Ghost of Sessions Past: Distributed and Forward Secure Key Establishment for Implantable Medical Devices

Roozbeh Sarenche[1], Sayon Duttagupta[1], Kevin Bogner[1], Varesh Mishra[1], Francesco Milizia[2], and Bart Preneel[1]

[1] COSIC, KU Leuven, Leuven, Belgium
firstname.lastname@esat.kuleuven.be
[2] Sapienza University of Rome, Rome, Italy
francesco.milizia@protonmail.com

Abstract. Implantable Medical Devices (IMDs) operate for decades in dynamic and adversarial environments, where device loss, backend compromise, and eventual post-explantation access are realistic long-term threats. Existing IMD security mechanisms largely focus on secure pairing and access control, but rely on long-lived secrets. As a result, a compromise occurring years after deployment can retroactively expose previously recorded patient telemetry. Limiting such damage requires forward secrecy. However, achieving forward secrecy in IMDs is challenging due to strict energy constraints, intermittent connectivity, and safety requirements, which make frequent public-key operations on the implant impractical. We present *Chronos*, a distributed, symmetric-only key establishment protocol that provides forward secrecy for IMDs without requiring public-key cryptography on the implant. *Chronos* evolves cryptographic state across sessions using lightweight primitives on the device, ensuring that past communications remain confidential even under post-compromise exposure. State consistency, resynchronisation, and recovery are coordinated by a threshold-based distributed backend, eliminating single points of failure and enabling controlled emergency operation when the user's device is unavailable, while preserving patient-centric access control. We formally analyse *Chronos* using ProVerif under a Dolev–Yao adversary with post-session compromise capabilities, establishing secrecy, agreement, forward secrecy, and correctness of emergency and recovery mechanisms. We implement *Chronos* end-to-end, including an Android client, distributed backend services, and implant-side firmware on both a 16-bit MSP430 and a 32-bit ARM Cortex-M33 microcontroller. Our evaluation shows that forward secrecy with recovery from state inconsistency is achievable within realistic IMD constraints, with negligible impact on device energy budgets even under pessimistic usage assumptions.

1 Introduction

Implantable Medical Devices (IMDs), such as pacemakers, cardiac defibrillators, and neurostimulators, operate for years inside the human body while support-

ing wireless access for clinical follow-up, configuration, monitoring, and emergency care. This creates an unusually demanding security setting. The implant is power-constrained and safety-critical, yet the ecosystem around it is long-lived and constantly changing: patients replace phones, clinician programmers are shared across sites, hospital infrastructure may be compromised, and devices may eventually be explanted and analysed. Prior work has repeatedly shown that these threats are realistic, from software radio attacks and low power defences [9], to attacks on cardiac and neurostimulation systems [14,16], and practical attacks on insulin therapy systems [12].

Most IMD security mechanisms focus on the moment of access: deciding whether an external device should be allowed to communicate with the implant now. Shielding approaches protect the physical channel [8,28]; physiological keying derives secrets from the body [10,22]; and delegation-based systems rely on companion devices such as phones [7]. These designs address important clinical constraints, especially emergency access and usability [3, 4]. However, they do not directly address a different long-term risk: *retrospective compromise*. If an adversary records IMD traffic today and obtains long-lived secrets years later, previously recorded telemetry or commands may become exposed. Physical and physiological assumptions may also erode over time, as shown by stronger receiver models against friendly jamming [27], remote recovery of heartbeat signals using cameras [25], and the instability and limited entropy of physiological features [13].

This paper argues that IMD security should be treated as a *session lifecycle* problem rather than only a pairing or access-control problem. The central question is not only whether a doctor may access an implant today, but whether yesterday's sessions remain confidential after tomorrow's compromise. This calls for forward secrecy [5]. At the same time, classical public-key key agreement is unattractive for frequent implant-side use: IMDs have strict energy budgets, limited computation, intermittent connectivity, and safety requirements that make permanent lockout unacceptable. A practical design must therefore provide forward secrecy without making the implant depend on expensive public-key operations, continuous connectivity, or a single trusted companion device.

We present *Chronos*, a distributed, symmetric-only key establishment protocol for IMDs. In *Chronos*, the implant performs only symmetric cryptographic operations on the critical path. Forward secrecy is achieved by evolving symmetric state across sessions, while state consistency, resynchronisation, and recovery are coordinated by a threshold set of backend servers. Building on the state-evolved key-establishment idea of SAKE [1], *Chronos* adapts this direction to the IMD setting, where failure must be safe, emergency access must remain possible, and no single phone, clinician device, or backend component should become a permanent point of failure.

The key design insight is that implant-side public-key cryptography is not necessary for forward secrecy if state evolution is cryptographically enforced and externally recoverable. *Chronos* splits this responsibility across a distributed backend: the implant remains lightweight, while a threshold of servers maintains

recovery material and coordinates state advancement. This allows *Chronos* to tolerate message loss, backend inconsistency, partial backend compromise, and emergency access when the user's device is unavailable. Outside emergencies, the user remains in the loop: access requires a user-side contribution, preserving patient agency rather than delegating permanent authority to the backend. This design is consistent with prior observations that IMD security must account not only for cryptographic strength, but also for clinical safety, usability, and human values [3, 4, 21].

We implement and evaluate *Chronos* as a complete system. The implementation includes an Android client, firmware for a 16-bit MSP430 and a 32-bit ARM Cortex-M33 microcontroller, and a distributed backend. Our measurements show that the additional cryptographic protection is practical even on constrained hardware. In particular, the implant-side cost remains small relative to realistic IMD battery budgets, demonstrating that forward secrecy and recovery from state inconsistency can be achieved without sacrificing long-term device lifetime.

Contributions: To summarise, this paper makes the following contributions.

(C1) A symmetric-only forward-secure protocol for IMDs. We design *Chronos*, a lightweight state-evolving protocol that provides forward secrecy for implantable medical devices while requiring only symmetric cryptographic operations on the implant. Past sessions remain confidential even if the IMD, associated client devices, and backend state are compromised later, subject to the stated availability threshold assumption. The protocol also provides session-key secrecy against an active network adversary.

(C2) Distributed state evolution, resynchronisation, and emergency access. We introduce a distributed, threshold-coordinated key-server backend that keeps implant and server state consistent despite message loss, intermittent connectivity, and partial backend unavailability. Additionally, we formalise the backend properties required for this consistency and prove that any implant–backend desynchronisation remains bounded and recoverable (Appendices 12 and 14). This distributed structure prevents any single backend key server from becoming a permanent point of failure and enables break-the-glass access when the user's mobile device is unavailable, while preserving the implant's symmetric-only execution model.

(C3) End-to-end implementation and practical evaluation. We implement *Chronos* as a complete end-to-end system, including an Android client application, a distributed backend, and implant-side firmware on both a 16-bit MSP430 and a 32-bit ARM Cortex-M33 microcontroller. Our evaluation measures protocol latency, implant-side cost, and long-term energy impact, showing that the added forward-secrecy and recovery mechanisms impose negligible battery overhead under realistic and pessimistic usage assumptions.

(C4) Formal verification under network control and post-compromise exposure. We formally analyse *Chronos* in ProVerif under a Dolev–Yao adversary with full network control and post-session compromise capabilities. The

model verifies agreement, session-key secrecy, forward secrecy, user-consent binding, emergency-mode correctness, and additional state-resynchronisation invariants in the presence of state evolution and threshold recovery.

Artefact Availability: The entire source code and artefact underlying this paper, can be found at `https://anonymous.4open.science/r/Chronos-IMD/`.

2 System Setting and Design Requirements

This section outlines the system settings and constraints that drive the design of *Chronos*. We focus only on aspects that directly impact the security model, in particular, the need to balance long-term confidentiality with safety-critical availability.

2.1 IMD Ecosystem

We consider an IMD ecosystem consisting of four entities: (i) an implantable device (IMD), (ii) external client devices, (iii) a clinical backend, and (iv) a trusted authority for initial provisioning.

The IMD communicates wirelessly for telemetry retrieval, configuration, and therapy-related operations. In practice, this communication is realised via short-range radios such as BLE or NFC. In our prototype, we use BLE as a representative low-power interface. External clients include a patient smartphone, used for consent and routine access, and clinician devices for medical interactions. The backend represents one or more clinical operators responsible for authorisation, auditing, and recovery support.

This ecosystem is long-lived and dynamic: devices are replaced, multiple clinicians interact with the implant over time, and backend services may be partially unavailable. The protocol must remain secure and operational under these conditions.

2.2 Operational and Safety Requirements

IMD interactions differ in urgency and clinical risk, but share a common requirement: security must not compromise patient safety.

We distinguish between routine and non-urgent operations, which require strong confidentiality and authentication, and emergency operations, which must remain possible even under degraded conditions. In particular, emergency access must not depend on a single external artefact such as a patient's phone or a single backend service.

We formalise the following safety requirements:

- **No permanent lockout.** The IMD must remain accessible even if external devices are unavailable or the cryptographic state becomes inconsistent.
- **Recoverability.** The system must support recovery from state desynchronisation without invasive procedures.
- **Controlled emergency access.** Emergency operations must be possible in a controlled and auditable manner, even without patient-side participation.

These requirements imply that the protocol must tolerate partial backend failures and avoid single points of failure, while still preserving confidentiality and authentication guarantees. This motivates distributing trust across multiple administrative domains, making collusion and simultaneous compromise of all critical components operationally non-trivial.

2.3 System Constraints

The protocol design is shaped by the following constraints:

Energy and computation. IMDs operate under strict power budgets over multi-year lifetimes. Expensive operations, in particular public-key cryptography, cannot be executed frequently on the implant, motivating a symmetric-only critical path on the implant.

Intermittent and lossy communication. Wireless links are unreliable due to body effects, interference, and mobility. Message loss, delay, and reordering must be treated as normal. Protocol state must therefore tolerate desynchronisation and support safe recovery.

Device churn and multi-client access. Patients may replace devices, and multiple clinicians may require access over time. The protocol must support changing external devices without reinitialising the implant or introducing a permanent trust anchor.

Partial backend availability. Backend services may be temporarily unavailable due to failures or security incidents. Relying on a single backend entity would introduce a safety-critical point of failure.

Patient agency and auditability. Outside emergencies, the patient should retain control over access via their personal device. At the same time, sensitive operations, especially emergency access, must be auditable.

2.4 Design Implications

The above setting leads to two key design implications. First, protecting long-term confidentiality requires forward secrecy under realistic post-compromise scenarios, yet the implant must avoid frequent public-key operations due to energy constraints.

Second, any state-evolving design must remain robust under system failures. In particular, it must tolerate backend unavailability, client device loss, and transient state inconsistencies without causing permanent lockout or preventing emergency access. *Chronos* is designed to satisfy these requirements by combining symmetric state evolution with threshold-assisted recovery.

3 System, Trust and Threat Model

In this section, we first describe the system architecture and the types of entities present in the system. Next, we discuss the capabilities of the assumed adversary and the associated security goals.

3.1 System and Trust Model

This subsection introduces our entities, as presented in Fig. 1.

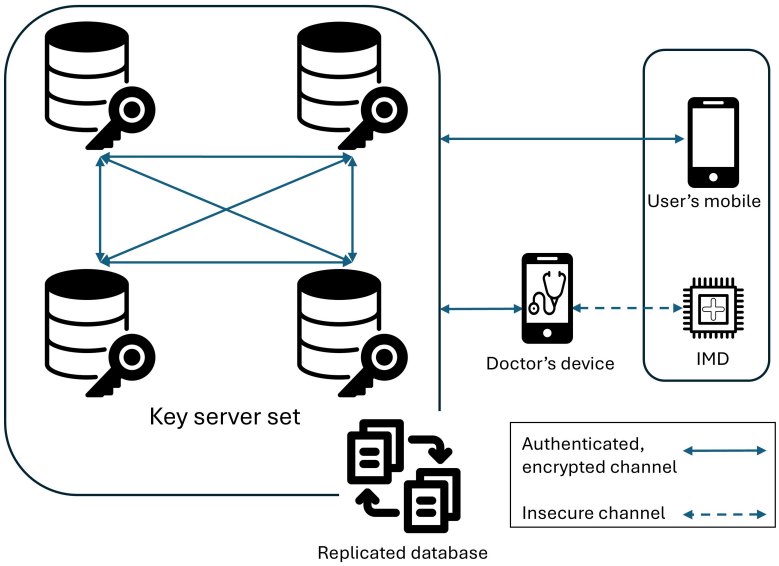


Fig. 1: System model overview.

(E1) Trusted Authority ($\mathcal{TA}$): The $\mathcal{TA}$ participates only in system bootstrapping and entity registration. It generates system parameters, issues initial key material for sensors, distributes secret shares of this material, and certifies the public keys of mobile devices and key servers. We assume that $\mathcal{TA}$ is fully trusted and uncompromised during key provisioning. After provisioning, $\mathcal{TA}$ is not involved in any subsequent key-derivation protocol execution. Certificate revocation may also involve $\mathcal{TA}$, but is outside the scope of this protocol.

(E2) Implantable Medical Device ($\mathcal{S}_i$): The sensor $\mathcal{S}_i$ stores its local key material and performs a lightweight symmetric protocol for key update and session key establishment.

(E3) Key Servers ($\mathcal{KS}$): The key servers maintain per-sensor state and assist in the key-establishment protocol between a clinician device and a sensor. Let n be the number of servers; $\mathcal{KS}$ denotes the set of all n servers, and KS_j denotes server j. Reconstructing any sensor state requires combining shares from at least t out of n servers.

(E4) User's Mobile Device (M_i^{user})**.** The protocol assumes that each individual with an IMD possesses a personal mobile device, denoted M_i^{user}, corresponding to the IMD $\mathcal{S}_i$. The user's mobile device enables the user to explicitly consent to or deny session-key establishment requests initiated by a doctor's device for the corresponding IMD. It stores the cryptographic material required for the user-side contribution to session-key derivation. The protocol further accounts for scenarios in which M_i^{user} is unavailable or unreachable and addresses them through a dedicated emergency-access mechanism.

(E5) Doctor's Device (D_{Dr})**.** The doctor's device D_{Dr} facilitates secure session establishment between a doctor and an IMD. It may initiate a session-key establishment request only if it holds a valid certificate issued by the trusted authority $\mathcal{TA}$ attesting to the doctor's authorisation. The key servers authenticate D_{Dr} via its certified public key.
(E6) Replicated Database/Ledger ($\mathsf{DB}_{\mathrm{rep}}$)**.** The $\mathsf{DB}_{\mathrm{rep}}$ is a replicated, append-only log maintained by the key servers. It records *committed* sensor state-update events, ensuring that all online servers derive a consistent view of each sensor's current state. An entry is committed once securely replicated to a majority of servers, after which it is final and irreversible. The majority-based commit rule of DBrep enables state synchronisation between sensors and the key-server infrastructure. A detailed discussion of the ledger requirements and a suitable realisation is given in Appendix 12.

3.2 Adversary Model and Assumptions

We analyse our protocol against an active network adversary $\mathcal{A}$ operating in the Dolev–Yao model [6]. $\mathcal{A}$ controls all communication between D_{Dr}, $\mathcal{S}_i$, M_i^{user} and all the key servers $\{\mathcal{KS}_j\}_{j\in[n]}$. It can eavesdrop on, block, delay, replay, reorder, and inject arbitrary messages exchanged between any system entities. Additionally, we make the following assumptions:

(A1) Secure Key Provisioning. We assume that the trusted authority $\mathcal{TA}$ remains uncompromised during the key provisioning phase. It performs its operation related to registration and distribution of initial key material such that no cryptographic material is leaked or compromised.
(A2) Availability Threshold & Synchrony. We assume that at any time at least t key servers are online and responsive, where $t \geq \lfloor n/2 \rfloor + 1$. We further assume a synchronous network among the online key servers, i.e., message delays are bounded by a known constant Δ_{delay}.
(A3) ***Honest-but-Curious*** **Key Servers.** We assume that key servers execute the protocol as specified and do not deviate from its prescribed steps, including erasing each sensor's local state upon update. However, up to $(t-1)$ servers may collude by sharing stored states, inputs, and outputs, which is insufficient to reconstruct secret-shared values or the user-side emergency contribution.
(A4) Accountable Emergency Execution Policy. When the user's mobile device M_i^{user} is unavailable, emergency access (i.e., a break-the-glass procedure that authorises a doctor to access a specific sensor without the user's consent) is permitted only under a predefined accountable policy. The key servers release emergency recovery material only for requests that satisfy this policy (e.g., authenticated doctor identity), and such emergency executions are logged to support auditability and post-hoc accountability.
(A5) Hardness of Cryptographic Primitives. We assume that $\mathcal{A}$ cannot break the cryptographic primitives. In particular, digital signatures are existentially unforgeable under adaptively chosen-message attacks (EUF-CMA), Message Authentication Codes (MACs) are unforgeable under adaptively chosen-message attacks, encryption schemes are IND-CPA secure, Key Derivation Func-

tions (KDFs) behave as pseudorandom functions, and one-way function inversion is hard. We further assume that the threshold secret sharing provides information-theoretic secrecy against any adversary controlling fewer than t shares.

It is important to note that we assume IMDs execute the protocol honestly; side-channel attacks on IMDs are out of scope due to their deployment nature. We also do not address radio-layer denial-of-service attacks and exclude them from our analysis.

3.3 Security Goals

We define the security goals of *Chronos* as follows:

(G1) Authenticated Session-Key Establishment. If the sensor $\mathcal{S}_i$ accepts and outputs a session key k_{session} for a session with D_{Dr}, then D_{Dr} has a matching session that outputs the same k_{session}, and $\mathcal{S}_i$ is assured (via backend-mediated authentication) that the peer is an authorised clinician device.
(G2) Session-Key Confidentiality. The established session key k_{session} is known only to the intended entities, namely the sensor $\mathcal{S}_i$ and the doctor's device D_{Dr}. In particular, an active adversary $\mathcal{A}$, including an *honest-but-curious* minority of key servers, a curious user mobile device, and other doctors, learns no information about the key $k_{session}$.
(G3) Forward Secrecy. Compromise of long-term secret material held by protocol entities (including the sensor, key servers, and mobile devices) at time T does not enable $\mathcal{A}$ to learn session keys established prior to T.
(G4) User Consent. In non-emergency operation, session-key establishment requires explicit user consent via the mobile device M_i^{user}. In particular, the protocol must not complete successfully in a non-emergency session unless M_i^{user} authorises the request and provides the user-side contribution.
(G5) Emergency Mode (Break-the-Glass). When M_i^{user} is unavailable, emergency access is executed under a predefined accountable policy (Assumption **A4**). In our case, this policy requires that the request be made by an authorised doctor, and post-hoc accountability is enforced using the audit logs. An emergency session is facilitated via a threshold-based mechanism among the key servers, providing controlled access without user participation while preserving the protocol's security guarantees.

4 Chronos: Protocol Setup

This section specifies the building blocks and initialisation mechanisms required by our protocol *Chronos*, including the primitives, pre-shared key material, entity registration procedures, and key-update rules.

4.1 Cryptographic Primitives and Functions

We use standard symmetric primitives and threshold secret sharing in our construction. Let λ denote the security parameter, and let $P > 2^{\lambda}$ be a public prime modulus used by the secret-sharing scheme. All keys and nonces in the protocol are elements of $\{0,1\}^{\lambda}$, and all key shares are elements of $\mathbb{Z}_P$.

Long-term secrets evolve across sessions via one-way update functions $\mathsf{Upd} : \{0,1\}^\lambda \to \{0,1\}^\lambda$ for λ-bit keys and $\mathsf{Upd}_{\mathbb{Z}_P} : \mathbb{Z}_P \to \mathbb{Z}_P$ for key-server shares, supporting forward secrecy. Session keys and auxiliary keys are derived using a key-derivation function $\mathsf{KDF} : \{0,1\}^* \to \{0,1\}^\lambda$, and message integrity/authenticity is ensured using a message authentication code $\mathsf{MAC} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$. To perform arithmetic over the public prime modulus P, we employ a public injective mapping $\mathsf{Encode} : \{0,1\}^\lambda \to \mathbb{Z}_P$ with a left-inverse $\mathsf{Decode} : \mathbb{Z}_P \to \{0,1\}^\lambda$. To derive a λ-bit value from an arbitrary field element, we further define a fixed public extraction hash $\mathsf{H}_{\mathsf{extract}} : \mathbb{Z}_P \to \{0,1\}^\lambda$. We distribute server-side secrets using Shamir secret sharing $\mathsf{SC} : \{0,1\}^\lambda \times [n] \to \mathbb{Z}_P$, enabling reconstruction from any t shares while hiding the secret from fewer than t shares. Finally, let $\mathsf{Enc}_{\mathsf{PK}}(m)$ and $\mathsf{Dec}_{\mathsf{SK}}(c)$ denote public-key encryption and decryption, and let $\sigma_{\mathsf{SK}}(m)$ denote a signature on m under secret key SK. Appendix 11 provides the full definitions of the introduced cryptographic primitives.

4.2 Pre-Shared Keys

In the Chronos construction, we assume four categories of pre-shared keys: (i) the synchronisation key, (ii) the user–sensor key, (iii) the server–sensor key, and (iv) the emergency-access key. Except for the user–sensor key, which is permanent, all other keys are updatable and may be refreshed during the session-establishment process, depending on the session's progression.

