

Proof-carrying payloads: End-to-end authorization for Pub/Sub messaging

Letterio Galletta 

IMT School for Advanced Studies Lucca, Lucca, Italy

ARTICLE INFO

Keywords:

Protocol verification
Attribute-based encryption
Zero-knowledge proofs

ABSTRACT

Pub/sub messaging is one of the most widely used communication mechanisms in modern distributed systems. It relies on a centralized broker that routes messages among clients and also acts as a policy enforcement point, storing the access-control policy and enforcing it by deciding who may publish to a topic and who may subscribe. An untrusted broker completely subverts the security of such systems, compromising both the confidentiality and authenticity of the data and the correctness of authorization. To address this, we introduce PCP (Proof-Carrying Payloads), an end-to-end authorization overlay that moves both data protection and authorization from the broker to the clients. PCP is broker-agnostic and deploys as a wrapper around an existing pub/sub system, without modifying the broker. In PCP, each message is a cryptographic envelope that carries a succinct zero-knowledge proof establishing that its publisher holds a valid write credential; a subscriber verifies this proof before attempting to decrypt the envelope, while the encryption itself enforces read authorization. PCP further introduces a two-tier authorization revocation mechanism: one tier revokes publishers, and the other revokes subscribers. We present the design of PCP together with a formal analysis of its security. On the design side, we define the system entities and their trust relationships, the enrollment protocols through which publishers and subscribers obtain their credentials, the wrapping procedures that turn an ordinary publish or receive operation into a proof-carrying, encrypted exchange, and the protocols that implement two-tier revocation. On the analysis side, we first state the threat model and the security goals, and then prove that PCP meets them. Goals expressible at the protocol level are verified symbolically in the Tamarin prover, while those that rely on the computational hardness of the underlying cryptographic primitives are established by hand, through standard game-based reductions.

1. Introduction

Pub/sub messaging is one of the most widely used communication mechanisms in modern distributed systems, both in cloud-native microservice architectures and in IoT systems. This mechanism enables communication that is both referentially and temporally decoupled. More specifically, in the pub/sub communication model, the entities sending messages, namely the *publishers*, and those receiving them, namely the *subscribers*, interact indirectly through the infrastructure provided by a third component, the *broker*, which is solely responsible for message delivery. The broker provides clients with virtual communication channels, called *topics*, on which they publish and receive messages.

Authorization in these systems is typically *centralized*: an operator declares, in an attribute-based access-control policy [1] (ABAC), which clients may publish to a topic and which clients may subscribe to a topic. The broker stores, manages, and enforces this policy

E-mail address: letterio.galletta@imtlucca.it

<https://doi.org/10.1016/j.jlamp.2026.101173>

Received 21 June 2026; Received in revised form 9 September 2026; Accepted 14 September 2026

Available online 15 September 2026

2352-2208/© 2026 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

as the single *policy enforcement point*. This arrangement assumes that the broker is trusted. However, this assumption is too strong and is misaligned with the *zero-trust* security model [2], which advocates that no component should be granted implicit trust merely because of its position or role in a network infrastructure. Indeed, a compromised broker undermines pub/sub security along two independent axes. First, because the broker sees every message it routes, the *confidentiality and authenticity* of the exchanged data are at risk. Second, the *correctness of authorization decisions and policy enforcement* can no longer be trusted: an untrusted broker may deliver messages to unauthorized subscribers, forward messages from unauthorized publishers, or silently disregard the policy altogether. In addition, distrust is not confined to the broker: publishers and subscribers cannot trust each other either. A subscriber cannot assume an arriving message is authorized merely because it was delivered, and a publisher must hide its linkable identity from subscribers, who may try to track its publication activity, just as it must from the broker. Securing this scenario, therefore, requires moving both data protection and authorization enforcement off the broker, while still handling mutual distrust among clients.

Existing proposals have targeted specific protocols, such as MQTT [3] for the IoT, and have addressed either the confidentiality or the integrity of messages [4,5], while providing no mechanism for credential and authorization management. To fill this gap, this paper proposes PCP (Proof-Carrying Payloads), an authorization mechanism that shifts policy enforcement from the broker to clients and supports client enrollment, end-to-end authorization, and credential revocation. More precisely, PCP is an end-to-end authorization overlay in which each message is encrypted in a protected envelope that carries its own authorization proof. PCP is broker-agnostic and designed to be deployed as a wrapper around an existing pub/sub messaging system, such as MQTT or NATS [6]: it reuses already-deployed infrastructure without any change to the broker or its communication protocol. This is possible because authorization in PCP is enforced on the client side, through cryptographic primitives, rather than by the messaging infrastructure. Each protected envelope embeds a succinct proof that the publisher holds a valid write credential for the target topic. Upon receiving a message, a subscriber first verifies the proof to determine whether the message was authorized, and only then attempts to decrypt it; decryption succeeds only if the subscriber is authorized to read the envelope. In this way, there is no centralized policy enforcement point: write authorization travels with the message as a proof checked by subscribers, while read authorization is enforced by encryption. This read-side authorization is itself confined behind an abstract, *pluggable* interface (ReadAuth), which leaves PCP agnostic to the underlying authorization backend. The proof embedded in each envelope is public, so any node in the network can verify it. This public verifiability raises another central question in our setting: how can a publisher securely prove to an untrusted verifier that it is authorized to write on a given topic? The answer must reconcile two requirements that pull in opposite directions. On the one hand, the proof must be unforgeable: neither the broker nor a colluding subscriber, having observed any number of authorized envelopes, may reuse their proofs to forge a message that appears authorized. On the other hand, the proof must preserve the publisher's privacy: beyond the bare fact that a message comes from an authorized publisher, it must reveal nothing that links the message to a specific identity, so that no unauthorized node can track a publisher's activity over time. A verifier that simply checks each publisher against a fixed, publisher-specific key would meet the first requirement but defeat the second, since the key would itself identify the writer. Zero-knowledge proofs can reconcile the two requirements: a publisher proves that its credential is valid and has not been revoked without revealing which credential it holds. Authorization is thus verifiable while the publisher's identity stays hidden. PCP thereby provides *bilateral authorization privacy*, shielding both ends of a message from the untrusted broker and from colluding subscribers. On the reader side, a subscriber verifies the proof before attempting to decrypt, so a rejected message is indistinguishable from one it simply cannot decrypt for lack of the key, and neither the broker nor a colluding subscriber learns whether the publisher was authorized. On the writer's side, the envelope never carries the publisher's long-term credentials but only a temporary pseudonym, so a publisher's messages are linkable for a limited period (an epoch).

Beyond authorizing publication and reading of messages, PCP also manages the revocation of authorizations, for which it provides two distinct mechanisms. The first revokes *publishers* and takes effect immediately: once a publisher's write authorization for a topic is withdrawn, every envelope it subsequently produces on that topic is rejected by every honest subscriber, with no further coordination required. The second revokes *subscribers* and is more involved. It requires that the protocol advances through discrete time steps *epochs*: a subscriber's read authorization is withdrawn only at an epoch boundary, after which the revoked subscriber can no longer read new messages published on the topic.

In this paper, we present the design of PCP together with a formal analysis of its security. On the design side, we define the system entities and their trust relationships; the enrollment protocols through which publishers and subscribers obtain their credentials; the wrapping procedures that turn an ordinary publish or receive into a proof-carrying, encrypted exchange; and the protocols that implement the two-tier revocation described above. On the analysis side, we first make the adversary explicit, stating a threat model and the security goals PCP must meet under it. We then prove that PCP meets these goals using two complementary techniques, chosen according to the nature of each goal. Goals expressible at the protocol level are verified in the *symbolic* model: we encode PCP's main protocols and message flows in the Tamarin prover [7] and obtain machine-checked proofs that the design satisfies them. Goals that instead rely on the *computational* hardness of the underlying cryptographic primitives are established by hand, through standard game-based reductions in the computational model [8].

In summary, we make the following contributions:

1. *An end-to-end authorization mechanism under an untrusted broker.* We design an end-to-end authorization mechanism, in the spirit of ABAC, which decides whether a publisher is entitled to publish on a topic and whether a subscriber is entitled to read a message, all under an untrusted broker. The mechanism is broker-agnostic and can be deployed as a wrapper around an existing pub/sub system, without modifying the broker or its protocol.

2. *Enrollment and message-wrapping protocols.* We define the protocols through which clients enroll and obtain their credentials, together with the publishing and receiving wrappers that turn an ordinary publish or receive into a proof-carrying, encrypted exchange.
3. *Two-tier authorization revocation.* We define a two-tier mechanism for revoking client authorizations. One tier revokes publishers and takes effect immediately, as it only invalidates a localized piece of state; the other revokes subscribers at an epoch boundary, as it replaces key material already delivered and involves every client enrolled the topic scope.
4. *Threat model and formal security analysis.* We state a threat model and the security goals PCP must meet, and we formally analyze the design, establishing that it achieves these goals in both a symbolic model (mechanized in Tamarin) and a computational model (through game-based reductions).

The rest of the paper is organized as follows. [Section 2](#) recalls the cryptographic building blocks and introduces protocol verification in the symbolic model; [Sections 3 and 4](#) fix the system and threat models together with our security goals. [Section 5](#) presents the PCP design. The security analysis of PCP is carried out in two sections: [Section 6](#) describes our mechanized symbolic analysis in Tamarin, and [Section 7](#) presents our analysis in the computational model. [Section 8](#) discusses the limitations of our security analysis; [Section 9](#) positions PCP against the relevant literature, and [Section 10](#) draws some conclusions. The Tamarin model is available in the online repository [9], while the manual proofs are in the appendices.

2. Background and notation

This section provides an overview of the cryptographic primitives and notation used throughout the paper. PCP composes five building blocks: a zk-SNARK for proving a statement without revealing its witness ([Section 2.1](#)), a zero-knowledge-friendly hash function ([Section 2.2](#)), Merkle trees for credential membership ([Section 2.3](#)), an attribute-based key encapsulation ([Section 2.4](#)), and standard cryptographic primitives ([Section 2.5](#)). We recall each component, present the APIs, and the property PCP relies on. Finally, we provide an overview of protocol verification under the symbolic model and of the Tamarin prover ([Section 2.6](#)). Below, we denote by $\lambda = 128$ the security level, which quantifies the attacker's effort in bits required to break a scheme.

2.1. zk-SNARK

A zero-knowledge succinct non-interactive argument of knowledge is defined relative to a relation $\mathcal{R}$ between public statements x and secret witnesses w of size polynomial in $|x|$ [10]. The relation $\mathcal{R}$ is NP: given a candidate pair (x, w) , testing whether $(x, w) \in \mathcal{R}$ is efficient, whereas deciding whether a witness for a given x exists may be intractable. The argument allows a prover who already knows a witness w to convince a verifier that x is true and that she knows such a w , without revealing w . To achieve that, the relation $\mathcal{R}$ is encoded as a computation that *checks* the pair (x, w) expressed as an arithmetic circuit over a finite field, or a system of constraints. Proving $(x, w) \in \mathcal{R}$ then becomes proving knowledge of an assignment to those wires of the circuit that satisfies every constraint, with the public wires pinned to x and the rest, including the witness w and all intermediate values, kept secret. We adopt Groth16 [11], a zk-SNARK scheme for exactly expressing and verifying such constraint systems, instantiated over the BN254 pairing-friendly elliptic curve [12]. Here, we neglect its internals and use it as a black box through the three algorithms below:

1. $(pk_{\mathcal{R}}, vk_{\mathcal{R}}) \leftarrow \text{Setup}(1^\lambda, \mathcal{R})$: this function is called only once offline to process (the circuit representing) the relation $\mathcal{R}$. It produces a *proving key* $pk_{\mathcal{R}}$ and a *verification key* $vk_{\mathcal{R}}$; the first key is the input to the Prove algorithm to prove a statement, while the second key is the input to the Verify algorithm to check a statement. Both keys are public and reused for every proof about $\mathcal{R}$.
2. $\pi \leftarrow \text{Prove}(pk_{\mathcal{R}}, x, w)$: this algorithm takes the proving key, a pair $(x, w) \in \mathcal{R}$ and returns a proof π that the statement x is true.
3. $b \leftarrow \text{Verify}(vk_{\mathcal{R}}, x, \pi) \in \{0, 1\}$: this algorithm allows one to check a proof π against the public statement x without seeing the witness w : it returns 1 if x holds and 0 otherwise.

A zk-SNARK additionally guarantees four properties:

1. *Completeness*: an honest proof of a true statement always verifies.
2. *Zero-knowledge*: there exists an efficient algorithm, called *simulator*, that given the statement x alone (and *not* the witness w), produces proofs distributed identically to the real ones. Its existence is exactly what guarantees that π reveals nothing about w beyond the truth of x : anything the verifier sees could have been generated without ever knowing w .
3. *Knowledge soundness*: for every efficient prover that produces a proof accepted by Verify, there is an efficient algorithm, called the *extractor*, that recovers a valid witness w , except with negligible probability. Its existence is what guarantees that an accepting proof can be produced only by someone who actually knows such a w . A strengthening of this property is *weak simulation-extractability* that asks the extractor to keep working even against a prover that has first observed simulated proofs of *other* statements; it implies knowledge soundness.
4. *Succinctness*: proof π has constant size and verification cost depends only on the length of the statement $|x|$, not on the size of relation $\mathcal{R}$.

2.2. Poseidon hash

Poseidon [13] is a cryptographic hash function built to be cheap inside arithmetic circuits, making hashing inside a zk-SNARK circuit practical. As usual, it provides two properties: *collision resistance* and, when keyed with a secret input, *pseudorandomness*.

2.3. Merkle trees

A Merkle tree [14] is a binary tree built on a collision-resistant hash function: each leaf of the tree stores the hash of one data item, and every internal node stores the hash of its two children, so the *root* represents a commitment to the whole set of leaves: changing, adding, or removing any item changes the root.

To prove that a given item belongs to the tree (*membership proof*), one reveals the hashes of the sibling nodes along the path from the corresponding leaf to the root. This sequence of hashes is called the *authentication path*, or *co-path*. A verifier can use it to recompute a candidate root starting from the item: if the resulting root matches the published root, then the item is a member of the tree. The collision resistance of the hash function makes this verification sound, since it prevents an adversary from producing a valid authentication path for an item that is not actually stored as a leaf.

We can use a Merkle tree to implement a *set accumulator*, a data structure that compresses an arbitrarily large set into a single fixed-size digest (the root of the tree). Using this digest, one can prove that a specific element is a member with a compact membership witness (the authentication path). The verifier checks this witness against the digest alone, without ever storing or seeing the set itself. Hereafter, we assume a function $\text{CheckMembership}(d, \text{root}, \text{mp})$ that returns 1 if mp is a valid co-path authenticating d as a leaf of the set accumulator *root*, and 0 otherwise.

2.4. Attribute-based key encapsulation

A key-encapsulation mechanism (KEM) is a cryptographic primitive based on asymmetric cryptography that allows two parties to securely share a symmetric key over a public channel. Basically, the sender generates a key k and an encapsulation ct computed using the receiver's public key, then sends ct to the receiver. The legitimate receiver can recover k from ct using her private key, while ct reveals no information about k to any other party. In an *attribute-based*, or *policy-based*, KEM, legitimate receivers are not specified individually. Instead, encapsulation is performed with respect to a given *policy* that predicates on the attributes of the involved parties, in the style of ABAC, and decapsulation succeeds only for receivers whose attributes satisfy that policy. Given a universe $\mathcal{U}$ of attributes and a policy class $\mathcal{P}$, a policy-based KEM offers four algorithms:

1. $(\text{pk}, \text{sk}_{\text{master}}) \leftarrow \text{Setup}(1^\lambda)$: it generates a public key pk and a master secret key $\text{sk}_{\text{master}}$. The public key pk is used to generate an encapsulation for a given policy, while the master secret key $\text{sk}_{\text{master}}$ is used to generate private keys to decapsulate ciphertexts associated with policies satisfied by their attributes.
2. $\text{sk}_s \leftarrow \text{KeyGen}(\text{sk}_{\text{master}}, \text{attrs})$: it issues a secret key bound to an attribute set $\text{attrs} \subseteq \mathcal{U}$.
3. $(k, \text{ct}) \leftarrow \text{Encaps}(\text{pk}, \text{pol})$: it samples a fresh key k and encapsulates it under a policy $\text{pol} \in \mathcal{P}$.
4. $k \text{ or } \perp \leftarrow \text{Decaps}(\text{sk}_s, \text{ct})$: it recovers k iff the holder's attributes satisfy the policy, returns $\perp$ otherwise.

This scheme can be instantiated in different ways according to the desired expressiveness range: *ciphertext-policy attribute-based encryption* (CP-ABE) [15] supports rich ABAC-style policies over many attributes; in a *group-key* scheme, instead, the policy checks mere membership and encapsulation are standard key wrap [16] under a shared group key.

2.5. Standard cryptographic primitives

We rely on standard symmetric encryption and digital signatures: the former ensures the confidentiality and integrity of payloads, while the latter ensures the authenticity of messages exchanged across the various protocols.

For the encryption, we assume the following operations:

1. $\text{ct} \leftarrow \text{Enc}(k, m)$: encrypts a message m with a key k , producing a ciphertext ct .
2. $m \text{ or } \perp \leftarrow \text{Dec}(k, \text{ct})$: decrypts a ciphertext ct using the key k . If the operation succeeds, it returns the corresponding plaintext m , otherwise an error value $\perp$.

In our computational analysis, we instantiate this scheme using standard AES-GCM authenticated encryption, but for brevity, we omit from our API the management of the nonce and the produced tag.

While for digital signatures, we assume:

1. $(\text{sk}, \text{vk}) \leftarrow \text{KeyGen}(1^\lambda)$: generates a key pair consisting of a signing key sk and a verification key vk .
2. $\sigma \leftarrow \text{Sign}(\text{sk}, m)$: signs a message m using the signing key sk .
3. $b \leftarrow \text{Verify}(\text{vk}, m, \sigma) \in \{0, 1\}$: checks a signature σ against the verification key vk .

In our computational analysis, we instantiate this scheme using standard EdDSA over a suitable curve.

2.6. Protocol verification in the symbolic model

We analyze the protocol-level security of PCP in the *symbolic model*, also known as the *Dolev–Yao model* [17]. In this model, we assume that the adversary completely controls the network: it may intercept, inject, reorder, drop, and replay messages at will, but it cannot break cryptography (*perfect cryptography assumption*). Moreover, cryptographic primitives are modeled as *function symbols* over a given signature, and messages are modeled as *terms* constructed from these functions. The semantics of the primitives is specified

by a set of *equations*, which define how terms built from these primitives can be simplified or related to one another. The adversary can compute new messages only by applying the available primitives and the corresponding equations. For instance, symmetric encryption is modeled by two function symbols *enc* and *dec*, where *enc*(*m*, *k*) is the encryption of message *m* under key *k* and *dec*(*x*, *k*) the decryption of *x* with *k*, together with the equation

$$\text{dec}(\text{enc}(m, k), k) = m,$$

establishing that an encrypted term *enc*(*m*, *k*) can be decrypted only by whoever holds *k*. Using the same approach, one can model richer primitives.

The symbolic model enables *automated* analysis of protocols via formal methods: the behaviors the model admits are in general infinitely many, yet a tool can reason about all of them mechanically, either exhibiting an attack or proving its absence. Here, we use one such tool for our security analysis: the Tamarin prover [7]. In Tamarin, the protocol and the adversary are specified by multiset-rewriting *rules* that define a labeled transition system. Its state is a multiset of *facts* that represent the adversary's knowledge, the messages currently on the network, the freshly generated values, and the honest parties' state.

Concretely, a rule has the form [*premises*]–[*actions*] → [*conclusions*]: firing it requires that all premise facts be present in the current state; it then consumes them, records the action facts on the execution trace, and adds the conclusion facts to the state. A *!*-prefixed fact is *persistent*: it is matched but never consumed, so it can be reused. The fact *Fr*(*n*) is a primitive to sample a fresh unguessable value (a nonce), and *\$x* denotes a public name. Three built-in facts wire a protocol to the network adversary: *Out*(*m*) sends a message over the network, *In*(*m*) receives one from it, and *K*(*m*) records *m* in the adversary's knowledge. Since the adversary controls the network, it learns every message in *Out* and may supply as *In* any term it can derive. Because rules may fire in any applicable order and persistent facts can be reused, a single model already captures arbitrarily many concurrent protocol sessions. As a minimal example, the following two rules register a key and later use it:

```
rule Register:
  [ Fr(~k) ]                // sample a fresh key
--[ Reg($A, ~k) ]->        // record the generation event on the trace
  [ !Key($A, ~k) ]         // record k is the key of A in a persistent fact
rule Use:
  [ !Key($A, k) ]          // reuse the stored key
--[ Used($A, k) ]->
  [ \, ]
```

rule *Register* can fire at any time: its only premise, *Fr*(*~k*), is always satisfiable because Tamarin's built-in freshness rule always supplies fresh values. Firing it creates a key *~k* for party *\$A*, logs the event *Reg*(*\$A*, *~k*) on the execution trace, and stores the key in the persistent fact *!Key*(*\$A*, *~k*). Rule *Use* may then fire afterward: it reads that stored key (without removing it, since the fact is persistent) and logs a *Used*(*\$A*, *k*) event on the trace. The two action facts *Reg* and *Used* are what a property can later refer to.

