. Os esquemas predominantes são o RSA, fundamentado na dificuldade da fatoração de inteiros [Rivest et al. 1978, Moriarty et al. 2016], e o ECDSA (*Elliptic Curve Digital Signature Algorithm*), baseado no problema do logaritmo discreto em curvas elípticas, que atinge segurança equivalente à do RSA-3072 com chaves de apenas 256 bits [Miller 1985]. Na prática, protocolos como o TLS (*Transport Layer Security*) combinam as duas famílias em esquemas híbridos: a criptografia assimétrica estabelece a chave de sessão e a simétrica protege o tráfego [Rescorla 2018].

Como qualquer parte pode publicar uma chave pública, é preciso vinculá-la de forma verificável a uma identidade. Essa função é cumprida pela **Infraestrutura de Chaves Públicas** (ICP), conjunto de tecnologias, políticas e procedimentos para emissão, distribuição, validação e revogação de **certificados digitais** no padrão X.509 [Adams and Lloyd 2002, International Telecommunication Union 2016]. Autoridades Certificadoras (ACs) raiz e intermediárias formam cadeias de confiança; a validação de um certificado verifica a assinatura de cada elo, o período de validade, as extensões (como *Key Usage* e *Basic Constraints*) e o estado de revogação, consultado via Listas de Certificados Revogados (LCR) ou pelo protocolo OCSP (*Online Certificate Status Protocol*) [Cooper et al. 2008, Santesson et al. 2013].

A interoperabilidade entre sistemas depende de formatos padronizados. O CMS (*Cryptographic Message Syntax*), derivado do PKCS#7, encapsula dados assinados ou cifrados com metadados, certificados e múltiplos signatários [Housley 2009]; o perfil CAdES (*CMS Advanced Electronic Signatures*) o estende

com carimbos de tempo e atributos exigidos para validade jurídica de longo prazo [European Telecommunications Standards Institute 2013]. Requisições de certificação seguem o PKCS#10, e a base sintática comum a esses formatos é a ASN.1 (*Abstract Syntax Notation One*) com codificação DER [International Telecommunication Union 2016].

A segurança dos esquemas assimétricos clássicos é ameaçada pela computação quântica: o algoritmo de Shor resolve em tempo polinomial a fatoração e o logaritmo discreto, comprometendo RSA e ECDSA [Shor 1994]. Isso motivou o processo de padronização pós-quântica do NIST (*National Institute of Standards and Technology*) [National Institute of Standards and Technology 2020], concluído em 2024 com a publicação do FIPS 203 (ML-KEM, mecanismo de encapsulamento de chaves derivado do CRYSTALS-Kyber) e do FIPS 204 (ML-DSA, assinatura digital derivada do CRYSTALS-Dilithium) [National Institute of Standards and Technology 2024b, National Institute of Standards and Technology 2024a, Ducas et al. 2018]. Ambos fundamentam-se no problema *Module Learning With Errors* sobre reticulados, para o qual não se conhecem algoritmos eficientes clássicos ou quânticos [Regev 2009]. A contrapartida é o tamanho dos artefatos: a chave pública do ML-KEM-768 ocupa 1.184 bytes e a assinatura do ML-DSA-44, 2.420 bytes, dezenas de vezes maiores que as equivalentes de curvas elípticas, com impacto direto em certificados e *handshakes* TLS [National Institute of Standards and Technology 2024b, National Institute of Standards and Technology 2024a, Chen and et al. 2024].

Por fim, dois mecanismos sustentam a implementação deste trabalho. A *Foreign Function Interface* (FFI) do Dart, provida pelo pacote `dart:ffi`, permite carregar bibliotecas nativas em C e invocar suas funções no mesmo processo, sem serialização, transferindo à aplicação a responsabilidade pelo mapeamento de tipos e pelo ciclo de vida da memória nativa [Dart Team 2024c]. Do lado nativo, o OpenSSL expõe suas operações pela **API EVP** (*Envelope*), interface uniforme baseada em contextos genéricos que abstrai o algoritmo subjacente e, por meio da arquitetura de *providers*, permite carregar dinamicamente implementações adicionais, incluindo os algoritmos pós-quânticos e o provedor FIPS [OpenSSL Project Authors 2026a, OpenSSL Project Authors 2026b].

