CipherBrick Pro

A Technical White Paper on Browser-Native, Zero-Trust Encrypted Messaging

Cliff Cazes
cliff@cipherbrick.com | cipherbrick.com
March 2026

1. Abstract

CipherBrick Pro is a browser-based application that solves an overlooked vulnerability in modern secure communications: the unintended recipient. Instead of being a messaging platform with end-to-end encryption (E2EE), CipherBrick Pro encrypts locally on the user's device and produces a ciphertext "payload" that can be shared through virtually any channel: a messaging app, email, QR code, or even an audio signal played speaker-to-microphone. E2EE solutions protect messages in transit, but once decrypted on the receiving end, the message is only as secure as the recipient's device and attention. A single misdirected message can expose financial data, personal identifying information, or confidential business communications, regardless of how strong the transport encryption was. CipherBrick Pro takes a fundamentally different approach: messages are encrypted locally before they are ever sent, making the transport layer irrelevant. Under the hood, the app uses AES-256-GCM authenticated encryption through the browser's built-in Web Crypto API, and supports two core credential paradigms: a shared Key+Salt model (Standard mode) with keys derived via PBKDF2, and a hardware key model (HKPM) where decryption requires a FIDO2 authenticator to be physically present. An intercepted or misdirected message remains unreadable without the correct credentials. The app is offline-capable after loading, requires no account, and does not transmit data to any server. This white paper details the design philosophy, cryptographic architecture, threat model, and security guarantees that underpin CipherBrick Pro.

---

2. The Problem: Digital Privacy in an Era of Surveillance

I originally began building CipherBrick as a fun side project after coming across a widely reported news story in which a secure messaging platform was implicated in the accidental exposure of confidential communications due to user error. Only being vaguely familiar with the app in question, I opted to look into it more and learned that it is a secure messaging app that

utilizes end-to-end encryption (E2EE) along with stringent security protocols to provide users with a private way to communicate. However, the news story revealed that once again, the weakest link in the chain was the end user. While the transport layer and the app itself are very secure, the issue in this particular instance was that users improperly added recipients to their messages, resulting in confidential information reaching the wrong people. Overall, transparency with government is paramount as it fosters a level of trust, however, classified and confidential information being leaked can be dangerous. Even though we rely on people to follow protocols, mistakes happen. That reality is what set me down this path.

Apps like Signal approach secure communication by taking input from the sender in the form of clear text, converting it into encrypted data packets, and sending those packets via a secure tunnel through the service provider's system. The packets are then delivered to the intended recipients where the application decrypts them and presents the message in clear text once again. All in all, this is a secure way to communicate if your intent is to protect messages while they are in transit. This method prevents interception by unwanted parties and only reveals the message once it arrives at its destination. However, the encryption ends upon delivery. Once the message reaches the end user, the application decrypts it and the content is readable by anyone with access to that device.

As I thought about this, I mentally compared it to a VPN tunnel which is essentially the same concept. Data is encrypted while in transit, and once it reaches its destination, it is decrypted and readily available to read. While this is a solid solution for protecting messages in motion, one problem that is easily overlooked is: what happens when the recipient is the unintended target? A misdirected message that is sent to the wrong person by mistake bypasses all of that transport security entirely. Overall, the weakest link isn't "the math," it's the moment a human misclicks. Identifying this as a real problem, I began to brainstorm potential solutions.

What if you were sending sensitive information and accidentally sent it to the wrong recipient? How could you protect that information even if you made that mistake? As I thought through various scenarios, the consequences became clear. Sending financial details, account credentials, or personal identifying information to the wrong recipient could result in compromised accounts, identity theft, or stolen finances. In a business context, a misdirected message containing confidential information could have catastrophic consequences. Compromised devices add yet another dimension to the problem — information sent in confidence could be captured via a screenshot the moment it arrives. End-to-end encryption, as robust as it is, simply does not address these scenarios.

