Insynd

Privacy-Preserving Secure One-Way Messaging 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 secure and privacy-preserving one-way messaging. Insynd is ideally suited for sending personalised breach notifications to end-users, enabling services to provide positive evidence to a third party that they have complied with their obligations (and conversely, for end-users to prove that the notification was not timely). Insynd provides *i*) secrecy of messages, *ii*) message integrity and authenticity, *iii*) protection against recipient profiling, and *iv*) publicly verifiable proofs of who sent what message to which recipient at what particular time. Our scheme is built on an authenticated data structure, named Balloon, enabling 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 uses modern cryptographic primitives, making it also a suitable secure logging system or evidence store, despite the use of "slow" public-key cryptography. Our prototype implementation shows improved performance over related work and competitive performance for more data-intensive settings like secure logging.

Keywords

Transparency-Enhancing Tool, Security and Privacy Protection; Authenticated data structure; Cryptographic controls; Logging.

1. INTRODUCTION

Insynd[1] is a cryptographic scheme for one-way messaging that is designed to be secure and privacy-preserving. The development of Insynd has been motivated by the need for protecting one-way communication by service providers to potentially offline users where technologies like e-mail, SMS, and push notifications fall short due to lack of adequate privacy and security protections. For direct communication with service providers, users already have access to, e.g., TLS, Tor [18], and Tor Hidden Services that provide different security and privacy protection for both users and services. These technologies assume two online communicating parties, and therefore are not directly a fit for asynchronous communication. Ongoing work, e.g., Pond by Langley[2], builds asynchronous communication on top of technologies like Tor and Tor Hidden Services, with the goal of addressing several pressing security and privacy issues in *two-way* asynchronous communication where both the sender and receiver may be unavailable, similar to email. While closely related to our setting, we are addressing the need of *one-way* asynchronous communication from a service provider with high availability to potentially less reliable users.

If service providers are available and users are not, the natural question is then why there is a need to communicate asynchronously at all? The user can just contact the service provider, using technologies like Tor and TLS for privacy and security protection, at their convenience. However, the service provider might get compromised between the time of generating messages and the time of recipients retrieving those messages (forward security model). Insynd covers this by modeling storage of the messages as being on intermediate servers that are considered to be active adversaries. Since these servers do not need to be trusted, a service provider can both run the server on its own or safely outsource it to, e.g., commodity cloud services. This means that we look at *how* the service provider should store messages. Insynd uses an authenticated data structure, named Balloon [27], to safely outsource data storage. Balloon is similar to the underlying data structures at the center of, e.g., Bitcoin [25] and Certificate Transparency [20].

The main applications that Insynd is designed for are as a mechanism for delivering detailed *breach notifications* to users and for continuously *sharing data processing* information. Additionally, Insynd can also be used as a *secure log*. In an ongoing EU FP7 research project, Insynd is currently being used for these three purposes.

For breach notifications, Insynd enables the sharing of detailed (and therefore highly personal) breach notifications where the service provider (and user) can generate publicly verifiable evidence to a third party of providing timely notifications (or conversely, users can prove that notification was not timely). Timely breach notification is required for legal compliance with, e.g., the EU e-Privacy Directive 2002/58/EC, HIPAA, the proposed Personal Data Notification and Protection Act in the US, and the ongoing EU Data Protection Regulation.

For sharing data processing information, the service provider continuously sends messages to users detailing how their personal data is processed, e.g., by logging access made in a privacy-policy engine for the PrimeLife Policy Language (PPL) [30, 28]. This increases transparency to users, and therefore Insynd can be seen as a privacy-preserving ex-post *transparency-enhancing tool*. Insynd secures the data pro-

[1]Insynd is a word-play on the Swedish word "insyn", roughly translatable to "insight", with a "d" suffix in the tradition of a deamon. The result, "Insynd", can be seen in Swenglish as "in sin", since "synd" means "sin" in Swedish.

[2]pond.imperialviolet.org, accessed 2015-01-29.

cessing information, both from an adversary and from the service provider tampering with sent messages (the service provider is *forward secure*), and protects the privacy of users. PPL also supports so-called downstream usage, where personal data may be shared by one service provider to another without interacting with the user. For this reason, Insynd also supports *non-interactive registration* of users, such that one service provider can enable other service providers to send messages to the user without interacting with the user.

Insynd can also be used as a secure log. Like the secure logs by, e.g., Ma and Tsudik [21] and Holt [19], Insynd offers publicly-verifiable forward integrity and deletion detection and competitive performance.

We make the following contributions:

- We present the cryptographic scheme Insynd, consisting of five protocols that are inspired by concepts from authenticated data structures, secure logging, and ongoing work on secure messaging protocols (Section 4).
- In addition, we show how to achieve publicly verifiable proofs of author, recipient, message, and time with minor impact on other properties (Section 4.4).
- We show that Insynd provides secrecy, forward-integrity and deletion detection, forward unlinkability of events, and publicly verifiable consistency with a forward-secure author (Section 5 and Appendix A).
- We show that our proof-of-concept implementation offers comparable performance for event generation to state-of-the-art secure logging systems (Section 7).

The rest of the paper is structured as follows. Section 2 provides an overview of our setting, adversary model, assumptions, and goals. Section 3 introduces the cryptographic building blocks used in the Insynd. The Insynd scheme is presented in Section 4. Section 5 evaluates the properties of Insynd. Section 6 presents related work, and Section 7 a brief performance analysis. Section 8 concludes this paper with some promising future work. Finally, Appendix A contains a more thorough security evaluation.

2. OVERVIEW

We take an event-centric view, where an event is a container for a message (the exact format of an event is specified in Section 4.2). An event is authored by an author A, and intended for a recipient R. Events are sent by the author A to an intermediate server S, and the recipient R polls the server for new events. We consider a *forward secure* [4] author, where our goal is to protect events sent *prior to compromise* of the author. Furthermore, we distinguish between two types of compromises: a compromise by an active adversary and a time-limited compromise by a passive adversary. If the compromise is by an active adversary, we provide forward security. If the compromise is time-limited by a passive adversary, we can recover once the author is no longer compromised. The server is always considered compromised by an active adversary.