3. Metodologia

Quanto à natureza, esta pesquisa classifica-se como aplicada, por envolver o desenvolvimento de um artefato de software destinado à integração entre Dart e OpenSSL. Quanto aos objetivos, é exploratória e descritiva: exploratória na investigação do uso de `dart:ffi` com uma biblioteca criptográfica nativa, e descritiva no registro das decisões de projeto e das condições de avaliação. A abordagem é quali-quantitativa, combinando a análise qualitativa das bibliotecas e decisões arquiteturais com a avaliação quantitativa de testes, cobertura de código e desempenho. Os procedimentos técnicos articulam pesquisa bibliográfica, estudo de caso e pesquisa experimental.

A execução organizou-se em quatro etapas sequenciais: (i) revisão bibliográfica e análise do estado da arte, concentrada em Infraestrutura de Chaves Públicas (ICP), formatos de assinatura digital, criptografia pós-quântica, *Foreign Function Interface* (FFI) e OpenSSL; (ii) análise comparativa entre bibliotecas do ecossistema Dart (Pointycastle, Crypto, Cryptography) e soluções consolidadas em outras plataformas, como Bouncy Castle e OpenSSL, considerando operações de ICP, suporte multiplata-

forma, integração nativa, manutenção e documentação [PointyCastle Contributors 2024, Bouncy Castle 2024]; (iii) desenvolvimento iterativo da biblioteca, em ciclos de implementação, teste e revisão, tratando separadamente problemas de *bindings* FFI, gerenciamento de memória e organização da API; e (iv) testes e validação, com testes unitários e de integração executados em Linux x86_64 e em dispositivo Android ARM64, complementados por modelagem de ameaças pela metodologia STRIDE sobre a superfície de ataque entre a API Dart, a fronteira FFI e os objetos nativos do OpenSSL.

O ambiente de desenvolvimento utilizou Ubuntu 22.04 (via WSL sobre Windows 11), Dart 3.x, Flutter 3.24+, OpenSSL 4.0.0 como *backend* criptográfico [OpenSSL Project Authors 2026a], GCC 13.x, Android NDK e as ferramentas `flutter_test`, `lcov` e `glados` (testes baseados em propriedades).

A avaliação adotou três categorias de métricas: correção funcional, exigindo aprovação integral da suíte de testes nas plataformas validadas; cobertura de código, medida com `lcov` e critério de aceitação mínimo de 60%; e desempenho computacional, medido em Linux x86_64 com aquecimento prévio e repetições por operação, reportando média, *throughput* e, quando disponível, desvio-padrão, além de comparação Linux/Android.

4. Desenvolvimento

4.1. Análise das Bibliotecas Existentes

O projeto da `dartcryptosec` partiu de uma análise comparativa entre duas bibliotecas de referência, Bouncy Castle e OpenSSL, maduras, auditadas e ubíquas em sistemas de produção [Bouncy Castle 2024, OpenSSL Project Authors 2026a], e as três principais bibliotecas criptográficas do ecossistema Dart: PointyCastle, Cryptography e Crypto [PointyCastle Contributors 2024, Dart Cryptography Team 2024, Dart Team 2024a]. A avaliação, baseada em documentação oficial, código-fonte público e literatura técnica, priorizou as funcionalidades essenciais para Infraestrutura de Chaves Públicas (ICP): manipulação de certificados X.509, verificação de revogação por Lista de Certificados Revogados (LCR) e OCSP, formatos de assinatura CMS/CAdES e geração de chaves. A Tabela 1 sintetiza os resultados.

Tabela 1. Comparação de funcionalidades entre bibliotecas de referência e bibliotecas Dart (S = presente, P = parcial, N = ausente).

