Formal Specificationand Verification of Protection in Transit (PIT) Protocol

Takwa Rhaimi, Hamed Aghayarzadeh, Rakesh Podder, *Indrakshi Ray*

Computer Science Department

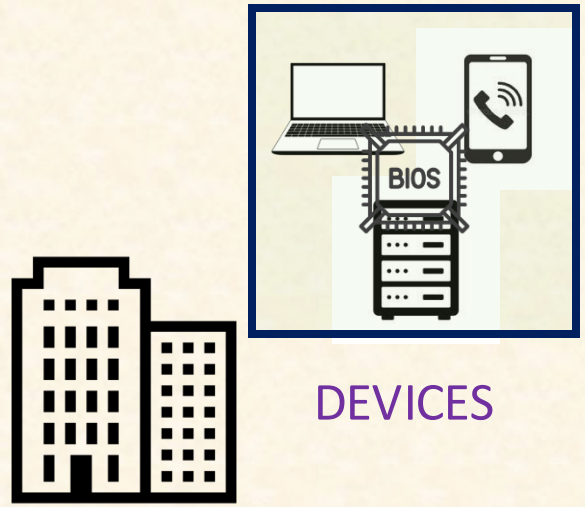
Colorado State University

Outline

- Motivation for the Protection In Transit Protocol
- Protection in Transit Protocol using UML Sequence Diagram
- Transforming UML Sequence Diagram to UPPAAL
- Formal Verification and Refinement
- Conclusion and Future Work

Motivation & Problem Statement

Installation of malware when device is on transit!



DEVICES

COMPANY

(Point of origin)

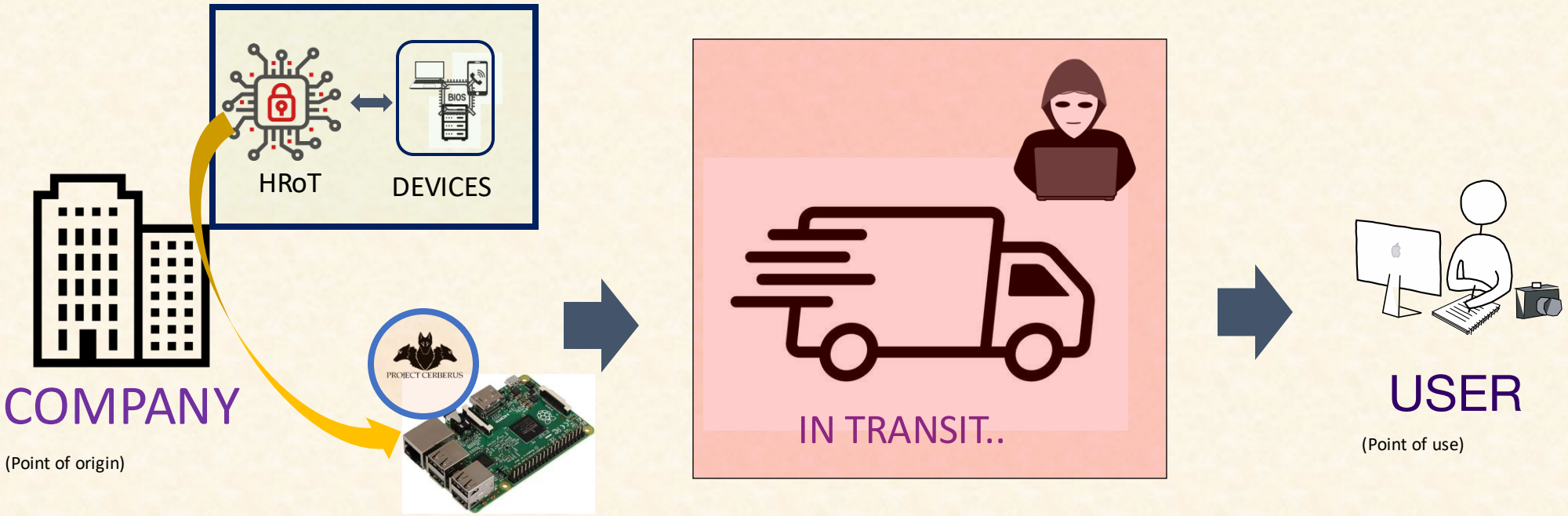
IN TRANSIT..

USER

(Point of use)

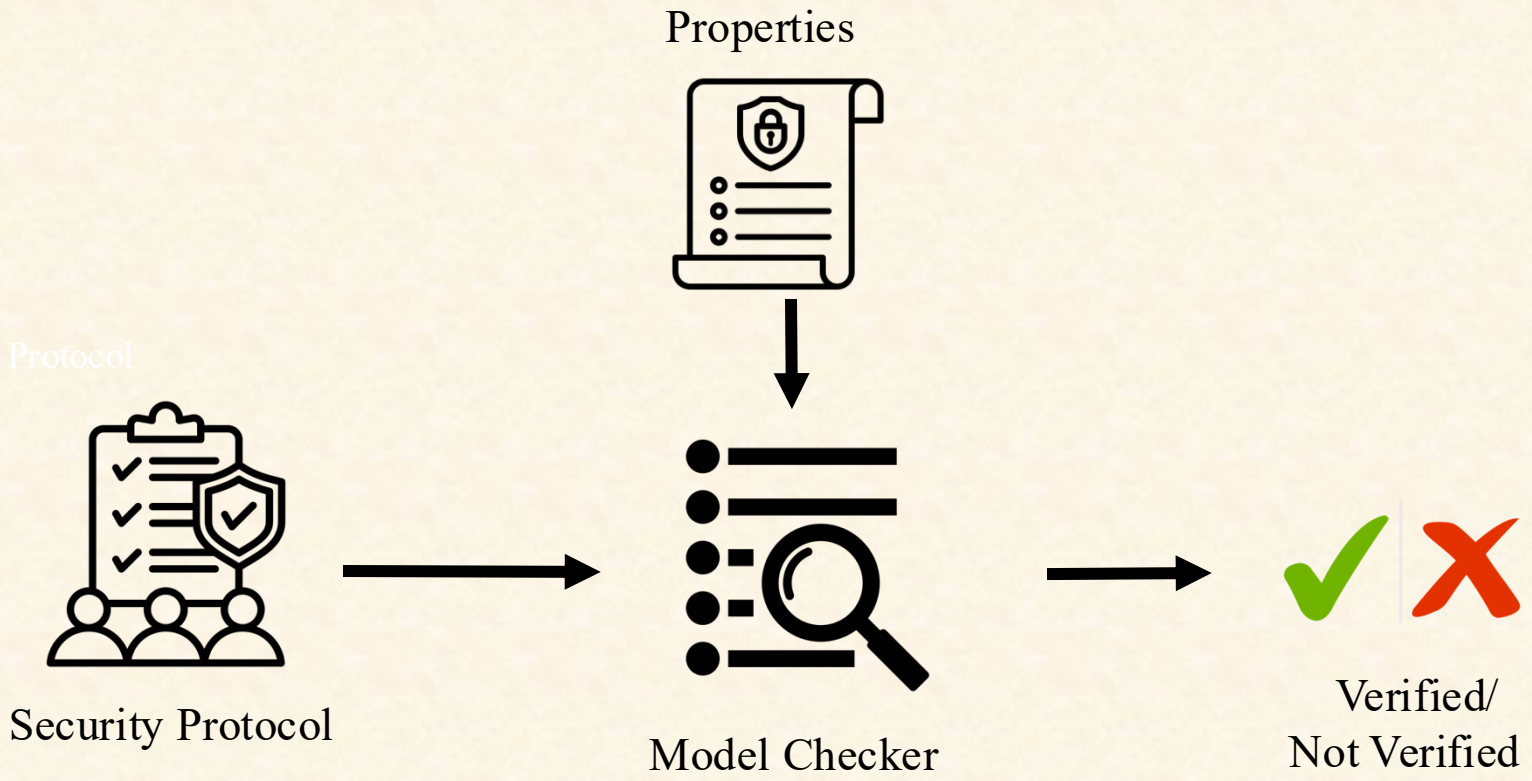
When a device (laptop, workstation, smartphone) is shipped, from the vendor to device owner, malware may be installed in the firmware!

Protection-In-Transit (PIT) Protocol



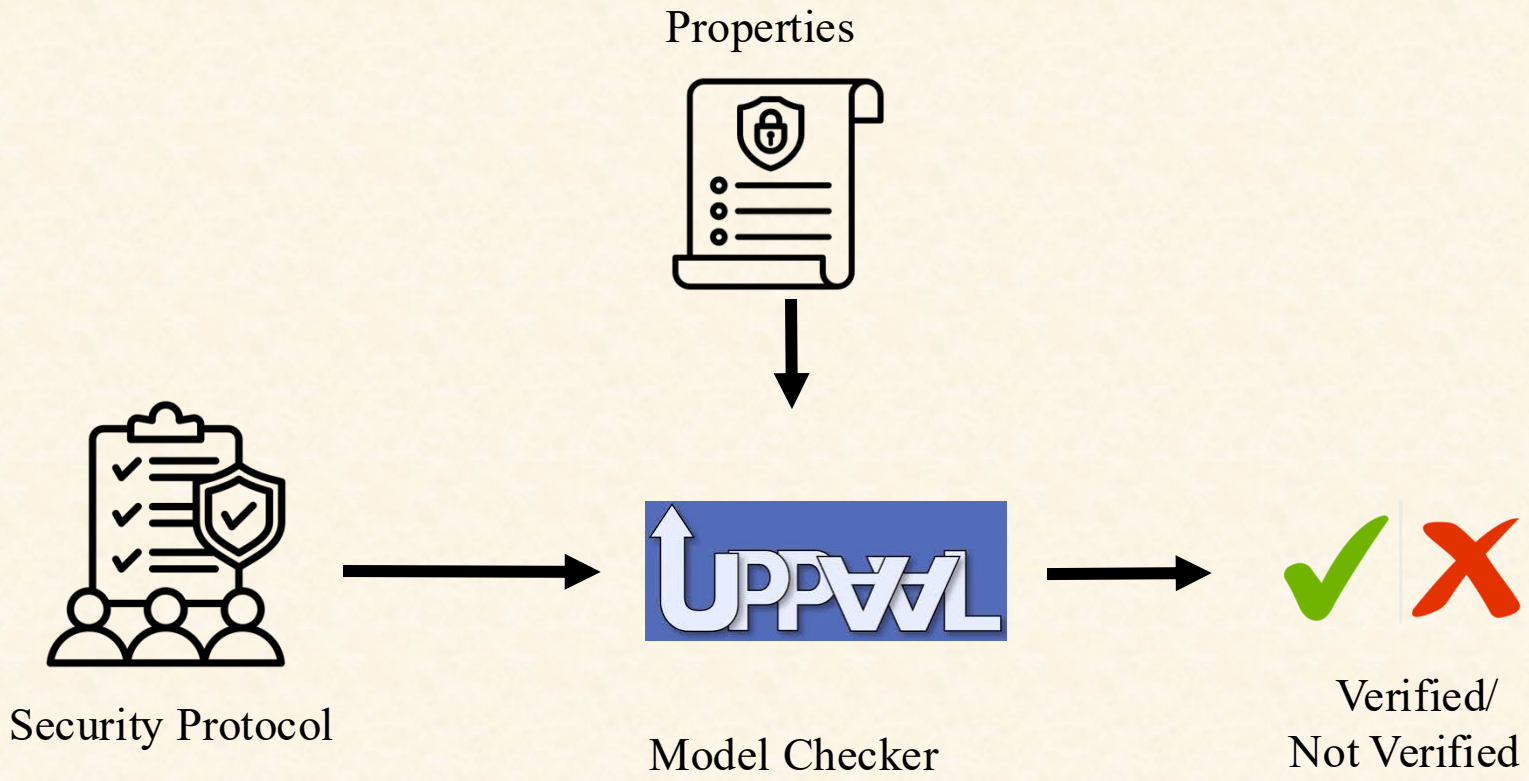
- Locks a device before transit – manufacturer implements a BIOS lock post-production
- Unlocking a device can be performed after the legitimate user has been authenticated

Problem: Is the PIT Protocol Correctly Specified?



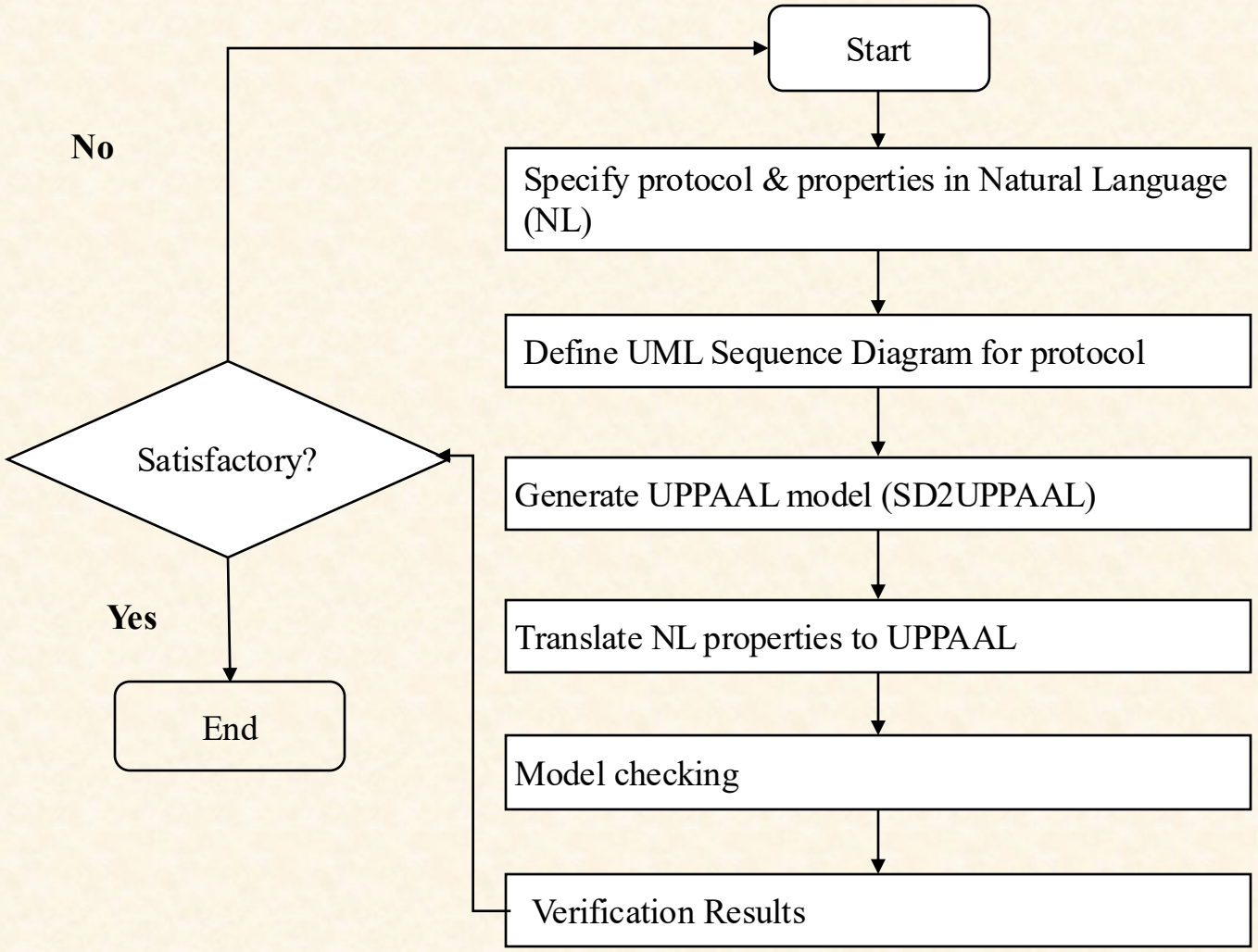
- Security protocols may have vulnerabilities or deviate from expected behavior
- Formal analysis is needed to provide assurance

Solution: Formal Analysis of PIT Protocol



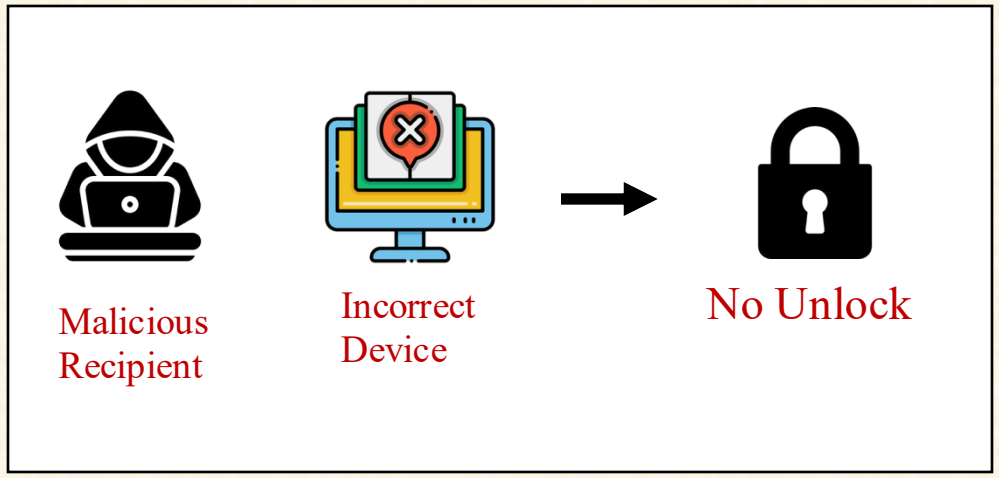
We propose the formal verification of the PIT protocol using the UPPAAL model checker