2.1 Threat Model and Assumptions

The ability to recover from the compromise of a passive adversary protects against a large number of threats, such as a memory dump of the author's system, which could happen for forensic analysis, as a result of a system crash, a privileged attacker looking for secrets in memory, lost or compromised backups, and legal obligations to provide data to law enforcement or other legal entities. Protocols like Off-the-Record Messaging (OTR) [11] provides protections against similar threats.

Commodity cloud services, while convenient, pose a large threat to both security and privacy due to the inherent loss of control over data and processing. As mentioned before, treating the server as untrusted covers both the case when the author wants to store its events itself and when the author wants to outsource storage.

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) 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 there exists a trustworthy time-stamping authority [13].

2.2 Goals

We note the following goals and properties for Insynd:

Secrecy Only the recipient of a message can read it.

Forward Integrity and Deletion Detection Nobody can modify or delete messages sent prior to author compromise without detection.

Forward Unlinkability of Events For each run by the author of the protocol to send new messages, all the events sent in that run are unlinkable. This also implies that an adversary cannot tell which events belong to which recipient. This prevents recipient profiling due to event generation.

Publicly Verifiable Consistency Anyone should be able to verify the consistency of all events stored at a server.

Publicly Verifiable Proofs Both the author and recipient receiving a message can create publicly verifiable proofs. The proofs are "publicly verifiable" in the sense that anyone can verify the proof given some cryptographic material (like a verification key). Insynd can generate the following publicly verifiable proofs:

Author Who was the author of an event.

Time When was the event sent.

Recipient Who was the recipient of an event.

Message What is the message in an event.

These proofs should not inadvertently require the disclosure of private keys. Each proof is an isolated disclosure (and potential violation of a property of Insynd, like message secrecy).

Non-Interactive Registration An author can enable another author to send messages to recipients already registered with the initiating author. This enables distributed settings with multiple authors, where authors do not need to interact with recipients to start sending messages. Furthermore, the identifiers for the recipient at the two authors are unlinkable for sake of privacy.

Ease of implementation Primitives should be chosen such to ease the implementation as much as possible, not shifting the security of the implementation entirely to the implementer.

3. BUILDING BLOCKS

The general idea behind Insynd is to store events at an untrusted server. Each event consists of a unique identifier that is only reconstructible by the intended recipient, and an encrypted message for that recipient. To store events, Insynd makes use of an authenticated data structure [23]. This enables us to support a stronger adversary model than related work, and it is crucial in providing our publicly verifiable proofs. Next, we describe and motivate our use of a particular data structure (Balloon), forward secure state generation mechanism (used as a building block to generate unique recipient-specific identifiers and to provide forward integrity and deletion detection) and encryption scheme to be able to reach the goals set in Section 2.2. Finally, we present and motivate our selection of the cryptographic primitives.

3.1 Balloon

Balloon is a forward-secure append-only persistent authenticated data structure by Pulls and Peeters [27]. Balloon is designed for an initially trusted author that generates events to be stored in a data structure (the Balloon) kept by an untrusted server, and clients that query this server for events intended for them based on keys and snapshots. Snapshots are generated by the author as new events are inserted and fix all data stored in the Balloon that far. Pulls and Peeters define the following algorithms for Balloon:

- $\texttt{B.Insert}(E) \rightarrow P$. Given a set of events E with unique keys and non-zero values, the system appends it to the Balloon B, and then outputs proof P.
- $\texttt{P.VerifyInsert}(E, \mathsf{s}_l) \rightarrow \{\mathsf{s}_{l+|E|}, \texttt{false}\}$. Verifies that P proves that events E were correctly inserted into the Balloon with the latest verified snapshot s_l. Outputs the next snapshot $\mathsf{s}_{l+|E|}$ if the proof is correct, `false` otherwise.
- $\texttt{B.MembershipQuery}(k, \mathsf{s}_j) \rightarrow (P, \mathsf{s}_l, e_i)$. Generates a (non-)membership proof P for the event with key k from snapshot s_j for the Balloon B. The algorithm outputs P, the latest snapshot s_l, and, in the case of membership, the event e_i, where $i \leq j \leq l$.
- $\texttt{P.MembershipVerify}(k, \mathsf{s}_j, \mathsf{s}_l, e'_i) \rightarrow \{\texttt{true}, \texttt{false}\}$. Verifies that P proves the (non-)membership of the event e'_i with key k in s_l and, if member, that e'_i is the i:th event in s_j, where $i \leq j \leq l$.

Internally, Balloon is composed of two authenticated data structures: a history tree and a hash treap. A history tree is a data structure by Crosby and Wallach [16], virtually identical to the data structure used in Certificate Transparency [20]. Thanks to being composed with a hash treap, Balloon can provide efficient *non-membership* proofs. This is required for recipients to be convinced (by the untrusted server) that there is no event with a given key, without recipients downloading the entire Balloon. While any persistent authenticated dictionary (PAD) [1] could provide efficient non-membership proofs and fit the setting, Balloon was designed to be significantly more efficient thanks to being append-only, unlike a PAD, that supports deletion.

Balloon depends on a collision resistant hash function, an unforgeable signature algorithm, and a *gossiping mechanism* for snapshots. Pulls and Peeters show, in the forward-security model, that snapshots cannot be undetectably inconsistent (removing or modifying data already stored in the Balloon) assuming the existence of *monitors*. A monitor downloads all events at the server to recompute and compare all generated snapshots. At compromise of the author, the latest snapshot s_l has (presumably) been gossiped to monitors, and s_l provably fixes all data stored up to that point. Future snapshots generated by the adversary can only append data to the Balloon under the assumption of a collision resistant hash function.

3.2 Forward-Secure State Generation

For the recipient to find its events, and maintain unlinkability of events, identifiers for events need to be unlinkable and deterministically generated for each recipient in a forward secure manner. To provide forward-integrity and deletion for individual recipients as well, there is also the need to authenticate all events entirely up to a certain point in time in a forward secure manner.

