Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Assinador de eventos nostr com hardware criptográfico padrão

Victor André de Moraes

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Orientador
Prof. Dr. JOSE EDIL GUIMARAES DE MEDEIROS

Brasília
2025

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Assinador de eventos nostr com hardware criptográfico padrão

Victor André de Moraes

Monografia apresentada como requisito parcial
para conclusão do Curso de Engenharia da Computação

Prof. Dr. JOSE EDIL GUIMARAES DE MEDEIROS (Orientador)
ENE/UnB

Prof. Dr. DANIEL CHAVES CAFÉ
ENE/UnB

Prof. Dr. JOÃO JOSÉ COSTA GONDIM
ENE/UnB

Prof. Dr. GUILHERME NOVARES RAMOS
Coordenador do Curso de Engenharia da Computação

Brasília, 15 de dezembro de 2025

Resumo

O protocolo Nostr introduz um paradigma de redes sociais descentralizadas onde a identidade digital é autocustodiada e definida por pares de chaves criptográficas. Embora este modelo elimine a dependência de autoridades centrais, ele transfere para o usuário a responsabilidade crítica pela segurança da sua chave privada (`nsec`). A prática comum de armazenar estas chaves em ambientes de software — como navegadores e arquivos locais — expõe os usuários a vetores de ataque severos, incluindo *malware* de varredura e extração de dados.

Este trabalho propõe e implementa uma arquitetura de segurança baseada em isolamento via hardware para mitigar estes riscos. A solução desenvolvida consiste num serviço de assinatura remota (*Nostr Bunker*, NIP-46) que utiliza *tokens* FIDO2 para a custódia das chaves. Superando a incompatibilidade nativa de curvas elípticas entre o padrão FIDO2 e o Nostr, a implementação utiliza as extensões CTAP2 `hmac-secret` e `largeBlob` para criar um mecanismo de "envelope criptográfico", onde a chave privada reside cifrada no hardware e é decifrada em memória volátil apenas no instante da assinatura.

A validação experimental, realizada na rede pública Nostr, confirmou a eficácia do protótipo em assinar e publicar eventos de forma válida, assegurando que a chave privada não permaneça exposta em repouso no disco. Os resultados indicam que é possível elevar significativamente a segurança do ecossistema Nostr sem sacrificar a usabilidade ou a compatibilidade com clientes existentes.

Palavras-chave: Nostr, FIDO2, YubiKey, Segurança de Hardware, Criptografia Aplicada, Identidade Descentralizada.

Abstract

The Nostr protocol introduces a paradigm of decentralized social networks where digital identity is self-custodied and defined by cryptographic key pairs. Although this model eliminates reliance on central authorities, it transfers the critical responsibility for private key (`nsec`) security to the user. The common practice of storing these keys in software environments—such as browsers and local files—exposes users to severe attack vectors, including scanning malware and data extraction.

This work proposes and implements a security architecture based on hardware-based isolation to mitigate these risks. The developed solution consists of a remote signing service (*Nostr Bunker*, NIP-46) that utilizes FIDO2 hardware tokens for key custody. Overcoming the native elliptic curve incompatibility between the FIDO2 standard and Nostr, the implementation leverages the CTAP2 `hmac-secret` and `largeBlob` extensions to create a "cryptographic envelope" mechanism, where the private key resides encrypted within the hardware and is decrypted in volatile memory only at the instant of signing.

Experimental validation, performed on the public Nostr network, confirmed the prototype's efficacy in validly signing and publishing events, ensuring that the private key does not remain exposed at rest on disk. The results indicate that it is possible to significantly elevate the security of the Nostr ecosystem without sacrificing usability or compatibility with existing clients.

Keywords: Nostr, FIDO2, YubiKey, Hardware Security, Applied Cryptography, Decentralized Identity.

Sumário

1 Introdução **1**

1.1 Contexto e Problemática: O Valor da Identidade e o Risco em Repouso . . 1

1.2 A Abordagem Proposta: Isolamento via Hardware 2

1.3 Objetivos 3

1.3.1 Objetivos Específicos 3

1.3.2 Delimitação do Escopo 3

2 Da Identidade Descentralizada à Segurança de Chaves em Hardware **5**

2.1 Arquiteturas de Rede e o Paradigma Nostr 5

2.1.1 O Modelo "Cliente Inteligente, Servidor Burro" 6

2.2 Anatomia do Protocolo: Eventos e Identidade 6

2.2.1 A Identidade Autocustodiada: Chaves e Assinaturas 7

2.3 Ecossistema de Aplicações 9

2.4 O Desafio da Custódia e o Nostr Bunker 9

2.4.1 Mitigação via NIP-46: Nostr Bunker 9

2.4.2 A Vulnerabilidade Residual 11

2.5 Segurança Baseada em Hardware 12

2.5.1 O Princípio do Isolamento Físico 12

2.5.2 O Padrão FIDO2 e o Protocolo de Transporte CTAP 12

2.5.3 Ideal vs. Realidade: O Problema das Curvas 12

2.5.4 A Solução: Envelope Criptográfico via Extensões FIDO2 13

3 Arquitetura e Validação Experimental da Solução Proposta **14**

3.1 Configuração do Ambiente de Validação 14

3.2 Operação do Protótipo: Do Registro à Assinatura 15

3.2.1 O Processo de Registro Seguro (Offline) 15

3.2.2 O Processo de Assinatura de Eventos (Online) 15

3.3 Validação e Critérios de Sucesso 18

3.4 Análise Crítica e Limitações do Protótipo 19

4 Implementação do Bunker com Suporte a Hardware FIDO2 **20**
4.1 Arquitetura do Software e Tecnologias Adotadas 20
4.2 Módulo de Gerenciamento de Identidade (Modo Offline) 21
4.2.1 Provisionamento de Chave (Store key) 21
4.2.2 Leitura e Exclusão de Chaves (Read/Delete key) 21
4.3 Módulo de Interface com o Hardware FIDO2 22
4.4 Módulo Criptográfico e de Armazenamento Seguro 22
4.4.1 Processo de Cifragem e Armazenamento da Chave Privada 23
4.4.2 Formato de Dados e Gestão do `largeBlob` 23
4.4.3 Processo de Decifragem e Limpeza de Memória 23
4.5 Módulo de Serviço (Bunker NIP-46) 24
4.5.1 O Gerenciador de Chave `YubikeyKeyManager` 24
4.5.2 Serviço Bunker (NIP-46) 24
4.6 Implementação do Cliente de Validação 25

5 Análise de Segurança e Validação dos Resultados **26**
5.1 Validação Experimental do Protótipo 26
5.1.1 Confirmação do Critério de Sucesso Primário 26
5.1.2 Confirmação do Critério de Sucesso Secundário 27
5.2 Análise Qualitativa de Segurança 27
5.2.1 Discussão Detalhada das Mitigações 29
5.2.2 Análise de Vetores de Ataque Residuais 29

6 Conclusão **31**
6.1 Conclusão 31
6.2 Trabalhos Futuros 32

Referências **33**

Lista de Figuras

2.1 Comunicação entre Cliente e Relays: O cliente publica (Write) para múltiplos relays e subscreve (Read) para receber atualizações, mantendo a inteligência na borda. 6
2.2 Representação visual de uma Curva Elíptica utilizada em criptografia assimétrica. A segurança baseia-se na dificuldade de reverter a multiplicação de pontos. 8
2.3 Fluxo de mensagens do protocolo NIP-46 (Nostr Bunker). 11

3.1 Fluxo de mensagens e interação com hardware no protocolo proposto. . . . 17

Lista de Tabelas

5.1 Matriz de Análise de Segurança: Vetores de Ataque vs. Mitigações 28

Capítulo 1

Introdução

A internet contemporânea atravessa uma transformação estrutural com a ascensão de redes sociais descentralizadas. Nesse novo paradigma, protocolos como o Nostr (*Notes and Other Stuff Transmitted by Relays*) surgem com a promessa de devolver ao usuário a soberania sobre sua identidade digital [1]. Diferente do modelo tradicional, onde o acesso é concedido por servidores corporativos, no Nostr a identidade é autocustodiada: ela existe apenas como um par de chaves criptográficas sob posse exclusiva do usuário.

Essa liberdade, no entanto, impõe um dilema crítico de segurança. Ao remover o intermediário, remove-se também a rede de proteção; não há recuperação de senha na descentralização. O usuário torna-se o único guardião de sua chave privada (`nsec`). Se ela for perdida, a identidade desaparece; se for roubada, a identidade é permanentemente comprometida.

1.1 Contexto e Problemática: O Valor da Identidade e o Risco em Repouso

É fundamental distinguir a criticidade das chaves envolvidas. Nem todas as credenciais exigem o nível máximo de proteção; chaves de sessão efêmeras ou identidades descartáveis (*burner keys*) podem, muitas vezes, ser gerenciadas com menor rigor. No entanto, a chave mestra de identidade Nostr pertence a uma categoria distinta. Ela atua como a âncora criptográfica de toda a reputação, do grafo social e do histórico acumulado pelo indivíduo ao longo dos anos. O comprometimento desta chave específica é catastrófico: resulta na perda irreversível da identidade soberana, visto que não existe uma autoridade central capaz de revogá-la ou restaurá-la. É devido a esse valor inestimável e insubstituível que a proteção desta chave justifica uma arquitetura de segurança robusta.