Funcionalidade	B. Castle	OpenSSL	PointyC.	Cryptogr.	Crypto
Cifras simétricas/assimétricas	S	S	S	S	N
Hash e MACs	S	S	S	S	S
Assinaturas digitais e KDFs	S	S	S	S	N
ASN.1 e certificados X.509	S	S	P	N	N
Validação de cadeias e revogação	S	S	N	N	N
CMS/CAdES e CSRs	S	S	N	N	N
Pós-quânticos (ML-KEM/ML-DSA)	S	S	N	N	N

A análise evidencia que as bibliotecas Dart cobrem adequadamente as primitivas fundamentais, mas apresentam lacunas críticas justamente nas capacidades de ICP: nenhuma implementa verificação de revogação, CMS/CAdES

ou validação completa de cadeias, e mesmo a Pointycastle, a mais abrangente, oferece apenas suporte parcial a ASN.1 e X.509. Nenhuma delas suporta os algoritmos pós-quânticos padronizados pelo NIST, já incorporados às bibliotecas de referência [National Institute of Standards and Technology 2024b, National Institute of Standards and Technology 2024a]. Essa lacuna motivou a estratégia adotada: em vez de reimplementar algoritmos em Dart, integrar o OpenSSL via *Foreign Function Interface* (FFI), herdando sua maturidade, auditabilidade e cobertura algorítmica.

4.2. A Ponte FFI entre Dart e C

A integração apoia-se no subsistema `dart:ffi` [Dart Team 2024c]: a biblioteca nativa (`.so`, `.dylib` ou `.dll`) é carregada em tempo de execução via `DynamicLibrary`, e cada função exposta segue um padrão triplo de declaração composto por um *typedef* com a assinatura C original, um *typedef* com a assinatura Dart correspondente e um *binding* `late final` resolvido por `lookup()` no primeiro acesso. O arquivo de integração reúne 189 *bindings* escritos manualmente, opção preferida à geração automática pelo `ffigen` porque a API pública utiliza um subconjunto pequeno e controlado do OpenSSL, e a escrita manual permite auditar cada assinatura junto às regras de propriedade de memória. Os *bindings* restringem-se à API EVP e às interfaces de alto nível (X.509, CMS, OCSP), evitando funções legadas [OpenSSL Project Authors 2026b]; a maior parcela atende a certificados, LCR, OCSP e requisições PKCS#10, coerente com o foco em ICP.

Estruturas internas do OpenSSL, como `EVP_PKEY`, `X509` e `CMS_ContentInfo`, são tratadas como ponteiros opacos (`Pointer<Void>`), transportados entre chamadas sem interpretação de conteúdo, o que preserva o encapsulamento idiomático da biblioteca nativa. Como o coletor de lixo do Dart não gerencia recursos nativos, alocação e liberação são mantidas próximas no código, com blocos `try/finally` garantindo a limpeza mesmo em caso de falha. O carregamento multiplataforma (Linux e Android) é isolado em um módulo dedicado, de modo que as demais camadas independem do ambiente de execução.

4.3. Arquitetura em Camadas

A `dartcryptosec` organiza-se em sete camadas com direção de dependência bem definida, ilustradas na Figura 1: (i) **OpenSSL**, biblioteca nativa que provê as primitivas; (ii) **FFI**, os *bindings* manuais; (iii) **Operations**, classes que encapsulam as chamadas nativas por domínio criptográfico e gerenciam o ciclo de vida dos recursos; (iv) **Models**, tipos de domínio como `CryptoResult<T>` (classe selada de sucesso ou falha tipada), `KeyPair` e `CertificateData`; (v) **Flows**, casos de uso de maior nível, como assinatura de arquivos e validação de cadeias; (vi) **Facade**, a classe `PluginCryptoAPI` (nome herdado da prova de conceito *PluginCrypto*), que expõe uma API unificada com 36 métodos públicos; e (vii) a aplicação de demonstração usada nos testes de integração.