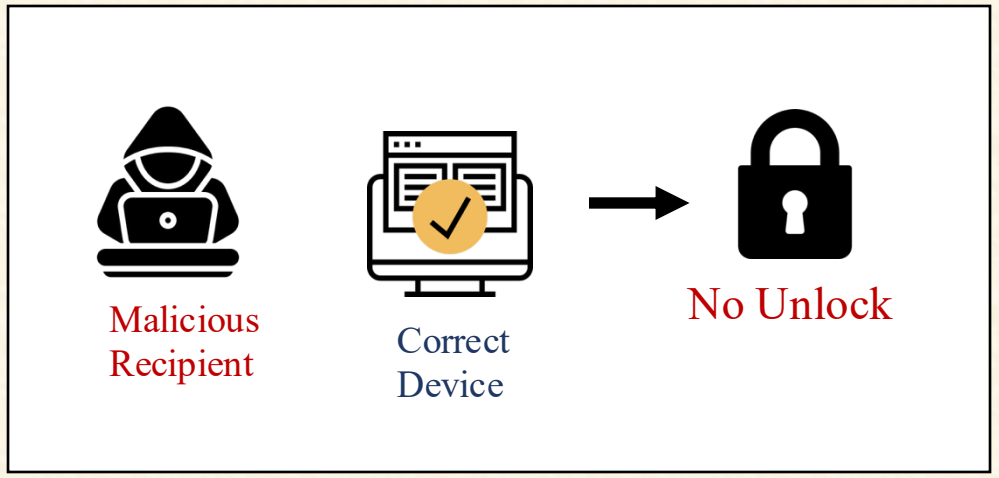
Methodology for Protocol Verification and Refinement

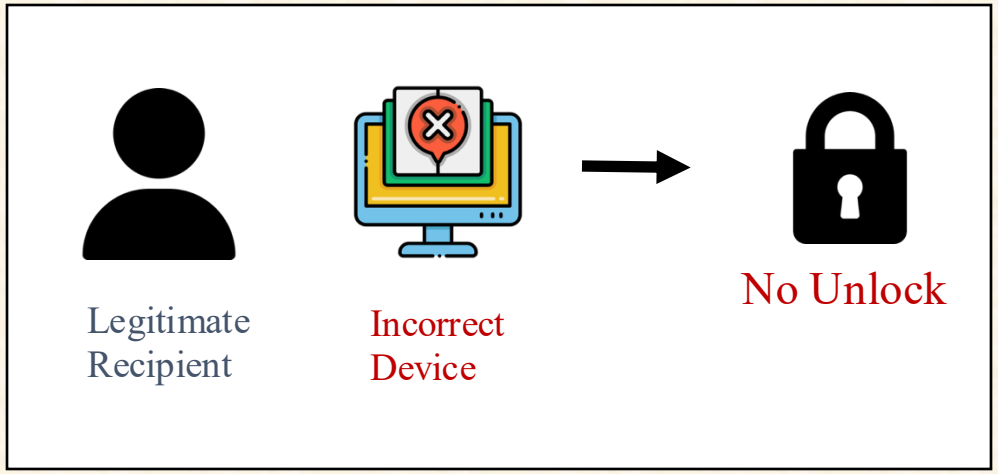


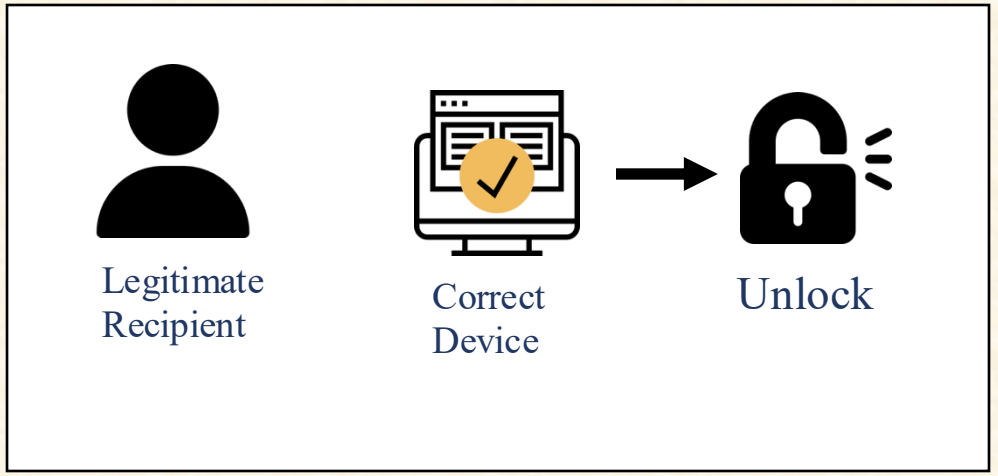
Properties & Protocol Description

Properties

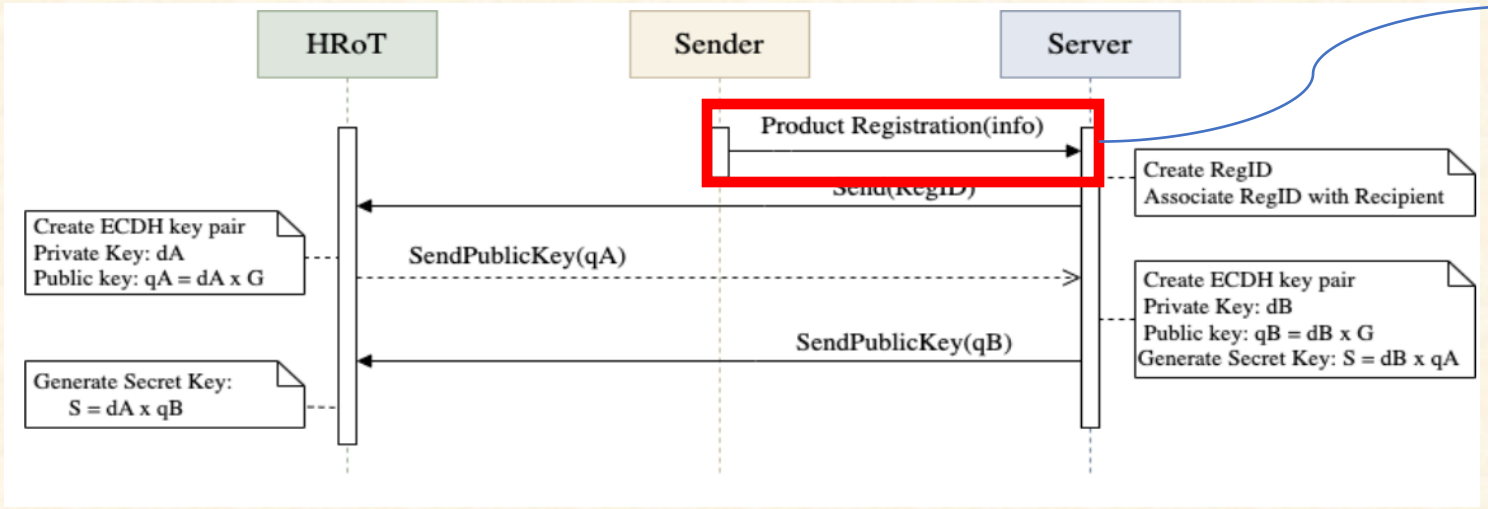

Malicious Recipient
Incorrect Device
No Unlock


Malicious Recipient
Correct Device
No Unlock


Legitimate Recipient
Incorrect Device
No Unlock

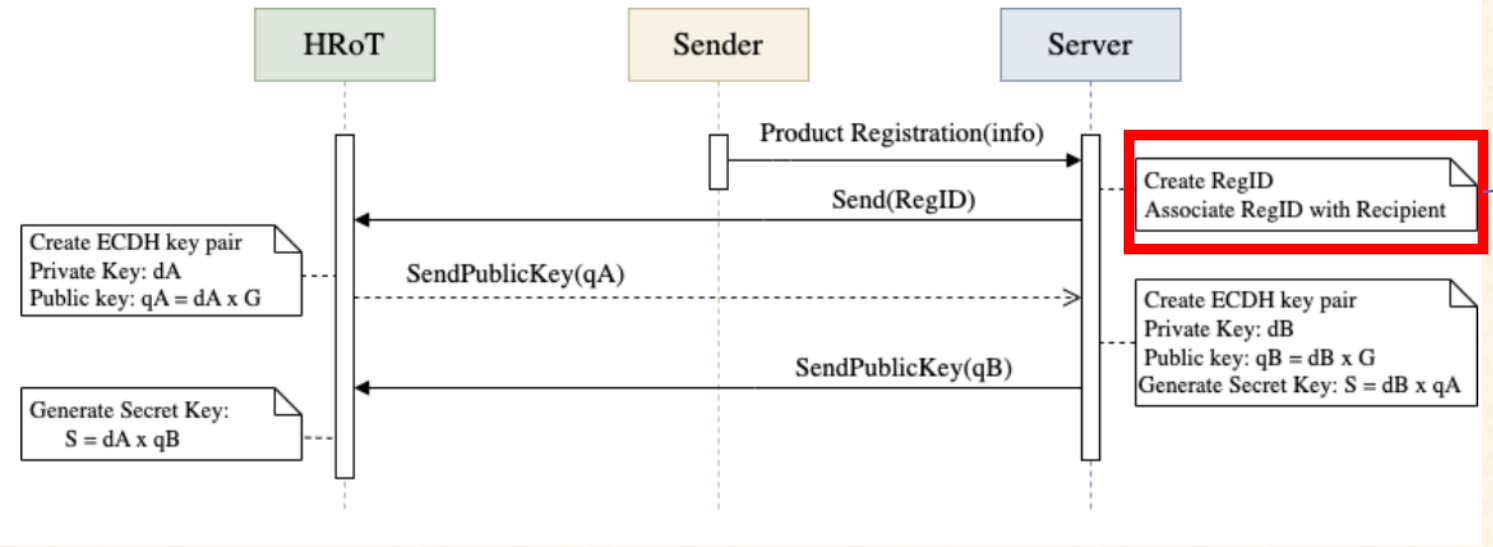

Legitimate Recipient
Correct Device
Unlock

UML Sequence Diagram for Locking Phase



Starts process, Sender sends the Product registration information to Server

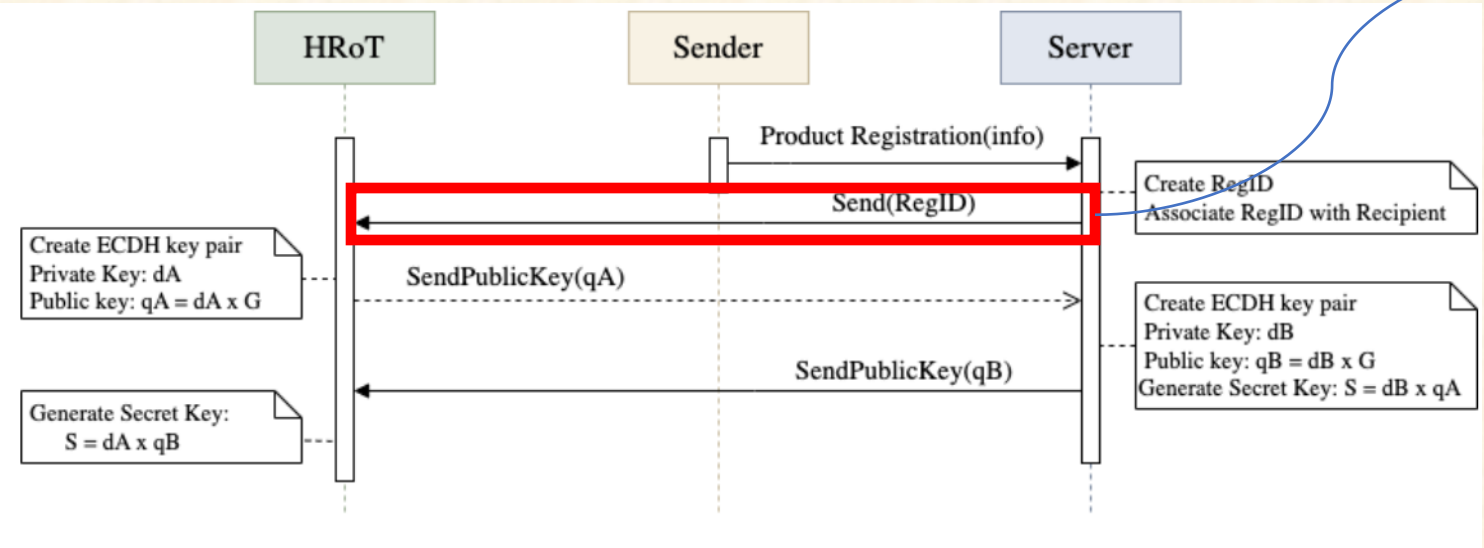
Locking Phase (2)



Server creates unique **RegID** (*Registration ID*) tied to Recipient

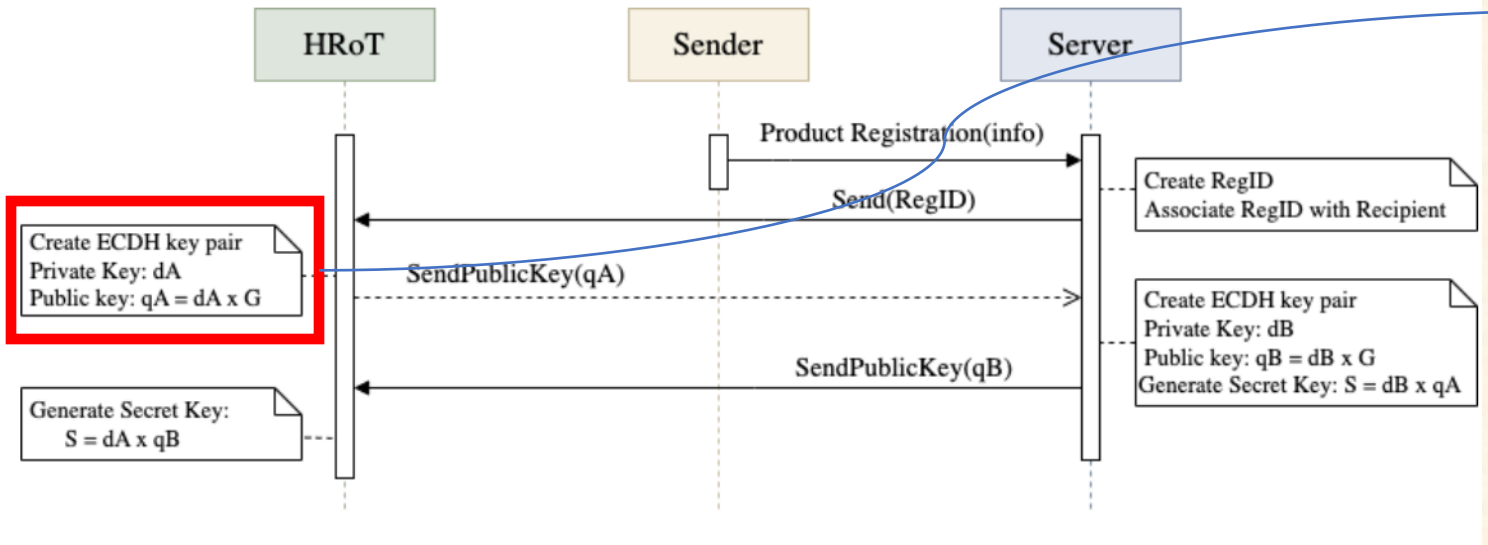
Figure: Sequence Diagram of the Locking Phase

Locking Phase (3)