Security properties predicate over protocol traces and are stated as *lemmas*. More precisely, a lemma states a property over the recorded actions in the traces and their timepoints *#i*, *#j*. For our example above, the following lemma says that a key is only ever used after it was registered:

```
lemma use_after_register:
  "All A k #i. Used(A,k)@i ==> Ex #j. Reg(A,k)@j & #j < #i"
```

To prove such a lemma, Tamarin negates it and searches for a trace that satisfies the negation. To achieve that, it solves symbolic constraints, each standing for infinitely many concrete traces. It either confirms the property or returns a counterexample, which is a concrete attack. The search is not guaranteed to terminate, but when it does, the answer is definitive.

Three further notions round out the machinery we use. First, besides lemmas Tamarin has *restrictions*: a restriction is a global side-condition that discards every run violating it, letting the modeler impose a requirement a single rule cannot enforce locally, e.g., a signature verifies before a rule proceeds, or a setup step happens at most once. Second, the lemmas themselves come in two flavors: the *all-traces* lemma above must hold on every run (“this never happens”), whereas an *exists-trace* lemma is proved by exhibiting a single run (“this can happen”). These last lemmas are the standard way to certify that a model is not *vacuously* satisfiable because it admits the empty set of traces. Third, an honest party's long-term secrets reach the adversary only through an explicit *reveal* rule, so the modeler fixes exactly which compromises are in scope. We use all these notions in our symbolic analysis of [Section 6](#).

3. System model and overview

The architecture of PCP consists of two planes ([Fig. 1](#)): a *data plane* that delivers application data, and a *control plane* that manages client enrollment, credentials and epochs. The two planes are channel-disjoint with one exception (see below) and rely on distinct trust assumptions ([Section 4.1](#)).

In the data plane, *publishers* produce application data inside protected envelopes and publish it on specific topics. *Subscribers* subscribe to specific topics, receive the protected envelopes, and validate and decrypt them. The *broker* manages subscriptions and forwards messages; PCP requires it to satisfy the following properties:

1. opaque payload forwarding, i.e., it does not transform payload bytes;

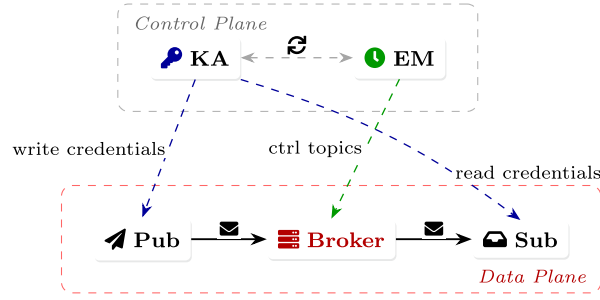


Fig. 1. PCP architecture: the *data plane* (bottom) carries opaque envelopes through an untrusted broker. The *control plane* (top) includes a Key Authority (KA), which issues write and read credentials over a direct authenticated channel to clients, and the Epoch Manager (EM), which publishes signed checkpoints and revocation notifications on dedicated broker control topics. KA and EM coordinate over a direct authenticated channel.

2. topic-based routing, i.e., message delivery is determined by topic matching.

Many real broker implementations, such as MQTT and NATS, satisfy these properties.

In the control plane, the *Key Authority (KA)* enrolls clients and issues write credentials to publishers and read credentials to subscribers. It communicates with clients over a direct, mutually-authenticated, and confidential channel: *enrollment*, *credential issuance*, and *rekeying* messages do not traverse the data plane, and thus, the broker. At system initialization, the KA also runs the one-time Groth16 trusted setup for PCP's proof circuit, producing the public proving/verification pair (pk_{gr}, vk_{gr}) , and then discards the secret parameters used during circuit and keys generation. We assume that the KA hands over, on the same authenticated channel, the proving key pk_{gr} to publishers and the verification key vk_{gr} to subscribers, together with the EM verification key vk_{em} that every client uses to authenticate checkpoints. At enrollment, the KA validates client attributes by invoking an abstract *attribute oracle* which returns an authoritative set of attributes held by the authenticated client. PCP is parametric on this oracle: any instantiation that returns authoritative attributes for clients to the KA is acceptable, e.g., an attribute store provisioned out-of-band, signed attribute assertions issued by an external attribute authority, or nonce-based challenge-response with a live attribute authority. The *Epoch Manager (EM)* drives revocation and epoch lifecycle: it tracks the set of currently valid credentials, signs, and broadcasts authenticated freshness updates and participates in the revocation mechanism. EM messages carry no secrets and are signed end-to-end, so the EM can rely on the broker to communicate with clients over dedicated control topics (an exception to plane disjointness). Clients verify messages received over the control topic against the EM's verification key vk_{em} .

KA and EM coordinate over a direct authenticated channel: every credential operation at the KA, e.g., issuance, soft revocation, hard revocation at the epoch boundary, is mirrored to the EM, which updates the credential set and reflects the change in the next freshness updates.

Finally, we outline how PCP works (the full details are in [Section 5](#)). Policies are attached to *topic scopes*: a topic scope *sid* is a topic, or a set of topics, whose lifetime is split into *epochs* *eid*. Each publisher enrolled in *sid* holds a long-term *write credential cred*, which the EM stores in a cryptographic *accumulator* whose root is broadcast in a signed *checkpoint*. A published message carries the encrypted payload, a zero-knowledge proof that the sender's credential is in the accumulator, and a per-epoch *pseudonym* that identifies the sender without revealing its long-term identity. Subscribers verify the proof before attempting decryption; read authorization is enforced by the encryption itself, as the symmetric key of each message is encapsulated under the read policy through the interface *ReadAuth*, whose keys are derived from a read-side master key by the KA. Revocation occurs via two mechanisms: a *soft revocation* revokes write-credentials from the EM accumulator within the current epoch, whereas a *hard revocation* withdraws a subscriber's read capability at the next epoch boundary: this mechanism requires the KA to rotate its master key and reissues read credentials to all the authorized subscribers for the topic scope *sid*. [Table 1](#) summarizes the notation used throughout the paper.

4. Threat model and security goals

Here, we present our assumptions about the adversary and the threats she can pose, followed by our security goals.

4.1. Assumptions and threats

We assume an adversary $\mathcal{A}$ that acts at two levels. At the network level, it follows the Dolev-Yao model on the data plane: it may intercept, modify, reorder, drop, delay, and inject messages. It may further adaptively corrupt a bounded fraction of publishers and subscribers, obtaining their complete local state. The control-plane channels from the KA to clients and from the KA to the EM are mutually authenticated and confidential, and lie outside $\mathcal{A}$'s direct control. At the level of the cryptographic primitives, $\mathcal{A}$ attempts to break them, but it behaves as a probabilistic polynomial-time Turing machine, and hence succeeds with negligible probability. The per-actor trust assumptions are:

- A1** Broker is untrusted and realizes the adversary's data-plane capabilities: it may observe, reorder, drop, delay, replay, or inject traffic, while remaining compliant with its native protocol.

Table 1
Summary of notation and symbols.

Symbol	Meaning
λ	security parameter (= 128)
sid, eid	topic scope; per-scope epoch index
$\text{policy}^w, \text{policy}^r$	write-/read-side attribute predicates
Poseidon	ZK-friendly hash function
dev	unique device identifier
cred	long-term writer credential
wid	accumulator leaf
nym	per-epoch writer pseudonym
$\text{Tree}_{\text{sid}, \text{eid}}, \text{accRoot}$	credential accumulator and its root
mp	Merkle authentication path of a leaf
$\text{seq}, \text{seq}_{\text{local}}$	per-scope counter; subscriber sequence number
chk	checkpoint ($\text{eid}, \text{accRoot}, \text{seq}, \sigma_{em}$)
$\text{sk}_{ka}, \text{vk}_{ka}$	KA signing keypair
$\text{sk}_{em}, \text{vk}_{em}$	EM signing keypair
σ_{ka}, σ_{em}	KA / EM signatures
$\text{pk}_{gr}, \text{vk}_{gr}$	Groth16 proving and verification keys
k, m	per-message symmetric key; plaintext
d	binding digest
$\text{ct}_{\text{auth}}, c$	ReadAuth encapsulation; ciphertext
π	Groth16 proof
$\text{attrs}_w, \text{attrs}_s$	publisher/subscriber attribute sets
$\text{pk}, \text{sk}_{\text{master}}, \text{sk}_s$	ReadAuth public, master, and subscriber read key

- A2** KA is semi-trusted for confidentiality. A KA compromise affects payload confidentiality but not proof soundness: an adversary holding the KA's read-side master key cannot forge proofs that pass verification at honest subscribers.
- A3** EM is semi-trusted for availability and freshness. Its messages to clients are broker-mediated but signed and verified end-to-end, so $\mathcal{A}$ may delay or drop them but cannot forge or modify them undetected. EM compromise may extend revocation windows or produce stale checkpoints, but does not break payload confidentiality or proof soundness.
- A4** Subscribers may collude with each other and with the broker; revoked credential holders may collude with current subscribers.
- A5** Publishers are not assumed to be honest: individual publishers may submit malformed envelopes, stale credentials, or invalid proofs.

Overall, the broker is fully untrusted, whereas the two control-plane authorities are only *semi-trusted*: their authority is split between them, so that compromising a single authority has a bounded and controlled impact. Given these assumptions, $\mathcal{A}$ may attempt the following threats against PCP:

- T1 Forged publish:** $\mathcal{A}$, without holding any valid write credential, attempts to inject an envelope that an honest subscriber accepts.
- T2 Replay:** $\mathcal{A}$ captures a valid envelope and re-emits it after the originating publisher's soft revocation or after an epoch boundary.
- T3 Cross topic acceptance:** adversary $\mathcal{A}$ holds a credential valid for topic sid_1 and attempts to have subscribers of topic $\text{sid}_2 \neq \text{sid}_1$ accept it, e.g., by relying on the broker to re-route a message across several topics.
- T4 Publisher compromise:** adversary $\mathcal{A}$ corrupts a publisher and inherits its write credential cred , attempting to produce accepted envelopes after compromise.
- T5 Subscriber compromise:** adversary $\mathcal{A}$ corrupts a subscriber and inherits its ReadAuth decryption material, attempting to decrypt payloads after compromise.
- T6 KA compromise:** adversary $\mathcal{A}$ obtains the KA's read-side master key, attempting to decrypt protected payloads.
- T7 EM compromise:** adversary $\mathcal{A}$ obtains the EM's checkpoint-signing key and the accumulator set of enrolled publishers, attempting to delay or suppress soft revocations and produce stale checkpoints.

4.2. Security goals

Given the threat model above, we aim to achieve the following security goals. For each goal, we report the trust assumptions it relies on in parentheses (assumption **A1** is implicit in all of them.)

- G1 Payload confidentiality:** an adversary $\mathcal{A}$ controlling the broker cannot distinguish the plaintext of a protected payload from a random value. (**A2, A4**)
- G2 Forward secrecy after revocation:** a client device revoked in epoch i cannot decrypt payloads of epoch $i + 1$ and beyond. (**A2, A3, A4**)
- G3 Backward secrecy after join:** A client joining in epoch j cannot decrypt payloads of epoch $j - 1$ and earlier. (**A2, A4**)
- G4 Write-authorization soundness:** only publishers with a valid credential can produce accepted proofs. (**A2, A3, A5**)
- G5 Bilateral authorization privacy:** the broker cannot determine whether a subscriber rejects a message because of a failed proof or because of missing decryption keys; the broker and any colluding subscribers cannot link envelopes from different epochs to the same publisher. (**A2, A4**)

- G6** *Binding soundness*: an adversary cannot produce a valid proof that accepts a payload different from the one chosen by the publisher. (A5)
- G7** *Contained authority compromise*: KA compromise, disclosure of its read-side master key (threat T6), does not break proof soundness, and EM compromise (threat T7) does not break payload confidentiality. (It holds when either A2 or A3 is dropped.)

The security analysis of Sections 6 and 7 formally shows that PCP achieves these security goals. More precisely, goals G4 and G6 are discharged in the symbolic model, goal G5 in the computational model, while goals G1–G3 and G7 are split between them, with the symbolic model establishing the safety facet and the computational model the corresponding bound. Tables 2 and 3 map each goal to the lemma or theorem that establishes it, together with the cryptographic assumptions it relies on.

However, these goals come with the following explicit caveats. Goal G5 does not claim intra-epoch unlinkability: messages from the same publisher remain linkable within a single epoch. Goal G7 does not extend to simultaneous KA and EM compromise, which is equivalent to full authority compromise. Publisher-endpoint integrity (trusted sensing, remote attestation) lies outside the threat model: only credential possession at publish time is attested. Likewise, availability is outside our goals: $\mathcal{A}$ may flood the clients with, e.g., TCP packets. At the PCP level, we cannot prevent it from doing so. A real deployment of PCP may address this attack by adopting standard network countermeasures. Moreover, a flood of invalid envelopes costs a subscriber one signature check and one proof verification, never a ReadAuth decapsulation (see Algorithm 2). Since envelopes carry a publisher pseudonym (Section 5.2), which is stable within an epoch, deployments of PCP may implement a rate limit per pseudonym. However, a complete denial-of-service defense is outside our scope.

5. Our proposal

This section presents the mechanics of PCP on top of the architecture of Section 3 and the threat model of Section 4. We proceed bottom-up: we first describe our authorization model (Section 5.1), and the envelope format (Section 5.2); then we present the four operational phases that implement PCP: *enrollment* (Section 5.3), *publish* (Section 5.4), *verify-before-decrypt* (Section 5.5), and *soft and hard revocation* (Section 5.6). Finally, we discuss how clients handle checkpoint receptions and how the late-join mechanism works (Section 5.7). Throughout the section, we use the notation of Table 1.

5.1. Authorization model: topic scopes and epochs

PCP adopts an *attribute-based* authorization model in the spirit of ABAC [1]: every client carries a set of attributes (role, tenant, device class, ...), and the right to write to or read from a specific topic is determined by predicates over those attributes. Two clients with the same attributes obtain the same authorization, and a single attribute update affects every policy that references that attribute. Unlike conventional ABAC, in which there is a central policy enforcement point, PCP performs policy enforcement *end-to-end* at the clients by checking cryptographic possession of attribute-bound credentials.

Policies are defined per *topic scope*. A topic scope sid identifies a protected authorization domain and may correspond to a single topic, e.g., `plant/line1/temp`, a single-level prefix, e.g., `plant/line1/+`, or a multi-level subtree, e.g., `plant/line1/#`; each protected message is associated with exactly one sid , and is governed by the policy attached to that scope. Within a topic scope, time is partitioned into a sequence of *epochs*, indexed by $eid = 1, 2, \dots$. During an epoch eid , the policies and the KA master key for sid do not change, and every read credential issued in eid remains valid. A transition to a new epoch is initiated by the KA whenever an event requires a *hard revocation* (Section 5.6) of read credentials; at that point, the KA rotates its master key and reissues read credentials to the surviving subscribers. Instead, the joins of new clients and revocations of write credentials are absorbed *within* the current epoch, without an epoch transition.

Policies are attached to pairs (sid, eid) and consists of two attribute predicates: $policy_{sid, eid}^w$ governs which publishers may write to sid during eid , and $policy_{sid, eid}^r$ governs which subscribers may read. The two are evaluated asymmetrically. The read-side predicate is enforced *per envelope*: a subscriber that does not satisfy $policy_{sid, eid}^r$ recovers no information about the payload. The write-side predicate is checked *once per enrollment*: on a successful check, the publisher receives a write credential bound to (sid, eid) , and on every subsequent message, it only proves *possession* of that credential. Skipping the write-policy check on every published message keeps the publishing process lightweight, but it also means that an issued credential remains valid until explicit invalidation: any change in publisher attributes causing the revocation of a credential must be enforced through a dedicated revocation procedure (Section 5.6).

PCP enforces the read-side predicate through a single interface, ReadAuth: an attribute-based key encapsulation (Section 2.4) whose policy class is the one above and which our construction uses exclusively via the following algorithms. A publisher calls ReadAuth.Encaps to wrap a fresh per-message key under $policy_{sid, eid}^r$, and a subscriber calls ReadAuth.Decaps to recover it only if its attributes satisfy that policy. Confining read-side authorization behind this interface makes the read backend *interchangeable* without touching the proof, envelope, or revocation machinery.

5.2. Protected envelope format

A protected envelope is abstractly represented as the tuple

$$\text{Env} = (\text{ver}, \text{sid}, \text{eid}, \text{bid}, \text{nym}, \text{chk}, d, \text{ct}, \pi),$$

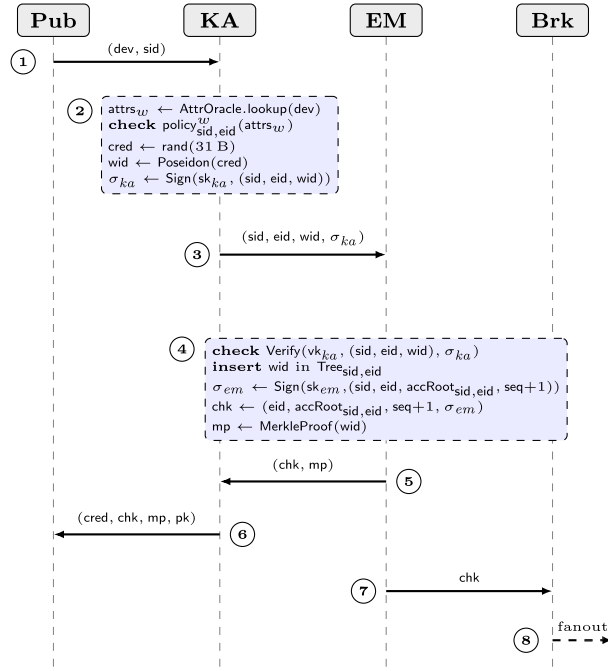


Fig. 2. Publisher enrollment: message-sequence diagram across publisher, KA, EM, and broker. Solid arrows are messages, dashed boxes are local computations, and numbered circles fix the step order; the dashed arrow at Step (8) is the broker's fan-out of the checkpoint to the clients currently attached to the control topic.

where the fields are interpreted as follows: *ver* denotes the protocol version; *sid* is the topic scope to which the message has been published; *eid* is the epoch active at publication time; *bid* identifies the read-credential backend used to produce the ciphertext *ct*; *nym* is a per-epoch writer pseudonym, stable within an epoch but computationally unlinkable across epochs; *chk* is a checkpoint reference (*eid*, *accRoot*, *seq*, σ_{em}) that anchors the envelope to the current per-scope authorization state, where *accRoot* is a cryptographic digest of the currently valid credentials for the scope, *seq* is a monotonic per-scope sequence number, and σ_{em} is the EM's signature over the scope *sid* together with this triple, so the checkpoint cannot be lifted into another scope; *d* is a binding commitment to the pair (*k*, *m*) used in *ct* that anchors π to that specific key and plaintext while keeping both values hidden from any verifier that does not hold *k*; *ct* is the protected ciphertext, structured as (*ct_{auth}*, *c*), where *ct_{auth}* encapsulates a symmetric key *k* such that only credentials matching $\text{policy}_{sid,eid}^w$ can recover it, and *c* is the authenticated encryption of the plaintext *m* under *k*; π is a zero-knowledge proof attesting that the publisher holds a valid write credential for (*sid*, *eid*) and that the envelope is well-formed with respect to *d*, *nym*, and *accRoot* (Section 5.4).

5.3. Client enrollment

Enrollment is performed by clients once per topic scope using the direct authenticated channel with KA, on which the device identifier *dev* is authenticated end-to-end. Two procedures exist: *publisher enrollment*, which delivers the write credentials according to predicate $\text{policy}_{sid,eid}^w$ and registers them at the EM, and *subscriber enrollment*, which delivers a read credential under $\text{policy}_{sid,eid}^r$. Both procedures evaluate the policies over the attributes returned by the attribute oracle, never on values self-claimed by clients.

5.3.1. Publisher enrollment

This is a four-message protocol between three authenticated parties (publisher, KA, EM) plus the broker that disseminates the resulting checkpoint. Fig. 2 shows the full sequence of messages where every step of the protocol is denoted by a numeric label.