Synchronisation key (k_i^{sync})**.** This key is shared for each sensor S_i between the sensor and the key-server set $\mathcal{KS}$. It maintains synchronisation of the key-update state across these entities and enables sensor-side authentication.

User–sensor key ($k_i^{\text{U-S}}$)**.** This key is shared between sensor S_i and the user's mobile device M_i^{user}. It provides the user-side contribution to the session-key generation process.

Server–sensor key ($k_i^{\text{KS-S}}$)**.** This key is shared between sensor S_i and the key-server set $\mathcal{KS}$. Concretely, $k_i^{\text{KS-S}}$ is secret-shared across the key servers, and each key server $\mathsf{KS}_j \in \mathcal{KS}$ stores the corresponding share $k_{i,j}^{\text{KS-S}}$. It provides the servers' contribution to the session-key generation process and is frequently updated to ensure forward secrecy.

Emergency-access key (k_i^{emg})**.** This key is shared between the key servers in $\mathcal{KS}$. After initialisation and once after each emergency when M_i^{user} reconnects, the mobile device runs the emergency-key update protocol (Sect. 4.4) to secret-share fresh emergency material across $\mathcal{KS}$, where each key server $\mathsf{KS}_j \in \mathcal{KS}$ stores the corresponding share $k_{i,j}^{\text{emg}}$. These shares are used only when M_i^{user} is unavailable and clinical emergency access is requested, enabling threshold-based reconstruction of the user-side session contribution.

4.3 Entity Registration

This subsection describes the registration phase for the entities involved in the protocol. This is the only phase in which $\mathcal{TA}$ participates; after registration, no further involvement from $\mathcal{TA}$ is required.

Key Server Registration. Upon registration, each key server KS_j for $j \in [n]$ generates a permanent public–private key pair $(\mathsf{PK}_j^{\mathrm{KS}}, \mathsf{SK}_j^{\mathrm{KS}})$. The trusted authority $\mathcal{TA}$ verifies the identity of each key server and issues a digital certificate binding the server identity to its public key $\mathsf{PK}_j^{\mathrm{KS}}$. All key-server certificates are stored in a publicly accessible registry that can be queried by registered users' mobile devices and doctors' devices for certificate verification.

Doctor's Device Registration. Upon registration of a doctor's device D_{Dr}, the device generates a permanent public–private key pair $(\mathsf{PK}_{\mathrm{Dr}}, \mathsf{SK}_{\mathrm{Dr}})$. The trusted authority $\mathcal{TA}$ verifies the doctor's identity and issues a digital certificate binding the doctor's device identity to its public key $\mathsf{PK}_{\mathrm{Dr}}$. The resulting certificate is recorded on the replicated ledger $\mathsf{DB}_{\mathrm{rep}}$ accessible to the key servers. New doctor devices can be registered at any time; this only appends a certificate record to $\mathsf{DB}_{\mathrm{rep}}$ and does not require reinitialising any registered IMD.

Implantable Medical Device and Mobile-Device Registration. When a new implantable medical device (sensor) S_i with a hardcoded identifier ID_{S_i} is registered, the trusted authority $\mathcal{TA}$ initialises the long-term key material for S_i, the corresponding user's mobile device M_i^{user}, and the key server set $\mathcal{KS} = \{\mathsf{KS}_j\}_{j\in[n]}$. The initialisation proceeds as follows.

• *Sensor Key Material:* For sensor S_i, $\mathcal{TA}$ samples the following pre-shared keys uniformly at random from $\{0,1\}^\lambda$: (i) the synchronisation key k_i^{sync}, (ii) the user–sensor key $k_i^{\text{U-S}}$, and (iii) the server–sensor key $k_i^{\text{KS-S}}$. The trusted authority then securely installs these values on S_i.

• *Mobile-Device Material:* The trusted authority $\mathcal{TA}$ securely installs the same user–sensor key $k_i^{\text{U-S}}$ on the corresponding user's mobile device M_i^{user}. In addition, M_i^{user} generates a long-term public–private key pair $(\mathsf{PK}_i^{\mathrm{user}}, \mathsf{SK}_i^{\mathrm{user}})$. The trusted authority verifies the user's identity, issues a certificate binding the user identifier to $\mathsf{PK}_i^{\mathrm{user}}$, and records this certificate on the replicated ledger $\mathsf{DB}_{\mathrm{rep}}$.

• *Key-Server Material:* For sensor S_i, $\mathcal{TA}$ initialises each key server KS_j with the following values: (i) synchronisation keys $(k_i^{\mathrm{sync,old}}, k_i^{\mathrm{sync,new}})$ and a state counter N_i^{state}, and (ii) a share of the server–sensor key $k_{i,j}^{\text{KS-S}}$. At initialisation, $\mathcal{TA}$ sets $k_i^{\mathrm{sync,old}} := \emptyset$, $k_i^{\mathrm{sync,new}} := k_i^{\mathrm{sync}}$, and $N_i^{\mathrm{state}} := 0$, and computes $k_{i,j}^{\text{KS-S}} := \mathsf{SC}\big(k_i^{\text{KS-S}}, j\big)$, where SC is the (t,n) secret-sharing scheme described in Sect. 4.1.

• *Emergency-Key Update Initialisation:* To enable future session-key establishment in clinical emergencies when the user's mobile device is unavailable, M_i^{user} executes the emergency-key update protocol, as detailed in Sect. 4.4. As a result of this protocol, the sensor stores fresh sensor-side emergency randomness $r_i^{\mathrm{S,emg}} \in \{0,1\}^\lambda$, and the key servers store the corresponding mobile-side emergency randomness $r_i^{\mathrm{U,emg}} \in \{0,1\}^\lambda$ together with fresh emergency-access key shares $\{k_{i,j}^{\mathrm{emg}}\}_{j\in[n]}$, where $k_{i,j}^{\mathrm{emg}}$ is the j-th share held by KS_j.

Stored Key Sets. After completing the above initialisation steps (including the emergency-key update protocol), the following key sets are stored *locally.*

Table 1: Key-update rules for sensor S_i and key server KS_j.

Scenario	**Update steps**
UPD_{S_i}**: Sensor-side update (at** S_i**)**	① $k_i^{\text{sync}} \leftarrow \mathsf{Upd}(k_i^{\text{sync}})$ ② $k_i^{\text{KS-S}} \leftarrow \mathsf{Upd}(k_i^{\text{KS-S}})$
UPD_{KS}**: Key-server-side update (at** KS_j**)** Upon receiving the committed state-update log $\mathcal{L}_i^{\text{KS-S}} = (\mathsf{sid},\ \mathsf{ID}_{S_i},\ N_i^{\mathsf{req}},\ \{\Delta_{i,j}\}_{j\in[n]})$, apply only if $N_i^{\text{state}} = N_i^{\mathsf{req}}$; otherwise ignore the log as stale.	① $N_i^{\text{state}} \leftarrow N_i^{\text{state}} + 1$ ② $k_i^{\text{sync,old}} \leftarrow k_i^{\text{sync,new}}$ ③ $k_i^{\text{sync,new}} \leftarrow \mathsf{Upd}(k_i^{\text{sync,new}})$ ④ $k_{i,j}^{\text{KS-S}} \leftarrow \Big(\mathsf{Upd}_{\mathbb{Z}_P}(k_{i,j}^{\text{KS-S}}) + \Delta_{i,j} \bmod P\Big)$
$\mathsf{UPD}_{\text{KS}}^{\text{emg}}$**: Key-server-side update (at** KS_j**)** Upon receiving the committed emergency log followed by its committed acknowledgement: $\mathcal{L}_i^{\text{emg}} = \Big(\mathsf{sid}^{\text{emg}},\ \mathsf{ID}_{S_i},\ r_i^{\text{U,emg,new}},\ \{\mathsf{Enc}_{\mathsf{PK}_j^{\text{KS}}}(k_{i,j}^{\text{emg,new}})\}_{j\in[n]}\Big)$.	① $k_{i,j}^{\text{emg}} \leftarrow k_{i,j}^{\text{emg,new}}$ ② $r_i^{\text{U,emg}} \leftarrow r_i^{\text{U,emg,new}}$

Sensor S_i stores $\mathcal{K}_i^{\text{sensor}} = \Big\{k_i^{\text{sync}},\ k_i^{\text{U-S}},\ k_i^{\text{KS-S}},\ r_i^{\text{S,emg}}\Big\}$. Each key server KS_j stores the following key set for sensor S_i:

$$\mathcal{K}_{i,j}^{\text{server}} = \Big\{\big(N_i^{\text{state}},\ k_i^{\text{sync,old}},\ k_i^{\text{sync,new}}\big),\ k_{i,j}^{\text{KS-S}},\ \big(k_{i,j}^{\text{emg}},\ r_i^{\text{U,emg}}\big)\Big\}.$$

4.4 Key Update

To provide forward secrecy and support emergency access, Chronos employs the following key-update procedures.

Sensor-Side Key Update. For sensor S_i with the stored key set $\mathcal{K}_i^{\text{sensor}}$ (see Sect. 4.3), the key-update procedure UPD_{S_i} is introduced in Table 1.

Server-Side Key Update. For key server KS_j with the locally stored key set $\mathcal{K}_{i,j}^{\text{server}}$ corresponding to sensor S_i (see Sect. 4.3), the key-update procedures UPD_{KS} and $\mathsf{UPD}_{\text{KS}}^{\text{emg}}$ are given in Table 1. All server-side updates are applied only upon receiving the corresponding *committed* log from the replicated ledger DB_{rep}.

Emergency-Key Update. During initialisation, and again after each emergency scenario once the user's mobile device is back online, the mobile device M_i^{user} initiates an emergency-key update with sensor S_i as detailed in Table 2. The goal is to refresh emergency material so that, if M_i^{user} is later unavailable, the key servers can reconstruct the user-side contribution in emergency sessions.

Upon successful completion, S_i stores $r_i^{\text{S,emg}}$ as its current sensor-side emergency randomness. The key server KS_j stores (via the committed emergency log) the corresponding mobile-side emergency randomness $r_i^{\text{U,emg}}$ together with the corresponding updated emergency-access shares $k_{i,j}^{\text{emg}}$, which are used only in a future emergency session.

5 Chronos: Protocol Flow

This section describes the flow of the Chronos protocol for establishing a session key between a doctor's device D_{Dr} and sensor S_i, with the assistance of the key-server set $\mathcal{KS}$. In the non-emergency setting, session establishment additionally requires user consent via the mobile device M_i^{user}; in the emergency setting, the protocol proceeds without M_i^{user}.

The protocol steps are summarised in Fig. 2, which specifies the message formats and computations performed in each step. Figs. 3 and 4 in App. 13 provide an intuitive message-flow diagram. A fully detailed specification is given in App. 13.

❶ **Protocol initiation ($D_{\mathrm{Dr}} \to S_i$).** To start a new session, D_{Dr} samples a fresh session identifier sid and nonce r_{Dr}, and sends m_1 to S_i. If resynchronising an ongoing session after step ❺, the message includes state-update information UI_{S_i} and the same session ID to help S_i synchronise with $\mathcal{KS}$.

❷ **Sensor-side state synchronisation via update token (S_i).** Upon receiving m_1, S_i processes UI_{S_i}. If present, $\mathsf{UI}_{S_i} = (\tau_{\mathsf{upd}}, r_{\mathrm{upd}})$ is an authenticated *update token* generated by the leader to indicate that S_i is exactly one state behind. If the tag τ_{upd} verifies under key $\mathsf{Upd}(k_i^{\mathrm{sync}})$ for the pending session context, S_i applies the sensor-side key update UPD_{S_i} (Sect. 4.4); otherwise it aborts.

Table 2: The emergency-key update protocol.

① **Initiation of emergency-key update protocol ($M_i^{\mathrm{user}} \to S_i$).** M_i^{user}: Sample $\mathsf{sid}^{\mathrm{emg}} \xleftarrow{\$} \{0,1\}^{\lambda_{\mathsf{sid}}}$; derive $k_i^{\text{emg-mac}} := \mathsf{KDF}\big(k_i^{\text{U-S}} \,\|\, \mathsf{sid}^{\mathrm{emg}} \,\|\, L = \texttt{"emg-upd-mac"}\big)$; compute $\tau_1 := \mathsf{MAC}_{k_i^{\text{emg-mac}}}\big(\mathsf{sid}^{\mathrm{emg}} \,\|\, \mathsf{ID}_{S_i} \,\|\, L = \texttt{"emg-upd-req"}\big)$; send $(\mathsf{sid}^{\mathrm{emg}}, \mathsf{ID}_{S_i}, \tau_1, L = \texttt{"emg-upd-req"})$ to S_i.
② **Sensor-side randomness contribution ($S_i \to M_i^{\mathrm{user}}$).** S_i: Verify τ_1; if valid, sample $r_i^{\mathrm{S,emg,new}} \xleftarrow{\$} \{0,1\}^{\lambda}$; store $(\mathsf{sid}^{\mathrm{emg}}, r_i^{\mathrm{S,emg,new}})$ until completion/timeout/abort; compute $\tau_2 := \mathsf{MAC}_{k_i^{\text{emg-mac}}}\big(\mathsf{sid}^{\mathrm{emg}} \,\|\, \mathsf{ID}_{S_i} \,\|\, r_i^{\mathrm{S,emg,new}} \,\|\, L = \texttt{"emg-upd-chal"}\big)$; return $(\mathsf{sid}^{\mathrm{emg}}, r_i^{\mathrm{S,emg,new}}, \tau_2)$ to M_i^{user}.
③ **Emergency-update log submission ($M_i^{\mathrm{user}} \to \mathcal{KS}$).** M_i^{user}: Verify τ_2; if valid, sample $r_i^{\mathrm{U,emg,new}} \xleftarrow{\$} \{0,1\}^{\lambda}$; derive $k_i^{\mathrm{emg,new}} := \mathsf{KDF}\big(k_i^{\text{U-S}} \,\|\, r_i^{\mathrm{S,emg,new}} \,\|\, r_i^{\mathrm{U,emg,new}} \,\|\, L = \texttt{"U-S:emg key"}\big)$; for all $j \in [n]$, compute $k_{i,j}^{\mathrm{emg,new}} := \mathsf{SC}(k_i^{\mathrm{emg,new}}, j)$ and $c_{i,j} := \mathsf{Enc}_{\mathsf{PK}_j^{\mathrm{KS}}}(k_{i,j}^{\mathrm{emg,new}})$; submit $\mathcal{L}_i^{\mathrm{emg}} := (\mathsf{sid}^{\mathrm{emg}}, \mathsf{ID}_{S_i}, r_i^{\mathrm{U,emg,new}}, \{c_{i,j}\}_{j\in[n]})$ for ledger commitment.
④ **Key-server-side emergency preparation ($\mathcal{KS} \to M_i^{\mathrm{user}}$).** Key servers: upon commitment of $\mathcal{L}_i^{\mathrm{emg}}$, store $(k_{i,j}^{\mathrm{emg,new}}, r_i^{\mathrm{U,emg,new}}, \mathsf{sid}^{\mathrm{emg}})$ as pending emergency material; send confirmation to M_i^{user}.
⑤ **Sensor-side finalization of the emergency update ($M_i^{\mathrm{user}} \to S_i$).** M_i^{user}: After confirmation that $\mathcal{L}_i^{\mathrm{emg}}$ is committed, compute $\tau_3 := \mathsf{MAC}_{k_i^{\text{emg-mac}}}\big(\mathsf{sid}^{\mathrm{emg}} \,\|\, \mathsf{ID}_{S_i} \,\|\, r_i^{\mathrm{S,emg,new}} \,\|\, L = \texttt{"emg-upd-fin"}\big)$; send $(\mathsf{sid}^{\mathrm{emg}}, \tau_3)$ to S_i.
⑥ **Sensor acknowledge ($S_i \to M_i^{\mathrm{user}}$).** S_i: Verify τ_3; if valid, set $r_i^{\mathrm{S,emg}} \leftarrow r_i^{\mathrm{S,emg,new}}$; compute $\tau_4 := \mathsf{MAC}_{k_i^{\text{emg-mac}}}\big(\mathsf{sid}^{\mathrm{emg}} \,\|\, \mathsf{ID}_{S_i} \,\|\, r_i^{\mathrm{S,emg}} \,\|\, L = \texttt{"emg-upd-ack"}\big)$; send $(\mathsf{sid}^{\mathrm{emg}}, \tau_4)$ to M_i^{user}.
⑦ **Mobile-side activation ($M_i^{\mathrm{user}} \to \mathcal{KS}$).** M_i^{user}: Verify τ_4. If invalid, restart from ①. If valid, send the acknowledgement $m_{\mathrm{act}} = \big(\mathsf{sid}^{\mathrm{emg}}, \mathsf{ID}_{S_i}, H(\mathcal{L}_i^{\mathrm{emg}}), L = \texttt{"emg-upd-activate"}\big)$ to $\mathcal{KS}$. Upon commitment of m_{act}, $\mathcal{KS}$ performs the update procedure $\mathsf{UPD}_{\mathrm{KS}}^{\mathrm{emg}}$ as presented in Sect. 4.4 and sends a confirmation to M_i^{user}. If no confirmation is received, M_i^{user} resends m_{act}.

❸ **Sensor response ($S_i \to D_{\mathrm{Dr}}$).** S_i samples r_{S_i} and computes two authentication tags: τ_1^S under k_i^{sync} for key-server synchronisation/authentication, and τ_2^S under a derived key $k_i^{\text{U-S,mac}}$ for the mobile-consent path. It sends m_2 to D_{Dr} and temporarily stores $(\mathsf{sid}, \mathsf{ID}_{\mathrm{Dr}}, r_{\mathrm{Dr}}, r_{S_i})$ as current session information.

❹ **Doctor session-establishment request to the key-server service ($D_{\mathrm{Dr}} \to \mathcal{KS}$).** D_{Dr} generates an ephemeral public key $\mathsf{ePK}_{\mathrm{Dr}}$ for an additively homomorphic encryption scheme and selects the emergency mode flag $\mathsf{Emg} \in \{0, 1\}$. It commits to the sensor tags and nonces via $h_{\mathrm{Dr}} = H(\mathsf{mac_token}_{\mathrm{Dr}})$, signs the core request $\mathsf{core_req}_{\mathrm{Dr}}$ under its long-term private key $\mathsf{SK}_{\mathrm{Dr}}$, and submits m_3 to the key-server service. The resulting signature enables all key servers to authenticate the doctor and the requested emergency mode.

❺ **Leader verification and synchronisation ($\mathsf{KS}_{\mathrm{leader}}$).** $\mathsf{KS}_{\mathrm{leader}}$, an online key server selected to coordinate the session, verifies the doctor's certificate and σ_{Dr} on $\mathsf{core_req}_{\mathrm{Dr}}$, checks the commitment h_{Dr}, and evaluates whether S_i is synchronised by validating τ_1^S under $k_i^{\text{sync,new}}$ (synced) or $k_i^{\text{sync,old}}$ (one-step behind). If S_i is one step behind, the leader returns a resynchronisation message containing the update token UI_{S_i} so the doctor can resynchronise the ongoing session; otherwise, it proceeds.

❻ **Broadcast and consent request ($\mathsf{KS}_{\mathrm{leader}} \to \mathcal{KS}$; if $\mathsf{Emg} = 0$: $\mathsf{KS}_{\mathrm{leader}} \to M_i^{\text{user}}$).** $\mathsf{KS}_{\mathrm{leader}}$ broadcasts m_4 containing $(\mathsf{core_req}_{\mathrm{Dr}}, \sigma_{\mathrm{Dr}}, \mathsf{ePK}_{\mathrm{leader}})$ to all KS_j to authenticate the doctor's request and solicit encrypted server shares under leader's ephemeral public key $\mathsf{ePK}_{\mathrm{leader}}$ for constructing the server-side session contribution. If $\mathsf{Emg} = 0$, the leader also sends m_5 to M_i^{user}, including $\mathsf{Info}_{\mathrm{Dr}}$ and $\mathsf{ePK}_{\mathrm{Dr}}$, to obtain user consent and the corresponding user-side contribution.

❼ **User consent (only if $\mathsf{Emg} = 0$: $M_i^{\text{user}} \to \mathsf{KS}_{\mathrm{leader}}$).** In non-emergency mode, M_i^{user} verifies τ_2^S to confirm request linkage to S_i and sid. Upon user consent, M_i^{user} derives the user contribution $k_{\text{session}}^{\text{U-S}}$ from pre-shared key $k_i^{\text{U-S}}$ and fresh nonces. It encrypts $\mathsf{Encode}(k_{\text{session}}^{\text{U-S}}) \in \mathbb{Z}_P$ under $\mathsf{ePK}_{\mathrm{Dr}}$ and returns it to $\mathsf{KS}_{\mathrm{leader}}$, ensuring only D_{Dr} can decrypt it.