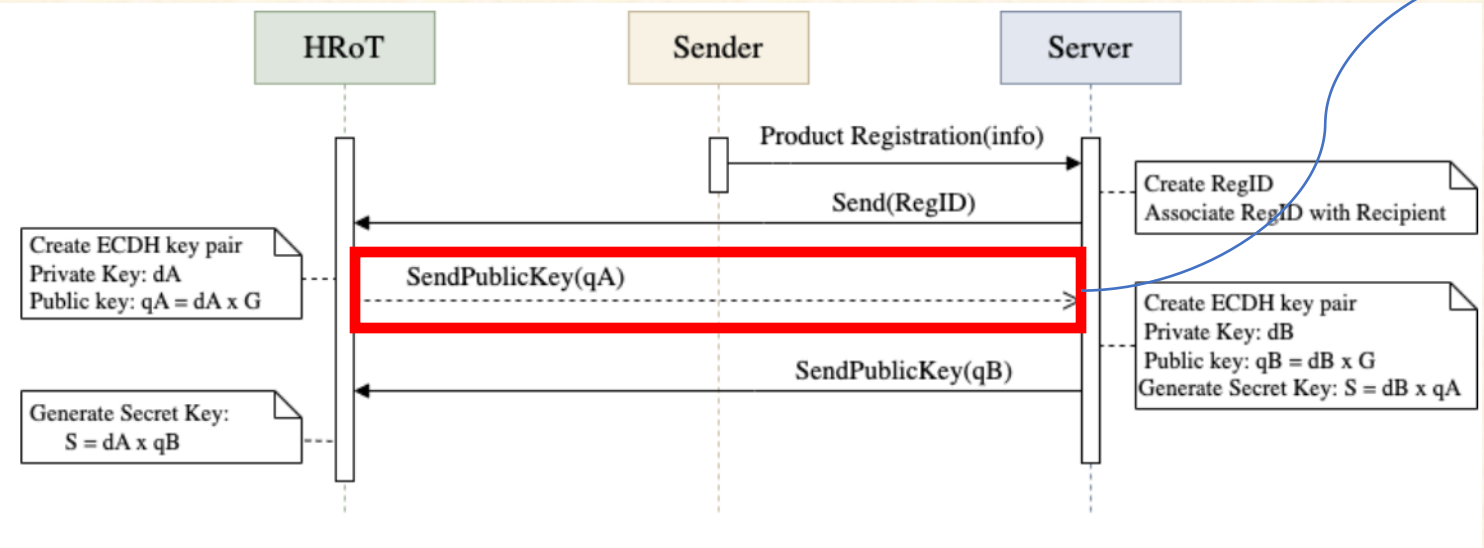
Server sends the Registration ID to HRoT

Locking Phase (4)



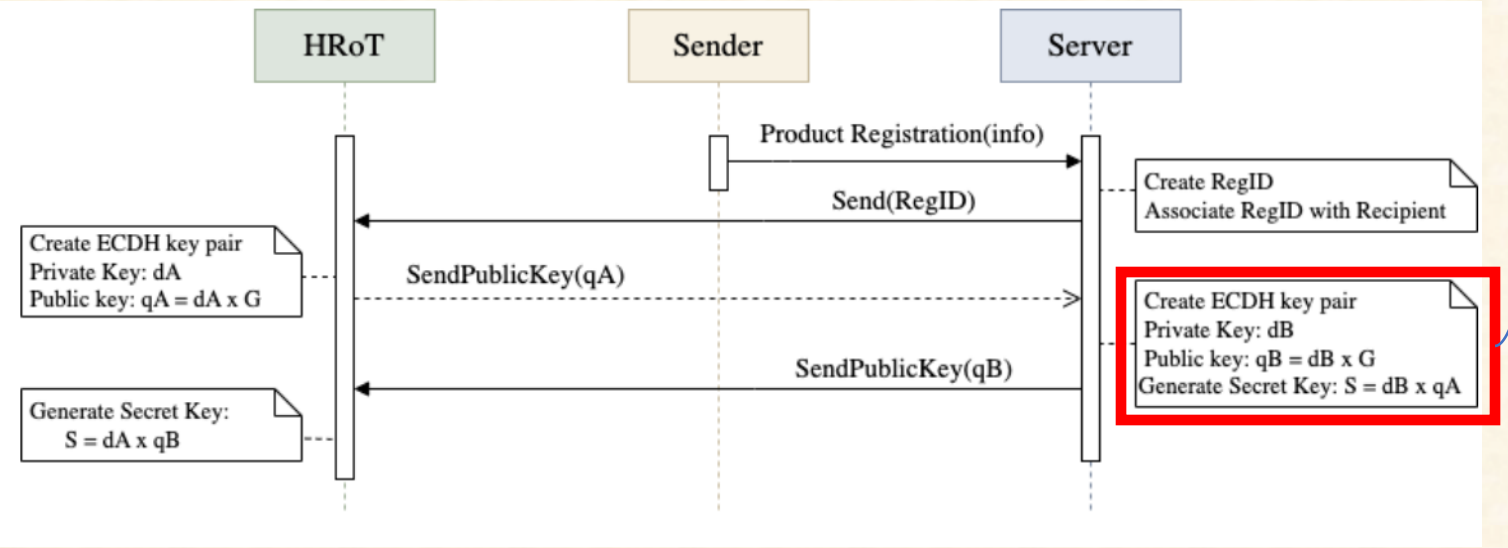
Key Generation – **HRoT** (*Hardware Root of Trust*) creates **ECDH** (*Elliptic Curve Diffie–Hellman*) key pair *(dA = private key, qA = public key)*

Locking Phase (5)



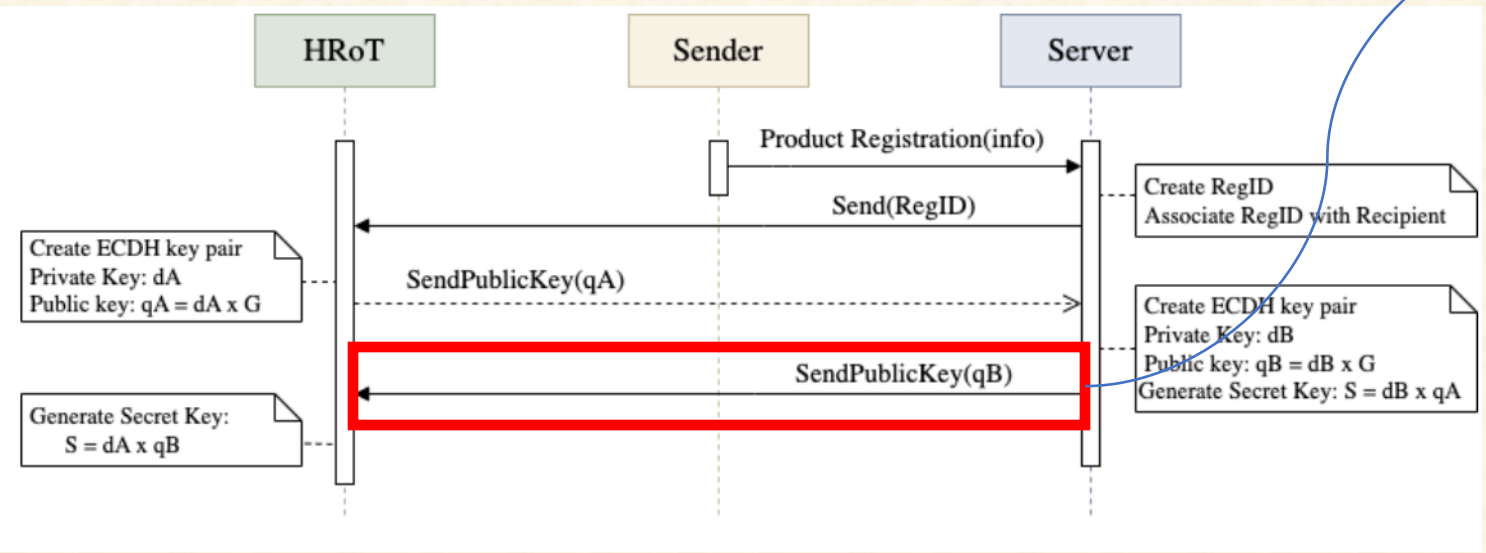
HRoT sends its public key (qA) to Server

Locking Phase (6)



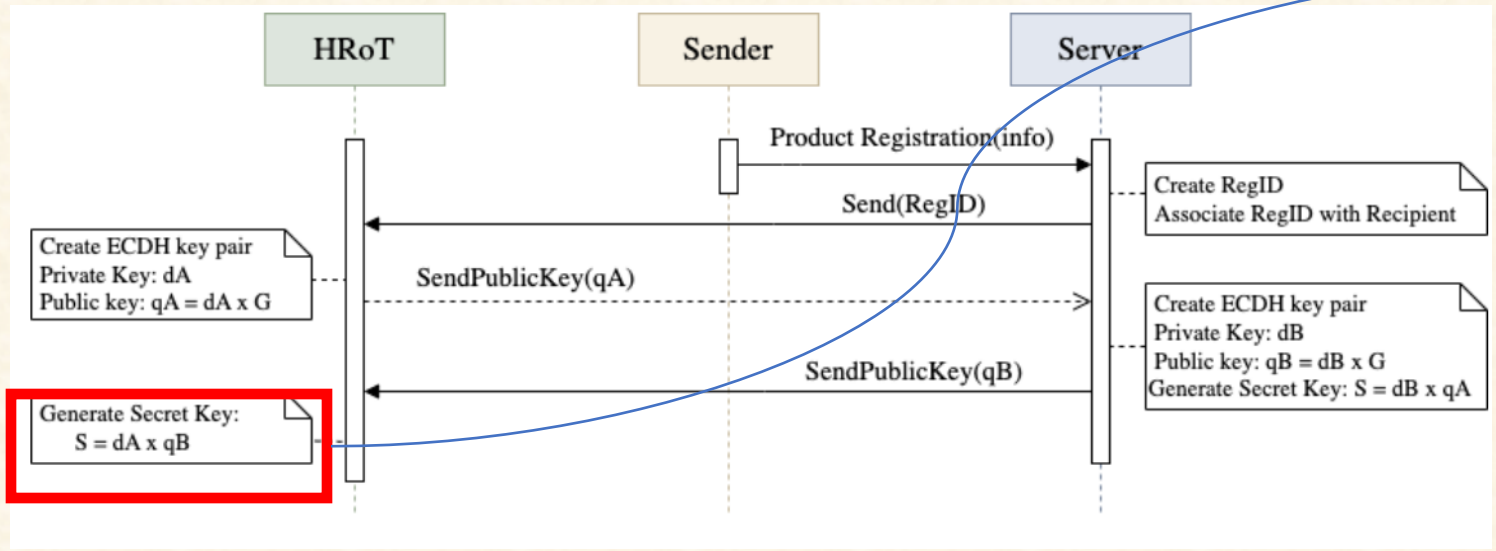
Server generates its key pair (dB, qB) and also **AES** (*Advanced Encryption Standard*) secret key *S* using qA

Locking Phase (7)



Server sends back its public key qB

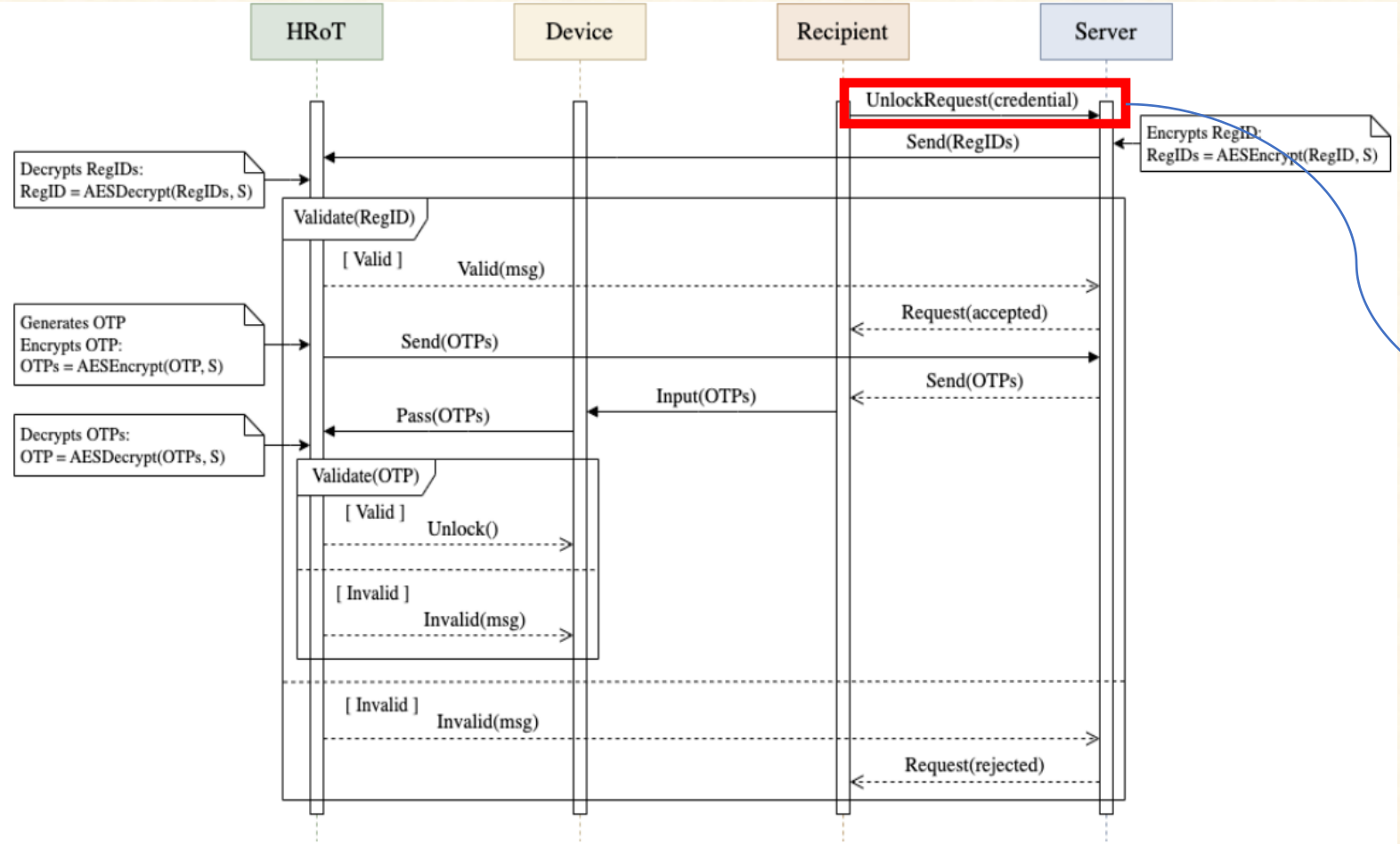
Locking Phase (8)



HRoT computes shared **AES** key *S* from qB (per sequence diagram)

At the end of Locking Phase Server and HRoT share an AES Secret Key *S*

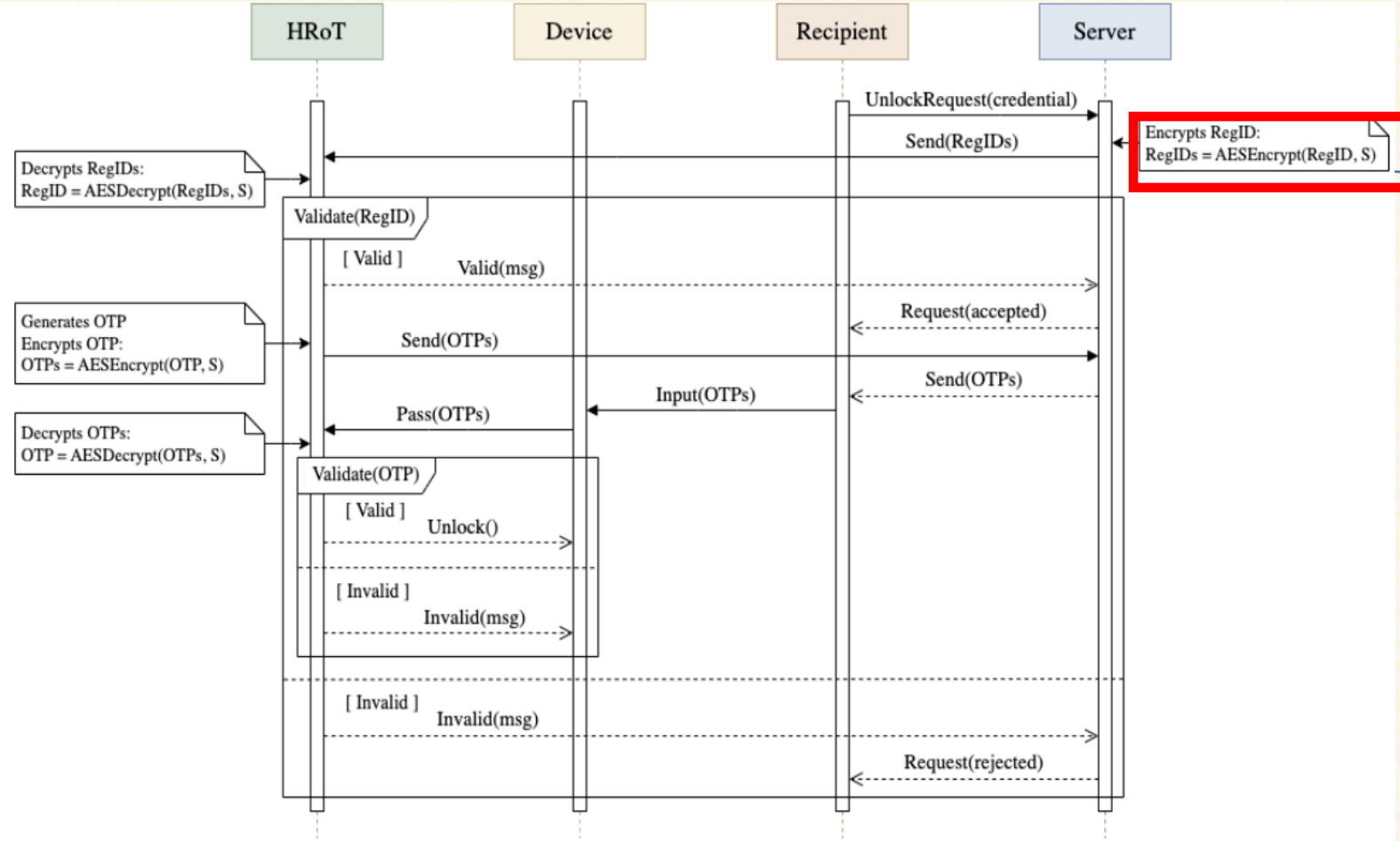
Unlocking Phase (1)



Start Process: Recipient sends unlock request to Server

.