In Step (1), the publisher sends an enrollment request to KA, carrying its channel-authenticated device identifier *dev* and the target topic scope *sid*. In Step (2), the KA looks up the device's write attributes $\text{attrsw} \leftarrow \text{AttrOracle.lookup}(\text{dev})$ from the attribute oracle and evaluates $\text{policy}_{sid,eid}^w(\text{attrsw})$. On success the KA mints a fresh random *cred*; commits to it via $\text{wid} \leftarrow \text{Poseidon}(\text{cred})$; and signs an attestation $\sigma_{ka} \leftarrow \text{Sign}(\text{sk}_{ka}, (\text{sid}, \text{eid}, \text{wid}))$ authorizing the admission of the publisher in the set of authorized writers. The attestation is sent to EM and deliberately omits *dev*: the EM never learns the device identity. The KA records the *dev* $\rightarrow$ *cred* binding in its local credential ledger for accountability. Step (3) is the cross-authority handover: KA forwards (*sid*, *eid*, *wid*, σ_{ka}) to the EM but retains *cred* locally. In Step (4), the EM first verifies σ_{ka} over (*sid*, *eid*, *wid*) under the KA's verifying key *vk_{ka}*, rejecting the registration if it does not verify; only then does it add the new entry into the scope's credential accumulator. Channel authentication proves the request originates at the KA endpoint, while σ_{ka} proves the KA authorized *this specific* *wid*. The accumulator is realized as a fixed-depth Merkle tree *Tree_{sid,eid}* whose leaves are the *wid* values of the publishers currently admitted to write on (*sid*, *eid*); its root *accRoot_{sid,eid}* is therefore a constant-size cryptographic digest of the full current credential set for that topic scope. The EM adds the received *wid* in the next

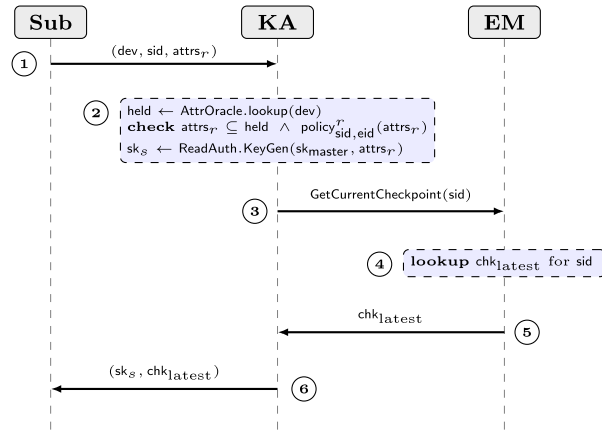


Fig. 3. Subscriber enrollment as a message sequence chart. Steps (1), (3), (5), (6) are message exchanges, while Steps (2) and (4) are local computations. Unlike publisher enrollment (Fig. 2), no accumulator update takes place at the EM, since subscribers are not admitted into $\text{Tree}_{\text{sid},\text{eid}}$; the EM only returns $\text{chk}_{\text{latest}}$, which serves as the subscriber's initial freshness anchor for sid.

free leaf, and recomputes the root of the tree by rehashing along the modified co-path. It then advances the per-topic-scope sequence number seq and signs the new state, producing the checkpoint $\text{chk} = (\text{eid}, \text{accRoot}_{\text{sid},\text{eid}}, \text{seq}+1, \sigma_{em})$ that anchors every subsequent envelope on sid to the updated authorization state. Note that the per-topic-scope sequence number seq moves monotonically on both admissions and revocations (Section 5.6), so that a subscriber can reject stale or replayed checkpoints from a single local counter. Finally, the EM extracts the Merkle membership witness mp for the freshly inserted leaf. This is an ordered list of co-path sibling hashes together with the leaf-index path bits, which the publisher uses as a private witness to prove $\text{wid} \in \text{Tree}_{\text{sid},\text{eid}}$ without revealing wid itself (Section 5.4). Steps (5) and (6) together form the publisher's bootstrap path: the EM returns (chk, mp) to the KA as the response of the registration, and the KA forwards $(\text{cred}, \text{chk}, \text{mp}, \text{pk})$ to the publisher. After these steps, the publisher holds cred as the private secret of every future ZK proof, chk as its initial freshness anchor for sid, mp as the private witness of accumulator-membership, and the ReadAuth public key pk to wrap each message key during the publish phase. In parallel with Steps (5) and (6), Step (7) propagates the new checkpoint. The EM publishes chk on its dedicated control topic for sid. In Step (8), the broker pushes chk to every client currently attached to the control topic, which advances their local state (see Section 5.7).

5.3.2. Subscriber enrollment

This is a four-message exchange between a subscriber, KA, and EM, shown in Fig. 3. The protocol mirrors the publisher enrollment: it presents the same authenticated channels, the same KA–EM interactions, but causes no accumulator update inside EM, since subscribers are not registered into $\text{Tree}_{\text{sid},\text{eid}}$. More precisely, subscriber enrollment proceeds as follows.

In Step (1), the subscriber sends an enrollment request to KA, carrying its device identifier dev , the target topic scope sid , and a requested attribute subset attrs_r over which it wishes its read key to be issued. Note that the set attrs_r affects the issued credential: under CP-ABE the attribute set is materialized inside a subscriber secret key sk_s , so a strict subset of the attributes yields a least-privilege key. In Step (2), the KA looks up the device's held attributes $\text{held} \leftarrow \text{AttrOracle.lookup}(\text{dev})$, checks both that $\text{attrs}_r \subseteq \text{held}$ (no elevation) and that $\text{policy}_{\text{sid},\text{eid}}^r(\text{attrs}_r)$ holds, and on success, derives the read credential $\text{sk}_s = \text{ReadAuth.KeyGen}(\text{sk}_{\text{master}}, \text{attrs}_r)$. In Step (3), the KA fetches the latest checkpoint from the EM via a $\text{GetCurrentCheckpoint}(\text{sid})$ message; the EM looks up $\text{chk}_{\text{latest}}$ in its scope store (Step (4)) and returns it (Step (5)) without updating $\text{Tree}_{\text{sid},\text{eid}}$. Finally, in Step (6), the KA delivers the tuple $(\text{sk}_s, \text{chk}_{\text{latest}})$ to the subscriber: the read credential sk_s and the latest checkpoint $\text{chk}_{\text{latest}}$ as anchor for sid.

5.4. Message publication

In a typical pub/sub middleware, a publisher can publish a message m on a topic t through a primitive $\text{broker.publish}(t, m)$. The broker receives the message and forwards it to the corresponding subscribers, provided that the publisher is authorized on that topic. PCP creates a wrapper around this standard publishing primitive to add authorization. Concretely, the publisher builds an envelope that carries

1. a ciphertext c obtained encrypting message m using a fresh symmetric key k ;
2. the key k wrapped via ReadAuth so that only subscribers satisfying the read policy can unwrap it and recover m ;
3. a per-epoch pseudonym nym that lets the broker recognizes repeated messages from the same publisher within the epoch eid without learning her identity; and
4. a cryptographic proof π that binds (i) and (ii) to the fact that the publisher is currently authorized to publish messages on the topic scope sid during the epoch eid.

The envelope is then sent to the broker, which forwards it on the topic without altering its contents. The rest of this subsection unpacks the three statements that π attests, then walks through the per-message envelope construction of Algorithm 1.

5.4.1. The authorization proof

The proof π inside the envelope simultaneously establishes the following three statements about the publisher:

1. *Authorized write*: the publisher is authorized to publish messages on (sid, eid) . This is established by showing that its wid is a leaf in the accumulator tree maintained by the EM. A copy of the root chk.accRoot signed by EM travels inside the envelope in the checkpoint field.
2. *Pseudonym coherence*: the per-epoch pseudonym nym of this publisher in the envelope is genuine and identifies the publisher in the epoch eid .
3. *Ciphertext binding*: the encrypted payload in the envelope is exactly the message sent by the publisher, and not a different one swapped in by anyone along the way. This is established by showing that the envelope-visible digest d equals $\text{Poseidon}(k, m)$ on the (k, m) pair that the publisher used to produce the encrypted payload.

Intuitively, the proof π attests that the publisher in the epoch eid knows (cred, k, m) such that

$$\underbrace{\text{Poseidon}(\text{cred}) \in \text{Tree}_{\text{sid}, \text{eid}}}_{\text{Authorized write}} \bigwedge \underbrace{\text{nym} = \text{Poseidon}(\text{cred}, \text{eid})}_{\text{Pseudonym coherence}} \bigwedge \underbrace{d = \text{Poseidon}(k, m)}_{\text{Ciphertext binding}} \quad (1)$$

represented as a single Groth16 circuit: the publisher's secret cred is a private input feeding both the Authorized write, and the Pseudonym coherence checks, and the (k, m) pair behind the ciphertext is another private input fed into the binding-digest check. A valid π therefore exists only when the publisher's inputs satisfy all three predicates at once, and by Groth16 knowledge soundness, no one can produce π without actually knowing such inputs. Two operational consequences follow. First, every envelope from a given publisher inside eid carries the same nym , so the broker can track the publisher within that epoch without learning her identity; across epochs nym rotates. Second, removing wid from the EM's tree (Section 5.6) neutralizes the publisher immediately: subscribers learn the new accRoot via the EM's signed checkpoint chk , and every future π from that publisher fails to verify against it.

5.4.2. Constructing the envelope

The publish phase is local: there is no interaction with the KA and the EM to build the envelope before it is sent to the broker. The provided API mirrors the standard one: a user invokes $\text{PCP.publish}(\text{sid}, m)$, supplying only the topic scope sid and the plaintext m . Everything else needed to build the envelope is drawn from the publisher's local state, established at setup and enrollment, and kept updated as the publisher consumes EM control-topic updates. Specifically, the publisher's local state includes:

1. the write credential cred , the private witness of every proof;
2. the latest checkpoint chk (carrying eid and accRoot) and the publisher's authentication path against chk.accRoot ;
3. the Groth16 proving key pk_{gr} and the EM verification key vk_{em} (used to authenticate control-topic checkpoints);
4. the ReadAuth public key pk for the current epoch, delivered at enrollment; and the read policy $\text{policy}_{\text{sid}, \text{eid}}^r$, which is local to the publisher.

A call to $\text{PCP.publish}(\text{sid}, m)$ executes Algorithm 1 in three stages: (i) encapsulating the read-side key and encrypting the payload (lines 1–2), (ii) computing the binding digest d that ties the proof to the ciphertext and the per-epoch pseudonym nym that ties it to the publisher (lines 3–4), and (iii) generating π , assembling the envelope, and publishing it (lines 5–9).

Algorithm 1 Publish an envelope.

Require: *Input:* topic scope sid , plaintext m

Local state: write credential cred , latest checkpoint chk , Merkle path for chk.accRoot , read policy $\text{policy}_{\text{sid}, \text{eid}}^r$, ReadAuth public key

pk , Groth16 proving key pk_{gr}

1: $(k, \text{ct}_{\text{auth}}) \leftarrow \text{ReadAuth.Encaps}(\text{pk}, \text{policy}_{\text{sid}, \text{eid}}^r)$

2: $c \leftarrow \text{Enc}(k, m)$

3: $d \leftarrow \text{Poseidon}(k, m)$

▷ binding digest

4: $\text{nym} \leftarrow \text{Poseidon}(\text{cred}, \text{eid})$

▷ per-epoch pseudonym

5: $w \leftarrow (m, k, \text{cred}, \text{mp})$

▷ circuit witness

6: $\text{pub} \leftarrow (d, \text{sid}, \text{eid}, \text{nym}, \text{chk}, \text{accRoot})$

7: $\pi \leftarrow \text{Groth16.Prove}(\text{pk}_{gr}, \text{pub}, w)$

8: $\text{Env} \leftarrow (\text{ver}, \text{sid}, \text{eid}, \text{bid}, \text{nym}, \text{chk}, d, (\text{ct}_{\text{auth}}, c), \pi)$

9: $\text{broker.publish}(\text{topic}_{\text{sid}}, \text{Env})$

In stage (i), we invoke $\text{ReadAuth.Encaps}(\text{pk}, \text{policy}_{\text{sid}, \text{eid}}^r)$ (line 1) to return a fresh per-message symmetric key k together with an encapsulation ct_{auth} from which only subscribers satisfying the read policy $\text{policy}_{\text{sid}, \text{eid}}^r$ can recover k via ReadAuth.Decaps . Then, we encrypt the plaintext m using k (line 2). In stage (ii), we compute the binding digest $d \leftarrow \text{Poseidon}(k, m)$ (line 3) that ties the ciphertext c to the proof π below. Then, we derive the per-epoch pseudonym $\text{nym} \leftarrow \text{Poseidon}(\text{cred}, \text{eid})$ (line 4). Both d and nym become public inputs of π and travel in the envelope.

Stage (iii) consumes the outputs of the previous two stages to build π . Formally, π attests that a specific pair (pub, w) , where pub is a public input visible to the verifier and w is a witness kept private at the prover, belongs to the following relation $\mathcal{R}_{\text{PCP}}$:

$$\mathcal{R}_{\text{PCP}} = \left\{ \left(\underbrace{(d, \text{sid}, \text{eid}, \text{nym}, \text{accRoot})}_{\text{pub}}, \underbrace{(m, k, \text{cred}, \text{mp})}_{w} \right) \mid \begin{aligned} &\text{CheckMembership}(\text{Poseidon}(\text{cred}), \text{accRoot}, \text{mp}) = 1 \wedge \\ &\text{nym} = \text{Poseidon}(\text{cred}, \text{eid}) \wedge \\ &d = \text{Poseidon}(k, m) \end{aligned} \right\}, \quad (2)$$

that instantiates (1): the abstract membership $\text{Poseidon}(\text{cred}) \in \text{Tree}_{\text{sid}, \text{eid}}$ becomes a Merkle-path check against accRoot . Note that the topic scope sid in pub plays a key role, even though no constraint involves sid , to ensure domain separation. The idea is that Groth16 binds the proof π to the exact pub input used at generation time, so binding π to the specific topic scope sid prevents an attacker from repackaging a valid envelope from one topic scope to another: changing the envelope's sid field would feed a different pub to the verifier, and the proof check fails. The publisher's task in this stage is therefore to assemble a concrete $(\text{pub}, w) \in \mathcal{R}_{\text{PCP}}$ and produce π attesting its membership.

To achieve that, we proceed as follows. We collect the witness $w = (m, k, \text{cred}, \text{mp})$ (line 5): the input plaintext m and the symmetric key k , the credential cred , and the Merkle authentication path required by the first conjunct of $\mathcal{R}_{\text{PCP}}$. Then, we assemble the public inputs $\text{pub} = (d, \text{sid}, \text{eid}, \text{nym}, \text{chk}, \text{accRoot})$ (line 6) from d and nym produced in stage (ii), plus the envelope-visible context $(\text{sid}, \text{eid}, \text{accRoot})$. Next, we build the proof by invoking $\pi \leftarrow \text{Groth16.Prove}(\text{pk}_{gr}, \text{pub}, w)$ (line 7). Finally, we package the envelope as described in Section 5.2 (line 8) and publish the result on the topic corresponding to the topic scope sid (line 9) using the primitive provided by the broker.

5.5. Message reception

In a typical pub/sub middleware, a subscriber registers interest in a topic t via a call to $\text{broker.subscribe}(t)$ primitive and receives every payload the broker routes to that topic. PCP provides a similar primitive $\text{PCP.subscribe}(\text{sid})$ to attach to a topic scope sid : every incoming envelope flows through a *verify-before-decrypt* process that delivers a plaintext m to the application only after the writer's and the reader's authorization have been cryptographically confirmed.

The *verify-before-decrypt* process is *subscriber-local*: no interaction with the KA or the EM is needed, but the subscriber relies on the following information from her local state populated at enrollment (Section 5.3) and refreshed on EM control-topic updates:

1. the read credential sk_s issued by the KA under policy $_{\text{sid}, \text{eid}}$;
2. the EM-signed checkpoint sequence $\text{seq}_{\text{local}}$, monotonic across all checkpoints seen so far;
3. the Groth16 verification key vk_{gr} and the EM verification key vk_{em} , both distributed at setup (Section 3);
4. the set of topic scopes $\{\text{sid}\}$ to which the subscriber is attached.

When the broker delivers an envelope, the subscriber runs the three stages of Algorithm 2: (i) input verification (lines 1–6) to check checkpoint freshness, topic scope match, proof π verification; (ii) read-side key decapsulation and payload decryption (lines 7–10); (iii) post-decryption binding check (lines 11–12) to ensure that key k and message m are those bound by the proof π . We detail below each stage.

Algorithm 2 Receiving an envelope.

Require: *Caller-supplied:* envelope Env

Local state: read credential sk_s , Groth16 verification key vk_{gr} , EM verification key vk_{em} , sequence number $\text{seq}_{\text{local}}$

- 1: **if** $\neg \text{Verify}(\text{vk}_{em}, (\text{Env.sid}, \text{Env.chk.eid}, \text{Env.chk.accRoot}, \text{Env.chk.seq}), \text{Env.chk.}\sigma_{em})$ **then return** $\perp$
 - 2: **if** $\text{Env.chk.seq} < \text{seq}_{\text{local}}$ **then return** $\perp$
 - 3: **if** Env.sid not subscribed **or** $\text{Env.eid} \neq \text{Env.chk.eid}$ **then return** $\perp$
 - 4: $\text{pub} \leftarrow (\text{Env.d}, \text{Env.sid}, \text{Env.eid}, \text{Env.nym}, \text{Env.chk.accRoot})$
 - 5: **if** $\neg \text{Groth16.Verify}(\text{vk}_{gr}, \text{pub}, \text{Env.}\pi)$ **then return** $\perp$
 - 6: $\text{seq}_{\text{local}} \leftarrow \text{Env.chk.seq}$
 - 7: $k \leftarrow \text{ReadAuth.Decaps}(\text{sk}_s, \text{Env.ct}_{\text{auth}})$
 - 8: **if** $k = \perp$ **then return** $\perp$
 - 9: $m \leftarrow \text{Dec}(k, \text{Env.c})$
 - 10: **if** $m = \perp$ **then return** $\perp$
 - 11: **if** $\text{Poseidon}(k, m) \neq \text{Env.d}$ **then return** $\perp$
 - 12: **return** m
-

In stage (i), we check the freshness of the incoming envelope: we verify the EM's signature (line 1), computed over the envelope's scope Env.sid together with the checkpoint triple $(\text{eid}, \text{accRoot}, \text{seq})$, so a checkpoint issued for one scope cannot be accepted under

another; and then we reject checkpoints whose sequence number is older than the local one $\text{seq}_{\text{local}}$ (line 2). This last check enforces that any checkpoint chk older than $\text{seq}_{\text{local}}$ is rejected, because it has been superseded by a newer accumulator state. After freshness, we verify that the envelope is on a subscribed topic scope and that its top-level epoch matches the one bound in the signed checkpoint, $\text{Env.eid} = \text{Env.chk.eid}$ (line 3); this binds the epoch used in the proof π to the epoch the EM actually signed, so an envelope cannot pair a checkpoint from one epoch with a proof labeled for another. We then reconstruct the public input $\text{pub} = (d, \text{sid}, \text{eid}, \text{nym}, \text{accRoot})$ from envelope-visible fields (line 4). Then, we invoke Groth16.Verify (line 5) to attest that the publisher knew a witness w with $(\text{pub}, w) \in \mathcal{R}_{\text{PCP}}$ ((2)); a publisher whose wid is no longer in $\text{Tree}_{\text{sid}, \text{eid}}$ under the latest accRoot fails the check. Upon successful proof verification, we update $\text{seq}_{\text{local}}$ with the value from the envelope (line 6), ensuring that the sequence number advances only for validated envelopes.

In stage (ii), we recover the symmetric key k from the KEM using $\text{ReadAuth.Decaps}(\text{sk}_s, \text{ct}_{\text{auth}})$ if the subscriber satisfies the reading policy $\text{policy}_{\text{sid}, \text{eid}}^r$ (lines 7–8), and decrypt c using k , obtaining a candidate plaintext m (lines 9–10).

In stage (iii), we check that the retrieved plaintext m and key k have been bound in the proof π by recomputing $\text{Poseidon}(k, m)$ and comparing it to the envelope's d (line 11). On success we deliver m to the application, otherwise we discard it.

5.6. Soft and hard revocation

Authorization can change after enrollment: a publisher may lose the right to write on a topic scope, a subscriber the right to read from it. PCP provides two revocation procedures, one that withdraws the right to write, one that withdraws the right to read. Withdrawing a *write* right is cheap and takes effect immediately: it invalidates a single localized piece of state, namely one leaf in the EM's tree (*soft revocation*), and so excludes one client without affecting the others. Withdrawing a *read* right instead affects every client enrolled in the topic scope. This is because all read keys for a given topic scope are derived from a single ReadAuth master key of the KA, and the ReadAuth instantiations we consider provide no way to invalidate a single subscriber's key in isolation. So the only way to actually lock out a revoked reader is for the KA to *rotate* the master key, that is, to generate a fresh one. Rotation, however, invalidates *every* read key for the topic scope, so all surviving subscribers must re-enroll to be re-keyed under the new master key, and the authorized publishers must fetch the ReadAuth public key matching it. This is an expensive procedure in terms of message exchanges. To avoid incurring this cost on every revocation event, we perform it at the epoch boundary, so that rotating the master key and re-keying the surviving clients occurs once per epoch (*hard revocation*). Finally, note that every procedure operates within a single topic scope: withdrawing rights in a specific topic scope leaves the device enrolled and operational elsewhere. However, removing a device altogether is not a separate primitive but the same procedures applied across every topic scope in which it is enrolled.

Below, we present both procedures that are triggered by an operator action at the KA.