The author will keep *state* for each recipient, consisting of a key k and a value v that is continuously evolved by overwriting the past key and value as new events e_i^j for recipient j are generated. The initial key, k_0, is agreed upon at recipient registration. The key is used to generate the recipient-specific event identifiers. It is constantly evolved, with each recipient-specific event e_i^j, using a simple forward-secure sequential key generator (forward-secure SKG) in the form of an evolving hash-chain [4, 29, 22]:

$$k_i = \begin{cases} \texttt{Hash}(k_{i-1}), & \text{if } i > 0 \\ k_0, & \text{if } i = 0 \end{cases} \tag{1}$$

For forward integrity and deletion detection, we use a Forward-Secure Sequential Aggregate (FssAgg) authenticator by Ma and Tsudik [21] in the form of combining an evolving hash-chain and a MAC. The forward-secure SKG, k_i, is used in the FssAgg as follows to compute the value v_i:

$$v_i = \begin{cases} \texttt{Hash}\big(v_{i-1} || \texttt{MAC}_{k_{i-1}}(e_{i-1}^j)\big), & \text{if } i > 0 \\ k_0, & \text{if } i = 0 \end{cases} \tag{2}$$

We set $v_0 = k_0$ to prevent a length distinguisher for one of our protocols, as described in Section 4.3.

3.3 Public Key Encryption Scheme

Messages stored for recipients are encrypted under their public keys. The encryption scheme must provide indistinguishably under adaptive chosen ciphertext attack (IND-CCA2) [3], and key privacy under adaptive chosen ciphertext attack (IK-CCA) [2]. IK-CCA is only needed for the ciphertexts contained within the events, this to avoid that the event's recipient can be deduced from the ciphertext. IND-CCA2 is needed since our publicly verifiable proofs of message reveals the decryption of ciphertext. Hence the adversary can be assumed to have access to a decryption oracle.

A publicly verifiable proof of message is in essence a proof that a given ciphertext, encrypted for a given recipient (public key), corresponds with the output plaintext (message).

These proofs can be generated by either the author or the recipient, as described in Section 4.4. The encryption scheme should be such that:

- these proofs do not require the recipient or the author to reveal any long term private or secret keys;
- it provides forward security at the author side: only at the time of encryption, the author can choose to store additinal information that allows it to recover the plaintext at a later point in time; however the author cannot recover any other plaintext for which it did not store this additional information, which was generated during the encryption.

3.4 Cryptographic Primitives

The cryptographic primitives needed for a Balloon are a hash function and a signature scheme. For the forward-secure state generation, additionally a MAC scheme is needed. Finally we need a public key encryption scheme.

To ease implementation, Insynd is designed around the use of NaCl [9]. 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. We selected the following primitives:

- SHA-512: a collision and pre-image resistant `Hash`;
- Ed25519 [8]: an existentially unforgeable under chosen-message attack signature algorithm `Sign`;
- Poly1305 [5]: a one-time existentially unforgeable `MAC`;
- crypto_box: a public-key authenticated $\texttt{Enc}^n_{\texttt{pk}}$ using Curve25519 [6], XSalsa20 [7] and Poly1305 [5].

We now go into the details of crypto_box and how we intend to use it. The encryption makes use of elliptic curve Diffie Hellman (ECDH) and a nonce to derive a symmetric key for the subsequent symmetric authenticated encryption of the message. This key can both be computed by the recipient and the receiver. Three functions make up `crypto_box`:

1. Public key `pk` and private key `sk` generation using `crypto_box_keypair(pk,sk)` such that `crypto_scalarmult_base(pk, sk)`, where `crypto_scalarmult_base` multiplies the scalar `sk` with the Curve25519 basepoint resulting in `pk`;
2. Authenticated encryption of a message `m` using a unique nonce `n`, the recipients public key `pk` and the sender's private key `sk`: `c = crypto_box(m,n,pk,sk)`;
3. Authenticated decryption of the ciphertext `c` using nonce `n`, the sender's public key `pk` and the recipients private key `sk`: `m = crypto_box_open(c,n,pk,sk)`;

For each message that the author wants to encrypt for a given recipient, it will generate a new ephemeral key pair `crypto_box_keypair(pk', sk')`. By revealing `sk'`, anyone can compute `pk'` using `crypto_scalarmult_base`, which enables us to prove to a third party that a ciphertext corresponds to a plaintext for a given recipient (as described later in Section 4.4.4). For the recipient to be able to do this, it needs to know the ephemeral `sk'`. Therefore we append `sk'` to the message before encryption. Moreover the encryption scheme provides forward security at the author side (the author needs to store `sk'` to be able to recover the plaintext afterwards). Note that even though crypto_box only requires the nonce n to be unique for a given pair of sender `sk` and recipient `pk`, we use the nonce to associate the ciphertext to a given event (see Section 4.2). For encryptions where no nonce is provided, n=0. Apart from the ciphertext c, the ephemeral public key `pk'` also needs to be stored to decrypt c. We do not store the nonce, since it is either 0 or linked to an event (as described in Section 4.2). We define the following algorithms for encryption and decryption:

- $\texttt{Enc}^n_{\texttt{pk}}(m) = (c, \texttt{pk'})$ for a random `sk'`, with `crypto_scalarmult_base(pk', sk')` and c = `crypto_box(`m`||sk',`n`,pk,sk')`.
- $\texttt{Dec}^n_{\texttt{sk}}(c, \texttt{pk'}) = (m, \texttt{sk'})$ with m`||sk'` = `crypto_box_open(`c`,`n`,pk',sk)`.
- $\texttt{Dec}^n_{\texttt{sk'},\texttt{pk}}(c, \texttt{pk'}) = m$ if `sk'`$\stackrel{?}{=}$`sk`* and `pk'`$\stackrel{?}{=}$`pk`*, where m`||sk`* = `crypto_box_open(`c`,`n`,pk,sk')` and `crypto_scalarmult_base(pk`*`, sk')`; otherwise $\perp$.

4. THE INSYND SCHEME

Figure 1 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 and present relevant algorithms.