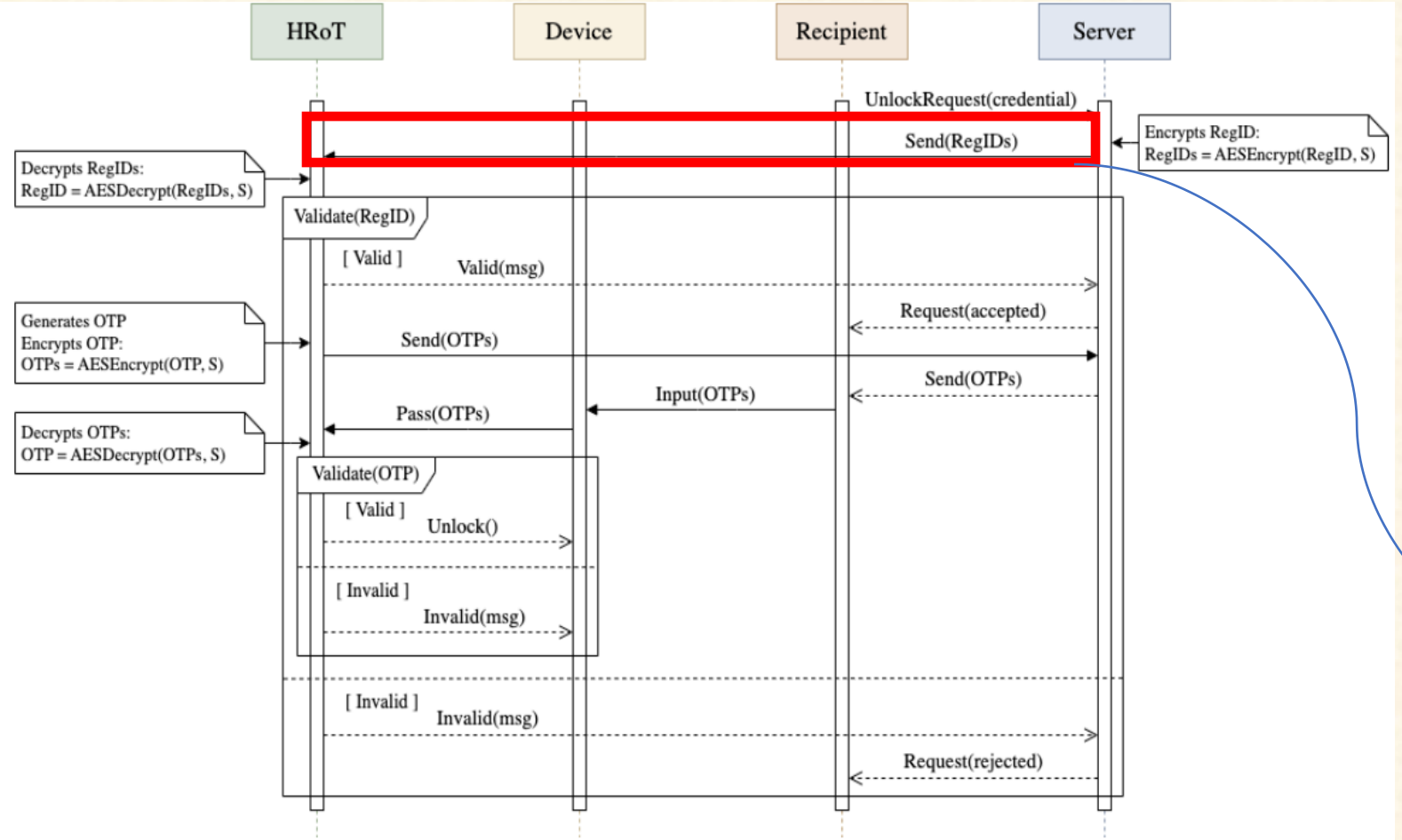
Unlocking Phase (2)



Server encrypts the RegID

.

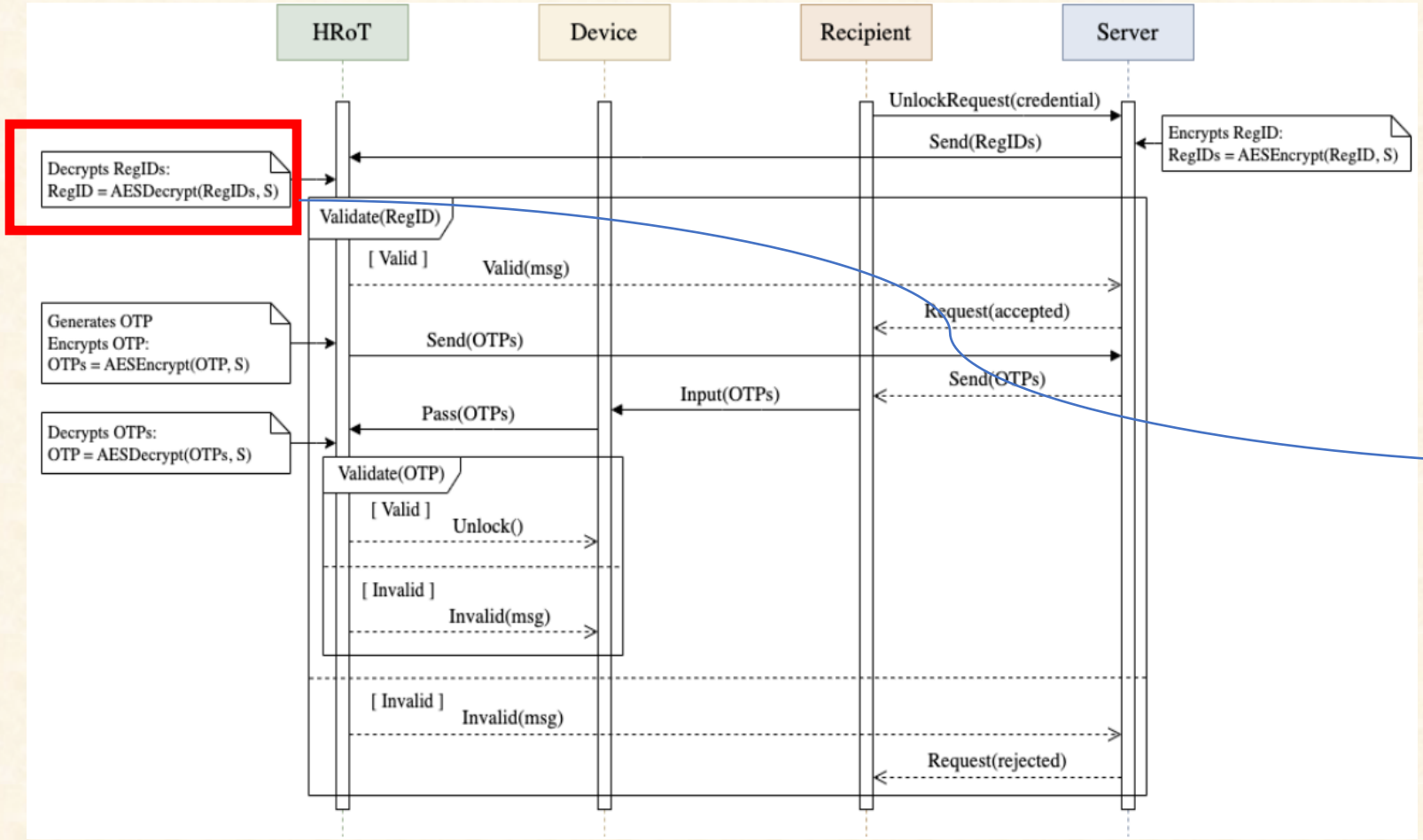
Unlocking Phase (3)



Server sends RegID to HRoT

.

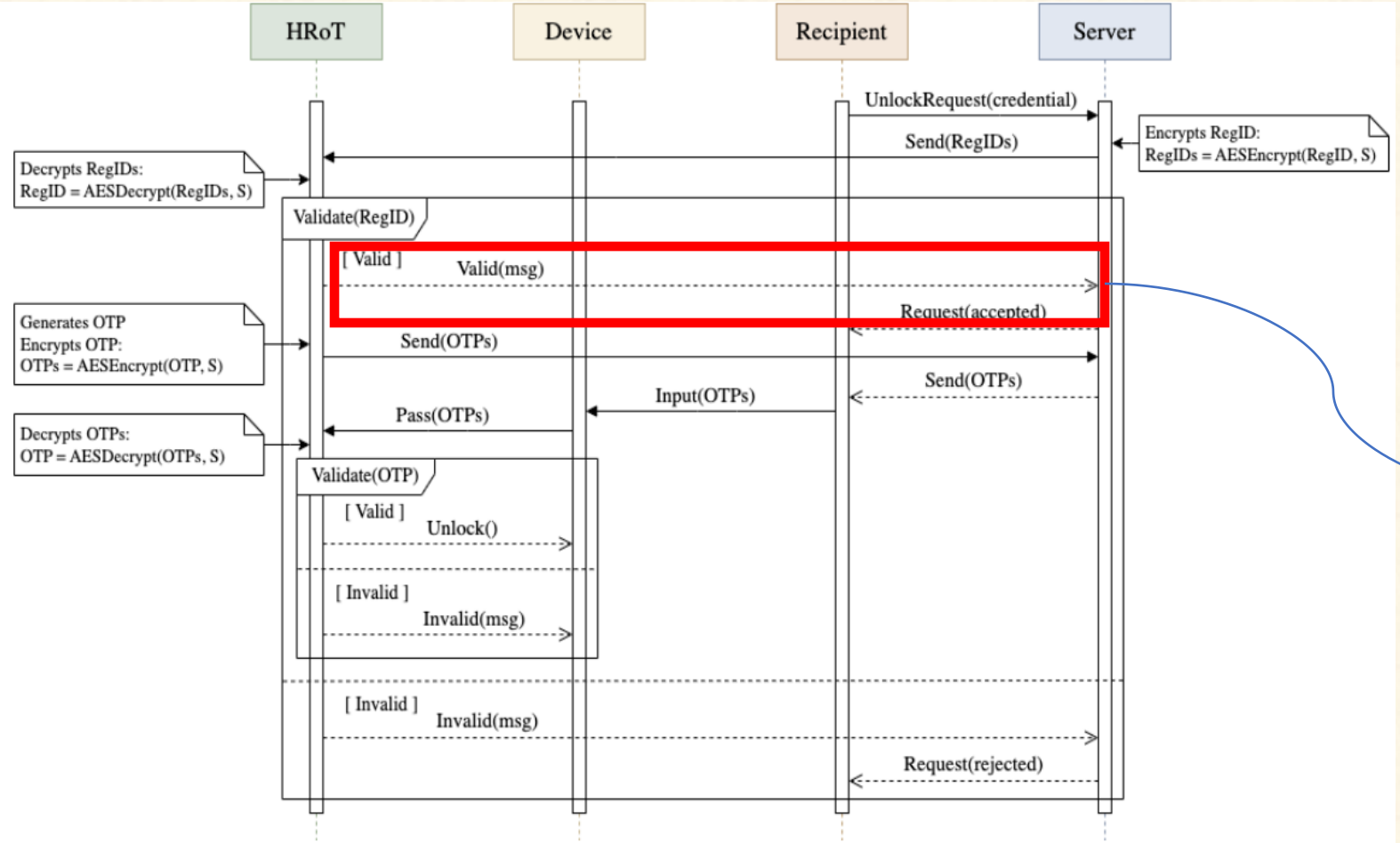
Unlocking Phase (4)



HRoT decrypts the RegID.

.

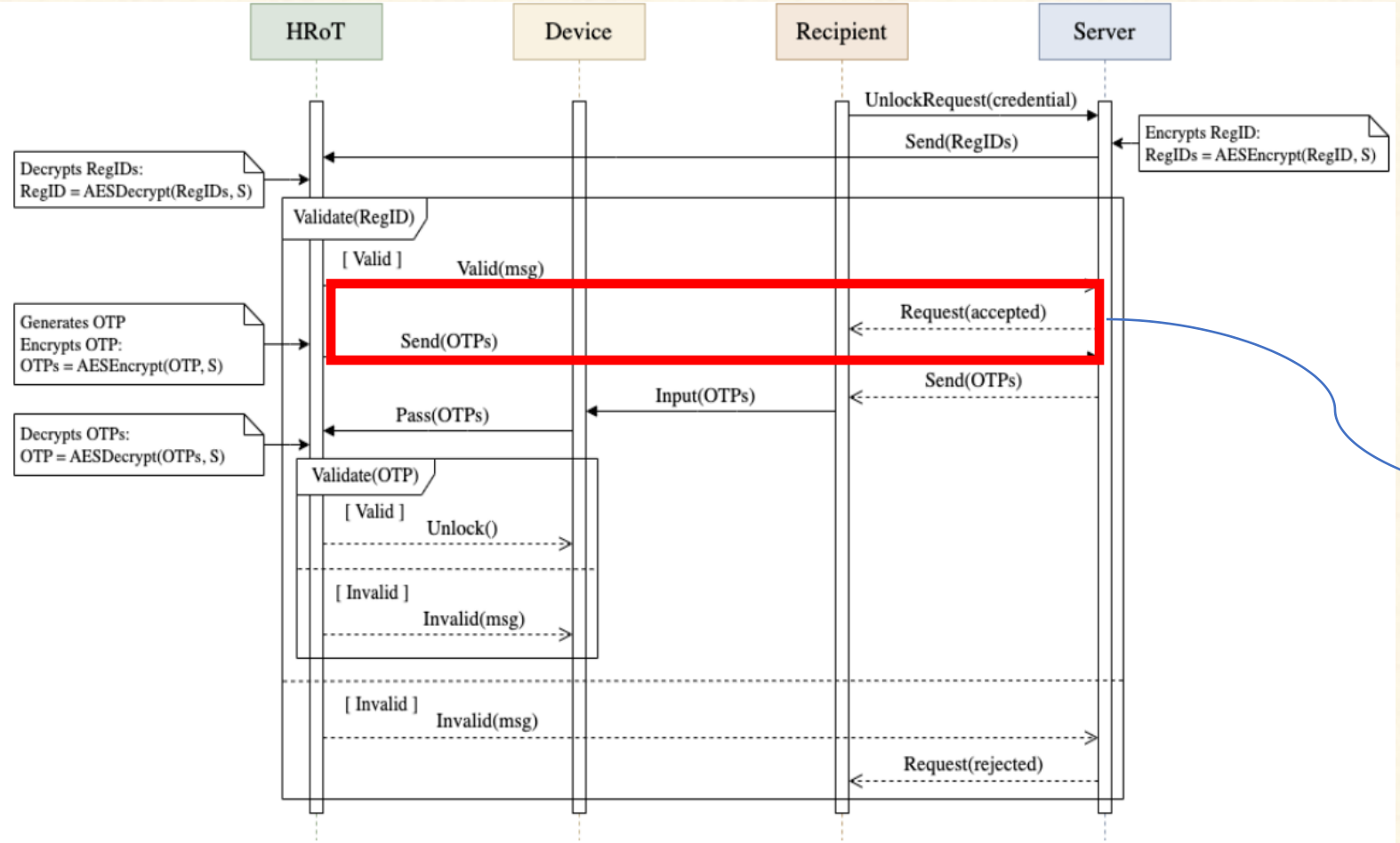
Unlocking Phase (5)



HRoT sends a “Valid” message to the Server if the RegID is correct

.