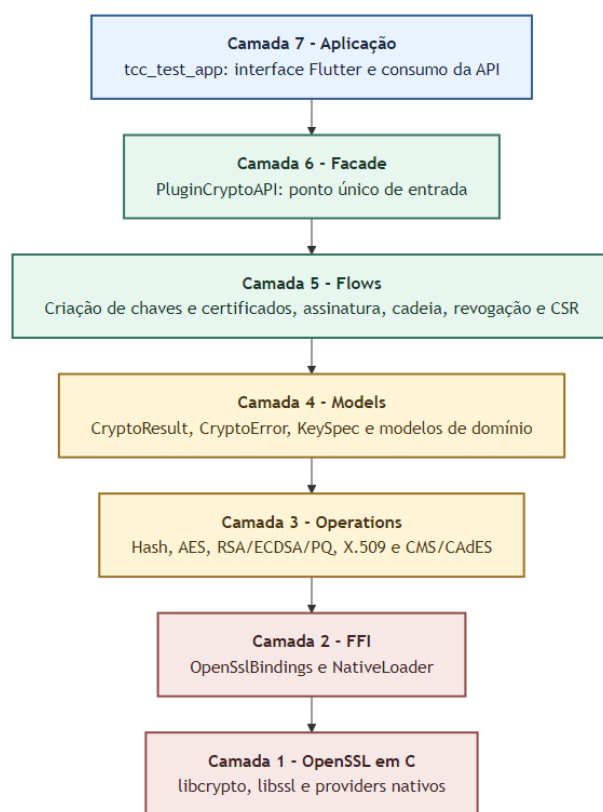


Figura 1. Fluxo principal da arquitetura em sete camadas da `dartcryptosec`.

O projeto emprega padrões clássicos com motivação concreta: *Facade* para reduzir a curva de aprendizado, *Factory* para instanciar objetos criptográficos a partir de dados ASN.1 ambíguos, *Strategy* para intercambiar algoritmos de assinatura e cifração em tempo de execução, *Singleton* para a inicialização única do OpenSSL e *Builder* para a criação programática de certificados X.509. Erros nativos são convertidos em uma hierarquia selada de `CryptoError`, cujos subtipos preservam a mensagem e o código originais do OpenSSL e permitem verificação exhaustiva pelo compilador. Complementarmente, uma análise de ameaças pela metodologia STRIDE catalogou quinze ameaças priorizadas por risco, orientando decisões como a validação de ponteiros nulos, a limpeza sistemática da fila de erros do OpenSSL e testes adversariais com entradas malformadas.

4.4. Integração dos Algoritmos Pós-Quânticos

O suporte a ML-KEM e ML-DSA aproveitou as interfaces genéricas da API EVP e exigiu alterações localizadas: novos *bindings* de encapsulamento e desencapsulamento para o ML-KEM (o ML-DSA reutiliza as funções genéricas de assinatura), novas operações e especificações de chave tipadas para selecionar as variantes padronizadas [National Institute of Standards and Technology 2024b, National Institute of Standards and Technology 2024a]. As camadas superiores mantiveram os mesmos fluxos de geração de chaves e assinatura. Duas particularidades exigiram tratamento: as operações do ML-KEM seguem o padrão nativo de duas chamadas (a primeira consulta os tamanhos dos *buffers*, a segunda os preenche), evitando supor tamanhos fixos na fronteira FFI; e o ML-DSA, diferentemente de RSA e ECDSA, realiza interna-

mente o processamento definido pelo FIPS 204, de modo que a inicialização da assinatura não recebe um *digest* separado [National Institute of Standards and Technology 2024a].

4.5. Desafios Técnicos

Além do gerenciamento de memória nativa, destacam-se quatro desafios. No AES-GCM, a *tag* de autenticação de 128 bits precisa ser configurada no contexto nativo antes da finalização da decifração, e a operação só é considerada válida quando o OpenSSL confirma a autenticação, impedindo que falhas de integridade sejam confundidas com decifrações bem-sucedidas. O *parsing* de `ASN1_TIME` delega ao OpenSSL a conversão textual das datas de validade dos certificados, preservando o uso de ponteiros opacos [Cooper et al. 2008]. Operações CMS que recebem pilhas de certificados exigem regras explícitas de propriedade de objetos para evitar vazamentos e liberações duplicadas. Por fim, os identificadores pós-quânticos só são resolvidos com o provedor FIPS compilado e carregado; no Android, esses artefatos precisaram ser empacotados com a aplicação e localizados em tempo de execução, com o carregamento do provedor separado das operações criptográficas para permitir a detecção antecipada de falhas. Como limitações, a camada FFI permanece acoplada à versão do OpenSSL utilizada, e a validação funcional restringiu-se a Linux e Android.

5. Resultados e Validação

A validação da `dartcryptosec` abrangeu quatro dimensões: corretude funcional (suíte de testes), cobertura de código, desempenho (*benchmarks* de latência e vazão) e segurança da implementação, complementadas pela comparação com o estado da arte no ecossistema Dart.