❽ **Server share submission ($\mathsf{KS}_j \to \mathsf{KS}_{\mathrm{leader}}$).** Each KS_j first verifies the integrity and freshness of the doctor's request. It then returns its server–sensor share encrypted under the leader's ephemeral public key, $\mathsf{Enc}_{\mathsf{ePK}_{\mathrm{leader}}}(k_{i,j}^{\text{KS-S}})$. If $\mathsf{Emg} = 1$, it additionally appends the emergency share encrypted under the doctor's ephemeral public key, $\mathsf{Enc}_{\mathsf{ePK}_{\mathrm{Dr}}}(k_{i,j}^{\text{emg}})$.

❾ **Reconstruction, ledger commit, and homomorphic combination ($\mathsf{KS}_{\mathrm{leader}}$).** $\mathsf{KS}_{\mathrm{leader}}$ decrypts and reconstructs the server–sensor key $k_i^{\text{KS-S}}$ from a *threshold* of encrypted shares. It then derives the server-side contribution and *homomorphically* combines it with the encrypted user contribution (directly from M_i^{user} when $\mathsf{Emg} = 0$, or reconstructed from emergency shares when $\mathsf{Emg} = 1$), obtaining $k_{\text{session}}^{\text{KS-U-S,enc}}$. The leader also computes per-server update information $\{\Delta_{i,j}\}$ and submits a state-update log $\mathcal{L}_i^{\text{KS-S}}$ to $\mathsf{DB}_{\mathrm{rep}}$ that enables distributing the information required for each KS_j to obtain its next-session share corresponding to $\mathsf{Upd}(k_i^{\text{KS-S}})$ while preventing recovery of the previous share. All key servers

perform the server-side update $\mathsf{UPD}_{\mathrm{KS}}$ only after the update log is *committed* (Sect. 4.4).

⑩ **Leader response ($\mathsf{KS}_{\mathrm{leader}} \rightarrow D_{\mathrm{Dr}}$).** After ledger commitment and key-server update, $\mathsf{KS}_{\mathrm{leader}}$ sends m_8 containing τ^{KS}, $k_{\mathrm{session}}^{\text{KS-U-S,enc}}$, and the public nonces $r_{\mathrm{KS}}, r_{\mathrm{U}}$ used by the endpoints for key derivation to D_{Dr}.

⑪ **Doctor-side key derivation ($D_{\mathrm{Dr}} \rightarrow S_i$).** D_{Dr} decrypts $k_{\mathrm{session}}^{\text{KS-U-S,enc}}$ to obtain κ_P, derives k_{session}, and sends τ^{Dr} to S_i as explicit key confirmation. The tag, calculated under $k^{\mathrm{Dr,mac}}$ derived from the session key, confirms to S_i that the doctor has derived the same session key.

⑫ **Sensor-side key derivation and confirmation ($S_i \rightarrow D_{\mathrm{Dr}}$).** S_i verifies τ^{KS}, which attests that the doctor is a legitimate, authorised clinician, derives k_{session}, and immediately applies the sensor-side key update UPD_{S_i} (Sect. 4.4). It then verifies τ^{Dr} and replies with τ^{S} under $k^{\mathrm{S,mac}}$, derived from the session key, for key confirmation to D_{Dr}.

⑬ **Doctor-side key confirmation (D_{Dr}).** D_{Dr} verifies τ^{S} and accepts the session if valid, concluding mutual key confirmation between D_{Dr} and S_i.

6 Implementation

We implement *Chronos* as a complete end-to-end system consisting of (i) a clinician-facing Android application, (ii) embedded firmware implementing the implant-side protocol on two representative microcontrollers and (iii) a distributed backend of threshold key servers. The Android application communicates with the implant over Bluetooth Low Energy (BLE) and with the backend over TLS. The implant firmware executes the symmetric-only protocol logic and stores long-term state in non-volatile memory. The backend implements threshold coordination and authenticated state updates.

6.1 Android Application

The clinician device is implemented as an Android application written in Kotlin. The application establishes a BLE connection to the implant and a TLS connection to the backend key servers, relaying protocol messages between the two. BLE-specific constraints, such as message chunking and connection management, are handled at the application layer. The additively homomorphic scheme used to generate the ephemeral public key is the Paillier cryptosystem [19]. The user interface is intentionally minimal and supports both routine operation and a controlled emergency workflow when the patient device is unavailable.

6.2 Implant Firmware

We implement the implant-side protocol on two embedded platforms to capture both legacy and modern device classes.

1) **16-bit MSP430:** We use a Texas Instruments MSP430FR2433 microcontroller (16-bit, 16 MHz, 16 KB FRAM, 4 KB SRAM) to represent an ultra-low-power legacy platform. Long-term protocol state is stored in FRAM. Wireless

communication is provided via an HM-10 BLE module over UART. Cryptographic primitives are implemented using wolfSSL configured for MSP430. Timing is measured using a 16-bit hardware timer in continuous mode with overflow handling.

2) **32-bit Cortex-M33:** We use a Nordic nRF5340 system-on-chip featuring dual ARM Cortex-M33 cores. The application core executes the protocol logic, while the network core handles BLE. The firmware is implemented using Zephyr RTOS via the nRF Connect SDK. We evaluate two cryptographic integration paths: (i) the PSA Cryptography API with hardware acceleration and (ii) wolfSSL, enabling comparison with the MSP430 setup. Timing measurements use Zephyr's high-resolution cycle counter.

6.3 Distributed Backend

The backend consists of one trust authority and four key servers deployed on separate Amazon EC2 instances in a 3-out-of-4 threshold configuration. One key server acts as the leader. Each server runs on an Ubuntu `t2.small` instance. The backend performs threshold key reconstruction, state coordination and server-authenticated update generation. Deployment details do not affect protocol semantics or measurements. The replicated database implementation is detailed in App. 15.

7 Formal Verification of Security Properties

In this section, we first present an informal security argument for *Chronos* for each security goal defined in Sect. 3.3. We then present our formal verification model, the security properties we verify, and explain how these verified properties map back to the goals in Sect. 3.3.

7.1 Informal Security Argument

We briefly argue that *Chronos* achieves goals (G1)-(G5) under the assumptions stated in Sect. 3. A key consistency property underlying the security and operation of Chronos is that the sensor and key-server states cannot diverge arbitrarily.

Theorem 1 (informal: sensor–key-server consistency). For every sensor S_i, the sensor is always either synchronised with the latest committed key-server state or at most one recoverable update behind.

The formal statement and proof are given in Theorem 1 of App. 14. In particular, the sensor–key-server desynchronisation is bounded by one state update, and the update token mechanism repairs only the possible desynchronised state.

(G1) Authenticated session-key establishment.

Key agreement. Both endpoints derive k_{session} from a common seed. The sensor computes the two contributions $k_{\text{session}}^{\text{KS-S}}$ and $k_{\text{session}}^{\text{U-S}}$ locally and derives k_{session}, while the doctor obtains the same combined seed via the key-server system. By Theorem 1, the sensor and key-server states remain consistent, or are first resynchronised through the update-token mechanism; hence, the sensor and the servers use the same evolved $k_i^{\text{KS-S}}$. Together with the permanent pre-shared key $k_i^{\text{U-S}}$, this ensures that both parties derive the same k_{session}.

Entity authentication. The sensor authenticates the doctor by verifying τ^{KS} under k_i^{sync}, where τ^{KS} is generated by the key-server system only after validating the doctor's certified signature on $\mathsf{core_req}_{\mathrm{Dr}}$ and binding it to ID_{S_i}. Symmetrically, the doctor authenticates that the session is for the intended sensor by accepting m_8 only if it is signed by $\mathsf{KS}_{\mathrm{leader}}$ and bound to $(\mathsf{sid}, \mathsf{ID}_{S_i})$.

Key confirmation. The tag τ^{Dr}, computed from k_{session}, confirms to S_i that D_{Dr} holds the same session key, and τ^{S} analogously confirms to D_{Dr} that S_i holds the same key.

(G2) Session-key confidentiality.

Non-emergency mode. The session key is derived from $\kappa_P = u + v \bmod P$, where the user-side contribution u and the server-side contribution v are homomorphically combined by the leader server in encrypted form under $\mathsf{ePK}_{\mathrm{Dr}}$. Thus, under IND-CPA security of the encryption scheme, no external adversary, no coalition of *honest-but-curious* key servers, and not even the mobile device alone can recover κ_P or k_{session}.

Emergency mode. In emergency operation, the user-side contribution is derived from at least t shares $\{k_{i,j}^{\mathrm{emg}}\}$. Any coalition of fewer than t key servers learns no information about the underlying emergency secret, while the key-server system can still form the encrypted value $\mathsf{Enc}_{\mathsf{ePK}_{\mathrm{Dr}}}(u)$ for combination with v. Therefore, under the threshold assumption and IND-CPA security, no curious minority of key servers (nor any external adversary) learns u or k_{session}.

(G3) Forward secrecy.[‡] This follows from the properties below. First, after each session the long-term secrets evolve one-way: the sensor updates its stored key set via Upd, and each key server updates its stored share via $\mathsf{Upd}_{\mathbb{Z}_P}$; thus, compromising the state at time T does not enable recovery of any prior state $T' < T$ under the assumed one-wayness of the update functions. Second, sensitive session values exchanged between D_{Dr}, M_i^{user}, and $\mathcal{KS}$ are protected under ephemeral public keys that are erased after each session, and the update logs $\mathcal{L}_i^{\text{KS-S}}$ support only forward state advancement. Third, server-side state evolution is ledger-driven: key servers advance state only upon majority commitment of the server–sensor update log. Under the assumption that at any time fewer than t servers are offline, any stale (pre-update) shares that an adversary might obtain by compromising the system at time T are fewer than t and therefore insufficient to reconstruct past server-side contributions for any $T' < T$; thus, previously established session keys remain hidden.

(G4) User consent. In non-emergency operation, the key-server leader requests explicit user authorisation by sending $\mathsf{Info}_{\mathrm{Dr}}$ to M_i^{user}; the mobile device proceeds to compute and submit the user-side encrypted contribution only upon consent, and otherwise the session does not complete.

[‡]Forward secrecy in Chronos relies on Assumption (A3): key servers are honest-but-curious and erase obsolete local state after each update. This is reasonable in deployments where compromised key servers can be detected and revoked. If a leader key server maliciously retains reconstructed backend state, a later compromise of the mobile device may endanger forward secrecy.

Table 3: Verified ProVerif queries, proved properties (P1–P6), and their mapping to security goals (G1–G5).

ProVerif query	Verified property	Outcome	Goal(s)
`event(SK_S(k)) => event(SK_D(k)).`	P1: Session-key establishment	Proved	G1, G5
`event(SK_D(k)) => event(SK_S(k)).`	P1: Session-key establishment	Proved	G1, G5
`not (event(SK(k)) && attacker_p2(k)).`	P2: Session-key secrecy (post full compromise)	Proved	G2, G5
`not (event(SK_D(k)) && attacker_p2(k)).`	P3: Doctor-side forward secrecy	Proved	G3, G5
`not (event(SK_S(k)) && attacker_p2(k)).`	P4: Sensor-side forward secrecy	Proved	G3, G5
`event(EndS_NonEmg(ru)) ==> event(UserContrib(ru)).`	P5: User-consent binding	Proved	G4
`event(EndS_Emg(rs)) ==> event(EmgSetupDone(rs)).`	P6: Emergency completion	Proved	G5

(G5) Emergency Mode (Break-the-Glass). In emergency operation, the protocol omits the mobile device and instead reconstructs the user-side contribution using the pre-distributed emergency shares produced by the emergency-key update protocol (Sect. 4.4), enabling controlled threshold-based access while preserving confidentiality and agreement.

7.2 Formal Verification

We formally verify *Chronos* using ProVerif [2]. The model captures concurrent executions of the doctor, sensor, user mobile device, a leader key server, three additional key servers (realising a 3-of-4 threshold), and a trusted authority. All protocol messages between the doctor, sensor, user device, and leader are exchanged over a public channel controlled by a Dolev–Yao adversary [6], while backend coordination and trusted-authority operations are modelled using private channels. The adversary can intercept, replay, reorder, and inject messages, and we assume perfect cryptography in the symbolic model.

The homomorphic combination of user and server contributions is abstracted as an ideal primitive, allowing the leader to combine encrypted user input with its own contribution while ensuring that only the doctor can recover the session key.

Compromise model. To capture strong adversarial capabilities and formally establish forward secrecy, we adopt a multi-phase compromise model. After an unbounded number of protocol sessions (via replication), the execution proceeds through two compromise phases.

In the *minority-compromise* phase, the adversary learns the post-update state of fewer than t key servers, modelling partial backend compromise.

In the subsequent *full-compromise* phase, the adversary learns all long-term secrets and the complete post-update system state, including doctor and server signing keys, user–sensor secrets, evolved backend state, and all threshold shares. Ephemeral session secrets and past session keys are assumed to be erased.

All secrecy and forward-secrecy queries are evaluated in the presence of these compromise phases, ensuring that proven properties hold even under strong post-compromise knowledge. Under the standard soundness assumptions of the symbolic model, ProVerif guarantees that any attack violating these properties would correspond to an executable trace in the model. Thus, the absence of such traces provides strong evidence that *Chronos* satisfies the stated security goals under the considered adversarial model.

Threshold and emergency modelling. The threshold mechanisms used for backend state evolution and emergency recovery are modelled via ideal reconstruction rules over shares. This abstraction captures the intended trust boundary: any coalition of fewer than t key servers learns no information about protected secrets, while authorised reconstruction remains possible.

Events and Verified Properties Let `SK`, `SK_D`, and `SK_S` denote events in which the session key is accepted by either or both endpoints. The events `EndS_NonEmg` and `EndS_Emg` represent successful completion of non-emergency and emergency executions, respectively, while `UserContrib` and `EmgSetupDone` capture user participation and prior emergency provisioning.

The verified properties are as follows:

(P1) Session-key establishment. Whenever the sensor outputs a session key, the initiating doctor outputs the same key. This captures agreement and explicit key confirmation.
(P2) Session-key secrecy. The adversary cannot derive the post-update session key, even after the minority-compromise phase.
(P3) Doctor-side forward secrecy. Session keys accepted by the doctor remain secret even after full compromise of all long-term secrets and backend state.
(P4) Sensor-side forward secrecy. Session keys accepted by the sensor remain secret under the same full-compromise conditions.
(P5) User-consent binding. In non-emergency mode, every completed session is causally linked to a user contribution generated with the same nonce.
(P6) Emergency completion. In emergency mode, every completed session is linked to a prior successful emergency provisioning of the corresponding sensor state.

Additional verified invariants. Beyond P1–P6, the ProVerif artefact also verifies auxiliary invariants for protocol robustness: resynchronisation retry and recovery are both bound to matching server-issued update tokens, and emergency completion is bound to valid doctor emergency authorisation. We keep Table 3 focused on the paper-level security goals, while these additional checks document that the state-recovery and break-the-glass paths are not replayable across doctor–sensor contexts. These checks further strengthen the robustness of *Chronos* against replay and state-mismatch attacks.

8 Evaluation

In this section, we evaluate the system to show that Chronos not only achieves its intended security guarantees but is also efficient and robust enough for practical deployment.

8.1 Security Evaluation

This section demonstrates how Chronos fulfils its primary security objectives while maintaining resilience against practical deployment threats and operational edge cases.

Distributed forward secrecy and state evolution: A key feature of Chronos is *forward secrecy* in a *distributed* key-server setting. After each session, the sensor and key servers evolve their long-term secrets via one-way updates; the ledger commits only the information needed for forward state advancement, without enabling recovery of prior states. Since state changes are ledger-driven and applied only after majority commitment (Raft/CFT [18] in our implementation, App. 15), all key servers converge to a single committed history even under transient faults or delayed replicas, preventing desynchronisation across the backend (see Appendices 12 and 14).

Confidentiality and privacy in clinical sessions: Chronos ensures that only the intended endpoints can derive the session key. The session key depends on both a server-side contribution (derived from server–sensor secret state unknown to the mobile) and a user-side contribution, which is provided to the backend only in encrypted form for combination and delivery to the doctor. Consequently, neither an external eavesdropper, nor a minority of key servers, nor the user device alone can reconstruct the session key.

Robustness under practical edge cases: Chronos is designed to remain secure and operational under realistic deployment failures. In particular, when the user's mobile device is unavailable, *emergency mode* enables session establishment for authenticated emergency-access requests, with threshold-based control at the backend. Moreover, the distributed key-server architecture avoids a single point of failure: *availability under partial outage* is preserved via quorum-based progress among key servers, so the protocol can still complete and establish a session key as long as a quorum remains online.

Practical attack resistance: Chronos is robust against common IoT/medical threats. (i) *Eavesdropping* does not expose key material because sensitive exchanges occur over authenticated encrypted channels and no key seed is sent directly over the sensor link. (ii) *Impersonation of a clinical device* is prevented because the key-server system authenticates doctor requests using certified signing keys before producing the authenticated evidence accepted by the sensor. (iii) *Replay/interleaving* is mitigated by fresh per-session nonces/session identifiers and domain-separated derivations/tags. (iv) *Mobile compromise* cannot by itself reveal the session key (it lacks server–sensor secret state), though consent decisions are necessarily authoritative in non-emergency mode.

Table 4: Implant-side performance of *Chronos*. Step ❷ has two cases: (a) backend-consistent (sensor and servers synchronised) and (b) backend-inconsistent (sensor validates leader update information after detecting a state mismatch).

Protocol step	Platform	Cycles	Time (ms)	Typ. Energy (μJ)	Max Energy (μJ)
Step ❷ – (a)	MSP430 + wolfSSL	806	0.0504	0.4607	0.5262
	Cortex-M33 + wolfSSL	663	0.0052	0.1217	0.2418
	Cortex-M33 + PSA	660	0.0052	0.1217	0.2418
	Cortex-M33 + PSA HW	662	0.0052	0.1217	0.2418
Step ❷ – (b)	MSP430 + wolfSSL	1,236,104	77.2565	706.2017	806.5579
	Cortex-M33 + wolfSSL	32,204	0.2516	5.8874	11.6994
	Cortex-M33 + PSA	48,937	0.3823	8.9458	17.7770
	Cortex-M33 + PSA HW	50,367	0.3935	9.2079	18.2978
Step ❸	MSP430 + wolfSSL	3,284,154	205.2596	1,876.2780	2,142.9102
	Cortex-M33 + wolfSSL	103,992	0.8124	19.0102	37.7766
	Cortex-M33 + PSA	143,480	1.1209	26.2291	52.1219
	Cortex-M33 + PSA HW	110,518	0.8634	20.2036	40.1481
Step ⓬	MSP430 + wolfSSL	17,162,823	1,072.6764	9,805.3350	11,198.7416
	Cortex-M33 + wolfSSL	1,312,968	10.2576	240.0278	476.9784
	Cortex-M33 + PSA	1,449,632	11.3253	265.0120	526.6265
	Cortex-M33 + PSA HW	1,353,590	10.5749	247.4527	491.7329

8.2 Performance Evaluation

Table 4 reports the execution cost of the *implant side* steps of *Chronos* on two representative embedded platforms. We report both clock cycles and wall clock time. The MSP430FR2433 measurements assume a 16 MHz system clock and the Cortex-M33 measurements assume a 128 MHz application core clock. The reported times are consistent with these clock rates (time equals cycles divided

Table 5: Per session timing breakdown (ms). Android total (Steps ❶, ❹, ⓫ and ⓭) is 66.7 ms. Backend total (Steps ❺ to ⓾) is 1,722.8 ms. MCU totals sum Steps ❷, ❸ and ⓬. End-to-end (consistent) is Android total plus backend total plus MCU total (consistent). The inconsistent path increases end-to-end latency by Δ (computed as $(\mathrm{E2E}_{inc} - \mathrm{E2E}_{con})/\mathrm{E2E}_{con}$).

Platform	MCU con.	MCU inc.	E2E con.
MSP430 + wolfSSL	1,278.0	1,355.2	3,067.5
Cortex-M33 + wolfSSL	11.08	11.32	1,800.6
Cortex-M33 + PSA	12.45	12.83	1,801.9
Cortex-M33 + PSA (HW)	11.16	11.55	1,800.6
Δ in end-to-end	MSP430: 2.52%	M33: 0.01 to 0.02%	

by clock frequency). Step ❷ has two paths: a fast path when the implant and backend are already consistent and a slower path when the implant must first update its state and sync with the leader.

For completeness, we also measure the non-implant components that contribute to end-to-end latency: the Android application steps (Steps ❶, ❹, ⓫ and ⓭) and the backend threshold processing (Steps ❺ to ❿). In our deployment, Steps ❺ to ❿ together take 1722.8110 ms across five cloud servers, while the Android steps contribute 0.1467 ms (Step ❶), 17.1310 ms (Step ❹), 48.6665 ms (Step ⓫) and 0.7320 ms (Step ⓭).

End-to-end latency: Table 5 summarises the per-session computation cost on the implant and the resulting end-to-end latency under both consistent and inconsistent execution paths. We distinguish between a *consistent* execution path, where the implant and backend servers are synchronised and an *inconsistent* path, where the implant validates leader update information before proceeding. Using the measured timings, the total end-to-end latency is approximately 3.07 s on the MSP430 when the backend is consistent and approximately 3.14 s when the implant additionally validates leader update information in Step ❷. On the Cortex-M33 (wolfSSL), the corresponding end-to-end latency is approximately 1.80 s in both cases, since the backend dominates and the Step ❷ inconsistency check remains sub-millisecond.