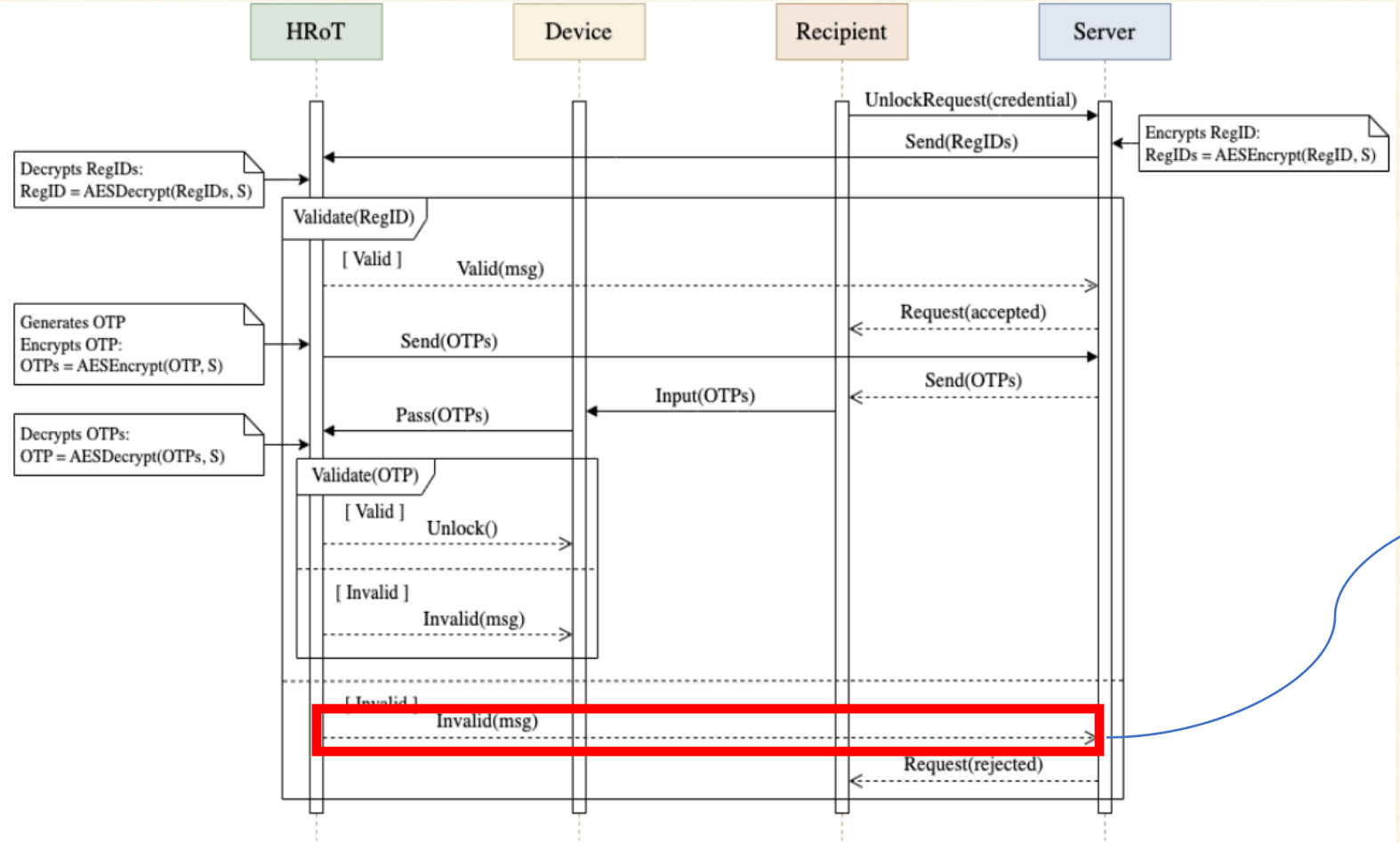
Unlocking Phase (6)



Server sends a “Request Accepted” message to the Recipient

.

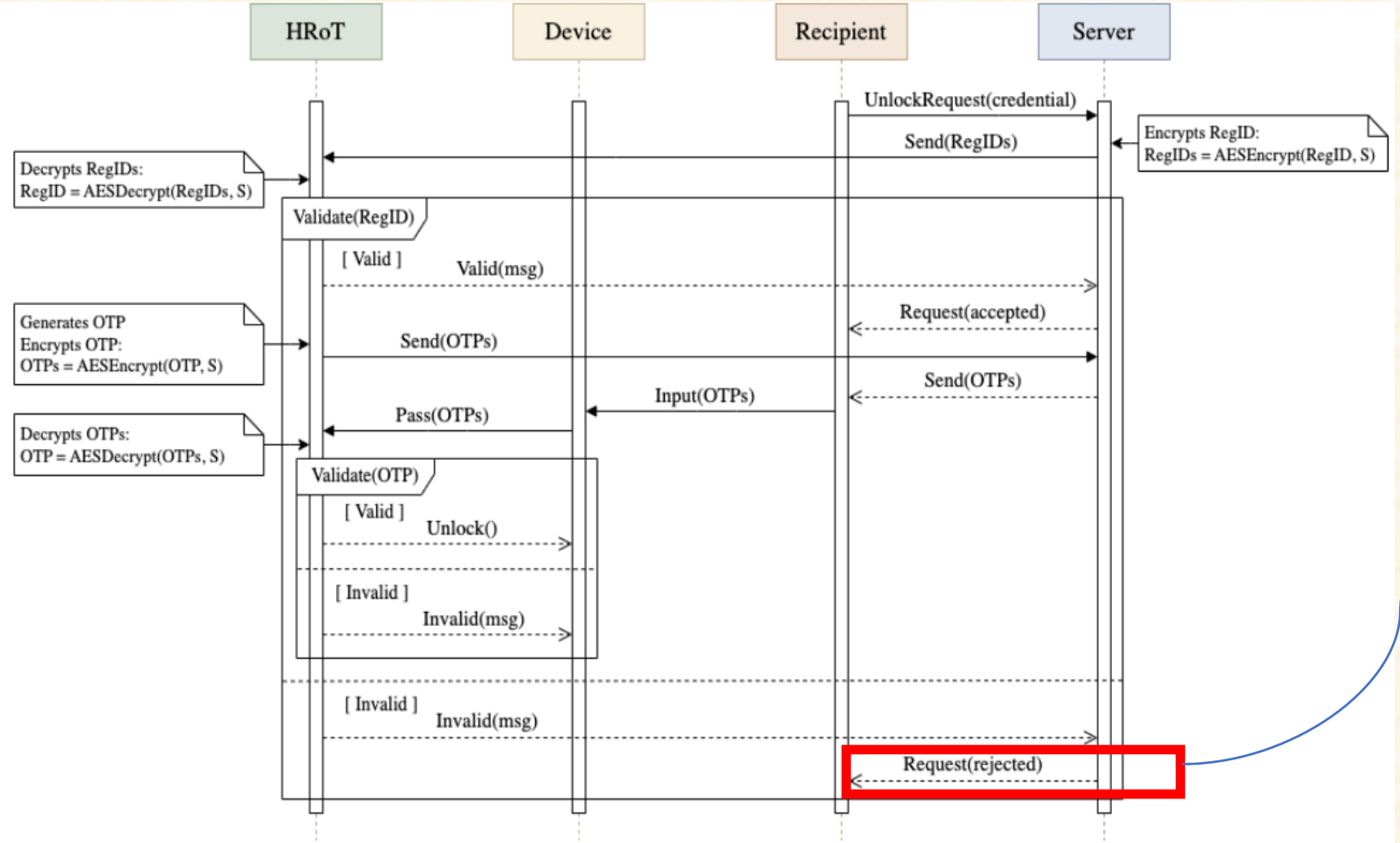
Unlocking Phase (7)



HRoT sends an “Invalid” message to the Server if the RegID is not valid.

.

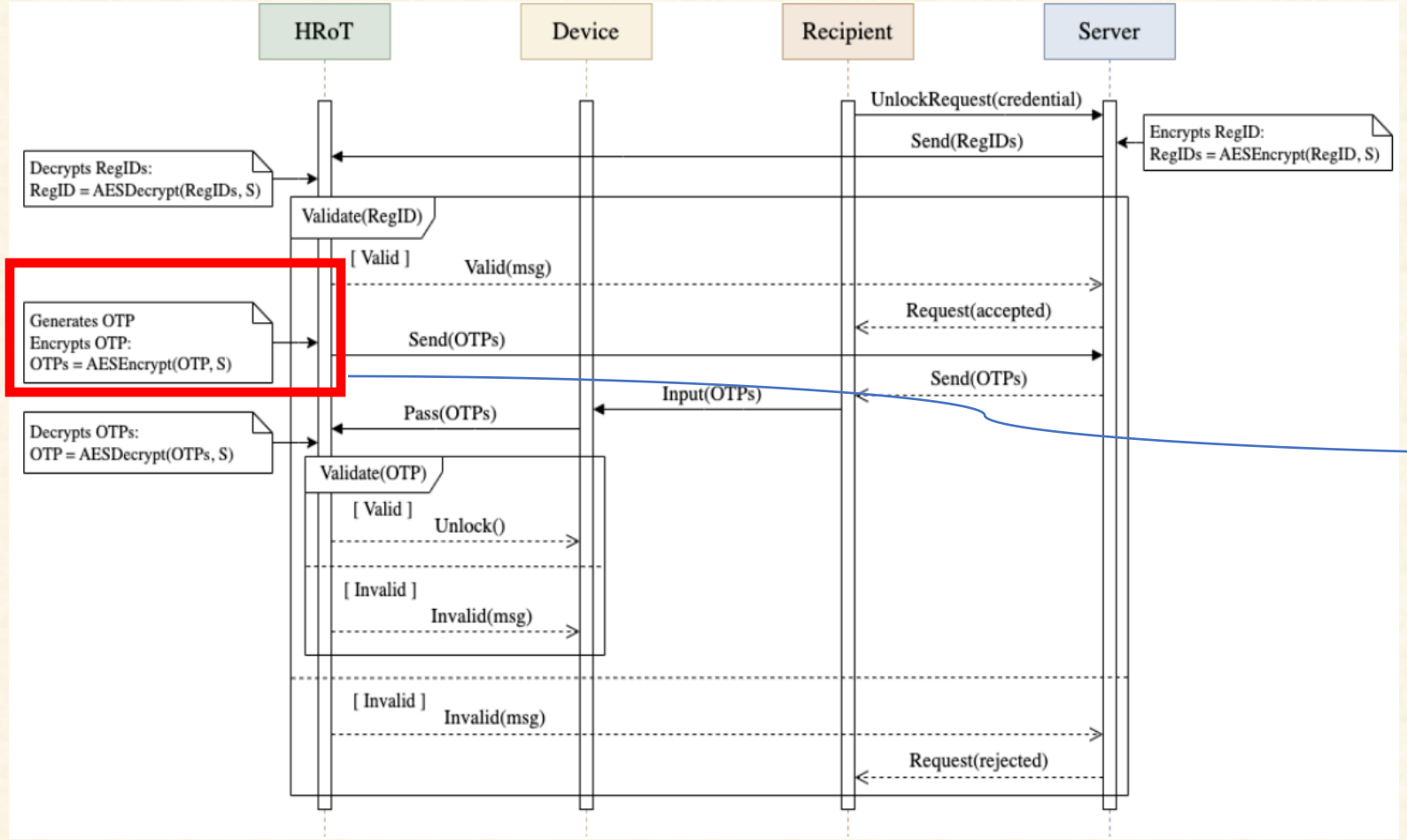
Unlocking Phase (8)



Server sends a “Request Rejected” message to the Recipient if the RegID is not valid.

.

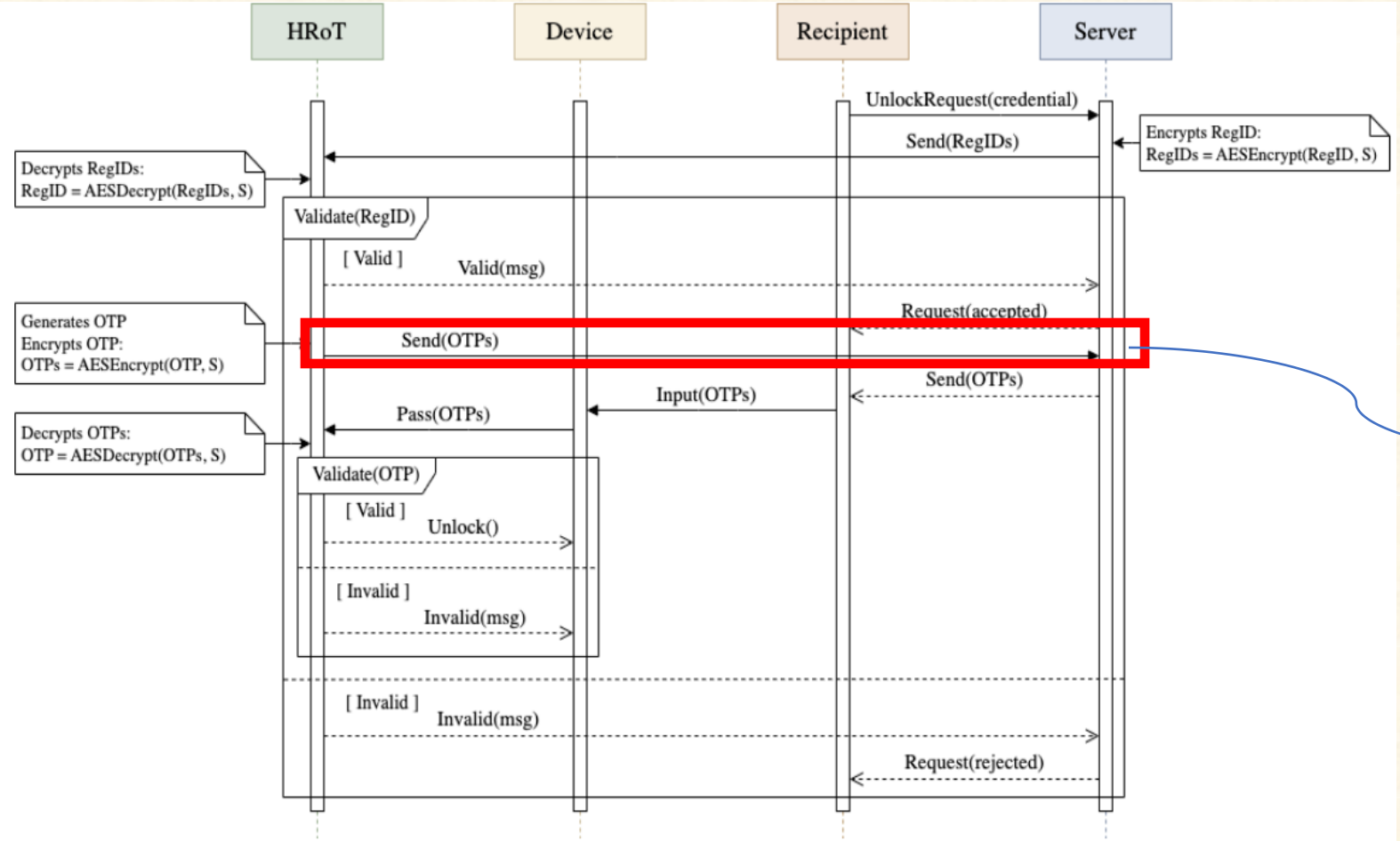
Unlocking Phase (9)



HRoT generates and encrypts the OTP

.

