Insynd: Improved Privacy-Preserving Transparency Logging

Roel Peeters[1] and Tobias Pulls[2]

[1] KU Leuven, ESAT/COSIC & iMinds, Belgium
[2] Karlstad University, Dept. of Mathematics and Computer Science, Sweden

Abstract. Service providers collect and process more user data then ever, while users of these services remain oblivious to the actual processing and utility of the processed data to the service providers. This leads users to put less trust in service providers and be more reluctant to share data. Transparency logging is about service providers continuously logging descriptions of the data processing on their users' data, where each description is intended for a particular user.
We propose Insynd, a new cryptographic scheme for privacy-preserving transparency logging. Insynd improves on prior work by (1) increasing the utility of all data sent through the scheme thanks to our publicly verifiable proofs: one can disclose selected events without having to disclose any long term secrets; and (2) enabling a stronger adversarial model: Inysnd can deal with an untrusted server (such as commodity cloud services) through the use of an authenticated data structure named Balloon. Finally, our publicly available prototype implementation shows greatly improved performance with respect to related work and competitive performance for more data-intensive settings like secure logging.

1 Introduction

In general, transparency logging allows service providers to show that they are compliant with a certain policy that can be imposed by legislation, sector regulations or internal procedures; but just as well through service level agreements for businesses to keep tabs on subcontractors [10,17]. For personal data, privacy regulations such as the EU General Data Protection Regulation empower users by granting them the right to obtain transparency about their data being processed and by improving their ability to hold the service providers accountable for their actions. Conceptually, through transparency logging, users that wish to know what is happening with their personal data after disclosure to a service provider can see whether or not the processing is inline with the prior agreed upon policy. This could, e.g., be a hospital with a privacy policy for processing patient data. Each access and modification to the patient's health record is logged for the patient. If patients discover someone prying, they can file a complaint with the hospital's ombudsperson.

In the setting of transparency logging [17] as depicted in Figure 1, the *author* generates *events* intended for *recipients* that describe data processing by the

author as it takes place. Events are stored at a *server*: an intermediate party that primarily serves to offload storage of events for authors. The recipient can then at a later point in time get insights into the author's data processing by consulting the events intended for him or her. With these insights, the recipient can hold the author accountable for its actions and if deemed necessary take remediation measures, e.g., file a complaint or switch service providers. Note that this paper focuses on a transparency logging scheme, which is only about the generation, storage and retrieval of events, not on what should be logged to describe data processing, or how policies should be structured to enable the comparison with stated data processing.

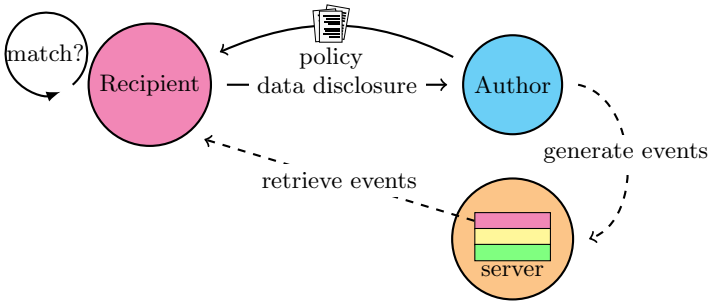


Fig. 1: Recipient comparing actual data processing of its data with the data processing that was agreed upon in the policy prior to data disclosure.

A transparency logging tool must provide security and privacy. The integrity of the stored events has to be guaranteed, as this is where the recipient bases its insights on to hold the author accountable. This means that it should be impossible to alter or delete any events after being stored at the server without being detectable. Data privacy and confidentiality of the stored events are important because the mere existence of events already reveals information, e.g., a patient visiting the hospital. This means that when a transparency logging tool does not consider privacy, one should deploy another transparency enhancing tool for monitoring data processing on the stored events from the first to make sure that data derived from these events are not used instead of the original data.

When users want to take action based upon the messages logged for them, they will inavoidably break some of the privacy properties. What we aim for with Insynd, our proposed cryptographic transparency logging tool, is to limit the privacy breaches to the events dislosed, i.e., enable selective disclosure, and as such greatly increase the utility of the transparency logging tool. Instead of having recipients reveal long term secrets, recipients can generate publicly verifiable proofs which allow them to disclose the content of stored events such that the content and other properties cannot be refuted. Furthermore, Insynd also improves on the scheme that was proposed by Pulls *et al.* [17] by allowing for a stronger adversarial model. While some trust in authors is inevitable (forward

security), since authors generate descriptions of their own processing, servers (e.g., commodity cloud services) should not have to be trusted. We primarily achieve the stronger adversary model through the use of Balloon [16], an authenticated data structure that was designed specifically for this setting. Lastly, our performance benchmarks show speeds comparable to state-of-the-art secure logging schemes. In summary, our contributions are:

- Increased utility of a transparency logging scheme through our publicly verifiable proofs: recipients and authors can produce publicly verifiable proofs of all data sent through Insynd, convincing a third-party of who sent what particular message to whom at approximately what time.
- A new transparency logging scheme in our stronger adversarial model where the server does not need to be trusted through the use of Balloon [16]. The resulting scheme also provides publicly verifiable consistency: anyone can verify that all events stored in a Balloon are consistent.
- A publicly available performant proof-of-concept implementation of Insynd using modern cryptographic primitives and benchmark code.

This paper is structured as follows. Section 2 states our assumptions and goals. Section 3 gives a high-level overview of our ideas. Section 4 presents Insynd in detail. Section 5 evaluates Insynd's properties. Section 6 presents related work. Section 7 shows the performance of our implementation.

2 Assumptions and Goals

We assume a setting with three parties: author, server and recipients. The author and recipients only have limited storage capabilities, while the server has high storage capabilities. The author is considered *forward secure*: the author is initially trusted until the time of compromise and the adversary, by compromising the author, gains no advantage towards breaking any of the security and privacy properties related to the events stored before compromise. The server is considered compromised from the start. Recipients are considered honest.

For communication, we assume a secure channel between the author and the server (such as TLS), and a secure and anonymous channel for recipients (such as TLS over Tor [9]) to communicate with the author and server. We explicitly consider availability out of scope, that is, the author and server will always reply (however, their replies may be malicious). For time-stamps, we assume the existence a trustworthy time-stamping authority [7].

For the core security and privacy properties: secrecy, forward integrity with deletion-detection and forward unlinkability of events, we make use of the model of Pulls *et al.* [17], with some modifications to account for possible information leakage through our introduced publicly verifiable proofs[3] and our stronger adversarial setting. The full updated model is available in Appendix A. Secrecy

[3] Since state is kept by the author instead of the server (which is assumed to be untrusted), the `CorruptServer` oracle is replaced by a `CorruptAuthor` oracle. To account for information leakage, additional oracles such as `GetState`, `DecryptEvent` and `RecipientEvent` are introduced.

is vital for recipients since events may contain sensitive personal data. Forward integrity with deletion-detection ensures that events are tamper evident: any modifications (including deletion) can be detected. Finally, forward unlinkability of events ensures that prior generated events do not leak information such as the number of events that belong to a particular recipient.

In addition to the core security and privacy properties, we provide publicly verifiable consistency and a number of publicly verifiable proofs to increase the utility of the data sent through the transparency logging scheme. Publicly verifiable consistency can be seen as a form of publicly verifiable deletion-detection and forward integrity for all events produced by the author at a server. Insynd allows for publicly verifiable proofs of (1) the author of an event, (2) the recipient of an event, (3) the message sent in an event, and (4) the time an event existed at a server. While a recipient is always able to produce these proofs, the author has to decide during event generation if it wishes to save material to be able to create these proofs. Each proof is an isolated disclosure and a potential violation of a property of Insynd, like secrecy and forward unlinkability of events.