Where the bottleneck lies: The implant side is the dominant computational bottleneck only on the legacy MSP430 platform. Even on the MSP430, the total implant-side computation for the consistent case is about 1.28 s (Steps ❷, ❸ and ⓬) and the additional Step ❷ validation cost in the inconsistent case adds 77.3 ms. On the Cortex-M33, the total implant-side time is about 11.1 ms (wolfSSL) and remains on the order of 10 to 12 ms across software- and hardware-assisted cryptography configurations. This gap supports the intended positioning of *Chronos* as deployable across both ultra-constrained and modern embedded platforms, while keeping the implant critical path symmetric.

Interpretation: The results show that *Chronos* remains practical even on a 16-bit microcontroller, where the most expensive implant side step takes about 1.07 s. In our measured end-to-end deployment, backend threshold processing is the largest contributor to total latency, suggesting that further engineering effort on backend parallelism and request scheduling can reduce overall session time without changing the implant-side design. In contrast, on the modern Cortex-M33 platform, the implant overhead becomes negligible relative to backend processing and Android-level overheads.

8.3 Energy Consumption Evaluation

Energy model: We estimate implant-side energy using the execution times in Table 4 and active-mode current profiles from the corresponding microcontroller datasheets. For each step, energy is computed as $E = P \times t = V \times I \times t$, where t is the measured execution time, V is the supply voltage, and I is the active current draw of the microcontroller. We use $V = 3$ V for both platforms and report typical and conservative upper-bound estimates where available.

MSP430 results: For the MSP430, we use the active FRAM profile at 16 MHz and 25°C with a 0% cache hit ratio, giving 3.047 mA typical and 3.480 mA maximum current draw. A complete implant-side execution of *Chronos* consumes 11.68 mJ in the backend-consistent case and 12.39 mJ in the backend-inconsistent case; under maximum current, these increase to 13.34 mJ and 14.15 mJ.

Cortex-M33 results: For the Cortex-M33, we use the IAPPCPU4 application-core profile at 128 MHz, giving a typical current draw of 7.8 mA. Since the datasheet does not provide maximum current values for the specific execution modes used in our evaluation, we conservatively use the highest reported application CPU current, 15.5 mA, as an upper-bound estimate.[§] Despite this higher active current, the Cortex-M33's shorter execution time yields substantially lower energy consumption: a backend-inconsistent run consumes at most 0.300 mJ using PSA and 0.277 mJ using hardware-accelerated PSA under typical current, and 0.597 mJ and 0.550 mJ under the conservative upper-bound model.

Lifetime impact: The main practical result is that *Chronos* provides forward secrecy, state recovery, and emergency-mode support without introducing a meaningful battery burden. Table 6 extrapolates worst-case backend-inconsistent executions using maximum-current estimates, namely 14.15 mJ per execution for the MSP430 and 0.597 mJ for the Cortex-M33 with PSA, against a 50 kJ battery budget.

Even under pessimistic assumptions, the long-term overhead remains negligible. In the daily-execution scenario, which already exceeds expected clinical usage, the total cost over a 10-year lifetime is only 0.103% of a 50 kJ battery on the MSP430 and 0.004% on the Cortex-M33. Extending the lifetime to 15 years still keeps the overhead below 0.16% and 0.01%, respectively. Even under a deliberately unrealistic hourly-execution workload sustained for 15 years, *Chronos* consumes only 3.72% of the MSP430 budget and 0.157% of the Cortex-M33 budget.

Communication overhead: Table 4 isolates implant-side computation and excludes radio energy, which depends on the wireless stack, packetisation, retransmissions, and connection parameters. *Chronos* adds only fixed-size authentication tags and nonces: for a 128-bit security parameter, this is approximately 100–150 bytes of additional protocol data per session from the IMD perspective. For a representative BLE-class transceiver with active TX/RX currents around 10 mA and data rates around 1 Mbps, the energy per transmitted bit is on the order of 10^{-2}–10^{-1} μJ/bit. The resulting radio overhead is only a few tens of μJ per session, comparable to compute cost on modern platforms and substantially smaller than on legacy devices such as the MSP430. Thus, even with representative communication energy, the additional overhead remains small relative to

[§]This upper bound is intentionally pessimistic. The 15.5 mA value corresponds to the highest reported application CPU current in the nRF5340 datasheet table, not to the specific profile used by our implementation.

Table 6: Long-term battery impact of *Chronos* under pessimistic usage assumptions. The table uses backend-inconsistent executions and maximum-current energy estimates.

Usage model	Platform	Total energy	50 kJ battery
Daily for 10 years	MSP430	51.65 J	0.103%
	Cortex-M33	2.18 J	0.004%
Daily for 15 years	MSP430	77.47 J	0.155%
	Cortex-M33	3.27 J	0.007%
Hourly for 10 years	MSP430	1.24 kJ	2.48%
	Cortex-M33	52.30 J	0.105%
Hourly for 15 years	MSP430	1.86 kJ	3.72%
	Cortex-M33	78.45 J	0.157%

realistic IMD battery budgets. A full radio-level profile is deployment-specific and orthogonal to the cryptographic feasibility results reported here.

These results indicate that *Chronos* achieves its security goals with negligible long-term energy impact. In particular, the cost of adding authenticated key establishment, forward secrecy, and state-recovery support is small compared with the energy budget reserved for the primary medical functions of an IMD. The cryptographic overhead is therefore not a practical battery-lifetime bottleneck, even under highly pessimistic usage assumptions.

9 Related Work

Research on IMD security spans multiple directions, shaped by the tension between safety, usability, energy constraints, and confidentiality. We summarise the most relevant lines of work and position *Chronos* within this landscape.

Wireless Attacks and System Analyses. Early work established the feasibility of practical wireless attacks on IMDs, including eavesdropping and command injection using software-defined radios [9]. Subsequent analyses confirmed that vulnerabilities persist in modern cardiac and neurostimulation devices, often due to proprietary protocols prioritising efficiency over strong cryptographic guarantees [14, 16]. Attacks on insulin delivery systems further demonstrate the risks of weak or absent authentication [12]. A systematisation of knowledge shows that existing defences often optimise individual properties such as access control or availability, while leaving long-term confidentiality insufficiently addressed [23]. These results highlight the need for cryptographic protection that remains robust over the full device lifetime.

Physiological and Proximity-Based Pairing. Several approaches derive shared secrets from physiological signals or physical proximity. Heart-to-Heart and related schemes use heartbeat-derived features [10, 22], while proximity-based methods rely on distance bounding or ultrasonic channels [20]. However, physiological signals exhibit limited entropy and instability, leading to unreliable

key establishment [13]. Moreover, remote sensing attacks have shown that such signals can be recovered using commodity cameras, undermining their secrecy assumptions [25]. In contrast, *Chronos* avoids reliance on physical or biological entropy and instead achieves security through cryptographic state evolution.

Physical-Layer Defences and External Proxies. Physical-layer defences attempt to secure IMDs by controlling the wireless channel, for example, via friendly jamming or wearable guardians [8, 28]. While effective against opportunistic attackers, these approaches rely on assumptions about adversary capabilities and environmental conditions. In particular, friendly jamming does not provide cryptographic confidentiality against adversaries with multiple antennas and advanced signal processing [27]. *Chronos* does not depend on physical-layer assumptions or additional hardware, and instead provides cryptographic forward secrecy independent of the attacker's radio capabilities.

Delegation and Smartphone-Centric Architectures. Recent work delegates access control and key management to a patient-held smartphone [7], enabling flexible, user-centric authorisation. However, such systems mainly address access control, not temporal confidentiality: compromised long-term credentials can still expose historical IMD communications. *Chronos* builds on this model but removes the smartphone as a single point of failure by distributing trust and protecting past sessions under compromise.

Cryptography on Constrained Medical Devices. Empirical studies show that symmetric cryptography is practical on constrained microcontrollers, while frequent public-key operations incur significant energy and latency costs [15, 21]. This motivates minimising asymmetric computation on the implant. *Chronos* follows this direction by restricting the IMD to symmetric primitives while achieving forward secrecy through state evolution and backend-assisted recovery.

Human Values, Safety, and Emergency Access. Qualitative studies show that IMD security must respect clinical workflows, patient autonomy, and emergency access [3, 4]. Mechanisms that risk lockout or impose complex procedures are unlikely to be adopted. *Chronos* incorporates these constraints by supporting recovery from desynchronisation and device loss, while enabling controlled, auditable emergency access without weakening long-term confidentiality.

In summary, prior work addresses pairing, authorisation, proximity protection, and usability, but not forward secrecy and recoverability across the implant lifecycle under realistic compromise assumptions. *Chronos* fills this gap with a cryptographic session-lifecycle approach providing forward secrecy, robustness to state inconsistency, and distributed trust without public-key operations on the implant.

10 Conclusion

Implantable Medical Devices operate for many years in adversarial and evolving environments, where device loss, backend compromise, and eventual physical access are realistic. In such settings, reliance on long-lived secrets exposes sensitive medical data under retrospective compromise, yet conventional approaches to forward secrecy remain difficult to deploy due to strict energy and safety con-

straints. This paper shows that forward secrecy for IMDs can be achieved without relying on public-key cryptography on the implant. We presented *Chronos*, a symmetric-only protocol that evolves cryptographic state across sessions while delegating consistency, recovery, and emergency access to a threshold-based keyservers. This design avoids single points of failure, preserves patient-centric control, and remains robust under desynchronisation, message loss, and device unavailability. Through formal verification and a complete end-to-end implementation on both 16-bit and 32-bit platforms, we demonstrate that forward secrecy is practical for IMDs, with negligible impact on device energy budgets. More broadly, our results suggest that lifecycle-oriented security, combining lightweight state evolution with distributed recovery, is a practical design paradigm for safety-critical and resource-constrained systems beyond IMDs.

References

1. Avoine, G., Canard, S., Ferreira, L.: Symmetric-Key Authenticated Key Exchange (SAKE) with Perfect Forward Secrecy. In: CT-RSA (2020)
2. Blanchet, B., Smyth, B., Cheval, V., Sylvestre, M.: ProVerif 2.00: Automatic Cryptographic Protocol Verifier, user manual and tutorial. Version from **16** (2018)
3. Denning, T., Borning, A., Friedman, B., Gill, B.T., Kohno, T., Maisel, W.H.: Patients, Pacemakers, and Implantable Defibrillators: Human Values and Security for Wireless Implantable Medical Devices. In: ACM International Conference on Human Factors in Computing Systems (CHI) (2010)
4. Denning, T., Kramer, D.B., Friedman, B., Reynolds, M.R., Gill, B., Kohno, T.: CPS: Beyond Usability: Applying Value Sensitive Design Based Methods to Investigate Domain Characteristics for Security for Implantable Cardiac Devices. In: Annual Computer Security Applications Conference (ACSAC) (2014)
5. Diffie, W., Van Oorschot, P.C., Wiener, M.J.: Authentication and Authenticated Key Exchanges. Designs, Codes and cryptography **2**(2) (1992)
6. Dolev, D., Yao, A.: On The Security Of Public Key Protocols. IEEE Transactions on Information Theory (1983)
7. Duttagupta, S., Marin, E., Singelée, D., Preneel, B.: HAT: Secure and Practical Key Establishment for Implantable Medical Devices. In: Proceedings of the Thirteenth ACM Conference on Data and Application Security and Privacy. CODASPY '23 (2023). `https://doi.org/10.1145/3577923.3583646`
8. Gollakota, S., Hassanieh, H., Ransford, B., Katabi, D., Fu, K.: They Can Hear Your Heartbeats: Non-Invasive Security for Implantable Medical Devices. In: ACM SIGCOMM (2011)
9. Halperin, D., Heydt-Benjamin, T.S., Ransford, B., Clark, S.S., Defend, B., Morgan, W., Fu, K., Kohno, T., Maisel, W.H.: Pacemakers and Implantable Cardiac Defibrillators: Software Radio Attacks and Zero-Power Defenses. In: IEEE Symposium on Security and Privacy (S&P) (2008)
10. Hu, C., Cheng, X., Zhang, F., Wu, D., Liao, X., Chen, D.: OPFKA: Secure and Efficient Ordered-Physiological-Feature-Based Key Agreement for Wireless Body Area Networks. In: IEEE International Conference on Computer Communications (INFOCOM) (2013)
11. Lewis-Pye, A., Roughgarden, T.: Permissionless consensus. arXiv preprint arXiv:2304.14701 (2023)

12. Li, C., Raghunathan, A., Jha, N.K.: Hijacking an Insulin Pump: Security Attacks and Defenses for a Diabetes Therapy System. In: IEEE International Conference on e-Health Networking, Applications and Services (Healthcom) (2011)
13. Marin, E., Argones Rúa, E., Singelée, D., Preneel, B.: On the Difficulty of Using Patient's Physiological Signals in Cryptographic Protocols. In: ACM Symposium on Access Control Models and Technologies (SACMAT) (2019)
14. Marin, E., Singelée, D., Garcia, F.D., Chothia, T., Willems, R., Preneel, B.: On the (in)Security of the Latest Generation Implantable Cardiac Defibrillators and How to Secure Them. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. ACSAC '16 (2016)
15. Marin, E., Singelée, D., Yang, B., Verbauwhede, I., Preneel, B.: On the Feasibility of Cryptography for a Wireless Insulin Pump System. In: ACM Conference on Data and Application Security and Privacy (CODASPY) (2016)
16. Marin, E., Singelée, D., Yang, B., Volski, V., Vandenbosch, G.A.E., Nuttin, B., Preneel, B.: Securing Wireless Neurostimulators. In: ACM Conference on Data and Application Security and Privacy (CODASPY) (2018)
17. Neu, J., Tas, E.N., Tse, D.: Ebb-and-flow protocols: A resolution of the availability-finality dilemma. In: 2021 IEEE Symposium on Security and Privacy (SP). pp. 446–465. IEEE (2021)
18. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm (raft). In: Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 14). USENIX Association (2014)
19. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) Advances in Cryptology — EUROCRYPT '99. Lecture Notes in Computer Science, vol. 1592 (1999)
20. Rasmussen, K.B., Castelluccia, C., Heydt-Benjamin, T.S., Capkun, S.: Proximity-Based Access Control for Implantable Medical Devices. In: ACM Conference on Computer and Communications Security (CCS) (2009)
21. Rostami, M., Burleson, W., Juels, A., Koushanfar, F.: Balancing Security and Utility in Medical Devices? In: ACM Design Automation Conference (DAC) (2013)
22. Rostami, M., Juels, A., Koushanfar, F.: Heart-to-Heart (H2H): Authentication for Implanted Medical Devices. In: ACM Conference on Computer and Communications Security (CCS) (2013)
23. Rushanan, M., Rubin, A.D., Kune, D.F., Swanson, C.M.: SoK: Security and Privacy in Implantable Medical Devices and Body Area Networks. In: IEEE Symposium on Security and Privacy (S&P) (2014)
24. Sarenche, R., Tas, E.N., Monnot, B., Schwarz-Schilling, C., Preneel, B.: Commitment attacks on ethereum's reward mechanism. In: 2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P). pp. 504–526. IEEE (2025)
25. Seepers, R.M., Wang, W., de Haan, G., Sourdis, I., Strydis, C.: Attacks on Heartbeat-Based Security Using Remote Photoplethysmography. IEEE Journal of Biomedical and Health Informatics **22**(3) (2018)
26. Shamir, A.: How to share a secret. Communications of the ACM **22**(11) (Nov 1979)
27. Tippenhauer, N.O., Malisa, L., Ranganathan, A., Capkun, S.: On Limitations of Friendly Jamming for Confidentiality. In: IEEE Symposium on Security and Privacy (S&P) (2013)
28. Xu, F., Qin, Z., Tan, C.C., Wang, B., Li, Q.: IMDGuard: Securing Implantable Medical Devices with the External Wearable Guardian. In: IEEE International Conference on Computer Communications (INFOCOM) (2011)

11 Primitives and Functions: Supplementary

Key-Update Functions $\mathsf{Upd}(\cdot)$ and $\mathsf{Upd}_{\mathbb{Z}_P}(\cdot)$. The protocol uses one-way update functions to evolve long-term secrets across sessions. For λ-bit keys, the update function is

$$\mathsf{Upd} : \{0,1\}^\lambda \to \{0,1\}^\lambda.$$

For key shares in $\mathbb{Z}_P$, an analogous update function is defined as

$$\mathsf{Upd}_{\mathbb{Z}_P} : \mathbb{Z}_P \to \mathbb{Z}_P.$$

Both functions are assumed to be one-way, i.e., given $\mathsf{Upd}(x)$ (resp., $\mathsf{Upd}_{\mathbb{Z}_P}(y)$), it is computationally infeasible to recover $x \in \{0,1\}^\lambda$ (resp., $y \in \mathbb{Z}_P$).

Key-Derivation Function $\mathsf{KDF}(\cdot)$. The key-derivation function

$$\mathsf{KDF} : \{0,1\}^* \to \{0,1\}^\lambda$$

derives session keys and intermediate secret values from a concatenation of input bit strings, including long-term keys, nonces, and public domain-separation labels. The output is a fixed-length key in $\{0,1\}^\lambda$.

Message Authentication Code $\mathsf{MAC}(\cdot)$. The message authentication code

$$\mathsf{MAC} : \{0,1\}^\lambda \times \{0,1\}^* \to \{0,1\}^\lambda$$

is a keyed function used to ensure message integrity and authenticity. The notation $\mathsf{MAC}_k(m)$ denotes the authentication tag computed over message m under key k. Public labels may be included in m for domain separation.

Bitstring–Field Mapping $(\mathsf{Encode}, \mathsf{Decode})$. The protocol uses a fixed public mapping from λ-bit strings into $\mathbb{Z}_P$,

$$\mathsf{Encode} : \{0,1\}^\lambda \to \mathbb{Z}_P, \qquad \mathsf{Decode} : \mathbb{Z}_P \to \{0,1\}^\lambda,$$

such that Decode is a left-inverse of Encode, i.e.,

$$\mathsf{Decode}(\mathsf{Encode}(x)) = x \quad \text{for all } x \in \{0,1\}^\lambda.$$

Equivalently, Encode is injective and its image $\mathrm{Im}(\mathsf{Encode}) \subseteq \mathbb{Z}_P$ has size 2^λ. We only require Decode to be correct on $\mathrm{Im}(\mathsf{Encode})$; for inputs in $\mathbb{Z}_P \setminus \mathrm{Im}(\mathsf{Encode})$, Decode may be defined arbitrarily (e.g., by returning $H_p(z) \in \{0,1\}^\lambda$ for a fixed public hash function $H_p : \mathbb{Z}_P \to \{0,1\}^\lambda$).

Extraction Hash $\mathsf{H}_{\mathsf{extract}}(\cdot)$. The protocol uses a fixed public extraction hash

$$\mathsf{H}_{\mathsf{extract}} : \mathbb{Z}_P \to \{0,1\}^\lambda$$

to derive a λ-bit value from an arbitrary field element. This function is used when the protocol needs to map an element of $\mathbb{Z}_P$ to a bit string for subsequent key derivation.

Secret-Sharing Scheme $\mathsf{SC}(\cdot)$ **[26].** The secret-sharing scheme

$$\mathsf{SC} : \{0,1\}^\lambda \times [n] \to \mathbb{Z}_P$$

is instantiated as Shamir's (t, n) threshold scheme over $\mathbb{Z}_P$. Let a secret key be $k \in \{0,1\}^\lambda$. We first map it into the field by setting $s \leftarrow \mathsf{Encode}(k) \in \mathbb{Z}_P$. We then sample coefficients $r_1, \ldots, r_{t-1} \xleftarrow{\$} \mathbb{Z}_P$ and define the degree-$(t-1)$ polynomial

$$f(x) = s + r_1 x + r_2 x^2 + \cdots + r_{t-1} x^{t-1} \bmod P.$$

For each $j \in [n]$, the j-th share is the field element

$$\mathsf{SC}(k, j) = f(j) \in \mathbb{Z}_P.$$

Any subset of at least t shares enables reconstruction of the secret, while fewer than t shares reveal no information about s (and hence about k). Reconstruction is performed by applying Lagrange interpolation over $\mathbb{Z}_P$ to recover $s = f(0)$, and then outputting $k \leftarrow \mathsf{Decode}(s)$.

Public-Key Encryption and Digital Signatures. Let $(\mathsf{PK}, \mathsf{SK})$ be a public/secret key pair. We denote by $\mathsf{Enc}_{\mathsf{PK}}(m)$ the encryption of a message m under public key PK. Analogously, we denote by $\mathsf{Dec}_{\mathsf{SK}}(c)$ the decryption of a ciphertext c under private key SK. We also denote by $\sigma_{\mathsf{SK}}(m)$ the signature on message m under secret key SK.

Verification Algorithms. For MAC verification, we write $\mathsf{Verif}(\tau, k) = 1$ if and only if the authentication tag τ is valid under key k. For signature verification, we write $\mathsf{Verif}(\sigma, \mathsf{PK}) = 1$ if and only if σ is a valid signature under public key PK.