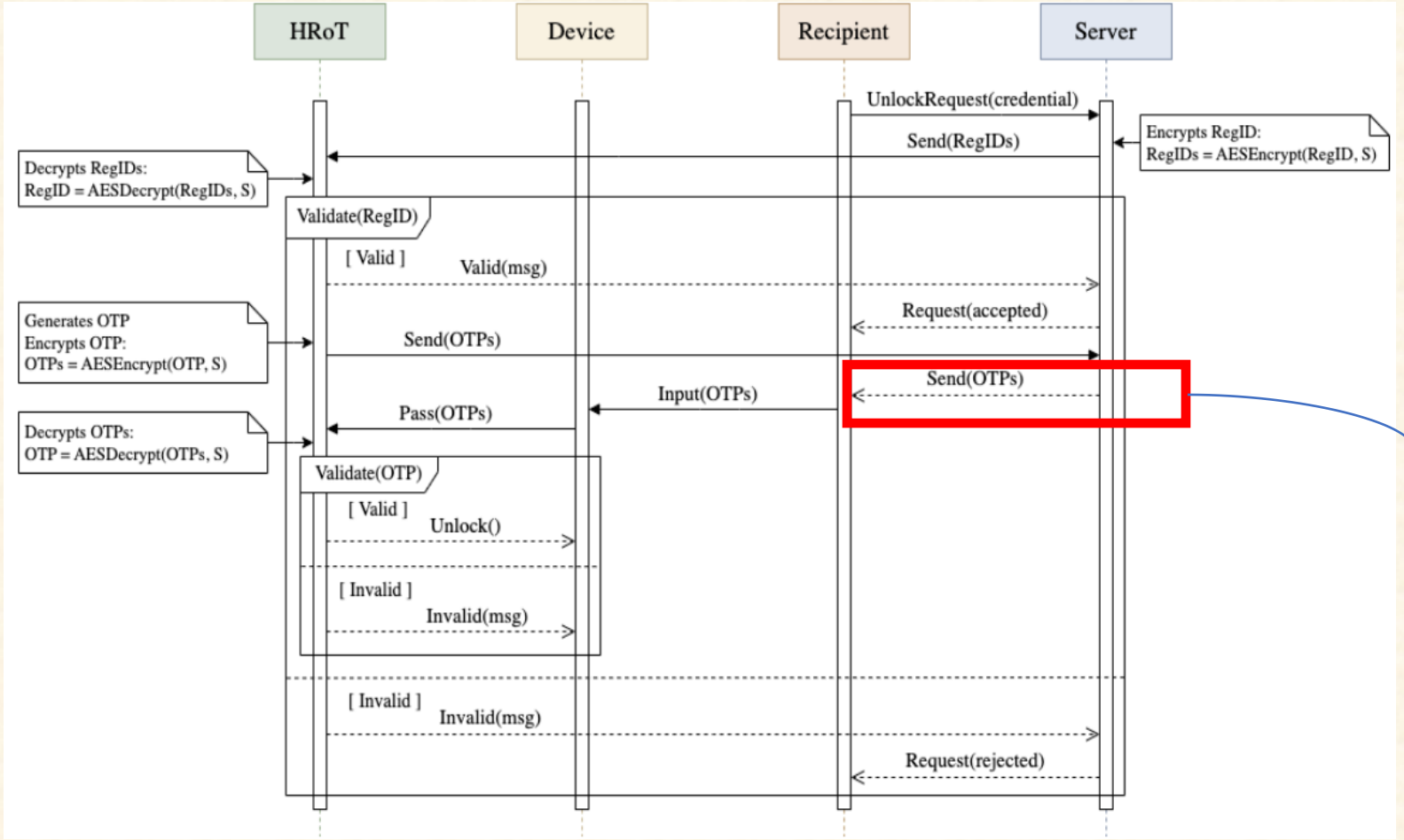
Unlocking Phase (10)



HRoT sends the OTP to the Server

.

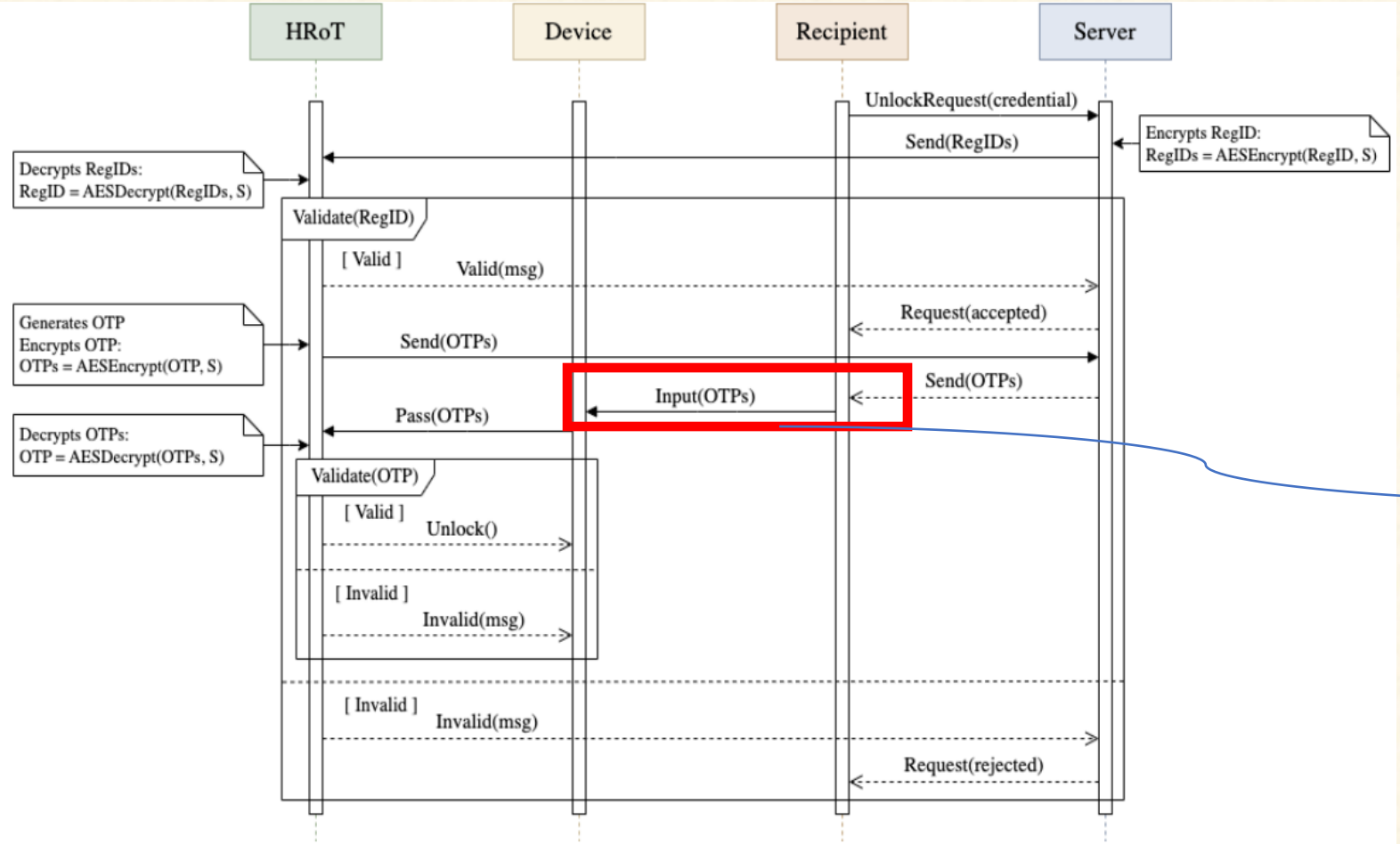
Unlocking Phase (11)



Server sends the OTP to the Recipient

.

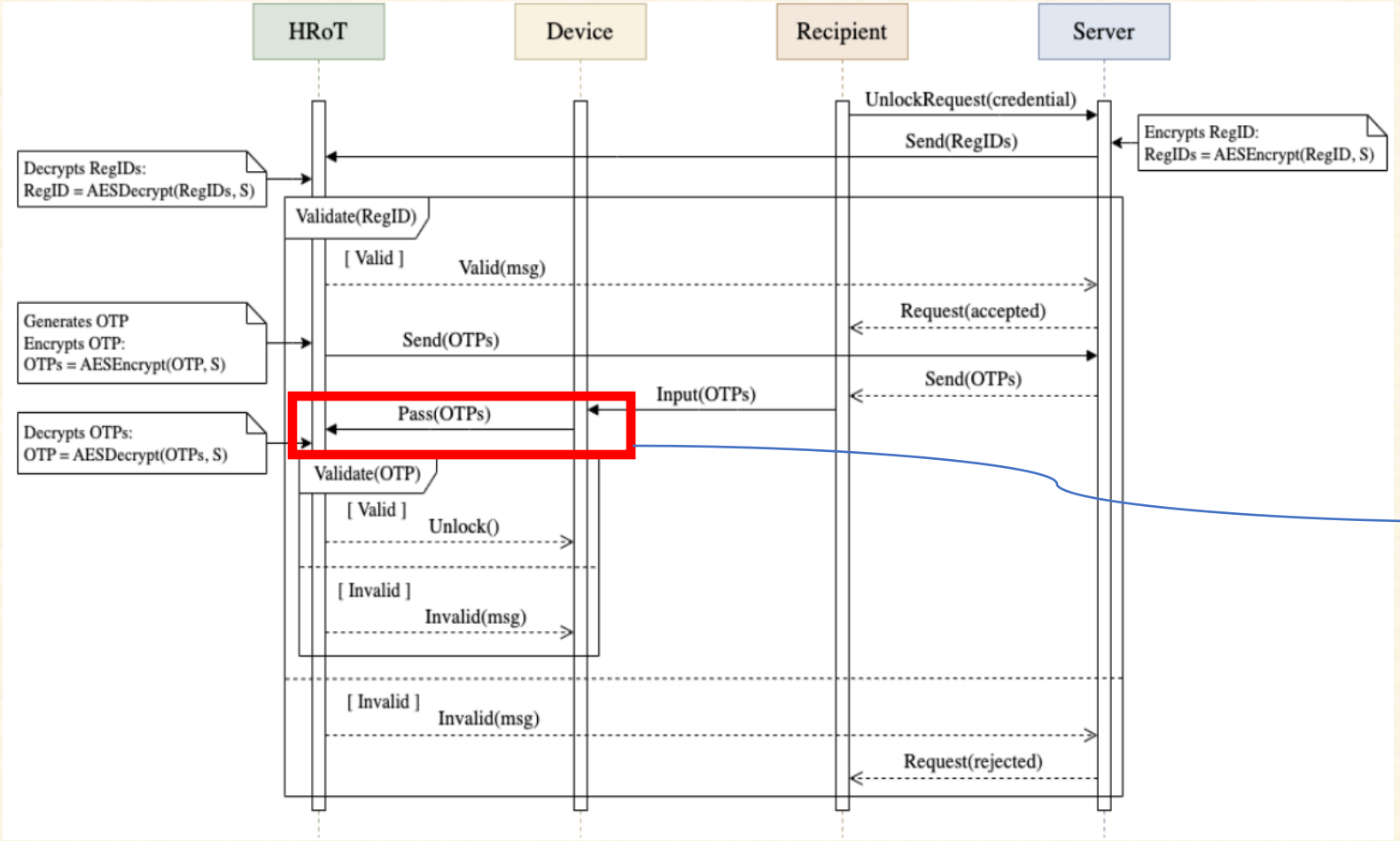
Unlocking Phase (12)



Recipient inputs the OTP into the Device

.

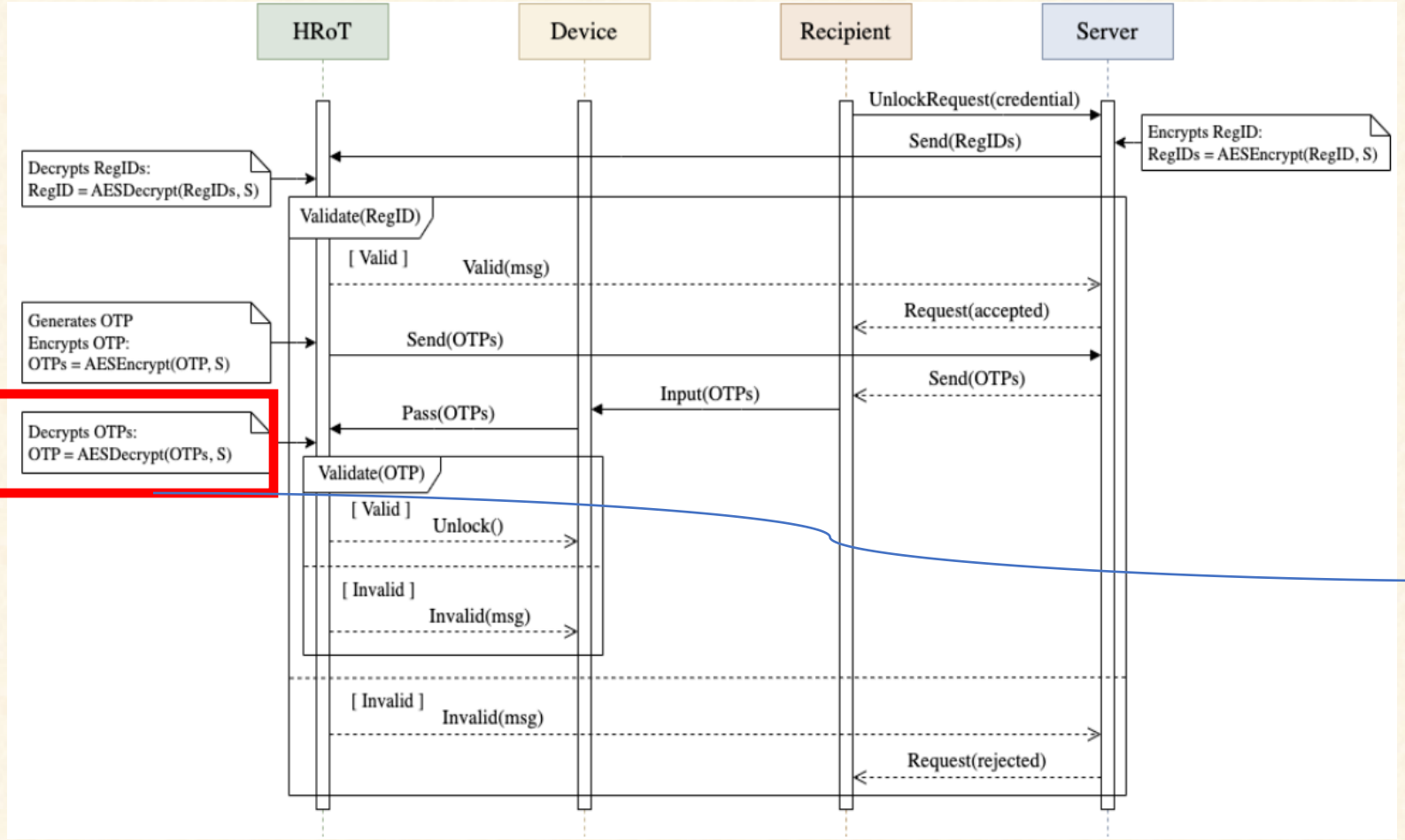
Unlocking Phase (13)



Device passes the OTP to HRoT for verification

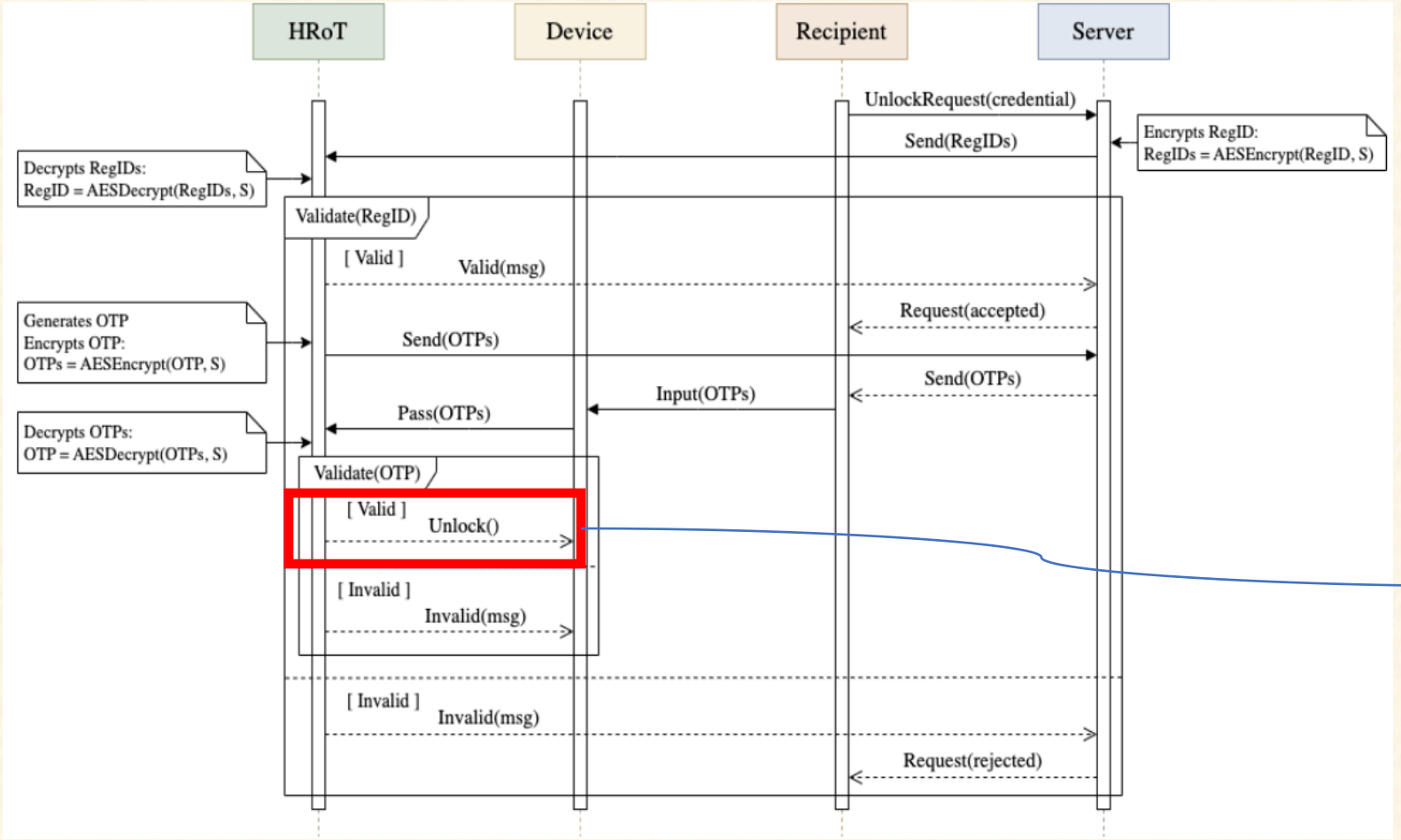
.

