Insynd

Privacy-Preserving Transparency Logging Using Balloons

Tobias Pulls
Dept. of Mathematics and Computer Science
Karlstad University
Universitetsgatan 2, Karlstad, Sweden
tobias.pulls@kau.se

Roel Peeters
KU Leuven, ESAT/COSIC & iMinds
Kasteelpark Arenberg 10 bus 2452
3001 Leuven, Belgium
roel.peeters@esat.kuleuven.be

ABSTRACT

Insynd is a cryptographic scheme for privacy-preserving transparency logging. In the setting of transparency logging, a service provider continuously logs descriptions of its data processing on its users' personal data, where each description is intended for a particular user. Our work focuses on protecting the privacy of users. Insynd provides secrecy of messages, message integrity and authenticity, protection against recipient profiling, and publicly verifiable proofs of who sent what message to which recipient at what particular time. Our scheme is built on an authenticated data structure (Balloon) that enables the safe outsourcing of storage of messages to an untrusted server (such as commodity cloud services). The author of messages is in the forward-security model. Insynd provides stronger privacy protections than prior work in this setting, improved efficiency in terms of event generation, and increases the utility of all data sent through the scheme thanks to the publicly verifiable proofs. Our prototype implementation shows greatly improved performance over related work and competitive performance for more data-intensive settings like secure logging.

1. INTRODUCTION

The concept of *transparency logging* for transparency-enhancing tools (TETs) has its roots in the Future of Identity in the Information Society (FIDIS) network [13], funded by the EU's 6th Framework Programme. For TETs to perform efficient *counter profiling* and successfully anticipate consequences of personal data processing, these need insight into the actual data processing being done by service providers. Transparency logging is a TET for transporting processing data from service providers to the users of that service. The processing data describe the processing on the user's personal data, where the users are data subjects as defined in the EU Data Protection Directive (DPD) 95/46/EC.

Figure 1 shows one use case of transparency logging, where a user Alice discloses personal data to a service provider (SP). The service provider provides a service, during which personal data is processed according to the privacy policy. The privacy policy should be provided to Alice prior to her data disclosure to the SP, so that she can provide *informed consent* to the data processing by the SP. For transparency logging, we view each system that processes data (in the technical sense and not necessarily in the legal sense as defined in, e.g., the DPD) as an *author*. The author generates *events* intended for *recipients*, like Alice, that describe data processing as it takes place. Events are stored at a *server*, an intermediate party that primarily serves to offload storage of events for authors. Recipients can then retrieve their events from the server and compare the actual data processing as described in the events with the stated processing in the privacy policy. Conceptually, this enables Alice to detect any inconsistencies and in a sense hold the service provider accountable for its actions.

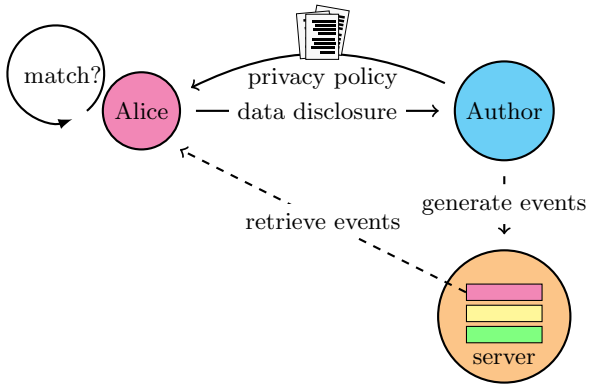


Figure 1: Comparing actual data processing of personal data with the data processing that was agreed upon in the privacy policy, prior to data disclosure by the user.

This paper focuses on how authors should generate these events for transparency logging in a secure and privacy-preserving way. We do *not* consider what should be logged to describe data processing, or how privacy policies should be structured to enable the comparison with stated data processing. Our primary goal is to construct a privacy-preserving transparency logging scheme, improving on prior work by Pulls *et al.* [21]. Our contributions are:

Stronger privacy protections Servers need not to be trusted in Insynd, compared to forward secure in prior work by Pulls *et al.* [21].

Publicly verifiable consistency Anyone can verify that snapshots, created as new events are generated, make consistent claims about the past (no events have been modified or deleted).

Increased utility Recipients and authors can produce publicly verifiable proofs of all data sent through Insynd, convincing a third-party of who sent a particular message to whom at approximately what time.

More efficient event generation By using modern cryptographic primitives, Insynd generates events at speeds comparable to state-of-the-art secure logging schemes.

The rest of this paper is structured as follows. Section 2 briefly states our assumptions and goals. Section 3 gives a high-level overview of our ideas and necessary building blocks. Section 4 presents the Insynd scheme. Section 5 evaluates Insynd's properties. Section 6 describes extensions to Insynd, enabling distributed settings, a stronger adversary model and privacy-preserving downloads. Section 7 presents related work. Section 8 shows the performance of our proof-of-concept implementation.

2. ASSUMPTIONS AND GOALS

In the framework by Pulls *et al.* [21], *all* authors and servers are modeled in the forward security model. All entities are assumed initially trusted up to a point in time t when *all* entities are assumed compromised simultaneously by an attacker. While some trust in authors is inevitable, since authors generate the descriptions of their own processing, servers should not necessarily have to be trusted. The forward security at the server in [21] is the result of their scheme requiring the server to keep a cryptographic state for the recipients. In Insynd, servers only provide storage of events. Hence, we support a stronger adversarial model where authors are assumed to be forward secure and servers do not have to be trusted.

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 [12]) 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 [8].

