TRIPLE: Time-Randomizing Interface Protocol Language Encryption

Next Generation Secure Communications

Breaking the boundaries of key-based encryption through polymorphic obfuscation. By decoupling itself from the constraints of established cryptographic traditions, TRIPLE evolves as a platform-agnostic solution that adapts to the rapidly shifting landscape of cybersecurity challenges. Its underlying architecture is intentionally modular, allowing seamless integration into a wide range of digital ecosystems—from lightweight IoT devices to enterprise cloud infrastructures—without imposing prohibitive overhead.

The protocol is also engineered with interoperability in mind. The annexed flow diagrams illustrate how TRIPLE can be embedded directly within browser platforms, ensuring secure transmission even in environments traditionally considered vulnerable. This flexibility means organizations can deploy TRIPLE alongside, or even in place of, existing security layers, reducing reliance on brittle, math-bound ciphers and hedging against emergent quantum threats.

Furthermore, TRIPLE's unique pointer array mechanism is complemented by an agile development approach, supporting continuous updates and rapid adaptation to new attack vectors. Its ephemeral, non-persistent logic minimizes the risk window for adversaries, while its session continuity feature bolsters usability for legitimate users.

TRIPLE is a newly patented solution from US 11,956, 352 B2 invented by Mark Taylor.

Prepared by:
TRIPLE Software Engineering Team
Date: July 2025

Contact: triple@systemsdi.com

Web: https://SystemsDesignInnovation.com/triple/

Systems Design Innovation LLC
301 West Broad Street
Suite 226
Falls Church
VA 22046

Contents

---

1. Introduction

TRIPLE (Time-Randomizing Interface Protocol Language Encryption) is a quantum-resistant encryption method that creates a unique and constantly evolving language protocol between two systems. It transforms the landscape of communication security by removing reliance on fixed keys and instead implementing randomized periodic key-morphing and refreshing, or replacement, encoding schemes using pointer arrays and shared seed data.

2. TRIPLE System Flow Diagrams

Refer to separately attached diagrams illustrating TRIPLE protocol operation, pointer encryption, sender-receiver exchange, and key generation logic (see Annex A).

TRIPLE's functionality is best illustrated using the four key flow diagrams described below:

(a) **TRIPLE Key Creation Diagram** – Depicts how session keys are generated from input (pre-key), using salting, hashing, and derivation functions. This allows secure and recoverable key bootstrapping.

(b) **TRIPLE Protocol Overview Diagram** – Shows how two devices ping to confirm readiness, create a shared pointer array, and enter an encrypted session that is periodically reset based on time or message volume.

(c) **TRIPL Pointer Array Encryption Diagram** – Illustrates how encryption is performed using randomized pointers into a shared array. The array is refreshed or rotated at session reset intervals. Pointers are stored and restored for continuity.

(d) **Sender and Receiver Diagram** – Describes the message flow: Sender encrypts a message using its TRIPLE array, transmits it to the receiver, which decrypts using the synchronized array.

3. Technology Overview and Benefits

TRIPLE is a protocol-level enhancement that replaces or compliments static encryption keys with randomly generated languages that can be set to evolve randomly over time and / or use randomly timed whole key replacements. It uses pointer-based referencing into shared seed data arrays to generate encrypted representations of characters, words, or phrases.

4. Technical Challenges and Implementation Notes

The known challenges and TRIPLE's responses, including pointer synchronization, protocol reset, session validation, and memory security.

Technical Challenge	TRIPLE Solution
Pointer Synchronization	Shared random seed and ping-validation protocol.
Session Reset Logic	Timed or traffic-volume-based array resets with autorotation.
Session Validation	Hash of pointer array and device ID timestamps ensure resync.

Secure Storage	Use volatile memory and Rust-based sandboxed modules.
Reconnection Handling	Saved pointer arrays (Figure 1C) enable resumption.

5. Pseudocode Prototypes

The following pseudocode outlines the structure and initialization process for the TRIPLE protocol's main module. This module is responsible for coordinating secure sessions between two devices by constructing a TRIPL array based on a predefined set of seed data. Each element in the TRIPL array is derived using a pointer generation function, ensuring uniqueness and randomness throughout the session.

Comparative Overview of Two TRIPLE_Main Prototypes:

The provided pseudocode describes two versions of a module named TRIPLE_Main, both focused on session management and secure message encryption/decryption between two devices. Each version employs a TRIPL (a list of pointers or values), initialized from a seed, and uses this structure for message operations. Below, we summarize and compare their structure and logic.