So, with this in mind, I began contemplating other ways to achieve secure communications, which got me thinking about past novels I had read and shows I had seen. Rather than

establishing a secure connection between two or more parties, what if you took the approach of encrypting the message itself before sending, bypassing the need for a secure transport layer entirely? It brought me back to reading Dan Brown's Digital Fortress as a teenager, in which ciphers and encoding techniques were used to protect messages. I had been fascinated by that and had even written a simple ASCII replacement script in C++ to experiment with basic message encryption at the time. More recently, shows like Death and Other Details have referenced similar techniques: ciphers and character substitution schemes used to protect sensitive notes. Along those same lines, I began thinking that a different approach, not necessarily better but fundamentally different, would be to encrypt the message before sending and then become completely agnostic about the transport layer. The message could be sent via text, a modern chat platform, or even published in a magazine, it would not matter, because only the intended recipient with the correct key could ever read it. End-to-end encryption is most valuable for live, real-time transmissions; for static messages, encrypting the content itself is a more resilient model. With that line of thinking, I began prototyping a solution using audited, browser-native cryptographic primitives, which eventually led to the development of CipherBrick Pro: a tool built on the premise that true message security begins before the send button is ever pressed.

A Shifting Landscape

What began as a personal side project has arrived at a moment when the industry's relationship with E2EE is under increasing strain. In early 2026, several converging developments highlight a broader trend away from platform-level encryption and toward content inspection regimes:

Meta confirmed it will discontinue end-to-end encrypted Instagram DMs starting May 8, 2026, citing low adoption and redirecting users to other products for encrypted messaging [1]. Apple announced it can no longer offer its optional iCloud Advanced Data Protection feature to new UK users following government pressure, though some iCloud categories and iMessage remain end-to-end encrypted by default [2]. In the EU, the ongoing "Chat Control" debate continues to advance proposals for mandatory message scanning [3]; civil-society groups argue that client-side scanning, where content is inspected before encryption, functionally transforms a private-by-design architecture into an inspect-by-default one, undermining the trust model that E2EE is built on [4].

This is precisely where CipherBrick Pro's transport-agnostic model becomes strategically relevant. If a platform downgrades its encryption, introduces scanning, or changes its privacy policy, none of that affects a CipherBrick payload, because the encryption happens entirely

outside the platform's control, before the message ever touches it. The ciphertext remains ciphertext regardless of where it travels or what policies govern the channel it passes through.

---

3. Design Philosophy

CipherBrick Pro's design philosophy can be summarized as: assume the network is hostile, assume platforms change, assume humans sometimes make mistakes… and build accordingly. When planning the build, I settled on the principle of "zero trust": don't ask the transport layer to be trustworthy; don't save any information outside of temporary variables; wipe the slate clean for each use; build in safety controls that automatically clear forms, fields, and session variables to ensure no data lingers. This way, only the sender and intended recipients are ever able to decipher the messages exchanged between them.

CipherBrick Pro is "minimum retention" by design. No encryption key or salt is written to disk; plaintext and decrypted output are held in memory only and cleared on reset or timeout; hardware-key private material is not extractable and is derived entirely in memory. The reasoning is straightforward: data that is never stored cannot be stolen, subpoenaed, or leaked. Every timeout, auto-clear, and session boundary in the app exists to enforce this principle, not as an afterthought, but as a core design constraint.

CipherBrick Pro is "real-world usable," not crypto-theory cosplay. Strong encryption is only useful if people can actually use it. Since the application does not store anything, the responsibility of managing keys and passphrases falls on the user. Parties can exchange credentials through any method they trust, but for scenarios where doing so openly could be compromising, an integrated key exchange wizard helps two people arrive at matching credentials without saying them out loud or typing them into the same channel they are trying to secure. The app supports multiple out-of-band transfer methods (QR code and audio transmission) so that bootstrapping a private conversation is practical even in situations where a secondary secure channel is not available.

CipherBrick Pro is local-first by default. All encryption, decryption, key generation, and key exchange operations are performed on the user's device. No network requests are made during normal operation, which reduces the attack surface for anyone attempting to intercept communications. The application can be self-hosted on a web server and is installable on mobile devices as a Progressive Web App (PWA); a native mobile release for major platforms is also planned. Once installed, the app requires no network access whatsoever. By design, message encryption is never processed through any external system. Your data stays on your

device.

---

4. Cryptographic Architecture

CipherBrick Pro's cryptographic design deliberately sticks to mainstream, widely analyzed primitives and browser-native implementations.

Authenticated Encryption with AES-256-GCM

CipherBrick Pro leverages AES-256-GCM via the browser's Web Crypto API to provide modern, standards-based encryption. AES-256 serves as the core symmetric encryption algorithm, while Galois/Counter Mode (GCM) enhances it with built-in authentication. This combination ensures not only confidentiality, but also integrity and tamper detection.