2.1 Core Security and Privacy Properties

For defining the core security and privacy properties of our transparency logging scheme, we adopt and modify the general model and definitions by Pulls *et al.* [21]. Their paper defines secrecy of events, deletion-detection forward integrity, forward unlinkability of recipient events, and unlinkability of recipient identifiers in the case of distributed settings. Our modifications (presented in Sect. 5) are primarily due to our stronger adversarial model.

2.2 Public Verifiability and Proofs

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 the author of an event, the recipient of an event, the message sent in an event, and 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 message secrecy and forward unlinkability of events.

3. IDEAS

To support a setting where the author does not trust the server, we make use of an authenticated data structure to store events at the server. We opted for Balloon [20], an authenticated data structure that was designed for the setting of transparency logging with an untrusted server. An authenticated data structure in this setting should allow for efficient publicly verifiable proofs of both membership and non-membership of keys. Otherwise, a recipient cannot distinguish between a server denying service and the lack of an event with a specific key. The main advantage of Balloon compared to other authenticated data structures that have this property (for a more in-depth discussion, we refer the reader to [20]), 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. An overview of Balloon is given in Sect. 3.1.

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 not only the event identifiers but the entire events must be unlinkable to each other to prevent information leaks due to event correlation. Therefore, 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. 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 event, the author updates the symmetric linking key for the recipient in question using a forward-secure sequential key generator (forward-secure SKG) in the form of an evolving hash chain [3, 22, 16]. The recipient can do the same to link the relevant event identifiers together.

To ensure that the author cannot change or delete any past events after being compromised (forward-secure author), we rely on the author keeping a 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. For each recipient, the current values of the forward-secure SKG and the forward-secure sequential aggregate authenticator (FssAgg) [15] over the relevant event values are kept in the author's state. Note that Balloon in itself provides forward integrity and deletion detection over the entire data structure and is not limited to a specific recipient, however this assumes a perfect gossiping mechanism for snapshots [20].

To provide the publicly verifiable proofs we need to go into the details of how the snapshots of the authenticated data structure are generated and how descriptions for recipients are encrypted. Details on the used encryption scheme are presented in Sect. 3.2.

To ease implementation, Insynd is designed around the use of NaCl [5]. The NaCl library provides all of the core operations needed to build higher-level cryptographic tools and provides state of the art security (also taking into account side-channels by having no data dependent branches, array indices or dynamic memory allocation) and high speed implementations.

3.1 Balloon

Balloon is a forward-secure append-only persistent authenticated data structure tailored to the transparency logging setting by Pulls and Peeters [20]. Insynd makes use of the following algorithms on Balloon:

- $(\texttt{sk}, \texttt{pk}) \leftarrow \texttt{B.genkey}(1^\lambda)$: 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 pk, outputs either accept or 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, sk and 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 upd.

Balloon relies upon a pre-image and collision resistant hash function, and an unforgeable signature algorithm. 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 Balloon and compare the calculated snapshots with those gossiped by the author.

3.2 Encryption

For encrypting messages to recipients, we use NaCl's crypto_box [5]. However, when encrypting a message, the author generates a fresh ephemeral key pair $pk' \leftarrow \texttt{crypto\_box\_keypair}(sk')$ to seal crypto_box and the message to be encrypted is appended with the ephemeral private key $m' = m||sk'$. The nonce n used for encryption/decryption is provided to the author/recipient. These modifications allow us to efficiently support the forward secure author setting, and help us to efficiently generate publicly verifiable proofs of message as will be shown later on. We define the following high level functions for a recipient with an encryption key pair $(\texttt{pk}, \texttt{sk})$:

- $(c, \texttt{pk}') \leftarrow \texttt{Enc}^n_{\texttt{pk}}(m)$: Encrypts a message m using an ephemeral key-pair $\texttt{pk}' \leftarrow \texttt{crypto\_box\_keypair}(\texttt{sk}')$, the public key pk, and the nonce n where the ciphertext $c = \texttt{crypto\_box}m||\texttt{sk}', n, \texttt{pk}, \texttt{sk}')$. Returns $(c, \texttt{pk}')$.
- $(m, \texttt{sk}') \leftarrow \texttt{Dec}^n_{\texttt{sk}}(c, \texttt{pk}')$: Decrypts ciphertext c using the private key sk, public key $\texttt{pk}'$, and nonce n where $p \leftarrow \texttt{crypto\_box\_open}(c, n, \texttt{pk}', \texttt{sk})$. If decryption fails $p \stackrel{?}{=} \perp$, otherwise $p \stackrel{?}{=} m||\texttt{sk}'$. Returns p.
- $m \leftarrow \texttt{Dec}^n_{\texttt{sk}',\texttt{pk}}(c, \texttt{pk}')$: Decrypts ciphertext c using the private key $\texttt{sk}'$, public key pk, and the nonce n where $p \leftarrow \texttt{crypto\_box\_open}(c, n, \texttt{pk}, \texttt{sk}')$. If decryption fails $p \stackrel{?}{=} \perp$, otherwise $p \stackrel{?}{=} m||\texttt{sk}^*$. If $\texttt{sk}' \stackrel{?}{=} \texttt{sk}^*$ and $\texttt{pk}' \stackrel{?}{=} \texttt{pk}^*$, where $\texttt{pk}^* \leftarrow \texttt{crypto\_box\_keypair}(\texttt{sk}')$, returns m, otherwise $\perp$.