Para navegar nesse ecossistema protegendo esse ativo de alto valor, o usuário enfrenta um desafio prático: ele precisa da chave acessível para assinar mensagens, mas deve mantê-la isolada para evitar roubos. Infelizmente, a solução adotada pela maioria das aplicações atuais prioriza a conveniência em detrimento da segurança. É comum que chaves mestras sejam armazenadas em arquivos de configuração locais, na área de transferência ou diretamente no armazenamento do navegador.

Este hábito de manter a "chave em repouso" em ambientes de software transforma a identidade do usuário em um alvo estático. A literatura acadêmica de segurança é enfática ao apontar que assegurar a integridade de chaves mestras em software é, na prática, um desafio de alta complexidade, dada a superfície de ataque inerente aos sistemas operacionais modernos [2]. Mesmo quando cifradas, as chaves persistidas em disco podem ser vulneráveis a ataques sofisticados de canal lateral (*side-channel attacks*) que exploram o próprio formato de armazenamento [3].

A indústria de segurança confirma essa fragilidade teórica com exemplos práticos alarmantes. *Malwares* dedicados, como as famílias **KPOT** [4] e **ElectroRAT** [5], foram documentados exfiltrando carteiras de criptomoedas simplesmente varrendo *logs* e arquivos locais, sem a necessidade de força bruta. O cenário é claro: um simples *software* malicioso que infecte o computador pode comprometer irrevogavelmente a identidade soberana do usuário. Pesquisas recentes em arquiteturas de recuperação de informação reforçam que a única defesa robusta para dados sensíveis é o isolamento criptográfico rigoroso, dificultando o acesso direto até mesmo pelo sistema hospedeiro [6].

1.2 A Abordagem Proposta: Isolamento via Hardware

É neste contexto de vulnerabilidade sistêmica que este trabalho se insere. A premissa aqui defendida é que, para um uso seguro de identidades descentralizadas de alto valor, a chave privada não deve residir no ambiente de software do usuário. Ela precisa estar fisicamente isolada.

Para concretizar essa visão, desenvolvemos e validamos uma solução que integra a robustez dos *tokens* de hardware (padrão FIDO2) com a flexibilidade do protocolo de assinatura remota Nostr Bunker (NIP-46). A proposta utiliza extensões avançadas do padrão FIDO2 para transformar dispositivos comerciais, como uma YubiKey, em cofres criptográficos portáteis.

Nesta arquitetura, a chave privada é provisionada de forma cifrada diretamente para o hardware seguro. Quando o usuário deseja publicar uma mensagem, o sistema utiliza o *token* para decifrar a chave momentaneamente na memória volátil, permitindo a assinatura

sem que o segredo toque o disco. Esse processo impõe, obrigatoriamente, múltiplos fatores de autenticação: algo que o usuário possui (o *token*), algo que ele sabe (o PIN) e a confirmação de presença física (o toque).

1.3 Objetivos

O objetivo geral deste trabalho é desenvolver e validar uma arquitetura de custódia de chaves para o protocolo Nostr que elimine a exposição da chave privada em repouso, mitigando vetores de ataque baseados em software por meio do uso de hardware de segurança comercialmente disponível.

1.3.1 Objetivos Específicos

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

- **Implementar um serviço Nostr Bunker (NIP-46):** Desenvolver uma aplicação em linguagem Rust capaz de atuar como um assinador remoto, intermediando a comunicação segura entre clientes Nostr e o hardware.
- **Integrar o padrão FIDO2/CTAP2:** Desenvolver módulos de comunicação de baixo nível para interagir com *tokens* de hardware, utilizando as extensões `hmac-secret` e `largeBlob` para operações criptográficas e armazenamento isolado.
- **Desenvolver um mecanismo de armazenamento cifrado:** Implementar um esquema de criptografia autenticada (AES-256-GCM) para assegurar que a chave privada só possa ser decifrada e utilizada mediante autenticação física no *token*.
- **Validar a solução em cenário real:** Demonstrar a funcionalidade da arquitetura por meio de um cliente de teste, realizando o ciclo completo de assinatura e publicação de eventos em *relays* públicos da rede Nostr.

1.3.2 Delimitação do Escopo

Para garantir a clareza sobre as contribuições deste trabalho, define-se o que **não** faz parte do escopo:

- **Não está no escopo** o desenvolvimento de novo hardware proprietário. A solução foca na utilização de *tokens* compatíveis com FIDO2 já existentes no mercado para garantir a acessibilidade.

- **Não está no escopo** o desenvolvimento de uma interface gráfica de usuário (GUI) para o serviço Bunker. O protótipo foca na validação da arquitetura de segurança e opera por meio de uma interface de linha de comando (CLI).
- **Não está no escopo** a proposição de alterações no protocolo Nostr. A solução adere estritamente aos padrões atuais (NIPs) para garantir compatibilidade imediata com o ecossistema.

Capítulo 2

Da Identidade Descentralizada à Segurança de Chaves em Hardware

Este capítulo estabelece os alicerces técnicos necessários para a compreensão da solução proposta. A discussão não parte de definições isoladas, mas segue a jornada do dado na rede: inicia-se pela arquitetura de redes, contrastando modelos centralizados com a abordagem do protocolo Nostr. Em seguida, detalha-se a estrutura fundamental desse dado (Eventos) e como a identidade do usuário está matematicamente vinculada a ele. Por fim, explora-se como o ecossistema atual lida com essas identidades, os riscos de segurança que emergem dessa prática e como as tecnologias de hardware (FIDO2) oferecem a base para mitigar esses riscos.

2.1 Arquiteturas de Rede e o Paradigma Nostr

A arquitetura da internet moderna consolidou-se majoritariamente sob o modelo centralizado. Neste paradigma, a inteligência da aplicação, a lógica de negócios e o armazenamento de dados residem em servidores controlados por uma autoridade central.

É importante reconhecer as vantagens que tornaram este modelo onipresente: ele oferece alta eficiência na indexação de dados, facilidade de escalabilidade vertical e uma experiência de usuário fluida, pois a complexidade é abstraída no servidor. No entanto, essas vantagens vêm acompanhadas de desvantagens estruturais graves para a soberania digital. O modelo cria "pontos únicos de falha", onde a entidade controladora possui poder unilateral para exercer censura ou remover identidades [7].

2.1.1 O Modelo "Cliente Inteligente, Servidor Burro"

O protocolo Nostr propõe uma ruptura com esse modelo. Conforme definido em sua especificação básica, ele opera como uma rede descentralizada baseada em chaves criptográficas [1]. É fundamental corrigir um equívoco comum: ele não opera como uma rede P2P pura (*mesh*). Ele mantém uma topologia cliente-servidor, mas inverte a distribuição de "inteligência".

No Nostr, os servidores são denominados ***Relays***. Diferente dos servidores tradicionais, os *Relays* são componentes passivos e "burros". Eles não executam lógica de negócios complexa e, crucialmente, **não se comunicam entre si** para sincronizar dados globalmente [8]. Cada *Relay* funciona como um silo de armazenamento independente e simplificado [9].

A inteligência da rede reside inteiramente nas bordas, ou seja, nos **Clientes** (*Clients*). É o aplicativo do usuário que decide quais *Relays* consultar para ler (*READ*) ou escrever (*WRITE*) informações, como ilustrado na Figura **??**. A descentralização emerge da liberdade de escolha do cliente [1]: se um *Relay* falha ou censura, o usuário move sua identidade para outro, sem perda de histórico.

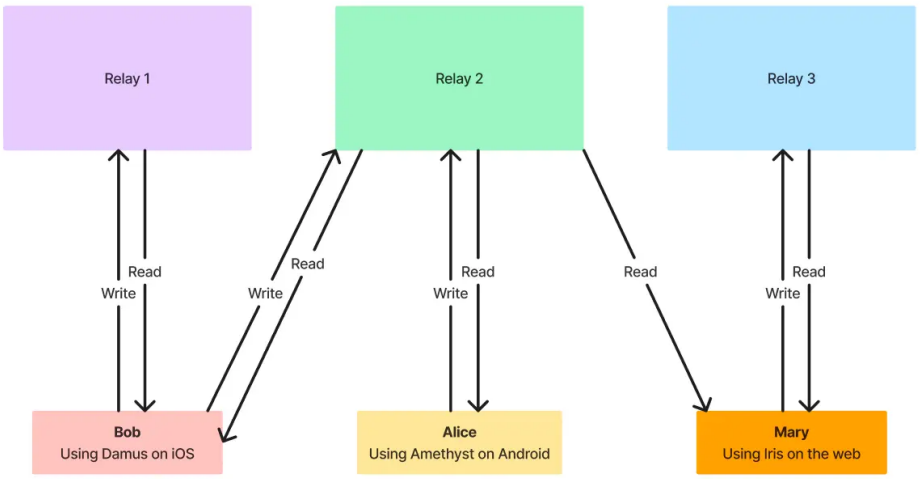

Figura 2.1: Comunicação entre Cliente e Relays: O cliente publica (Write) para múltiplos relays e subscreve (Read) para receber atualizações, mantendo a inteligência na borda.

2.2 Anatomia do Protocolo: Eventos e Identidade

Para que essa portabilidade de identidade funcione na prática, os dados precisam ser padronizados independentemente do servidor onde residem. A unidade atômica do protocolo é o **Evento** (*Event*).