Unlike traditional encryption modes that only obscure data, AES-GCM produces an authentication tag that verifies the encrypted payload has not been altered. If any modification occurs, intentional or otherwise, decryption fails entirely. As a result, it guarantees that decrypted messages are both authentic and intact.

By utilizing the Web Crypto API, CipherBrick relies on native, hardware-accelerated cryptographic implementations rather than custom or browser-based code, ensuring both performance and security. The app was purposefully built to use strong encryption, not to invent or create new encryption. The result is a system that delivers authenticated encryption (AEAD) in a lightweight, platform-agnostic environment without compromising cryptographic rigor.

CipherBrick shifts trust away from platforms and into mathematics. The system does not rely on transport security, institutional guarantees, or centralized key management. It only relies on the correctness of the cryptographic primitives and the discipline of the user.

Key Derivation: From User Key to Encryption Key

CipherBrick Pro supports multiple methods for generating the encryption key used by AES-256-GCM, depending on the selected mode. Rather than relying on a single mechanism, the system allows different trust models to be applied based on the user's needs.

In Standard mode, the user provides both a Key and a Salt. These inputs are combined using a password-based key derivation function (PBKDF2) to produce a 256-bit encryption key. The

inclusion of a Salt ensures that identical Keys do not produce identical outputs, while the derivation process increases the computational cost of guessing attacks. This approach aligns with established guidance for password-based cryptography [6].

Simple mode follows the same PBKDF2 derivation path, but rather than requiring the user to provide a Salt, the app generates a random one automatically at the time of encryption. The recipient does not need to know or enter the Salt separately, because it travels with the payload. This reduces the number of things a user has to manage and share, making the mode more accessible while still preserving the cryptographic benefit of a unique Salt per message.

Rather than simply appending the Salt to the beginning or end of the ciphertext, where it would be trivially identifiable to anyone inspecting the output, CipherBrick weaves the Salt character-by-character throughout the base64-encoded ciphertext string using a fixed interleaving pattern. The result is a single string that looks uniform from the outside, with no obvious markers indicating where the Salt begins or ends. On decryption, the app applies the same pattern in reverse to extract the Salt and reconstruct the clean ciphertext before proceeding. This provides a layer of obfuscation that makes casual inspection or manual extraction of the Salt significantly harder, without adding complexity for the user.

The trade-off is that the security of the encrypted message relies entirely on the strength and secrecy of the user-provided Key, since the Salt is carried within the payload rather than kept separately.

HKPM (Hardware Key Personal Message) mode represents a fundamentally different model. Rather than deriving an encryption key from a passphrase, HKPM uses a FIDO2 authenticator to generate a stable, hardware-backed cryptographic key pair. The authenticator derives a deterministic P-256 elliptic curve key pair using the FIDO2 PRF (Pseudo-Random Function) extension, meaning the same key pair is produced every time the same authenticator responds to a challenge. No private key material ever leaves the hardware [10, 11].

This key pair is what enables HKPM's one-to-one messaging model. Before exchanging encrypted messages, both parties share their public keys with each other, which CipherBrick facilitates through its key exchange flow. When encrypting, the sender uses the recipient's public key along with their own private key to derive a shared secret via ECDH (Elliptic Curve Diffie-Hellman). The sender's public key is embedded directly in the encrypted payload so the recipient can complete the same derivation in reverse, without any prior setup on the receiving end beyond possessing their own hardware key.

The result is a self-contained payload: anyone can receive it, but only the holder of the correct hardware key can decrypt it. This is the most secure mode CipherBrick offers. A message

encrypted for a specific recipient can only be decrypted by that recipient. Not by anyone intercepting the payload, and not even by the sender, because while decryption requires both a hardware key and the other party's public key, the recipient's public key is used during encryption but is never written into the payload. Even with their own hardware key in hand, the sender cannot reverse the encryption from the payload string alone. When the recipient replies, the app pre-fills the sender's public key extracted from the original payload. The reply is encrypted the same way in reverse, and the original sender decrypts it simply by presenting their hardware key. HKPM is compatible with both dedicated hardware security keys and mobile platform authenticators (such as passkeys backed by Touch ID or Face ID), as long as the device supports the FIDO2 PRF extension.

Across all modes, CipherBrick does not store, transmit, or recover encryption keys. In Standard and Simple modes, security depends on the strength and secrecy of the user's chosen Key, as well as disciplined handling practices, particularly the separation of the Key from the encrypted message during transmission. In HKPM mode, security depends on maintaining control over the hardware or external key source, as loss of that control may result in permanent loss of access to encrypted data.