4.1 Setup and Registration

The author and server have signature key-pairs, $(\mathsf{A}_{\mathsf{sk}}, \mathsf{A}_{\mathsf{vk}})$ and $(\mathsf{S}_{\mathsf{sk}}, \mathsf{S}_{\mathsf{vk}})$, respectively. We assume that A_{vk} and S_{vk} are publicly attributable to the respective entities, e.g., by the use of some trustworthy public-key infrastructure. Note that, since we assume that the author may issue new signing keys as a consequence of becoming compromised, $(\mathsf{A}_{\mathsf{sk}}, \mathsf{A}_{\mathsf{vk}})$ may change over time. We consider the exact mechanism for revocation and re-issuance to be out of scope.

4.1.1 Author-Server Setup

The purpose of the `setup` protocol (pink box Figure 1) is for the author and server to create a new Balloon, stored at the server, with a given set of parameters. The protocol is started by the author who sends $(m, r, \mathsf{A}_{\texttt{URI}})$ to the server. The first parameter, m, is an integer specifying the maximum number of events to be stored in the Balloon. The second parameter, r, is the retention time in Unix time that specifies until which point in time the server must retain the Balloon. These two parameters are merely a practicality and does not influence any of our security or privacy properties (although obviously these play a crucial role in availability and may be impractical). The last parameter, $\mathsf{A}_{\texttt{URI}}$, specifies the uniform resource identifier (URI) of the state associated with the to-be created Balloon at the author. This parameter plays a crucial role, since it will later be used by the recipient to query its state.

After receiving the parameters for the `setup` protocol, the server verifies the parameters (m is a positive integer, the retention time is in the future, and $\mathsf{A}_{\texttt{URI}}$ identifies a resource under control by the author) and, if acceptable,

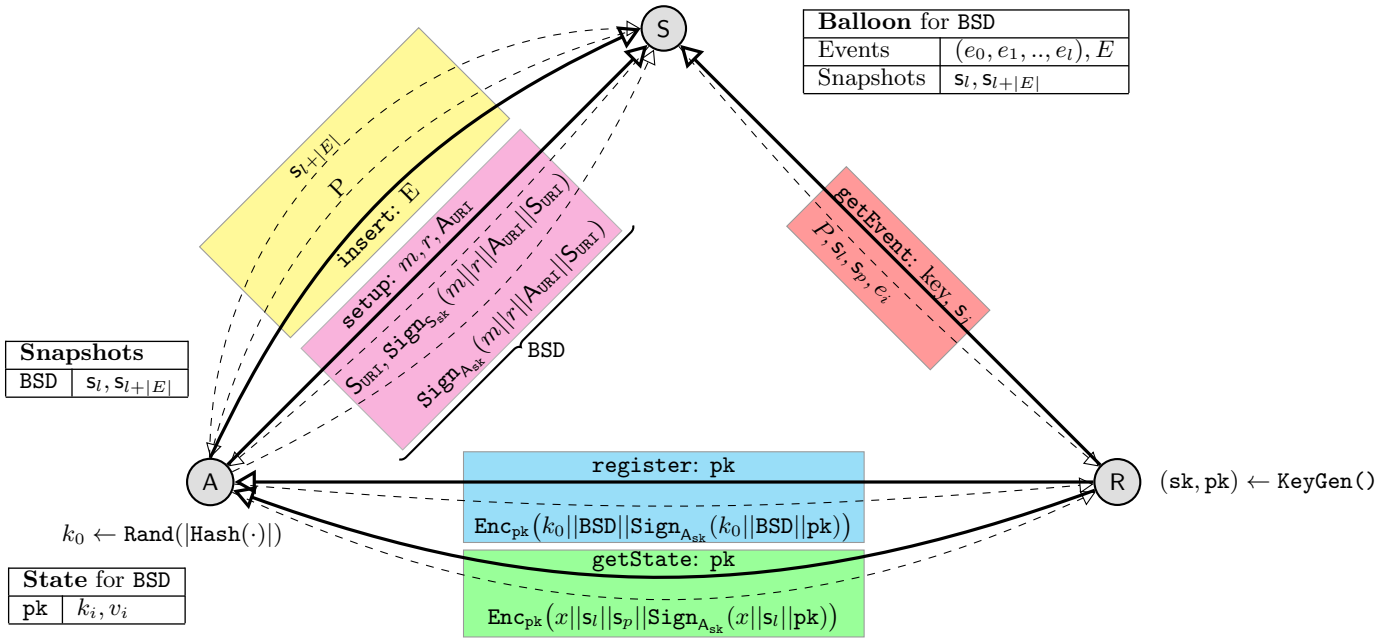


Figure 1: The Insynd scheme, consisting of five protocols (colored boxes), between an author A, a server S, and a recipient R. A solid line indicates the start of protocol and a dashed line a response. Section 4 describes each protocol in detail.

signs the parameters together with the URI to the new Balloon at the server, S_{URI}. The server replies with S_{URI} and $Sign_{S_{sk}}(m||r||A_{URI}||S_{URI})$. The signature commits the server to the specified Balloon.

Upon receiving the reply, the author verifies the signature and S_{URI} from the server, and signs the parameters together with the two URIs. The final signature is sent to the server to acknowledge that the new Balloon is now setup. We refer to the two parameters, two signatures, and two URIs generated as part of the `setup` protocol as the *Balloon setup data* (`BSD`). `BSD` commits both the author and the server to the newly created Balloon, acts as an identifier for the run of the `setup` protocol, and is later used by the recipient for reconstruction. Once the server receives the final signature, it constructs `BSD` on its own and now accepts both the `insert` and `getEvent` protocols for the Balloon.

4.1.2 Recipient Registration

The purpose of the `register` protocol (blue box in Figure 1) is to enable the author to send messages to the recipient, and in the process have the author commit to how these messages will be delivered to the recipient. The commitment is necessary, just like in the secure logging area for FssAgg schemes as noted by Ma and Tsudik [21], to prevent the author from fully refuting that there should exist any messages. The recipient generates a new key-pair, $(sk, pk) \leftarrow KeyGen()$, and sends the public key to the author to initiate the protocol.