5.6.1. Soft revocation

Fig. 4(a) shows the protocol for evicting a write credential within a given epoch. In Step (1), an operator triggers a $\text{revoke}(\text{dev}, \text{sid})$ action to withdraw the write right of device dev for topic scope sid that reaches the KA. In Step (2), the KA computes the device's wid taking the corresponding cred from its credential ledger, prepares, and signs the eviction request $\sigma_{ka} \leftarrow \text{Sign}(\text{sk}_{ka}, (\text{sid}, \text{eid}, \text{wid}))$ for the EM. In Step (3), the KA sends the signed $\text{SoftRevoke}(\text{sid}, \text{eid}, \text{wid}, \sigma_{ka})$ request to the EM over their shared authenticated channel. In Step (4), the EM first verifies the request using the KA's public key: the authenticated channel only attests that the peer is the KA endpoint, whereas the signature attests that the KA authorized this specific eviction of wid from (sid, eid) . Then, the EM deletes wid from the tree $\text{Tree}_{\text{sid}, \text{eid}}$, recomputes the root $\text{accRoot}'$, advances the sequence number, and signs a fresh checkpoint chk' ; finally, it publishes chk' on its control topic (Step 5), so that the broker broadcasts it to the interested clients (Step 6).

The eviction takes effect at a subscriber only when it *consumes* chk' . Recall from Section 5.5 that a subscriber verifies each proof π against the accRoot carried in the envelope and rejects any checkpoint whose seq lies below its local $\text{seq}_{\text{local}}$. Once the subscriber consumes chk' and raises $\text{seq}_{\text{local}}$ to seq' , the revoked publisher can no longer get an envelope accepted, whatever checkpoint it attaches. A pre-revocation checkpoint now fails the freshness check since its sequence number has fallen behind ($\text{seq} < \text{seq}_{\text{local}}$). Using the current checkpoint chk' allows passing the freshness check, but then the proof π is verified against the new root $\text{accRoot}'$, where the publisher wid is no longer a member, so verification fails.

Finally, note that the revocation process is asynchronous and depends on checkpoint propagation latency: it may happen that at any instant some subscribers have adopted chk' while others have not (a transition window), so a revoked envelope may be rejected by the former yet accepted by the latter. This is common in any asynchronous distributed system, and what PCP aims for is *eventual consistency* of the revocation state: every subscriber eventually consumes chk' and converges to enforcing the eviction. Under an honest broker assumption, the time required to reach a consistent state depends only on broadcast latency since it is the sole propagation path; however, an adversarial broker can prolong this transition window by withholding chk' .

5.6.2. Hard revocation

Withdrawing a read right requires the KA to *rotate* the master key, forcing every non-revoked subscriber to be re-keyed. For this reason, rather than pay that cost on each revocation event, PCP batches them and performs a single key rotation at the epoch boundary, so a revoked device retains a valid decryption key until the end of the current epoch. The sequence diagram describing the revocation protocol is in Fig. 4(b). A cumulative request $\text{rotate}(\text{sid})$ for the topic scope sid to the KA (Step 1) is triggered. In response to this request, the KA generates fresh ReadAuth public and master keys (Step 2), and then asks the EM to advance the epoch (Step 3) for the topic scope sid . Upon receiving this request, the EM increments the epoch to $\text{eid}' = \text{eid} + 1$, updates the per-scope tree to the

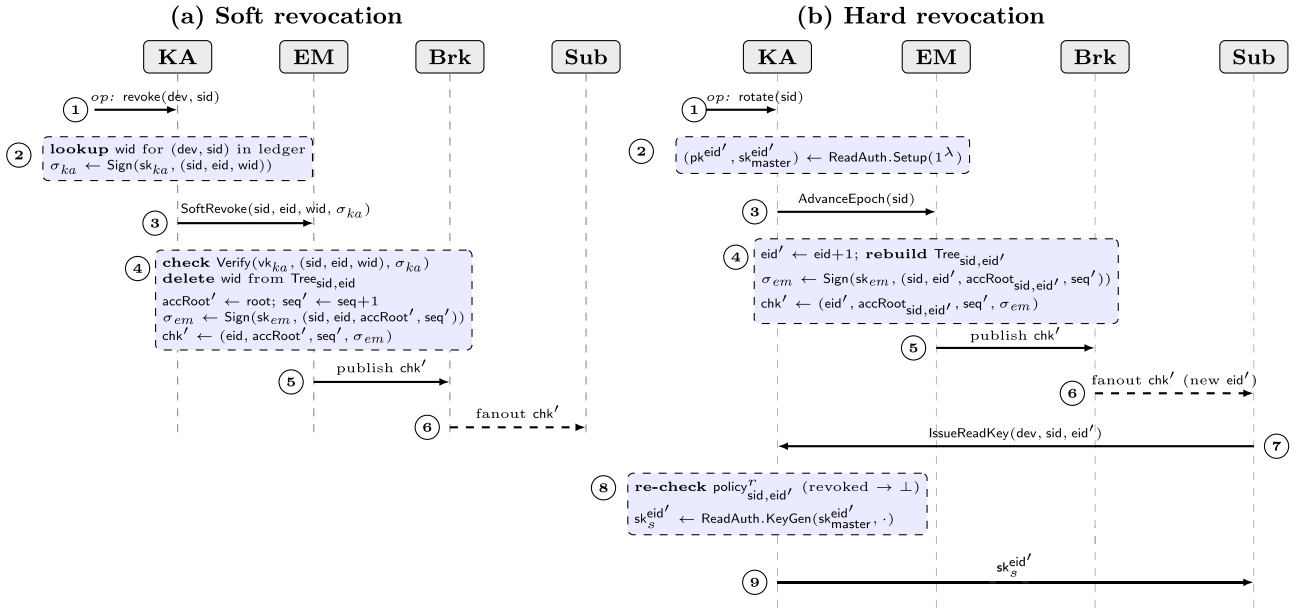


Fig. 4. Revocation as message sequence charts. *Soft revocation*: the KA resolves the device’s accumulator leaf wid from its credential ledger (2) and asks the EM to evict it (3); the EM signs a fresh checkpoint chk' (4) that the broker disseminates (5–6). *Hard revocation*: the KA rotates the $ReadAuth$ master key (2) and tells the EM to advance the epoch (3); the EM re-roots the tree to the new epoch and publishes a new checkpoint (4–6). A subscriber, upon receiving the new eid' in chk' , re-runs the enrollment (7–9); the KA re-checks policy' and a revoked device is denied a new key.

new epoch, obtaining a fresh root $accRoot_{sid,eid'}$, and prepares and signs checkpoint chk' , which it publishes through the control topic (Step 5). The broker in turn broadcasts chk' (Step 6). From eid' on, a revoked subscriber can no longer decrypt any payload; its previous keys open only pre-revocation ciphertexts, so it can no longer read anything published after the epoch transition. Until that boundary, though, the device still uses its valid key to keep reading messages: this lag, from the moment a read right is withdrawn to the next epoch, is bounded by the epoch duration.

Note that re-rooting the tree here only re-synchronizes it with the new epoch; the actual read revocation happens *asynchronously*, as each surviving subscriber re-enrolls into eid' to obtain a fresh read key while the revoked one is denied. More precisely, a subscriber learns the new epoch from chk' on the control topic and re-runs the subscriber-enrollment exchange for eid' (similar to Fig. 3): it sends a request (dev, sid, eid') , the KA re-evaluates the policy $policy'_{sid,eid'}$, and on success issues $sk_s^{eid'}$ under the new master key. A revoked device no longer satisfies the policy, so the KA denies the new key.

5.7. Checkpoint synchronization and late join

A checkpoint $chk = (eid, accRoot, seq, \sigma_{em})$ sent by EM represents a snapshot of the current authorization state within a topic scope, and is the means clients can use to synchronize themselves to the new state.

Both publishers and subscribers wait for checkpoints over the EM’s per-scope control topic and consume them internally, never delivering them to the application. Upon the arrival of a checkpoint, before updating any local state, the client verifies σ_{em} under vk_{em} and discards it in case of mismatch. The two client roles then proceed differently: a subscriber aligns its sequence number seq_{local} to that of the checkpoint $chk.seq$ and stores nothing else. This update closes the door to a revoked publisher without waiting for the next data envelope. When publishing, a publisher must prove that its credential representation wid is in the tree with a Merkle path matching the current root $accRoot$. When it consumes a control update that carries a new root, the stored path no longer matches, so the publisher fetches a fresh one from the KA along with the new $ReadAuth$ public key. A revoked publisher cannot: it is no longer in the tree, so the KA returns no path, and it can no longer build an accepted proof.

A *subscriber* joining mid-epoch is just the degenerate case of this synchronization: it fetches its scope’s chk_{latest} , adopts it as its initial seq_{local} , and from then on verifies envelopes against the same accumulator every other client uses, with no history to replay. A joining *publisher* needs no analogous step: enrollment already hands it the current checkpoint.

6. Security analysis in the symbolic model

We prove that PCP achieves the security goals of Section 4.2 with two complementary techniques, each operating under a different cryptographic model: the symbolic and the computational. This section presents our security analysis in the symbolic model with the Tamarin prover; Section 7 carries out the security analysis in the computational model by a game-based reduction argument.

Tamarin treats each cryptographic primitive as an idealized black box and reasons symbolically about all the sequences of messages a protocol together with a Dolev–Yao attacker could produce, which are in general infinitely many. It is suited to goals expressible as *safety* and *reachability* properties, e.g., that an envelope is accepted only if its sender was legitimately enrolled. We therefore use it to fully establish goal **G4** (write-authorization soundness) and goal **G6** (binding soundness); to establish the safety/reachability part of goal **G1** (payload confidentiality), goal **G2** (forward secrecy after revocation), and goal **G3** (backward secrecy after join); and to establish an invariant that goal **G7** (contained authority compromise) relies on: the EM performs no action without an explicit KA authorization. The remaining parts of these goals, together with goal **G5** (bilateral authorization privacy) are proved in [Section 7](#).

We first describe how each cryptographic primitive is abstracted ([Section 6.1](#)), the assumptions the model relies on ([Section 6.2](#)), a glimpse of Tamarin rules ([Section 6.3](#)), and the map from security goals to Tamarin lemmas ([Section 6.4](#)). The complete Tamarin model is available in the online repository [9].

6.1. Modeling the cryptographic primitives

PCP rests on the cryptographic building blocks of [Section 2](#), and in Tamarin we model them through an idealized symbolic counterpart.

Poseidon hashing. PCP uses the Poseidon hash to compute three digests with different roles: the accumulator leaf $wid = \text{Poseidon}(cred)$ for the Merkle tree, the binding digest $d = \text{Poseidon}(k, m)$ to link the Groth16 proof with the envelope payload, and the per-epoch pseudonym $nym = \text{Poseidon}(cred, eid)$ of publishers. In Tamarin, a hash function is rendered as a constructor with no rewrite equations, so distinct inputs can never produce the same output, and no operation can recover an input from its hash. To prevent the model from reporting spurious collisions across the three roles, e.g., a leaf colliding with a digest, we introduce three *separate* function symbols with different arity, one per role, mimicking the usages done in the previous sections: $h/1$, $hpair/2$, and $pym/2$.

Symmetric encryption. We use built-in symmetric-encryption theory ($senc, sdec$) to handle encrypted payloads: given a ciphertext $c = senc(m, k)$, only a holder of k can recover m with the function $sdec$.

Digital signatures. We use the signature theory ($sign, verify, pk$) to handle the two authority signatures σ_{ka} and σ_{em} . To distinguish a signature issued for one protocol from being used in another, we attach a tag (`'enroll'`, `'softrev'`, or `'epochadv'`) to each signed message indicating the relevant protocol.

ReadAuth key wrapping. The full ReadAuth interface of [Section 5](#) is outside the reach of the symbolic model, since attribute sets and policy satisfaction have no faithful representation in the equational theory used to define and reason about the cryptographic primitives. We therefore model a simpler concrete instance, the Epoch Group Key profile. Concretely, we treat ReadAuth as a public-key encryption scheme indexed by (topic scope, epoch): each topic scope is associated with a single secret key msk shared among all subscribers authorized to read that scope, while publishers wrap their per-message key k under the corresponding public key. Each epoch has its own key pair, which is regenerated when the epoch advances. Equivalently, our instance is the special case of the policy-based KEM of [Section 2.4](#) in which the policy and attribute machinery collapse to mere scope-and-epoch membership.

We capture this instance with three function symbols ($encaps, decaps, pp$) and one equation

$$decaps(msk, encaps(pp(msk), k)) = k$$

meaning that a key k wrapped under the public key $pp(msk)$ can be unwrapped only by a holder of the matching private key msk . During the enrollment, publishers obtain only $pp(msk)$, while subscribers obtain msk . Like $senc$ and $sign$, the wrap is idealized: the equation treats it as a perfect KEM.

Since no equation derives the private key msk from the public key $pp(msk)$, a publisher, holding only $pp(msk)$, can produce envelopes for subscribers but cannot decrypt them. A compromised publisher (threat **T4** of [Section 4](#)) can still forge envelopes, but cannot decrypt those addressed to subscribers.

Although our abstraction of ReadAuth is deliberately coarse, since any subscriber of a scope can read every message in it using the single shared private key msk , it still yields a sound confidentiality guarantee: any confidentiality result proved against an adversary without the private key carries over to the general case, where each subscriber instead holds its own key bound to its attributes.

Credential accumulator. In PCP, a publisher's right to write on a topic scope is recorded by storing its $wid = h(cred)$ in a per-epoch Merkle accumulator. So, a publisher proves its write authorization with a membership proof for its leaf against the published root by EM: as said in [Section 2.3](#), the collision resistance property of the underlying hash makes such a proof sound, so that no authentication path validates a leaf that was never inserted, and membership in the tree proves a legitimate enrollment.

For our analysis, only this membership matters. For this reason, we abstract the accumulator as a *set*: rather than representing the tree explicitly and checking paths, we record every leaf as the persistent fact `!Enrolled(sid, eid, wid)` and test membership against it. Note that here we represent a set by encoding it in the protocol state, so the limitation mentioned above about sets in the equational theory does not apply. The underlying idea is that the `!Enrolled` fact is produced only by the enrollment rule, after the EM verifies the KA's authorization, and no rule lets the adversary mint this fact. Thus, the fact `!Enrolled` symbolically represents when a publisher was genuinely enrolled.

Groth16 zero-knowledge proof. PCP uses a Groth16 proof π to let a publisher attest, without revealing its credential $cred$, that it is authorized. Such a proof π has two properties: *zero-knowledge*, meaning that the proof leaks nothing about its witness; and *knowledge soundness*, meaning that a valid proof can exist only if its producer knows a witness that satisfies the relation $\mathcal{R}_{PCP}$ of (2). We capture the two properties with different modeling techniques.

We represent a proof π as the term $zkp(crs, x, w, r)$, where crs abstractly stands for the public proving and verification keys, x is the public statement, w the secret witness, and r the proof randomness. The zero-knowledge property is captured by limiting what

one can obtain from this term. We introduce the destructor

$$\text{pubParams}(\text{zkp}(\text{crs}, x, w, r)) = x$$

which returns the public statement from π . Since no operation extracts the witness w , anyone (including the adversary) with a proof π learns nothing beyond its statement.

Modeling soundness is more involved because Tamarin's term algebra is free, so a term can always be constructed from its arguments: a proof term, in particular, can be assembled from *any* statement and witness. The constructor zkp alone therefore does not force w to satisfy the relation $\mathcal{R}_{\text{PCP}}$: nothing prevents a forged $\text{zkp}(\text{crs}, x, w, r)$ carrying a bogus w . We therefore enforce soundness using restrictions [18]: the publisher's rule emits an action ValidZKP in the trace that carries the proof term $\text{zkp}(\text{crs}, x, w, r)$, and a suitable restriction, called PCP_ZK_OK , keeps only the traces in which (x, w) genuinely satisfies $\mathcal{R}_{\text{PCP}}$. More precisely, the relation's three conjuncts are realized through the three Poseidon symbols introduced above, but in two different ways. The first two are algebraic equalities, checked directly by the restriction PCP_ZK_OK : pseudonym coherence $\text{nym} = \text{Poseidon}(\text{cred}, \text{eid})$ becomes $\text{nym} = \text{pnym}(\text{cred}, \text{eid})$, and ciphertext binding $d = \text{Poseidon}(k, m)$ becomes $d = \text{hpair}(k, m)$. The third conjunct is a membership statement, handled by the accumulator abstraction above: it requires the leaf $\text{wid} = h(\text{cred})$ to be enrolled, that is, the fact $\text{!Enrolled}(\text{sid}, \text{eid}, \text{wid})$ to hold in the current state. Because the envelope exposes only the pseudonym $\text{nym} = \text{pnym}(\text{cred}, \text{eid})$ and never wid , a further restriction nym_wid_same_cred ties the two through their shared credential cred , so an accepted proof's pseudonym must belong to an enrolled writer.

6.2. Modeling assumptions

Our Tamarin model explicitly captures the control plane and the clients of PCP, and implicitly represents the broker as the network. Moreover, making the model tractable required a few modeling decisions that we report here so as to clarify the extent of the symbolic analysis.

First, every rule that renders a protocol of Section 5, e.g., issuing or revoking a credential, producing or accepting an envelope, advancing an epoch, fires non-deterministically whenever its premises hold, without an explicit application-level event that would trigger it in practice. The model is therefore an *over-approximation* of the system, in that its traces form a superset of the real ones. This does not affect the soundness of our proofs: any attack in the implementation is also a run of the model, so ruling it out in the model rules it out in the implementation.

We introduce suitable setup rules Gen_CRS , KA_Setup , and EM_Setup to render setup procedures for the Groth16 parameters and the authority keys. These rules fire at most once and reveal their public material to the adversary, namely, the Groth16 reference string and the verification keys vk_{ka} , vk_{em} , and store the private material in dedicated persistent facts so they can be used later by other rules.

Since we represent the Merkle tree storing the credentials through a set of !Enrolled facts that are decoupled from the root a subscriber receives, we assume that the EM faithfully keeps the two in correspondence. Write-authorization soundness (goal **G4**) rests on this assumption, so the symbolic guarantee holds only relative to an honest EM. We include freshness constraints for checkpoints directly inside rules: a subscriber admits a new checkpoint only if the EM's signature is valid and its sequence number is larger than the highest sequence number it has already seen. In this way, replayed checkpoints are rejected by the subscriber rule.

Finally, we introduce suitable non-deterministic reveal rules to let the adversary obtain private cryptographic material or corrupt a principal. Specifically, we provide four rules to capture a compromised publisher and subscriber, covering threats **T4** and **T5** of Section 4. However, we do not provide the reveal rules for KA and EM corruption. For the KA case, the way we model ReadAuth lets us treat **T6** (leak of the KA read-side master key) as a special case of **T5**: in the Epoch Group Key instantiation, the KA's master key coincides with the private key msk each subscriber holds, so revealing it explicitly adds nothing. For the EM case, **T7** (leak of the EM signing key) concerns the integrity of the Merkle tree and checkpoints, and we analyze it in Section 7.4.

6.3. A glimpse of the model: publisher enrollment in Tamarin

We show how the publisher enrollment of Fig. 2 turns into Tamarin rules. These rules capture the essence of the protocol: Step (2), where the KA mints credentials; Step (3), about the handover between KA and EM that becomes a single message on the public network; Step (4), where the EM adds the new publisher into the tree.

More precisely, Step (2) translates to the following rule:

```
rule KA_IssueWriteCredential:
  let wid = h(~cred)
  m_en = <'enroll', $sid, $eid, wid>
  in
  [ !KAKey(~skKA)
    , Fr(~cred)
  ]
--[ KA_Issued($P, $sid, $eid, wid)
  , KA_AuthEnroll($P, $sid, $eid, wid)
]->
```

```
[ !Credential($P, $sid, $eid, ~cred)
  , Out(<m_en, sign(m_en, ~skKA)>)
]
```

The premises of the rule require the presence in the state of a persistent KA signing key $!KAKey(\sim skKA)$ and a freshly sampled credential $Fr(\sim cred)$. The `let` block computes the accumulator leaf $wid = h(\sim cred)$ and the signed enrollment message $m_{en} = \langle 'enroll', sid, eid, wid \rangle$: the constant tag `'enroll'` lets us distinguish these messages, preventing possible confusion with the soft-revocation and epoch-advance messages. The actions `KA_Issued` and `KA_AuthEnroll` are recorded on the trace, so they can be referred to by security lemmas. The publisher name $\$P$ appears free in the actions and on the right-hand side, so the rule mints a credential for any requesting publisher (the rule fires non-deterministically). The right-hand side produces two facts: a persistent $!Credential$ fact that retains the secret $\sim cred$ at the KA, and an `Out` that broadcasts m_{en} together with the signature $sign(m_{en}, \sim skKA)$ on the public network. Note that the check policy $check_{sid,eid}^w(attrs_w)$ of Fig. 2 is not modeled, according to our modeling assumption.