First Version of TRIPLE_Main

- MAX_TRIPL_LENGTH: A constant set to 1024, defining the length of the TRIPL array.

InitializeSession(DeviceA, DeviceB):
- Loads seed data and initializes an empty TRIPL list.
- Generates TRIPL by iterating from 0 to MAX_TRIPL_LENGTH - 1, populating it using GeneratePointer(seedArray, i).
- The TRIPL is stored for session continuation.
- Returns the TRIPL.

EncryptMessage(message, TRIPL):
For each character in the message, it maps the character to an index, retrieves the corresponding pointer in TRIPL, and adds it to the encrypted list.

DecryptMessage(encrypted, TRIPL):
For each pointer in the encrypted list, reverses the mapping using TRIPL to retrieve the original character and reconstructs the message.

CheckResetTimer():
If the session is expired, calls InitializeSession to renew TRIPL.
If not expired, rotates pointers in the current TRIPL.

Second Version of TRIPLE_Main

StartSession(DeviceA, DeviceB):

- Initializes an empty TRIPL and loads seed data.
- Iterates from 1 to MAX_TRIPL_LENGTH, generating random pointers using GetRandomPointer(seedArray).
- Exchanges each generated value between DeviceA and DeviceB, then adds it to TRIPL.
- Returns the TRIPL.

EncryptMessage(message, TRIPL):
For each character, looks up a pointer in TRIPL and adds it to the encrypted list.

DecryptMessage(encrypted, TRIPL):
For each pointer, performs a reverse lookup in TRIPL to retrieve the original character, reconstructing the message.

Comparison and Key Differences
Session Initialization:
- The first version uses InitializeSession starting at index 0 and iterates to MAX_TRIPL_LENGTH - 1, while the second version uses StartSession starting at index 1 up to MAX_TRIPL_LENGTH. This subtle difference changes the number of iterations by one.
- The first version uses GeneratePointer with an explicit index, while the second uses GetRandomPointer without a direct index, suggesting a different approach to pointer generation.
- The second version includes ExchangeValue between DeviceA and DeviceB during TRIPL generation, emphasizing communication or key exchange, whereas the first does not.

Message Encryption/Decryption:
The mapping and reverse mapping methods differ slightly by name: MapCharToIndex and LookupPointer for encryption; ReverseMap and ReverseLookup for decryption. Functionally, both achieve the transformation of characters to pointers and vice versa.

Session Maintenance:
The first version contains CheckResetTimer to handle session expiration and pointer rotation; the second version lacks explicit timer reset or pointer rotation logic.

Summary

Both versions of TRIPLE_Main aim to establish a shared session context and provide secure procedures for encrypting and decrypting messages using a pointer-based lookup table (TRIPL). The primary differences lie in session initialization details, pointer generation, and session lifecycle management. Each implementation presents a variant on how sessions might be initiated and maintained for secure communication between devices.

```
MODULE TRIPLE_Main
  CONSTANT MAX_TRIPL_LENGTH ← 1024

  PROCEDURE InitializeSession(DeviceA, DeviceB)
    seedArray ← LoadSeedData()
    TRIPL ← []
```

```
    FOR i FROM 0 TO MAX_TRIPL_LENGTH - 1 DO
      value ← GeneratePointer(seedArray, i)
      TRIPL.ADD(value)
    END FOR
    StoreTRIPL(TRIPL) // Enables session continuation
    RETURN TRIPL
  END PROCEDURE

  PROCEDURE EncryptMessage(message, TRIPL)
    encrypted ← []
    FOR EACH char IN message DO
      index ← MapCharToIndex(char)
      pointer ← TRIPL[index]
      encrypted.ADD(pointer)
    END FOR
    RETURN encrypted
  END PROCEDURE

  PROCEDURE DecryptMessage(encrypted, TRIPL)
    message ← []
    FOR EACH pointer IN encrypted DO
      char ← ReverseMap(TRIPL, pointer)
      message.ADD(char)
    END FOR
    RETURN message
  END PROCEDURE

  PROCEDURE CheckResetTimer()
    IF SessionExpired() THEN
      TRIPL ← InitializeSession(DeviceA, DeviceB)
    ELSE
      RotatePointers(TRIPL)
    END IF
  END PROCEDURE
END MODULE

MODULE TRIPLE_Main
  PROCEDURE StartSession(DeviceA, DeviceB)
    TRIPL ← []
    seedArray ← LoadSeedData()

    FOR i FROM 1 TO MAX_TRIPL_LENGTH DO
      value ← GetRandomPointer(seedArray)
      ExchangeValue(DeviceA, DeviceB, value)
      TRIPL.ADD(value)
```

```
    END FOR
    RETURN TRIPL
  END PROCEDURE

  PROCEDURE EncryptMessage(message, TRIPL)
    encrypted ← []
    FOR EACH char IN message DO
      pointer ← LookupPointer(TRIPL, char)
      encrypted.ADD(pointer)
    END FOR
    RETURN encrypted
  END PROCEDURE

  PROCEDURE DecryptMessage(encrypted, TRIPL)
    message ← []
    FOR EACH pointer IN encrypted DO
      char ← ReverseLookup(TRIPL, pointer)
      message.ADD(char)
    END FOR
    RETURN message
  END PROCEDURE
END MODULE
```