3 Ideas

To protect the privacy of the recipients, the author turns all descriptions for recipients into events consisting of an identifier and a payload, where the identifiers are unlinkable to each other and the payloads contain the encrypted descriptions for the recipient. It should be noted that the entire events must be unlinkable to each other, hence the encryption scheme must also provide key privacy [2]. Later on the recipient must be able to retrieve its relevant events and decrypt the logged descriptions. For each event, the author updates the symmetric event linking key for the recipient in question using a forward-secure sequential key generator (SKG) in the form of an evolving hash chain [3,18,13]. The recipient can do the same to link the relevant event identifiers together.

To provide the publicly verifiable proofs of message and recipient, we need to go into the details of the used encryption scheme and how the event linking key and nonce for encryption are derived from the forward secure sequential key. We make use of an IND-CCA2[4] *public-key authenticated encryption* scheme [1] in a non-traditional manner. A public key authenticated encryption scheme allows a sender to encrypt a message for a receiver using the receiver's public key and its own private key, such that the receiver can decrypt the message using its own private key and the sender's public key. In this way, both sender and receiver can decrypt the message and be assured that only someone who knows either private key can have created the ciphertext. To avoid a deterministic encryption scheme, a nonce is usually included for each message to be encrypted. Instead of taking the author's private key as input, we generate a fresh ephemeral public private key pair for each message, send along the public key and append the private key to the message to be encrypted. As such the recipient can prove, by revealing

[4] Every publicly verifiable proof is an isolated disclosure, hence the encryption scheme must provide secrecy even when the adversary has access to a decryption oracle.

the ephemeral private key and the nonce, that the ciphertext contains the said plaintext. The author can do the same if it stores the ephemeral private key at the time of creating the event. We define the following algorithms, based on the algorithms of the public key authenticated encryption scheme $\Pi = \{(\mathtt{sk}, \mathtt{pk}) \leftarrow \mathbf{KeyGen}(1^\lambda),\ c \leftarrow \mathbf{Enc}^n_{\mathtt{pk}}(m),\ m \leftarrow \mathbf{Dec}^n_{\mathtt{sk}}(c, \mathtt{pk})\}$:

- $(c, \mathtt{pk}') \leftarrow \mathtt{Enc}^n_{\mathtt{pk}}(m)$: Encrypts a message m using an ephemeral key-pair $(\mathtt{sk}', \mathtt{pk}') \leftarrow \mathbf{KeyGen}(1^\lambda)$, the public key $\mathtt{pk}$, and the nonce n. The resulting ciphertext c is $\mathbf{Enc}^n_{\mathtt{pk}'}(m||\mathtt{sk}')$. Returns $(c, \mathtt{pk}')$.
- $(m, \mathtt{sk}') \leftarrow \mathtt{Dec}^n_{\mathtt{sk}}(c, \mathtt{pk}')$: Decrypts a ciphertext c using the private key $\mathtt{sk}$, public key $\mathtt{pk}'$, and nonce n where $p \leftarrow \mathbf{Dec}^n_{\mathtt{sk}}(c, \mathtt{pk}')$. If decryption fails $p = \perp$, otherwise $p = m||\mathtt{sk}'$. Returns p.
- $m \leftarrow \mathtt{Dec}^n_{\mathtt{sk}',\mathtt{pk}}(c, \mathtt{pk}')$: Decrypts a ciphertext c using the private key $\mathtt{sk}'$, public key $\mathtt{pk}$, and the nonce n where $p \leftarrow \mathbf{Dec}^n_{\mathtt{sk}'}(c, \mathtt{pk})$. If decryption fails $p = \perp$, otherwise $p = m||\mathtt{sk}^*$. If $\mathtt{sk}' = \mathtt{sk}^*$ and corresponds to $\mathtt{pk}'$, returns m, otherwise $\perp$.

The event linking key k' and nonce n for encryption are derived from the current authentication key k (Figure 2). The event linking key is used to prove the recipient of the event. By deriving the event linking key from the nonce, we prove that the recipient corresponds to the decrypted message.

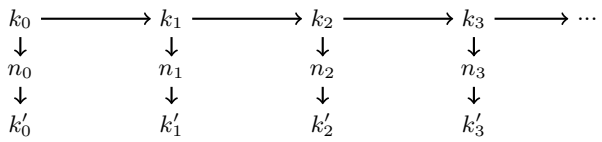


Fig. 2: Deriving the event key k' and nonce n. Each arrow represents a one-way relation, e.g., from k it is easy to compute n, but the other way around is hard.

Through using Balloon [16], an authenticated data structure that was designed for the setting of transparency logging with an untrusted server, we can support our stronger adversarial model and provide publicly verifiable proofs of consistency. Balloon allows for efficient publicly verifiable proofs of both membership and non-membership of keys. This is needed, since otherwise a recipient cannot distinguish between a server denying service and the lack of an event with a specific identifier. The main advantage of Balloon compared to other authenticated data structures that have this property[5], is that the author only needs to keep constant storage (instead of storing a copy of the data structure) and that proof generation is more efficient for the server. The main algorithms from Balloon that are used by Insynd are:

- `B.query` (**Membership**) and `B.verify` (**Membership**) to generate as well as verify (non-)membership proofs.

[5] For a more in-depth discussion, we refer the reader to Pulls and Peeters [16].

– `B.query` (**Prune**), `B.verify` (**Prune**), `B.update*`, and `B.refresh` to insert a new set of events into a Balloon and generate a new *snapshot*, which commits the author to all the events that are stored until now.

The full algorithm descriptions can be found in Appendix B. To support a forward-secure author (preventing it from creating snapshots that delete or modify events inserted prior to compromise), Balloon requires trusted *monitors* and a *perfect gossiping mechanism* for the snapshots. Monitors continuously reconstruct the Balloon and compare calculated snapshots with those gossiped (spread simultaneously to all recipients) by the author. We relax these requirements by linking snapshots together and periodically timestamping these; and by introducing forward integrity with deletion-detection for each recipient.

To provide forward integrity with deletion detection, we rely on the author keeping an evolving forward-secure state for each recipient. By enabling the recipient to query for this state and verifying the response, it is impossible for the author to alter events for this recipient (sent to the server prior to the time of compromise) as it will not be able to generate a valid state to send to the recipient. During recipient registration, cryptographic key material will be set up for the recipient: an asymmetric key-pair, for encryption and decryption, and a symmetric key to be able to link relevant events together. For each recipient, the current values of the forward-secure SKG and the forward-secure sequential aggregate authenticator (FssAgg) [12] over the relevant event values are kept in the author's state.

4 Insynd

Now we will go into the details of the different protocols that make up Insynd. Figure 3 shows five protocols between an author A, a server S, and a recipient R. The protocols are `setup` (pink box), `register` (blue box), `insert` (yellow box), `getEvent` (red box), and `getState` (green box). The following subsections describe each protocol in detail.

4.1 Setup and Registration

The author and server each have signature key pairs, $(\mathsf{A}_{\mathtt{sk}}, \mathsf{A}_{\mathtt{vk}})$ and $(\mathsf{S}_{\mathtt{sk}}, \mathsf{S}_{\mathtt{vk}})$, respectively. We assume that $\mathsf{A}_{\mathtt{vk}}$ and $\mathsf{S}_{\mathtt{vk}}$ are publicly attributable to the respective entities, e.g., by the use of some trustworthy public-key infrastructure. For the author, the key pair is generated using the `B.genkey` algorithm of Balloon, as this key pair is also used to sign the snapshots, which are part of Balloon.

