Seminar Preparation: Developer-Side Cryptography API Usability

Focus: End-to-End Encrypted Messaging (E2EE)

Lorenzo Deflorian

January 28, 2026

1 Technology Selection

1.1 What it is

End-to-end encrypted messaging (E2EE) is a communication protocol where messages are encrypted on the sender's device and decrypted only on the recipient's device. Intermediaries such as servers, ISPs, or platform providers cannot access the plaintext content.

1.2 What it is used for

E2EE is used for secure private conversations (1:1 or group chat), confidential business communications, sensitive healthcare/legal information sharing, and protection for activists and journalists in censored or high-risk environments.

1.3 How it works

Each user has a public/private key pair. When User A sends a message to User B:

1. The app retrieves B's public key from a server or key directory.
2. The app uses B's public key to derive a shared secret and encrypts the message with a symmetric cipher (e.g., AES-GCM or ChaCha20-Poly1305).
3. The ciphertext is sent to the messaging server.
4. User B's device receives the ciphertext and decrypts it using B's private key and the derived shared secret.

In practice, modern E2EE messaging (e.g., Signal/WhatsApp) uses more advanced protocols like X3DH and the Double Ratchet, usually implemented via crypto libraries such as libsodium, NaCl, or OpenSSL.

2 Papers Read (Developer Side)

1. **Green, Matthew, and Matthew Smith.** "Developers are Not the Enemy!: The Need for Usable Security APIs." IEEE Security & Privacy, 14(5), 2016.
2. **Patnaik, Nikhil, Joseph Hallett, and Awais Rashid.** "Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries." SOUPS 2019.

3 Main Insights from the Papers

3.1 Green & Smith (2016) – "Developers Are Not the Enemy"

3.1.1 Core thesis

Developers are not malicious or careless "enemies"; they are professionals whose main goal is to ship working software. Crypto APIs are often designed as if every caller were a cryptographer. When APIs are hard to understand or easy to misuse, the *API design* is at fault, not the developer.

3.1.2 Ten principles for usable crypto APIs

Green & Smith adapt general API design principles to the security/crypto context and propose that good crypto APIs should be:

1. **Abstract**: Integrate cryptography into standard, high-level APIs so most application developers never need to work with low-level crypto operations directly.
2. **Powerful**: Capable of satisfying both security requirements and non-security requirements (performance, interoperability, deployment constraints).
3. **Comprehensible**: Learnable without deep cryptographic expertise; documentation and interface should make intended use clear.
4. **Ergonomic**: Fit naturally into the programming language and frameworks developers already use; should not force unnatural patterns.
5. **Intuitive**: Usable correctly with minimal documentation; APIs should match developers' expectations and common mental models.
6. **Failing (hard to misuse)**: Make incorrect usage hard or impossible; where misuse is possible, failures should be explicit and visible rather than silently insecure.
7. **Safe (secure defaults)**: Provide safe defaults for algorithms, parameters, and modes. Avoid ambiguous options; insecure combinations should not be easily reachable.
8. **Testable**: Allow convenient testing, ideally including reduced-security testing modes that still preserve the structure of the real protocol.
9. **Readable**: Code written using the API should be easy to read and review; security reviewers must be able to see at a glance what is happening.
10. **Explained**: Provide good error messages and guidance for user interaction around security (prompts, warnings, trust dialogs).

3.1.3 Key arguments

- Many developers using crypto libraries are *not* cryptography experts and never will be.
- Traditional crypto APIs expose things like algorithm choices, key sizes, cipher modes, padding schemes, nonce/IV management, etc., directly to non-experts.
- Because crypto failures are often catastrophic and non-obvious, crypto APIs must be held to a higher usability standard than typical APIs.
- Shifting blame to "stupid developers" is counterproductive; security engineering must design for realistic developer capabilities.

3.2 Patnaik et al. (2019) – "Usability Smells"

3.2.1 Goal of the paper

Empirically investigate how developers actually struggle with cryptographic libraries in the wild, using Stack Overflow questions as a data source.

3.2.2 Method

- Collected 2,491 questions (2,317 relevant after filtering) about seven crypto libraries: OpenSSL, NaCl, libsodium, Bouncy Castle, SJCL, Crypto-JS, PyCrypto
- Used qualitative coding to categorize problems developers reported.
- Derived 16 usability issues which they grouped into 4 higher-level "usability smells."

3.2.3 Four "usability smells"

These smells summarize recurring patterns of difficulty that indicate deeper usability problems:

1. **Needs a Super Sleuth**: Developers must hunt through scarce or scattered information to understand how to use the API ("missing docs," "no examples"). Violates: *Comprehensible, Ergonomic, Intuitive, Explained*.
2. **Confusion Reigns**: Even with documentation, developers remain unsure about how or when to use certain functions, and how to set parameters. Violates: *Comprehensible, Intuitive, Ergonomic*.
3. **Needs a Post-Mortem**: When something goes wrong (crypto fails, decryption fails, signatures don't verify), it is very hard to understand why; error messages are poor or absent. Violates: *Failing, Safe, Readable, Intuitive, Ergonomic*.
4. **Doesn't Play Well With Others**: Practical problems integrating the library into real systems: build problems, linking errors, cross-platform issues, version incompatibilities, performance issues. Violates: *Powerful, Testable*.