---

6. C-Code Prototype

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define MAX_TRIPL_LENGTH 256  // One entry per ASCII character
#define OBFUSCATION_MASK 0xAB // XOR mask for reversible
transformation

typedef struct {
    char character;
    char encoded_value[5]; // e.g., "x5F"
} TRIPLPointer;

typedef struct {
    TRIPLPointer map[MAX_TRIPL_LENGTH];
    int count;
    time_t start_time;
} TRIPLArray;

// Build deterministic ASCII map with encoded representations
TRIPLArray create_tripl_array() {
    TRIPLArray array;
```

```
    array.count = MAX_TRIPL_LENGTH;
    array.start_time = time(NULL);

    for (int i = 0; i < MAX_TRIPL_LENGTH; i++) {
        array.map[i].character = (char)i;
        snprintf(array.map[i].encoded_value,
sizeof(array.map[i].encoded_value), "x%02X", i ^ OBFUSCATION_MASK);
    }

    return array;
}

// Encrypt using direct ASCII index (deterministic mapping)
void encrypt_message(const char *message, TRIPLArray *array, char
**encrypted) {
    int i = 0;
    while (message[i] != '\0') {
        unsigned char index = (unsigned char)message[i];
        encrypted[i] = strdup(array->map[index].encoded_value);
        i++;
    }
    encrypted[i] = NULL; // Mark end of encrypted sequence
}

// Decrypt using reverse lookup (linear scan)
void decrypt_message(char **encrypted, TRIPLArray *array, char
*output) {
    int i = 0;
    while (encrypted[i] != NULL) {
        for (int j = 0; j < array->count; j++) {
            if (strcmp(encrypted[i], array->map[j].encoded_value) ==
0) {
                output[i] = array->map[j].character;
                break;
            }
        }
        i++;
    }
    output[i] = '\0'; // Proper null-termination
}

// Demonstration
int main() {
    TRIPLArray tripl = create_tripl_array();
    const char *original = "HELLO TRIPLE!";
    char *encrypted[1024];
    char decrypted[1024];

    encrypt_message(original, &tripl, encrypted);
    printf("Encrypted:\n");
    for (int i = 0; encrypted[i] != NULL; i++) {
        printf("%s ", encrypted[i]);
    }
    printf("\n");
```

```
    decrypt_message(encrypted, &tripl, decrypted);
    printf("Decrypted: %s\n", decrypted);