5.1. Suíte de Testes e Cobertura

A suíte de testes totaliza 692 testes, sendo 515 testes unitários executados em Linux x86_64 (Ubuntu 22.04, OpenSSL 4.0.0 com `enable-fips`) e 177 testes de integração executados em dispositivo Android físico (arm64-v8a, Android 15), todos aprovados sem nenhuma falha. A suíte Linux, organizada em 40 zonas funcionais, cobre desde o carregador nativo da Foreign Function Interface (FFI), as primitivas de *hash* das famílias SHA-2 e SHA-3 e a cifração AES nos modos CBC e GCM até validação de cadeias X.509, Lista de Certificados Revogados (LCR), Online Certificate Status Protocol (OCSP), Cryptographic Message Syntax (CMS)/CAvES, *fuzzing*, baterias estatísticas NIST SP800-22 e SP800-90B e testes de *timing*. Os 177 testes Android exercitam fluxos completos *end-to-end*, incluindo 31 cenários adversariais com entradas malformadas e 24 testes dedicados aos algoritmos pós-quânticos (22 de *pipeline* e 2 de conformidade FIPS), confirmando paridade funcional entre as plataformas em todos os domínios criptográficos.

A estabilidade sob carga foi verificada por testes de longa duração (*soak*): quatro operações executadas por 5 minutos ininterruptos cada, acumulando cerca de 85,5 milhões de *hashes* SHA-256, 54 milhões de *round-trips* AES-256-GCM e 390 mil assinaturas RSA-2048, com crescimento de memória residente inferior a 10 MB por operação e nenhum vazamento detectado. Testes de concorrência com 1, 2, 4 e 8 *isolates* Dart e testes combinatórios com 63 combinações de parâmetros também foram integralmente aprovados.

A cobertura de código medida com `lcov` em Linux é de 68,4% (2.871 de 4.195 linhas em 60 arquivos). A distribuição por camada reflete a arquitetura: a camada FFI atinge 96,8% (os 3,2% restantes são caminhos de erro de falhas catastróficas do OpenSSL, de difícil reprodução), enquanto a camada de orquestração (*Flows*) fica em 58,3%, concentrando as lacunas em caminhos alternativos e ramos de erro. A métrica não contabiliza os 177 testes Android, pois o Flutter não gera `lcov.info` para execuções em dispositivo remoto; projeta-se cobertura efetiva de ao menos 75% considerando ambos os ambientes.

5.2. Desempenho

Os *benchmarks* foram coletados em Linux x86_64 (Intel Core i7-13620H) com aquecimento prévio e cronometragem em resolução de microssegundos. A Tabela 2 resume as latências das operações mais representativas.

Tabela 2. Latência média das principais operações (Linux x86_64, OpenSSL 4.0.0).

Categoria	Operação	Tamanho	Latência
Hash	SHA-256	1 KB	2,8 μ s
Hash	SHA-256	1 MB	0,54 ms
Cifra	AES-256-GCM <i>encrypt</i>	1 KB	5,7 μ s
Geração de chaves	RSA-2048	–	31,8 ms
Geração de chaves	ECDSA P-256	–	0,18 ms
Assinatura	RSA-2048 (SHA-256)	32 B	0,77 ms
Assinatura	ECDSA P-256	32 B	0,06 ms
Pós-quântico	ML-KEM-768 <i>keygen</i>	–	0,07 ms
Pós-quântico	ML-DSA-44 <i>sign</i>	32 B	0,42 ms
Pós-quântico	ML-DSA-44 <i>verify</i>	32 B	0,10 ms

Em termos de vazão, o SHA-256 atinge 1.857,5 MB/s (1 MB) e o AES-256-GCM, 2.387,9 MB/s (64 KB), beneficiado pela aceleração AES-NI; em operações por segundo, o SHA-256 processa cerca de 357.000 ops/s e o AES-256-GCM cerca de 180.000 ops/s para *payloads* de 1 KB. Entre as *cipher suites* simétricas, o AES-256-GCM lidera a cifração de 1 MB com 718,4 MB/s, superando o modo CBC em cerca de 40% graças ao paralelismo do modo CTR subjacente e ao GHASH via `PCLMULQDQ` [Gupta and Prakash 2024]. Destaca-se o desempenho pós-quântico: a geração de chaves ML-KEM-768 (0,073 ms) é ordens de grandeza mais rápida que a do RSA-2048 (31,8 ms), e a assinatura ML-DSA-44 ($\sim$ 2.400 ops/s) supera a assinatura RSA-2048 ($\sim$ 1.300 ops/s), evidenciando que a transição pós-quântica não implica penalidade de desempenho.