De acordo com a NIP-01 (*Nostr Implementation Possibilities*) [10], todo conteúdo na rede — seja um texto, uma transação ou um perfil — é um objeto JSON com a seguinte estrutura básica:

```
{
  "id": "4376c65d...",        // Hash SHA-256 do conteúdo (Identificador)
  "pubkey": "3bf0c63f...",    // Chave pública do autor (Identidade)
  "created_at": 1673347337,   // Timestamp Unix
  "kind": 1,                  // Tipo do evento (1 = Texto, 0 = Metadados...)
  "tags": [["e", "5d6f..."]], // Referências a outros eventos
  "content": "Hello World",   // O conteúdo da mensagem
  "sig": "706af467..."        // Assinatura Digital Schnorr
}
```

Ao analisar essa estrutura, percebe-se que a segurança e a integridade do objeto dependem inteiramente de dois campos: `pubkey` e `sig`. Isso nos leva, necessariamente, aos primitivos criptográficos.

2.2.1 A Identidade Autocustodiada: Chaves e Assinaturas

A identidade no Nostr é desvinculada de qualquer registro civil ou e-mail; ela é puramente matemática, baseada em pares de chaves assimétricas sobre a curva elíptica `secp256k1`, a mesma padronizada pelo grupo SECG [11] e utilizada pelo Bitcoin. A Figura 2.2 ilustra a representação visual de uma curva elíptica sobre um corpo finito, base matemática para a geração destes pares.

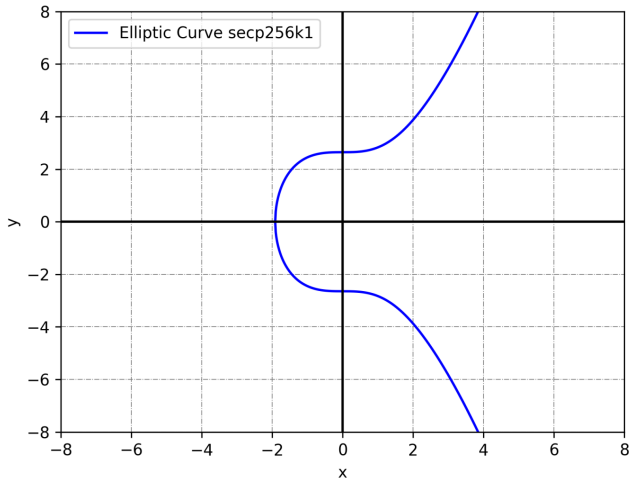

Figura 2.2: Representação visual de uma Curva Elíptica utilizada em criptografia assimétrica. A segurança baseia-se na dificuldade de reverter a multiplicação de pontos.

- **Chave Privada (`nsec`):** Um escalar secreto de 256 bits (um número inteiro muito grande). A posse deste número define a propriedade da identidade.
- **Chave Pública (`npub`):** Um ponto na curva derivado da chave privada. Atua como o identificador público na rede.

Assinaturas Schnorr (BIP-340)

Para assegurar que o campo `pubkey` no JSON não foi forjado por um terceiro, o protocolo exige que o campo `sig` contenha uma assinatura Schnorr válida. Este esquema foi escolhido por sua linearidade e eficiência, seguindo a especificação BIP-340 [12].

Matematicamente, a assinatura é gerada pela equação:

$$s = r + e \cdot x \pmod{n}$$

Onde as variáveis representam:

- s: A assinatura resultante (o valor escalar).
- r: O *nonce* secreto (um número aleatório gerado apenas para esta assinatura).
- e: O *hash* da mensagem concatenado com a chave pública e o nonce público (o desafio).
- x: A chave privada do usuário (o segredo que nunca deve ser revelado).
- n: A ordem da curva elíptica (uma constante do protocolo).

A validação, realizada por qualquer cliente ou *relay*, utiliza apenas a chave pública P:

$$s \cdot G = R + e \cdot P$$

Onde G é o ponto gerador da curva e R é o ponto público do *nonce*. Se a igualdade se sustenta, a rede aceita o evento como autêntico sem jamais ter acesso à chave privada x. A segurança repousa na intratabilidade do Problema do Logaritmo Discreto em curvas elípticas [13, 14].

2.3 Ecossistema de Aplicações

A flexibilidade desse modelo de eventos permitiu o surgimento de um ecossistema diversificado de aplicações, demonstrando a versatilidade do protocolo:

- **Primal e Damus:** Clientes de rede social que implementam *feeds* resistentes à censura [15, 16, 17].
- **Cashu:** Um sistema de *ecash* que utiliza o Nostr como camada de comunicação [18].
- **Zaplatte:** Aplicações de financiamento coletivo integradas nativamente ao protocolo via NIP-57 [19].

Apesar da variedade de casos de uso, todas essas aplicações compartilham um vetor crítico: para funcionar, elas exigem acesso à chave privada (`nsec`) para assinar as operações.

2.4 O Desafio da Custódia e o Nostr Bunker

Essa necessidade de acesso cria o problema central de segurança: o armazenamento inadequado da chave privada em múltiplos dispositivos e aplicações web expõe o usuário a riscos severos. Pesquisas indicam que chaves persistidas em ambientes de software estão vulneráveis a ataques de canal lateral e extração direta [2]. Exemplos práticos incluem *malwares* como o ElectroRAT, que drenam carteiras varrendo arquivos locais [5], e o KPOT [4].

2.4.1 Mitigação via NIP-46: Nostr Bunker

Para mitigar o risco de expor a chave mestra a cada novo aplicativo, a comunidade desenvolveu a especificação NIP-46, denominada **Nostr Bunker** [20].

O Bunker atua como um "assinador remoto". O fluxo de operação introduz uma camada de indireção vital:

1. **Conexão:** O Cliente solicita permissão de uso ao Bunker.
2. **Requisição:** O Cliente cria um evento, mas não o assina. Ele envia o JSON para o Bunker através de um *Relay*.
3. **Assinatura:** O Bunker recebe o evento, decifra a requisição e, se aprovado, assina com a `nsec` custodiada.
4. **Retorno:** O Bunker devolve a assinatura (`sig`) ao Cliente.

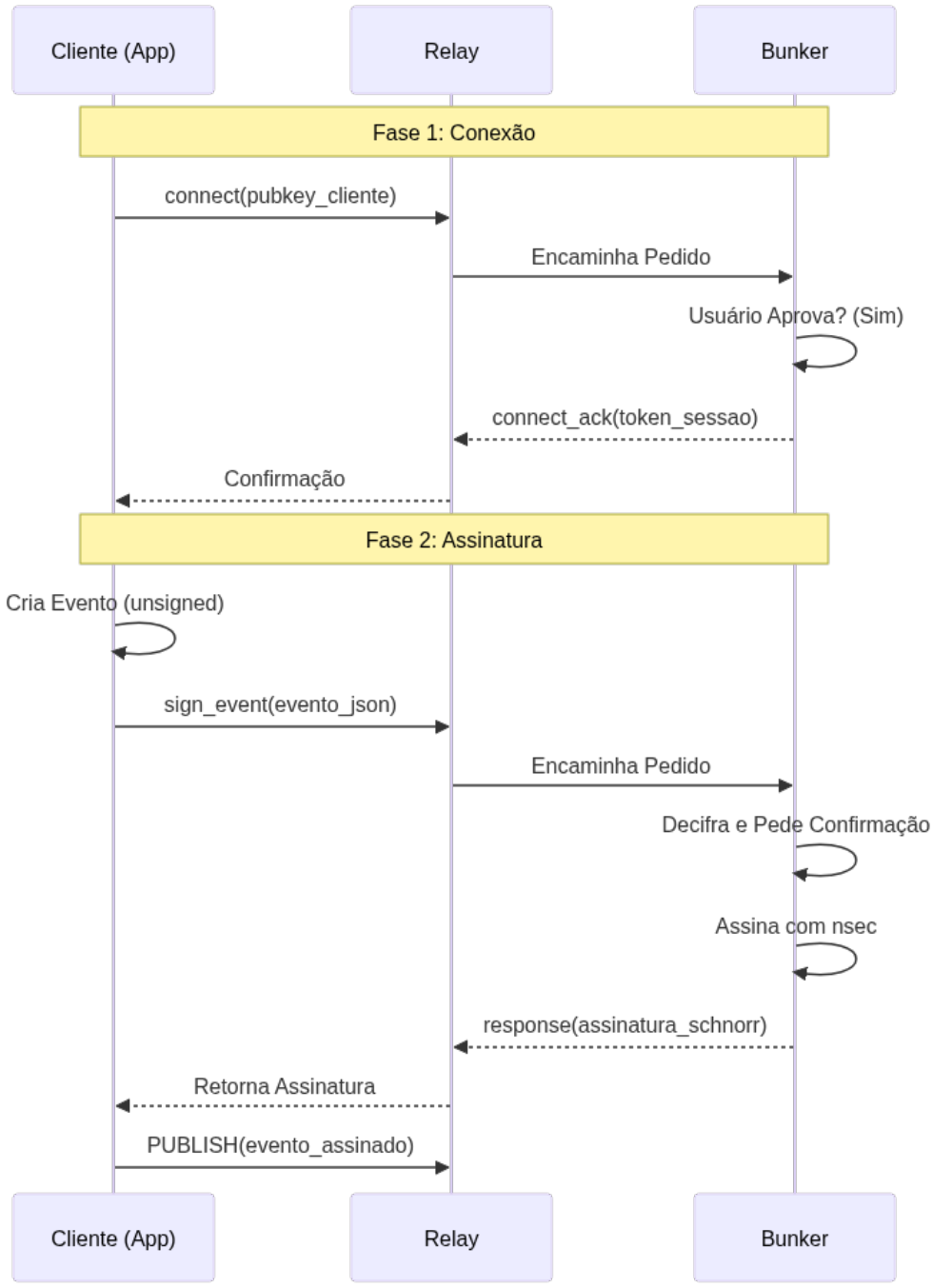

Figura 2.3: Fluxo de mensagens do protocolo NIP-46 (Nostr Bunker).