IV/Nonce Handling

One of the subtler but critical aspects of AES-GCM is correct handling of the initialization vector, or IV (sometimes called a nonce). The IV is a unique value that must be different for every encryption operation performed with the same key. Reusing an IV with the same key is a serious cryptographic mistake that can undermine the security of the entire scheme, regardless of how strong the key is, as detailed in NIST SP 800-38D [5].

CipherBrick Pro handles this correctly by generating a fresh, cryptographically random 96-bit IV for every single encryption operation, using the browser's built-in random number generator. Think of it like a unique serial number that is stamped on every message at the moment it is locked. Even if you encrypt the exact same message twice with the exact same key, the serial number is different each time, which means the resulting ciphertext looks completely different. This prevents anyone observing the output from detecting patterns, even across repeated messages.

The IV is not secret. It is prepended to the encrypted payload and travels alongside the ciphertext. When the recipient decrypts the message, the app reads the IV from the front of the payload and uses it to unlock the message correctly. The authentication tag generated by AES-GCM is validated automatically during decryption, so any tampering with the payload, including the IV itself, causes decryption to fail entirely.

Browser-Native Primitives and Avoided Pitfalls

CipherBrick Pro uses only the browser's built-in Web Crypto API for all cryptographic operations. No third-party cryptographic libraries are used, and no custom cryptographic logic was written. This is a deliberate and important decision.

The history of software security is littered with applications that attempted to implement their own encryption and got it wrong in ways that were not obvious at the time. The Web Crypto API [7] exists precisely to give developers access to well-vetted, hardware-accelerated implementations without requiring them to be cryptographers. By relying on it exclusively, CipherBrick sidesteps an entire category of risk. The goal was never to invent new cryptography; it was to apply established, audited cryptography correctly.

The Web Crypto API is actively maintained and patched by the teams at Google, Mozilla, and Apple. Any vulnerability discovered in the implementation gets fixed across all users automatically through browser updates, requiring no action from the app itself. Third-party cryptographic libraries, by contrast, introduce dependency risk: a compromised or outdated package can silently replace audited code with something less trustworthy. With no external cryptographic dependencies, CipherBrick has no supply chain surface to attack. Paired with the zero-trust design philosophy, in which no keys, messages, or credentials are ever written to disk and working data exists only in temporary session variables, the result is a system with virtually no persistent attack surface.

---

5. Threat Model and Limitations

A security tool is only honest if it states clearly, upfront, what it does not protect against. CipherBrick Pro is no exception.

CipherBrick Pro meaningfully protects against transport interception. Ciphertext can be transmitted over any channel and will remain unreadable without the correct credentials. Even if a platform downgrades its messaging transport layer [1], CipherBrick Pro's payload remains encrypted because the encryption occurs outside the platform entirely. In the event of a server or provider being compromised, the ciphertext is still secure. The app makes no outbound calls, no network requests, and pulls no external information. It runs offline.

On the other side, CipherBrick Pro does not protect against compromised endpoints. If malware, keyloggers, hostile browser extensions, or an OS-level compromise exists on the device doing the encrypting or decrypting, local encryption will not hold up. This is true of all

client-side encryption. Another persistent weakness in information security is the human element. User key handling failures, in which credentials are shared improperly or insecurely, can result in compromised message security. Weak or easily guessable keys undermine strong encryption regardless of the algorithm used. As CipherBrick does not store credentials on behalf of the user, the responsibility falls on the user to store their sensitive information securely, whether that be a key, a key and salt pair, or a public/private key pair. If that information is stored or shared carelessly, it creates an opening for bad actors to compromise message security regardless of how strong the underlying encryption is. Ultimately, encryption alone cannot protect against operational mistakes. Finally, CipherBrick Pro protects content, not the fact that communication occurred. Providers at the transport layer may still be able to determine who communicated with whom, when it occurred, and possibly how often [4].

Each mode carries its own security considerations. In Standard mode, the salt is user-controlled and kept separate from the payload, which is the strongest posture among the passphrase-based modes. The trade-off is that both the key and the salt must be securely shared with the recipient. In Simple mode, the salt is auto-generated and embedded in the payload, reducing what the recipient needs to know. However, because the salt is not a separate secret, the full security burden falls on the strength of the key alone. HKPM is the most secure mode but introduces its own constraints: it requires physical possession of compatible hardware at the time of decryption, is scoped to one-to-one messaging between parties who have exchanged public keys, and depends on FIDO2 PRF extension support, which is not universally available across all browsers and devices.