    for (int i = 0; encrypted[i] != NULL; i++) {
        free(encrypted[i]);
    }

    return 0;
}
```

---

7. Action Plan for Developers and Deployment

Phase 1 – Specification & Design (Month 1)

- Finalize pointer logic and reset triggers

- Confirm message format and protocol framing

Phase 2 – Core Implementation (Months 2–3)

- Build TRIPLE engine in Rust with WebAssembly support

- Construct UI shell for browser extension

- Add pointer mutation and storage/resume logic

Phase 3 – Testing & Validation (Months 4–5)

- Test dropped session recovery using Figure 1C logic

- Simulate cross-device pointer sharing

- Harden against memory leaks and sync errors

Phase 4 – Deployment & Pilots (Month 6+)

- Deploy to Chrome Web Store and Edge Add-ons

- Launch enterprise pilots and feedback loop

8. Red Zone Threat Analysis

TRIPLE anticipates threats from both classical and quantum attackers, removing fixed key logic, avoiding pattern-based encryption, and producing cryptographically meaningless outputs even when intercepted.

9. Unlike Regular Math-based Encryption A Feature Not a Bug!

TRIPLE does not conform to conventional encryption frameworks based on prime factorization, elliptic curves, or modular arithmetic. It is not built around a fixed key, a cipher block, or a published algorithmic primitive. As a result, some automated tools and critics may dismiss it as "not suitable for real-world encryption."

That potential for critique, however, would arise from misunderstanding the TRIPLE architecture entirely. Because TRIPLE works by creating a shared, randomized pointer array between two communicating devices. Each pointer acts as a cipher mapping for characters or bits. This shared map is:

- Created mutually, not transmitted or stored
- Reset periodically, based on time or message volume
- Ephemeral, residing in volatile memory only