2.4.2 A Vulnerabilidade Residual

Embora o Bunker resolva o problema de compartilhar a chave com sites não confiáveis, ele apenas transfere a custódia para o servidor onde o Bunker roda [20]. Na maioria das implementações, a chave ainda reside no disco rígido desse servidor ("chave em repouso"),

vulnerável a ataques de sistema operacional [21]. Além disso, se o canal de autenticação do Bunker for comprometido, um atacante pode assinar remotamente.

2.5 Segurança Baseada em Hardware

Uma solução eficaz para o problema da "chave em repouso" reside na remoção da chave do ambiente de software (disco/RAM persistente), delegando sua custódia a um hardware especializado projetado para resistir a ataques físicos e lógicos.

2.5.1 O Princípio do Isolamento Físico

Tokens de segurança de hardware, como a YubiKey, são dispositivos construídos em torno de um componente crítico: o ***Secure Element*** (SE). Este é um microchip dedicado, resistente a adulterações físicas e lógicas, projetado para armazenar segredos criptográficos e realizar operações sensíveis internamente. Diferente de um processador de computador comum, o *Secure Element* é arquitetado para impedir que as chaves privadas deixem o dispositivo em texto claro.

2.5.2 O Padrão FIDO2 e o Protocolo de Transporte CTAP

Para padronizar a segurança desses dispositivos, a indústria desenvolveu o **FIDO2** (*Fast Identity Online*). Este padrão define não apenas os mecanismos de autenticação forte, mas também um conjunto de extensões e capacidades avançadas que o hardware deve implementar [22].

Para que a aplicação (o Bunker) possa acessar essas funcionalidades do padrão FIDO2, é necessário um meio de transporte. É aqui que entra o **CTAP** (*Client to Authenticator Protocol*). O CTAP atua estritamente como o canal de comunicação de baixo nível, padronizando como os comandos binários trafegam via USB ou NFC entre o sistema operacional e o *token* [23].

2.5.3 Ideal vs. Realidade: O Problema das Curvas

No cenário ideal, utilizaríamos o fluxo padrão do FIDO2 diretamente: a chave privada Nostr seria gerada dentro do *Secure Element* e todas as assinaturas seriam feitas nativamente pelo hardware.

No entanto, enfrentamos uma barreira matemática. O padrão FIDO2 prioriza a curva elíptica **NIST P-256**. O protocolo Nostr, seguindo a base do Bitcoin, exige a curva `secp256k1` [11]. Análises técnicas confirmam que essas curvas são incompatíveis [24],

impedindo que a maioria dos *tokens* comerciais realize nativamente a assinatura Schnorr necessária para o Nostr.

2.5.4 A Solução: Envelope Criptográfico via Extensões FIDO2

Diante da impossibilidade de assinatura nativa, este trabalho propõe uma abordagem alternativa utilizando extensões do protocolo FIDO2 acessíveis via CTAP [25]:

- **`hmac-secret`:** Uma extensão que permite derivar uma chave simétrica efêmera a partir de um segredo mestre interno. Essa chave derivada serve como o segredo criptográfico para proteger dados externos, e sua liberação exige prova de presença (toque) e PIN [26].
- **`largeBlob`:** Uma funcionalidade que permite armazenar objetos binários arbitrários na memória não volátil do *token*, associados a uma credencial.

A estratégia adotada é utilizar o *token* para **cifrar e armazenar** a chave Nostr. A `nsec` repousa dentro do `largeBlob`, protegida pelo segredo do `hmac-secret`. Ela é recuperada para a memória RAM momentaneamente apenas para a assinatura, mitigando o risco de extração em repouso e viabilizando a portabilidade segura da identidade.

Justificativa Arquitetural: Princípio da Exposição Mínima

A arquitetura proposta segue o princípio de **minimização da superfície de ataque**, recomendado pelo NIST SP 800-57 Part 1 [21]. É crucial distinguir que a chave de identidade (`nsec`) é estática e não pode ser derivada matematicamente a cada operação, sob pena de perda da identidade digital. A inovação reside na derivação da **chave de envelope** (a chave simétrica obtida via `hmac-secret`).

Ao manter a `nsec` cifrada em repouso e derivar a chave de decifragem apenas no instante da operação (descartando-a da memória volátil imediatamente após o uso), a solução adere às recomendações de gerenciamento de chaves do NIST SP 800-57 sobre a proteção de informações sensíveis durante o armazenamento. Diferentemente do armazenamento em texto claro (vulnerável a CWE-312 [27]), ou em disco, essa abordagem de "decifragem *just-in-time*" reduz drasticamente a janela de tempo em que a chave reside em locais acessíveis à memória do sistema, dificultando vetores de extração forense e leitura de RAM.

Capítulo 3

Arquitetura e Validação Experimental da Solução Proposta

Este capítulo detalha a metodologia adotada para validar a hipótese de que é possível conciliar a segurança do isolamento de hardware com a usabilidade do protocolo Nostr. O objetivo é demonstrar a viabilidade prática da arquitetura, descrevendo o ambiente de testes, os fluxos operacionais que governam o sistema e os critérios rigorosos utilizados para validar seu funcionamento. Por fim, analisa-se as limitações inerentes ao protótipo e suas potenciais evoluções.

3.1 Configuração do Ambiente de Validação

Para assegurar que os resultados reflitam um cenário de uso real, e não apenas teórico, o ambiente de testes foi configurado para operar diretamente na infraestrutura pública da rede Nostr [1]. A arquitetura de validação é composta por três elementos principais que interagem entre si:

Primeiramente, a comunicação é mediada por ***Relays* Nostr Públicos** (como `wss://relay.damus.io` e `wss://nos.lol`) [15]. A escolha por não utilizar um ambiente local controlado (como um contêiner Docker isolado) foi deliberada: o objetivo é assegurar a interoperabilidade com a infraestrutura de produção e validar a robustez da implementação frente a *relays* reais, garantindo que a solução funcione conforme a especificação em rede pública.

Na ponta do usuário, foi desenvolvido o **Cliente de Teste** (`bunker_client.rs`). Trata-se de uma aplicação de linha de comando em Rust, projetada especificamente para este trabalho. Sua função é dupla: atuar como um cliente NIP-46 padrão (iniciando a conexão e enviando requisições) e, crucialmente, atuar como auditor, verificando se os eventos assinados e publicados são válidos e aceitos pela rede [20].

O núcleo da solução é o **Nostr Bunker com Suporte a Hardware**. Esta aplicação central, também em Rust, executa na máquina hospedeira e gerencia a ponte crítica entre o mundo inseguro (a internet/*relays*) e o mundo seguro (o dispositivo FIDO2 via USB) [22, 25].

3.2 Operação do Protótipo: Do Registro à Assinatura

A operação da solução divide-se em dois momentos distintos: um fluxo de preparação (*offline*), onde a segurança é estabelecida, e um fluxo de operação (*online*), onde a utilidade é entregue.

3.2.1 O Processo de Registro Seguro (Offline)

Antes de qualquer interação com a rede, é necessário vincular a identidade do usuário ao hardware. Este procedimento ocorre inteiramente local, sem tráfego de rede:

1. **Criação da Credencial FIDO2:** A aplicação instrui o *token* a criar uma nova credencial vinculada ao domínio do Bunker, habilitando a extensão `hmac-secret` [25]. Este passo exige a definição ou inserção do PIN do usuário, protegendo o *token* contra uso não autorizado.
2. **Derivação da Chave de Cifragem:** Uma vez autenticado, a aplicação gera um *salt* aleatório de 32 bytes e o envia ao *token*. O dispositivo utiliza seu segredo mestre interno para derivar uma chave simétrica (AES-256) única para essa identidade [26]. É fundamental notar que essa chave de cifragem só existe na memória volátil durante este processo.
3. **Cifragem e Armazenamento:** A chave privada (`nsec`) é então cifrada com o algoritmo **AES-256-GCM**. O resultado — uma carga útil (*payload*) contendo o *salt*, o *nonce* e o texto cifrado com a tag de autenticação — é gravado na memória persistente do *token* através da extensão `largeBlob` [25]. Ao final, a aplicação realiza a limpeza segura da memória (*zeroize*), removendo qualquer vestígio da chave em texto claro [27].

3.2.2 O Processo de Assinatura de Eventos (Online)

Com a identidade segura no hardware, o fluxo de assinatura demonstra a operação do sistema. É importante destacar que o túnel de comunicação entre o Cliente e o Bunker

(NIP-46) é, ele próprio, cifrado (NIP-44) usando chaves efêmeras, preservando a privacidade do usuário mesmo em *relays* públicos [20, 28].

O diagrama apresentado na Figura 3.1 ilustra o fluxo detalhado que ocorre dentro do bunker customizado, destacando a interação entre software e hardware.