CipherBrick is not "better cryptography" than E2EE; it's a different trust boundary. It reduces dependency on any single platform's encryption promises by externalizing the encryption to a user-controlled layer. In a world where platforms and regulations can change, that separation is the strategic advantage.

---

6. Transmission Modes

Text/Clipboard

Text and clipboard copy/paste is the simplest and most universally compatible transmission method in CipherBrick Pro. It is the default and recommended approach, particularly because encrypted payload strings are often long, lack spaces, and would be impractical to transcribe manually. Since security is a core concern, the app has built-in functionality to clear clipboard contents and session data automatically. By default, any text copied using the app's copy

buttons is cleared from the clipboard after 30 seconds, and session inactivity for more than 5 minutes will result in session variables, forms, and the clipboard being cleared. This is done with the zero-trust philosophy in mind, to limit attack surfaces and prevent unwanted access to sensitive information. Both timers are user-configurable.

QR Code

Recognizing that the ciphertext is long and impractical to transcribe manually, the application provides the option to share encrypted strings via QR codes. CipherBrick Pro can both generate and decode QR codes created by the app. Messages are limited to 500 characters to prevent QR code density issues that would make codes difficult or impossible to scan. For convenience, when on the Decrypt tab, the user has the option to scan a QR code, which will autofill the encrypted string input field with the encoded ciphertext.

Audio Transmission

Audio transmission is a distinctive feature that uses the ggwave library [8] to convert a text string into audio tones that can be received by a recipient's microphone and decoded. This requires both parties to be within reasonable proximity of each other for the sound to transmit and be received clearly. If the audio is successfully decoded, the encrypted string field on the Decrypt tab is populated automatically and the user can proceed from there. The ggwave repository is an open-source library that enables communication of small amounts of data between air-gapped devices using sound, without requiring any network connection between them.

Limitations

Each transmission method has its practical constraints, and message length has a direct impact on all of them. The longer the payload, the more challenging QR code decoding and audio transmission become. For very long payloads, copying and pasting the payload string directly is often the most reliable option.

For QR codes specifically, increasing density as message length grows can make codes harder to scan, particularly with lower-quality cameras. Recipients who receive a QR code as an image in a message or email may also find it impossible to scan on the same device the image is displayed on. In those cases, the recipient can paste the ciphertext directly into the app's Payload String input, or use a third-party QR code reader to extract the text before pasting it into the Decrypt tab.

For audio transmission, effectiveness is affected by distance between devices, background noise levels, and the quality of the speaker and microphone on each device. Microphone permissions must also be granted for the feature to function, which on some platforms requires an explicit user approval step.

---

7. Key Exchange

Secure communication with CipherBrick Pro begins before the first message is sent. In Standard mode, both parties must agree on a shared key and salt, and ideally that exchange happens through a different channel than the one being secured. In practice, however, a separate secure channel is not always available. The Key Exchange Wizard solves this bootstrapping problem: it allows two parties to arrive at matching credentials over the same channel they plan to use, without ever transmitting those credentials in plain text.

The wizard uses ECDH (Elliptic Curve Diffie-Hellman) to let two parties derive a shared key and salt independently. Each party generates an ephemeral key pair and exchanges only their public key. Through the mathematical properties of ECDH, both sides arrive at the same shared secret without either party ever sending it directly. That shared secret is then used to deterministically derive the Key and Salt used for AES-256-GCM encryption. NIST SP 800-56A describes approved key-establishment schemes using discrete logarithm cryptography, including elliptic-curve-based Diffie-Hellman variants [9].

The wizard is explicit about operational security: generated key pairs can be downloaded for reuse, but the downloaded file contains the private key and must be protected accordingly, since possession of that file enables impersonation during the key exchange. Once both parties have successfully agreed on a key and salt, the key pair used for the exchange is no longer needed. Ideally, both parties would delete their key pairs once it is confirmed that they both have the key and salt.

Enabling users to save and import key files means the exchange does not need to happen in real time. Regardless of the channel used, as long as each person safely manages their private key, both parties can obtain the agreed-upon key and salt at their convenience, whether they are the sender or the recipient. From that point forward, they can use Standard mode directly with the shared credentials.

CipherBrick Pro also supports HKKE (Hardware Key Key Exchange) mode, an alternative wizard flow that uses a hardware key identity instead of an ephemeral software key pair. Rather than requiring both parties to paste public keys manually, HKKE produces a CBHKX1

payload containing encrypted credentials that the recipient can unwrap simply by using their hardware key.