Upon receiving the public key, the author verifies that the key is 32 bytes long (All 32-byte strings are valid Curve25519 public keys [6] for our use-case), generates an initial authentication key $k_0 \leftarrow Rand(|Hash(\cdot)|)$, and sets the authenticator value $v_0 \leftarrow k_0$. We set $v_0 = k_0$ to ensure that the length of v is constant, which prevents a length distinguisher for the `getState` protocol (see Section 4.3). The initial authentication key and authenticator value is associated with the public key `pk` of the recipient in the author's *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 registered in the Balloon for `BSD`. This is used for our combined SKG and FssAgg, as described in Section 3.2. As a reply to the recipient, the author returns k_0, the Balloon setup data `BSD`, and a signature by the author: $Sign_{A_{sk}}(k_0||BSD||pk)$. The signature also covers the public key of the recipient to bind the registration to a particular public key. Lastly, before sending the reply, the author encrypts the reply with the provided public key. This may appear superfluous, but plays an important in preventing a compromised passive adversary from learning k_0 when extending the registration to another author, as described in Section 4.2.4.

On receiving the reply, the recipient decrypts the reply, verifies all three signatures (two in `BSD`), and stores the decrypted reply. It now has everything it needs to run the `getEvent` protocol, but first, we need to generate events.

4.2 Generating Events

Before describing the `insert` protocol, we need to define an event. An event e consists of an identifier and a payload. The identifier, e^{ID}, identifies the event in a Balloon and is used by a recipient to retrieve its events. The event payload, e^{P}, contains the encrypted message from the author.

Algorithm 1 describes how an event is generated by the author. First, the author derives a nonce n and an event key k' from the recipient's current authentication key k (step 1). The current authentication key is stored in the author's state table. The first hash prefixes a 1 to distinguish the generation of the nonce with the update of the authentication key in step 5. The structure of generating the nonce and event key is used for publicly verifiable proofs of message, see Section 4.4.4. Figure 2 visualises the key derivation. In step 2, the event identifier is generated by computing a `MAC` on the recipient's public key using the event key. This links the event to a particular recipient, which can be used for publicly verifiable proofs of recipient, see Section 4.4.2. In

step 3, the message is encrypted using the recipient's public key and the generated nonce, linking the event identifier and event payload together. In step 4, the authenticator value v for the recipient aggregates the entire event, using the construction from Section 3.2, and overwrites the current authenticator in state for the recipient. Finally, in step 5, the current authentication key is evolved using a hash function, overwriting the old value in the state table.

Algorithm 1 Generate an event for a recipient.

Require: A message m, a recipient's public key $\texttt{pk}$ and the associated authentication key k and value v.
Ensure: An event and the recipient's state has been updated.

1: $k' \leftarrow \texttt{Hash}(n), n \leftarrow \texttt{Hash}(1||k)$
2: $e^{ID} \leftarrow \texttt{MAC}_{k'}(\texttt{pk})$
3: $e^{P} \leftarrow \texttt{Enc}^{n}_{\texttt{pk}}(m)$
4: $v \leftarrow \texttt{Hash}\big(v||\texttt{MAC}_k(e)\big)$
5: $k \leftarrow \texttt{Hash}(k)$
6: **return** e

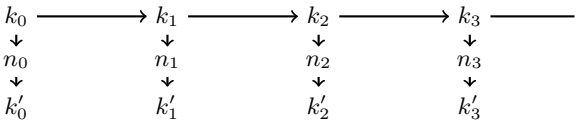


Figure 2: How we derive the nonce n and event key k' from the authentication key k for event generation.

4.2.1 Insert

The purpose of the `insert` protocol (yellow box in Figure 1) is to insert a set of events E, into a Balloon kept by the server. The server generates one or more events using Algorithm 1 and sends E to the server at $\mathsf{S}_{\texttt{URI}}$ to initiate the protocol. The server uses $P \leftarrow \texttt{B.Insert}(E)$ on the Balloon, described in Section 3.1, to generate a proof P of correct insertion into the Balloon. The server replies with P.

The author in turn uses $\texttt{P.VerifyInsert}(E, \mathsf{s}_l)$ to verify the proof of correct insert, where s_l is the latest snapshot generated by the author. The latest snapshot for an empty Balloon is null. If the verification fails, the author restarts the protocol. If the verification succeeds, then `P.VerifyInsert` produces a new snapshot $\mathsf{s}_{l+|E|}$. The author stores the snapshot in its *snapshot table* for `BSD`, and sends the snapshot to the server.

The server verifies the snapshot, in particular comparing it to what it expects from P: both roots from the updated hash treap and history tree match, and the snapshot follows the format specified shortly. Finally, the server stores the snapshot $\mathsf{s}_{l+|E|}$ and events E in its *Balloon table* for `BSD`. Given the data in a Balloon table, the Balloon data structure can be deterministically reconstructed.

4.2.2 Snapshots and Gossiping

Inspired by CONIKS [24], we modify the snapshot construction from the specification of Balloon. CONIKS works in a closely related setting to ours and links snapshots[3] together into a snapshot chain, as part of their work on specifying their snapshot gossiping mechanism for an authenticated data structure similar to Balloon. We define a snapshot as follows:

$$\mathsf{s}_i \leftarrow \big(i, c_i, r_i, t_i, \texttt{Sign}_{\mathsf{sk}}(i||c_i||r_i||BSD||s_p||t_i)\big) \qquad (3)$$

The i:th version snapshot s_i contains the latest commitment c_i on the history tree and root r_i of the hash treap, both as part of Balloon, to fix the entire Balloon. `BSD` is included in the signature of the snapshot to link the snapshot to a particular Balloon. The previous snapshot, s_p, is included to form a snapshot chain. Finally, an *optional* time-stamp t_i from a time-stamping authority is included both as part of the snapshot and in the signature. The time-stamp must be on $(i||c_i||r_i||\texttt{BSD}||s_p)$. How frequently a time-stamp is included in snapshots directly influences how useful proofs of time are, as described in Section 4.4.2.