Since we generate a random key-pair for every encryption, the scheme does not retain any meaningful *public-key authenticated encryption* on its own. Our construction for encryption and decryption is very similar to the DHETM (Diffie-Hellman based Encrypt-then-MAC) construction by An [1], which is proven IND-CCA2 secure. The differences are that we use Curve25519, instead of a random DH group, and a symmetric authenticated key cipher based on crypto_secretbox as part of NaCl.

4. INSYND

Now we will go into the details of the different protocols that make up Insynd. Figure 2 shows the five protocols that make up Insynd 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, $(\mathsf{A}_{\texttt{sk}}, \mathsf{A}_{\texttt{vk}})$ and $(\mathsf{S}_{\texttt{sk}}, \mathsf{S}_{\texttt{vk}})$, respectively. We assume that $\mathsf{A}_{\texttt{vk}}$ and $\mathsf{S}_{\texttt{vk}}$ 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.

4.1.1 Author-Server Setup

The purpose of the setup protocol (pink box in Figure 2) is for the author and the server to create a new Balloon, stored at the server, with two associated uniform resource identifiers: one for the author $\mathsf{A}_{\texttt{URI}}$, and one for the server $\mathsf{S}_{\texttt{URI}}$. At the former the recipient can later-on query for its current state, while at the latter it can retrieve stored events. The 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 who sends $\mathsf{A}_{\texttt{URI}}$ to the server. $\mathsf{A}_{\texttt{URI}}$ specifies the URI of the state associated

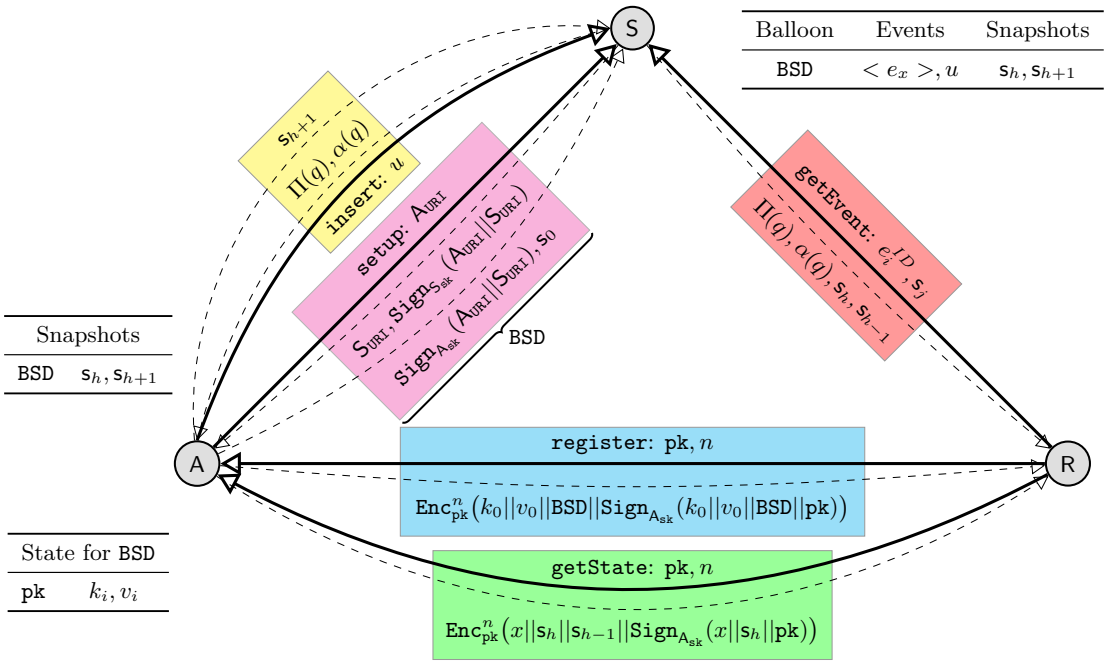


Figure 2: The Insynd scheme, consisting of five protocols (coloured boxes), between a author A, a server S, and a recipient R. A solid line indicates the start of protocol and a dashed line a response. Section 4 describes each protocol in detail.

with the to-be created Balloon at the author. After receiving $\mathsf{A}_{\mathsf{URI}}$, the server verifies that $\mathsf{A}_{\mathsf{URI}}$ identifies a resource under the control of the author. If so, the server signs this URI together with the URI to the new Balloon at the server, $\mathsf{S}_{\mathsf{URI}}$. The server replies with $\mathsf{S}_{\mathsf{URI}}$ and $\mathtt{Sign}_{\mathsf{S}_{\mathsf{sk}}}(\mathsf{A}_{\mathsf{URI}}||\mathsf{S}_{\mathsf{URI}})$. The signature commits the server to the specified Balloon. Upon receiving the reply from the server, the author verifies that $\mathsf{S}_{\mathsf{URI}}$ is under the server's control and the signature. If both verify, the author creates an empty Balloon $(\mathtt{auth}(D_0), \mathsf{s}_0) \leftarrow \mathtt{B.setup}(D_0, \mathsf{A}_{\mathsf{sk}}, \mathsf{A}_{\mathsf{vk}})$ for an empty data structure D_0. The author sends its signature on the two URIs $\mathtt{Sign}_{\mathsf{sk}}(\mathsf{A}_{\mathsf{URI}}||\mathsf{S}_{\mathsf{URI}})$ 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 authors signature and can finish the setup of the empty balloon now that it has s_0. The two signatures, two URIs, and the initial snapshot s_0 together form the BSD.

4.1.2 Recipient Registration

The purpose of the register protocol (blue box in Figure 2) 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 (pk, sk) according to the agreed upon encryption scheme: $\mathsf{pk} \leftarrow \mathtt{crypto\_box\_keypair}(\mathsf{sk})$.