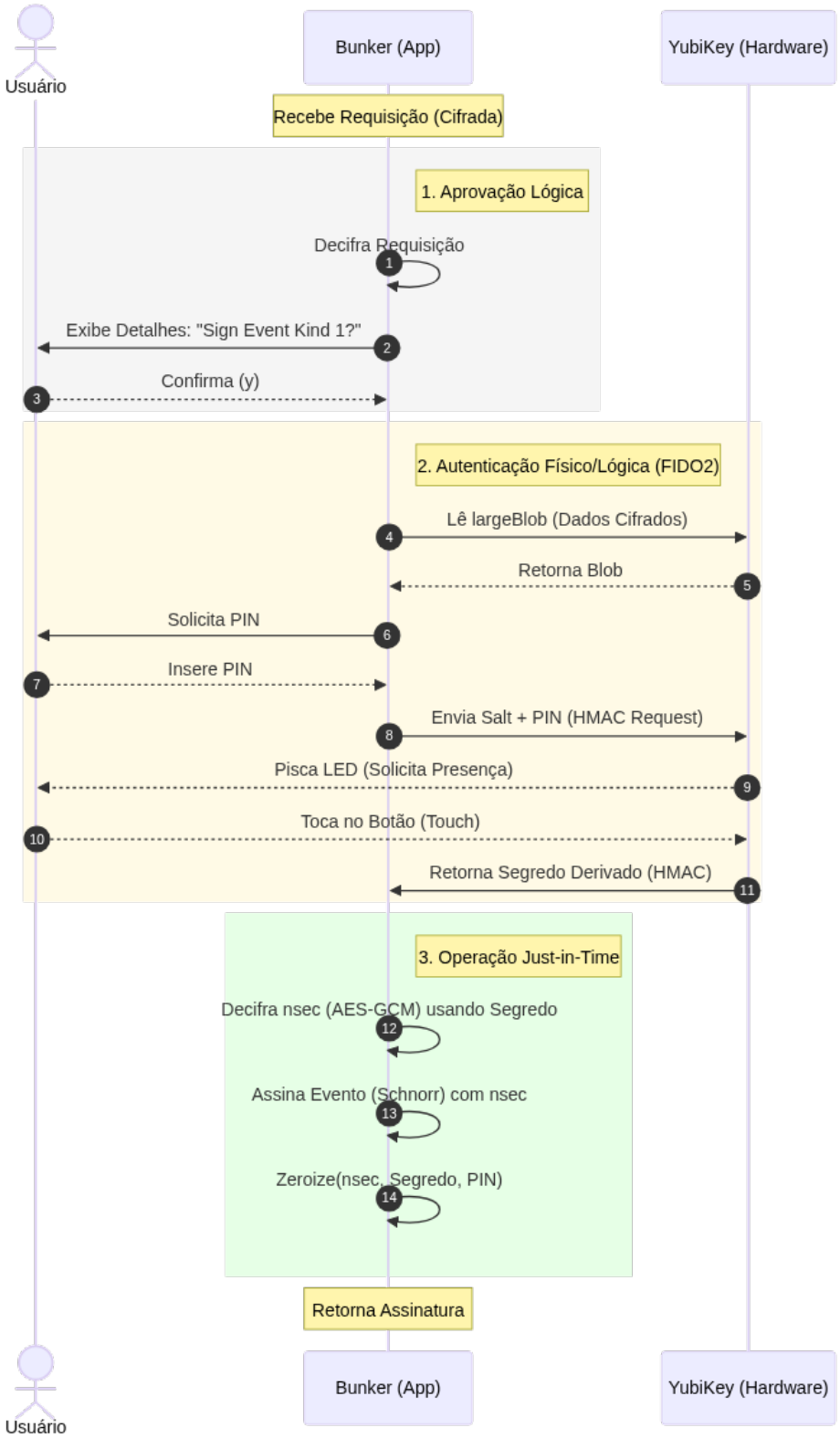

Figura 3.1: Fluxo de mensagens e interação com hardware no protocolo proposto.

O processo detalhado ocorre da seguinte forma:

1. **Requisição:** O usuário, através do `bunker_client.rs`, solicita a assinatura de um evento (ex: uma nota de texto). O cliente envia a representação JSON do evento (ainda não assinado) encapsulado em uma requisição JSON-RPC cifrada para o Bunker.

2. **Processamento Seguro:** O Bunker recebe e decifra a requisição. Antes de qualquer acesso ao hardware, ele exibe o conteúdo do evento ao usuário e solicita uma confirmação explícita.

3. **Interação com Hardware:** Após a aprovação, o sistema solicita o PIN e o toque físico no *token*. O Bunker lê o `largeBlob` cifrado, envia o *salt* para o *token* e recupera a chave de cifragem.

4. **Decifragem *Just-in-Time*:** A `nsec` é decifrada na memória RAM apenas pelo tempo estritamente necessário (milissegundos) para realizar a assinatura Schnorr no *hash* do evento [12, 14]. Imediatamente após a operação, tanto a chave privada quanto a chave de cifragem são sobrescritas com zeros na memória [21].

5. **Resposta e Verificação:** A assinatura gerada é enviada de volta ao cliente. O `bunker_client.rs` então publica o evento assinado nos *relays* e, como etapa final de validação, monitora a rede para confirmar que o evento foi propagado e aceito, validando criptograficamente a assinatura pública [12].

3.3 Validação e Critérios de Sucesso

Para validar o funcionamento da prova de conceito, foram definidos os seguintes critérios:

- **Critério de Sucesso Primário (Funcionalidade):** O cliente de teste (`bunker_client.rs`) deve ser capaz de completar com sucesso o fluxo de assinatura e, em seguida, ler de volta o evento publicado nos *relays* públicos, validando criptograficamente sua própria assinatura [12]. Isso evidencia que o ciclo de decifragem, assinatura, publicação e verificação foi bem-sucedido e produziu um evento Nostr válido.

- **Critério de Sucesso Secundário (Prova de Dependência):** O sistema deve falhar de forma segura (*fail-safe*) caso qualquer fator de autenticação esteja ausente. Testes de "caminho infeliz" devem confirmar que a operação é abortada se: (a) o *token* não estiver conectado, (b) o PIN for inválido, ou (c) o usuário negar a confirmação de presença (toque). Isso demonstra o isolamento efetivo da chave [22].

3.4 Análise Crítica e Limitações do Protótipo

Embora funcional, este protótipo focado na validação arquitetural possui limitações inerentes:

- **Dependência de Hardware Específico:** A solução exige *tokens* que suportem especificamente as extensões `largeBlob` e `hmac-secret` do CTAP 2.1 [25]. Dispositivos FIDO2 mais antigos ou modelos básicos podem não ser compatíveis.
- **Experiência do Usuário (UX):** A configuração inicial e a operação via linha de comando (CLI) exigem um nível de conhecimento técnico que representa uma barreira de entrada para o usuário comum.
- **Capacidade de Armazenamento:** O uso do `largeBlob` impõe um limite físico de armazenamento (geralmente 1KB a 4KB dependendo do fabricante). Com a implementação atual, isso limita o número de identidades simultâneas que podem ser armazenadas no mesmo *token* (aproximadamente 7 chaves no protótipo atual) [25].

Capítulo 4

Implementação do Bunker com Suporte a Hardware FIDO2

Este capítulo apresenta os detalhes técnicos da implementação da prova de conceito, focando na arquitetura de software do serviço Bunker e seu cliente de validação. O objetivo é descrever como os conceitos teóricos e a metodologia foram traduzidos em uma aplicação funcional, detalhando os módulos de gerenciamento de identidade, interação com o hardware, o sistema criptográfico de armazenamento e a lógica de serviço que habilita a assinatura de eventos Nostr de forma segura.

4.1 Arquitetura do Software e Tecnologias Adotadas

A aplicação foi desenvolvida integralmente em Rust, linguagem escolhida por suas fortes garantias de segurança de memória e performance. A arquitetura foi dividida em módulos lógicos, cada um com responsabilidades bem definidas, refletidos na estrutura de arquivos do projeto (ex: `credential.rs`, `encryption.rs`, `yubikey_bunker.rs`). As principais tecnologias e conceitos empregados foram:

- **Interface FIDO2/CTAP2:** A biblioteca `ctap-hid-fido2` foi utilizada para a comunicação de baixo nível com dispositivos FIDO2 via USB [22, 25].
- **Criptografia Autenticada:** Módulos criptográficos (`aes-gcm`) para a implementação do algoritmo **AES-256-GCM** [21].
- **Ecossistema Nostr:** As bibliotecas `nostr-sdk` e `nostr-relay-pool` foram usadas para toda a comunicação de rede, manipulação de eventos, assinaturas Schnorr [12] e criptografia NIP-44. A biblioteca `nostr-connect` foi a base para a implementação do cliente NIP-46 [20, 28].

- **Interação com Usuário:** A biblioteca `dialoguer` foi utilizada para as confirmações de assinatura (ex: "Sign this event?") e `rpassword` para a leitura segura do PIN.
- **Segurança de Memória:** A biblioteca `zeroize` foi empregada para sobrescrever com zeros os segredos em memória (PINs, chaves de cifragem, chaves privadas) imediatamente após o uso [21].

4.2 Módulo de Gerenciamento de Identidade (Modo Offline)

A aplicação principal (`main.rs`) fornece uma interface de linha de comando com duas modalidades: um módulo de gerenciamento de chaves e o serviço Bunker. O módulo de gerenciamento ("Manage Keys") opera de forma *offline*, permitindo ao usuário provisionar e gerenciar com segurança as identidades `nsec` dentro do *token* FIDO2.

4.2.1 Provisionamento de Chave (Store key)

Este fluxo permite ao usuário adicionar uma `nsec` existente ao *token*:

1. O usuário seleciona a opção "Store key".
2. A aplicação solicita um ID textual para a nova entrada e, em seguida, que o usuário cole a sua chave privada `nsec`.
3. **Este é o momento de maior vulnerabilidade da chave no ciclo de vida da solução.** A `nsec` existe momentaneamente em texto claro na memória da aplicação, no *buffer* de entrada.
4. Imediatamente, a função `write_blob` é invocada. Esta função coordena o Módulo Criptográfico (seção 4.4) para executar o processo completo de cifragem e armazenamento da `nsec` no *token* FIDO2 [25].