O custo da fronteira FFI é fixo por chamada (1–2 μ s): domina apenas primitivas simétricas de latência muito baixa e representa menos de 0,01% do tempo de operações assimétricas, sendo duas a três ordens de grandeza inferior à invocação do `openssl` por linha de comando. A Tabela 3 compara as plataformas: o Android é mais lento em todas as operações, com fatores maiores nas primitivas aceleradas por instruções dedicadas do x86_64 (SHA-NI, AES-NI) e menores nas operações pós-quânticas, que não se beneficiam dessas extensões.

Tabela 3. Comparação de latência Linux (x86_64) vs. Android (arm64-v8a).

Operação	Linux	Android	Fator
SHA-256 (1 KB)	2,8 μ s	36,2 μ s	12,9×
AES-256-GCM (1 KB)	5,7 μ s	16,0 μ s	2,8×
RSA-2048 <i>keygen</i>	31,8 ms	223,4 ms	7,0×
ECDSA P-256 <i>sign</i>	0,062 ms	1,0 ms	16,2×
ML-KEM-768 <i>keygen</i>	0,073 ms	0,154 ms	2,1×
ML-DSA-44 <i>sign</i>	0,418 ms	0,957 ms	2,3×

Uma simulação de sessão TLS 1.3 (TLS_AES_128_GCM_SHA256) registrou *handshake* completo em 0,848 ms — com 98,5% do custo nas operações assimétricas (ECDHE e verificação de certificado ECDSA P-256) — e vazão de cifração de 922,8 MB/s na transferência em massa, demonstrando a viabilidade da biblioteca como *backend* criptográfico para servidores HTTPS.

5.3. Segurança da Implementação

Foram executadas 63 verificações automatizadas de segurança, todas aprovadas. O gerador de números aleatórios apresentou entropia de Shannon de 7,981 bits/byte (limiar 7,950) e p-valor de 0,511 no teste qui-quadrado; a bateria NIST SP800-22 registrou 15 de 16 testes consistentemente aprovados, com uma ocorrência isolada no *Serial Test* classificada como flutuação estatística esperada. A unicidade de material criptográfico foi de 100% em amostras de 100 pares RSA, 100 pares EC e 10.000 vetores de inicialização AES-GCM. As três curvas elípticas NIST utilizadas (P-256, P-384 e P-521) passaram em todos os critérios do *checklist* SafeCurves, e a rejeição de *tags* de autenticação GCM sob chave incorreta foi confirmada. Quatro limitações conhecidas permanecem documentadas, uma de severidade média (exceção não capturada no *parsing* CMS de entrada malformada) e três baixas, associadas sobretudo à variabilidade de medição no ambiente WSL1.

5.4. Comparação com o Estado da Arte

Frente às alternativas *pure Dart pointycastle* e *cryptography* [PointyCastle Contributors 2024, Dart Cryptography Team 2024], a *dartcryptosec* é a única a oferecer validação de cadeias X.509, verificação de revogação por LCR e OCSP, CSRs PKCS#10, CMS/CAvES, conformidade FIPS 140-3 (herdada do OpenSSL) e algoritmos pós-quânticos ML-KEM e ML-DSA [National Institute of Standards and Technology 2024b, National Institute of Standards and Technology 2024a]. As alternativas conservam como vantagens a ausência de dependências nativas e a instalação sem etapa de compilação, ao custo de menor desempenho e de lacunas funcionais em Infraestrutura de Chaves Públicas (ICP). Para aplicações com requisitos regulatórios, alto *throughput* ou criptografia pós-quântica, a *dartcryptosec* constitui atualmente a opção mais completa do ecossistema Dart/Flutter.