The protocol is initiated by the recipient sending its public key together with a nonce to the author. The author verifies that the provided public key is a valid public key according to the agreed upon encryption scheme.[1] If so, 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, the Balloon setup data BSD, and the following signature: $\mathtt{Sign}_{\mathsf{A}_{\mathsf{sk}}}(k_0||v_0||\mathtt{BSD}||\mathsf{pk})$. 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 using the provided 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 event identifier and payload in Insynd correspond to the event key and value in Balloon.

The nonce n, used in 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):

$$n \leftarrow \mathtt{Hash}(1||k) \quad (1)$$

$$k' \leftarrow \mathtt{Hash}(n) \quad (2)$$

[1] Specifically for the used encryption scheme, the author should validate that it received a 32 byte value.

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

$$k_i \leftarrow \texttt{Hash}(k_{i-1}) \tag{3}$$

Figure 3 visualises the derivation of these different values.

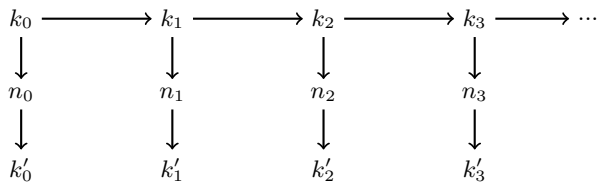


Figure 3: Derivation of the nonce n and event key k' from the authentication key k.

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

$$e^{ID} \leftarrow \texttt{MAC}_{k'}(\texttt{pk}) \tag{4}$$

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: $e^P \leftarrow \texttt{Enc}^n_{\texttt{pk}}(m)$. 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 a FssAgg [15]:

$$v_i \leftarrow \texttt{Hash}\big(v_{i-1}||\texttt{MAC}_{k_{i-1}}(e)\big) \tag{5}$$

Then the recipient's current authentication key is updated, by using Equation 3.

4.2.1 Insert

The purpose of the insert protocol (yellow box in Figure 2) 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. For this protocol we make extensive use of the algorithms defined for a Balloon (see Sect. 3.1).

Upon recieving 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 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.

4.2.2 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 [18], which works in a setting closely related to ours and links snapshots together into a snapshot chain. We redefine a snapshot as:

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

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. 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. Since snapshots are both linked and occasionally timestamped, this greatly restricts adversaries in the forward-security model.

4.2.3 The Last Event

When the author no longer wishes to be able to send messages to a recipient, the author creates one last event with the message set to a stop marker M_s and the recipient's current authenticator value v from state. The stop marker should be globally known, and have length $|M_s| = 2|\texttt{Hash}(\cdot)|$ to match the length of (k, v). The stop marker and authenticator are necessary for enabling recipients to distinguish between the correct stop of sending messages and modifications by an attacker. After the event containing the stop marker and authenticator value has been inserted, the author removes the recipient from its BSD state table.

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.

4.3.1 Getting Events

The purpose of the getEvent protocol (red box in Figure 2) 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. For this protocol we make use of the algorithms defined for a Balloon (see Sect. 3.1). Before running this protocol, the recipient generates the event identifier it is interested in, by using

equations 1-4 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 snapshot $\mathtt{s}_h$ 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:

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

4.3.2 Getting State

The `getState` protocol (green box in Figure 2) 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 check if its state table for the `BSD` in question[2] contains an entry for `pk`. In case there is an entry, the author sets $x \leftarrow (k_i, v_i)$, where k_i is the current authentication key and v_i the current authenticator value. Otherwise, the author sets $x \leftarrow M_s$. Recall that M_s has the same length as (k_i, v_i). 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 (and the length of the plaintext fixed), no third party in possession of the recipient's public key can determine if new events are generated for the recipient. The nonce ensures *freshness* of the reply.

4.3.3 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. If these are not identical, the recipient should rerun `getEvent` for the last event identifier and continue until it receives a non-membership proof to ensure that no events were inserted by the author in the meanwhile.

[2]Based upon which URI at the author, $\mathsf{A}_{\mathtt{URI}}$, the request was sent to, the author determines which `BSD`.

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. Steps 1–8 decrypt all events using the nonce, event key, and authentication key generation determined by equations 1-3. If a stop marker (M_s) is found (steps 6–7), the authenticator v' in the last event should match the calculated authenticator v and x must match M_s. Otherwise, if no stop marker are found, the x should match the calculated state in the form of (k, v) (step 9).

Algorithm 1 Verify message consistency for a recipient.

Require: $\mathtt{pk}, \mathtt{sk}, k_0, v_0$, 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: $k' \leftarrow \mathtt{Hash}(n), n \leftarrow \mathtt{Hash}(1||k), k \leftarrow k_0, v \leftarrow v_0$ ▷ k' is the event key, n the event nonce, k and v the calculated state
2: **for all** $e \in l$ **do** ▷ in the order events were inserted
3: $p \leftarrow \mathtt{Dec}^n_{\mathtt{sk}}(e^P)$
4: **if** $p \stackrel{?}{=} \perp$ **then**
5: **return** `false` ▷ failed to decrypt event
6: **if** p contains (M_s, v') **then** ▷ check for M_s, v' unknown
7: **return** $x \stackrel{?}{=} M_s \wedge v \stackrel{?}{=} v'$ ▷ no state and matching authenticator
8: $k' \leftarrow \mathtt{Hash}(n), n \leftarrow \mathtt{Hash}(1||k), k \leftarrow \mathtt{Hash}(k), v \leftarrow \mathtt{Hash}\big(v||\mathtt{MAC}_k(e)\big)$ ▷ calculated from right to left
9: **return** $x \stackrel{?}{=} (k, v)$ ▷ state should match calculated state

4.4 Publicly Verifiable Proofs

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 proof 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).