Author-Server Setup. The purpose of the `setup` protocol (pink box in Figure 3) is for the author and the server to create a new Balloon, stored at the server, with two associated uniform resource identifiers (URIs): one for the author $\mathsf{A}_{\mathtt{URI}}$, and one for the server $\mathsf{S}_{\mathtt{URI}}$. At the former the recipient can later on query for its current state, while at the latter it can retrieve stored events. The

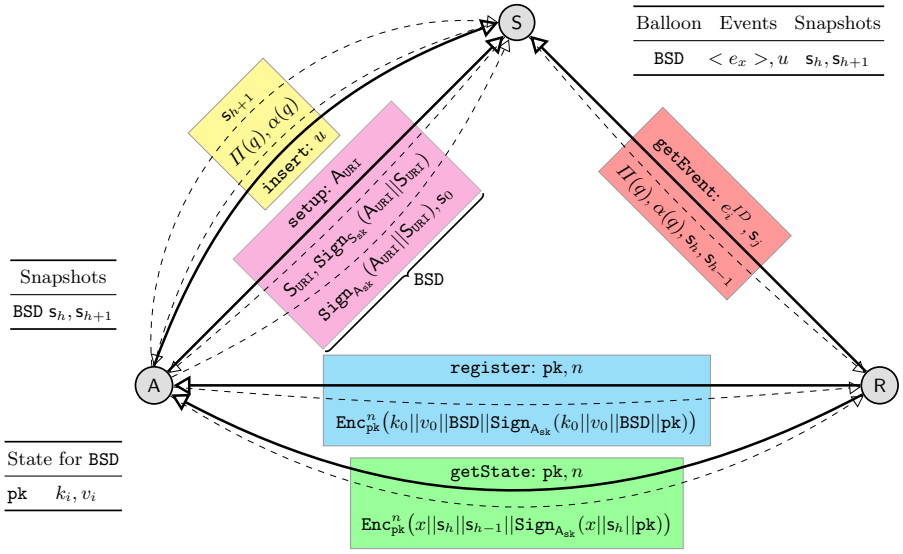


Fig. 3: Insynd consists of five protocols (coloured boxes), between an author A, server S, and recipient R. A solid line indicates the start of protocol and a dashed line a response.

result of this protocol, the *Balloon setup data* (BSD), commits both the author and the server to the newly created Balloon.

The protocol is started by the author sending its $\mathsf{A}_{\mathtt{URI}}$ to the server. The server replies with $\mathsf{S}_{\mathtt{URI}}$ and $\mathtt{Sign}_{\mathsf{S}_{\mathtt{sk}}}(\mathsf{A}_{\mathtt{URI}}||\mathsf{S}_{\mathtt{URI}})$. The signature commits the server to the specified Balloon. Upon receiving the reply from the server, the author verifies the server's signature. If this verifies, the author creates an empty Balloon $(\mathtt{auth}(D_0), \mathsf{s}_0) \leftarrow \mathtt{B.setup}(D_0, \mathsf{A}_{\mathtt{sk}}, \mathsf{A}_{\mathtt{vk}})$ for an empty data structure D_0. The author sends $\mathtt{Sign}_{\mathsf{A}_{\mathtt{sk}}}(\mathsf{A}_{\mathtt{URI}}||\mathsf{S}_{\mathtt{URI}})$ together with the initial snapshot s_0 to the server to acknowledge that the new Balloon is now set up. Once the server receives this message, it verifies the author's signature and can complete the setup of the empty Balloon now that it has s_0. The two signatures, the two URIs, and the initial snapshot s_0 together form the BSD.

Recipient Registration. The purpose of the `register` protocol (blue box in Figure 3) is to enable the author to send messages to the recipient later on, and at the same time have the author commit to the recipient on how these messages will be delivered. Before running the protocol, the recipient is assumed to have generated its encryption key pair $(\mathtt{pk}, \mathtt{sk})$.

The protocol is initiated by the recipient sending its public key together with a nonce to the author. The author generates the initial authentication key $k_0 \leftarrow \mathtt{Rand}(|\mathtt{Hash}(\cdot)|)$ and authenticator value $v_0 \leftarrow \mathtt{Rand}(|\mathtt{Hash}(\cdot)|)$ for this recipient and stores these values in its *state table* for BSD. The state table contains

the *current* authentication key k_i and authenticator value v_i for each recipient's public key that is registered in the Balloon for BSD. By generating a random v_0, the state of newly registered recipients is indistinguishable from the state of recipients that have already one or more events created for them.

The author returns to the recipient k_0, v_0, BSD, and the following signature: $\mathtt{Sign}_{\mathtt{A}_{\mathtt{sk}}}(k_0||v_0||\mathtt{BSD}||\mathtt{pk})$. The signature covers the public key of the recipient to bind the registration to a particular public key (and hence recipient). The signature (that commits the author) is necessary to prevent the author from fully refuting that there should exist any messages for this recipient. The reply to the recipient is encrypted by the author under the provided public key and nonce. On receiving the reply, the recipient decrypts the reply, verifies all three signatures (two in BSD), and stores the decrypted reply. The recipient now has everything it needs to retrieve its relevant events and state later on.

4.2 Event Generation

An event $e = (e^{ID}, e^{P})$ consists of an identifier and a payload. The event identifier e^{ID} identifies the event in a Balloon and is used by the recipient to retrieve an event. The event payload e^{P} contains the encrypted message from the author. The nonce n, used for encrypting the event payload, and the event key k', used for generating the event identifier, are derived from the recipient's current authentication key k (which the author retrieves from its state table):

$$n \leftarrow \mathtt{Hash}(1||k) \qquad \text{and} \qquad k' \leftarrow \mathtt{Hash}(n) \tag{1}$$

For deriving the nonce, a prefix 1 is added to k to distinguish between deriving the nonce and updating the authentication key, which is done as follows:

$$k_i \leftarrow \mathtt{Hash}(k_{i-1}) \tag{2}$$

The event identifier is generated by computing a MAC on the recipient's public key using the event key:

$$e^{ID} \leftarrow \mathtt{MAC}_{k'}(\mathtt{pk}) \tag{3}$$

This links the event to a particular recipient, which can be used for publicly verifiable proofs of recipient. The event payload is generated by encrypting the message under the recipient's public key and the generated nonce: $e^{P} \leftarrow \mathtt{Enc}^{n}_{\mathtt{pk}}(m)$. Since k' is derived from n, this links the event identifier and event payload together and can be used for publicly verifiable proofs of message.

After generating the event, the author updates its state table, effectively overwriting previous values. First the current authenticator value v for the recipient, which aggregates the entire event, is updated using an FssAgg [12]:

$$v_i \leftarrow \mathtt{Hash}\big(v_{i-1}||\mathtt{MAC}_{k_{i-1}}(e)\big) \tag{4}$$

Then the recipient's current authentication key is updated using Equation 2.

Insert. The purpose of the `insert` protocol (yellow box in Figure 3) is for an author to insert a set of generated events u into a Balloon kept by the server. The author sends u to the server and gets back a proof that the events can be correctly inserted. If this proof verifies, the author creates a new snapshot, committing to the current version of the Balloon.