4.2.2 Leitura e Exclusão de Chaves (Read/Delete key)

O módulo também fornece funcionalidades de manutenção:

- **Leitura de Chaves:** Ao selecionar "Read key", a função `read_blob` é chamada. Ela lê o `largeBlob`, processa as entradas (`ID:payload_base64`), exibe um menu numerado de IDs ao usuário, e, após a seleção, invoca a função `decrypt_data` para decifrar e exibir a `nsec` selecionada.

- **Exclusão de Chaves:** Ao selecionar "Delete key", a função `delete_single_entry` é chamada. Ela exibe o mesmo menu de IDs, mas, após a seleção, remove a entrada correspondente da *string* do `largeBlob` e reescreve o objeto binário atualizado no dispositivo FIDO2.

Todas as operações de leitura, escrita e exclusão exigem autenticação do usuário via PIN do *token* [22].

4.3 Módulo de Interface com o Hardware FIDO2

Este módulo, implementado em `device.rs` e `credential.rs`, é responsável por toda a comunicação direta com o *token* de segurança.

1. **Descoberta e Validação:** A função `find_fido_device()` localiza o *token*. Em seguida, a função `is_supported()` verifica se o dispositivo possui suporte à extensão `LargeBlobs`, abortando a operação caso negativo [25].
2. **Gerenciamento de Credencial:** A função `get_credential_id()` gerencia a credencial FIDO2. Ela utiliza o `RP_ID` "nostr.bunker.yubikey". De forma inteligente, ela primeiro tenta obter uma asserção (`get_assertion`); se falhar, assume que uma credencial não existe e inicia o fluxo de criação (`make_credential`), habilitando explicitamente a extensão `HmacSecret(Some(true))` [22].
3. **Derivação da Chave:** A função `get_hmac_secret()` implementa a derivação da chave de cifragem. Ela solicita o PIN ao usuário (via `get_pin_from_user`), envia o *salt* de 32 bytes ao *token* via `GetExtension::HmacSecret`, e recebe de volta a chave AES-256 de 32 bytes. O PIN é zerado da memória com `pin.zeroize()` após cada uso [21].

4.4 Módulo Criptográfico e de Armazenamento Seguro

Este módulo (implementado em `encryption.rs` e `blob_operations.rs`) constitui o núcleo da solução de segurança.

4.4.1 Processo de Cifragem e Armazenamento da Chave Privada

A implementação utiliza o algoritmo **AES-256-GCM**, que oferece confidencialidade, integridade e autenticidade [21]. O processo de cifragem (`encrypt_data`) ocorre da seguinte forma:

1. Um *salt* de 32 bytes e um *nonce* de 12 bytes são gerados aleatoriamente.
2. A chave de cifragem AES-256 é derivada pelo Módulo de Interface com o Hardware, utilizando o *salt* [25].
3. A `nsec` é cifrada com AES-GCM, produzindo o texto cifrado e a *tag* de autenticação.
4. A carga útil (*payload*) binária final é construída pela concatenação de [*salt* (32 bytes) | *nonce* (12 bytes) | *ciphertext-com-tag* (48 bytes)], totalizando 92 bytes.

4.4.2 Formato de Dados e Gestão do `largeBlob`

O *payload* binário de 92 bytes é codificado em **Base64**. Cada identidade armazenada no `largeBlob` segue um formato de texto: `ID:payload_base64`, com múltiplas entradas separadas pelo caractere '|'. A implementação gerencia o limite de 1024 bytes do `largeBlob` através da função `handle_space_management`, que permite ao usuário remover entradas antigas para liberar espaço [25].

4.4.3 Processo de Decifragem e Limpeza de Memória

O processo de decifragem (`decrypt_data`) é executado de forma *just-in-time*:

1. O *payload* em Base64 é lido do `largeBlob` e decodificado para seu formato binário.
2. O *salt* (bytes 0-31), o *nonce* (bytes 32-43) e o *ciphertext-com-tag* (bytes 44 em diante) são extraídos do *payload*.
3. O *salt* é usado para re-derivar a chave de decifragem junto ao *token* (via `get_hmac_secret`).
4. A decifragem é executada. A *tag* de autenticação GCM é verificada implicitamente pelo algoritmo; qualquer modificação nos dados armazenados resultará em uma falha de decifragem, assegurando a integridade da `nsec`.
5. Após o uso, a chave de cifragem (`hmac_secret`) é sobrescrita com zeros via `zeroize()` [21].

4.5 Módulo de Serviço (Bunker NIP-46)

Este módulo, implementado em `yubikey_bunker.rs` e `yubikey_helper.rs`, é a camada de aplicação que opera como o Nostr Bunker [20].

4.5.1 O Gerenciador de Chave `YubikeyKeyManager`

O `YubikeyKeyManager` (`yubikey_helper.rs`) é a principal camada de abstração de segurança.

1. **Inicialização:** Ao iniciar o serviço Bunker, uma instância do `YubikeyKeyManager` é criada. Em seu construtor (`new()`), ele inicializa o dispositivo FIDO2 e chama `select_and_read_entry`. Isso força o usuário a selecionar qual identidade, dentre as armazenadas no `largeBlob`, será usada para a sessão atual do Bunker. A *chave pública* desta identidade é lida, validada e armazenada em cache (`cached_public_key`).
2. **Operação *Just-in-Time*:** A função `with_key()` é o coração desta arquitetura. Ela recebe uma operação (um *closure*) como argumento. Dentro dela, ela chama `load_private_key()`, que executa todo o processo de decifragem *just-in-time* (solicitação de PIN, derivação de HMAC, decifragem AES-GCM) para obter a `nsec`. A `nsec` (em um objeto `Keys`) é então passada para a operação. Imediatamente após a conclusão, a `Keys` é descartada (`drop()`), e sua implementação de `Zeroize` garante que a chave privada seja limpa da memória.

4.5.2 Serviço Bunker (NIP-46)

O módulo `yubikey_bunker.rs` implementa a lógica do servidor NIP-46.

1. **Identidade do Bunker:** Ao ser inicializado, o Bunker gera um par de chaves Nostr **temporário** (`Keys::generate()`), que é usado exclusivamente para a comunicação NIP-46. A chave pública deste par é usada para gerar o URI `nostrconnect://`.
2. **Comunicação de Rede:** O serviço se conecta aos *relays* públicos definidos no `.env` e assina eventos `Kind::NostrConnect` direcionados à sua chave pública de serviço.
3. **Processamento de Requisições:** A função `handle_request` processa os eventos recebidos. Ela utiliza `nip44::decrypt` para decifrar as requisições do cliente.
4. **Confirmação do Usuário:** Antes de executar qualquer operação sensível (ex: `SignEvent`, `Nip04Encrypt`, etc.), a função `should_approve` é chamada. Ela usa `dialoguer::Confirm` para exibir um *prompt* legível ao usuário (ex: “Sign this event?”) e aguarda uma aprovação explícita.

5. **Execução Segura:** Para todas as operações que requerem a chave privada do usuário (ex: `SignEvent`, `Nip04Decrypt`), o Bunker invoca o `yubikey_manager.with_key(...)`, garantindo que a `nsec` só exista na memória pelo tempo estritamente necessário para a operação.

6. **Resposta:** A resposta da operação é encapsulada em uma `NostrConnectResponse`, cifrada com `nip44::encrypt`, e enviada de volta ao cliente via *relay*.

4.6 Implementação do Cliente de Validação

Para validar a funcionalidade do Bunker, foi desenvolvido um cliente de teste dedicado, denominado `bunker_client.rs`, utilizando a biblioteca `nostr-connect` [20, 28]. Este cliente não faz parte do Bunker, mas sim da metodologia de teste, e é responsável por simular um cliente Nostr real.

1. **Conexão NIP-46:** O cliente solicita ao usuário que cole o URI `nostrconnect://` gerado pelo Bunker. Ele processa este URI, gera seu próprio par de chaves temporário (`Keys::generate()`) e instancia um objeto `NostrConnect`.

2. **Aprovação da Conexão:** O cliente inicia a conexão enviando uma requisição `get_public_key`. Isso dispara a primeira interação com o Bunker, que por sua vez solicita ao usuário (via `dialoguer`) que aprove a conexão do novo cliente.

3. **Interface de Teste Interativa:** Após a aprovação, o cliente exibe um menu interativo que permite ao usuário executar todas as principais operações do NIP-46: `Sign event (kind 1)`, `NIP-04 Encrypt/Decrypt` ou `NIP-44 Encrypt/Decrypt`.

4. **Fluxo de Assinatura, Publicação e Verificação:** Ao testar a assinatura, o cliente constrói um `EventBuilder` e invoca `event.sign(signer).await`. A biblioteca `nostr-connect` lida com o envio da requisição `sign_event` para o Bunker e aguarda a resposta. Após receber o evento assinado, o cliente de teste se conecta a um *relay* público e publica o evento. Conforme sugerido na saída do programa, o cliente então realiza (ou instrui o usuário a realizar) uma leitura de volta do *relay* para confirmar que o evento foi aceito e que sua assinatura é criptograficamente válida, provando o sucesso do ciclo completo.

Capítulo 5

Análise de Segurança e Validação dos Resultados

Após detalhar a arquitetura e a implementação da solução, torna-se imperativo verificar se as hipóteses levantadas na introdução se sustentam na prática. Este capítulo apresenta os resultados obtidos com a prova de conceito, discutindo suas implicações diretas para a segurança do usuário no protocolo Nostr.