The KA's `Out` sends $\langle m_{en}, \sigma_{ka} \rangle$ on the network: this is how we render Step (3) of the figure. The EM can receive this message with a matching `In`. Step (4) is then captured by the following rule:

```
rule EM_Enroll:
  let m_en = <'enroll', $sid, $eid, wid>
  in
  [ In(<m_en, sig>)
    , !KAVk(pkKA)
  ]
  --[ Eq(verify(sig, m_en, pkKA), true)
    , EM_Enrolled($sid, $eid, wid)
    , OnlyOnce(<'emenroll', $sid, $eid, wid>)
  ]->
  [ !Enrolled($sid, $eid, wid) ]
```

The premise of the rule requires receiving a signed message from the network and the KA verifying key $!KAVk(pkKA)$ present in the state. The rule does not refer to the $!Credential$ fact, capturing that the only information EM has on the enrolling publisher is wid , read from the signed message. The action `Eq(verify(sig, m_en, pkKA), true)`, enforced by a suitable restriction, lets the rule fire only when σ_{ka} is a valid signature of KA. Inclusion into the Merkle accumulator $Tree_{sid,eid}$ is rendered as the single persistent fact $!Enrolled(\$sid, \$eid, wid)$ and the `OnlyOnce` allows us to ensure that each wid is enrolled at most once. The action `EM_Enrolled` is emitted and will be referred to in the security lemmas. Since the Merkle tree is modeled by $!Enrolled$ facts, the other tree-related actions of Step (4), e.g., inserting the leaf and producing its membership witness mp , are abstracted in that fact. The checkpoint generation (σ_{em}, chk) is modeled by a dedicated rule that fires non-deterministically and is not shown here.

The other protocols of Section 5 (publication, subscription, soft and hard revocation, and epoch advance) are rendered by suitable rules following the same pattern above.

6.4. Goal verification

We verify the security goals by automatically proving some *lemmas* over the model traces, where a trace is a sequence of action facts raised while the rewriting rules fire.

Table 2 maps the security goals of Section 4.2 to the lemmas that establish them, together with an informal statement of what each lemma proves. For goals **G1–G3**, the table covers only their safety and reachability facet; their indistinguishability facet is treated computationally in Section 7. For goal **G7**, the corresponding lemmas cover only the authority-separation invariant: the EM takes no control-plane action without a matching KA signature; whereas the rest is proved computationally.

The *correspondence* lemmas (for goals **G2–G4**, **G6**, **G7**) express safety properties: each lemma is an implication, universally quantified over all traces, of the form “an observable event implies an earlier authorization”. The premise concerns a subscriber acceptance, a message delivery, or an EM accumulator action; the conclusion concerns the KA issuance, KA authorization, read-key issuance, or honest emission that must precede it. As an example of a correspondence lemma, consider `enroll_requires_ka_auth` for goal **G7**:

```
lemma enroll_requires_ka_auth:
  "All sid eid wid #j. EM_Enrolled(sid, eid, wid) @ j ==> Ex P #i. KA_AuthEnroll(P, sid, eid, wid) @ i & i < j"
```

It states that whenever the EM admits a leaf wid at some step j in the trace (`EM_Enrolled`), an earlier step $i < j$ must carry a KA authorization for the same triple (sid, eid, wid) (`KA_AuthEnroll`). The other correspondence lemmas share this shape, differing only in the event and the authorization they relate. The `payload_secrecy` lemma for goal **G1** instead states that a given secret never enters adversary knowledge, provided the publisher- and subscriber-compromise rules are excluded.

```
lemma payload_secrecy:
  "All P sid eid nym d m seq #i.
   PubEmit(P, sid, eid, nym, d, m, seq) @ i
   & not (Ex #r. Reveal(P) @ r)
   & not (Ex S msk #r. RevealSubReadKey(S, sid, eid, msk) @ r)
  ==> not (Ex #k. K(m) @ k)"
```

Table 2

Mapping between security goals and Tamarin lemmas with their informal statements.

Goal	Lemma	What it states
G1	payload_secrecy	An envelope payload m never becomes part of the adversary's knowledge, unless the corresponding publisher or subscriber is compromised.
G2	post_hard_rev_forward_secrecy	A subscriber can decapsulate a key at epoch eid only if it has been issued a read key for that epoch; a subscriber revoked before epoch eid cannot.
G3	post_join_backward_secrecy	A subscriber can decapsulate a key at epoch eid only if it holds a read key for that same epoch; a subscriber that joined after eid cannot.
G4	acceptance_implies_enrollment	Every accepted envelope traces back to leaf in the Merkle tree storing a wid issued to a publisher that had not been revoked before acceptance.
G4	post_soft_rev_no_acceptance	After a publisher has been revoked, no subsequent envelope originating from it is accepted.
G6	binding_soundness	Any delivered (k, m) pair corresponds exactly to the pair bound by an honest publisher to the digest $d = \text{Poseidon}(k, m)$.
G7	enroll_requires_ka_auth	The EM admits a wid into the Merkle tree only if the signature σ_{ka} on $(\text{sid}, \text{eid}, \text{wid})$ correctly verifies.
G7	softrevoke_requires_ka_auth	The EM removes a publisher wid from the Merkle tree only if the request carries a signature σ_{ka} that correctly verifies.
G7	epochadvance_requires_ka_auth	The EM advances an epoch only if it receives a message correctly signed by KA.

The map from goals to lemmas is not one-to-one. A single lemma can discharge a goal on its own, as the secrecy lemma does for goal **G1**; conversely, one goal may need several lemmas, as does goal **G7**, which requires three correspondence lemmas to establish that the EM can neither admit a credential, evict one, nor advance an epoch without a matching KA signature. All of these lemmas are proved *fully automatically* by Tamarin [7], over an unbounded number of sessions and fresh values, terminating in seconds. The nine lemmas of the table, together with five auxiliary lemmas, amount to fourteen verified lemmas.

A safety lemma of the form “event $\Rightarrow$ earlier authorization” is vacuously true when its triggering event can never occur: a model in which no envelope is ever accepted would satisfy every correspondence lemma trivially. To rule this out, we prove two *witness* lemmas that exhibit the relevant traces: one in which an honest publisher produces an envelope that an honest subscriber accepts, and one in which a revoked writer is rejected while a legitimate one is still accepted. Searching for such traces over unbounded length does not terminate, so we restrict each witness to a single acceptance event. Since the restriction only removes traces, each witness remains a genuine run of the full model, and non-vacuity follows.

As an example of a witness lemma, consider the following:

```
lemma selective_rejection: exists-trace
  "Ex sid eid wid_rev wid_ok S nym d seq #r #a.
    EM_Revoked(sid, eid, wid_rev) @ r
    & SubAcceptEnv(S, sid, eid, nym, wid_ok, d, seq) @ a
    & not(wid_rev = wid_ok)"
```

The lemma states that there is a trace (exists-trace keyword) in which the EM evicts a leaf wid_rev at some step r (EM_Revoked), while at some step a a subscriber accepts an envelope originating from a distinct leaf wid_ok (SubAcceptEnv), so the model does discriminate between writers.

7. Security analysis in the computational model

The symbolic analysis of Section 6 establishes the safety and reachability guarantees of PCP, but leaves open the *quantitative* ones. Symbolic provers like Tamarin cannot provide a bound on the adversary's success probability. This section establishes those bounds through game-based reductions to standard cryptographic assumptions: we show that any efficient attacker who broke PCP could be turned into one breaking a well-studied primitive (Groth16, ReadAuth, AES-GCM, or Poseidon). Since these primitives are assumed computationally hard, the attacker's advantage over blind guessing is *negligible*. We discharge some goals the symbolic model left open: the computational aspects of payload confidentiality (goal **G1**), forward and backward secrecy (goals **G2** and **G3**); the bilateral authorization privacy (goal **G5**); and contained authority compromise (goal **G7**). Table 3 summarizes each result and the goal it discharges.

Below, we first fix the cryptographic primitives and the assumptions the reductions rely on, and the adversary advantage notation. We then state and link each result to the corresponding security goal.

7.1. Cryptographic assumptions and notation

Our security analysis rests on two distinct classes of assumptions: the *trust* assumptions **A1**–**A5** of Section 4, which define who the adversary is allowed to corrupt, and the cryptographic assumptions **C1**–**C4** below, which bound the success probability of any efficient adversary against the cryptographic primitives introduced in Section 2 and adopted by PCP. However, no proof depends on the internal structure of the used schemes, so we can replace them with a scheme enjoying the same properties without affecting any statement of this section, supporting the generality of our results. In addition, we assume that all algorithms below run in probabilistic polynomial time (PPT) in the security parameter λ .

Table 3
Computational results and the security goals they discharge.

Goal	Result	Assumptions	What it establishes
G1	Theorem 1	C1–C4	No PPT adversary distinguishes which message a challenge envelope protects, even after corrupting $\leq \iota$ devices and driving the revocation machinery.
G2	Corollary 2	C1–C4	Forward secrecy: confidentiality holds at epochs strictly after a hard revocation.
G3	Corollary 2	C1–C4	Backward secrecy: confidentiality holds at epochs strictly before a join.
G5 (i)	Theorem 4	—	An envelope rejection reveals nothing about whether a subscriber is authorized to read.
G5 (ii)	Theorem 4	C1, C2	A per-epoch writer pseudonym nym is unlinkable across epochs.
G7 (EM)	Theorem 6	C1–C4	EM compromise preserves confidentiality.
G7 (KA)	Theorem 6	C1, C2, C4	Partial KA compromise (threat T6) preserves proof soundness: the read master key yields no accepted envelope.

- C1** *Groth16 is zero-knowledge and weakly simulation-extractable* over BN254 [11]. This last property means that producing an accepted proof on a statement not previously simulated requires knowing a witness, even for an adversary that has observed other proofs [19]. Zero-knowledge supports establishing the confidentiality and writer-privacy results (Theorems 1 and 4), weak simulation-extractability supports establishing the KA part of goal G7.
- C2** *Poseidon is collision-resistant and pseudorandom when keyed with a secret input* [13]. Collision resistance [20, Def. 5.2] supports the proof of Theorem 4: no efficient adversary finds two distinct inputs with the same Poseidon output. Pseudorandomness [20, Def. 3.25] here means that the values derived from a credential (the per-epoch pseudonym $\text{nym} = \text{Poseidon}(\text{cred}, \text{eid})$ and the accumulator leaf $\text{wid} = \text{Poseidon}(\text{cred})$) are indistinguishable from independent random values, even when observed together, and that evaluating Poseidon on one input and on two behaves as two independent functions. This supports the proof of the unlinkability of writer pseudonyms across epochs (Theorem 4). The random-oracle model [20, §5.5], in which Poseidon is treated as an ideal hash function with perfectly random outputs, is used in the confidentiality proof of Theorem 1 and in that of Theorem 4. Note that the last property implies the other two; however, we report them separately because each result is proved under the weakest property that suffices for it.
- C3** *ReadAuth is IND-CPA-secure*. We assume that the abstract read-side interface `ReadAuth` is instantiated by a concrete scheme that is IND-CPA-secure [20, Def. 11.11]; examples include CP-ABE using FAME/AC17 [15] for attribute-rich policies, or an Epoch Group Key using AES key wrap [16] for membership-only ones. The IND-CPA assumption is directly used in the proof of Theorem 1.
- C4** *AES-GCM is IND-CPA and key-recovery hard, and EdDSA is existentially unforgeable under chosen-message attack (EUF-CMA)*. We instantiate the authenticated encryption interface of Section 2.5 with AES-GCM and assume, under a fresh-nonce assumption [21], that ciphertexts of equal-length messages under a uniform key are indistinguishable [20, Def. 3.22]: a ciphertext hides which message it carries and seeing a message next to its ciphertext does not reveal the key. Both support the proof of Theorem 1. The signature interface is instantiated with EdDSA, whose EUF-CMA security [20, Def. 12.2] is used for the authority signatures σ_{ka} and σ_{em} in the proof of Theorem 6.

Games, advantage, and bounds. We formalize our computational properties as *indistinguishability games* between two parties: a *challenger*, which plays the role of the honest system, and an *adversary*, which drives the game. The challenger secretly flips a bit b encoding the secret that the property at hand protects. For confidentiality, e.g., the adversary provides two messages, and the challenger encrypts them in the challenge envelope the one selected by b ; for unlinkability, b encodes which of the two publishers produced the challenge envelope. The adversary interacts with the system by issuing *queries*: it can ask the challenger to publish messages, corrupt devices, revoke credentials, or advance epochs. At the end of this interaction, the adversary outputs a guess for b . The property holds if no adversary can guess b noticeably better than random guessing. We introduce the *adversary advantage* to measure exactly this margin. For a PPT adversary $\mathcal{A}$ playing a game G , its advantage is

$$\text{Adv}^G(\mathcal{A}; \iota) = \left| \Pr[\mathcal{A} \text{ outputs } b] - \frac{1}{2} \right|,$$

where ι bounds the number of devices $\mathcal{A}$ may corrupt and $\Pr[\mathcal{A} \text{ outputs } b]$ is the probability that its guess matches the hidden bit. Since b is assumed to be generated by a fair coin, a blind guess is correct with probability $\frac{1}{2}$, so the adversary's advantage is the distance from this baseline. Formally, a property holds when the advantage of every PPT adversary is *negligible* (asymptotically smaller than the inverse of any polynomial in the security parameter λ).

We prove each theorem below by *reduction*, organized as a sequence of games [22]: we show that any adversary against PCP can be turned into an adversary against one of the primitives of assumptions C1–C4; therefore, the advantage against PCP is bounded by a sum of per-primitive terms, each denoting the best advantage a PPT adversary achieves against that primitive. Since a sum of negligible terms is negligible, the advantage against PCP is negligible as well. Below, we refer to the following bounds: ϵ_{ZK} for Groth16 zero-knowledge; ϵ_{PRF} for the keyed-PRF use of Poseidon; ϵ_{KEM} for the IND-CPA security of the `ReadAuth` instantiation; and ϵ_{AEAD} and ϵ_{KR} for the IND-CPA and key-recovery security of AES-GCM. Assumptions C1–C4 make each of these terms negligible. Each bound ϵ_X is a function of the security parameter λ , but, for the sake of notation, we write ϵ_X leaving the argument λ implicit. Moreover, a term can be counted more than once: in Theorem 1, e.g., the zero-knowledge loss is paid once per proof produced in the game, so ϵ_{ZK} carries a factor $q + 1$, where q is the number of PUBLISH queries issued by the adversary. Such counts are polynomial for a PPT adversary, so the products remain negligible.

7.2. Confidentiality under membership change

We introduce *Epoch-Adaptive Revocation Security* (EARS), an indistinguishability notion that formalizes payload confidentiality of goal **G1** under the full revocation lifecycle. EARS instantiates the classic indistinguishability techniques [23,24]; its novel element is the epoch constraint that folds PCP's two-tier revocation into the confidentiality game. Intuitively, it is a game between a challenger that runs an honest PCP deployment on a single topic scope sid^* and an adversary $\mathcal{A}$ that plays the untrusted broker together with up to t corrupted devices. Adversary $\mathcal{A}$ drives the deployment through the operational phases of Section 5 by calling oracles until it reaches a target epoch eid^* (the *challenge epoch*), then names a target publisher and tries to tell which of two messages of its choice a freshly produced envelope protects; confidentiality must therefore withstand adaptively chosen enrollments, corruptions and epoch transitions, not merely hold in a static snapshot. Each oracle implements one protocol flow:

- JOIN runs *client enrollment* (Section 5.3): the KA issues a write credential to a publisher or a read credential (a ReadAuth key) to a subscriber; for publishers, the EM also adds the corresponding wid into the accumulator.
- CORRUPT hands $\mathcal{A}$ a device's secret state (threats **T4** and **T5**) subject to the constraint that the number of corrupted devices is at most t .
- PUBLISH runs the *publish* phase (Section 5.4) on a chosen $(\text{sid}^*, \text{eid}, m)$, returning a complete envelope.
- SOFTREVOKE runs *soft revocation* (Section 5.6): it evicts a wid from the accumulator immediately.
- DEVICEREVOKE marks a subscriber for revocation, to take effect at the next epoch boundary.
- EPOCHTRANSITION runs *hard revocation* (Section 5.6): it advances eid and rotates the ReadAuth master key, re-keying the surviving readers.

At challenge time adversary $\mathcal{A}$ submits two equal-length messages m_0, m_1 and a target publisher dev^* ; the challenger returns an honest envelope published by dev^* using m_b for a secret bit b , and $\mathcal{A}$ must guess b . We require that no corrupted device holds a ReadAuth key valid at the challenge epoch eid^* , either because it was hard-revoked before that epoch or because it joined after it. Note that this constraint is critical because otherwise $\mathcal{A}$ would win trivially by decrypting the challenge envelope with the stolen keys. The full formalization of the game and of the oracles is in Appendix A.1. In the analysis, we model Poseidon as a random oracle accessible to $\mathcal{A}$ as the other oracle, and q_H below counts those queries.

Theorem 1. *Under assumptions C1–C4, for every PPT adversary $\mathcal{A}$ that corrupts at most t devices, makes q PUBLISH queries and q_H random-oracle queries, and runs for at most q_E epochs, we have*

$$\text{Adv}^{\text{EARS}}(\mathcal{A}; t) \leq (q + 1) \epsilon_{\text{ZK}} + (q_E + 1) \epsilon_{\text{KEM}} + q_H \epsilon_{\text{KR}} + q 2^{-\lambda} + \epsilon_{\text{AEAD}}$$

where $2^{-\lambda}$ denotes the probability that an honest publisher samples the challenge key.

Goals **G2** and **G3** follow from Theorem 1 as a corollary.

Corollary 2 (Forward and backward secrecy). *Under assumptions C1–C4:*

1. *Forward secrecy (goal G2): no PPT adversary that corrupts a device hard-revoked in epoch i can tell which of two messages an envelope of epoch $\text{eid}^* > i$ protects, beyond the advantage bound of Theorem 1.*
2. *Backward secrecy (goal G3): no PPT adversary that corrupts a device joining in epoch j can tell which of two messages an envelope of epoch $\text{eid}^* < j$ protects, beyond the same bound.*

7.3. Bilateral authorization privacy

Goal **G5** is *bilateral* authorization privacy: on the one hand, it protects subscribers against an adversary who wants to understand why an envelope was rejected; on the other hand, it protects publishers against an adversary who wants to track the envelopes they produce across epochs. We formalize goal **G5** with two indistinguishability games, one for each part of the goal (see Appendix A.1).

The first is the *subscriber-privacy game*: the adversary, possibly colluding with other subscribers, picks a target subscriber S and a valid envelope Env and must determine, from S 's observable behavior, whether S holds a valid ReadAuth key for Env , i.e., whether S is authorized to read the envelope. More precisely, the secret bit selects between two attribute sets that $\mathcal{A}$ itself submits, one satisfying the envelope's read policy and one not, and $\mathcal{A}$ is given the network transcript. Since Algorithm 2 runs locally the decision of delivering a message is *internal* to the subscriber, thus, the resulting decision is recorded by the network transcript. For this game, we have the following result:

Theorem 3 (Subscriber authorization privacy). *If delivery is internal to subscribers, every adversary $\mathcal{A}$, even computationally unbounded, has $\text{Adv}^{\text{ReadPriv}}(\mathcal{A}; t) = 0$.*

The theorem ensures that an envelope rejected before decryption leaks nothing on a subscriber holding or not a valid read key: an unauthorized subscriber is observationally indistinguishable from an authorized one. Theorem 4 is proved by a structural argument over Algorithm 2.

The second is the *publisher-linkability game*: the adversary interacts with the system through the oracles above and chooses two honest publishers $\text{dev}_0, \text{dev}_1$, holding the credentials $\text{cred}_0, \text{cred}_1$, whose envelopes it has observed in earlier epochs. The challenger produces a challenge envelope under cred_b at the challenge epoch eid^* for a secret bit b . Then, the adversary must recover b , i.e.

link the envelope back to its publisher across the epoch boundary. We require that the two publishers remain uncorrupted for the whole duration of the game and that the challenge envelope is the only one either of them publishes at the epoch eid^* : without this constraint, $\mathcal{A}$ would win by comparing the challenge pseudonym with one the same publisher had already produced at that epoch. For this game, we have the following result:

Theorem 4 (Writer unlinkability). *Under assumptions C1 and C2, for every PPT $\mathcal{A}$ with corruption bound t that makes q PUBLISH queries*

$$\text{Adv}^{\text{Link}}(\mathcal{A}; t) \leq (q + 1) \varepsilon_{\text{ZK}} + 2 \varepsilon_{\text{PRF}}.$$

The bound follows the way credentials enter the envelope. The proofs in the envelope hide cred by zero-knowledge, contributing one term for each proof generated by the two challenged publishers, hence at most $q + 1$ terms. The remaining credential-derived values, namely the pseudonym nym and the leaf in the Merkle tree wid , are pseudorandom, contributing one PRF term per publisher. Consequently, two honest publishers cannot be linked across an epoch boundary. Note that any scheme in which the verifier holds a stable key bound to the publisher would let the adversary link a writer across epochs, so reconciling writer unlinkability with efficient revocation requires zero-knowledge.

Finally, note that the unlinkability guarantee of Theorem 4 holds only when we consider the broker and colluding subscribers as adversaries. It does not hold in the case of a KA full compromise. As credential issuer, the KA mints each cred and maintains the $\text{dev} \rightarrow \text{cred}$ ledger. So if the attacker has access to this ledger, given an observed nym , it can recompute $\text{Poseidon}(\text{cred}, \text{eid})$ over the ledger returning the matching device.