Upon receiving u, the server runs:

$$(\Pi(u), \alpha(u)) \leftarrow \texttt{B.query}(u, D_h, \texttt{auth}(D_h), \mathsf{A}_{\texttt{vk}})(\textbf{Prune})$$

to generate a proof $\Pi(u)$ and answer $\alpha(u)$ and sends these back to the author. To verify the correctness of the server's reply, the author runs:

$$\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{B.verify}(u, \alpha, \Pi, \mathsf{s}_h, \mathsf{A}_{\texttt{vk}})(\textbf{Prune})$$

where s_h is the latest snapshot generated by the author. If the verification fails, the author restarts the protocol. Next, the author runs:

$$(\mathsf{s}_{h+1}, \texttt{upd}) \leftarrow \texttt{B.update*}(u, \Pi, \mathsf{s}_h, \mathsf{A}_{\texttt{sk}}, \mathsf{A}_{\texttt{vk}})$$

to create the next snapshot s_{h+1} (which is also stored in `upd`). The author stores the snapshot in its *snapshot table* for `BSD`, and sends `upd` to the server. The server verifies the snapshot and then runs:

$$(D_{h+1}, \texttt{auth}(D_{h+1}), \mathsf{s}_{h+1}) \leftarrow \texttt{B.refresh}(u, D_h, \texttt{auth}(D_h), \mathsf{s}_h, \texttt{upd}, \mathsf{A}_{\texttt{vk}})$$

to update the Balloon. Finally, the server stores the snapshot s_{h+1} and events u in its *Balloon table* for `BSD`.

Snapshots and Gossiping. Balloon assumes perfect gossiping of snapshots. In order to relax this requirement, we modify the snapshot construction. This modification was inspired by CONIKS [15], which works in a setting closely related to ours and links snapshots together into a snapshot chain. We redefine a snapshot as:

$$\mathsf{s}_h \leftarrow \left(i, c_i, r, t, \texttt{Sign}_{\mathsf{A}_{\texttt{sk}}}(i||c_i||r||s_{h-1}||t)\right)$$

Note that h is an index for the number of updates to Balloon, while i is an index for the number of events in the Balloon. The snapshot s_h contains the latest commitment c_i on the history tree and root r on the hash treap for $\texttt{auth}(D_h)$, fixing the entire Balloon[6]. The previous snapshot s_{h-1} is included to form the snapshot chain. Finally, an *optional* timestamp t from a trusted time-stamping authority is included both as part of the snapshot and in the signature. The timestamp must be on $(i||c_i||r||s_{h-1})$. How frequently a timestamp is included in snapshots directly influences how useful proofs of time are. Timestamping of snapshots is irrelevant for our other properties.

Gossiping of snapshots is done by having the author and server making all snapshots available, e.g., on their websites. Furthermore, the latest snapshots are gossiped to the recipients as part of the `getState` and `getEvent` protocols (described next). Since snapshots are both linked and occasionally timestamped, this greatly restricts adversaries in the forward-security model.

[6] Balloon is the composition of a history tree and hash treap [16].

4.3 Event Reconstruction

A recipient uses two protocols to reconstruct its relevant messages sent by the author: `getEvent` and `getState`. After explaining how to get the relevant events and the current state, we show how recipient can verify the consistency of its retrieved messages.

Getting Events. The purpose of the `getEvent` protocol (red box in Figure 3) is for a recipient to retrieve an event with a given identifier and an optional snapshot. The server replies with the event (if it exists) and a proof of membership. Before running this protocol, the recipient generates the event identifier it is interested in, by using Equations 1-3 together with the data it received from the author during registration.

Upon receiving the event identifier e^{ID} and optional snapshot $\mathtt{s}_j$ from the recipient, the server runs for $q = (e^{ID}, \mathtt{s}_j)$:

$$\Big(\Pi(q), \alpha(q)\Big) \leftarrow \mathtt{B.query}(q, D_h, \mathtt{auth}(D_h), \mathsf{A}_{\mathtt{vk}})(\textbf{Membership})$$

If no snapshot is provided, the server uses the latest snapshot $\mathtt{s}_h$. Allowing the recipient to query for any snapshot $\mathtt{s}_j$, where $j \leq h$, is important for our publicly verifiable proofs of time. The server replies to the recipient with $(\Pi(q), \alpha(q), \mathtt{s}_h, \mathtt{s}_{h-1})$. Including the two latest snapshots $\mathtt{s}_h$ and $\mathtt{s}_{h-1}$ is part of our gossiping mechanism and allows for fast verification at the recipient without having to download all snapshots separately. The recipient verifies the reply by verifying the last snapshot and running:

$$\{\mathtt{accept}, \mathtt{reject}\} \leftarrow \mathtt{B.verify}(q, \alpha, \Pi, \mathtt{s}_h, \mathsf{A}_{\mathtt{vk}})(\textbf{Membership})$$

Getting State. The `getState` protocol (green box in Figure 3) plays a central role in determining the consistency of the events retrieved from the server.

The recipient initiates the protocol by sending its public key `pk` and a nonce $n \leftarrow \mathtt{Rand}(|\mathtt{Hash}(\cdot)|)$ to the author. Upon receiving the public key and nonce, the author validates the public key and sets $x \leftarrow (k_i, v_i)$, with k_i and v_i being the current state for `pk`, retrieved from its state table. The author replies with $\mathtt{Enc}^n_{\mathtt{pk}}\big(x||\mathtt{s}_h||\mathtt{s}_{h-1}||\mathtt{Sign}_{\mathsf{A}_{\mathtt{sk}}}(x||\mathtt{s}_h||\mathtt{pk})\big)$. This reply also covers the two latest snapshots $\mathtt{s}_h$ and $\mathtt{s}_{h-1}$, as part of the gossiping mechanism and a signature of the author over $(x||\mathtt{s}_h||\mathtt{pk})$. With this signature the author commits itself to its reply for the recipient with respect to the latest snapshot. The recipient decrypts the reply, verifies the signature and latest snapshot.

The reply to the claimed recipient is encrypted using the provided public key and nonce to ensure that only the recipient with corresponding the private key can decrypt it. Since the encryption is randomised with the nonce and ephemeral key-pair generation (note that the length of the plaintext is fixed), no third party in possession of the recipient's public key can determine if new events are generated for the recipient. The nonce also ensures the *freshness* of the reply.

Verifying Consistency. A recipient can verify the consistency of the messages contained in its events as follows. First, it requests all its events until the server provides a non-membership proof. Next, the recipient retrieves its current state from the author. Note that in order to be able to verify the consistency of the received messages it is essential that the latest snapshot received during `getEvent` for the last downloaded message (for which a non-membership proof is received) and the latest snapshot received during `getState` are identical.

With the list of events downloaded and the reply x from `getState`, the recipient can now use Algorithm 1 to decrypt all events and verify the consistency of the messages sent by the author. First all events (in the order of insertion) are decrypted using the nonce and authentication key generation determined by Equations 1-2 and the calculated state (Equation 4) is updated. Finally the calculated state is compared to the x.

Algorithm 1 Verify message consistency for a recipient.

Require: $\texttt{pk}, \texttt{sk}, k_0, v_0$, the reply x from `getState`, an ordered list l of events.
Ensure: `true` if all events are authentic and the state x is consistent with the events in l, otherwise `false`.