A discussão inicia-se com a validação experimental, provendo as evidências empíricas de que o sistema opera conforme desenhado. Em seguida, eleva-se o nível de abstração para uma análise qualitativa de segurança, sintetizada em uma matriz comparativa que confronta a arquitetura proposta contra os vetores de ataque clássicos e discute, com transparência técnica, os riscos residuais que permanecem no sistema.

5.1 Validação Experimental do Protótipo

A eficácia de um sistema de segurança não pode ser medida apenas teoricamente. Conforme a metodologia descrita, o protótipo foi submetido a um cenário de uso realista, operando diretamente na infraestrutura pública da rede Nostr [1]. Os testes foram desenhados para confirmar dois pilares fundamentais: a funcionalidade operacional e o isolamento criptográfico.

5.1.1 Confirmação do Critério de Sucesso Primário

O objetivo primário era demonstrar que é possível assinar eventos validamente sem que a chave toque o disco. Este teste foi realizado utilizando o cliente de auditoria `bunker_client.rs` em um ciclo completo:

1. O cliente gerou um evento Nostr (*Kind 1*) e o enviou para o Bunker através de um *relay* público, seguindo a especificação NIP-46 [20].
2. O Bunker processou a requisição, exigindo interação física (toque) e lógica (PIN) com o *token* FIDO2.
3. A assinatura foi gerada com sucesso e retornada ao cliente.
4. Crucialmente, o cliente publicou o evento na rede e, em seguida, consultou *relays* independentes para verificar a propagação.

A validação criptográfica da assinatura Schnorr [12] confirmou que o evento foi assinado corretamente pela identidade protegida no hardware. **Este resultado fornece evidência empírica da viabilidade da solução**, executando a decifragem segura *just-in-time* e a publicação sem falhas de integridade.

5.1.2 Confirmação do Critério de Sucesso Secundário

Tão importante quanto funcionar quando deve, é falhar quando necessário. Para verificar o isolamento da chave, o sistema foi submetido a testes de interrupção intencionais: (a) remoção física do *token* durante a operação, (b) inserção de credenciais (PIN) inválidas e (c) recusa explícita no *prompt* de confirmação do usuário.

Em todos os cenários, o Bunker abortou a operação imediatamente, assegurando que nenhum dado sensível fosse exposto ou utilizado indevidamente. **Isso indica que o isolamento da chave privada `nsec` é efetivo**, dependendo estritamente da tríade de segurança: posse do *token*, conhecimento do segredo e consentimento humano [22].

5.2 Análise Qualitativa de Segurança

Além da validação funcional, é necessário avaliar a resiliência da arquitetura frente ao cenário de ameaças hostil descrito na fundamentação teórica. A Tabela 5.1 apresenta a Matriz de Análise de Segurança, confrontando os vetores de ataque contra as mitigações implementadas.

Tabela 5.1: Matriz de Análise de Segurança: Vetores de Ataque vs. Mitigações

Vetor de Ataque	**Análise de Mitigação na Solução Proposta**
Ataque à Chave em Repouso (*Malware* de varredura de disco)	**Seguro.** A chave reside exclusivamente no armazenamento do *token* (`largeBlob`). A arquitetura garante que não existam arquivos persistentes contendo a chave no disco rígido.
Ataque por Leitura Direta (Atacante obtém posse física do *token*)	**Seguro.** A chave está cifrada (AES-256-GCM) e o objeto binário é protegido pela extensão `hmac-secret`, exigindo o PIN do usuário para liberar a chave de decifragem.
Ataque de Assinatura Maliciosa (Assinatura cega ou não autorizada)	**Mitigado.** Pode ser evitado pela exigência de confirmação visual no Bunker (verificação do conteúdo) e pela obrigatoriedade do toque físico (*User Presence*) na chave.
Comprometimento do Canal NIP-46 (Atacante obtém chaves temporárias do cliente)	**Parcialmente Mitigado.** Isso permitiria ao atacante ler as notas antes de serem assinadas (perda de privacidade), mas ele **não pode assinar** novas mensagens devido à barreira física do hardware no lado do Bunker.
Ataque de Leitura de Memória (*RAM Scraping* no hospedeiro)	**Vulnerável.** Durante a leitura da chave para efetuar a assinatura, a `nsec` fica exposta em texto claro na memória RAM por milissegundos. Este é o risco residual inerente à incompatibilidade de curvas.

5.2.1 Discussão Detalhada das Mitigações

A seguir, detalha-se como as camadas de proteção apresentadas na Tabela 5.1 operam tecnicamente para mitigar cada classe de risco.

Mitigação de Ataques de Extração de Chave

O vetor de ataque mais crítico para usuários de identidade soberana é o roubo da chave privada ("extração").

Em carteiras de *software* tradicionais, a chave reside no sistema de arquivos, vulnerável a *malwares* de varredura como o ElectroRAT [5]. A arquitetura proposta **mitiga eficazmente este vetor**. A `nsec` nunca é gravada no disco do computador hospedeiro; ela reside exclusivamente, de forma cifrada, dentro da memória não volátil (`largeBlob`) do *token* FIDO2 [25].

Mesmo no cenário de acesso físico ao *token* (Leitura Direta), o acesso ao `largeBlob` e a operação de derivação da chave de cifragem são protegidos pelo *Secure Element*. O uso de criptografia autenticada (AES-GCM) protege a integridade dos dados, dificultando ataques que tentem modificar o texto cifrado para induzir erros reveladores (ataques de oráculo) [21].

Mitigação de Ataques de Abuso de Assinatura

Nesta categoria, o atacante não tenta roubar a chave, mas sim usar a "caneta" do usuário para assinar algo malicioso, como em um ataque de Assinatura Cega (*Blind Signing*).

Um cliente comprometido poderia tentar enviar eventos maliciosos para o Bunker assinar silenciosamente. A implementação mitiga este risco através da função `should_approve`, que intercepta a operação e exige uma confirmação explícita no terminal [20]. A interface exibe um resumo do conteúdo do evento, truncando mensagens excessivamente longas para prevenir ataques de inundação visual, permitindo que o usuário inspecione o contexto da assinatura. Isso reinsere o ser humano no ciclo de decisão, transformando uma assinatura cega em uma assinatura consentida.

5.2.2 Análise de Vetores de Ataque Residuais

Uma análise de segurança transparente deve mapear não apenas o que foi resolvido, mas os riscos que permanecem.

Comprometimento do Canal NIP-46

A comunicação entre Cliente e Bunker utiliza chaves efêmeras e criptografia NIP-44 [28]. Se um atacante comprometer a máquina do cliente (ataque de intermediário), ele poderia ler o conteúdo de notas privadas antes de serem publicadas. No entanto, é crucial distinguir o impacto: **este ataque não compromete a chave privada principal (`nsec`)**, que jamais trafega pelo canal. Além disso, tentativas de assinatura fraudulenta seriam barradas pela etapa de confirmação manual no Bunker.

Ataque de Leitura de Memória (*RAM Scraping*)

Conforme indicado na Tabela 5.1, este é o vetor de maior complexidade e o principal risco residual da solução. Como o hardware não suporta a curva `secp256k1` nativamente, a `nsec` precisa ser decifrada na memória RAM do computador hospedeiro para que a assinatura Schnorr seja calculada pela CPU.

Durante esses milissegundos, a chave existe em texto claro. Um *malware* sofisticado com privilégios de administrador poderia, teoricamente, monitorar a memória e capturar a chave nesse instante exato [2].

A arquitetura mitiga esse risco através da **minimização da janela de exposição**. Utilizamos o padrão `Zeroize` para sobrescrever a memória com zeros imediatamente após o uso, e a abstração do `YubikeyKeyManager` assegura que a chave nunca seja carregada em variáveis globais ou persistentes. Embora não elimine a vulnerabilidade (o que exigiria um *Secure Enclave* ou novo hardware), essa abordagem reduz a superfície de ataque de "tempo indefinido" (chave no disco) para "milissegundos" (durante a assinatura), tornando a exploração significativamente mais difícil.

Capítulo 6

Conclusão

A jornada percorrida neste trabalho partiu de uma premissa fundamental: a soberania digital prometida pelo protocolo Nostr depende inteiramente da segurança da chave privada. Ao longo dos capítulos, demonstrou-se que o modelo atual de armazenamento em software é insuficiente frente às ameaças modernas. Em resposta, propusemos e implementamos uma arquitetura de segurança baseada em isolamento via hardware, utilizando a robustez dos *tokens* FIDO2. Esta seção final sintetiza as conquistas do projeto, o impacto da solução alcançada e o horizonte de possibilidades que se abre para pesquisas futuras.

6.1 Conclusão

A solução de Nostr Bunker baseada em hardware FIDO2 cumpriu seu objetivo central: isolar a chave privada (`nsec`) do ambiente hostil do sistema operacional. A validação experimental demonstrou a viabilidade de realizar assinaturas criptográficas válidas sem expor a chave ao armazenamento persistente (disco rígido). O protótipo operou com sucesso em um cenário real de rede, condicionando cada publicação à tríade de segurança: posse do *token*, conhecimento do PIN e presença física (toque).

Sob a ótica da segurança, a análise qualitativa indicou que a arquitetura mitiga eficazmente vetores de ataque críticos, como o roubo de chaves em repouso por *malware* e a extração não autorizada. O risco residual de *RAM scraping*, inerente à limitação matemática das curvas elípticas no hardware atual, foi tratado pela estratégia de decifragem *just-in-time* e limpeza imediata de memória (*zeroize*). Embora não se trate de uma solução de "segurança absoluta" — conceito inexistente em computação — a abordagem eleva o custo do ataque para um patamar proibitivo para a grande maioria dos adversários.