4.4.1 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**).

4.4.2 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.

4.4.3 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 $\tilde{e}^{ID} = \texttt{MAC}_{k'}(\texttt{pk})$ 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.

4.4.4 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 that is 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 $\texttt{sk}'$ that is needed for decryption.

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

5.1 Security and Privacy Properties

We make use of the model (and notation) of Pulls *et al.* [21], with some modifications to account for our stronger adversarial setting (untrusted instead of forward-secure server). In Insynd, it is the author instead of the server (which is assumed to be untrusted) that keeps the state, hence the `CorruptServer` oracle needs to be replaced by a `CorruptAuthor` oracle. Without loss of generality, we assume a single author A and server S. For sake of clarity, we present full updated definitions.

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 will be defined, specifically for each property. For properties with the prefix "forward", the adversary is not allowed to make any further queries after it makes a call to `CorruptAuthor`.

5.1.1 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.

Specifically for Insynd, 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), which also needs to be modeled. For this reason we provide the adversary with a decryption oracle:

- $m \leftarrow \texttt{DecryptEvent}(e)$: Decrypts 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 $\mathbf{Adv}^{SE}_{\mathcal{A}}(k) \leq \epsilon(k)$.*

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

Proof sketch. For each encryption, an ephemeral keypair is generated, as specified in Sect. 3.2. The ephemeral private key is what is being disclosed as part of proofs of message together with a nonce. This means that the information provided in proofs of message reveals nothing to the adversary beyond the already known public key of the recipient and, notably, the actual message (plaintext) contained

within the ciphertext and the nonce. Conservatively, this means that proofs of message can be seen as a decryption oracle where the adversary provides the ciphertext, and as part of constructing the ciphertext picks a nonce. Since the used encryption scheme provides IND-CCA2 security, the adversary has no advantage in winning this game. □

5.1.2 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.

The forward integrity (FI) experiment is defined as:

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

1. $l \leftarrow \mathcal{A}^{\mathcal{O}_{base}, \texttt{GetState}}()$
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.*

Theorem 2. *Given an unforgeable signature algorithm, an unforgeable one-time MAC, and a IND-CCA2 secure public-key encryption algorithm, Insynd provides computational deletion-detection forward integrity in the random oracle model, according to Definition 3.*

Proof sketch. Insynd provides deletion-detection forward integrity for recipients' events thanks to the use of the FssAgg authenticator by Ma and Tsudik [15], which is provably secure in the random oracle model for an unforgeable MAC function. The verification is "all or nothing", i.e., a recipient can detect if its events are authentic or not, not which events have been tampered with. We look at three distinct phases of a recipient in Insynd: before registration, after registration, and after the last event.

Before registration, the author has not generated any events for the recipient that can be modified or deleted. 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 either *state* (in the form of the initial key and value) or at least one event (the last event with the stop marker and authenticator) for this recipient.

After registration, but before the last event, the author must keep state for the recipient. The state is in the form of the current authentication key k and authenticator value v. 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. This 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 sequential key generator in the form of a hash chain. Note that each event uses a unique key for the MAC, where the key is derived from a hash function for which the adversary does not know the input. This means that it is an unforgeable one-time MAC function is sufficient.

After the last event, the author has deleted state. The last event contains a stop marker M_s and the authenticator value v before the last event was generated. The verification algorithm, Algorithm 1, verifies the authenticator value in the case of state (step 9) and no state (step 7). Compared to the privately verifiable FssAgg construction, we store the authenticator value in an event instead of state. However the authenticator value is still verified, and the algorithms fails if it is unable to do so (steps 4 and 7).

Finally, we note that the use of an IND-CCA2 secure encryption scheme prevents an adversary from learning any authenticator keys and values from the `GetState` oracle. □

5.1.3 Forward Unlinkability of Events

Informally, we define forward unlinkability of event by stating that it should be hard for an adversary to determine whether or not a relationship exists between two events in one round[3] of the `insert` protocol, even when given at the end of the round the entire state of the author: $(\texttt{pk}, k, v)$ of all recipients. We also need to take into account that for certain events, publicly verifiable proofs of recipient or message (or data stored at the author to be able to make these proofs) are available to the adversary. Obviously, the adversary knows the recipient of these messages, and does not break forward unlinkability of events, if it can only establish a link between two events in one round of the insert protocol for which it knows the recipient.

Even though we do not define forward unlinkability of events in terms of recipients, it can still be modeled as in [21], where 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:

- $vuser \leftarrow \texttt{DrawUser}(\texttt{pk}_i, \texttt{pk}_j)$: Generates a virtual user reference, as a monotonic counter, $vuser$ and stores $(vuser, \texttt{pk}_i, \texttt{pk}_j)$ 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 $vuser$.

- $\texttt{Free}(vuser)$: Removes the triple $(vuser, \texttt{pk}_i, \texttt{pk}_j)$ from table $\mathcal{D}$.

[3]Since the adversary is (presumably) in control of the server, it can trivially tell in which round events were added.

- $e \leftarrow$ `CreateEvent'`$(vuser, m)_b$: 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 $vuser$ in the table $\mathcal{D}$. Returns e.