Security Level and Parameter Selection. The parameter λ is the nominal security parameter used to size cryptographic values in Chronos. The concrete security level is the minimum of the concrete guarantees of the instantiated primitives and depends on the number of protocol executions, verification attempts, and state updates. For a λ-bit MAC tag, the generic online tag-guessing probability over q_{ver} verification attempts is at most $q_{\mathsf{ver}}/2^\lambda$, and the concrete MAC must be instantiated so that its SUF-CMA advantage is below the desired security target. A λ-bit KDF output gives a generic key-guessing bound of $q_{\mathsf{key}}/2^\lambda$, and the KDF must be instantiated so that its pseudorandomness advantage is below the target. The update functions must be one-way, and an iterated update chain over a λ-bit state also has a generic birthday/cycle bound of about $q_{\mathsf{upd}}^2/2^{\lambda+1}$ over q_{upd} updates; therefore, λ must be chosen with the intended deployment lifetime in mind. Public-key encryption and signatures must be instantiated so that their IND-security and EUF-CMA advantages meet the desired target. Shamir secret sharing provides information-theoretic secrecy against fewer than t shares.

12 Requirements of the Replicated Ledger

The replicated ledger $\mathsf{DB}_{\mathrm{rep}}$ is the key-server component that serialises state-update log records for Chronos. Its role is to provide a single committed order for

key-server state transitions, so that key servers maintain compatible per-sensor states despite message delay, crashes, recovery, and concurrent doctor requests. The replicated ledger may contain transient, uncommitted records during replication. Chronos state evolution, however, is driven only by the committed ledger. The ledger does not perform the cryptographic state update itself; it commits an ordered sequence of update records, while the replicated state machine determines which committed records are applied as valid Chronos state transitions.

Key-server assumptions. We recall the key-server assumptions relevant to the replicated ledger. The key servers are *honest-but-curious*: they execute the protocol as specified, but may be temporarily unavailable due to crashes or disconnection. The inter-key-server network is synchronous among online key servers: messages between online key servers are delivered within a known delay bound Δ_{delay}. At any time, at least t key servers are online and responsive, where $t \geq \lfloor n/2 \rfloor + 1$.

Notation. For two finite sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_q)$, we write $X \preceq Y$ if $m \leq q$ and $x_r = y_r$ for every $r \in \{1, \ldots, m\}$. A key-server–sensor state-update log record for sensor S_i is denoted by $\mathcal{L}_i^{\mathsf{KS-S}} = (\mathsf{sid}, \mathsf{ID}_{S_i}, N_i^{\mathsf{req}}, \{\Delta_{i,j}\}_{j\in[n]})$, where N_i^{req} is the predecessor update counter claimed by the log record. Let $\mathsf{Ledger}_j(T)$ denote the replicated ledger in the view of key server KS_j at time T, possibly including records that have not yet been committed. Let $\mathsf{CLedger}_j(T)$ denote the committed ledger in the view of key server KS_j at time T, where $\mathsf{CLedger}_j(T) \preceq \mathsf{Ledger}_j(T)$. For a sensor index i, let $\mathsf{CLedger}_j(T; i)$ denote the ordered subsequence of $\mathsf{CLedger}_j(T)$ containing the committed state-update log records for sensor S_i, preserving their order in $\mathsf{CLedger}_j(T)$. Two state-update log records for the same sensor S_i are conflicting if they have the same predecessor update counter but different contents. That is, $\mathcal{L}_i^{\mathsf{KS-S}} = (\mathsf{sid}, \mathsf{ID}_{S_i}, N_i^{\mathsf{req}}, \{\Delta_{i,j}\}_{j\in[n]})$ and $\widetilde{\mathcal{L}}_i^{\mathsf{KS-S}} = (\widetilde{\mathsf{sid}}, \mathsf{ID}_{S_i}, \widetilde{N}_i^{\mathsf{req}}, \{\widetilde{\Delta}_{i,j}\}_{j\in[n]})$ conflict if $N_i^{\mathsf{req}} = \widetilde{N}_i^{\mathsf{req}} \quad \text{and} \quad \mathcal{L}_i^{\mathsf{KS-S}} \neq \widetilde{\mathcal{L}}_i^{\mathsf{KS-S}}$.

Ledger requirements. We now state the three replicated-ledger properties used by Chronos: *i) committed-ledger consistency, ii) committed-ledger liveness under synchrony and crash faults, and iii) Chronos transition validity.* The first two properties are generic replicated-ledger properties under synchrony and crash faults [11, 17]. The third is Chronos-specific and is enforced by the replicated state machine when deciding whether a committed log record is applied as a valid key-server state transition.

Definition 1 (Committed-ledger consistency). *A replicated ledger achieves committed-ledger consistency if the following two conditions hold. First, for any two key servers $\mathsf{KS}_j, \mathsf{KS}_\ell$ and any times T, T' at which their committed ledger views are defined, $\mathsf{CLedger}_j(T) \preceq \mathsf{CLedger}_\ell(T')$ or $\mathsf{CLedger}_\ell(T') \preceq \mathsf{CLedger}_j(T)$. Second, for every key server KS_j and any times $T \leq T'$, $\mathsf{CLedger}_j(T) \preceq \mathsf{CLedger}_j(T')$.*

Committed-ledger consistency means that the committed ledger has a single history, although different key servers may observe different-length prefixes of that history. The first condition rules out forks across key servers, while the second rules out rollback at an individual key server. Hence, if a prefix $P =$

$(\mathcal{L}_1, \ldots, \mathcal{L}_q)$ is committed in the view of some key server KS_j at time T, then any committed ledger view of any key server that contains at least q records must have the same prefix P. Thus, a committed log record cannot be removed, reordered, or replaced by a different committed log record.

The same implication holds for sensor-specific update histories. Since $\mathsf{CLedger}_j(T; i)$ contains exactly the committed state-update records for S_i, committed-ledger consistency implies $\mathsf{CLedger}_j(T; i) \preceq \mathsf{CLedger}_\ell(T'; i)$ or $\mathsf{CLedger}_\ell(T'; i) \preceq \mathsf{CLedger}_j(T; i)$. Therefore, all key servers agree on a single committed order of update records for each sensor S_i, up to possible lag. A recovered key server may be behind, but it cannot recover into a conflicting per-sensor history; it catches up by replaying the committed ledger.

Definition 2 (Committed-ledger liveness under synchrony and crash faults). *A replicated ledger achieves committed-ledger liveness under Δ_{delay}-synchrony and t-out-of-n crash-fault availability if there exists a bound $\Gamma_{\mathsf{commit}} = O(\Delta_{\mathsf{delay}})$ such that the following holds. Suppose a valid state-update log record $\mathcal{L}_i^{\mathsf{KS-S}}$ is submitted at time T by an online key server to the replicated ledger. Then for every key server KS_ℓ that is online at time T', where $T' \geq T + \Gamma_{\mathsf{commit}}$, the committed ledger in the view of KS_ℓ at time T' contains $\mathcal{L}_i^{\mathsf{KS-S}}$.*

This is the liveness property required by Chronos. Under an online quorum and a synchronous inter-key-server network, any submitted state-update record is eventually included in the committed ledger. In particular, for such a submitted record $\mathcal{L}_i^{\mathsf{KS-S}}$, every key server KS_j that is online at some time $T' \geq T + \Gamma_{\mathsf{commit}}$ satisfies $\mathcal{L}_i^{\mathsf{KS-S}} \in \mathsf{CLedger}_j(T')$.

Note on online key servers. In our analysis, we follow the standard slot-based abstraction in distributed-systems analyses [17, 24]: an awake node at slot T starts slot T with all messages scheduled for delivery before T already available, and then executes the protocol faithfully. Therefore, a key server being online at the start of slot T means that it is awake and synchronised with the replicated ledger, and that all ledger messages delivered before slot T have been incorporated into its local view. Messages sent to that key server during or after slot T may still be delayed by up to Δ_{delay}.

Definition 3 (Chronos transition validity). *For each sensor S_i, let $N_i^{\mathsf{state}}(P)$ denote the key-server update counter obtained after sequentially processing the records in a committed per-sensor prefix $P \preceq \mathsf{CLedger}_j(T; i)$, starting from the initial state $N_i^{\mathsf{state}}(\emptyset) = 0$. Let a committed state-update log record $\mathcal{L}_i^{\mathsf{KS-S}} = (\mathsf{sid}, \mathsf{ID}_{S_i}, N_i^{\mathsf{req}}, \{\Delta_{i,j}\}_{j\in[n]})$ appear immediately after prefix P, and let P' denote the sequence obtained by appending $\mathcal{L}_i^{\mathsf{KS-S}}$ to P. The record $\mathcal{L}_i^{\mathsf{KS-S}}$ is transition-valid if and only if $N_i^{\mathsf{req}} = N_i^{\mathsf{state}}(P)$. If transition-valid, it is applied as a key-server state transition, yielding $N_i^{\mathsf{state}}(P') = N_i^{\mathsf{state}}(P) + 1$. Otherwise, the record is stale and has no effect on the key-server cryptographic state, yielding $N_i^{\mathsf{state}}(P') = N_i^{\mathsf{state}}(P)$.*

This requirement is Chronos-specific. It is not merely a ledger-ordering property; it is the application-level validity rule that determines whether a committed

record changes key-server cryptographic state. For every sensor S_i, each predecessor update counter h is consumed at most once: if several committed records for S_i have $N_i^{\mathsf{req}} = h$, only the first one processed while $N_i^{\mathsf{state}} = h$ is applied; all later ones are stale. This handles concurrent doctor sessions for the same sensor: the replicated ledger may serialise multiple competing proposals, but the ledger layer applies at most one of them. This prevents the same key-server state from being consumed twice and is essential for the sensor–key-server bounded-desynchronisation argument.

Suitable realisation. Under the Chronos threat model, $\mathsf{DB}_{\mathrm{rep}}$ can be realised by a permissioned crash-fault-tolerant replicated ledger, such as a Raft-style [18] replicated ledger, provided that the ledger layer enforces the Chronos transition-validity rule above.

Let the *Raft leader* denote the leader of the replicated-ledger protocol, which is distinct from the Chronos key-server leader that coordinates a session. In a stable-leader Raft-style implementation, if a state-update record is first submitted by an arbitrary online key server, the record reaches the Raft leader within at most Δ_{delay}. The Raft leader then replicates the record to followers and receives quorum acknowledgements within one leader–follower round trip, at most $2\Delta_{\mathsf{delay}}$. The commit decision is then disseminated to online followers in the next heartbeat or append message, requiring one additional delay. Therefore, online key servers learn the committed record within $\Gamma_{\mathsf{commit}} = 4\Delta_{\mathsf{delay}}$.

If the Chronos key-server leader is also the Raft leader, the forwarding delay is absent. In that case, the Raft leader can commit the record after quorum replication within $2\Delta_{\mathsf{delay}}$, and online followers learn the commit decision after one additional delay, giving $\Gamma_{\mathsf{commit}} = 3\Delta_{\mathsf{delay}}$. Thus, the record can be processed by the Raft leader within only $2\Delta_{\mathsf{delay}}$.

13 Session-Key Establishment Protocol: Supplementary

This section provides a detailed description of our distributed session-key establishment protocol. Figure 3 illustrates the message flow among the sensor S_i, the doctor's device D_{Dr}, and the leader key server $\mathsf{KS}_{\mathrm{leader}}$, while Fig. 4 zooms in on the key-server operations by showing the interactions between $\mathsf{KS}_{\mathrm{leader}}$, the remaining key servers $\{\mathsf{KS}_j\}_{j\in[n]}$, and the mobile device M_i^{user}.

- **Protocol initiation.** The doctor's device D_{Dr} samples a fresh nonce $r_{\mathrm{Dr}} \xleftarrow{\$} \{0,1\}^{\lambda}$. If starting a fresh session, it samples a fresh session identifier $\mathsf{sid} \xleftarrow{\$} \{0,1\}^{\lambda_{\mathsf{sid}}}$ and sets $\mathsf{UI}_{S_i} := \emptyset$. If resynchronising after m_{upd}, it sets sid as the session identifier of the resync-required session and sets $\mathsf{UI}_{S_i} = (\tau_{\mathsf{upd}}, r_{\mathrm{upd}})$. It sends
$$m_1 = \mathsf{sid} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, \mathsf{UI}_{S_i} \,\|\, r_{\mathrm{Dr}}$$
to sensor S_i.
- **Sensor-side update check.** Upon receiving m_1, sensor S_i processes UI_{S_i}. If $\mathsf{UI}_{S_i} = \emptyset$, the protocol proceeds to the next step. Otherwise, if
$$\mathsf{UI}_{S_i} = \big(\tau_{\mathsf{upd}},\ r_{\mathrm{upd}}\big),$$

the sensor retrieves the pending tuple $(\mathsf{sid}^\star, \mathsf{ID}^\star_{\mathrm{Dr}}, r^\star_{\mathrm{Dr}}, r^\star_{S_i})$ stored after the previous m_2. It aborts if no such tuple exists, if $\mathsf{sid} \neq \mathsf{sid}^\star$, or if $\mathsf{ID}_{\mathrm{Dr}} \neq \mathsf{ID}^\star_{\mathrm{Dr}}$. The sensor verifies the update token by checking whether

$$\tau_{\mathsf{upd}} \stackrel{?}{=} \mathsf{MAC}_{\mathsf{Upd}(k_i^{\mathrm{sync}})}\Big(\mathsf{sid}^\star \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}^\star_{\mathrm{Dr}} \,\|\, r^\star_{S_i} \,\|\, r^\star_{\mathrm{Dr}} \,\| \\ r_{\mathrm{upd}} \,\|\, L = \texttt{"KS-S:mac-upd"}\Big).$$

If verification succeeds, the sensor applies the update procedure UPD_{S_i} (Sect. 4.4) and proceeds to the next step using the fresh r_{Dr} received in m_1; otherwise, it aborts.

– **Sensor-side authentication tag generation.** The sensor samples a nonce $r_{S_i} \xleftarrow{\$} \{0,1\}^\lambda$. It first derives a MAC key for the user-sensor authentication tag as

$$k_i^{\text{U-S,mac}} \;:=\; \mathsf{KDF}\Big(k_i^{\text{U-S}} \,\|\, \mathsf{sid} \,\|\, L = \texttt{"S-U:kdf-mac"}\Big).$$

It then computes

$$\tau_1^S = \mathsf{MAC}_{k_i^{\mathrm{sync}}}\Big(\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \\ \,\|\, L = \texttt{"S-KS:mac"}\Big),$$

and

$$\tau_2^S = \mathsf{MAC}_{k_i^{\text{U-S,mac}}}\Big(\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \\ \,\|\, L = \texttt{"S-U:mac"}\Big).$$

The sensor sends

$$m_2 = \mathsf{sid} \,\|\, \tau_1^S \,\|\, \tau_2^S \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i}$$

to the doctor's device and temporarily stores sid, $\mathsf{ID}_{\mathrm{Dr}}$, r_{Dr}, and r_{S_i} for the current session until completion, timeout, abort, or resynchronisation.

– **Doctor request to key servers.** The doctor's device generates an ephemeral key pair $(\mathsf{ePK}_{\mathrm{Dr}}, \mathsf{eSK}_{\mathrm{Dr}})$ for an additively homomorphic public-key encryption scheme[¶]. The plaintext modulus of this scheme must be sufficiently larger than the prime modulus P to ensure that homomorphic computations on values in $\mathbb{Z}_P$ do not wrap around the encryption modulus.
The doctor's device also samples a timestamp t_{Dr}. Let $\mathsf{Emg} \in \{0,1\}$ denote whether emergency access is requested ($\mathsf{Emg} = 1$) or not requested ($\mathsf{Emg} = 0$). The doctor constructs the MAC token

$$\mathsf{mac_token}_{\mathrm{Dr}} \;:= \tau_1^S \,\|\, \tau_2^S \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}},$$

[¶]In our implementation, we instantiate this scheme using the Paillier cryptosystem [19].

and computes the commitment

$$h_{\mathrm{Dr}} \;:=\; H(\mathsf{mac_token}_{\mathrm{Dr}}),$$

where $H : \{0,1\}^* \to \{0,1\}^\lambda$ is a collision-resistant hash function.
The doctor constructs the core request

$$\mathsf{core_req}_{\mathrm{Dr}} \;:= \mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, \mathsf{ePK}_{\mathrm{Dr}} \,\|\, t_{\mathrm{Dr}} \,\|\, \mathsf{Emg} \,\|\, h_{\mathrm{Dr}}$$

and signs $\mathsf{core_req}_{\mathrm{Dr}}$ using its long-term private key:

$$\sigma_{\mathrm{Dr}} \;:=\; \sigma_{\mathsf{SK}_{\mathrm{Dr}}}\big(\mathsf{core_req}_{\mathrm{Dr}}\big).$$

The doctor finally generates

$$m_3 = \mathsf{mac_token}_{\mathrm{Dr}} \,\|\, \mathsf{core_req}_{\mathrm{Dr}} \,\|\, \sigma_{\mathrm{Dr}},$$

and sends m_3 to the key-server service.

– **Leader selection and synchronisation.** The key-server service designates an online key server, denoted by $\mathsf{KS}_{\mathrm{leader}}$, to coordinate the current session. The leader retrieves the doctor's certificate from the replicated database, verifies σ_{Dr} on $\mathsf{core_req}_{\mathrm{Dr}}$, and checks the commitment by recomputing $H(\mathsf{mac_token}_{\mathrm{Dr}})$ and comparing it to the h_{Dr} value embedded in $\mathsf{core_req}_{\mathrm{Dr}}$. If verification fails, it aborts.
Let $\mathcal{K}^{\mathrm{server}}_{i,\mathrm{leader}}$ denote the local key array stored by $\mathsf{KS}_{\mathrm{leader}}$ for sensor S_i at the time the leader processes the request:

$$\mathcal{K}^{\mathrm{server}}_{i,\mathrm{leader}} = \Big\{(N_i^{\mathrm{state}}, k_i^{\mathrm{sync,old}}, k_i^{\mathrm{sync,new}}),\; k^{\mathrm{KS\text{-}S}}_{i,\mathrm{leader}},\; (k^{\mathrm{emg}}_{i,\mathrm{leader}},\; r_i^{\mathrm{U,emg}})\Big\}.$$

For the rest of this step, we write this array with respect to the latest committed state-update record incorporated in the leader's local view of the replicated ledger.[‖]
The leader verifies τ_1^S under $k_i^{\mathrm{sync,old}}$. If verification succeeds, then S_i is one update state behind. The leader generates update information $\mathsf{UI}_{S_i} = \big(\tau_{\mathsf{upd}}, r_{\mathrm{upd}}\big)$ where $r_{\mathrm{upd}} \xleftarrow{\$} \{0,1\}^\lambda$ and

$$\tau_{\mathsf{upd}} = \mathsf{MAC}_{k_i^{\mathrm{sync,new}}}\Big(\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \,\|\, r_{\mathrm{upd}} \,\|\, L = \texttt{"KS-S:mac-upd"}\Big).$$

[‖] Due to Δ_{delay}-synchrony, the leader's local view may lag behind the most advanced committed ledger view. If the sensor has already advanced beyond the leader's local state, τ_1^S will not verify under the leader's local $(k_i^{\mathrm{sync,old}}, k_i^{\mathrm{sync,new}})$, and the leader aborts. If the sensor has not yet advanced beyond that stale state, the leader may treat the sensor as synchronised and submit a record with a stale N_i^{req}. Such a record is rejected by Chronos transition validity, since the corresponding predecessor counter has already been consumed in a more advanced committed prefix. Hence, a lagging leader can delay session establishment but cannot create a conflicting sensor–key-server state.

It responds to the doctor with $m_{\text{upd}} = \mathsf{resync} \,\|\, \mathsf{UI}_{S_i}$, and the doctor resynchronises the protocol including UI_{S_i} in m_1.
Otherwise, the leader verifies τ_1^S under $k_i^{\text{sync,new}}$. If verification succeeds, the sensor and key servers are synchronised and the leader proceeds; if not, it aborts.

– **Session request broadcast.** After successful synchronisation, the leader key server $\mathsf{KS}_{\text{leader}}$ generates an ephemeral public–private key pair $(\mathsf{ePK}_{\text{leader}}, \mathsf{eSK}_{\text{leader}})$ for the current session and broadcasts the following message to all key servers over authenticated and encrypted channels:

$$m_4 = \mathsf{core_req}_{\text{Dr}} \,\|\, \sigma_{\text{Dr}} \,\|\, N_i^{\mathsf{req}} \,\|\, \mathsf{ePK}_{\text{leader}},$$

where N_i^{req} denotes the latest committed state counter corresponding to sensor S_i stored locally at the leader key server.

Non-emergency case ($\mathsf{Emg} = 0$*).* In parallel with broadcasting m_4, the leader constructs

$$m_5' = \mathsf{core_req}_{\text{Dr}} \,\|\, \sigma_{\text{Dr}} \,\|\, \mathsf{Cert}_{\text{Dr}} \,\|\, \mathsf{Info}_{\text{Dr}} \,\|\, \tau_2^S \,\|\, r_{S_i} \,\|\, r_{\text{Dr}},$$

signs it, and sends $m_5 = m_5' \,\|\, \sigma_{\mathsf{SK}_{\text{leader}}}(m_5')$ to the user's mobile device M_i^{user}. Here, certificate $\mathsf{Cert}_{\text{Dr}}$ enables verification of σ_{Dr}, and $\mathsf{Info}_{\text{Dr}}$ contains user-displayable doctor metadata for consent.

