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

Green & Smith argue that developers are not malicious or careless "enemies"; they're professionals trying to ship working software. The problem is that crypto APIs are often designed as if every caller were a cryptographer. When APIs are hard to understand or easy to misuse, the fault lies with the *API design*, 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

Patnaik et al. set out to empirically investigate how developers actually struggle with cryptographic libraries in practice. Rather than relying on surveys or lab studies, they analyzed real-world questions from Stack Overflow to understand what problems developers face when using crypto libraries.

3.2.2 Method

The study collected 2,491 questions (2,317 after filtering) about seven popular crypto libraries: OpenSSL, NaCl, libsodium, Bouncy Castle, SJCL, Crypto-JS, and PyCrypto. Using qualitative coding, they categorized the problems developers reported and identified 16 specific usability issues. These were then grouped into 4 higher-level patterns they called "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

The study's findings are sobering: all seven libraries exhibit all four usability smells, meaning no library is truly "problem-free." Even "modern" libraries like libsodium and NaCl, which are often praised for usability, still produce significant developer friction.

The empirical evidence strongly supports Green & Smith's principles, but it also reveals areas they under-emphasized. Build and deployment processes, cross-platform compatibility, and performance concerns turn out to be major pain points that deserve more attention in API design.

4 Implications for the Chosen Technology (E2EE Messaging)

Applying the insights from Green & Smith and Patnaik et al. to end-to-end encrypted messaging reveals several critical areas where API usability directly impacts security outcomes.

4.1 Documentation gaps leading to insecure key management in messaging apps

- Both papers identify missing or unclear documentation as a major source of confusion. In E2EE messaging, this becomes dangerous when developers do not understand proper key management. They might use weak randomness sources, store keys insecurely, or omit key rotation entirely.
- Following Green & Smith's *Abstract* principle, E2EE messaging SDKs should provide task-based documentation and examples (like "register a user and generate keys" or "rotate device keys") rather than forcing developers to work with low-level key operations directly.

4.2 Parameter confusion leading to weak or misconfigured encryption

- Patnaik et al. found developers frequently struggle with choosing algorithms, modes, key sizes, and parameters like IVs/nonces. For messaging apps, this can lead to catastrophic failures: developers might choose insecure modes like AES-ECB, reuse IVs/nonces, or omit authentication entirely.
- Libraries should follow the *Safe* principle by offering single, high-level encryption primitives that enforce good choices internally (AEAD only, strong defaults) rather than exposing algorithm menus that invite misuse.

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

- When developers misuse functions or can't understand what went wrong, forward secrecy becomes vulnerable. Misuse here means key reuse across messages, broken ratcheting (where one compromised key reveals entire conversations), or missing verification of remote identity keys, opening the door to man-in-the-middle attacks.
- Messaging protocols need to expose ready-made, high-level implementations (like "startSessionWithUser(userId)") rather than raw ECDH/ED25519 operations, automatically enforcing correct ratcheting to make misuse difficult or impossible.

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

- Many Stack Overflow questions reveal developers don't understand *why* certain design choices matter. In E2EE apps, this shows up as developers removing "complicated" steps like rekeying or signature checking, shortening key sizes "for performance," or skipping integrity checks because "decryption already succeeded."
- Messaging SDKs need stronger conceptual documentation that explains forward secrecy, authentication, and integrity, not just how to use the API, but why these features are essential. This could include short sections like "Threat model and why we use this protocol" alongside code examples.

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

- Patnaik et al. found a significant portion of questions concerned build, linking, and platform compatibility, especially for OpenSSL and PyCrypto. E2EE messaging apps run

across iOS, Android, desktop, and web platforms, so build pain can lead to shipping outdated library versions with known vulnerabilities, dropping platforms, or inconsistent security guarantees.

- E2EE messaging stacks should ship with pre-built binaries for all target platforms, minimize external build dependencies, and maintain strict version management to keep crypto libraries updated everywhere.

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

- Patnaik et al. identified "borrowed mental models" as a problem: developers assume one library behaves like another. When an E2EE messaging stack mixes libraries (say, OpenSSL for transport and libsodium for end-to-end encryption), mismatched assumptions about parameter order, error codes, or encoding formats (DER vs. PEM vs. raw bytes) can cause subtle but serious bugs.
- The solution is to prefer a single crypto library for the core protocol and provide an app-facing API layer that completely hides underlying library details.

5 Summary

5.1 Main insights from the papers

- Green & Smith argue that developers are not "the enemy"; crypto APIs must be designed for non-expert developers, not cryptographers. They propose ten principles for usable crypto APIs: Abstract, Powerful, Comprehensible, Ergonomic, Intuitive, Failing, Safe, Testable, Readable, and Explained.
- Patnaik et al.'s empirical study of Stack Overflow questions validates these principles while revealing the scale of the problem. They identify four recurring "usability smells": Needs a Super Sleuth (missing documentation), Confusion Reigns (uncertainty about usage), Needs a Post-Mortem (poor error handling), and Doesn't Play Well With Others (integration problems).
- Notably, all major libraries, including those marketed as "usable" like libsodium and NaCl, exhibit these problems. The study also highlights areas Green & Smith under-emphasized, particularly build processes, performance concerns, and cross-platform deployment challenges.

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, as developers may use weak randomness sources or omit key rotation entirely.
- Confusion over algorithms and parameters raises the risk of using weak cipher modes (like AES-ECB) or reusing nonces, which can completely break confidentiality or integrity.
- Misuse of key exchange and ratcheting logic can break forward secrecy, allowing one compromised key to reveal entire conversations or enabling man-in-the-middle attacks.
- Developers without crypto background may disable or weaken critical security features (like rekeying or signature checking) to "make things work" or improve performance.

- Build and compatibility pain can result in outdated crypto libraries with known vulnerabilities and inconsistent security guarantees across platforms (iOS, Android, desktop, web).
- Mixing different crypto libraries without a clear abstraction layer can cause subtle integration bugs due to mismatched assumptions about parameter order, error codes, or encoding formats.