- Restorable, allowing secure session continuity without rekeying

TRIPLE creates a stream of synchronized, transient, polymorphic ciphers. This means:

- There is no fixed decryption logic for an attacker to reverse-engineer
- Intercepted data is meaningless without access to the current pointer array
- There is no key to steal or mathematically factor
- There is no persistent pattern to discover, since the cipher logic resets
- It resists both brute force and quantum modeling, because TRIPLE is not based on algebraic math — it's based on entropy, mutation, and obfuscation.

Conventional wisdom is to evaluate encryption against standard patterns: AES, RSA, ECC, OpenSSL, etc. TRIPLE is none of those. It's deliberately orthogonal. Based on the math-based encryption types that are all vulnerable to intelligence services' capabilities already. Which vulnerability will only worsen as quantum computing and artificial intelligence-based code-breaking and decryption capabilities increase and proliferate into the hands of non-state actors. The whole point of the approach of TRIPLE is driven by the inventor's insight that this piece of conventional wisdom is going obsolete soon. This is not a bug in technology. It's the point of it.

TRIPLE is designed as a new category of encryption, built not on math, but on logic, logical-mutation, language divergence, and session-specific context. It's more akin to a

synchronized randomized obfuscation engine than a conventional cipher — and that's what gives it its strength.

10. Conclusion

TRIPLE is a fundamentally new direction for data security. It addresses real-world encryption shortfalls, may future proof communications now against future decryption risks, provides resilience against modern adversaries, and enables scalable secure communication. Because text-based processing is not resource-hungry TRIPLE is not processor-heavy unlike pure math-based encryption. So that TRIPLE does not add significantly to the demand for computing resources or to operating costs.