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

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

1. $b \in_R \{0,1\}$
2. $g \leftarrow \mathcal{A}^{\mathcal{O}_{base},\texttt{DrawUser},\texttt{Free},\texttt{CreateEvent'},\texttt{GetState}}()$
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 $\mathbf{Adv}^{FU}_{\mathcal{A}}(k) \leq \epsilon(k)$.*

Theorem 3. *For an 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 according to Definition 4.*

Proof sketch. The adversary has access to the following information for events for which no publicly verifiable proofs of recipient or message (or the data stored at the author to be able to create these) are available: $e^{ID} = \texttt{MAC}_{k'}(\texttt{pk})$ 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.

In the random oracle model, the output of a hash function can be replaced by a random value. Note that the output of the random oracle remains the same for the same input. 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[4]. Hence the adversary that does not know the inputs of these hashes, n and k respectively, has no advantage towards winning the indistinguishability game.

Now we need to 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 active recipients; and values k' and n for events where a publicly verifiable proof of recipient or message is available or can be constructed from data the author stored additionally at the time of generating these events. The state variable k is generated using a forward-secure sequential key generator in the form of an evolving hash chain. There is no direct link between the values n, respectively k', of multiple events for the same recipient. Instead, for each event these values are derived from the recipient's current authentication key k at that time, using a random oracle. The adversary can thus not learn the value of the past authentication keys from the values n and k'. Lastly, from state, the adversary learns `pk` for all recipients. We note that the encryption for events provides key privacy, because the outputted ciphertext is encrypted using a symmetric key that is derived from both the Diffie-Hellman value and the nonce. Even when assuming that the adversary can compute the Diffie-Hellman value, it has no information on the nonce and hence the encryption provides key privacy, i.e., one cannot tell that the ciphertext was generated for a given public key.

We need to show that the adversary will not be able to link events together from the state variable v it keeps for every recipient. If $v \stackrel{?}{=} v_0$, 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.

Finally, we note that the use of an IND-CCA2 secure encryption scheme prevents an adversary from learning any authenticator keys and values from the `GetState` oracle. □

5.2 Public Verifiability

5.2.1 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 [20]). 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 [17] 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.

5.2.2 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 of [20] proves the security of a membership query in a Balloon. However, in this theorem, the adversary only has oracle access to producing new snapshots. In the forward security model, the adversary can learn the author's private signing key, and can therefore create arbitrary snapshots on its own. Given that the signature algorithm is unforgeable, the existence of a signature is therefore non-repudiable evidence of the snapshot having been made with the signing key.

5.2.3 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. The exact time-stamping mechanism is out of scope.

5.2.4 Recipient

Assuming a collision resistant hash function, an unforgeable signature algorithm and an unforgeable one-time MAC

[4]If the adversary has no knowledge of n, it cannot tell whether or not a given ciphertext was encrypted for a given recipient (with known public key), even when given the ephemeral secret key $\texttt{sk}'$ used to seal `crypto_box`. This implies that the adversary will also not gain any advantage from learning the corresponding ephemeral public key $\texttt{pk}'$, as part of the ciphertext.

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

5.2.5 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 `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 `pk'`. The prover provides `sk'`, such that `pk'` $\stackrel{?}{=}$ `pk`*, where `pk`$^* \leftarrow$`crypto_box_keypair(sk')`. This fixes `sk'`, since there is only one `sk'` for each `pk'` for Curve25519[5]. This fixes all the input to `crypto_box_open`: $c, n,$ `pk` and `sk'`, and `crypto_box_open` is deterministic.

6. EXTENSIONS

6.1 Non-interactive registration

With non-interactive registration it is possible for the author to register one of its recipients with another author, e.g., in distributed settings, while the recipient will still be able to reconstruct all events both at the original author as at the next author. Very similar to how this is done in [21], the original author does the following to register a recipient non-interactively at the next author:

1. Given `pk`, generate a *blinded* public key `pk'` using the blinding value b, where b is randomly generated.
2. Register `pk'` with the next author.
3. Generate a new event for the recipient `pk`, where the message consists the extend marker M_e concatenated with the blinding value b concatenated with the result of the register protocol.

At reconstruction, when the recipient encounters the extension marker M_e, it computes its new private key using b and `sk`, decrypts the information from the register protocol and follows the same procedures to retrieve its relevant events at the next author.

Blinding the public key of the recipient serves three purposes: First, it hides information from an adversary that compromises multiple authors, preventing trivial correlation of state tables to determine if the same recipient has been registered at both authors, ensuring forward unlinkability of recipient identifiers; Second, the blinding approach (compared to having the author create a new key pair) has the added benefit of, if run correctly, it is still only the recipient that at any point in time has the ability to decrypt events intended for it; Third, Insynd uses the public key as an identifier for the registration of a recipient. For each registration, we need a new identifier.

THEOREM 4. *Insynd provides forward unlinkability of recipient identifiers for the non-interactive registration assuming that the DDH assumption is valid for Curve25519.*

Note that the author can also run the non-interactive registration with itself. This serves two purposes: First, this introduces new randomness for the recipient, recovering *future events* from a limited compromise of a passive adversary in the past; Secondly, this enables the author to register the recipient in a new Balloon, e.g., when the author want to start using another server.

6.2 Stronger Adversarial model

By making use of the non-interactive registration, it is possible to support an even stronger adversary model at the author: namely that of a *time-limited passive adversary* which allows the adversary to recover from compromise. The fresh randomness for its registered recipients, disables any further profiling by the adversary. If the adversary got access to the author's signing key, used for showing authorship and consistency, the author needs to revoke its old signature key pair and generate a new one which from now on will be associated with the author. If not, e.g., when the author uses a Hardware Security Module (HSM) to generate its signatures, this is not necessary.