1: $n \leftarrow \texttt{Hash}(1||k), k \leftarrow k_0, v \leftarrow v_0$ ▷ n is the event nonce, k and v the computed state
2: **for all** $e \in l$ **do** ▷ in the order events were inserted
3: $\quad p \leftarrow \texttt{Dec}^n_{\texttt{sk}}(e^P)$
4: $\quad$ **if** $p \stackrel{?}{=} \perp$ **then**
5: $\quad\quad$ **return** `false` ▷ failed to decrypt event
6: $\quad n \leftarrow \texttt{Hash}(1||k), k \leftarrow \texttt{Hash}(k), v \leftarrow \texttt{Hash}\big(v||\texttt{MAC}_k(e)\big)$ ▷ computed right to left
7: **return** $x \stackrel{?}{=} (k, v)$ ▷ state should match calculated state

4.4 Publicly Verifiable Proofs

Similar to Balloon, Insynd allows for publicly verifiable consistency. On top of this, Insynd allows for four types of publicly verifiable proofs: author, time, recipient, and message. These proofs can be combined to, at most, prove that the author had sent a message to a recipient at a particular point in time. While the publicly verifiable proofs of author and time can be generated by anyone, the publicly verifiable proofs of recipient and message can only be generated by the recipient (always) and the author (if it has stored additional information at the time of generating the event).

Author. To prove who the *author* of a particular event is, i.e., that an author created an event, we rely on Balloon. The proof is the output from `B.query` (**Membership**) for the event. Verifying the proof uses `B.verify` (**Membership**).

Time. To prove *when* an event *existed*. The granularity of this proof depends on the frequency of timestamped snapshots. The proof is the output from `B.query`

(**Membership**) for the event from a timestamped snapshot $\mathtt{s}_j$ that shows that the event was part of the data structure fixed by $\mathtt{s}_j$. Verifying the proof involves using `B.verify` (**Membership**) and whatever mechanism is involved in verifying the timestamp from the time-stamping authority. Note that a proof of time proves that an event existed at the time as indicated by the time-stamp, not that the event was inserted or generated at that point in time.

Recipient. To prove who the *recipient* of a particular event is. This proof consists of:

1. the output from `B.query` (**Membership**) for the event, and
2. the event key k' and public key `pk` used to generate the event identifier e^{ID}.

Verifying the proof involves using `B.verify` (**Membership**), calculating $\tilde{e}^{ID} \leftarrow \mathtt{MAC}_{k'}(\mathtt{pk})$, and comparing it to the event identifier e^{ID}.

The recipient can always generate this proof, while the author needs to store the event key k' and public key `pk` at the time of event generation. If the author stores this material, then the event is linkable to the recipient's public key. If linking an event to a recipient's public key is not adequately attributing an event to a recipient (e.g., due to the recipient normally being identified by an account name), then the `register` protocol should also include an extra signature linking the public key to additional information, such as an account name.

Message. The publicly verifiable proof of message includes a publicly verifiable proof of recipient, which establishes that the ciphertext as part of an event was generated for a specific public key (recipient). The proof is:

1. the output from `B.query` (**Membership**) for the event,
2. the nonce n needed for decryption and used to derive the event key k',
3. the public key `pk` used to generate e^{ID}, and
4. the ephemeral secret key $\mathtt{sk}'$ that is needed for decryption.

Verifying the proof involves first verifying the publicly verifiable proof of recipient by deriving $k' = \mathtt{Hash}(n)$. Next, the verifier can use $\mathtt{Dec}^{n}_{\mathtt{sk}',\mathtt{pk}}(c, \mathtt{pk'})$ to learn the message m.

The recipient can always generate this proof, while the author needs to store the nonce n, public key `pk`, and the ephemeral private key `sk'` at event generation. Note that even thought we allow the author to save the ephemeral key material to produce publicly verifiable proofs of message, the author is never allowed to do so for the encrypted replies to the `getState` or `register` protocols.

5 Evaluation

The proof sketches use the model in Appendix A.

5.1 Security and Privacy Properties

Theorem 1. *For an IND-CCA2 secure public-key encryption scheme, Insynd provides computational secrecy of the messages contained in events.*

This follows trivially from the definition of IND-CCA2 security.

Theorem 2. *Given an unforgeable signature algorithm, an unforgeable one-time MAC, and an IND-CCA2 secure public-key encryption algorithm, Insynd provides computational deletion-detection forward integrity in the random oracle model.*

Proof (sketch). This follows from the use of the FssAgg authenticator by Ma and Tsudik [12], which is provably secure in the random oracle model for an unforgeable MAC function.

The `register` protocol establishes the initial key and value for the forward secure SKG and FssAgg authenticator. These values, together with the `BSD` and the public key of the recipient, are signed by the author and returned to the recipient. Assuming an unforgeable signature algorithm, this commits the author to the existence of *state*. The recipient gets the current state using the `getState` protocol for its public key and a fresh nonce. The reply from the author is encrypted under the recipient's provided public key and the nonce provided by the recipient. The nonce ensures the *freshness* of the reply, preventing the adversary from caching replies from the `getState` protocol made prior to compromise of the author (using the `GetState` oracle). The current authenticator value and authentication key are updated (and overwritten) by using the FssAgg construction and a forward secure SKG. Note that for each FssAgg invocation, the key for the MAC is unique and derived from the output of a hash function for which the adversary has no information on the input. This means that an unforgeable one-time MAC function is sufficient.

The adversary does not learn any authenticator values and keys through the `GetState`, `DecryptEvent` or `RecipientEvent` oracles. This is due to the use of an IND-CCA2 encryption scheme, and the values k' and n in the proofs Π are derived from the current authentication key at that time using a random oracle. □

Theorem 3. *For a key-private IND-CCA2 secure public-key encryption algorithm, Insynd provides computational forward unlinkability of events within one round of the `insert` protocol in the random oracle model.*

Proof (sketch). For events created with the `CreateEvent'` oracle the adversary has access to the following information: $e^{ID} = \texttt{MAC}_{k'}(\texttt{pk})$ and $e^{P} = \texttt{Enc}^{n}_{\texttt{pk}}(m)$ for which $k' = \texttt{Hash}(n)$ and $n = \texttt{Hash}(1||k)$ where k is the current authentication key for the recipient at the time of generating the event.

By assuming the random oracle model, the key to the one-time unforgeable MAC function and the nonce as input of the encryption are truly random. Hence the adversary that does not know the inputs of these hashes, n and k respectively, has no advantage towards winning the indistinguishability game. We will now

show that the adversary will not learn these values n and k, even when given the author's entire state $(\texttt{pk}, k, v)$ for all recipients and access to the `GetState`, `DecryptEvent` or `RecipientEvent` oracles. From the previous proof we already know that the adversary does not learn any authenticator values and keys from the latter three oracles. Hence, it will also not learn any n values for events generated with the `CreateEvent'` oracle, since there is no direct link between the values n_i of multiple events for the same recipient. Instead n is derived from the recipient's current authentication key k at that time, using a random oracle.

The state variable k is generated using a forward-secure sequential key generator in the form of an evolving hash chain. Since the encryption scheme of events is key private, the adversary does not learn anything from all the recipients' public keys `pk`. Finally, we need to show that the adversary will not be able to link events together from the state variable v. If $v = v_0$, then v is random. Otherwise, $v_i = \texttt{Hash}\big(v_{i-1} || \texttt{MAC}_{k_{i-1}}(e_{i-1})\big)$. The `MAC` is keyed with the previous authentication key k_{i-1}, which is either the output of a random oracle (if $i > 1$) or random (k_0). This means the adversary does not know the output of $\texttt{MAC}_{k_{i-1}}(e^j_{i-1})$ that is part of the input for the random oracle to generate v. □

5.2 Publicly Verifiable Proofs

Consistency. Assuming a collision resistant hash function, an unforgeable signature algorithm, monitors, and a perfect gossiping mechanism for snapshots, this follows directly from the properties of Balloon (Theorem 3 of [16]). However, our gossiping mechanisms are imperfect. We rely on the fact that (1) recipients can detect any modifications on their own events and (2) snapshots are chained together and occasionally timestamped, to deter the author from creating inconsistent snapshots. The latter one ensures that at least fork consistency as defined by Mazières and Shasha [14] is achieved. This means that in order to remain undetected the adversary needs to maintain a fork for every recipient it disclosed modified snapshots to.