7.4. Contained authority compromise

The control plane is split across two authorities with disjoint state: the EM holds the credential accumulator, the KA holds the ReadAuth master key and is the credential issuer. Goal G7 asserts that the impact of a single-authority compromise is bounded by this split: compromising the EM does not break confidentiality of envelopes, while compromising the ReadAuth master key of KA does not break proof soundness.

The following theorem establishes that the confidentiality of payloads is preserved with a corrupted EM:

Theorem 5 (EM compromise preserves confidentiality). *The bound of Theorem 1 holds even when the adversary is additionally given the EM's entire state.*

Intuitively, the theorem above holds because the EM has no access to ReadAuth material, and nothing in the bound of Theorem 1 depends on the secrecy of its state, so that bound survives the disclosure. However, an attacker controlling a leaked sk_{em} together with the accumulator set can insert a rogue leaf and sign a checkpoint over it: goal G4 (write-authorization soundness) is then not preserved.

The following theorem deals with a partial KA compromise where the attacker manages to obtain the master key for releasing read authorization, while the signing key sk_{ka} that authorizes accumulator admissions and the ledger storing the mapping between devices and issued credentials stays secure. We call an envelope a *forged accepted envelope* when an honest subscriber accepts it even though its public statement was not produced by an honest publisher. This excludes verbatim replays and re-randomizations of honest envelopes: they reuse an honest statement and carry the honest payload unchanged, thereby reproducing a legitimate message rather than forging one. Appendix A.1 formalizes this game, where $\mathcal{A}$ holds the read master key of every epoch and corrupts no publisher, and wins when an envelope whose statement no honest publisher produced passes the subscriber's input verification. Intuitively, the following theorem establishes that, although goal G1 on payload confidentiality is lost, proof soundness survives: every accepted envelope traces to a party that knew an enrolled credential at proof time, and no attacker makes an honest subscriber accept an envelope without *knowing* a witness.

Theorem 6 (Partial KA compromise preserves proof soundness). *Under assumptions C1, C2, and C4, an adversary holding the KA's read-side master key but neither its signing key sk_{ka} nor any enrolled credential cannot produce a forged accepted envelope, except with negligible probability.*

Finally, note that *full* KA compromise hands the adversary the issuance authority itself, letting it mint, enroll, and use credentials at will, so proof soundness is trivially lost. Additionally, simultaneous compromise of both authorities corresponds to full control-plane compromise: this breaks the security of PCP completely and is out of scope here.

8. Limitations of the security analysis

The security analyses in the symbolic and computational models show that PCP achieves its security goals. However, our results are not without limitations. Here, we explicitly state the boundaries and the extent of our security claims. More specifically, we first discuss the missing formal link between the two analyses, then the assumptions made but left unmodeled.

8.1. The symbolic-computational composition

The analyses of Sections 6 and 7 are carried out in two different models, and we keep them separate by design: each goal is discharged in the single model suited to its nature, so no stated goal requires a result bridging the two. Where the two analyses

address the same goal, they cover different aspects of it: for the secrecy goals **G1–G3**, Tamarin proves a trace property (no derivation of m , and no acceptance/decapsulation outside the issued read-key window, without an explicit reveal), while [Theorem 1](#) and its corollaries prove an indistinguishability bound under **C1–C4**.

The difference between the two models lies in how they treat the primitives. In the symbolic model, these primitives are assumed perfect, and their security holds by construction, e.g., Poseidon is a free function symbol, so collisions do not exist. The computational analysis instead makes those properties explicit, e.g., a statistical weakness of Poseidon is invisible to Tamarin but surfaces in the EARS game. Hence, attacks against the primitives are visible only to the computational analysis, while the exhaustive exploration of protocol behavior, e.g., message reordering, replay, and interleaving of sessions, is checked only by Tamarin. The two analyses do not subsume each other, so a flaw that is at once structural and contingent on a primitive weakness could in principle escape both; we regard this residual as small, since our computational games already consider the protocol's message flow rather than reasoning about primitives in isolation.

8.2. Assumptions left unmodeled

Both the Tamarin model and the game-based proofs rely on trust assumptions that are not made explicit in the formal statements. A further limitation concerns the scope of the computational bound. We detail them below.

Groth16 trusted setup. We assume that the setup of Groth16 is trusted. A subverted setup could compromise PCP because a malicious prover may forge proofs without holding a witness, defeating both goal **G4** and [Theorem 6](#). We did not formally analyze setup compromise in our analysis, leaving it as a deployment concern, in the sense that whoever deploys the protocol is responsible of ensuring that the setup is securely established. A similar observation also applies to the keypair generation for EM and KA that is assumed to be trusted.

Attribute oracle at the KA. In the Tamarin model, `KA_IssueWriteCredential` rule fires non-deterministically, without a policy check: we did not check that the KA issues credentials only when $\text{policy}_{\text{sid},\text{eid}}^w$ holds. A faulty or compromised attribute oracle and policy evaluation could make the KA issue credentials to devices that should not have them. This could invalidate goal **G4**. We will investigate this possibility in future work.

Credential-attribute provenance. We did not explicitly model an authority for the binding between devices and attributes. The introduction of an explicit third-party attribute authority distinct from the KA would change the authority-separation invariant established by the symbolic analysis, which would need to be reconsidered in the presence of the new authority.

Single-scope EARS. The bound of [Theorem 1](#) is for a single topic scope sid^* in which the number of corrupted devices is at most t . Capturing an adversary that reuses key material across scopes would require a multi-scope analysis, which we leave to future work. The single-scope bound is nonetheless representative: the secrets that protect confidentiality (the per-scope `ReadAuth` keys and the per-message keys) are bound to one (sid, eid) , and the scope binding of the EM signature (threat **T3**) prevents an envelope or checkpoint from one scope being accepted in another, so a multi-scope adversary decomposes into independent per-scope instances, each governed by [Theorem 1](#).

Implicit assumptions on the ReadAuth instantiation. Two confidentiality-relevant properties are left to the concrete `ReadAuth` instantiation rather than being established by our analysis. The first is *collusion resistance*: no coalition of subscribers whose attributes individually fail $\text{policy}_{\text{sid},\text{eid}}^r$ may recover the key by pooling their decryption material. Our results do not capture it, since the EARS game leaves no corrupted device holding a `ReadAuth` key at the challenge epoch, and the symbolic abstraction of [Section 6.1](#) collapses `ReadAuth` to a single shared key with no attribute granularity. The second is *constant-time decapsulation*, which would extend the reader privacy of [Theorem 4](#) to an adversary with a local timing side channel on a subscriber, beyond the network adversary of our threat model. We assume that both properties are provided by the actual implementation of `ReadAuth`. For example, using CP-ABE with FAME [15] implementation provides collusion resistance. A deployment facing only the network adversary of [Section 4.1](#) needs neither property beyond what [Theorems 1](#) and [4](#) already guarantee.

9. Related work

PCP combines several cryptographic mechanisms to implement a secure authorization layer on top of an untrusted broker, while preserving the confidentiality and integrity of the exchanged messages. Below, we position and compare PCP against three lines of research from the literature: confidentiality and integrity for MQTT pub/sub messaging; zero-knowledge constructions for authentication and authorization; cloud-native zero-trust microservices. Although there are some similarities between PCP and the papers analyzed below, to our knowledge, PCP is the only system that is at once broker-agnostic, independent of any blockchain, and backed by provable security guarantees.

MQTT Pub/Sub Security. Several research efforts have addressed the problem of securing the MQTT protocol, the de facto standard pub/sub messaging middleware for the IoT, against untrusted or honest-but-curious brokers. The closest proposal to ours is by Gupta et al. [4], who address the confidentiality problem by proposing a CP-ABE wrapper around an MQTT messaging system to encrypt the payloads of messages under attribute policies, so that the broker can forward the ciphertext without being able to read it. Their focus is only on payload confidentiality: they entirely neglect publish authorization, provide no proof that a publisher is entitled to write a topic, and do not consider revocation. Crucially, encrypting only the payload conceals nothing about who is publishing: the honest-but-curious broker still links every message to its originating client, so the scheme provides no writer unlinkability. In contrast, our work deals with both publish and read authorization and explicitly handles credential revocation. MQTT-I [5] is instead a modification of the MQTT protocol that addresses message integrity against a malicious broker, protecting communication from

message reordering and injection. It modifies the broker to maintain a per-publisher state and to serve Merkle paths on a dedicated topic. In contrast, PCP provides confidential, privacy-preserving authorization, and leaves the broker entirely unmodified, enforcing authorization client-side. DLST-MQTT [25] provides lightweight, end-to-end MQTT security with per-topic, dynamically negotiable security levels, but it targets confidentiality and integrity rather than authorization: its messages are not proof-carrying, and it offers neither authorization-private rejection nor writer unlinkability.

Additionally, the papers above do not support their proposals with a formal security analysis that spans both the symbolic and the computational models. By contrast, PCP establishes its guarantees as provable security: in the symbolic model through lemmas machine-checked in Tamarin, and in the computational model through game-based reductions.

Zero-Knowledge Authentication and Authorization. Some proposals build authentication and authorization mechanisms upon zero-knowledge proofs and Merkle accumulators. zk-Guard [26] is the closest proposal in terms of authorization: it uses a zero-knowledge circuit to prove that a requester's attributes satisfy an ABAC access policy without revealing those attribute values; the proof is verified on-chain by a Hyperledger Fabric smart contract, which returns the access decision without a trusted verifier. It relies on the blockchain and binds each user to a stable on-chain identifier: although it hides attribute values, the requester stays linkable across requests, so it offers no writer unlinkability. In contrast to ours, their dynamic revocation mechanism is neither epoch-structured nor accompanied by a formal revocation security model. Similarly, DPS-IIoT [27] pairs a non-interactive zero-knowledge proof with CP-ABE for pub/sub access control in IIoT, and deploys its access policies on a Hyperledger Fabric ledger. In contrast to PCP, DPS-IIoT does not assume a broker, anchors its trust in a blockchain, and provides no formal guarantees about its security goals.

Cloud-Native Zero-Trust Microservices. Several proposals have been put forward to support zero-trust in microservice architectures. Service meshes such as Istio [28] realize zero-trust networking [2] for synchronous, service-to-service traffic, relying on mutual TLS to secure the channel and on OPA [29] to evaluate request authorization. However, their security model does not treat the message broker as an adversary. In contrast, PCP treats the broker as untrusted and aims to secure the broker-mediated message flows. In these architectures, microservice authorization is externalized to a trusted policy decision point and enforced by the infrastructure [30,31]; even decentralized variants keep the policy enforcement point trusted [31]. In contrast, PCP makes every message self-authorizing through a proof verified client-side, so no policy enforcement point needs to be trusted. As a result, PCP offers properties that these deployments typically lack: writer unlinkability and authorization-private rejection.

10. Conclusion

We have presented PCP (Proof-Carrying Payloads), an end-to-end authorization overlay for pub/sub messaging systems that moves both data protection and authorization from the broker to the clients. In PCP, messages are carried inside protected envelopes that combine encryption with succinct authorization proofs: subscribers verify that a publisher was authorized to write to the target topic before attempting decryption, while read authorization is enforced cryptographically. As a result, PCP removes the broker as a centralized policy enforcement point while remaining broker-agnostic and deployable as a wrapper around existing systems. We have also presented the design of the full authorization lifecycle: enrollment protocols for publishers and subscribers, end-to-end authorization for message publication and delivery, and two complementary revocation mechanisms for publishers and subscribers. We have formalized the adversary and the security goals and analyzed them using two complementary methods. Protocol-level guarantees are verified symbolically in Tamarin, while guarantees that depend on the hardness of cryptographic primitives are established via game-based reductions. Together, these results show that proof-carrying, encrypted payloads provide a way to retrofit zero-trust authorization onto existing pub/sub infrastructures without modifying the broker or the underlying communication protocol.

Several interesting directions for future work remain open. A full implementation and empirical evaluations of the proposal in different settings, such as IoT and cloud-native microservices, would quantify the overhead our authorization overlay imposes on clients. On the protocol side, we would like to extend PCP to manage a hash-chained, transparency-log checkpoint, enabling subscribers to detect EM equivocation without trusting it for correctness. A blockchain platform is one way to implement such a log: it would give ordering and tamper resistance for both checkpoints and the accumulator root, at the price of consensus latency and of a dependency on an infrastructure that pub/sub deployments typically lack. Whether those guarantees outweigh these costs in a PCP deployment is a question worth investigating. Additionally, we plan to strengthen the analysis of the multi-scope adaptive EARS game, and mechanize the game-based proof using either an automatic prover or proof assistant.

Dedication

This paper is dedicated to *Gian-Luigi Ferrari* on the occasion of his Festschrift: as my PhD advisor, Gangi taught me that formal methods play a key role in the design of robust and secure systems, offering principled solutions to concrete problems. This work follows his teaching and addresses a practical problem in distributed systems, a field to which Gangi made significant contributions. At the same time, it builds on his teaching from my own perspective, by combining formal-methods approaches, such as proofs in the symbolic model, with techniques closer to cryptographic practice, such as game-hopping proofs in the computational model.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank the anonymous reviewers for their comments and suggestions, which helped improve the paper. This work has been partially supported by RDS PTR 25-27 CYBER 2.1 “Progetto Cybersecurity” WP3 LA 3.21 - CUP: D63C24001060001.

Data availability

No data was used for the research described in the article.

Appendix A. Security proofs in the computational model

Here, we define the games underlying our security proof, provide an overview of the proof strategy, and then we prove our security results of Section 7.

A.1. Oracles and games

Following [22], each game X is defined as a procedure, denoted with $\text{Exp}^X(\lambda, t)$, and parameterized by the security parameter λ and by the number t of devices the adversary may corrupt. The adversary is represented as a procedure $\mathcal{A}$ that the challenger invokes. More precisely, in an indistinguishability game we perform two invocations of $\mathcal{A}$: in the first invocation, the adversary takes the public parameters of PCP as input and uses the oracle to drive the deployment of PCP to a desired state; then, it commits to the challenge input (the *find* phase); in the second, it receives the challenge and outputs its guessed bit b' (the *guess* phase). The constraints on the adversary’s behavior are captured by an admissibility predicate Adm that is evaluated over the *whole* execution trace of the game, so an adversary that violates it before and after the challenge time loses the game. The game procedure returns 1 when the adversary wins, that is, when it guesses the secret bit and its behavior is admissible. Accordingly, the advantage of an admissible $\mathcal{A}$ in such a game is $\text{Adv}^X(\mathcal{A}; t) = \left| \Pr[\text{Exp}^X(\lambda, t) = 1] - \frac{1}{2} \right|$, while we set the advantage of an inadmissible one to 0, so that violating Adm never pays off. The game of Fig. A.5 departs from this schema: it invokes $\mathcal{A}$ once, has no secret bit, and its measure is the probability $\Pr[\text{Exp}^{\text{Forge}}(\lambda, t) = 1]$.

As notation, in the game and oracle pseudo-code, we write $x \xleftarrow{\$} \mathcal{X}$ for a uniform sample from a set. We assume that the adversary keeps its state across the challenge, which we leave implicit, and keeps its oracle access afterwards. We assume a procedure $\text{Setup}(1^\lambda, \text{sid}^*)$ that takes the security parameter and target topic scope and returns the public parameters of PCP denoted as pp . These include the Groth16 reference string, the KA and EM verification keys.

<pre> JOIN(role, dev, attrs): if role = pub then (cred, wid, σ_{ka}) $\leftarrow$ KA.Enroll(sid*, eid, dev) EM.Register(sid*, eid, wid, σ_{ka}) st[dev] $\leftarrow$ cred else st[dev] $\leftarrow$ KA.IssueRead(eid, attrs) RK[dev] $\leftarrow$ RK[dev] $\cup$ {eid} end if CORRUPT(dev): $\mathcal{C} \leftarrow \mathcal{C} \cup \{\text{dev}\}$; return st[dev] PUBLISH(dev, m): return Env built by Algorithm 1 from (st[dev], sid*, eid, policy$_{\text{sid}^*, \text{eid}}^r$, m) SOFTREVOKE(dev): EM.Evict(sid*, eid, wid$_{\text{dev}}$); return chk </pre>	<pre> DEVICEREVOKE(dev): $\mathcal{D} \leftarrow \mathcal{D} \cup \{\text{dev}\}$ EPOCHTRANSITION(): eid $\leftarrow$ eid + 1 rotate the ReadAuth master key for all subscribers dev $\notin \mathcal{D}$ do st[dev] $\leftarrow$ KA.IssueRead(eid, attrs$_{\text{dev}}$) RK[dev] $\leftarrow$ RK[dev] $\cup$ {eid} end for return chk H(x): if $T[x] = \perp$ then $T[x] \xleftarrow{\\$} \mathbb{F}_p$ end if return $T[x]$ </pre>
--	--

Fig. A.1. The oracles the adversary uses to interact with an honest deployment of PCP in a specific topic scope sid^* .

Oracles and challenger state. Fig. A.1 shows the pseudo-code of the oracles informally introduced in Section 7.2. Each oracle runs one protocol flow and updates the challenger state, which persists across calls. The challenger state includes the current epoch eid ; the set $\mathcal{C}$ of corrupted devices; the set $\mathcal{D}$ of subscribers marked for hard revocation; the per-device secret state $\text{st}[\cdot]$; a table $\text{RK}[\cdot]$ recording, for each subscriber, the epochs for which it has been issued a *ReadAuth* key; and a table $T[\cdot]$ of random-oracle answers. We assume that all state values are implicitly initialized by the challenger at the beginning of each game: sets and tables to empty, the current epoch to 0. The challenger also fixes, for each epoch, the scope’s read policy $\text{policy}_{\text{sid}^*, \text{eid}}^r$ under which every *PUBLISH* call encapsulates the message key. The oracle *JOIN* enrolls a device: for a publisher, *KA.Enroll* runs the enrollment protocol of Section 5 and *EM.Register* verifies σ_{ka} and then inserts wid into the accumulator; for a subscriber, *KA.IssueRead* issues a *ReadAuth* key for the

$\text{Exp}^{\text{EARS}}(\lambda, t):$ 1: $pp \leftarrow \text{Setup}(1^\lambda, \text{sid}^*)$ 2: $(\text{dev}^*, m_0, m_1) \leftarrow \mathcal{A}(pp)$ 3: $\text{eid}^* \leftarrow \text{eid}; b \xleftarrow{\\$} \{0, 1\}$ 4: $\text{Env}^* \leftarrow \text{PUBLISH}(\text{dev}^*, m_b)$ 5: $b' \leftarrow \mathcal{A}(\text{Env}^*)$ 6: return $b' = b \wedge \text{Adm}$	Adm: $ m_0 = m_1 \wedge$ $\text{dev}^* \text{ is enrolled as a publisher } \wedge$ $ C \leq t \wedge$ $\forall \text{dev} \in C : \text{eid}^* \notin \text{RK}[\text{dev}]$
---	---

Fig. A.2. The EARS game (left) with its admissibility predicate (right).

$\text{Exp}^{\text{ReadPriv}}(\lambda, t):$ 1: $pp \leftarrow \text{Setup}(1^\lambda, \text{sid}^*)$ 2: $(S, \text{Env}, \text{attrs}_0, \text{attrs}_1) \leftarrow \mathcal{A}(pp)$ 3: $b \xleftarrow{\\$} \{0, 1\}$ 4: $\text{st}[S] \leftarrow \text{KA.IssueRead}(\text{eid}, \text{attrs}_b)$ 5: deliver Env to S, which runs Algorithm 2 locally 6: Let view be the record of everything that occurred in the network 7: $b' \leftarrow \mathcal{A}(\text{view})$ 8: return $b' = b \wedge \text{Adm}$	Adm: $ C \leq t \wedge$ $S \notin C \wedge$ $\text{Env} \text{ is well formed for } \text{sid}^* \wedge$ $\text{attrs}_0 \text{ satisfies the read policy of}$ $\text{Env}, \text{attrs}_1 \text{ does not}$
---	--

Fig. A.3. The subscriber authorization privacy game (left) with its admissibility predicate (right).

current epoch. CORRUPT hands $\mathcal{A}$ the secret state of a device and records it in C ; it is the only way for $\mathcal{A}$ to obtain a credential or a ReadAuth key of its own. PUBLISH returns an envelope built by Algorithm 1 on behalf of an honest publisher; since it reads $\text{st}[\text{dev}]$, it is defined only for devices previously enrolled as publishers through JOIN. SOFTREVOKE runs the soft revocation of Section 5 through EM.Evict , which removes wid from the accumulator and returns the resulting checkpoint. DEVICEREVOKE marks a subscriber for hard revocation, which takes effect only at the next epoch boundary: the device stays in D and is skipped when read keys are reissued. EPOCHTRANSITION oracle advances eid , rotates the ReadAuth master key, reissues a read key to every subscriber not in D while updating $\text{RK}[\cdot]$, and returns the new checkpoint. The last oracle H is a standard random oracle that is used in the proofs to model Poseidon: its argument x ranges over the tuples Poseidon is evaluated on, and its answers are sampled one point at a time and kept in $T[\cdot]$. The reductions for Theorems 1 and 6 appeal to H , whereas the proof of Theorem 4 only uses the pseudorandomness of Poseidon.