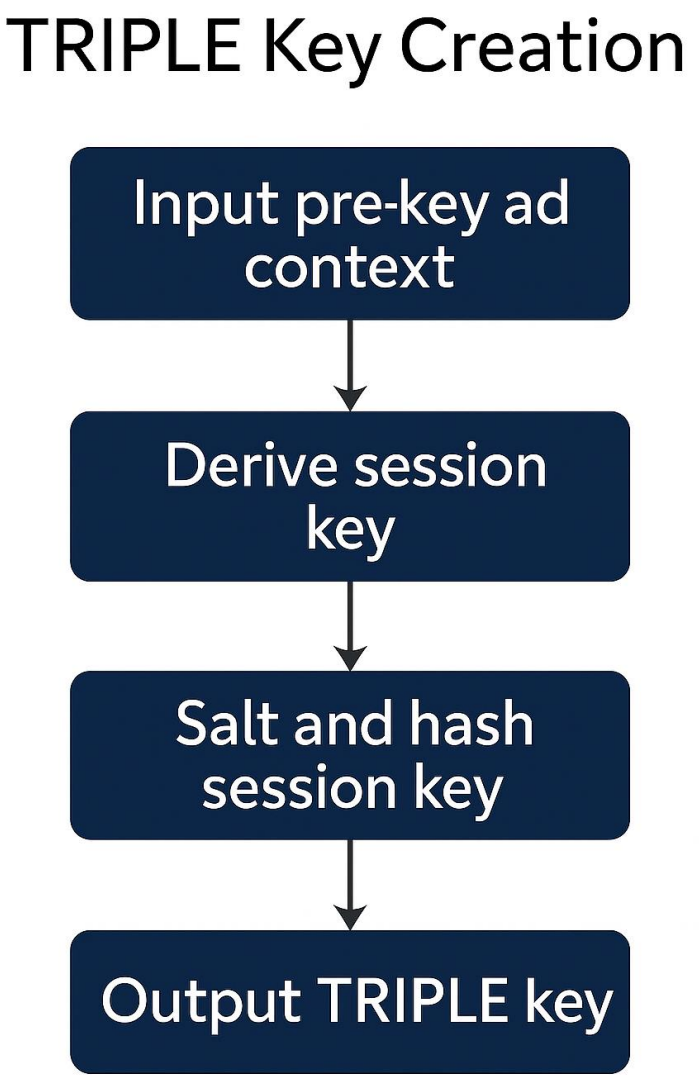
11. Partner and Investor Opportunities

We welcome partnerships and investment from cloud vendors, defense contractors, communication platforms, and cybersecurity stakeholders who recognize the need for zero-trust, future-proof and quantum resistant messaging.

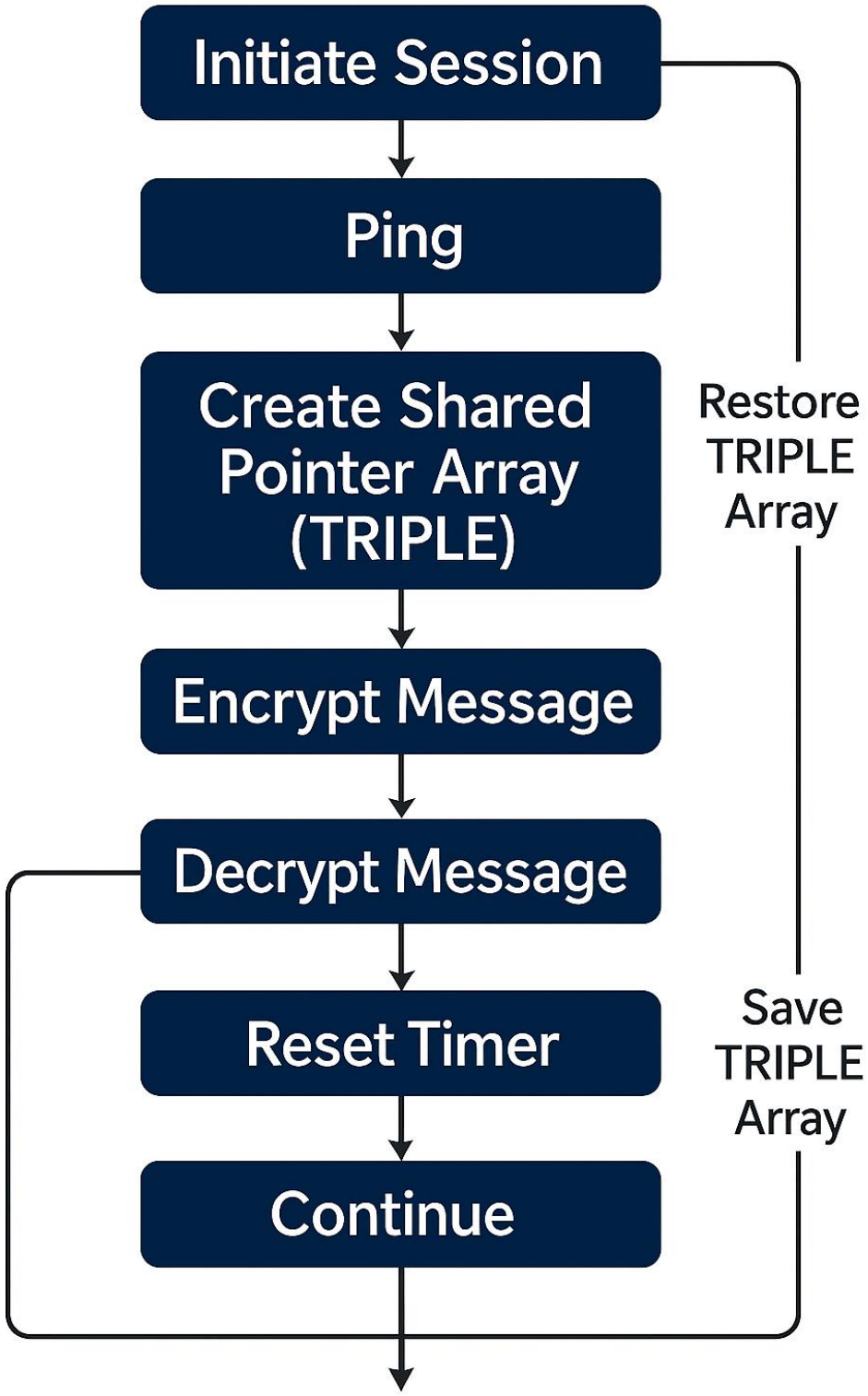
Annex A – Flow Diagrams (for detail see patent US 11,956, 352 B2)

This annex defines the browser integration strategy to ensure TRIPLE remains platform-flexible, standards-compliant, and quantum-resistant even in web-native contexts.

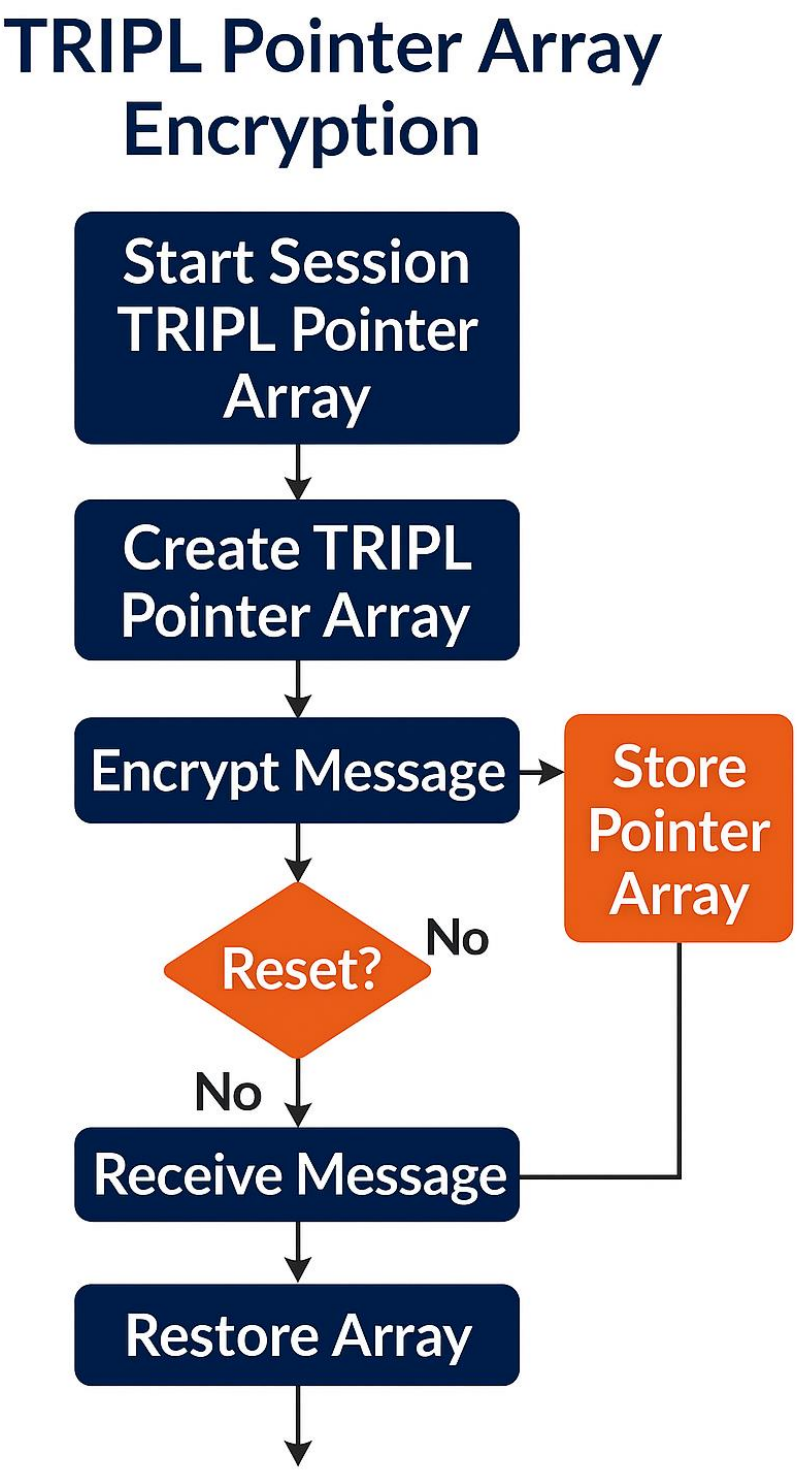
TRIPLE Key Creation Diagram



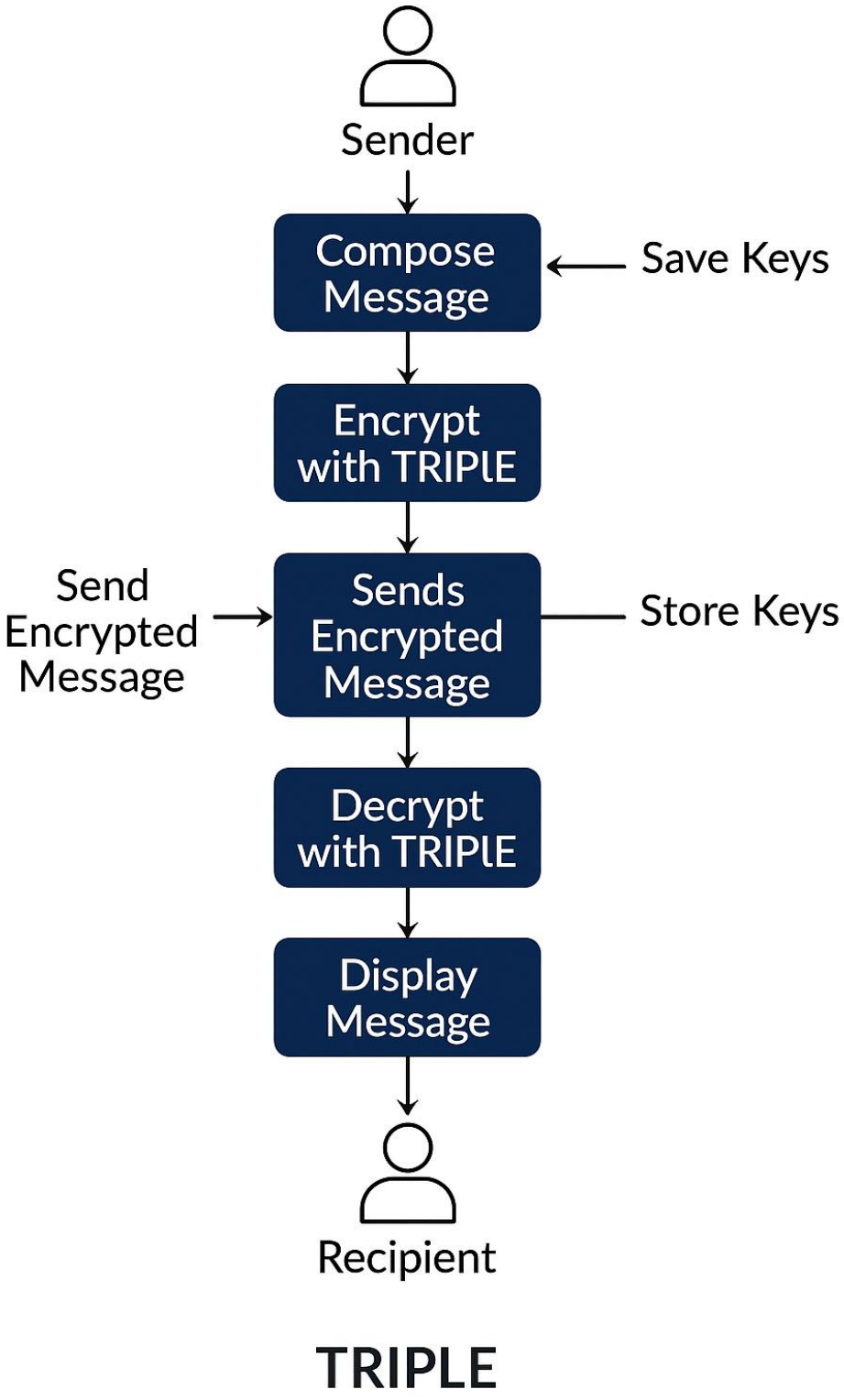
TRIPLE Protocol Overview Diagram



TRIPL Pointer Array Encryption Diagram



TRIPLE Sender and Receiver Diagram



Annex B – Signal Architecture & Language Analysis

Annex B – Signal Architecture & Language Analysis

1. Signal-Android

- Languages Used: Java (core), Kotlin (partial), XML (UI layouts), C/C++ (via JNI)

- Rationale: Java offers wide compatibility and a mature ecosystem; Kotlin brings modern safety and development advantages.

- TRIPLE Recommendation: Use Kotlin for new components (e.g., chat UI, TRIPLE controls); retain Java only where legacy libraries are required.

2. Signal-Desktop

- Languages Used: TypeScript (main logic), HTML/CSS (UI), Electron (runtime environment using Chromium + Node.js)

- Rationale: Enables rapid cross-platform UI development using web technologies; Electron simplifies deployment for Windows, macOS, and Linux.

- TRIPLE Recommendation: Maintain TypeScript + Electron for initial versions; consider Tauri or Rust-native GUI options in future iterations if performance needs grow.