Emergency case ($\mathsf{Emg} = 1$*).* If emergency access is requested, the leader key server does not contact M_i^{user}.

– **User consent message (if** $\mathsf{Emg} = 0$**).** Upon receiving m_5, the user's mobile device verifies the leader's signature and verifies τ_2^S under $k_i^{\text{U-S,mac}}$ derived from $k_i^{\text{U-S}}$. If verification succeeds, it displays $\mathsf{Info}_{\text{Dr}}$ and requests consent. Upon consent, the mobile device samples $r_{\text{U}}^{\text{non-emg}} \xleftarrow{\$} \{0,1\}^\lambda$, derives its session-key contribution

$$k_{\text{session}}^{\text{U-S}} = \mathsf{KDF}\Big(k_i^{\text{U-S}} \,\|\, r_{S_i} \,\|\, r_{\text{U}}^{\text{non-emg}} \,\|\, L = \texttt{"U-S:non-emg key"}\Big),$$

and sends

$$m_6' = \mathsf{sid} \,\|\, \mathsf{Enc}_{\mathsf{ePK}_{\text{Dr}}}\big(\mathsf{Encode}(k_{\text{session}}^{\text{U-S}})\big) \,\|\, r_{\text{U}}^{\text{non-emg}}$$

to the leader key server, together with a signature, forming $m_6 = m_6' \,\|\, \sigma_{\mathsf{SK}_i^{\text{user}}}(m_6')$. If the user does not provide consent, or if the user's mobile device is unavailable, the session will be aborted. In such cases, the doctor may decide whether to restart the key-establishment protocol with the emergency flag enabled, allowing the session to proceed without the user's contribution.

– **Key-server share submission to leader.** Each online key server KS_j processes m_4 as follows. First, KS_j verifies the validity of $\mathsf{core_req}_{\text{Dr}}$ contained in m_4. Concretely, KS_j verifies σ_{Dr} under the doctor's certified public key and checks that the timestamp t_{Dr} embedded in $\mathsf{core_req}_{\text{Dr}}$ is fresh according to its local policy. These checks are necessary to ensure that $\mathsf{core_req}_{\text{Dr}}$

is generated by a legitimate doctor. In particular, this prevents a curious key-server leader from faking emergency mode and the doctor's ephemeral public key to obtain emergency shares, thereby compromising session-key confidentiality.
The next step for KS_j is to verify the request index corresponding to sensor S_i. Since a key server's local state-update counter may temporarily lag behind the latest committed ledger state (e.g., due to delayed log delivery or temporary unavailability), KS_j first checks whether its locally stored state-update counter for S_i equals N_i^{req} carried in m_4. Only if this check succeeds does KS_j generate

$$m'_{7,j} = \mathsf{sid} \,\|\, N_i^{\mathsf{req}} \,\|\, \mathsf{Enc}_{\mathsf{ePK}_{\text{leader}}}(k_{i,j}^{\text{KS-S}}) \ .$$

Otherwise, the request is treated as stale or conflicting, and KS_j aborts and returns no share.

Non-emergency case ($\mathsf{Emg} = 0$*).* Under non-emergency access, KS_j sets $m_{7,j} = m'_{7,j}$.

Emergency case ($\mathsf{Emg} = 1$*).* Under emergency access, KS_j additionally appends the encrypted emergency share under the doctor's ephemeral public key ePK_{Dr} included in m_4:

$$m_{7,j} = m'_{7,j} \,\|\, \mathsf{Enc}_{\mathsf{ePK}_{\text{Dr}}}(k_{i,j}^{\text{emg}}).$$

Finally, KS_j sends $m_{7,j}$ to $\mathsf{KS}_{\text{leader}}$ over an authenticated and encrypted channel.

– **Distributed encrypted key derivation.** In the non-emergency setting, once $\mathsf{KS}_{\text{leader}}$ has collected valid responses from at least $t-1$ other key servers and has received m_6, it proceeds; in the emergency setting, it proceeds analogously after collecting valid responses from at least $t-1$ other key servers.
First, the leader decrypts the received ciphertexts $\mathsf{Enc}_{\mathsf{ePK}_{\text{leader}}}(k_{i,j}^{\text{KS-S}})$ to obtain field shares $k_{i,j}^{\text{KS-S}} \in \mathbb{Z}_P$, and reconstructs the underlying field secret $s_i \in \mathbb{Z}_P$ via Lagrange interpolation. It then recovers the corresponding bit-string key by decoding

$$k_i^{\text{KS-S}} \ := \ \mathsf{Decode}(s_i) \in \{0,1\}^\lambda.$$

Next, the leader computes the updated server–sensor key

$$k_i^{\text{KS-S,new}} \ := \ \mathsf{Upd}\big(k_i^{\text{KS-S}}\big).$$

For each $j \in [n]$, it derives the update token in $\mathbb{Z}_P$ as

$$\Delta_{i,j} = \mathsf{SC}\big(k_i^{\text{KS-S,new}},\, j\big) - \mathsf{Upd}_{\mathbb{Z}_P}\big(k_{i,j}^{\text{KS-S}}\big) \bmod P.$$

Note that all n shares $\mathsf{SC}(k_i^{\text{KS-S,new}}, j)$ are computable from any set of t shares; hence $\Delta_{i,j}$ can be computed for every server KS_j, regardless of

whether it is currently online or offline. After computing $\{\Delta_{i,j}\}_{j\in[n]}$, the leader securely erases the received shares $\{k_{i,j}^{\text{KS-S}}\}$ and the reconstructed update key $k_i^{\text{KS-S,new}}$. The leader submits the following server–sensor log entry to the replicated ledger:

$$\mathcal{L}_i^{\text{KS-S}} = \big(\mathsf{sid},\ \mathsf{ID}_{S_i},\ N_i^{\mathsf{req}},\ \{\Delta_{i,j} \mid j \in [n]\}\big),$$

where N_i^{req} denotes the latest committed state counter for sensor S_i stored locally at the leader.

The leader key server samples a fresh nonce $r_{\text{KS}} \xleftarrow{\$} \{0,1\}^\lambda$ and computes

$$\tau^{\text{KS}} = \mathsf{MAC}_{k_i^{\text{sync,new}}}\Big(\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\text{Dr}} \,\|\, r_{S_i} \,\|\, r_{\text{KS}} \,\|\, \mathsf{Emg} \,\|\, L = \texttt{"KS-S:mac"}\Big).$$

The server-side contribution to the session key is then derived as

$$k_{\text{session}}^{\text{KS-S}} = \mathsf{KDF}\Big(k_i^{\text{KS-S}} \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\text{KS}} \,\|\, L = \texttt{"KS-S:kdf"}\Big).$$

Non-emergency case ($\mathsf{Emg} = 0$*).* Let $u := \mathsf{Encode}(k_{\text{session}}^{\text{U-S}}) \in \mathbb{Z}_P$ and $v := \mathsf{Encode}(k_{\text{session}}^{\text{KS-S}}) \in \mathbb{Z}_P$. Using the encrypted user contribution $\mathsf{Enc}_{\mathsf{ePK}_{\text{Dr}}}(u)$ from m_6, the leader homomorphically computes

$$k_{\text{session}}^{\text{KS-U-S,enc}} = \mathsf{Enc}_{\mathsf{ePK}_{\text{Dr}}}\big(u + v\big).$$

The leader sets $r_{\text{U}} = r_{\text{U}}^{\text{non-emg}}$ as included in m_6.

Emergency case ($\mathsf{Emg} = 1$*).* The leader uses the encrypted emergency key shares $\mathsf{Enc}_{\mathsf{ePK}_{\text{Dr}}}(k_{i,j}^{\text{emg}})$ from at least $t-1$ key servers, together with its own emergency share, to homomorphically reconstruct $\mathsf{Enc}_{\mathsf{ePK}_{\text{Dr}}}(u)$, where $u \bmod P$ corresponds to the user-side contribution for this session. It then homomorphically computes

$$k_{\text{session}}^{\text{KS-U-S,enc}} = \mathsf{Enc}_{\mathsf{ePK}_{\text{Dr}}}\big(u + v\big).$$

In this case, the leader sets $r_{\text{U}} = r_i^{\text{U,emg}}$ from its local storage for sensor S_i (initialised/updated as in Sect. 4.4).

At this step, the leader securely erases the reconstructed key $k_i^{\text{KS-S}}$. Once $\mathcal{L}_i^{\text{KS-S}}$ is committed within Γ_{commit} (see Definition 2), the replicated state machine checks whether it is transition-valid (see Definition 3), namely, when the record is processed in committed ledger order after all preceding committed records for S_i, the key server's local counter N_i^{state} must equal the record's N_i^{req}. If $N_i^{\mathsf{state}} \neq N_i^{\mathsf{req}}$, the record is stale, has no effect on the key-server cryptographic state, and the leader aborts without sending m_8. If the

record is transition-valid, all online key servers, including the leader, update their local key material as specified in $\mathsf{UPD}_{\mathrm{KS}}$ (see Sect. 4.4). The leader then constructs

$$m'_8 = \mathsf{sid} \,\|\, \tau^{\mathrm{KS}} \,\|\, k_{\mathrm{session}}^{\text{KS-U-S,enc}} \,\|\, r_{\mathrm{KS}} \,\|\, r_{\mathrm{U}},$$

signs it, and sends

$$m_8 = m'_8 \,\|\, \sigma_{\mathsf{SK}_{\mathrm{leader}}}(m'_8)$$

to the doctor's device. If $\mathcal{L}_i^{\text{KS-S}}$ is not committed within Γ_{commit}, the leader aborts without sending m_8.

– **Session-key derivation at the doctor's device.** Upon receiving m_8, the doctor's device verifies the leader's signature and decrypts $k_{\mathrm{session}}^{\text{KS-U-S,enc}}$ to recover

$$\kappa_P := u+v \bmod P \;=\; \mathsf{Dec}_{\mathsf{eSK}_{\mathrm{Dr}}}\big(k_{\mathrm{session}}^{\text{KS-U-S,enc}}\big) \bmod P.$$

It then computes $\kappa := \mathsf{H}_{\mathsf{extract}}(\kappa_P) \in \{0,1\}^\lambda$ and derives the final session key as

$$k_{\mathrm{session}} = \mathsf{KDF}\Big(\kappa \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \,\|\, L = \texttt{"Dr-S:kdf"}\Big).$$

The doctor's device derives a MAC key

$$k^{\mathrm{Dr,mac}} = \mathsf{KDF}\Big(k_{\mathrm{session}} \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \,\|\, L = \texttt{"Dr-S:kdf-mac"}\Big),$$

and computes

$$\tau^{\mathrm{Dr}} = \mathsf{MAC}_{k^{\mathrm{Dr,mac}}}\Big(\mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\mathrm{KS}} \,\|\, r_{\mathrm{U}} \,\|\, r_{\mathrm{Dr}} \,\|\, L = \texttt{"Dr-S:mac"}\Big).$$

It sends

$$m_9 = \mathsf{sid} \,\|\, \tau^{\mathrm{KS}} \,\|\, \tau^{\mathrm{Dr}} \,\|\, \mathsf{Emg} \,\|\, r_{\mathrm{KS}} \,\|\, r_{\mathrm{U}}$$

to sensor S_i.

– **Session-key derivation at the sensor.** Upon receiving m_9, sensor S_i verifies τ^{KS} under k_i^{sync}. If verification fails, it aborts. The emergency flag Emg is included in the tag τ^{KS}, thereby enabling the sensor to authenticate which mode (emergency or non-emergency) is applied for the current session. The sensor first derives the user-side contribution. If $\mathsf{Emg} = 0$, the sensor computes

$$k_{\mathrm{session}}^{\text{U-S}} = \mathsf{KDF}\Big(k_i^{\text{U-S}} \,\|\, r_{S_i} \,\|\, r_{\mathrm{U}} \,\|\, L = \texttt{"U-S:non-emg key"}\Big).$$

Otherwise, if $\mathsf{Emg} = 1$, the sensor computes

$$k_{\text{session}}^{\text{U-S}} = \mathsf{KDF}\Big(k_i^{\text{U-S}} \,\|\, r_i^{\text{S,emg}} \,\|\, r_{\text{U}} \,\|\, L = \texttt{"U-S:emg key"}\Big).$$

The sensor then derives the key-server contribution:

$$k_{\text{session}}^{\text{KS-S}} = \mathsf{KDF}\Big(k_i^{\text{KS-S}} \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\text{KS}} \,\|\, L = \texttt{"KS-S:kdf"}\Big).$$

It then computes, in $\mathbb{Z}_P$,

$$\begin{aligned} \kappa_P \;&:=\; \big(\mathsf{Encode}(k_{\text{session}}^{\text{U-S}}) + \mathsf{Encode}(k_{\text{session}}^{\text{KS-S}})\big) \bmod P, \\ \kappa \;&:=\; \mathsf{H}_{\mathsf{extract}}(\kappa_P), \end{aligned}$$

and derives the session key

$$k_{\text{session}} = \mathsf{KDF}\Big(\kappa \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\text{Dr}} \,\|\, L = \texttt{"Dr-S:kdf"}\Big).$$

After obtaining the session key, the sensor applies the key-update procedure UPD_{S_i} once (Sect. 4.4).
The sensor also verifies τ^{Dr} under $k^{\text{Dr,mac}}$ derived from k_{session}; If verification succeeds, the sensor is assured that D_{Dr} has derived the same session key. The sensor also derives a key-confirmation MAC key

$$k^{\text{S,mac}} = \mathsf{KDF}\Big(k_{\text{session}} \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\text{Dr}} \,\|\, L = \texttt{"S-Dr:kdf-mac"}\Big),$$

computes the confirmation tag

$$\tau^{\text{S}} = \mathsf{MAC}_{k^{\text{S,mac}}}\Big(\mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\text{Dr}} \,\|\, L = \texttt{"S-Dr:mac"}\Big),$$

and sends $m_{10} = \mathsf{sid} \,\|\, \tau^{\text{S}}$ to D_{Dr}.

– **Final key confirmation at the doctor's device.** Upon receiving m_{10}, the doctor's device derives $k^{\text{S,mac}}$ from k_{session} and verifies τ^{S}. If verification succeeds, D_{Dr} is assured that S_i has derived the same session key.

14 Consistency of Sensor and Key-Server State

The goal of this section is to prove that Chronos preserves a consistent per-sensor state across the sensor and the key servers. For a fixed sensor s_i, the protocol may face message loss, temporary disconnection, key-server crashes and recovery, replayed requests, or several concurrent session attempts initiated by one or more

doctor devices. We show that these events cannot create incompatible per-sensor states. The committed key-server state may become one state update ahead of the sensor, but this desynchronisation is bounded and recoverable: the sensor is never ahead of the committed key-server state, the key servers never consume the same predecessor update counter twice, and a subsequent successful recovery execution restores synchrony.

Relevant protocol remarks. We recall the protocol details relevant to the consistency proof.

• At provisioning, the sensor s_i and the key servers start from the same per-sensor cryptographic state and the same update counter.

• A key-server state update for s_i is driven only by committed records in the committed per-sensor ledger $\mathsf{CLedger}_j(T;i)$. A committed record changes the key-server cryptographic state only if it is transition-valid according to Definition 3.

• The sensor s_i executes UPD_{S_i} only in two cases: either it consumes a valid update token UI_{S_i}, or it reaches the final sensor-side update point in Step 12 after verifying τ^{KS} and deriving the session key. In either case, the sensor executes UPD_{S_i} at most once.

• The key-server leader uses $(k_i^{\mathsf{sync,old}}, k_i^{\mathsf{sync,new}})$ to determine whether s_i is synchronised with the committed key-server state or one state update behind. A sensor-authentication tag τ_1^S verifying under $k_i^{\mathsf{sync,new}}$ allows the session to proceed to ledger submission. A tag verifying under $k_i^{\mathsf{sync,old}}$ causes the leader to issue UI_{S_i} and not to submit a fresh state-update log record for the stale sensor state.

• Sensor-authentication tags, update tokens, leader messages, doctor signatures, and key-confirmation tags are unforgeable except with negligible probability in the security parameter. More precisely, the exceptional event in this proof is bounded by the sum of the SUF-CMA forging probabilities of the MAC instances used for τ_1^S with label "$\mathsf{S\text{-}KS : mac''}$, τ_{upd} with label "$\mathsf{KS\text{-}S : mac\text{-}upd''}$, and τ_{KS} with label "$\mathsf{KS\text{-}S : mac''}$, together with the EUF-CMA forging probabilities of the signatures used to authenticate doctor and leader messages. By Assumption A6, this total probability is negligible.

State counters and desynchronisation functions. For sensor s_i, let $c_i^{\mathsf{sens}}(T)$ denote the number of executions of UPD_{S_i} by the sensor up to time T.

For key server KS_j, define the key-server update counter induced by its committed per-sensor ledger as $c_{i,j}^{\mathsf{KS}}(T) = N_i^{\mathsf{state}}(\mathsf{CLedger}_j(T;i))$. By committed-ledger consistency, the committed per-sensor ledgers $\mathsf{CLedger}_j(T;i)$ are prefix-comparable across key servers. Hence, among the committed per-sensor ledgers defined at time T, there is a unique longest committed per-sensor prefix, denoted by $\mathsf{CLedger}^{\max}(T;i)$. Define $c_i^{\max}(T) = N_i^{\mathsf{state}}(\mathsf{CLedger}^{\max}(T;i))$. This is the key-server update counter induced by the most advanced committed per-sensor ledger prefix known at time T.

The *prefix lag* of KS_j for sensor s_i is $\Lambda_{i,j}(T) = c_i^{\max}(T) - c_{i,j}^{\mathsf{KS}}(T)$. The sensor–key-server desynchronisation in the view of KS_j is $D_{i,j}(T) = c_{i,j}^{\mathsf{KS}}(T) - c_i^{\mathsf{sens}}(T)$. The *committed-ledger desynchronisation* is $D_i^{\max}(T) = c_i^{\max}(T) - c_i^{\mathsf{sens}}(T)$, namely

the desynchronisation between the sensor and the most advanced committed per-sensor ledger prefix known at time T. This value is well-defined because committed-ledger consistency makes all committed per-sensor views prefix-comparable. By definition, $D_{i,j}(T) = D_i^{\max}(T) - \Lambda_{i,j}(T)$.

State-changing events. The proof uses the following three state-changing events. These are the only protocol events that can change $D_i^{\max}$.

• E_{KS}: a committed state-update log record for s_i is transition-valid according to Definition 3 and is applied as a key-server state transition. This increases $c_i^{\max}$ by 1.

• E_{tok}: the sensor s_i consumes a valid update token UI_{S_i} and executes UPD_{S_i} once. This increases c_i^{sens} by 1.

• E_{fin}: the sensor reaches the final sensor-side update point in Step 12 of a valid Chronos session and executes UPD_{S_i} once. This increases c_i^{sens} by 1.

Thus, $E_{\mathsf{KS}} : D_i^{\max} \leftarrow D_i^{\max} + 1$, whereas $E_{\mathsf{tok}}, E_{\mathsf{fin}} : D_i^{\max} \leftarrow D_i^{\max} - 1$. Uncommitted records in $\mathsf{Ledger}_j(T)$, stale committed records in $\mathsf{CLedger}_j(T; i)$, and lost protocol messages that do not trigger one of these events do not change $D_i^{\max}$.

Theorem 1 (Per-sensor consistency and self-synchronisation of Chronos). *Fix a sensor s_i. Assume the replicated ledger satisfies committed-ledger consistency, committed-ledger liveness, and Chronos transition validity as defined in Definitions 1, 2, and 3. Then, except with negligible probability in the security parameter, the following hold for every adversarial network schedule, every sequence of doctor devices, every admissible sequence of key-server leaders, and every pattern of aborts.*

1. ***Key-server prefix-lag convergence.*** *For every key server KS_j and every time T, $\Lambda_{i,j}(T) \geq 0$. Moreover, if $c_i^{\max}(T + \Gamma_{\mathsf{commit}}) = c_i^{\max}(T)$, then every key server KS_j online at time $T + \Gamma_{\mathsf{commit}}$ satisfies $\Lambda_{i,j}(T + \Gamma_{\mathsf{commit}}) = 0$. Hence, after a commit-quiet interval of length Γ_{commit}, all online key servers have the same induced update counter for s_i.*
2. ***Bounded committed-ledger desynchronisation.*** *For every time T, $D_i^{\max}(T) \in \{0, 1\}$. Consequently, if $c_i^{\max}(T + \Gamma_{\mathsf{commit}}) = c_i^{\max}(T)$, then every key server KS_j online at time $T + \Gamma_{\mathsf{commit}}$ satisfies*

$$D_{i,j}(T + \Gamma_{\mathsf{commit}}) = D_i^{\max}(T + \Gamma_{\mathsf{commit}}) \in \{0, 1\}.$$

3. ***Recovery from one-step desynchronisation.*** *If $D_i^{\max}(T) = 1$, then any later successful recovery execution for s_i causes the sensor to consume a valid update token and returns the system to $D_i^{\max} = 0$. If that execution continues to a successful full session, it also ends with $D_i^{\max} = 0$. After the following commit-quiet interval, all online key servers and the sensor are synchronised with respect to the evolved per-sensor state.*

Proof. Let Forge denote the event that an honest party accepts an authenticated protocol message that was not generated by an honest Chronos execution. This includes forged sensor-authentication tags τ_1^S, forged update-token tags τ_{upd},

forged key-server or doctor confirmation tags τ_{KS} and τ_{Dr}, and forged doctor or leader signatures. By the assumed SUF-CMA security of the MAC instances and EUF-CMA security of the signature schemes, $\Pr[\mathsf{Forge}]$ is negligible in the security parameter. We prove the theorem conditioned on $\neg\mathsf{Forge}$, and remove this conditioning at the end.