Author. Assuming a collision resistant hash function and an unforgeable signature algorithm, the proof of author cannot be forged. A proof of author for an event is the output from `B.query` (**Membership**) for the event. Theorem 2 in [16] proves the security of a membership query in a Balloon. For an unforgeable signature algorithm, the existence of a signature is therefore non-repudiable evidence of the snapshot having been created with the signing key.

Time. Assuming a collision resistant hash function, an unforgeable signature algorithm and a secure time-stamping mechanism, the proof of author cannot be forged. A proof of time depends on the time-stamping mechanism, which is used in the snapshot against which the proof of author was created.

Recipient. Assuming a collision resistant hash function, an unforgeable signature algorithm and an unforgeable one-time MAC function, the proof of recipient

cannot be forged. A proof of recipient consists of a proof of author, a public key $\texttt{pk}$, and an event key k'. The proof of author fixes the event, which consists of an event identifier e^{ID} and an event payload e^{P}. Now that the output of MAC function is fixed by the event identifier $e^{ID} = \texttt{MAC}_{k'}(\texttt{pk})$, for the adversary to come up with a different $\texttt{pk}$ and k', it has to break the unforgeability of the one-time MAC function.

Message. Assuming a collision and pre-image resistant hash function, an unforgeable signature algorithm and an unforgeable one-time MAC function, the proof of message cannot be forged. From the proof of message, the proof of recipient can be derived by computing the event key $k' \leftarrow \texttt{Hash}(n)$. The proof of recipient fixes the payload e^{P}, the recipient's public key $\texttt{pk}$ and the nonce n, since the prover provided a pre-image to k'. The payload consists of the ciphertext c and the ephemeral public key $\texttt{pk}'$, which also fixes the corresponding $\texttt{sk}'$. The prover provides $\texttt{sk}'$, which can easily be verified to be correct. This fixes all the input to our deterministic decryption function.

6 Related Work

In the setting of transparency logging, we build further upon the model and scheme by Pulls *et al.* [17] and Balloon [16] as introduced before. The scheme by Pulls *et al.* is based on hash- and MAC-chains, influenced by the secure log design of Schneier and Kelsey [18].

Ma and Tsudik [12] proposed a publicly verifiable FssAgg scheme by using an efficient aggregate signature scheme. The main drawbacks are a linear number of verification keys with the number of runs of the key update, and relative expensive bilinear map operations. Similarly, Logcrypt by Holt [11] also needs a linear number of verification keys with key updates. The efficient public verifiability, of both the entire Balloon and individual events, of Insynd comes from taking the same approach as (and building upon) the History Tree system by Crosby and Wallach [8] based on authenticated data structures. The main drawback of the work of Crosby and Wallach, and to a lesser degree of Insynd, is the reliance on a gossiping mechanism. Insynd takes the best of both worlds: the public verifiability from authenticated data structures based on Merkle trees, and the private all-or-nothing verifiability of the privately verifiable FssAgg scheme from the secure logging area. Users do not have to rely on perfect gossiping of snapshots, while the existence of private verifiability for recipients deters an adversary from abusing the lack of a perfect gossiping mechanism to begin with. This is similar to the approach of CONIKS [15], where users can verify their entries in a data structure as part of a privacy-friendly key management system. In CONIKS, users provide all data (their public key and related data) in the data structure concerning them. This is fundamentally different to Insynd, where the entire point of the scheme is for the author to inform recipients of the processing performed on their personal data. Therefore, the private verifiability mechanism for Insynd needs to be forward-secure with regard to the author.

PillarBox is a fast forward-secure logging system by Bowers *et al.* [6]. Beyond integrity protection, PillarBox also provides a property referred to as "stealth" that prevents a forward-secure adversary from distinguishing if any messages are inside an encapsulated buffer or not. This indistinguishability property is similar to our forward unlinkability of events property. PillarBox has also been designed to be fast with regard to securing logged messages. The goal is to minimise the probability that an adversary that compromises a system will be able to shut down PillarBox before the events that (presumably) were generated as a consequence of the adversary compromising the system are secured.

Pond and WhisperSystem's Signal[7] are prime examples of related secure asynchronous messaging systems. While these systems are for two-way communication, there are several similarities, such as dedicated servers for storing encrypted messages. Both Pond and Signal use the Signal protocol (previously known as Axolotl) [19]. The Signal protocol is inspired by the Off-the-Record (OTR) Messaging protocol [5] and provides among other things forward secrecy. Note that the goal of Insynd is for messages to be non-repudiable, unlike Pond, Signal and OTR that specifically want *deniability*. Insynd achieves non-repudiation through the use of Balloon and how we encrypt messages.

7 Performance

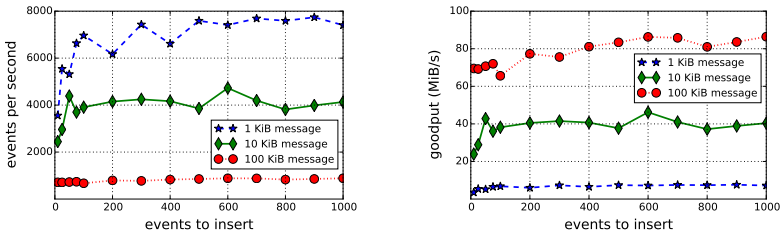
We implemented Insynd in the Go programming language, making use of the NaCl [4] library for the cryptographic building blocks. The performance benchmark focuses on the `insert` protocol since the other protocols are less frequently used. The source code and steps to reproduce our benchmark are publicly available at `http://www.cs.kau.se/pulls/insynd/`.

Figure 4 presents our benchmark, based on averages after 10 runs using Go's built-in benchmarking tool. We used a Debian 7.8 (x64) installation on a laptop with an Intel i5-3320M quad core 2.6 GHz CPU and 7.7 GB DDR3 RAM to run both the author and server. Note that the proofs of correct insertion into Balloon between author and server are still generated and verified.

Clearly, the smaller the message are, the more events can be sent (and the more potential recipients that can be served) per second. With at least 100 events to insert per run, we get ≈7000 events per second with 1 KiB messages. Using the same data as in Figure 4a, Figure 4b shows the goodput (the throughput excluding the event overhead of 112 bytes per event) for the different message sizes. At ≈800 100-KiB-messages per second (around at least 200 events to insert), the goodput is ≈80 MiB/s. 10 KiB messages offer a trade-off between goodput and number of events, providing 4000 events per second with ≈40 MiB/s goodput.

Insynd improves greatly on related work on transparency logging, and shows comparable performance to state-of-the-art secure logging systems. Ma and Tsudik[12], for their FssAgg schemes, achieve event generation (signing) in the order of milliseconds per event (using significantly older hardware than us). Marson and Poettering [13], with their seekable sequential key generators, generate

[7] `https://whispersystems.org`, accessed 2016-07-06.



(a) Events per second in a 2^{20} Balloon. (b) Goodput in a 2^{20} Balloon.

Fig. 4: A performance benchmark related to inserting events. The x-axis specifies the number of events to insert per run of the `insert` protocol.

key material in a few microseconds. Note that for both these schemes, messages are not encrypted and hence the performance results only take into account the time for providing integrity protection. The performance results of Insynd, together with the two following schemes, include the time to encrypt messages in addition to providing integrity protection. Pulls *et al.* [17], for their transparency logging scheme, generate events in the order of tens of milliseconds per event. For PillarBox, Bowers *et al.* [6] generate events in the order of hundreds of microseconds per event, specifying an average time for event generation at 163 μs when storing syslog messages. Syslog messages are at most 1 KiB, so the average for Insynd of 142 μs at 7000 events per second is comparable.