3. Signal Core Cryptographic Engine (libsignal)

- Language: Rust

- Rationale: Rust provides memory safety, performance, and reliability, ideal for encryption modules.

- TRIPLE Recommendation: Use Rust for TRIPLE's core encryption logic — pointer arrays, TRIPL negotiation, salting/hashing, session state validation.

4. Language Suitability Review

All of Signal's chosen languages remain optimal for their platforms in 2025. TRIPLE will adopt them where appropriate to:

- Stay compatible with industry toolchains

- Ensure developer familiarity

- Maintain performance and security

5. Platform Alignment Summary

Platform	**Signal Stack**	**TRIPLE Recommendation**
Android	Java, Kotlin, XML	Kotlin preferred, Java-compatible
Desktop	TypeScript, Electron	TypeScript + Electron
Crypto Core	Rust	Rust

6. Development Strategy

TRIPLE will design legally distinct source code that is functionally similar to Signal, but not copied. By studying Signal's:

- Message flow architecture

- UI component structures

- Session management and sync logic

...we can build a TRIPLE-native equivalent that matches its capabilities while adding next-generation encryption features (e.g., pointer-based language, salting, and quantum resistance).

7. Purpose of This Annex

This annex establishes the architectural compatibility of TRIPLE with known secure messaging platforms and outlines a migration path from reference implementation to proprietary, scalable, and defendable software.

Annex C – TRIPLE Web Browser Integration Strategy

1. Objective

To develop a TRIPLE-powered peer-to-peer secure chat system as a browser extension for Chrome, Edge, and compatible browsers. This system will enable TRIPLE-based encryption directly in the browser environment, layered over existing HTTPS connections or operating natively through bootstrapped session protocols.