6.3 Privacy-Preserving Event Retrieval

By retrieving its events, the recipient inadvertently leaks information that could impact its privacy. For instance, the naive approach of retrieving events as fast as possible risks linking the events together, and their relative order within each run of the `insert` protocol, due to time, despite our (assumption of) an anonymous communication. To minimise information leakage, one could:

- Have recipients wait between requests. The waiting period should be randomly sampled from an exponential distribution, which is the maximum entropy probability distribution having mean $1/N$ with rate parameter N [19], assuming that some recipients may never access their events ($[0, \infty]$).
- Randomise the order in which events are requested. The `getState` protocol returns k_i, which enables the calculation of how many events have been generated so far using k_o.
- Introduce request and response padding to minimise the traffic profile, e.g., to 16 KiB as is done in Pond.[6]
- Use private information retrieval (PIR) [9, 11]. This is relatively costly but may be preferred over inducing significant waiting time between requests.
- Events on servers could safely be mirrored (since servers are not trusted), enabling recipients to spread their requests (presumably not all mirrors collude).

For now, we opt for the first two options of adding delays to requests and randomising the order. PIR-enabled servers

[5]There are two points on the curve for Curve25519 such that `pk'` $\leftarrow$`crypto_box_keypair(sk')` due to Curve25519 being a Montgomery curve, but Curve25519 only operates on the x-coordinate [4].

[6]https://pond.imperialviolet.org

would provide added value to recipients. Note that the mirroring approach goes hand in hand with our publicly verifiable consistency, since mirrors could *monitor* snapshot generation as well as serve events to recipients.

7. RELATED WORK

Ma and Tsudik [15] proposed a publicly verifiable FssAgg scheme by using an efficient aggregate signature scheme, BLS. 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 [14] 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 [10] based on authenticated data structures. The main drawback of the work of Crosby and Wallach, and to a lesser degree of ours on Insynd, is the reliance upon 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 [18], 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. The private verifiability mechanism for Insynd is therefore forward-secure (with regard to the author), unlike in CONIKS, where the directory is completely untrusted and only holds data provided by users.

PillarBox is a fast forward-secure logging system by Bowers *et al.* [7]. 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.

The transparency logging scheme by Pulls *et al.* [21] has a similar setting to ours with multiple recipients, our non-interactive registration was inspired by their work. The purpose of their scheme is to enable a one-way communication channel to make data processing more transparent to users of data processors. The scheme is based on hash- and MAC-chains, influenced by the secure log design of Schneier and Kelsey [22]. We make use of their general framework for proving the security and privacy properties of Insynd, with modifications to account for our stronger adversary model at the server (untrusted instead of forward secure).

Pond and WhisperSystem's TextSecure[7] are prime examples of related secure asynchronous messaging systems. While these systems are for two-way communication, there are several similarities. Pond, like Insynd, relies on an anonymity network to function. Both Pond and TextSecure introduce dedicated servers for storing encrypted messages. In Pond, clients pull their server for new messages with exponentially distributed connections. The Axolotl ratchet[8] is used by both Pond and TextSecure. Axolotl is inspired by the Off-the-Record Messaging protocol [6] and provides among other things forward secrecy and recovery from compromise of keys by a passive adversary. Our non-interactive registration mimics the behaviour in Axolotl, in the sense that new ephemeral keying material is sent with messages. Note that the goal of Insynd is for messages to be non-repudiable, unlike Pond, TextSecure and OTR who specifically want *deniability*. Insynd achieves non-repudiation through the use of Balloon and how we encrypt messages.

8. PERFORMANCE

We implemented Insynd in the Go programming language.[9] The source code and steps to reproduce our benchmark are publicly available at `http://www.cs.kau.se/pulls/insynd/`. We performed our benchmark on Debian 7.8 (x64) using an Intel i5-3320M quad core 2.6GHz CPU and 7.7 GB DDR3 RAM. The performance benchmark focuses on the `insert` protocol since the other protocols are relatively infrequently used, and reconstruction time is presumably dominated by the mechanism used to protect the recipient's privacy when downloading events. For our benchmark, we ran the author and server on the same machine but still generated and verified proofs of correct insertion into Balloon.

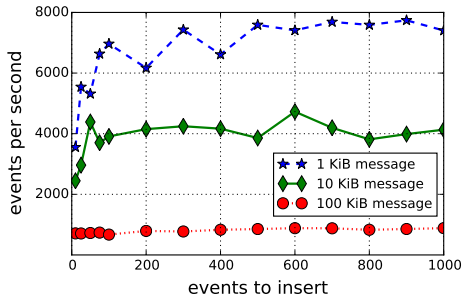
Figure 4 presents our benchmark, based on averages after 10 runs using Go's built-in benchmarking tool, where the x-axis specifies the number of events to insert per run of the `insert` protocol. Figure 4a shows the number of events per second for different message sizes in a Balloon of 2^{20} events. 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 1KiB 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 100KiB-messages per second (around at least 200 events to insert), the goodput is ≈80 MiB/s. 10KiB 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[15], for their FssAgg schemes, achieve event generation (signing) in the order of milliseconds per event (using significantly older hardware than us). Marson and Poettering [16], with their seekable sequential key generators, generate *key material* in a few microseconds. Note that for both these schemes, message 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 en-