The Exp^{EARS} game. Fig. A.2 shows the pseudo-code of the EARS game. At line 1, we generate the public parameters. We then invoke adversary $\mathcal{A}$ at line 2: the adversary uses the oracles to drive PCP into the state it wants by enrolling publishers and subscribers with JOIN, obtaining envelopes with PUBLISH, corrupting devices, revoking credentials, advancing epochs; at the end, it commits to a (possibly corrupted) target publisher dev^* and to two messages m_0, m_1 . At line 3, we set the challenge epoch eid^* to the epoch reached by $\mathcal{A}$ through its invocations of the EPOCHTRANSITION oracle, and we randomly select the secret bit b uniformly. At line 4, the target publisher produces the challenge envelope Env^* honestly. At line 5, we invoke $\mathcal{A}$ on the challenge envelope; it may keep corrupting and revoking after seeing Env^* , but at the end it produces a guess b' . At line 6, we check whether the adversary's guess was correct and the execution trace admissible; if both hold, the adversary wins the game. The admissibility predicate Adm consists of four clauses: $|m_0| = |m_1|$ requires the two messages to have the same length; the second requires that the dev^* have been enrolled as a publisher through JOIN; $|C| \leq t$ bounds the number of corrupted devices by the corruption budget t ; and $\forall \text{dev} \in C : \text{eid}^* \notin \text{RK}[\text{dev}]$ requires that no corrupted device was ever issued a ReadAuth key for the challenge epoch.

The $\text{Exp}^{\text{ReadPriv}}$ game. Fig. A.3 shows the pseudo-code of the subscriber-privacy game, which follows the same schema as the Exp^{EARS} game. At line 2, the adversary $\mathcal{A}$ exploits the oracles to drive PCP to a desired state and commits to a target subscriber S , to an envelope Env , and to two attribute sets $\text{attrs}_0, \text{attrs}_1$. Note that it selects the subscriber *before* it is issued a read key and that attrs_0 satisfies the policy of Env , whereas attrs_1 does not. At line 3, we sample the secret bit b , and, at line 4, we issue S a read key for attrs_b and store its private state in the table $\text{st}[\cdot]$. Note that since Adm forces attrs_0 to satisfy the read policy of Env and attrs_1 not to, the bit selects exactly whether S is authorized for the envelope Env . At line 5, the envelope is delivered to S , which runs Algorithm 2 locally. At line 6, we let the variable view be the log of everything that occurred in the network during the execution. Note that view does *not* contain the result of Algorithm 2 since delivery is internal to the subscriber. At line 7, we hand this trace to adversary $\mathcal{A}$: from view , $\mathcal{A}$ must decide which of the two attribute sets, attrs_0 or attrs_1 was used at line 4, that is, whether S is authorized to read Env . At line 8, we return 1 if the adversary wins the game. The admissibility predicate Adm imposes the constraint on the two attribute sets mentioned above; bounds the corrupted devices by t ; requires the target subscriber honest, since corrupting S would hand $\mathcal{A}$ the read key itself; and requires Env to be well formed for the topic scope sid^* .

The Exp^{Link} game. Fig. A.4 shows the pseudo-code of the publisher-linkability game. It follows the same schema as Exp^{EARS} , except that the secret bit selects the publisher that produces the challenge envelope rather than the message the envelope protects. At line 2, the adversary $\mathcal{A}$ after interacting with the oracles commits to two honest publishers $\text{dev}_0, \text{dev}_1$, whose credentials we write $\text{cred}_0 = \text{st}[\text{dev}_0]$ and $\text{cred}_1 = \text{st}[\text{dev}_1]$. Note that $\mathcal{A}$ commits to the two devices and not to their credentials, which it never sees: both

$\text{Exp}^{\text{Link}}(\lambda, t):$ 1: $pp \leftarrow \text{Setup}(1^\lambda, \text{sid}^*)$ 2: $(\text{dev}_0, \text{dev}_1) \leftarrow \mathcal{A}(pp)$ 3: $\text{eid}^* \leftarrow \text{eid}; b \xleftarrow{\\$} \{0, 1\}; m \xleftarrow{\\$} \{0, 1\}^\ell$ 4: $\text{Env}^* \leftarrow \text{PUBLISH}(\text{dev}_b, m)$ 5: $b' \leftarrow \mathcal{A}(\text{Env}^*)$ 6: return $b' = b \wedge \text{Adm}$	Adm: $ C \leq t \wedge$ $\text{dev}_0, \text{dev}_1 \notin \mathcal{C} \text{ at every point}$ $\text{of the run} \wedge$ $\text{both are enrolled in } \text{sid}^* \text{ and not}$ $\text{soft-revoked at } \text{eid}^* \wedge$ $\text{the challenge is the only PUBLISH}$ $\text{query on } \text{dev}_0 \text{ or } \text{dev}_1 \text{ at } \text{eid}^*$
---	--

Fig. A.4. The writer unlinkability game (left), with its admissibility predicate (right).

$\text{Exp}^{\text{Forge}}(\lambda, t):$ 1: $pp \leftarrow \text{Setup}(1^\lambda, \text{sid}^*)$ 2: $\text{Env} \leftarrow \mathcal{A}(pp, \text{sk}_{\text{master}}^{\text{eid}})$ 3: return Env passes lines 1–6 of Algorithm 2 $\wedge \text{Adm}$ EPOCHTRANSITION also returns the fresh $\text{sk}_{\text{master}}^{\text{eid}}$ to $\mathcal{A}$	Adm: $ C \leq t \wedge$ $\text{no } \text{dev} \in \mathcal{C} \text{ is enrolled as a}$ $\text{publisher} \wedge$ $\text{the statement of } \text{Env} \text{ was not}$ $\text{produced by any PUBLISH query}$
--	---

Fig. A.5. The write-authorization soundness game under partial KA compromise (left), with its admissibility predicate (right).

publishers are honest, and the oracles of Fig. A.1 are all indexed by device. At line 3, we set the challenge epoch, and besides the secret bit we uniformly sample a challenge message m from $\{0, 1\}^\ell$, for an arbitrary fixed length ℓ . Since the secret bit selects a publisher and not a message, the payload is drawn independently of b and carries no information about it. At line 4, we produce the challenge envelope under the credential of dev_b , and at line 5 we pass it to $\mathcal{A}$ and obtain its guess on the device that produced the envelope. At line 6, we check if the adversary wins the game. The admissibility predicate Adm bounds the number of corrupted devices ($|C| \leq t$); requires that both publishers $\text{dev}_0, \text{dev}_1$ are honest for the duration of the game, and that are enrolled in sid^* and not soft-revoked at eid^* ; and that the challenge envelope must be the only published message of dev_0 or dev_1 at eid^* . This last condition rules out the trivial win of comparing the challenge pseudonym with one the same publisher produced in the same epoch, nym being constant within an epoch by construction.

The $\text{Exp}^{\text{Forge}}$ game. Fig. A.5 shows the pseudo-code of the write-authorization soundness game underlying Theorem 6. Unlike the previous three, it is not an indistinguishability game and thus flips no secret bit. In addition, it models the partial compromise of the KA: the adversary holds the KA's read-side master key but has access to neither its signing key sk_{ka} nor any enrolled credential. To capture this compromise, at line 2 the adversary $\mathcal{A}$ is given the ReadAuth master key $\text{sk}_{\text{master}}^{\text{eid}}$ of the current epoch. When it drives the deployment through the oracles of Fig. A.1, EPOCHTRANSITION returns the fresh master key of each new epoch, so $\mathcal{A}$ decapsulates every message key of every epoch. By contrast, no oracle ever discloses the signing key sk_{ka} used by the KA to authorize accumulator admissions, so $\mathcal{A}$ can have no wid of its choice admitted. Line 2 ends with $\mathcal{A}$ outputting a candidate envelope Env ; it wins the game if, at line 3, that envelope passes the input-verification stage of Algorithm 2, i.e., its lines 1–6, and the run is admissible. What matters here is only that the envelope passes this first check, nothing else. Consistently, besides the corruption budget $|C| \leq t$, Adm forbids corrupting a publisher, since CORRUPT would return its credential, whereas we assume that the adversary holds none, and requires the envelope to carry a statement that no PUBLISH query produced.

A.2. Overview of the proof strategy

Both Theorems 1 and 4 are proved by *game hopping* [8,22]. Starting from the real security game (Game 0), we transform it through a finite sequence of intermediate games (Games i) into a final game (Game N) whose challenge envelope is independent of the secret bit b . We write S_i for the event that the adversary guesses b correctly in Game i , so its adversary advantage is $|\Pr[S_i] - \frac{1}{2}|$. We then bound the adversary's advantage in terms of the changes in its advantage across the sequence of intermediate games:

$$\left| \Pr[S_0] - \frac{1}{2} \right| = \left| \sum_{i=0}^{N-1} (\Pr[S_i] - \Pr[S_{i+1}]) \right| \leq \sum_{i=0}^{N-1} |\Pr[S_{i+1}] - \Pr[S_i]|.$$

The equality derives from telescoping consecutive games (the final game has $\Pr[S_N] = \frac{1}{2}$); the inequality follows from the triangle inequality. Each proof fixes its own number of games: the proof of Theorem 1 uses five games, that of Theorem 4 four. In the proof below, we bound each step distance $|\Pr[S_{i+1}] - \Pr[S_i]|$, depending on the transformation step applied to pass from a game to the next one.

We perform two kinds of step. In an *indistinguishability step* we rewrite a single envelope component, e.g., a Groth16 proof, an encapsulated key, or an AEAD body, and bound its distance by one assumption of Section 7 through an explicit reduction: an adversary that tells the two games apart is turned into one that breaks that assumption. In a *failure-event step* we instead make the two games behave identically unless a designated event F occurs; by the difference lemma [8] their distance is then at most $\Pr[F]$. Throughout the proofs, ϵ_{ZK} , ϵ_{KEM} , ϵ_{KR} , ϵ_{AEAD} , and ϵ_{PRF} denote the advantage terms fixed in Section 7, while the proof of Theorem 6 introduces at first use the further terms it needs. Recall that each bound ϵ_X is a negligible function of the security parameter λ .

A.3. Security proofs

Theorem 1. *Under assumptions C1 – C4, for every PPT adversary $\mathcal{A}$ that corrupts at most t devices, makes q Publish queries and q_H random-oracle queries, and runs for at most q_E epochs, we have*

$$\text{Adv}^{\text{EARS}}(\mathcal{A}; t) \leq (q + 1)\epsilon_{\text{ZK}} + (q_E + 1)\epsilon_{\text{KEM}} + q_H \epsilon_{\text{KR}} + q 2^{-\lambda} + \epsilon_{\text{AEAD}}$$

where $2^{-\lambda}$ denotes the probability that an honest publisher samples the challenge key.

Proof. The proof proceeds through five games. The idea is that each game removes the dependence of one field of the envelope produced by the challenger on the secret bit b . The order of the games is determined by these dependencies. More specifically, the Groth16 proofs are simulated first (Game 1), since the later games falsify the challenge statement and only a simulated proof, needing no witness, can then be produced. The wrapped key (Game 2), the binding digest (Game 3), and the AEAD body (Game 4) are in this order, because each hop can randomize its field only once the previous one has detached k^* from the others.

Game 0 (Real EARS). This is the game $\text{Exp}_{\mathcal{A}}^{\text{EARS}}(\lambda, t)$ of Fig. A.2: the challenger runs an honest deployment of the topic scope sid^* , answers the oracle queries of $\mathcal{A}$, and returns a challenge envelope Env^* built honestly on $(\text{sid}^*, \text{eid}^*, m_b)$ for a secret bit b . Since the run is admissible, no corrupted device holds a ReadAuth key for the challenge epoch eid^* . By definition, $\text{Adv}^{\text{EARS}}(\mathcal{A}; t) = |\Pr[S_0] - \frac{1}{2}|$.

Game 1 (Simulated proofs). We perform an indistinguishability step from Game 0. We replace the $q + 1$ Groth16 proofs produced in the game (q PUBLISH-oracle proofs and the challenge proof) one at a time with the output of a simulator $\tilde{\pi} \leftarrow S_{\text{ZK}}(td, x)$, where td is the simulation trapdoor associated with the public material of Groth16 on same public statement x . In the challenger's PUBLISH oracle, this amounts to rewriting in every call line 7 of Algorithm 1 as follows:

$$\pi \leftarrow \text{Groth16.Prove}(\text{pk}_{gr}, \text{pub}, w) \mapsto \tilde{\pi} \leftarrow S_{\text{ZK}}(td, \text{pub}).$$

All the other fields of the envelope (nym , ct_{auth} , c , d , chk , σ_{em}) are produced identically. From Groth16 properties [11], we have that each simulated proof is computationally indistinguishable from an honest proof generated by someone who actually knows the secret data. Therefore, a single swap shifts the adversary's advantage by at most ϵ_{ZK} . Over all $q + 1$ swaps, $|\Pr[S_1] - \Pr[S_0]| \leq (q + 1)\epsilon_{\text{ZK}}$.

Game 2 (ReadAuth ciphertext randomized). We perform an indistinguishability step from Game 1. We replace the encapsulation ct_{auth}^* with one that wraps a fresh random key $\tilde{k}$, while the body and the digest keep the real key k^* , as in Game 0. To achieve that, when we produce the challenge envelope, we replace line 1 of Algorithm 1 as follows:

$$(k^*, ct_{\text{auth}}^*) \leftarrow \text{ReadAuth.Encaps}(\cdot) \mapsto (\tilde{k}, ct_{\text{auth}}^*) \leftarrow \text{ReadAuth.Encaps}(\cdot), k^* \xleftarrow{\$} \{0, 1\}^\lambda,$$

with the arguments of the call unchanged. The underlying idea is to exploit the IND-CPA property of ReadAuth: wrapping k^* is indistinguishable from wrapping $\tilde{k}$, so the swap cannot give $\mathcal{A}$ more advantage than the IND-CPA advantage. To show this, we build a reduction as follow. From $\mathcal{A}$, we construct an IND-CPA adversary $\mathcal{B}_{\text{KEM}}$ that runs $\mathcal{A}$ as a subroutine: it acts as the EARS challenger for $\mathcal{A}$, answering its oracle queries as in the honest game, and as the IND-CPA adversary for its own ReadAuth challenger (assumption C3, [20, Def. 11.11]). Whenever $\mathcal{A}$ tells apart Game 1 from Game 2, $\mathcal{B}_{\text{KEM}}$ wins IND-CPA.

There is, however, one detail to address in the reduction sketched above. IND-CPA is a statement about a *single* ReadAuth key, whereas the EARS game ranges over up to q_E epochs, each with its own independent ReadAuth key, but we want to randomize the wrapped key only in the challenge epoch eid^* . Thus, $\mathcal{B}_{\text{KEM}}$ must embed its IND-CPA instance in exactly that epoch: it generates the keys for all other epochs, while in the challenge epoch it adopts the public key provided by its IND-CPA challenger, whose corresponding secret key it does not know. The difficulty is that eid^* is chosen by $\mathcal{A}$ only at challenge time, after all epochs have already been set up, whereas $\mathcal{B}_{\text{KEM}}$ must decide in advance which epoch will contain its IND-CPA instance. It therefore guesses an epoch $\hat{j} \in \{0, \dots, q_E\}$ and aborts with a random bit if $\text{eid}^* \neq \hat{j}$. The guess is correct with probability $1/(q_E + 1)$, which accounts for the corresponding factor in the bound.

With this guess in place, $\mathcal{B}_{\text{KEM}}$ knows every ReadAuth secret except the one for epoch $\hat{j}$. It can still answer all of $\mathcal{A}$'s queries, because none of them needs that secret. The only oracle that reveals key material is CORRUPT, which hands $\mathcal{A}$ the keys of a corrupted device; but admissibility predicate Adm requires $\hat{j} \notin \text{RK}[\text{dev}]$ for every $\text{dev} \in \mathcal{C}$, whether because the device was hard-revoked before $\hat{j}$ or because it joined after it, so no corrupted device ever holds the epoch- $\hat{j}$ key. Its keys are therefore never materialized: JOIN and EPOCHTRANSITION update $\text{RK}[\cdot]$ as usual, and a CORRUPT that would expose one makes the run inadmissible, at which point $\mathcal{B}_{\text{KEM}}$ aborts with a random bit. The PUBLISH oracle needs no secret at all: encapsulation uses only the ReadAuth public parameters, which $\mathcal{B}_{\text{KEM}}$ holds for every epoch (including $\hat{j}$), and its proof is simulated as in Game 1. The only remaining use of epoch $\hat{j}$ is the challenge envelope: $\mathcal{B}_{\text{KEM}}$ receives from its challenger a pair $(ct, \hat{k})$, where $\hat{k}$ is either the key that ct encapsulates or an independent uniform one; it plants ct as ct_{auth}^* and uses $\hat{k}$ as the key k^* under which the challenge body and digest are produced. If $\hat{k}$ is the encapsulated key this reproduces Game 1; if it is independent, it reproduces Game 2. Hence $|\Pr[S_2] - \Pr[S_1]| \leq (q_E + 1)\epsilon_{\text{KEM}}$. This hop has detached k^* from ct_{auth}^* , which now wraps a random key $\tilde{k}$. Since no honest envelope ever wrapped k^* either, no ciphertext anywhere reveals it under any ReadAuth key. The real key k^* is thus a random secret hidden from $\mathcal{A}$, surviving only in the body c^* and the digest d^* . The next two games rely on this property: Games 3 and 4 randomize those two fields in turn, each arguing that $\mathcal{A}$ cannot recover k^* .

Game 3 (Binding digest randomized). We perform a failure-event step from Game 2 to Game 3. We replace the challenge digest with a uniformly random value $\tilde{d}$. This amounts to replacing line 3 of Algorithm 1 when generating the challenge envelope alone with:

$$d^* \leftarrow \text{Poseidon}(k^*, m_b) \mapsto d^* \leftarrow \tilde{d} \xleftarrow{\$} \mathbb{F}_p.$$

To achieve that, we model Poseidon by the random oracle H of Fig. A.1, as assumption C2 allows: the value $H(k^*, m_b)$ is then uniform, distributed exactly like $\tilde{d}$, as long as $\mathcal{A}$ never queries H at a point with first component k^* . Let F be the event that it does. Until F , the two games are identical, so the difference lemma [8] gives $|\Pr[S_3] - \Pr[S_2]| \leq \Pr[F] + q 2^{-\lambda}$. The second term accounts for the case in which an honest PUBLISH samples a key $k_i = k^*$ and reuses the corresponding digest: this event occurs with a negligible probability of at most $q 2^{-\lambda}$. This term is negligible but enters the final bound.

We need to bound $\Pr[F]$. After Game 2, the only part that depends on k^* is the ciphertext $c^* = \text{Enc}(k^*, m_b)$, so finding k^* means recovering an AES key from a single ciphertext. Just as in Game 2, we turn this into a reduction. We build a key-recovery adversary B_{KR} that runs $\mathcal{A}$ as a subroutine: it acts as the Game 3 challenger towards $\mathcal{A}$ and as the key-recovery adversary towards its own challenger (the key-recovery game of assumption C4), so that whenever $\mathcal{A}$ triggers F , B_{KR} recovers k^* . Concretely, it samples b and sends m_b to its key-recovery challenger, which returns (N^*, c^*) , the encryption of m_b under the unknown key k^* . It uses c^* as the challenge body and generates the rest of the Game 3 envelope itself: the wrapped key, the randomized digest, and the simulated proofs are all independent of k^* . The randomized digest is the crucial one: with d^* no longer tied to the oracle's value at k^* , B_{KR} can answer every H query with a fresh uniform value. The simulation runs without k^* . During its run $\mathcal{A}$ makes up to q_H queries to H , and F is the event that one of them lands on k^* ; that query then carries k^* in the clear. Since B_{KR} cannot tell which one it is, it picks one of the q_H at random and outputs its first component, recovering k^* with probability at least $\Pr[F]/q_H$. Hence, $\Pr[F] \leq q_H \epsilon_{KR}$.

Game 4 (AEAD body randomized). This is an indistinguishability step from Game 3. We replace the challenge body with an encryption of zeros. To achieve that we rewrite line 2 of Algorithm 1, with the following where the key we use is k^* :

$$c^* \leftarrow \text{Enc}(k^*, m_b) \mapsto c^* \leftarrow \text{Enc}(k^*, 0^{|m_b|}).$$

Since AES-GCM is IND-CPA, a ciphertext of m_b is indistinguishable from one of the same-length zero string. Just as before, we turn $\mathcal{A}$ into an IND-CPA attacker B_{AEAD} : it runs $\mathcal{A}$ as a subroutine, playing the challenger towards it and the IND-CPA adversary towards its own AES-GCM challenger (assumption C4, [20, Def. 3.22]), so that $\mathcal{A}$ distinguishing the two bodies makes B_{AEAD} win. Concretely, it samples b , submits the equal-length pair $(m_b, 0^{|m_b|})$, embeds the returned ciphertext as c^* , and builds the rest of the envelope without k^* . Since k^* encrypts only this one body, the AES-GCM nonce is trivially unique. Hence $|\Pr[S_4] - \Pr[S_3]| \leq \epsilon_{AEAD}$.