The proof proceeds in three steps: first, we prove key-server prefix convergence, then use it to prove bounded sensor–key-server desynchronisation, and finally use the one-step case to prove recovery.

1. Key-server prefix lag and convergence. By committed-ledger consistency (Definition 1), for every key server KS_j and time T,

$$\mathsf{CLedger}_j(T;i) \preceq \mathsf{CLedger}^{\max}(T;i).$$

The counter $N_i^{\mathsf{state}}(\cdot)$ is induced deterministically by processing committed records in per-sensor ledger order. Therefore,

$$c_{i,j}^{\mathsf{KS}}(T) = N_i^{\mathsf{state}}(\mathsf{CLedger}_j(T;i)) \leq N_i^{\mathsf{state}}(\mathsf{CLedger}^{\max}(T;i)) = c_i^{\max}(T).$$

Hence,

$$\Lambda_{i,j}(T) = c_i^{\max}(T) - c_{i,j}^{\mathsf{KS}}(T) \geq 0.$$

Now suppose

$$c_i^{\max}(T + \Gamma_{\mathsf{commit}}) = c_i^{\max}(T).$$

Thus no additional transition-valid committed record for s_i advances the longest induced counter during $(T, T + \Gamma_{\mathsf{commit}}]$. By committed-ledger liveness (Definition 2) and the online key-server convention, every key server online at time $T + \Gamma_{\mathsf{commit}}$ has incorporated the committed prefix needed to induce $c_i^{\max}(T)$. Therefore,

$$c_{i,j}^{\mathsf{KS}}(T + \Gamma_{\mathsf{commit}}) = c_i^{\max}(T) = c_i^{\max}(T + \Gamma_{\mathsf{commit}}),$$

and hence

$$\Lambda_{i,j}(T + \Gamma_{\mathsf{commit}}) = 0.$$

This proves Item 1.

2. Bounded committed-ledger desynchronisation. We prove Item 2 by showing that the committed-ledger desynchronisation $D_i^{\max}$ remains in $\{0, 1\}$, and then using Item 1 to lift this bound to the per-key-server values $D_{i,j}$ after ledger convergence.

The quantity $D_i^{\max}$ is determined only by the two counters $c_i^{\max}$ and c_i^{sens}:

$$D_i^{\max}(T) = c_i^{\max}(T) - c_i^{\mathsf{sens}}(T).$$

Thus, to prove that $D_i^{\max}$ remains bounded, it is enough to consider the events that change one of these two counters. The counter $c_i^{\max}$ changes only when a committed record for s_i is transition-valid and is applied as a key-server state transition; we denote this event by E_{KS}. The counter c_i^{sens} changes only when s_i consumes a valid update token or reaches the Step 12 final sensor-side update

point of a valid session; we denote these events by E_{tok} and E_{fin}, respectively. All other protocol actions, including message loss, stale committed records, and uncommitted ledger records, leave $D_i^{\max}$ unchanged.

Table 7 summarises the resulting transition system. The table shows that normal execution, aborts, recovery, and restart keep $D_i^{\max}$ inside $\{0, 1\}$, provided that two bad transitions cannot occur: a second effective key-server update while $D_i^{\max} = 1$, or a sensor-side update while $D_i^{\max} = 0$. We now rule out these two cases.

First, we show that $D_i^{\max}$ cannot increase from 1 to 2. Suppose $D_i^{\max} = 1$ at time T. Then

$$c_i^{\max}(T) = c_i^{\mathsf{sens}}(T) + 1.$$

Consider any later request for s_i before the sensor performs the matching update. If the coordinating key-server leader is synchronised with the committed ledger for s_i, then its local committed state is induced by $\mathsf{CLedger}^{\max}(T; i)$. The sensor is therefore exactly one state update behind the leader's committed state. Hence an honestly generated sensor-authentication tag τ_1^S verifies under $k_i^{\mathsf{sync,old}}$, not under $k_i^{\mathsf{sync,new}}$. The leader issues UI_{S_i} and does not submit a fresh state-update record for this stale sensor state. Thus this request cannot create a new effective E_{KS}.

If the coordinating key-server leader is not yet synchronised with $\mathsf{CLedger}^{\max}(T; i)$, then it has prefix lag:

$$c_{i,\ell}^{\mathsf{KS}}(T) < c_i^{\max}(T).$$

Any state-update record it can submit is rooted in a predecessor counter already below the longest committed counter $c_i^{\max}(T)$. Such a record is stale when processed against the longest committed per-sensor prefix and has no effect on key-server cryptographic state by Definition 3. Hence, a prefix-lagged leader may delay recovery or submit a stale proposal, but it cannot advance $c_i^{\max}$ beyond the already committed state.

This covers both single and concurrent requests. In a concurrent execution, each coordinating leader is either synchronised with $\mathsf{CLedger}^{\max}$ or prefix-lagged. A synchronised leader issues an update token, while a prefix-lagged leader can only produce a stale proposal rooted below $c_i^{\max}$. Moreover, if several proposals are rooted at the same predecessor counter h, Chronos transition validity applies at most one of them: once one committed record with $N_i^{\mathsf{req}} = h$ is applied, the induced counter becomes $h + 1$, and all later committed records with the same predecessor counter are stale. Thus concurrent or replayed doctor requests cannot create two effective E_{KS} events from the same sensor state.

Therefore, under $\neg\mathsf{Forge}$, $D_i^{\max}$ cannot increase from 1 to 2. For such an increase to occur, the leader would have to accept a sensor-authentication tag τ_1^S under $k_i^{\mathsf{sync,new}}$ for a stale sensor state. In the protocol, such a tag is computed over the current session values, including $\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{D_r} \,\|\, r_{S_i} \,\|\, r_{D_r} \,\|\, \text{"S-KS : mac"}$. Since the honest sensor does not compute this tag under $k_i^{\mathsf{sync,new}}$ while it is one update behind, accepting such a tag is contained in Forge.

We also show that no valid execution can make the sensor move ahead of the committed key-server state. Suppose $D_i^{\max} = 0$. Then the sensor is synchronised

with the most advanced committed per-sensor ledger prefix. A sensor-side update from this state without a corresponding committed key-server update would decrease $D_i^{\max}$ from 0 to -1. Such an event is not enabled under $\neg\mathsf{Forge}$. An unauthorised E_{tok} requires the sensor to accept a forged update token, namely a forged τ_{upd} under the update-token MAC key with label $\mathsf{KS\text{-}S : mac\text{-}upd}$. An unauthorised E_{fin} requires the sensor to accept a forged Step 12 key-server path, namely a forged τ_{KS} under the current synchronisation key with label $\mathsf{KS\text{-}S : mac}$. Both cases are contained in Forge.

At provisioning,

$$c_i^{\max}(0) = c_i^{\mathsf{sens}}(0) = 0,$$

so

$$D_i^{\max}(0) = 0.$$

From the arguments above, if $D_i^{\max} = 0$, then the only enabled non-forgery state-changing event that can increase it is E_{KS}, which increases $D_i^{\max}$ to 1; all other protocol actions preserve $D_i^{\max}$. If $D_i^{\max} = 1$, then no non-forgery execution can create another effective E_{KS} before the sensor catches up. The only enabled non-forgery state-changing events are E_{tok} and E_{fin}, each of which decreases $D_i^{\max}$ to 0; all other protocol actions preserve $D_i^{\max}$. Therefore, by induction over state-changing events,

$$D_i^{\max}(T) \in \{0, 1\}$$

for every time T.

It remains to derive the per-key-server statement in Item 2. From the definitions,

$$D_{i,j}(T) = c_{i,j}^{\mathsf{KS}}(T) - c_i^{\mathsf{sens}}(T) = D_i^{\max}(T) - \Lambda_{i,j}(T).$$

If

$$c_i^{\max}(T + \Gamma_{\mathsf{commit}}) = c_i^{\max}(T),$$

then Item 1 gives

$$\Lambda_{i,j}(T + \Gamma_{\mathsf{commit}}) = 0$$

for every key server KS_j online at time $T + \Gamma_{\mathsf{commit}}$. Hence

$$D_{i,j}(T + \Gamma_{\mathsf{commit}}) = D_i^{\max}(T + \Gamma_{\mathsf{commit}}) \in \{0, 1\}.$$

If the interval is not commit-quiet, then at least one newer transition-valid committed record for s_i is applied during the interval. In that case, some online key servers may temporarily have positive prefix lag for the newer committed prefix, but this is only ledger-prefix lag:

$$D_{i,j}(T) = D_i^{\max}(T) - \Lambda_{i,j}(T).$$

By Item 1, the lag disappears after a subsequent commit-quiet interval of length Γ_{commit}. This proves Item 2.

3. Recovery from the one-step state. Assume $D_i^{\max}(T) = 1$. Then the committed key-server state is one update ahead of the sensor. Consider a later

successful recovery execution at time $T' \geq T$ coordinated by a leader key server whose local committed per-sensor prefix has caught up, i.e., $\Lambda_{i,\text{leader}}(T') = 0$. In this execution, the sensor is exactly one update behind the leader's local key-server state, and therefore the sensor-authentication tag τ_1^S is accepted under $k_i^{\mathsf{sync,old}}$. By the protocol, the leader issues a valid update token

$$\mathsf{UI}_{S_i} = (\tau_{\mathsf{upd}}, r_{\mathsf{upd}}),$$

where τ_{upd} is an authentication tag computed under $k_i^{\mathsf{sync,new}}$ and bound to the recovery session values. Since the sensor is one update behind, its verification key $\mathsf{Upd}(k_i^{\mathsf{sync}})$ equals the committed key-server synchronisation key $k_i^{\mathsf{sync,new}}$. Therefore, when s_i receives and verifies UI_{S_i}, it executes UPD_{S_i} once. This is event E_{tok}, so c_i^{sens} increases by 1, while $c_i^{\max}$ does not change. Consequently,

$$D_i^{\max} : 1 \longrightarrow 0.$$

If the update token is lost, no state-changing event occurs and the system remains at $D_i^{\max} = 1$. The same recovery condition is presented again in the next successful attempt; repeated token loss delays recovery but cannot increase desynchronisation.

If the recovery execution continues to a full successful session, the fresh session starts from $D_i^{\max} = 0$. It may then produce one transition-valid committed key-server update E_{KS}, moving $D_i^{\max}$ from 0 to 1, followed by the Step 12 sensor update E_{fin}, moving $D_i^{\max}$ from 1 back to 0. Thus a successful recovery execution restores the committed-ledger desynchronisation to 0. By Item 1, after the following commit-quiet interval, all online key servers have zero prefix lag, and therefore their individual desynchronisation values also equal 0. This proves Item 3.

We have proved all three claims conditioned on $\neg\mathsf{Forge}$. Since $\Pr[\mathsf{Forge}]$ is negligible in the security parameter, the theorem follows except with negligible probability.

15 Replicated Database Implementation

We implement DB_{rep} as a crash-fault-tolerant (CFT) replicated, append-only log using a Raft-style leader-based replication protocol [18]. Our implementation uses the same key server as both the Chronos session leader and the Raft leader. For each log entry, the leader first appends the entry to stable storage and then disseminates AppendEntries RPCs to the $n-1$ followers over mutually authenticated channels. A follower persists the entry and returns an acknowledgement. The leader marks an entry as *committed* once it has received acknowledgements from a quorum of size t, where $t \geq \lfloor n/2 \rfloor + 1$ (i.e., the CFT-majority bound). Under our synchrony assumption with bounded delay Δ_{delay} among online key servers, the leader reaches a commit decision within one round trip, i.e., within at most $2\Delta_{\text{delay}}$ after transmission of AppendEntries. The leader then propagates the updated commit index to followers via the next heartbeat, so that all online replicas apply committed entries in the same order; thus, followers learn the

commit within an additional Δ_{delay}. This majority-quorum commit rule ensures that all online servers converge to a consistent per-sensor state and prevents desynchronization between the key-server infrastructure under crash faults.

Fig. 2: Compact specification of the session-key establishment protocol. Each bullet corresponds to one protocol step.

❶ Initiation.

D_{Dr}: $r_{\mathrm{Dr}} \xleftarrow{\$} \{0,1\}^{\lambda}$;

if starting a fresh session then sample $\mathsf{sid} \xleftarrow{\$} \{0,1\}^{\lambda_{\mathsf{sid}}}$; set $\mathsf{UI}_{S_i} := \emptyset$.
else if resynchronising after m_{upd} (at step ❺) then set sid as the session identifier of the resync-required session, set $\mathsf{UI}_{S_i} = (\tau_{\mathsf{upd}}, r_{\mathrm{upd}})$.
Send $m_1 := \mathsf{sid} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, \mathsf{UI}_{S_i} \,\|\, r_{\mathrm{Dr}}$ to S_i.

---

❷ Sensor-side state synchronisation via update token.

S_i: upon m_1,
if $\mathsf{UI}_{S_i} = \emptyset$ then continue to step ❸ .
else parse $(\tau_{\mathsf{upd}}, r_{\mathrm{upd}})$; retrieve the pending tuple $(\mathsf{sid}^\star, \mathsf{ID}^\star_{\mathrm{Dr}}, r^\star_{\mathrm{Dr}}, r^\star_{S_i})$ stored after the previous m_2; abort if no such tuple exists, if $\mathsf{sid} \neq \mathsf{sid}^\star$, or if $\mathsf{ID}_{\mathrm{Dr}} \neq \mathsf{ID}^\star_{\mathrm{Dr}}$; abort unless τ_{upd} verifies under key $\mathsf{Upd}(k_i^{\mathrm{sync}})$ on message $\mathsf{sid}^\star \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}^\star_{\mathrm{Dr}} \,\|\, r^\star_{S_i} \,\|\, r^\star_{\mathrm{Dr}} \,\|\, r_{\mathrm{upd}} \,\|\, L =$ `"KS-S:mac-upd"`; apply sensor update UPD_{S_i} once (Sect. 4.4); continue to step ❸ using the fresh r_{Dr} in m_1.

---

❸ Sensor response.

S_i: Sample $r_{S_i} \xleftarrow{\$} \{0,1\}^{\lambda}$;
derive $k_i^{\text{U-S,mac}} := \mathsf{KDF}\big(k_i^{\text{U-S}} \,\|\, \mathsf{sid} \,\|\, L =$ `"S-U:kdf-mac"`$\big)$;
compute $\tau_1^S := \mathsf{MAC}_{k_i^{\mathrm{sync}}}\big(\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \,\|\, L =$ `"S-KS:mac"`$\big)$ and $\tau_2^S := \mathsf{MAC}_{k_i^{\text{U-S,mac}}}\big(\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \,\|\, L =$ `"S-U:mac"`$\big)$;
send $m_2 := \mathsf{sid} \,\|\, \tau_1^S \,\|\, \tau_2^S \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i}$ to D_{Dr};
store $(\mathsf{sid}, \mathsf{ID}_{\mathrm{Dr}}, r_{\mathrm{Dr}}, r_{S_i})$ until completion/-timeout/abort/resynchronisation.

---

❹ Doctor session-establishment request to the key-server service.

D_{Dr}: generate an ephemeral key pair $(\mathsf{ePK}_{\mathrm{Dr}}, \mathsf{eSK}_{\mathrm{Dr}})$;
sample timestamp t_{Dr}; set $\mathsf{Emg} \in \{0,1\}$;
set $\mathsf{mac_token}_{\mathrm{Dr}} := \tau_1^S \,\|\, \tau_2^S \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}}$;
compute $h_{\mathrm{Dr}} := H(\mathsf{mac_token}_{\mathrm{Dr}})$;
set $\mathsf{core_req}_{\mathrm{Dr}} := \mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, \mathsf{ePK}_{\mathrm{Dr}} \,\|\, t_{\mathrm{Dr}} \,\|\, \mathsf{Emg} \,\|\, h_{\mathrm{Dr}}$;
sign $\sigma_{\mathrm{Dr}} := \sigma_{\mathsf{SK}_{\mathrm{Dr}}}(\mathsf{core_req}_{\mathrm{Dr}})$;
send $m_3 := \mathsf{mac_token}_{\mathrm{Dr}} \,\|\, \mathsf{core_req}_{\mathrm{Dr}} \,\|\, \sigma_{\mathrm{Dr}}$ to the key-server service over an encrypted channel.

---

❺ Leader verification and synchronisation.

Key-server service selects an online key server as the leader to coordinate this session.
$\mathsf{KS}_{\mathrm{leader}}$: retrieve doctor certificate; verify σ_{Dr} on $\mathsf{core_req}_{\mathrm{Dr}}$; recompute $H(\mathsf{mac_token}_{\mathrm{Dr}})$ and compare to embedded h_{Dr}; abort if any check fails.
Verify τ_1^S under $k_i^{\mathrm{sync,old}}$:
if valid then sensor is one state behind: sample $r_{\mathrm{upd}} \xleftarrow{\$} \{0,1\}^{\lambda}$;
compute $\tau_{\mathsf{upd}} := \mathsf{MAC}_{k_i^{\mathrm{sync,new}}}\big(\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \,\|\, r_{\mathrm{upd}} \,\|\, L =$ `"KS-S:mac-upd"`$\big)$; set $\mathsf{UI}_{S_i} := (\tau_{\mathsf{upd}}, r_{\mathrm{upd}})$, reply $m_{\mathrm{upd}} := \mathsf{resync} \,\|\, \mathsf{UI}_{S_i}$ to D_{Dr}. The doctor resynchronises at step ❶ using m_{upd};
else verify τ_1^S under $k_i^{\mathrm{sync,new}}$: proceed if valid, else abort.

---

❻ Broadcast and consent request.

$\mathsf{KS}_{\mathrm{leader}}$: generate an ephemeral key pair

Each online KS_j: upon m_4, first verify the integrity and freshness of the doctor's request (via σ_{Dr} and timestamp t_{Dr}); check whether the local committed counter for S_i equals the received value N_i^{req}. Only if this check succeeds, compute
$m'_{7,j} := \mathsf{sid} \,\|\, N_i^{\mathsf{req}} \,\|\, \mathsf{Enc}_{\mathsf{ePK}_{\mathrm{leader}}}(k_{i,j}^{\text{KS-S}})$.
if $\mathsf{Emg} = 0$: set $m_{7,j} := m'_{7,j}$.
if $\mathsf{Emg} = 1$: set $m_{7,j} := m'_{7,j} \,\|\, \mathsf{Enc}_{\mathsf{ePK}_{\mathrm{Dr}}}(k_{i,j}^{\mathrm{emg}})$.
Send $m_{7,j}$ to $\mathsf{KS}_{\mathrm{leader}}$ over an authenticated, encrypted channel.

---

❾ Reconstruction, ledger commit, and homomorphic combination.

$\mathsf{KS}_{\mathrm{leader}}$: collect $\geq t$ key-server responses (incl. itself); **if** $\mathsf{Emg} = 0$ also receive m_6.
Decrypt $\mathsf{Enc}_{\mathsf{ePK}_{\mathrm{leader}}}(k_{i,j}^{\text{KS-S}})$ to obtain shares $k_{i,j}^{\text{KS-S}} \in \mathbb{Z}_P$ and
reconstruct $s_i \in \mathbb{Z}_P$ by Lagrange interpolation; set $k_i^{\text{KS-S}} := \mathsf{Decode}(s_i)$.
Sample $r_{\mathrm{KS}} \xleftarrow{\$} \{0,1\}^{\lambda}$;
compute $\tau^{\mathrm{KS}} := \mathsf{MAC}_{k_i^{\mathrm{sync,new}}}\big(\mathsf{sid} \,\|\, \mathsf{ID}_{S_i} \,\|\, \mathsf{ID}_{\mathrm{Dr}} \,\|\, r_{S_i} \,\|\, r_{\mathrm{KS}} \,\|\, \mathsf{Emg} \,\|\, L =$ `"KS-S:mac"`$\big)$, and $k_{\mathrm{session}}^{\text{KS-S}} := \mathsf{KDF}\big(k_i^{\text{KS-S}} \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\mathrm{KS}} \,\|\, L =$ `"KS-S:kdf"`$\big)$.
Let $v := \mathsf{Encode}(k_{\mathrm{session}}^{\text{KS-S}}) \in \mathbb{Z}_P$.
if $\mathsf{Emg} = 0$: use $\mathsf{Enc}_{\mathsf{ePK}_{\mathrm{Dr}}}(u)$ from m_6 (where $u = \mathsf{Encode}(k_{\mathrm{session}}^{\text{U-S}})$) and set $r_{\mathrm{U}} := r_{\mathrm{U}}^{\text{non-emg}}$.
if $\mathsf{Emg} = 1$: reconstruct $\mathsf{Enc}_{\mathsf{ePK}_{\mathrm{Dr}}}(u)$ homomorphically from encrypted emergency shares and set $r_{\mathrm{U}} := r_{\mathrm{U}}^{\mathrm{emg}}$.
Homomorphically compute $k_{\mathrm{session}}^{\text{KS-U-S,enc}} := \mathsf{Enc}_{\mathsf{ePK}_{\mathrm{Dr}}}(u + v)$.
Next compute the next server–sensor key $k_i^{\text{KS-S,new}} := \mathsf{Upd}(k_i^{\text{KS-S}})$ and the per-server update tokens $\Delta_{i,j} := \mathsf{SC}(k_i^{\text{KS-S,new}}, j) - \mathsf{Upd}_{\mathbb{Z}_P}(k_{i,j}^{\text{KS-S}}) \bmod P$.
Erase received shares $\{k_{i,j}^{\text{KS-S}}\}$ and keys $k_i^{\text{KS-S}}, k_i^{\text{KS-S,new}}$;
submit the state-update log $\mathcal{L}_i^{\text{KS-S}} := (\mathsf{sid}, \mathsf{ID}_{S_i}, N_i^{\mathsf{req}}, \{\Delta_{i,j}\}_{j\in[n]})$ to the replicated ledger $\mathsf{DB}_{\mathrm{rep}}$. Upon log commitment, each KS_j performs $\mathsf{UPD}_{\mathrm{KS}}$ to update its stored local key set (Sect. 4.4).