Talvez a contribuição mais significativa deste trabalho seja a demonstração de que segurança robusta não exige o sacrifício da usabilidade. Ao implementar estritamente

a especificação **NIP-46**, a solução assegura interoperabilidade imediata: **clientes Nostr compatíveis com o padrão podem passar a usar segurança de hardware instantaneamente**, sem necessidade de adaptação. O projeto entrega, portanto, não apenas um artefato acadêmico, mas uma ferramenta prática que devolve ao usuário o controle efetivo e seguro de sua identidade digital.

6.2 Trabalhos Futuros

O desenvolvimento deste protótipo funcional abriu caminhos claros para a evolução da ferramenta rumo a um produto final de larga escala. As principais oportunidades identificadas são:

- **Interface Gráfica (GUI) e UX:** A barreira de entrada da linha de comando deve ser removida. O desenvolvimento de uma interface gráfica amigável para a configuração inicial e para os *prompts* de confirmação de assinatura é o passo natural para democratizar o acesso a essa tecnologia para usuários não técnicos.
- **Portabilidade e Mobile via NFC:** A dependência do desktop pode ser superada explorando a interface NFC (*Near Field Communication*) presente em *tokens* como a YubiKey. Portar a lógica do Bunker para dispositivos móveis permitiria que o usuário carregasse sua "assinadora segura" no bolso, interagindo com o protocolo Nostr diretamente do *smartphone* com a mesma segurança do ambiente *desktop*.
- **Otimização de Armazenamento:** A limitação de espaço do `largeBlob` pode ser mitigada com formatos de serialização binária mais eficientes que o Base64 atual. Além disso, investigar o suporte a múltiplos *slots* ou hardware com maior capacidade permitiria o gerenciamento de dezenas de identidades simultâneas.
- **Expansão para Outras Raízes de Confiança em Hardware:** A lógica de "envelope criptográfico" desenvolvida aqui é agnóstica ao dispositivo. Pesquisas futuras podem adaptar essa arquitetura para utilizar TPMs (*Trusted Platform Modules*) já presentes em *laptops* modernos ou Enclaves Seguros (como SGX), oferecendo segurança de hardware sem a necessidade de adquirir um *token* externo.

Essas direções apontam para um futuro onde a custódia de chaves deixa de ser um fardo cognitivo para o usuário e se torna uma garantia transparente da infraestrutura, consolidando o Nostr como uma plataforma verdadeiramente segura para a comunicação livre.

Referências

[1] Nostr Protocol Contributors: *Nostr: A decentralized network based on cryptographic keypairs*. `https://nostr.com`, 2023. Acessado em: 20 nov. 2025. 1, 6, 14, 26

[2] Salin, Hannes e Dennis Fokin: *Mission impossible: Securing master keys*. arXiv. Disponível em: `https://arxiv.org/abs/2107.00580`, 2021. Acesso em: 20 nov. 2025. 2, 9, 30

[3] García, Cesar Pereida: *Certified side channels*. Em *29th USENIX Security Symposium (USENIX Security 20)*, 2020. Acesso em: 20 nov. 2025. 2

[4] Proofpoint Threat Insight Team: *New kpot v2.0 stealer brings zero persistence and in-memory features to silently steal credentials*. Relatório Técnico, Proofpoint, May 2019. Acesso em: 20 nov. 2025. 2, 9

[5] Intezer Research Team: *Operation electrorat: Attacker creates fake companies to drain your crypto wallets*. Relatório Técnico, Intezer, January 2021. Acesso em: 20 nov. 2025. 2, 9, 29

[6] Zhou, Pengcheng, Yinglun Feng e Zhongliang Yang: *Privacy-aware rag: Secure and isolated knowledge retrieval*. arXiv. Disponível em: `https://arxiv.org/abs/2503.15548`, 2025. Acesso em: 20 nov. 2025. 2

[7] Naik, Neha e Paul Jenkins: *Digital identity architectures: comparing goals and vulnerabilities*. arXiv, 2023. Análise comparativa formal entre arquiteturas de identidade. Disponível em: `https://arxiv.org/pdf/2302.09988.pdf`. Acesso em: 20 nov. 2025. 5

[8] Nostr Protocol Contributors: *Nostr - notes and other stuff transmitted by relays*. `https://github.com/nostr-protocol/nostr`, 2024. Repositório oficial e especificação do protocolo. Acessado em: 20 nov. 2025. 6

[9] Clehaxze: *Nostr, my thoughts on a new decentralized pubsub protocol*. `https://clehaxze.tw/gemlog/2025/03-26-nostr-my-new-thoughts-on-decentralized-pubsub-protocol.gmi`, March 2025. Análise técnica do design pubsub. Acessado em: 20 nov. 2025. 6

[10] Nostr Protocol Contributors: *Nip-01: Basic protocol flow*. `https://github.com/nostr-protocol/nips/blob/master/01.md`, 2024. Definição da estrutura de Eventos. Acesso em: 20 nov. 2025. 7

[11] Certicom Research: *Sec 2: Recommended elliptic curve domain parameters*. Relatório Técnico, Standards for Efficient Cryptography Group (SECG), 2010. Define a curva secp256k1. Acesso em: 20 nov. 2025. 7, 12

[12] Wuille, Pieter, Jonas Nick e Tim Ruffing: *Bip 340: Schnorr signatures for secp256k1*. `https://github.com/bitcoin/bips/blob/master/bip-0340.mediawiki`, 2020. Padronização das assinaturas Schnorr. Acesso em: 20 nov. 2025. 8, 18, 20, 27

[13] Certicom Research: *The elliptic curve discrete logarithm problem*. `https://www.certicom.com/content/certicom/en/52-the-elliptic-curve-discrete-logarithm-problem.html`, 2024. Acesso em: 20 nov. 2025. 9

[14] Fleischhacker, Nils e Dominique Schröder: *On tight security proofs for schnorr signatures*. `https://www.dominique-schroeder.de/publications/2019_On_Tight_Security_Proofs_forSchnorr_Signatures.pdf`, 2014. Acesso em: 20 nov. 2025. 9, 18

[15] Damus Contributors: *damus-io/damus: ios nostr client*. `https://github.com/damus-io/damus`, 2024. Cliente iOS. Acesso em: 20 nov. 2025. 9, 14

[16] Primal Contributors: *Primalhq/primal-web-app: Primal web client*. `https://github.com/PrimalHQ/primal-web-app`, 2024. Acesso em: 20 nov. 2025. 9

[17] Primal Contributors: *Primalhq/primal-ios-app: Primal ios client*. `https://github.com/PrimalHQ/primal-ios-app`, 2024. Acesso em: 20 nov. 2025. 9

[18] Cashu Contributors: *cashubtc/nutshell: Cashu ecash wallet and mint*. `https://github.com/cashubtc/nutshell`, 2024. Acesso em: 20 nov. 2025. 9

[19] Nostr Protocol Contributors: *Nip-57: Lightning zaps*. `https://github.com/nostr-protocol/nips/blob/master/57.md`, 2024. Especificação de Zaps. Acesso em: 20 nov. 2025. 9

[20] Nostr Protocol Contributors: *Nip-46: Nostr remote signing*. `https://github.com/nostr-protocol/nips/blob/master/46.md`, 2024. Especificação do Nostr Bunker. Acesso em: 20 nov. 2025. 9, 11, 14, 16, 20, 24, 25, 27, 29

[21] Barker, Elaine: *Recommendation for key management: Part 1 - general*. Relatório Técnico SP 800-57 Part 1 Rev. 5, National Institute of Standards and Technology, May 2020. Guia definitivo de proteção e ciclo de vida de chaves. Acesso em: 20 nov. 2025. 12, 13, 18, 20, 21, 22, 23, 29

[22] FIDO Alliance: *Client to authenticator protocol (ctap)*. Proposed standard, January 2019. Especificação oficial FIDO2. Acesso em: 20 nov. 2025. 12, 15, 18, 20, 22, 27

[23] FIDO Alliance: *Client to authenticator protocol (ctap) v2.2*. Relatório Técnico, July 2025. Especificação atualizada. Acesso em: 20 nov. 2025. 12

[24] Cook, John D: *A tale of two elliptic curves: secp256k1 vs secp256r1 (nist p-256)*. https://www.johndcook.com/blog/2018/08/21/a-tale-of-two-elliptic-curves/, 2018. Comparativo de curvas elípticas. Acesso em: 20 nov. 2025. 12

[25] Yubico Documentation: *Ctap2.1 feature reference: hmac-secret and largeblob extensions*. Relatório Técnico, 2022. Detalhes técnicos das extensões. Acesso em: 20 nov. 2025. 13, 15, 19, 20, 21, 22, 23, 29

[26] Corbado Team: *Passkeys & webauthn prf for end-to-end encryption*. https://www.corbado.com/blog/passkeys-prf-webauthn, 2025. Explicação sobre derivação segura via PRF. Acesso em: 20 nov. 2025. 13, 15

[27] MITRE CVE/CWE: *Cwe-312: Cleartext storage of sensitive information*. https://cwe.mitre.org/data/definitions/312.html, 2024. Vulnerabilidade crítica de armazenamento em texto plano. Acesso em: 20 nov. 2025. 13, 15

[28] NostrConnect Contributors: *Get started with nostr connect - implementation guide*. https://nostrconnect.org, 2024. Guia de implementação. Acesso em: 20 nov. 2025. 16, 20, 25, 30