8 Conclusions

Insynd is a cryptographic scheme for privacy-preserving transparency logging where messages are sent through an authenticated data structure (Balloon). The main contribution of Insynd is to provide publicly verifiable proofs of recipient and message of events within the setting of transparency logging, which dictates that events should be encrypted and unlinkable towards non-recipients. This significantly increases the utility of a transparency logging scheme as it enables users to take action without having to disclose everything that was logged for them. On top of this, Insynd improves further on existing transparency logging schemes by combining concepts from authenticated data structures, forward-secure key generation from the secure logging area, and on-going work on secure messaging protocols. Insynd provably achieves the security and privacy properties for a transparency logging scheme, as defined within the general framework of Pulls *et al.* [17], which was adjusted to take into account our publicly verifiable proofs and stronger adversarial model that assumes a forward-secure author and an untrusted server. Furthermore, our freely available proof of concept implementation shows that Insynd offers comparable performance for event generation to state-of-the-art secure logging systems like PillarBox [6].

Acknowledgements We would like to thank Rasmus Dahlberg, Simone Fischer-Hübner, Stefan Lindskog, and Leonardo Martucci for their valuable feedback. Tobias Pulls has received funding from the Seventh Framework Programme for Research of the European Community under grant agreement no. 317550 and the HITS research profile funded by the Swedish Knowledge Foundation.

References

1. An, J.H.: Authenticated encryption in the public-key setting: Security notions and analyses. IACR Cryptology ePrint Archive 2001, 79 (2001)
2. Bellare, M., Boldyrevay, A., Desaiz, A., Pointcheval, D.: Key-Privacy in Public-Key Encryption. In: ASIACRYPT. LNCS, vol. 2248, pp. 566–582. Springer (2001)
3. Bellare, M., Yee, B.S.: Forward-Security in Private-Key Cryptography. In: CT-RSA. LNCS, vol. 2612, pp. 1–18. Springer (2003)
4. Bernstein, D.J., Lange, T., Schwabe, P.: The Security Impact of a New Cryptographic Library. In: LATINCRYPT. LNCS, vol. 7533. Springer (2012)
5. Borisov, N., Goldberg, I., Brewer, E.A.: Off-the-record communication, or, why not to use PGP. In: WPES. pp. 77–84. ACM (2004)
6. Bowers, K.D., Hart, C., Juels, A., Triandopoulos, N.: PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging. In: Research in Attacks, Intrusions and Defenses Symposium. LNCS, vol. 8688, pp. 46–67. Springer (2014)
7. Buldas, A., Laud, P., Lipmaa, H., Willemson, J.: Time-Stamping with Binary Linking Schemes. In: CRYPTO. LNCS, vol. 1462, pp. 486–501. Springer (1998)
8. Crosby, S.A., Wallach, D.S.: Efficient Data Structures For Tamper-Evident Logging. In: USENIX Security Symposium. pp. 317–334. USENIX (2009)
9. Dingledine, R., Mathewson, N., Syverson, P.F.: Tor: The Second-Generation Onion Router. In: USENIX Security Symposium. pp. 303–320. USENIX (2004)
10. FIDIS WP7: D 7.12: Behavioural Biometric Profiling and Transparency Enhancing Tools. Future of Identity in the Information Society (March 2009)
11. Holt, J.E.: Logcrypt: forward security and public verification for secure audit logs. In: Australasian Workshops on Grid Computing and e-Research. ACS (2006)
12. Ma, D., Tsudik, G.: A new approach to secure logging. TOS 5(1) (2009)
13. Marson, G.A., Poettering, B.: Even more practical secure logging: Tree-based seekable sequential key generators. In: ESORICS 2014. LNCS, Springer (2014)
14. Mazières, D., Shasha, D.: Building secure file systems out of byzantine storage. In: Symposium on Principles of Distributed Computing. pp. 108–117. ACM (2002)
15. Melara, M.S., Blankstein, A., Bonneau, J., Felten, E.W., Freedman, M.J.: CONIKS: A Privacy-Preserving Consistent Key Service for Secure End-to-End Communication. In: USENIX Security Symposium. pp. 383–398. USENIX (2015)
16. Pulls, T., Peeters, R.: Balloon: A forward-secure append-only persistent authenticated data structure. In: ESORICS. LNCS, Springer (2015)
17. Pulls, T., Peeters, R., Wouters, K.: Distributed privacy-preserving transparency logging. In: WPES. pp. 83–94. ACM (2013)
18. Schneier, B., Kelsey, J.: Cryptographic Support for Secure Logs on Untrusted Machines. In: USENIX Security Symposium. pp. 53–62. USENIX (1998)
19. Unger, N., Dechand, S., Bonneau, J., Fahl, S., Perl, H., Goldberg, I., Smith, M.: Sok: Secure messaging. In: 2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015. IEEE Computer Society (2015)

A Security and Privacy Model

An adversary $\mathcal{A}$ can adaptively control the scheme through a set of oracles. The adversary has access to a base set of oracles $\mathcal{O}_{base}$:

- $\texttt{pk} \leftarrow \texttt{CreateRecipient}()$: Generates a new public key $\texttt{pk} \leftarrow \texttt{crypto\_box\_keypair}(\texttt{sk})$ to identify a recipient, registers the recipient pk at the author A, and finally returns pk.
- $e \leftarrow \texttt{CreateEvent}(\texttt{pk}, m)$: Creates an event e with message m at the author A for the recipient pk registered at A. Returns e.
- $(\texttt{State}, \#events) \leftarrow \texttt{CorruptAuthor}()$: Returns the entire state of the author and the number of created events before calling this oracle.

Additional oracles that are specific to a property will be defined when defining the properties. For properties with the prefix "forward", the adversary is not allowed to make any further queries after it makes a call to CorruptAuthor.

A.1 Secrecy

For the security experiment, the adversary can access the CreateEvent* oracle once for the challenge bit b:

- $e \leftarrow \texttt{CreateEvent*}(\texttt{pk}, m_0, m_1)_b$: Creates an event e with message m_b at the author A for the recipient pk registered at A. Returns e. Note that this oracle can only be called once with distinct messages m_0 and m_1 of the same length.

We also need to model that the adversary can get hold of side information through publicly verifiable proofs of message (or possibly the stored data at the author to be able to make these proofs). For this reason we provide the adversary with a decryption oracle:

- $m, \Pi \leftarrow \texttt{DecryptEvent}(e)$: Returns the message m and a proof Π of an event e with the restriction that e was outputted by the CreateEvent oracle.

The security (SE) experiment is:

$\mathbf{Exp}^{SE}_{\mathcal{A}}(k)$:

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathcal{A}^{\mathcal{O}_{base},\texttt{CreateEvent*},\texttt{DecryptEvent}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as:

$$\mathbf{Adv}^{SE}_{\mathcal{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{SE}_{\mathcal{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{SE}_{\mathcal{A}}(k) = 1 | b = 1\right] - 1 \right| .$$

Definition 1. *A scheme provides computational secrecy of the data contained within events, if and only if for all polynomial time adversaries $\mathcal{A}$, it holds that* $\mathbf{Adv}^{SE}_{\mathcal{A}}(k) \leq \epsilon(k)$.

A.2 Deletion-Detection Forward Integrity

For deletion-detection forward integrity, a helper algorithm $\texttt{valid}(e, i)$ is defined. This algorithm returns whether or not the full log trail for every recipient verifies when e_i (the event e created at the i-th call of `CreateEvent`) is replaced by e.