Unlocking Phase (14)



HRoT decrypts the OTP

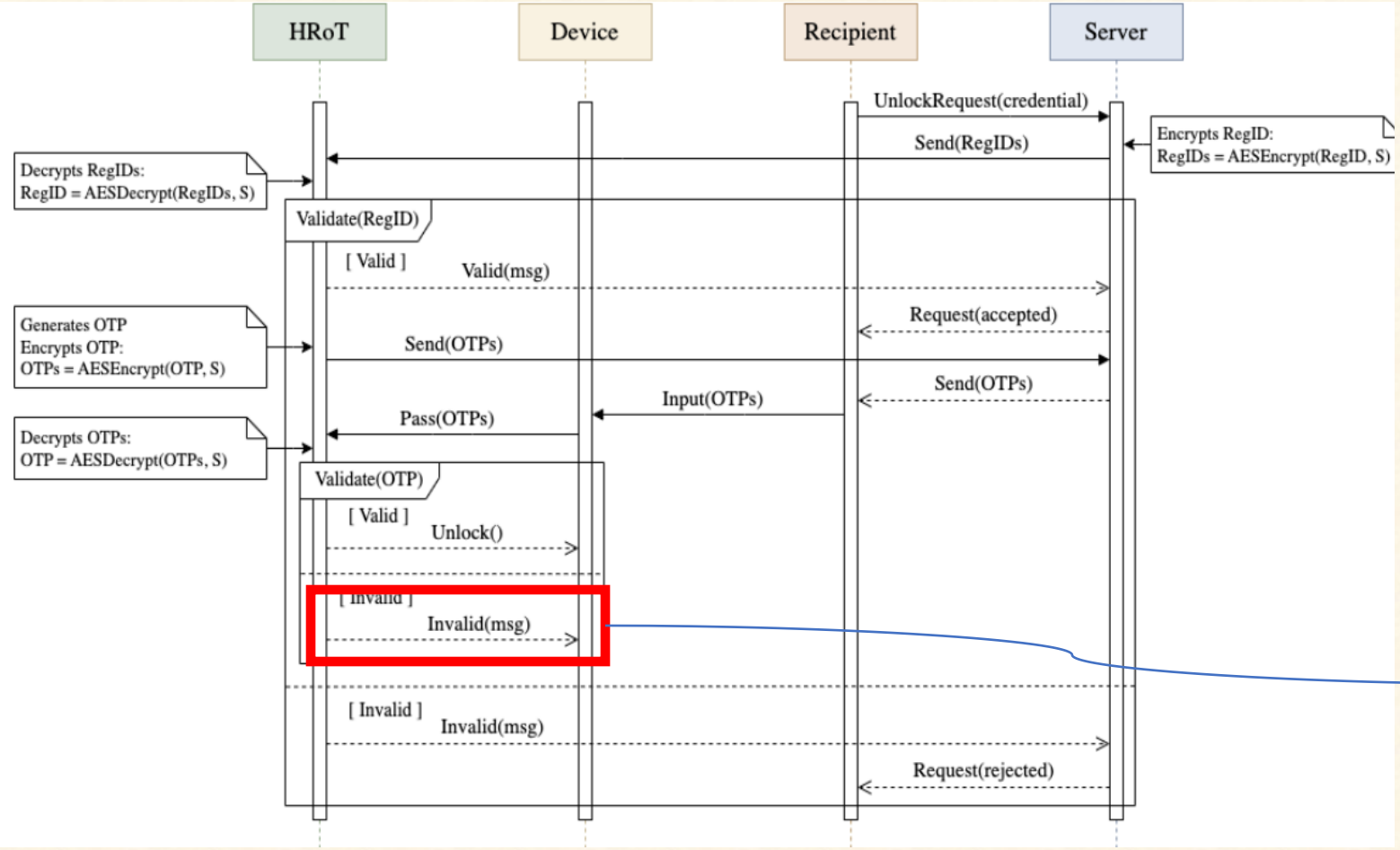
Unlocking Phase (15)



HRoT sends an “Unlock” message to the Device if the OTP is valid

.

Unlocking Phase (16)

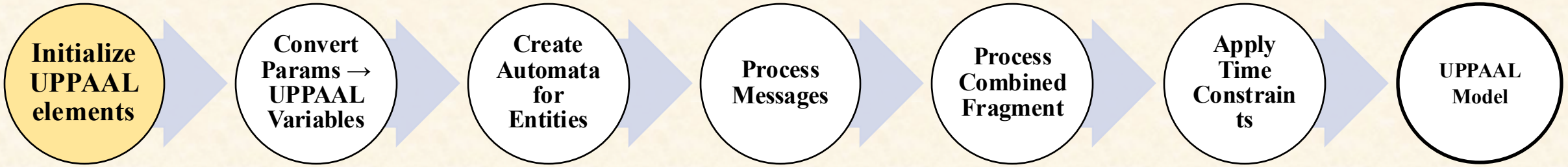


HRoT sends an “Invalid” message to the Device if the OTP is not valid

.

Sequence Diagram to UPPAAL (SD2UPPAAL)

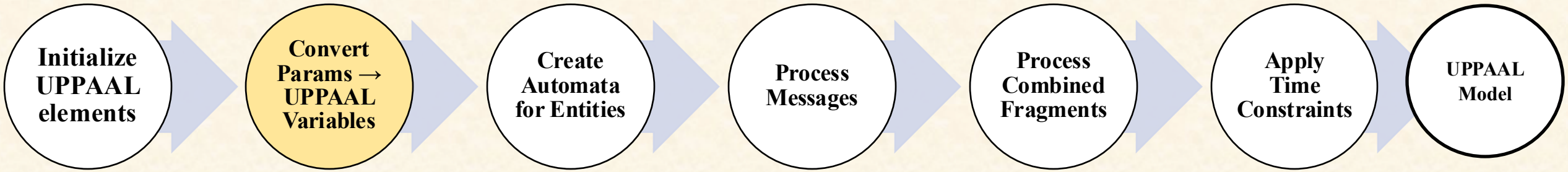
SD2UPPAAL Overview (1)



→ Create empty sets for locations, variables, clocks, actions, edges, and invariants.

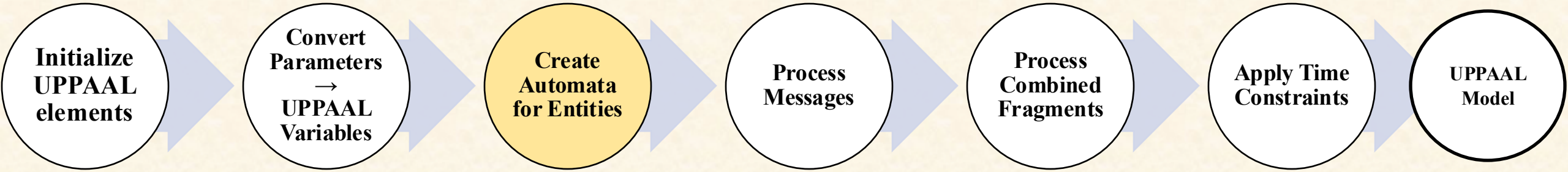
→ **Location** = a *node* in one automaton

SD2UPPAAL Overview(2)



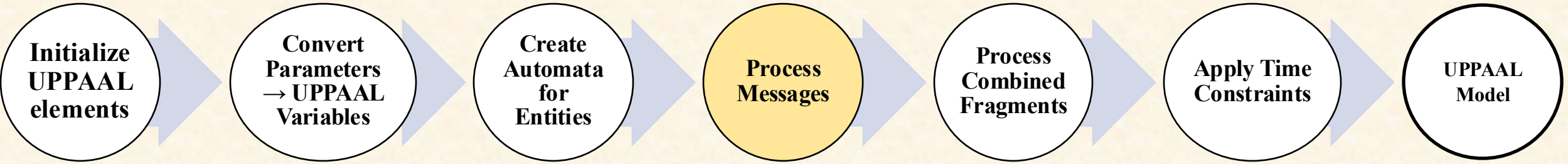
→ For each parameter in the UML Sequence Diagram, create a corresponding variable, assign initial values

SD2UPPAAL Overview (3)



→ Insert `globalClock` to track time-dependent behaviors.

SD2UPPAAL Overview (4)



→ Each UML entity (e.g., HRoT, Server, Sender) becomes a UPPAAL template with an initial "Idle" state.

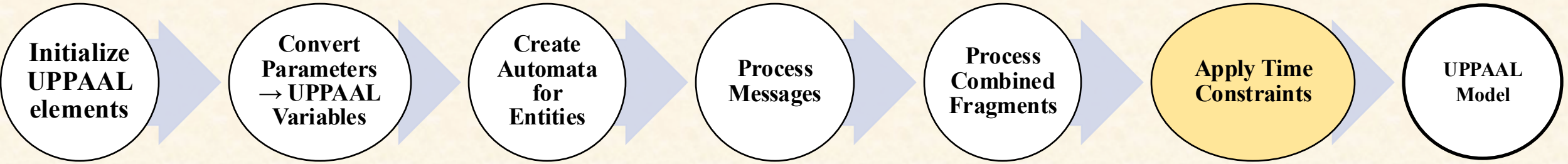
→ **State** = a *snapshot* of the entire system (all automata locations + clock/variable valuations).

SD2UPPAAL Overview (5)

→ For each message:

- Identify sender & receiver
- Create send/receive actions (`msg!`, `msg?`) for synchronous; send action + clock for asynchronous
- Add edges with guards, actions, and clock resets

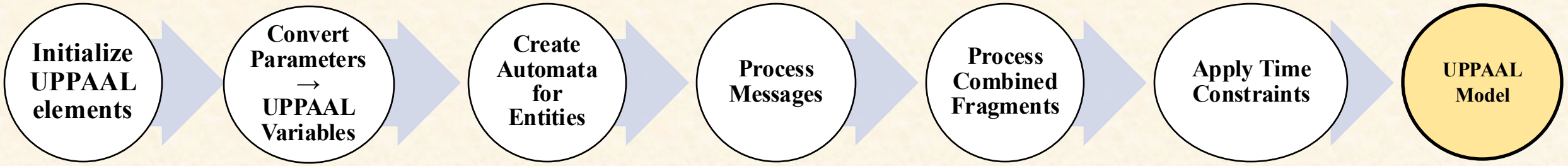
SD2UPPAAL Overview (6)



→ Add invariants to locations to enforce timing conditions

SD2UPPAAL Overview (7)

High Level Process of SD2UPPAAL Algorithm



Outcome: A structurally and behaviorally equivalent UPPAAL model ready for formal verification.

Example of the SD2UPPAAL: Locking Phase of PIT Protocol

From UML to UPPAAL

- **UML Entities:** `Sender`, `Server`, `HRoT`
- **UML Messages:**
  - `ProductRegistration(Info)` — Sender → Server
  - `Send(RegID)` — Server → HRoT
  - `SendPublicKey(qA)` — HRoT → Server
  - `SendPublicKey(qB)` — Server → HRoT

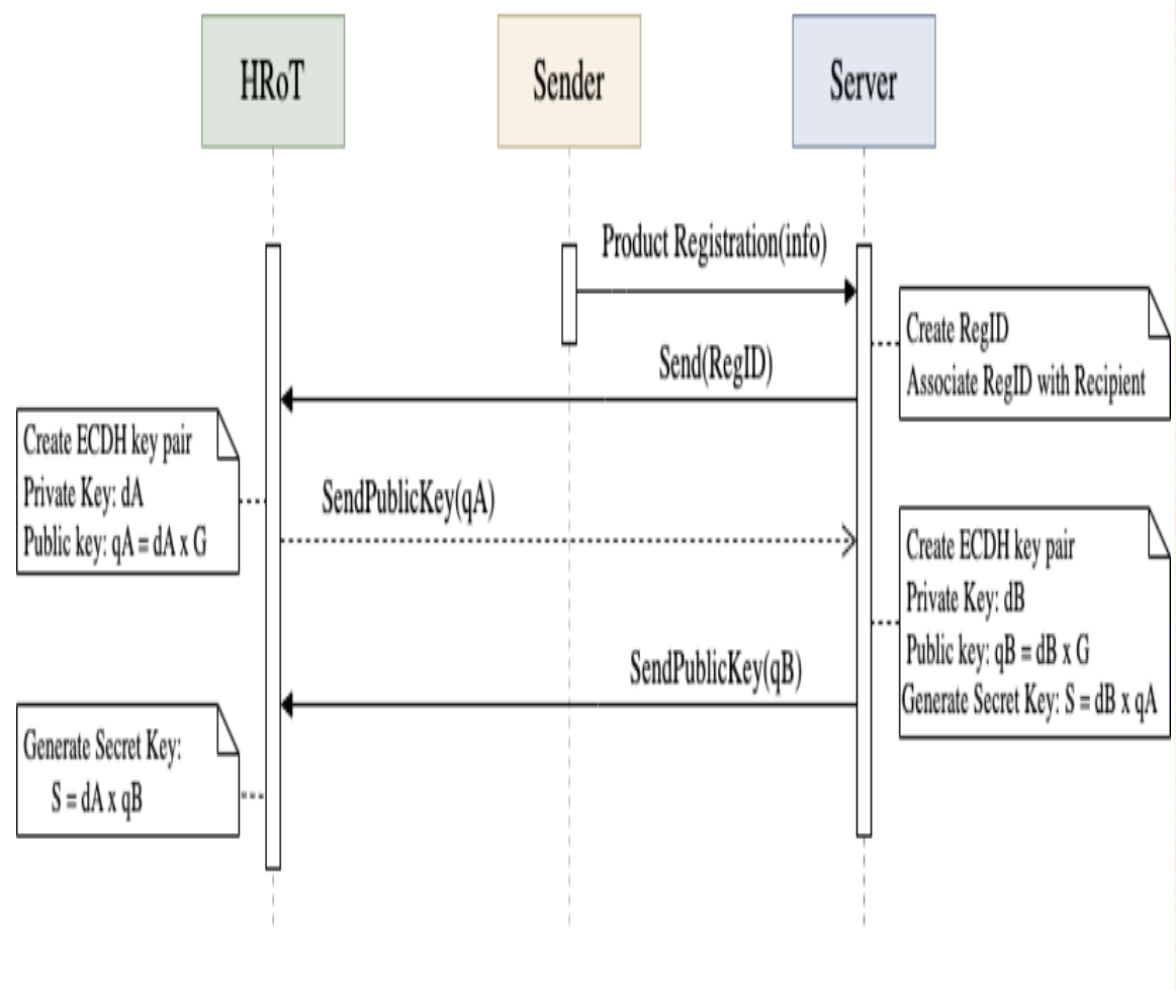
Success state is a terminal location where the system can remain indefinitely, signifying completion of the modeled behavior with no further transitions required

UPPAAL Transformation (Algorithm 1 Applied)

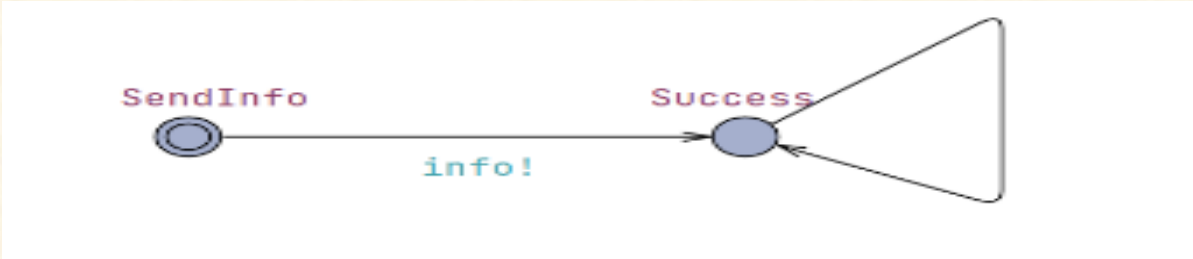
- **Templates:** One automaton per entity.
- **States:**
  - Sender: `IdleSender` ⟶ sends `info!` ⟶ `SuccessSender`
  - Server: `IdleServer` ⟶ receives `info?` ⟶ sends `regid!` ⟶ waits for `qa?` ⟶ sends `qb!` ⟶ `SuccessServer`
  - HRoT: `IdleHRoT` ⟶ receives `regid?` ⟶ sends `qa!` ⟶ receives `qb?` ⟶ `SuccessHRoT`
- **Actions:** Synchronous message channels (`info!` / `info?`, `regid!` / `regid?`, etc.) ensure correct ordering.
- **Clocks & Invariants:** Used if timing constraints exist for key exchanges.