Note that timestamps do not serve any other purpose than to enable publicly verifiable proofs of time in Insynd. Time-stamping of snapshots *are irrelevant* for our other properties (which we show in Section 5 and Appendix A), and snapshots in general only play a minor role for our properties. This means that our gossip mechanism for snapshots can be relaxed. As will become apparent, we gossip the latest snapshot as part of the `getState` and `getEvent` protocols. Since snapshots are both linked and occasionally timestamped, this greatly restricts our adversary in the forward-security model. While the `getState` protocol identifies the recipient, the `getEvent` protocol does not. The author and server should make all snapshots available, e.g., on their websites.

4.2.3 The Last Event

If the author wishes to no longer be able to send messages to the recipient, the author creates one last event with the message set to a stop marker M_s and the authenticator 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) in state (see Section 4.3). The stop marker and authenticator are necessary for enabling recipients to distinguish between the correct stop of sending messages and modifications by an attacker, as noted by Ma and Tsudik [21]. After the event with the stop marker and authenticator message has been inserted, the author can delete the entry in the `BSD` state table for the recipient. For each registered recipient who has not had a last event inserted, the author has to keep the recipient entry in the state table at least until the retention time of the Balloon (as set in `BSD`) has past.

4.2.4 Extend

The author has the ability to *extend* a registration made by a recipient. Extending a registration for a recipient is done in three steps:

1. Given $\texttt{pk}$, generate a *blinded* public key $\texttt{pk}'$ using the blinding value b, where b is randomly generated.
2. Run the `register` protocol, as described in Section 4.1.2, using $\texttt{pk}'$ to initiate the protocol.
3. Concatenate the result from step 2, together with blinding value b from step 1 and the extend marker M_e, as a message and send it as an event to the recipient.

An extension, as a consequence of step 2, is a protocol between two parties. The initiating author can either run the protocol with itself, or with another author, where the initiating author takes on the role of the recipient. Running the protocol with itself serves two purposes:

[3] Snapshots are referred to as "commitments" in CONIKS.

- First, this introduces new randomness for the recipient, recovering *future events* from the limited compromise of a passive adversary in the past.
- Secondly, this enables the author to register the recipient in a new Balloon without interacting with the recipient (due to, e.g., the current Balloon being close to full or the retention time is too soon).

When the initiating author runs the protocol with another author, then the initiating author is in effect registering the recipient for receiving messages from another author. This enables non-interactive registration in distributed settings, once the recipient has registered with at least one author.

We blind the public key of the recipient for three reasons: 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). Secondly, Insynd uses the public key as an identifier for the registration of a recipient. For each registration, we need a new identifier. Last, but not least, the blinding approach (compared to, e.g., just having the author create a new key-pair which would still fit our adversary model) 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.

4.3 Reconstruction

A recipient uses two protocols to reconstruct its messages sent by the author: `getEvent` and `getState`. We first explain how to get events using the `getEvent` protocol, then how to verify the authenticity of the events with the help of the `getState` protocol, and end by discussing some options for minimising information leaks during reconstruction.

4.3.1 Getting Events

To download its i:th event from the server, the recipient calculates the event identifier e_i^{ID} using k_0 and `pk` from the registration (as described in Section 4.1.2) together with equation (1) in Section 3.2 to calculate k_i, as follows:

$$k' = \texttt{Hash}\big(\texttt{Hash}(1||k_i)\big) \tag{4}$$

$$e_i^{ID} = \texttt{MAC}_{k'}(\texttt{pk}) \tag{5}$$

The structure of event identifiers is determined by Algorithm 1. The Balloon setup data, `BSD`, contains the URIs to contact the server and author at. To get an event, the recipient initiates the `getEvent` protocol (red box in Figure 1) by sending the identifier e_i^{ID} to the server together with an optional snapshot $\texttt{s}_j$. The server generates a reply by running `B.MembershipQuery` using the provided identifier and snapshot. If no snapshot is provided, the server uses the latest snapshot $\texttt{s}_l$. We want to retain the ability to query for any snapshot $\texttt{s}_j$ for sake of proofs of event time (see Section 4.4.2). The server replies with $(P, \texttt{s}_l, \texttt{s}_p, e_i)$, where P is the proof from the membership query in the Balloon, $\texttt{s}_l$ the latest snapshot, $\texttt{s}_p$ the previous snapshot, and e_i the queried for event (if found). We include the previous snapshot to enable the recipient to verify the signature on $\texttt{s}_l$ in case it does not know all snapshots. This also acts as a gossip mechanism, using the server to distribute snapshots. The recipient can verify the correctness of the reply by using `P.MembershipVerify` together with the snapshot format specified in Section 4.2.2. The recipient continues requesting events until P provides a non-membership proof, then, the recipient verifies the authenticity of all retrieved events.

4.3.2 Verifying Authenticity

The `getState` protocol (green box in Figure 1) plays a central role in determining the authenticity of the events retrieved from the server. The recipient initiates the protocol by sending its public key `pk` to the author.

Upon receiving the public key, the author validates the public key and inspects its state table for the `BSD` in question[4] and checks if it contains an entry for `pk`. If 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 in the state table for `pk`. If there is not, then the author sets $x \leftarrow M_s$. Note that M_s has the same length as (k_i, v_i). The author sends as its reply $\texttt{Enc}_{\texttt{pk}}\big(x||\texttt{s}_l||\texttt{s}_p||\texttt{Sign}_{\texttt{A}_{\texttt{sk}}}(x||\texttt{s}_l||\texttt{pk})\big)$. The reply is encrypted under the provided public key since anyone can initiate the protocol. We want to prevent any party with a recipient's public key to determine if new events are generated for the recipient based on observing the reply (`Enc` is randomised so subsequent ciphertexts do not leak if the underlying plaintext changes or not). The reply contains x (previously set based upon the state table), the latest and previous snapshots $(\texttt{s}_l, \texttt{s}_p)$, and a signature. The signature covers x, the latest snapshot, and the public key. This forces the author to commit to a particular state x for a particular public key `pk` at a particular state of the Balloon $\texttt{s}_l$. The two snapshots are included in the reply both to enable the recipient to verify the signature from the author and to act as a form of gossiping mechanism. Note that this gossiping mechanism is weaker than the gossiping taking place as part of the `getEvent` protocol, since the author knows which recipient is performing `getState`.