For completeness, we modified the model to also take into account that the adversary might benefit from requesting the (encrypted) current state for a recipient at the author:

- $c \leftarrow \texttt{GetState}(\texttt{pk}, n)$: Returns the ciphertext c that is generated as the reply to the `getState` protocol for the recipient `pk` and nonce n.

We also need to model that the adversary can get hold of side information through publicly verifiable proofs of message (or possibly the stored data at the author to be able to make these proofs). For this reason we provide the adversary with a decryption oracle (as for secrecy) and a RecipientEvent oracle:

- $\texttt{pk}, \Pi \leftarrow \texttt{RecipientEvent}(e)$: Returns the recipient `pk` and a proof Π of an event e with the restriction that e was outputted by the `CreateEvent` oracle.

The forward integrity (FI) experiment is defined as:

$\mathbf{Exp}^{FI}_{\mathcal{A}}(k)$:

1. $l \leftarrow \mathcal{A}^{\mathcal{O}_{base},\texttt{GetState},\texttt{DecryptEvent},\texttt{RecipientEvent}}()$
2. Return $e \neq e_i \wedge \texttt{valid}(e, i) \wedge i \leq \#events$.

The advantage of the adversary is defined as:

$$\mathbf{Adv}^{FI}_{\mathcal{A}}(k) = Pr\left[\mathbf{Exp}^{FI}_{\mathcal{A}}(k) = 1\right].$$

Definition 2. *A scheme provides computational forward integrity, if and only if for all polynomial time adversaries $\mathcal{A}$, it holds that $\mathbf{Adv}^{FI}_{\mathcal{A}}(k) \leq \epsilon(k)$.*

Definition 3. *A scheme provides computational deletion-detection forward integrity, if and only if it is FI secure and the verifier can determine, given the output of the scheme and the time of compromise, whether any prior events have been deleted.*

A.3 Forward Unlinkability of Events

Since the adversary is (presumably) in control of the server, it can trivially tell in which round events were added. Therefore, the adversary can only create recipients before each round of the `insert` protocol and is limited to finding a relationship between two events within one round. The following oracles are defined:

- $vuser \leftarrow \texttt{DrawRecipient}(\texttt{pk}_i, \texttt{pk}_j)$: Generates a virtual user reference, as a monotonic counter, *vrecipient* and stores $(vrecipient, \texttt{pk}_i, \texttt{pk}_j)$ in a table $\mathcal{D}$. If $\texttt{pk}_i$ is already referenced as the left-side user in $\mathcal{D}$ or $\texttt{pk}_j$ as the right-side user, then this oracle returns $\perp$ and adds no entry to $\mathcal{D}$. Otherwise, it returns *vuser*.

– $\texttt{Free}(vrecipient)$: Removes the triple $(vrecipient, \texttt{pk}_i, \texttt{pk}_j)$ from table $\mathcal{D}$.
– $e \leftarrow \texttt{CreateEvent'}(vrecipient, m)_b$: Creates an event e with message m at the author A for a recipient registered at A. Which recipient depends on the value b and the event $vrecipient$ in the table $\mathcal{D}$. Returns e.

As for deletion-detection forward integrity, we also allow the adversary access to the `GetState`, `DecryptEvent`, `RecipientEvent` oracles. The experiment for forward unlinkability of events (FU) is defined as:

$\mathbf{Exp}^{FU}_{\mathcal{A}}(k)$:
1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathcal{A}^{\mathcal{O}_{base},\texttt{DrawUser},\texttt{Free},\texttt{CreateEvent'},\texttt{GetState},\texttt{DecryptEvent},\texttt{RecipientEvent}}()$
3. Return $g \stackrel{?}{=} b$.

The advantage of the adversary is defined as

$$\mathbf{Adv}^{FU}_{\mathcal{A}}(k) = \frac{1}{2} \cdot \left| Pr\left[\mathbf{Exp}^{FU}_{\mathcal{A}}(k) = 1 | b = 0\right] + Pr\left[\mathbf{Exp}^{FU}_{\mathcal{A}}(k) = 1 | b = 1\right] - 1 \right| .$$

Definition 4. *A scheme provides computational forward unlinkability of events, if and only if for all polynomial time adversaries $\mathcal{A}$, it holds that $\mathbf{Adv}^{FU}_{\mathcal{A}}(k) \leq \epsilon(k)$.*

B Balloon

Detailed descriptions of the algorithms from Balloon [16] that are used by Insynd:

– $(\texttt{sk}, \texttt{pk}) \leftarrow \texttt{B.genkey}(1^\lambda)$: On input of a security parameter λ, outputs a secret key `sk` and public key `pk`.
– $(\texttt{auth}(D_0), \texttt{s}_0) \leftarrow \texttt{B.setup}(D_0, \texttt{sk}, \texttt{pk})$: On input of a (plain) data structure D_0, `sk` and `pk`, computes the authenticated data structure $\texttt{auth}(D_0)$ and the corresponding snapshot $\texttt{s}_0$.
– $(D_{h+1}, \texttt{auth}(D_{h+1}), \texttt{s}_{h+1}) \leftarrow \texttt{B.refresh}(u, D_h, \texttt{auth}(D_h), \texttt{s}_h, \texttt{upd}, \texttt{pk})$: On input of an update u on the data structure D_h, the authenticated data structure $\texttt{auth}(D_h)$, the snapshot $\texttt{s}_h$, relative information `upd` and `pk`, outputs the updated data structure D_{h+1} along with the updated authenticated data structure $\texttt{auth}(D_{h+1})$ and the updated snapshot $\texttt{s}_{h+1}$.
– $(\Pi(q), \alpha(q)) \leftarrow \texttt{B.query}(q, D_h, \texttt{auth}(D_h), \texttt{pk})$ (**Membership**): On input of a membership query q on data structure D_h, $\texttt{auth}(D_h)$ and `pk`, returns the answer $\alpha(q)$ to the query, along with proof $\Pi(q)$.
– $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{B.verify}(q, \alpha, \Pi, \texttt{s}_h, \texttt{pk})$ (**Membership**): On input of a membership query q, an answer α, a proof Π, a snapshot $\texttt{s}_h$ and the public key `pk`, outputs either `accept` or `reject`.
– $(\Pi(q), \alpha(q)) \leftarrow \texttt{B.query}(q, D_h, \texttt{auth}(D_h), \texttt{pk})$ (**Prune**): On input of a prune query q on data structure D_h, $\texttt{auth}(D_h)$ and `pk`, returns the answer $\alpha(q)$ to the query, along with proof $\Pi(q)$.

– $\{\texttt{accept}, \texttt{reject}\} \leftarrow \texttt{B.verify}(q, \alpha, \Pi, \texttt{s}_h, \texttt{pk})$ (**Prune**): On input of a prune query q, an answer α, a proof Π, $\texttt{s}_h$ and $\texttt{pk}$, outputs either $\texttt{accept}$ or $\texttt{reject}$.
– $(\texttt{s}_{h+1}, \texttt{upd}) \leftarrow \texttt{B.update*}(u, \Pi, \texttt{s}_h, \texttt{sk}, \texttt{pk})$: On input of an update u for the (authenticated) data structure fixed by $\texttt{s}_h$ with an accepted verified prune proof Π, $\texttt{sk}$ and $\texttt{pk}$, outputs the updated snapshot $\texttt{s}_{h+1}$, that fixes an update of the data structure D_{h+1} along with the updated authenticated data structure $\texttt{auth}(D_{h+1})$ using u, and some relative information $\texttt{upd}$.