---

❿ Leader response.

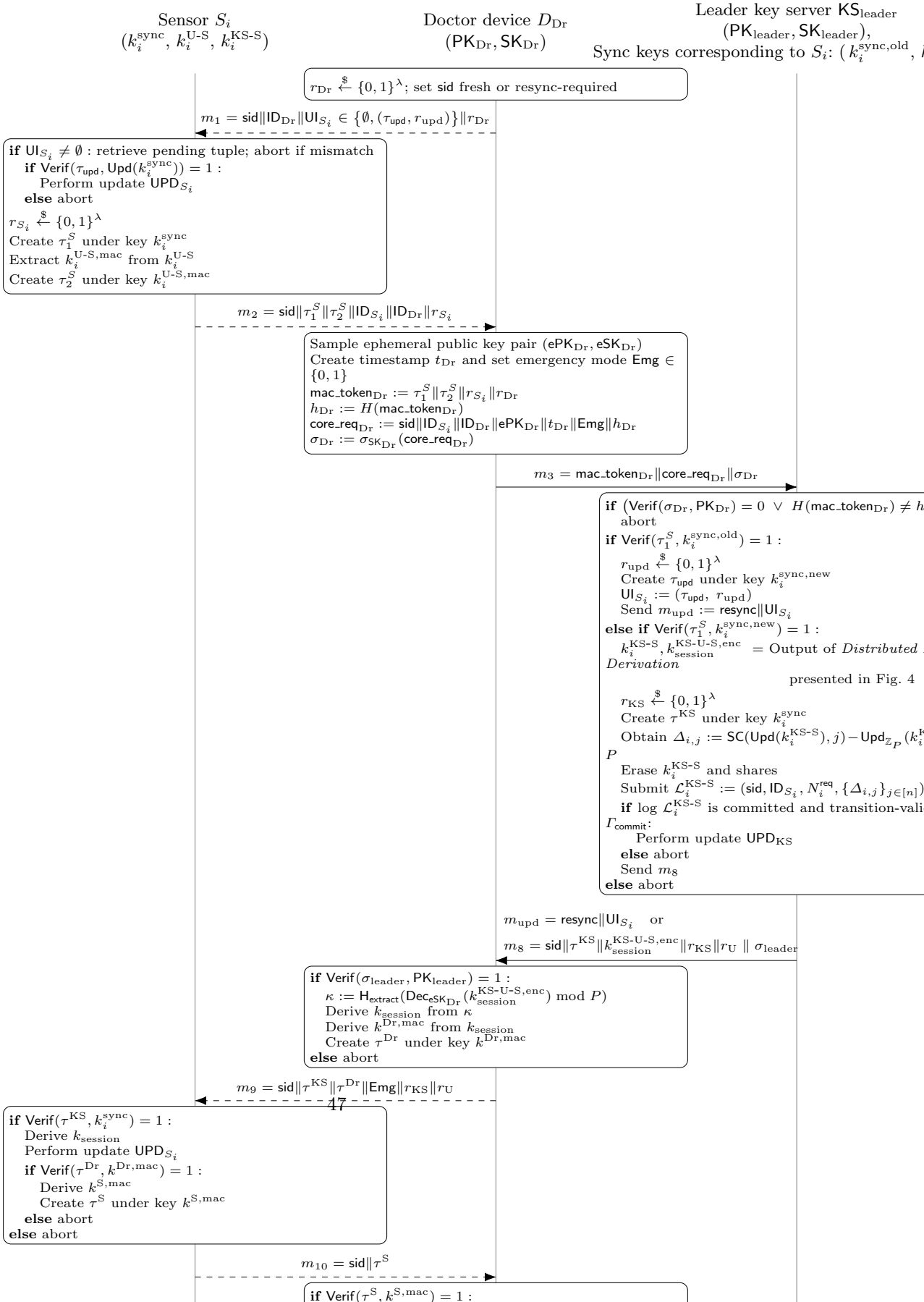
After $\mathcal{L}_i^{\text{KS-S}}$ is committed and the leader applies its update, the leader sends
$m'_8 := \mathsf{sid} \,\|\, \tau^{\mathrm{KS}} \,\|\, k_{\mathrm{session}}^{\text{KS-U-S,enc}} \,\|\, r_{\mathrm{KS}} \,\|\, r_{\mathrm{U}}$ with signature $m_8 := m'_8 \,\|\, \sigma_{\mathsf{SK}_{\mathrm{leader}}}(m'_8)$ to D_{Dr} over an encrypted channel.

⓫ Doctor-side key derivation.

D_{Dr}: verify leader signature on m_8 and
decrypt $\kappa_P := \mathsf{Dec}_{\mathsf{eSK}_{\mathrm{Dr}}}(k_{\mathrm{session}}^{\text{KS-U-S,enc}}) \bmod P = u + v \bmod P$;
set $\kappa := \mathsf{H}_{\mathsf{extract}}(\kappa_P)$; derive $k_{\mathrm{session}} := \mathsf{KDF}(\kappa \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \,\|\, L =$ `"Dr-S:kdf"`$)$.
Derive $k^{\mathrm{Dr,mac}} := \mathsf{KDF}\big(k_{\mathrm{session}} \,\|\, \mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\mathrm{Dr}} \,\|\, L =$ `"Dr-S:kdf-mac"`$\big)$;
compute $\tau^{\mathrm{Dr}} := \mathsf{MAC}_{k^{\mathrm{Dr,mac}}}\big(\mathsf{sid} \,\|\, r_{S_i} \,\|\, r_{\mathrm{KS}} \,\|\, r_{\mathrm{U}} \,\|\, r_{\mathrm{Dr}} \,\|\, L =$ `"Dr-S:mac"`$\big)$;
send $m_9 := \mathsf{sid} \,\|\, \tau^{\mathrm{KS}} \,\|\, \tau^{\mathrm{Dr}} \,\|\, \mathsf{Emg} \,\|\, r_{\mathrm{KS}} \,\|\, r_{\mathrm{U}}$ to S_i.

⓬ Sensor-side key derivation and confirmation.

S_i: verify τ^{KS} under k_i^{sync}; doctor's legitimacy


Sensor S_i
$(k_i^{\text{sync}}, k_i^{\text{U-S}}, k_i^{\text{KS-S}})$
Doctor device D_{Dr}
$(\mathsf{PK}_{\text{Dr}}, \mathsf{SK}_{\text{Dr}})$
Leader key server $\mathsf{KS}_{\text{leader}}$
$(\mathsf{PK}_{\text{leader}}, \mathsf{SK}_{\text{leader}})$,
Sync keys corresponding to S_i: ($k_i^{\text{sync,old}}$, $k_i^{\text{sync,new}}$)
$r_{\text{Dr}} \xleftarrow{\$} \{0,1\}^\lambda$; set sid fresh or resync-required
$m_1 = \mathsf{sid} \| \mathsf{ID}_{\text{Dr}} \| \mathsf{UI}_{S_i} \in \{\emptyset, (\tau_{\mathsf{upd}}, r_{\text{upd}})\} \| r_{\text{Dr}}$
if $\mathsf{UI}_{S_i} \neq \emptyset$: retrieve pending tuple; abort if mismatch
if $\mathsf{Verif}(\tau_{\mathsf{upd}}, \mathsf{Upd}(k_i^{\text{sync}})) = 1$:
Perform update UPD_{S_i}
else abort
$r_{S_i} \xleftarrow{\$} \{0,1\}^\lambda$
Create τ_1^S under key k_i^{sync}
Extract $k_i^{\text{U-S,mac}}$ from $k_i^{\text{U-S}}$
Create τ_2^S under key $k_i^{\text{U-S,mac}}$
$m_2 = \mathsf{sid} \| \tau_1^S \| \tau_2^S \| \mathsf{ID}_{S_i} \| \mathsf{ID}_{\text{Dr}} \| r_{S_i}$
Sample ephemeral public key pair $(\mathsf{ePK}_{\text{Dr}}, \mathsf{eSK}_{\text{Dr}})$
Create timestamp t_{Dr} and set emergency mode $\mathsf{Emg} \in \{0,1\}$
$\mathsf{mac_token}_{\text{Dr}} := \tau_1^S \| \tau_2^S \| r_{S_i} \| r_{\text{Dr}}$
$h_{\text{Dr}} := H(\mathsf{mac_token}_{\text{Dr}})$
$\mathsf{core_req}_{\text{Dr}} := \mathsf{sid} \| \mathsf{ID}_{S_i} \| \mathsf{ID}_{\text{Dr}} \| \mathsf{ePK}_{\text{Dr}} \| t_{\text{Dr}} \| \mathsf{Emg} \| h_{\text{Dr}}$
$\sigma_{\text{Dr}} := \sigma_{\mathsf{SK}_{\text{Dr}}}(\mathsf{core_req}_{\text{Dr}})$
$m_3 = \mathsf{mac_token}_{\text{Dr}} \| \mathsf{core_req}_{\text{Dr}} \| \sigma_{\text{Dr}}$
if $(\mathsf{Verif}(\sigma_{\text{Dr}}, \mathsf{PK}_{\text{Dr}}) = 0 \ \vee \ H(\mathsf{mac_token}_{\text{Dr}}) \neq h$
abort
if $\mathsf{Verif}(\tau_1^S, k_i^{\text{sync,old}}) = 1$:
$r_{\text{upd}} \xleftarrow{\$} \{0,1\}^\lambda$
Create τ_{upd} under key $k_i^{\text{sync,new}}$
$\mathsf{UI}_{S_i} := (\tau_{\mathsf{upd}},\ r_{\text{upd}})$
Send $m_{\text{upd}} := \mathsf{resync} \| \mathsf{UI}_{S_i}$
else if $\mathsf{Verif}(\tau_1^S, k_i^{\text{sync,new}}) = 1$:
$k_i^{\text{KS-S}}, k_{\text{session}}^{\text{KS-U-S,enc}}$ = Output of *Distributed*
Derivation
presented in Fig. 4
$r_{\text{KS}} \xleftarrow{\$} \{0,1\}^\lambda$
Create τ^{KS} under key k_i^{sync}
Obtain $\Delta_{i,j} := \mathsf{SC}(\mathsf{Upd}(k_i^{\text{KS-S}}), j) - \mathsf{Upd}_{\mathbb{Z}_P}(k_i$
P
Erase $k_i^{\text{KS-S}}$ and shares
Submit $\mathcal{L}_i^{\text{KS-S}} := (\mathsf{sid}, \mathsf{ID}_{S_i}, N_i^{\mathsf{req}}, \{\Delta_{i,j}\}_{j \in [n]})$
if log $\mathcal{L}_i^{\text{KS-S}}$ is committed and transition-vali
Γ_{commit}:
Perform update UPD_{KS}
else abort
Send m_8
else abort
$m_{\text{upd}} = \mathsf{resync} \| \mathsf{UI}_{S_i}$ or
$m_8 = \mathsf{sid} \| \tau^{\text{KS}} \| k_{\text{session}}^{\text{KS-U-S,enc}} \| r_{\text{KS}} \| r_{\text{U}} \ \| \ \sigma_{\text{leader}}$
if $\mathsf{Verif}(\sigma_{\text{leader}}, \mathsf{PK}_{\text{leader}}) = 1$:
$\kappa := \mathsf{H}_{\mathsf{extract}}(\mathsf{Dec}_{\mathsf{eSK}_{\text{Dr}}}(k_{\text{session}}^{\text{KS-U-S,enc}}) \bmod P)$
Derive k_{session} from κ
Derive $k^{\text{Dr,mac}}$ from k_{session}
Create τ^{Dr} under key $k^{\text{Dr,mac}}$
else abort
$m_9 = \mathsf{sid} \| \tau^{\text{KS}} \| \tau^{\text{Dr}} \| \mathsf{Emg} \| r_{\text{KS}} \| r_{\text{U}}$
if $\mathsf{Verif}(\tau^{\text{KS}}, k_i^{\text{sync}}) = 1$:
Derive k_{session}
Perform update UPD_{S_i}
if $\mathsf{Verif}(\tau^{\text{Dr}}, k^{\text{Dr,mac}}) = 1$:
Derive $k^{\text{S,mac}}$
Create τ^{S} under key $k^{\text{S,mac}}$
else abort
else abort
$m_{10} = \mathsf{sid} \| \tau^{\text{S}}$
if $\mathsf{Verif}(\tau^{\text{S}}, k^{\text{S,mac}}) = 1$:

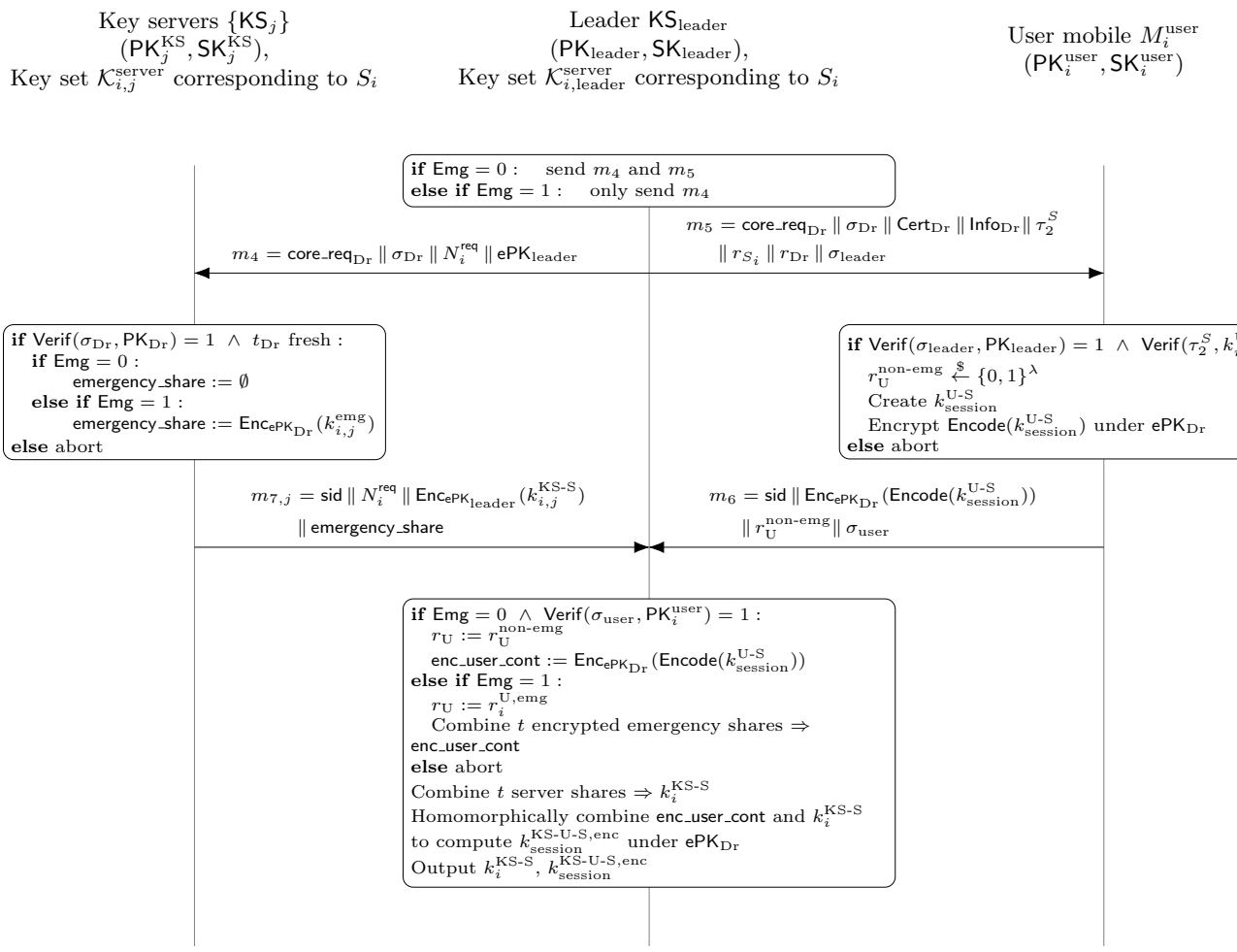


Fig. 4: Distributed Key Derivation. Arrows denote authenticated, encrypted channels.

Table 7: Closed transition system for the committed-ledger desynchronisation $D_i^{\max}$. Each state is a pair $(\mathsf{P}, D_i^{\max})$, where P is the relevant protocol phase. Specifically, P_0 denotes the synchronised pre-commit phase, P_1 the post-commit/pre-Step 12 phase, P_2 the token-recovery phase, and P_3 the post-Step 12 completion phase. Every target state also appears as a source state, so the table describes normal execution, abort, recovery, and restart.

Current state	**Event / attempted transition**	**State-counter effect**	**Next state**
$(\mathsf{P}_0, 0)$	Loss before ledger commitment, e.g., m_1, m_2, m_3, m_5, m_6, or m_4/m_7 before quorum	No E_{KS}, E_{tok}, or E_{fin} occurs; the attempt restarts from synchrony.	$(\mathsf{P}_0, 0)$
$(\mathsf{P}_0, 0)$	E_{KS}: one transition-valid committed key-server update	The key-server counter advances once, while the sensor counter is unchanged.	$(\mathsf{P}_1, 1)$
$(\mathsf{P}_0, 0)$	Concurrent or replayed proposals for the same predecessor counter h	By Chronos transition validity, at most one proposal triggers E_{KS}; later records with $N_i^{\mathsf{req}} = h$ are stale.	$(\mathsf{P}_1, 1)$
$(\mathsf{P}_0, 0)$	Attempted E_{tok} or E_{fin}	Not enabled from synchrony under $\neg\mathsf{Forge}$; accepting such an update requires a forged update token or a forged τ^{KS}.	$(\mathsf{P}_0, 0)$
$(\mathsf{P}_1, 1)$	Loss after commitment but before Step 12, e.g., m_8 or m_9	The key-server update has occurred, but the matching sensor update has not; the next successful attempt enters recovery.	$(\mathsf{P}_2, 1)$
$(\mathsf{P}_1, 1)$	E_{fin}: Step 12 final sensor update	The sensor executes the matching UPD_{S_i} once and catches up.	$(\mathsf{P}_3, 0)$
$(\mathsf{P}_1, 1)$	Attempted fresh E_{KS} from the stale sensor state	Not enabled under $\neg\mathsf{Forge}$: an up-to-date leader issues UI_{S_i}, while a prefix-lagged leader can only produce a stale proposal.	$(\mathsf{P}_2, 1)$
$(\mathsf{P}_2, 1)$	Token loss, e.g., m_{upd} or m_1 carrying UI_{S_i}	No E_{tok} occurs; recovery is delayed, but desynchronisation does not grow.	$(\mathsf{P}_2, 1)$
$(\mathsf{P}_2, 1)$	E_{tok}: sensor consumes valid UI_{S_i}	The sensor executes UPD_{S_i} once and catches up.	$(\mathsf{P}_0, 0)$
$(\mathsf{P}_2, 1)$	Attempted E_{KS} before token recovery	Not enabled under $\neg\mathsf{Forge}$: the sensor is stale, so the leader must issue UI_{S_i}, not submit a fresh update.	$(\mathsf{P}_2, 1)$
$(\mathsf{P}_3, 0)$	Loss after Step 12, e.g., invalid τ^{Dr} or loss of m_{10}	The matching sensor update has already occurred; only confirmation or notification is lost.	$(\mathsf{P}_3, 0)$
$(\mathsf{P}_3, 0)$	New session begins	The protocol returns to the synchronised pre-commit phase.	$(\mathsf{P}_0, 0)$