2. Operating Modes

TRIPLE browser extensions can operate in two secure modes:

- Bootstrap via HTTPS (Figure 1 Logic): Initiates secure TRIPLE handshake within an HTTPS-protected session, using it as a launching point.

- Resume from Persistent TRIPL Arrays (Figure 1C Logic): Resumes encryption using saved session state for long-term or dropped/reconnected communications.

3. Architecture

- Frontend (UI): HTML, TypeScript, CSS (extension popup and chat interfaces)

- Extension Scripts: Browser APIs, secure local storage, message listeners, TRIPLE logic overlay

- Encryption Core: WebAssembly-compiled Rust module implementing TRIPLE's pointer array engine

- Key and Session Handling: Salted and hashed session fingerprints, ephemeral or persistent TRIPL restoration

4. Compatibility Goals

- Seamless overlay within Gmail, LinkedIn, WhatsApp Web, Slack, or other web-based environments

- Full keyboard input and copy/paste integration

- Minimal memory footprint and sandboxed operations

5. Security Considerations

- No exposure of TRIPL arrays to browser sync or cloud backups

- Local keying and mutation logic never exits runtime memory

- Cross-origin safeguards against injection or manipulation

6. Future Capabilities

- Extend to handle encrypted attachments and eventually TRIPLE-based image encryption

- Integrate real-time sync or message caching for low-connectivity scenarios

7. Deployment Format

- Chrome Web Store release

- Edge Add-ons store compatibility

- Self-hosted CRX files for custom enterprise deployments

Annex D – Signal Open-Source Code References

- Signal-Desktop: https://github.com/signalapp/Signal-Desktop
- Signal-Android: https://github.com/signalapp/Signal-Android
- libsignal: https://github.com/signalapp/libsignal