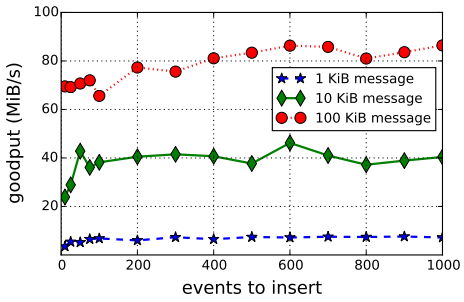
[7] `https://whispersystems.org`

[8] `https://github.com/trevp/axolotl/wiki`

[9] `https://golang.org`



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



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

Figure 4: A benchmark related to inserting events. Figure 4a shows the number of events that can be inserted per second in a 2^{20} Balloon for different message sizes. Figure 4b shows the corresponding goodput in MiB/s for the data in Figure 4a.

crypt messages in addition to providing integrity protection. Pulls *et al.* [21], for their transparency logging scheme, generate events in the order of tens of milliseconds per event. For PillarBox, Bowers *et al.* [7] 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.

9. CONCLUSIONS

Insynd's design is based around concepts from authenticated data structures, forward-secure key generation from the secure logging area, and ongoing 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.* [21], which was adjusted to take into account our stronger adversarial model that assumes a forward-secure author and an untrusted server. Furthermore, Insynd achieves publicly verifiable consistency and allows for a number of publicly verifiable proofs to show the author, recipient, message and approximate time of events. This further increases the utility of a transparency logging scheme. Our proof of concept implementation, which is freely available, shows that Insynd offers comparable performance for event generation to state-of-the-art secure logging systems, like PillarBox [7].

Acknowledgements

We would like to thank 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.

References

[1] J. H. An. Authenticated encryption in the public-key setting: Security notions and analyses. *IACR Cryptology ePrint Archive*, 2001:79, 2001.

[2] M. Bellare, A. Boldyrevay, A. Desaiz, and D. Pointcheval. Key-Privacy in Public-Key Encryption. In *ASIACRYPT*, volume 2248 of *LNCS*, pages 566–582. Springer, 2001.

[3] M. Bellare and B. S. Yee. Forward-Security in Private-Key Cryptography. In *CT-RSA*, volume 2612 of *LNCS*, pages 1–18. Springer, 2003.

[4] D. J. Bernstein. Curve25519: New diffie-hellman speed records. In *PKC*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006.

[5] D. J. Bernstein, T. Lange, and P. Schwabe. The Security Impact of a New Cryptographic Library. In *LATINCRYPT*, volume 7533 of *LNCS*, pages 159–176. Springer, 2012.

[6] N. Borisov, I. Goldberg, and E. A. Brewer. Off-the-record communication, or, why not to use PGP. In *WPES*, pages 77–84. ACM, 2004.

[7] K. D. Bowers, C. Hart, A. Juels, and N. Triandopoulos. PillarBox: Combating Next-Generation Malware with Fast Forward-Secure Logging. In *Research in Attacks, Intrusions and Defenses Symposium*, volume 8688 of *LNCS*, pages 46–67. Springer, 2014.

[8] A. Buldas, P. Laud, H. Lipmaa, and J. Willemson. Time-Stamping with Binary Linking Schemes. In *CRYPTO*, volume 1462 of *LNCS*, pages 486–501. Springer, 1998.

[9] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *FOCS*, pages 41–50. IEEE Computer Society, 1995.

[10] S. A. Crosby and D. S. Wallach. Efficient Data Structures For Tamper-Evident Logging. In *USENIX Security Symposium*, pages 317–334. USENIX, 2009.

[11] C. Devet and I. Goldberg. The Best of Both Worlds: Combining Information-Theoretic and Computational PIR for Communication Efficiency. In *PETS*, volume 8555 of *LNCS*, pages 63–82. Springer, 2014.

[12] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The Second-Generation Onion Router. In *USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[13] FIDIS WP7. D 7.12: Behavioural Biometric Profiling and Transparency Enhancing Tools. Future of Identity in the Information Society, http://www.fidis.net/resources/deliverables/profiling/, March 2009.

[14] J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Australasian Workshops on Grid Computing and e-Research*, volume 54 of *CRPIT*, pages 203–211. Australian Computer Society, 2006.

[15] D. Ma and G. Tsudik. A new approach to secure logging. *TOS*, 5(1), 2009.

[16] G. A. Marson and B. Poettering. Even more practical secure logging: Tree-based seekable sequential key generators. In *ESORICS 2014*, volume 8713 of *LNCS*, pages 37–54. Springer, 2014.

[17] D. Mazières and D. Shasha. Building secure file systems out of byzantine storage. In *Symposium on Principles of Distributed Computing*, pages 108–117. ACM, 2002.

[18] M. S. Melara, A. Blankstein, J. Bonneau, M. J. Freedman, and E. W. Felten. CONIKS: A Privacy-Preserving Consistent Key Service for Secure End-to-End Communication. Cryptology ePrint Archive, Report 2014/1004, 2014.

[19] S. Y. Park and A. K. Bera. Maximum Entropy Autoregressive Conditional Heteroskedasticity Model. *Journal of Econometrics*, 150(2):219–230, June 2009.

[20] T. Pulls and R. Peeters. Balloon: A forward-secure append-only persistent authenticated data structure. Cryptology ePrint Archive, Report 2015/007, 2015.

[21] T. Pulls, R. Peeters, and K. Wouters. Distributed privacy-preserving transparency logging. In *WPES*, pages 83–94. ACM, 2013.

[22] B. Schneier and J. Kelsey. Cryptographic Support for Secure Logs on Untrusted Machines. In *USENIX Security Symposium '98*, pages 53–62. USENIX, 1998.