Locking Phase Transformation

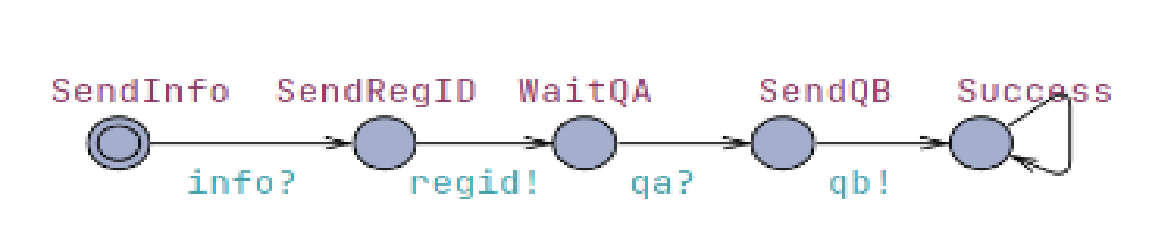
Sequence Diagram for Locking Phase



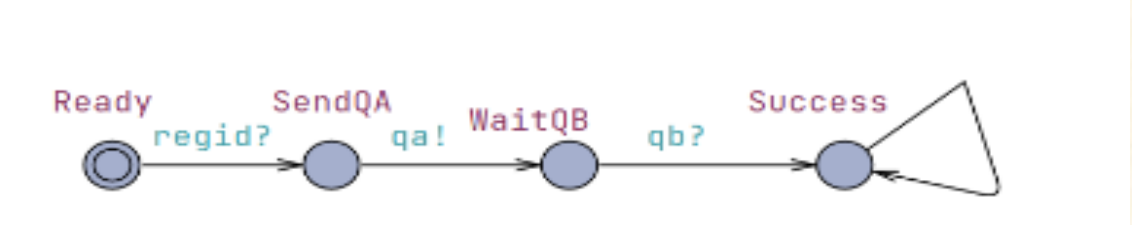
UPPAAL Automatas for Locking Phase



Sender

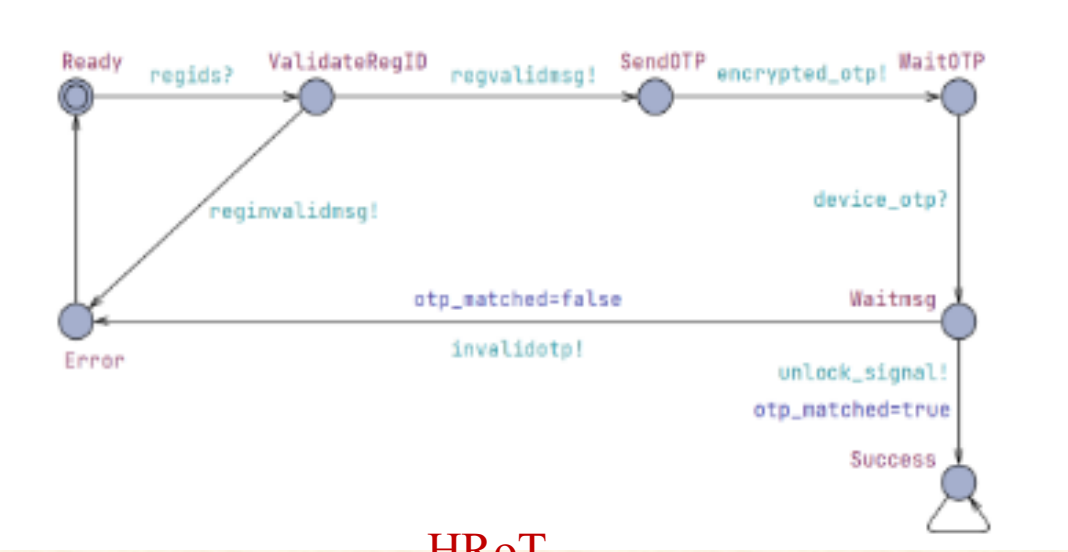


Server

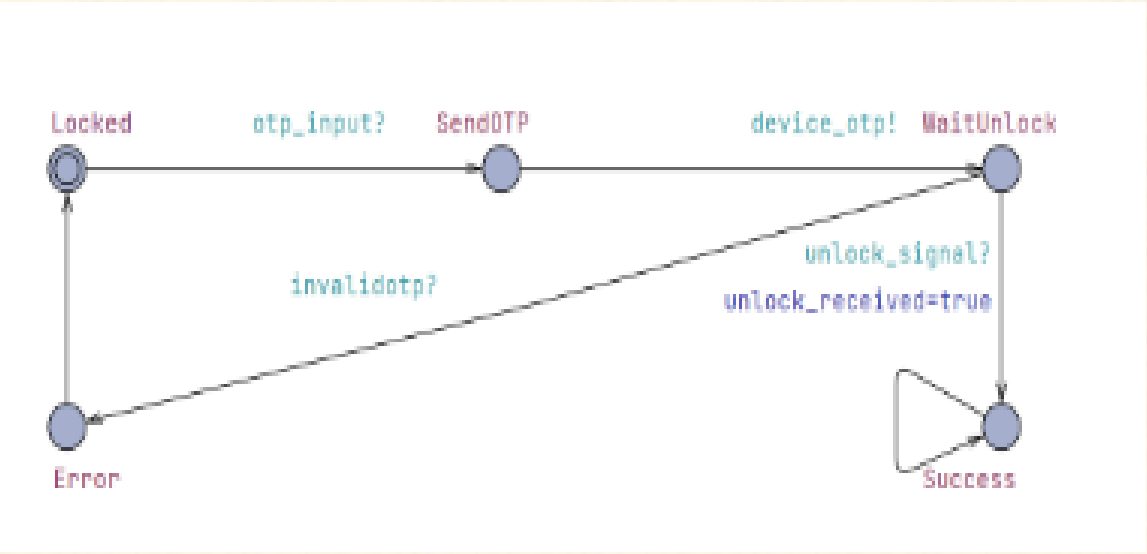


HRoT

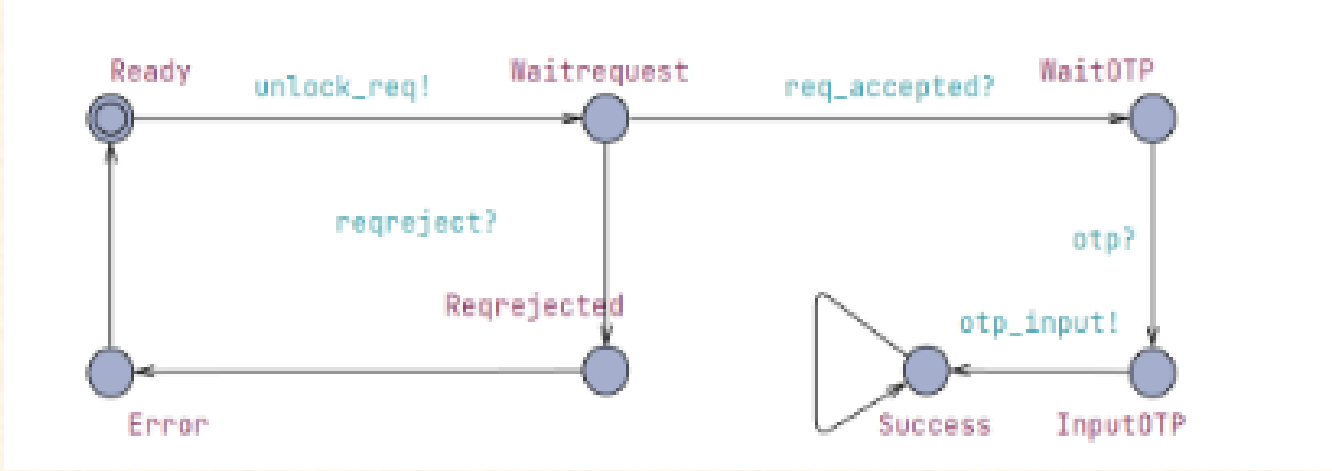
Unlocking Phase UPPAAL Diagrams



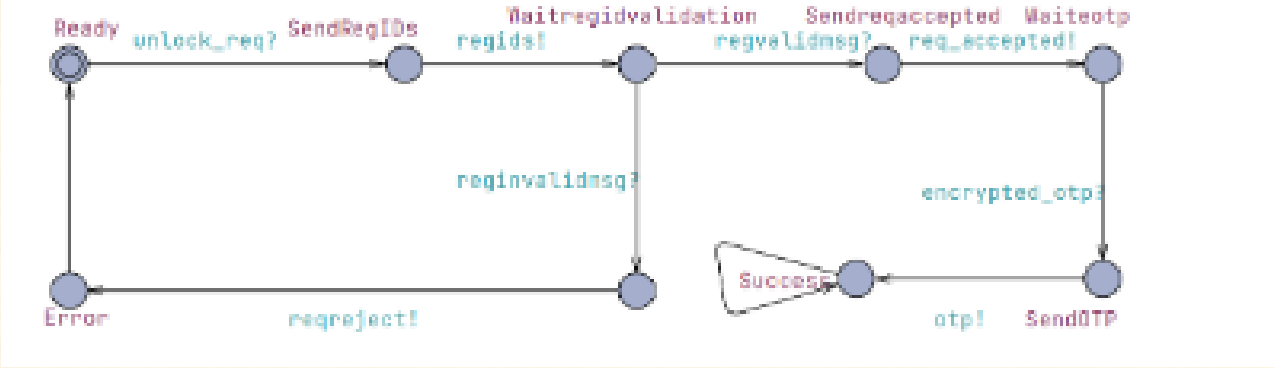
HRoT



Device



Recipient



Server

51

Properties in UPPAAL

Property Specification in UPPAAL

- Deadlock Detection
  - A[] not deadlock
  - There is no state where the system gets stuck
- Safety Properties
  - A[]ϕ
  - Something bad never happens
- Liveness Properties
  - E <> ϕ
  - Something good will eventually happen
- Conditional Properties
  - $\Phi \rightarrow \Psi$
  - Whenever Φ happens, Ψ will eventually happen

Reachability

- All components successfully complete the locking phase
  - E<> HRoTT.Success (received the public key qb) && ServerT.Success (finished sending the public key qb) && SenderT.Success (finished sending the device info)

- All components successfully complete the unlocking phase
  - E<> ServerT.Success (finished sending OTP) && HRoTT.Success (OTP verified successfully) && DeviceT.Success (device unlocked) && RecipientT.Success (recipient accessed the device)

Safety Properties

- Always Device unlocks if HRoT completes successfully (OTP valid)
  - A[] DeviceT.Success imply HRoTT.Success

- Always if HRoT completes successfully (OTP valid), the device is unlocked
  - A[] HRoTT.Success imply DeviceT.Success

Safety Properties Continued

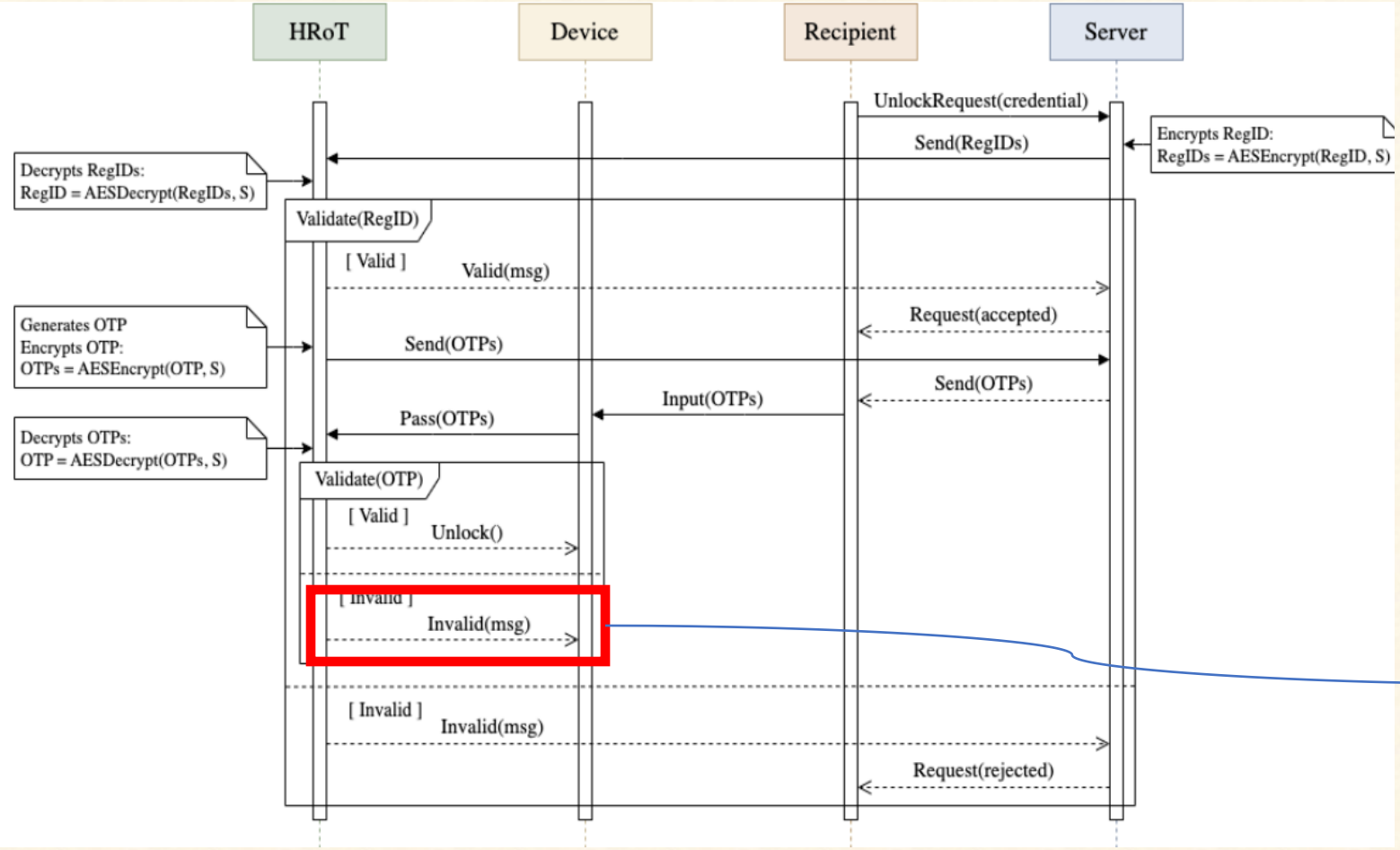
- Always Recipient cannot reach success if the Device is still locked
  - A[] DeviceT.Locked imply not RecipientT.Success
- Retry is always allowed on failure
  - A[] HRoTT.Error imply not RecipientT.Success
- Server can reach success only if the device is unlocked
  - A[] DeviceT.Locked imply not ServerT.Success
- Recipient is always allowed to make another unlock request if OTP input fails
  - A[] HRoTT.Error (OTP not correct) imply not RecipientT.Success

Analysis Results

Violated Properties

- *Recipient cannot succeed if Device is locked*
  - Recipient can reach success while Device remains locked
- *Server succeeds only if Device is unlocked*
  - Server may succeed without actual unlock
- *Allow retry on failure*
  - Lack of proper error handling after invalid OTP

Root Cause of Error



"Invalid" message if the OTP is not valid is not communicated to the Recipient or Server

HRoT sends an "Invalid" message to the Device if the OTP is not valid

Protocol Refinement

Introducing Additional Steps

- Version 1:
  - HRoT → Device: Valid/Invalid OTP
- Version 2:
  - HRoT → Server: Valid/Invalid OTP
  - Server → Recipient: Unlock/Invalid Signal
  - HRoT → Device: Unlock/Invalid OTP
- Effects:
  - Ensures the Device unlocks only after successful OTP verification
  - Enables retry upon failure
  - Synchronizes state across all entities
  - All properties now verified

Conclusion and Future Work

- Contributions
  - Developed SD2UPPAAL algorithm for transforming UML Sequence Diagram to UPPAAL
  - Formally verified PIT protocol and found errors
  - Provided a new version of the PIT protocol
- Future Work
  - Address scalability issues of formal verification
  - Investigate the use of AI planners for protocol verification and analysis

Thank you