The recipient decrypts the reply, verifies the signature and latest snapshot. With the list of events downloaded as described in Section 4.3.1, the recipient can now use Algorithm 2 to decrypt all events and check the consistency between events and the reply from `getState`. Steps 1–8 decrypt all events using the nonce, event key, and authentication key generation determined by Algorithm 1. 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 `getState` should have returned M_s. If no stop marker is found in an event, then the reply from `getState` should match the calculated state in the form of (k, v) (step 11). For the verification algorithm to correctly check consistency between events and `getState`, the non-membership proof that caused the recipient to stop downloading events must be for the same snapshot as the reply from `getState`. If the snapshots do not match, then events may have been inserted after in subsequent snapshots causing the verification to incorrectly fail.

For each extension marker, M_e, found in the message of a decrypted event, the recipient simply reruns the reconstruction steps outlined in this section after generating the blinded private key.

4.3.3 Privacy-Preserving Download

By downloading its events, the recipient inadvertently leaks information that could impact the recipient's privacy.

[4]The author can determine which `BSD` based upon the URI at the author, $\texttt{A}_{\texttt{URI}}$, the request was sent to.

Algorithm 2 Verify authenticity of events for one recipient.

Require: $(\texttt{pk}, \texttt{sk}), k_0$, 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`.

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

For instance, the naive approach of downloading events as fast as possible risks linking the events together due to time, despite our (assumption of) an anonymous channel of communication. To minimise information leakage, we 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 [26], 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.
- Use private information retrieval (PIR) [14, 17]. This is relatively costly but may be preferable over inducing significant waiting time between requests.
- Since the server is untrusted, their contents could safely be mirrored, enabling recipients to spread their requests (presumably some mirrors do not collude).

For now, we opt for the first two options of adding delays to requests and randomising the order. PIR-enabled servers would provide added value to recipients. We note that the mirroring approach goes well in hand with verifying the consistency of snapshots, since mirrors could *monitor* snapshot generation as well as serve events.

4.4 Publicly Verifiable Proofs

Insynd allows for four types of publicly verifiable proofs: authorship, time of existence, 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 author and time proofs can be generated by anyone, the 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 is the *author* of a particular event, we rely on Balloon to generate a proof. The proof is the output from `B.MembershipQuery` for the event. Verifying the proof uses `P.MembershipVerify` with the provided output.

4.4.2 Time

To prove *when* an event existed. The granularity of this proof depends on the frequency of time-stamped snapshots. The proof is the output from `B.MembershipQuery` for the event from a time-stamped snapshot $\texttt{s}_j$ that shows that the event has position i in the history tree, where $i \leq j$. Verifying the proof involves using `P.MembershipVerify` and whatever mechanism is involved in verifying the time-stamp from the time-stamping authority. The proof shows that the event existed at the time of the time-stamp.

4.4.3 Recipient

To prove who the *recipient* of a particular event is, the proof is (i) the output from `B.MembershipQuery` for the event, and (ii) the event key k' and public key `pk` used to generate the event identifier e^{ID}. Verifying the proof involves using `P.MembershipVerify`, 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 a 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 ciphertext contained in the event ciphertext was generated for an outputted public key (recipient). The proof is (i) the output from `B.MembershipQuery` for the event, (ii) the nonce n that is needed for decryption and used to derive the event key k' (see Figure 2), (iii) the public key `pk` used to generate e^{ID}, and (iv) the ephemeral secret key `sk'` that is needed for decryption (see Section 3.4). Verifying the proof involves first verifying the proofs of author and recipient. Next, the verifier can use $\texttt{Dec}^n_{\texttt{sk'},\texttt{pk}}(c, \texttt{pk'})$ from Section 3.4 to learn the message m. The recipient can always generate this proof, while the author needs to keep the nonce n, public key `pk`, and the ephemeral private key `sk'` at event generation.

5. EVALUATION

Here we summarise the results from our evaluation of Insynd found in Appendix A. We evaluate all properties in the forward security model. For recovery from a time-limited passive adversary, as described in Section 2, we rely on our extension messages (see Section 4.2.4). Once the author has recovered from the time-limited compromise by the passive adversary, the next extension message introduces new randomness by rerunning the `register` protocol for a recipient. We assume that the author, as part of recovering from a time-limited compromise, revokes its old signature key and generates a new one that is associated to the author. If the

author uses an Hardware Security Module (HSM) to generate its signatures, this is not necessary.

5.1 Secrecy

Secrecy of ciphertexts is provided by using emphermal key pairs at encryption in order to seal crypto_box, and hence under the Decisional Diffie-Hellman (DDH) assumption on Curve25519.

5.2 Forward Integrity and Deletion Detection

Insynd provides forward integrity and deletion detection for each recipient's events due to the use of the FssAgg authenticator by Ma and Tsudik [21] and the signature in the `register` protocol. Our evaluation shows that we require an unforgeable signature algorithm, an unforgeable MAC, and a collision and pre-image resistant hash function.

5.3 Forward Unlinkability of Events

Forward unlinkability of events is provided in Insynd in the random oracle model. Both the nonce n used for encryption of the message in the event's payload as the event key k' used to generate the event identifier are derived using a hash function from the current authentication key k. Because both state variables k and v are generated in a forward-secure way (see Sect. 3.2), even an adversary that learns the author's entire state at some point in time will not be able to link past events together. The public keys in state leaks no information about event payloads under the DDH assumption for Curve25519.

5.4 Publicly Verifiable Consistency

This follows directly from Theorem 2 by Pulls and Peeters [27], assuming a collision resistant hash function, an unforgeable signature algorithm, and correctly gossiped snapshots. For Insynd, 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.

5.5 Publicly Verifiable Proofs

The unforgeability of these proofs relies on:

Author an unforgeable signature algorithm and a collision resistant hash function.