6. Conclusão

Este trabalho apresentou a `dartcryptosec`, biblioteca criptográfica para Dart que integra o OpenSSL 4.0.0 por meio de Foreign Function Interface (FFI), motivada por lacunas identificadas no projeto *Medida Inteligente* (LabSEC/UFSC em parceria com o Inmetro) [Idalino et al. 2022]. O objetivo geral — prover suporte abrangente a operações de Infraestrutura de Chaves Públicas (ICP) em Dart — foi alcançado: todos os objetivos específicos foram cumpridos, incluindo a análise do estado da arte [PointyCastle Contributors 2024, Dart Cryptography Team 2024, Dart Team 2024a], a arquitetura modular em sete camadas, a implementação de validação de cadeias X.509 [Cooper et al. 2008], verificação de revogação por Lista de Certificados Revogados (LCR) e Online Certificate Status Protocol (OCSP) [Santesson et al. 2013], criação de CSRs PKCS#10 [Nystrom and Kaliski 2000], operações CMS/CAdES, a integração nativa via 189 *bindings* FFI e a produção de documentação abrangente, com referência da API, descrição arquitetural e exemplos de uso. A incorporação dos algoritmos pós-quânticos ML-KEM e ML-DSA [National Institute of Standards and Technology 2024b, National Institute of Standards and Technology 2024a], inicialmente prevista como expansão, foi concluída durante o desenvolvimento sem alterar a interface exposta aos casos de uso.

A validação reuniu 692 testes unitários e de integração executados sem falhas em Linux x86_64 e Android arm64-v8a, com cobertura de código de 68,4% em Linux. Os resultados demonstram que a integração direta com o OpenSSL evita a reimplementação de primitivas criptográficas e entrega desempenho próximo ao nativo, com *overhead* de FFI inferior a 0,01% nas operações assimétricas. Como contribuições, o trabalho oferece à comunidade Dart um conjunto de APIs de ICP e criptografia pós-quântica até então indisponível, disponibilizado como código aberto [Martins 2026], além de constituir ativo tecnológico reutilizável pelo LabSEC em projetos com dispositivos móveis e comunicação segura.

Entre as limitações, destacam-se a ausência de validação empírica em iOS, macOS e Windows, a cobertura parcial da camada de orquestração (*Flows*, 58,3%) e as dificuldades inerentes à integração Dart-C: mapeamento manual de tipos, gerenciamento de memória nativa fora do alcance do coletor de lixo do Dart e a migração forçada para a API EVP na transição do OpenSSL para a série 4.0 [OpenSSL Project Authors 2026b]. Como trabalhos futuros, propõem-se a expansão da validação multiplataforma, a ampliação da cobertura de testes nos fluxos secundários, o suporte a HMAC, a avaliação da geração assistida de *bindings* com `ffigen` [Dart Team 2024c] e a publicação do pacote no `pub.dev`, preparando a biblioteca para adoção fora do ambiente em que foi validada.

Referências

- Adams, C. and Lloyd, S. (2002). *Understanding PKI: Concepts, Standards, and Deployment Considerations*. Addison-Wesley Professional, Boston, MA, second edition.
- Aranha, D. d. F. (2007). Serviço de nomes e roteamento para redes de anonimização de tráfego. Tese de mestrado, Instituto de Computação, UNICAMP. Financiada pelo CNPq, processo número 131820/2005-2.
- Bouncy Castle (2024). *Bouncy Castle Cryptography APIs*. Bouncy Castle, Cambridge.