3.2.4 Key conclusions

- All seven libraries exhibit all four usability smells; no library is "problem-free."
- Even "modern" and supposedly usable libraries like libsodium and NaCl still produce significant developer friction.
- The empirical evidence *supports* Green & Smith's principles but also highlights areas they under-emphasize: build and deployment processes, cross-platform compatibility, performance and practical integration constraints.

4 Implications for the Chosen Technology (E2EE Messaging)

Below are the main implications of the papers' insights specifically for implementing an end-to-end encrypted messaging application.

4.1 Documentation gaps leading to insecure key management in messaging apps

- **Issue from papers:** Missing or unclear documentation and examples are a major source of confusion.
- **Effect on E2EE apps:** Developers may not understand how to properly generate, store, and rotate keys. They may use weak randomness sources, store keys insecurely, or omit key rotation.
- **Implication:** E2EE messaging SDKs and libraries should provide *task-based* docs and examples (e.g., "register a user and generate keys," "send an encrypted message," "rotate device keys") rather than just low-level API references. Hide low-level key operations behind simpler high-level APIs.

4.2 Parameter confusion leading to weak or misconfigured encryption

- **Issue from papers:** Developers often ask which algorithm/mode/key size to use and how to set parameters like IVs/nonces.
- **Effect on E2EE apps:** Developers might choose insecure modes (AES-ECB), reuse IVs/nonces, or omit authentication. This can completely break confidentiality or integrity while "looking" encrypted.
- **Implication:** Libraries used for messaging should offer *single, high-level* encryption primitives that enforce good choices internally (e.g., AEAD only, strong defaults). Avoid exposing algorithm menus or easily misused building blocks where possible.

4.3 API misuse in key exchange and ratcheting leading to broken forward secrecy

- **Issue from papers:** Misuse of functions and difficulty understanding what went wrong.
- **Effect on E2EE apps:** Forward secrecy depends on correct use of key exchange protocols and ratcheting. Misuse can lead to key reuse across many messages, missing or broken ratcheting (so one compromised key reveals the whole conversation), or no verification of remote identity keys (enabling man-in-the-middle attacks).
- **Implication:** Messaging protocols should expose ready-made, high-level protocol implementations (e.g., "startSessionWithUser(userId)") rather than raw ECDH/ED25519 operations. Enforce correct ratcheting and key replacement automatically.

4.4 Lack of crypto literacy leading to developers disabling or weakening security features

- **Issue from papers:** Many questions show developers lack basic crypto concepts; they do not understand *why* certain design choices matter.
- **Effect on E2EE apps:** Developers may remove "complicated" steps like rekeying or signature checking, shorten key sizes or iteration counts "for performance," or skip integrity/authentication checks because "decryption already succeeded."
- **Implication:** Stronger conceptual documentation is needed: explain forward secrecy, authentication, integrity, and why they are essential. For messaging SDKs, provide short conceptual sections like "Threat model and why we use this protocol" alongside code.

4.5 Build and integration issues leading to outdated or inconsistent crypto across platforms

- **Issue from papers:** Large share of questions about build, linking, and platform compatibility (especially for OpenSSL and PyCrypto).
- **Effect on E2EE apps:** Messaging apps run on iOS, Android, desktop, and sometimes web. Build pain can lead to shipping old library versions with known vulnerabilities, dropping some platforms to avoid complexity, or inconsistent behavior and security guarantees between platforms.
- **Implication:** E2EE messaging stacks should ship with pre-built, well-maintained binaries for all target platforms, reduce or eliminate external build dependencies, and strictly manage versions to keep crypto libraries updated everywhere.

4.6 Cross-library mental model mismatch leading to subtle integration bugs

- **Issue from papers:** "Borrowed mental models" – developers assume one library behaves like another.
- **Effect on E2EE apps:** A stack that mixes libraries (e.g., OpenSSL for transport, libsodium for end-to-end layer) risks mismatched assumptions about parameter order and semantics, error codes and success conditions, encoding formats (DER, PEM, raw bytes).
- **Implication:** For E2EE messaging, prefer a single crypto library for the core protocol. Offer an app-facing API layer that hides underlying library details completely.

5 Summary

5.1 Main insights from the papers

- Developers are not "the enemy"; crypto APIs must be designed for non-expert developers.
- There are ten key principles for usable crypto APIs (Abstract, Powerful, Comprehensible, Ergonomic, Intuitive, Failing, Safe, Testable, Readable, Explained).
- Real-world Q&A data shows pervasive problems: missing docs, confusion about usage, debugging failures, and build/compatibility issues.
- Patnaik et al. identify four "usability smells": Needs a Super Sleuth, Confusion Reigns, Needs a Post-Mortem, Doesn't Play Well With Others.
- All major libraries (OpenSSL, Bouncy Castle, PyCrypto, etc.) exhibit these smells, even those marketed as "usable."
- The empirical study validates Green & Smith's principles and highlights additional pain points (build, performance, cross-platform deployment).

5.2 Main implications for end-to-end encrypted messaging

- Poor documentation and lack of examples can lead to incorrect key generation and storage in messaging apps.
- Confusion over algorithms and parameters raises the risk of using weak cipher modes or reusing nonces.

- Misuse of key exchange and ratcheting logic can break forward secrecy and make attacks devastating.
- Developers without crypto background may disable or weaken critical security features to “make things work.”
- Build and compatibility pain can result in outdated crypto libraries and inconsistent security guarantees across platforms.
- Mixing different crypto libraries without a clear abstraction layer can cause subtle integration bugs.