---

8. Privacy & Data Handling

CipherBrick Pro documents exactly what it stores and what it never stores.

Stored: user preferences (timer values, language, mode selection, audio protocol) and hardware key state (credential ID and PRF capability flag) in localStorage; key-exchange wizard session keys in sessionStorage (cleared on tab close, session timeout, or one-hour expiry).

Never stored: encryption key and salt, plaintext messages, decrypted output, or hardware-key private material (which is non-extractable by design).

That distinction has a direct security consequence: there is nothing to breach. No database of keys, no history of messages, no credentials at rest. An attacker who compromises the server gets a static web application and nothing else.

The session model is intentionally aggressive. By default, five minutes of inactivity clears all sensitive fields, wipes sessionStorage, and attempts to clear the system clipboard, reducing the window in which a lost or unattended device exposes anything useful. The clipboard is also independently cleared 30 seconds after a copy operation, addressing the very human habit of copying a sensitive value and forgetting it exists. This is the right kind of paranoia: not theatrical, but practical. It acknowledges that modern threats include not only active attackers but also lost devices, shoulder-surfing, and inattention.

CipherBrick Pro is installable as a Progressive Web App (PWA) and works fully offline after installation, with no background sync, no analytics, and no network calls of any kind once loaded.

---

9. Conclusion

CipherBrick Pro is designed to provide strong, standards-based encryption in a form that is portable, flexible, and independent of any specific platform or service. By leveraging AES-256-GCM for authenticated encryption and well-established key derivation practices, it ensures that messages remain confidential and tamper-evident when handled correctly.

Unlike traditional secure messaging systems, CipherBrick does not rely on persistent identities, centralized infrastructure, or managed key exchange. Instead, it places control directly in the hands of the user, allowing encrypted data to be transmitted through any medium while remaining protected.

This model makes CipherBrick particularly well-suited for individuals and organizations that need to securely handle sensitive information outside of controlled environments. This includes use cases such as sharing credentials, transmitting financial or legal data, protecting personal communications over untrusted channels, or operating in scenarios where offline or air-gapped communication is required.

At the same time, this flexibility comes with responsibility. CipherBrick does not manage or recover keys, and it does not enforce secure key exchange. In passphrase-based modes, its guarantees depend on the strength of the chosen key material and the discipline with which those secrets are handled. In HKPM mode, they depend on the security and physical possession of the hardware key.

CipherBrick Pro is therefore best understood not as a replacement for existing secure communication platforms, but as a complementary tool: one that secures the message itself, independent of how or where it is transmitted.

References

[1] "Instagram to remove end-to-end encryption for private messages in May," The Guardian, March 18, 2026.
https://www.theguardian.com/technology/2026/mar/18/instagram-to-remove-end-to-end-encryption-for-private-messages-in-may

[2] "Advanced Data Protection for iCloud," Apple Support (UK).
https://support.apple.com/en-gb/122234

[3] European Parliament Press Release, March 6, 2026 — Extension of voluntary detection derogation.
https://www.europarl.europa.eu/pdfs/news/expert/2026/3/press_release/20260306IPR37531/20260306IPR37531_en.pdf

[4] "Why Client-Side Scanning Is a Lose-Lose Proposition," Access Now.
https://www.accessnow.org/why-client-side-scanning-is-lose-lose-proposition/

[5] NIST SP 800-38D — Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.
https://csrc.nist.gov/pubs/sp/800/38/d/final

[6] NIST SP 800-132 — Recommendation for Password-Based Key Derivation.
https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-132.pdf

[7] W3C Web Cryptography API, Level 2.
https://www.w3.org/TR/webcrypto-2/

[8] ggwave — Data-over-Sound Library (ggerganov).
https://github.com/ggerganov/ggwave

[9] NIST SP 800-56A Rev. 3 — Recommendation for Pair-Wise Key-Establishment Schemes Using Discrete Logarithm Cryptography.
https://csrc.nist.gov/pubs/sp/800/56/a/r3/final

[10] W3C Web Authentication Level 3 — Web Authentication: An API for accessing Public Key Credentials.
https://www.w3.org/TR/webauthn-3/

[11] Yubico Developer Docs — WebAuthn PRF Extension.
https://developers.yubico.com/WebAuthn/Concepts/PRF_Extension/index.html