Closing. After Game 4, the challenge envelope now has no field that depends on b . The pseudonym nym^* is a function of the publisher's credential, chosen independently of b . The encapsulation ct^* wraps an independent random $\tilde{k}$, and the digest d^* is uniform. The body c^* encrypts the fixed string $0^{|m_b|}$, and since $|m_0| = |m_1|$ even its length leaks nothing. The checkpoint chk^* produced by the EM is b -independent. Finally, the simulated proof $\tilde{\pi}^* = S_{ZK}(td, x^*)$ depends only on the public statement x^* , whose components are all b -independent by the above. Hence $\Pr[S_4] = \frac{1}{2}$, and the triangle inequality across the four hops gives

$$\text{Adv}^{\text{EARS}}(\mathcal{A}; t) = \left| \Pr[S_0] - \frac{1}{2} \right| \leq (q + 1) \epsilon_{ZK} + (q_E + 1) \epsilon_{KEM} + q_H \epsilon_{KR} + q 2^{-\lambda} + \epsilon_{AEAD},$$

the bound of Theorem 1. $\square$

Corollary 2 (Forward and backward secrecy). *Under assumptions C1 – C4:*

1. *Forward secrecy (goal G2): no PPT adversary that corrupts a device hard-revoked in epoch i can tell which of two messages an envelope of epoch $\text{eid}^* > i$ protects, beyond the advantage bound of Theorem 1.*
2. *Backward secrecy (goal G3): no PPT adversary that corrupts a device joining in epoch j can tell which of two messages an envelope of epoch $\text{eid}^* < j$ protects, beyond the same bound.*

Proof. Both goals restrict Theorem 1 to a challenge epoch eid^* at which no corrupted device holds a valid ReadAuth key. Goal G2 (forward secrecy) places eid^* strictly after every corrupted device is hard-revoked, so re-keying has already removed its key. Goal G3 (backward secrecy) fixes eid^* strictly before the corrupted device joins, and re-keying hands a joining device the keys of its joining epoch onward and of no earlier one, so it never held the eid^* key. Either way no corrupted device can decrypt at eid^* , and Theorem 1 applies verbatim, with no additional games. $\square$

Theorem 3 (Subscriber authorization privacy). *If delivery is internal to subscribers, every adversary $\mathcal{A}$, even computationally unbounded, has $\text{Adv}^{\text{ReadPriv}}(\mathcal{A}; t) = 0$.*

Proof. The game is $\text{Exp}_{\mathcal{A}}^{\text{ReadPriv}}(\lambda, t)$ of Fig. A.3, where the secret bit selects whether the target subscriber S is issued a read key that satisfies the envelope's policy. Algorithm 2 runs entirely at the subscriber: it inspects the received envelope and returns either a plaintext to the application or $\perp$, with no interaction with the broker or the network. By the theorem's hypothesis, delivery is internal, so the subscriber emits no outcome-dependent message either. Whatever the verdict, and at whichever check it is reached, the subscriber's network behavior is therefore identical and independent of whether it holds a valid ReadAuth key for the envelope. Since the adversary observes only the network, its view of an authorized and of an unauthorized subscriber coincides: there is nothing to reduce to, and $\text{Adv}^{\text{ReadPriv}} = 0$, even against an unbounded adversary. $\square$

Theorem 4 (Writer unlinkability). *Under assumptions C1 and C2, for every PPT $\mathcal{A}$ with corruption bound t that makes q Publish queries*

$$\text{Adv}^{\text{Link}}(\mathcal{A}; t) \leq (q + 1) \epsilon_{ZK} + 2 \epsilon_{\text{PRF}}.$$

Proof. The proof proceeds through four games. The idea is that each game removes the dependence of one field of the envelope produced by the challenger on the secret bit b . Here that bit selects which of the two publishers produces the challenge envelope, hence the credential $\text{cred}_b = \text{st}[\text{dev}_b]$ the envelope is built from. The only values that depend on that credential are the Groth16

proof, whose witness is the credential, and the two Poseidon outputs it computes: the per-epoch pseudonym $\text{nym} = \text{Poseidon}(\text{cred}, \text{eid})$, carried in the envelope, and the accumulator leaf $\text{wid} = \text{Poseidon}(\text{cred})$, committed by the checkpoint. We replace each occurrence of the credential in this order: we first simulate the proofs, so they no longer carry the credential as a witness (Game 1); we then replace the credential's Poseidon outputs by random ones (Games 2 and 3). After these steps, no envelope field reveals the produced credentials.

Game 0 (Real Link). This is the game $\text{Exp}_{\mathcal{A}}^{\text{Link}}(\lambda, t)$ of Fig. A.4: $\mathcal{A}$ selects two enrolled, never-corrupted publishers $\text{dev}_0, \text{dev}_1$, holding $\text{cred}_0, \text{cred}_1$, whose envelopes it has observed in earlier epochs, and the challenger returns an envelope produced honestly from cred_b at the challenge epoch eid^* . The admissibility predicate Adm rules out any other envelope from those two publishers at eid^* , so $\mathcal{A}$ wins only by linking Env^* back to its producer across the epoch boundary. By definition, $\text{Adv}^{\text{Link}}(\mathcal{A}; t) = |\Pr[S_0] - \frac{1}{2}|$.

Game 1 (Simulated proofs). We replace the Groth16 proofs produced under either credential (their PUBLISH-oracle proofs and the challenge proof π^*), at most $q + 1$ in total, with simulator outputs $\tilde{\pi} \leftarrow S_{\text{ZK}}(td, x)$ on the same statement, leaving every other field untouched. To achieve that, we rewrite line 7 of Algorithm 1 in every call made by cred_0 or cred_1 as

$$\pi \leftarrow \text{Groth16.Prove}(\text{pk}_{gr}, \text{pub}, w) \mapsto \tilde{\pi} \leftarrow S_{\text{ZK}}(td, \text{pub}),$$

so by repeating the same reasoning we have $|\Pr[S_1] - \Pr[S_0]| \leq (q + 1)\epsilon_{\text{ZK}}$. The point of this hop is that simulated proofs need no witness, so the later games can randomize the credentials' Poseidon outputs.

Game 2 (cred_0 's Poseidon outputs randomized). The adversary never sees cred_0 in the clear: it enters its view only through Poseidon values keyed by it, namely the per-epoch pseudonyms $\text{Poseidon}(\text{cred}_0, \text{eid})$ and the accumulator leaf $\text{wid}_0 = \text{Poseidon}(\text{cred}_0)$. Since Poseidon keyed by a high-entropy credential is a pseudorandom function, by C2 these outputs are jointly indistinguishable from independent random values, so we replace the pseudonyms by a lazily sampled random function f_0 and the leaf by an independent uniform w_0 :

$$\text{Poseidon}(\text{cred}_0, \text{eid}) \mapsto f_0(\text{eid}), \quad \text{Poseidon}(\text{cred}_0) \mapsto w_0,$$

so we change line 4 of Algorithm 1 to produce $\text{nym} \leftarrow f_0(\text{eid})$ in every call made under cred_0 , at the challenge epoch and earlier. We make this precise with a reduction, as in the previous games. From $\mathcal{A}$ we build a PRF distinguisher $\mathcal{B}_{\text{PRF}}$ that runs $\mathcal{A}$ as a subroutine, playing the Link challenger towards it and the distinguisher towards its own PRF challenger (assumption C2, [20, Def. 3.25]). Its PRF oracle is either Poseidon keyed by cred_0 or a random function; $\mathcal{B}_{\text{PRF}}$ answers each of cred_0 's Poseidon outputs by querying that oracle. It generates every other field of the game, in particular, since it still knows cred_1 , it computes cred_1 's pseudonyms and leaf as real Poseidon evaluations, exactly as in Game 1. Note that pseudonyms and leaf cannot be randomized in separate hops: a distinguisher holding an oracle for one of the two derivations would need cred_0 itself to compute the other. If the oracle is the real keyed Poseidon it reproduces Game 1, if a random function it reproduces Game 2, so $\mathcal{A}$'s power to tell the two apart is exactly $\mathcal{B}_{\text{PRF}}$'s PRF advantage. Therefore, $|\Pr[S_2] - \Pr[S_1]| \leq \epsilon_{\text{PRF}}$.

Game 3 (cred_1 's Poseidon outputs randomized). Game 2 left cred_0 random but cred_1 still real. We therefore randomize cred_1 as well, with the identical process as above: the PRF oracle now sits in cred_1 , whose pseudonyms and leaf it replaces by an independent random function f_1 and an independent uniform w_1 , while the already-random cred_0 outputs are simulated by the reduction itself. So, we change line 4 of Algorithm 1 in $\text{nym} \leftarrow f_1(\text{eid})$ for cred_1 , so after this hop the pseudonym of either publisher is the output of an independent random function. Hence $|\Pr[S_3] - \Pr[S_2]| \leq \epsilon_{\text{PRF}}$.

Analysis of Game 3. The envelope resulting from Game 3 has no field that depends on b . The pseudonym $\text{nym}^* = f_b(\text{eid}^*)$ is the value of the random function f_b at the challenge epoch eid^* . That epoch is fresh by Adm , so f_b was never evaluated there before, and a random function returns a new, independent, uniform value at every unseen point. Hence nym^* is uniform regardless of b , revealing nothing about which of the two publishers produced it, and independent of the earlier-epoch pseudonyms the adversary has already seen. The simulated proof $\tilde{\pi}^*$ depends only on its public statement, whose components are by now b -independent. The ciphertext c^* and digest d^* are computed from the challenge message, which line 4 samples uniformly, and from a key the honest publisher draws afresh, both independent of b . The encapsulation ct_{auth}^* is produced under a read-side policy unrelated to writer identity. Finally, the accumulator root inside chk^* authenticates the set of enrolled leaves, which holds both publishers' leaves whichever credential is challenged. Hence $\Pr[S_3] = \frac{1}{2}$, and the triangle inequality across the hops gives

$$\text{Adv}^{\text{Link}}(\mathcal{A}; t) \leq (q + 1)\epsilon_{\text{ZK}} + 2\epsilon_{\text{PRF}},$$

the bound of Theorem 4. $\square$

Theorem 5 (EM compromise preserves confidentiality). *The bound of Theorem 1 holds even when the adversary is additionally given the EM's entire state.*

Proof. The EM's state includes the credential accumulator, the checkpoint-signing key sk_{em} , and the checkpoint counter. The reduction of Theorem 1 never relies on the secrecy of any of these elements. The EARS challenger can therefore reveal the EM's entire state to the adversary at the outset and still run every game verbatim, so the bound of Theorem 1 remains unchanged. $\square$

Theorem 6 (Partial KA compromise preserves proof soundness). *Under assumptions C1, C2, and C4, an adversary holding the KA's read-side master key but neither its signing key sk_{ka} nor any enrolled credential cannot produce a forged accepted envelope, except with negligible probability.*

Proof. This is the game $\text{Exp}_{\mathcal{A}}^{\text{Forge}}(\lambda, t)$ of Fig. A.5, where q counts the PUBLISH queries of $\mathcal{A}$ and q_H its queries to $\mathcal{H}$. An honest subscriber accepts an envelope only if its checkpoint chk carries a valid EM signature σ_{em} and its Groth16 proof verifies against the accumulator root recorded in chk . We bound the probability that the adversary produces a forged accepted envelope meeting both conditions.

The proof proceeds in four steps.

Step 1. By the unforgeability of EdDSA, the accepted checkpoint was issued by the honest EM except with probability ε_{em} , and every leaf admitted by the EM into the accumulator carried a valid KA signature σ_{ka} except with probability ε_{ka} . Therefore, the root r against which the proof is verified commits exactly to the set of KA-enrolled credentials, and the adversary, lacking sk_{ka} , cannot have enlarged that set.

Step 2. Replace each of the q honest proofs observed by the adversary with a simulated proof for the same statement. By the zero-knowledge of Groth16, this changes the adversary's success probability by at most $q \cdot \varepsilon_{ZK}$. The statement x of the adversarial envelope is forged by hypothesis, since no simulated proof was ever requested for it. Therefore, by the weak simulation-extractability of Groth16, there exists an extractor that, except with probability ε_{SE} , outputs a witness: a credential $cred$ together with a Merkle path authenticating $wid = \text{Poseidon}(cred)$ under the honest root r .

Step 3. By Step 1, every leaf of r is of the form $\text{Poseidon}(cred_i)$ for some enrolled credential $cred_i$. Since the extracted witness authenticates $wid = \text{Poseidon}(cred)$ under the same root r , wid must be one of the honest leaves. Thus, $\text{Poseidon}(cred) = \text{Poseidon}(cred_i)$ for some enrolled $cred_i$, and by the collision resistance of Poseidon, we have $cred = cred_i$, except with probability ε_{CR} .

Step 4. Steps 2–3 give us an algorithm that outputs an enrolled credential $cred_i$ from the adversary's view alone. We show no such algorithm can exist. After the simulation of Step 2, the credentials appear in that view only as images under H . The simulated proofs depend only on public statements, and the read-side master key is independent of every credential. Recovering a credential from these image values thus means inverting H , which requires querying it exactly on that credential. Since the KA samples each credential uniformly, any single query coincides with an enrolled credential with probability at most $2^{-\mu}$ (min-entropy μ), and over q_H queries with probability at most $q_H \cdot 2^{-\mu}$. Hence, Step 4 fails except with probability $q_H \cdot 2^{-\mu}$. Collecting the six error terms of the four steps by the union bound, the probability that the adversary produces a forged accepted envelope is at most

$$\varepsilon_{em} + \varepsilon_{ka} + q \varepsilon_{ZK} + \varepsilon_{SE} + \varepsilon_{CR} + q_H 2^{-\mu},$$

which is negligible. $\square$

References

- [1] E. Yuan, J. Tong, Attributed based access control (ABAC) for web services, in: IEEE International Conference on Web Services (ICWS'05), 2005, pp. 561–569. <https://doi.org/10.1109/ICWS.2005.25>
- [2] S. Rose, O. Borchert, S. Mitchell, S. Connelly, Zero Trust Architecture, Technical Report NIST Special Publication 800-207, National Institute of Standards and Technology, 2020. <https://doi.org/10.6028/NIST.SP.800-207>
- [3] A. Banks, E. Briggs, K. Borgendale, R. Gupta, MQTT Version 5.0, 2019, <https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html>. accessed 2026-06-13.
- [4] S. Gupta, T. Sacchetti, B. Crispo, End-to-end encryption for securing communications in industry 4.0, in: 2022 4th IEEE Middle East and North Africa Communications Conference (MENACOMM), 2022, pp. 153–158. <https://doi.org/10.1109/MENACOMM57252.2022.9998272>
- [5] F. Buccafurri, V. De Angelis, S. Lazzaro, MQTT-I: achieving end-to-end data flow integrity in MQTT, IEEE Trans. Dependable Secure Comput. 21 (5) (2024) 4717–4734. <https://doi.org/10.1109/TDSC.2024.3358630>
- [6] The NATS Authors, NATS Client Protocol, 2024, <https://docs.nats.io/reference/reference-protocols/nats-protocol>. accessed 2026-06-13.
- [7] S. Meier, B. Schmidt, C. Cremers, D. Basin, The TAMARIN prover for the symbolic analysis of security protocols, in: Computer Aided Verification (CAV), 2013, pp. 696–701. https://doi.org/10.1007/978-3-642-39799-8_48
- [8] V. Shoup, Sequences of games: a tool for taming complexity in security proofs, 2004, (Cryptology ePrint Archive, Paper 2004/332).
- [9] This paper, Proof-carrying payloads: end-to-end authorization for pub/sub messagin (Tamarin model and artifact), 2026, (Online repository). <https://github.com/lillo/pcp>
- [10] J. Liang, D. Hu, P. Wu, Y. Yang, Q. Shen, Z. Wu, SoK: Understanding zk-SNARKs: the gap between research and practice, in: 34th USENIX Security Symposium (USENIX Security 25), USENIX Association, Seattle, WA, 2025, pp. 2085–2104. <https://www.usenix.org/conference/usenixsecurity25/presentation/liang-sok>
- [11] J. Groth, On the size of pairing-based non-interactive arguments, in: M. Fischlin, J.-S. Coron (Eds.), Advances in Cryptology – EUROCRYPT 2016, Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pp. 305–326. https://doi.org/10.1007/978-3-662-49896-5_11
- [12] Y. Sakemi, S. Kanno, R.S. Wahby, Pairing-Friendly Curves, 2025, (Internet-Draft draft-irtf-cfrg-pairing-friendly-curves-12, IRTF CFRG). <https://www.ietf.org/archive/id/draft-irtf-cfrg-pairing-friendly-curves-12.html>
- [13] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, M. Schofnegger, Poseidon: a new hash function for zero-knowledge proof systems, in: 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, 2021, pp. 519–535. <https://www.usenix.org/conference/usenixsecurity21/presentation/grassi>
- [14] R.C. Merkle, A digital signature based on a conventional encryption function, in: C. Pomerance (Ed.), Advances in Cryptology — CRYPTO '87, Springer Berlin Heidelberg, Berlin, Heidelberg, 1987, pp. 369–378. https://doi.org/10.1007/3-540-48184-2_32
- [15] S. Agrawal, M. Chase, FAME: fast attribute-based message encryption, in: ACM CCS, 2017, pp. 665–682. <https://doi.org/10.1145/3133956.3134014>
- [16] J. Schaad, R. Housley, Advanced Encryption Standard (AES) Key Wrap Algorithm, 2002, (RFC 3394). <https://doi.org/10.17487/RFC3394>
- [17] D. Dolev, A.C. Yao, On the security of public key protocols, IEEE Trans. Inf. Theory 29 (2) (1983) 198–207. <https://doi.org/10.1109/TIT.1983.1056650>
- [18] S. Fischlin, Formalising Zero-Knowledge Proofs in the Symbolic Model, Master's thesis, ETH Zurich, 2021.
- [19] K. Bagheri, M. Kohlweiss, J. Siim, M. Volkhov, Another look at extraction and randomization of groth's zk-SNARK, in: N. Borisov, C. Diaz (Eds.), Financial Cryptography and Data Security – FC 2021, Volume 12674 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, Berlin, Heidelberg, 2021, pp. 457–475. https://doi.org/10.1007/978-3-662-64322-8_22
- [20] J. Katz, Y. Lindell, Introduction to Modern Cryptography, Chapman & Hall/CRC, 2nd edition, 2014.
- [21] M. Dworkin, NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, 2007, (NIST). <https://doi.org/10.6028/NIST.SP.800-38D>
- [22] M. Bellare, P. Rogaway, The security of triple encryption and a framework for code-based game-playing proofs, in: Advances in Cryptology – EUROCRYPT 2006, Springer, 2006, pp. 409–426. https://doi.org/10.1007/11761679_25
- [23] S. Goldwasser, S. Micali, Probabilistic encryption, J. Comput. Syst. Sci. 28 (2) (1984) 270–299. [https://doi.org/10.1016/0022-0000\(84\)90070-9](https://doi.org/10.1016/0022-0000(84)90070-9)
- [24] M. Bellare, P. Rogaway, Entity authentication and key distribution, in: Advances in Cryptology – CRYPTO '93, Springer, 1993, pp. 232–249. https://doi.org/10.1007/3-540-48329-2_21
- [25] F. De Rango, M.G. Spina, A. Iera, DLST-MQTT: dynamic and lightweight security over topics MQTT, Future Gener. Comput. Syst. 166 (2025) 107625. <https://doi.org/10.1016/j.future.2024.107625>
- [26] N. Liu, Y. Lei, W. Ren, L. Zhang, X. Zhang, T. Zhu, G. Min, zk-guard: a privacy-preserving access control framework based on zk-SNARKs and blockchain for decentralized data sharing, IEEE Trans. Dependable Secure Comput. 23 (3) (2026) 5637–5654. <https://doi.org/10.1109/TDSC.2026.3657485>
- [27] D. Li, N. Crespi, R. Minerva, W. Liang, K.-C. Li, J. Kolodziej, DPS-IIoT: non-interactive zero-knowledge proof-inspired access control towards information-centric industrial internet of things, Comput. Commun. 233 (2025) 108065. <https://doi.org/10.1016/j.comcom.2025.108065>

- [28] Istio Authors, Istio: connect, secure, control, and observe services, 2024, <https://istio.io/>. accessed 2026-04-27.
- [29] Cloud Native Computing Foundation, Open Policy Agent (OPA), 2024, <https://www.openpolicyagent.org/>
- [30] G. Sirigu, A. Burastero, A. Armando, Risk-based authorization in safety critical microservice architectures, in: Proceedings of the Joint National Conference on Cybersecurity (ITASEC & SERICS 2026), Volume 4198 of CEUR Workshop Proceedings, CEUR-WS.org, 2026. <https://ceur-ws.org/Vol-4198/paper70.pdf>
- [31] N. Sanger, S. Abeck, Externalized and decentralized authorization of microservices, in: Service-Oriented Computing – ICSOC 2024 Workshops, Springer, 2024, pp. 182–193. https://doi.org/10.1007/978-981-96-7238-7_15