- Chen, L. and et al. (2024). *Report on Post-Quantum Cryptography*. National Institute of Standards and Technology, Gaithersburg.
- Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. (2008). RFC 5280: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, Internet Engineering Task Force.
- Dang, T. A. and Zhao, B. (2015). Sha-3: The new sha standard.
- Dart Cryptography Team (2024). *Dart Cryptography Package*. Dart Team, Cambridge.
- Dart Team (2024a). *Dart Packages: crypto*. Dart Team, Cambridge.
- Dart Team (2024b). *Dart Programming Language*. Dart Team, Cambridge.
- Dart Team (2024c). *Using FFI in Dart*. Dart Team, Cambridge.
- Diffie, W. and Hellman, M. E. (1976). New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654.
- Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., and Stehlé, D. (2018). CRYSTALS-Dilithium: A lattice-based digital signature scheme. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(1):238–268.
- European Telecommunications Standards Institute (2013). Etsi ts 101 733 v2.2.1: Electronic signatures and infrastructures (esi); cms advanced electronic signatures (cades). Technical Specification ETSI TS 101 733 V2.2.1, European Telecommunications Standards Institute, Sophia Antipolis.
- Google LLC (2024). *Flutter Documentation*. Google LLC, Cambridge.
- Gupta, S. and Prakash, S. (2024). Performance analysis of cryptographic algorithms in mobile applications. *International Journal of Computer Applications*, 184(5):1–6.
- Housley, R. (2009). Rfc 5652: Cryptographic message syntax (cms). Request for Comments 5652, RFC Editor.
- Idalino, T. B., Vigil, M., Oliveira, G. E. d., Abel, P. d. O., Na, E. G., Gomes, C. E. V., Hardt, V. H., Tomaszewsk, M., and Loli, G. B. (2022). Estudo de usabilidade de assinaturas digitais em objetos metrológicos. Relatório Técnico INE 004/2022, Universidade Federal de Santa Catarina, Departamento de Informática e Estatística, Laboratório de Segurança em Computação, Florianópolis. Relatório público do projeto Medida Inteligente, desenvolvido em parceria com o Inmetro.
- International Telecommunication Union (2016). *X.509 Information Framework*. International Telecommunication Union, Geneva.
- Krawczyk, H., Bellare, M., and Canetti, R. (1997). Rfc 2104: Hmac: Keyed-hashing for message authentication. Request for Comments 2104, RFC Editor.
- Martins, I. (2026). dartcryptosec: Biblioteca criptográfica para dart. Repositório institucional de código-fonte da UFSC.
- Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. (1996). *Handbook of Applied Cryptography*. CRC Press, 1st edition. Available at <http://cacr.uwaterloo.ca/hac/>.

- Menezes, A. J., van Oorschot, P. C., and Vanstone, S. A. (2018). *Handbook of Applied Cryptography*. CRC Press.
- Miller, V. S. (1985). Use of elliptic curves in cryptography. In *Advances in Cryptology — CRYPTO’85*, pages 417–426. Springer.
- Moriarty, K., Kaliski, B., Jonsson, J., and Rusch, A. (2016). Rfc 8017: Pkcs #1: Rsa cryptography specifications version 2.2. Request for Comments 8017, RFC Editor.
- National Institute of Standards and Technology (2001). *FIPS 197: Advanced Encryption Standard (AES)*. NIST, Gaithersburg.
- National Institute of Standards and Technology (2020). Post-quantum cryptography. <https://csrc.nist.gov/Projects/post-quantum-cryptography>. Accessed: 2024-11-07.
- National Institute of Standards and Technology (2024a). Module-Lattice-based Digital Signature Standard. Technical Report FIPS 204, NIST.
- National Institute of Standards and Technology (2024b). Module-Lattice-based Key-Encapsulation Mechanism Standard. Technical Report FIPS 203, NIST.
- Nystrom, M. and Kaliski, B. (2000). Rfc 2986: Pkcs #10: Certification request syntax specification version 1.7. Request for Comments 2986, RFC Editor.
- OpenSSL Project Authors (2026a). *OpenSSL Documentation*. OpenSSL Project.
- OpenSSL Project Authors (2026b). *OpenSSL Guide: Migrating from Older OpenSSL Versions*. OpenSSL Project.
- OWASP Foundation (2024). *Cryptography Best Practices*. OWASP Foundation, Salt Lake City.
- Pointycastle Contributors (2024). *Pointycastle: A Dart Library for Cryptography*. Pointycastle, Cambridge.
- Regev, O. (2009). On lattice-based cryptography. In *Post-Quantum Cryptography*, pages 1–20. Springer.
- Rescorla, E. (2018). Rfc 8446: The transport layer security (tls) protocol version 1.3. Request for Comments 8446, RFC Editor.
- Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.
- Santesson, S., Myers, M., Ankney, R., Malpani, A., Galperin, S., and Adams, C. (2013). Rfc 6960: X.509 internet public key infrastructure online certificate status protocol - ocsp. Request for Comments 6960, RFC Editor.
- Shor, P. W. (1994). Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134. IEEE.
- Stallings, W. (2017). *Cryptography and Network Security: Principles and Practice*. Pearson, 7 edition.
- Stinson, D. R. (2006). *Cryptography: Theory and Practice*. CRC Press, Boca Raton, 3 edition.