Time a proof of author and the security of the time-stamping mechanism. The exact time-stamping mechanism is considered out of scope.

Recipient a proof of author and an unforgeable MAC.

Message a proof of recipient, the pre-image resistance of the hash function.

5.6 Non-Interactive Registration

Our extension messages, as described in Section 4.2.4, does not require the recipient to interact. Blinded recipient identifiers are unlinkable assuming that the DDH assumption is valid for Curve25519.

6. RELATED WORK

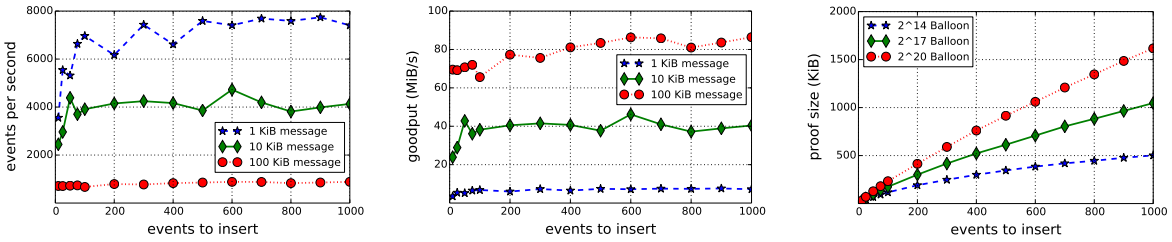
Ma and Tsudik also propose a publicly verifiable FssAgg scheme by using an efficient aggregate signature scheme, BLS [10, 21]. The main drawbacks are a linear number of verification keys with the number of runs of the key update, and relative expensive bilinear map operations. Similarly, Logcrypt by Holt [19] 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 [16] 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. Recipients 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.

PillarBox is a fast forward-secure logging system by Bowers *et al.* [12]. 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 wrt. 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.* [28] has a similar setting to ours with multiple recipients. 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 [29]. We make use of and modify their general framework for some security and privacy properties of Insynd, as presented in Section 5 and Appendix A, to account for our stronger adversary model of an untrusted server, compared to their forward secure counterpart.

Pond[5] and WhisperSystem's TextSecure[6] 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[7] is used by both Pond and TextSecure. Axolotl is inspired by the Off-the-Record Messaging protocol [11] and provides among other things forward secrecy and recovery from compromise of keys by a passive adversary. Our extension messages, Section 4.2.4, mimics the behavior in Axolotl, in the sense that new ephemeral keying material is sent with mes-

[5] `pond.imperialviolet.org`, accessed 2015-01-29.
[6] `whispersystems.org`, accessed 2015-01-29.
[7] `github.com/trevp/axolotl/wiki`, accessed 2015-01-29.



(a) Events per second in a 2^{20} Balloon. (b) Goodput in a 2^{20} Balloon. (c) The size of an insert proof.

Figure 3: Three benchmarks related to inserting events. Figure 3a shows the number of events that can be inserted per second in a 2^{20} Balloon for different message sizes. Figure 3b shows the corresponding goodput in MiB/s for the data in Figure 3a. Figure 3c shows the size of the insert proof in KiB based on the number of events to insert for different sizes of the Balloon.

sages. Note that the goal of Insynd is for messages to be non-repudiable, unlike Pond, TextSecure and OTR who specifically wants *deniability*. Insynd achieves non-repudiation through the use of Balloon and how we encrypt messages, as outlined for proofs of messages in Section 4.4.4.

7. PERFORMANCE

We implemented Insynd in the Go[8] programming language. The source code and steps to reproduce our benchmarks are available at `http://www.cs.kau.se/pulls/insynd/`. We performed our benchmarks on Debian 7.8 (x64) using an Intel i5-3320M quad core 2.6GHz CPU and 7.7 GB DDR3 RAM. The performance benchmarks focus on the `insert` protocol since the other protocols are relatively infrequently used, and reconstruction time (as described in Section 4.3) is presumably dominated by the mechanism used to protect privacy when downloading events. For our benchmarks, we ran the author and server on the same machine but still generated and verified proofs of correct insertion into the Balloon.

Figure 3 presents our three benchmarks, 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 3a shows the number of events per second for different message sizes in a Balloon of 2^{20} events. Clearly, the smaller the message the more events (and therefore potential recipients) can be sent 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 3a, Figure 3b shows the goodput (not including 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. Pulls *et al.* [28], for their transparency logging scheme, generate events in the order of tens of milliseconds per event. Ma and Tsudik, for their FssAgg schemes, achieve event generation (signing) in the order of milliseconds per event [21] (using significantly older hardware than us). Marson and Poettering, with their seekable sequential key generators, generate *key material* in a few microseconds [22]. For PillarBox, Bowers *et al.* [12] 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. Insynd therefore improves greatly on related work on transparency logging, and shows comparable performance to state-of-the-art secure logging systems. Figure 3c shows the size (in KiB) of the proof of correct insertion provided by the server for different Balloon sizes. For a 2^{20} Balloon, the proof size is ≈2KiB per event, requiring messages to be at least 2KiB for a 1 : 1 ratio between data sent to the server (the events) and the reply to the author (the proof).

[8] `golang.org`, accessed 2015-02-12.

8. CONCLUSIONS

Insynd provides a number of privacy and security properties for one-way messaging. 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 is built around Balloon, a forward-secure append-only authenticated data structure by Pulls and Peeters [27], inspired and built upon a history tree system by Crosby and Wallach [16]. For forward-secure key generation, Insynd borrows from FssAgg by Ma and Tsudik [21] and forward-secure SKGs originating from Schneier and Kelsey [4, 29]. Finally, Insynd shares similarities with Pond, TextSecure, and OTR [11], by evolving key material and supporting "self healing" in case of time-limited compromise by a passive adversary. Insynd offers comparable performance for event generation to state-of-the-art secure logging systems, like PillarBox [12].

The use of recipient-specific FssAgg authenticators acts as a deterrent for an adversary to create snapshots that modify or delete events sent prior to compromise. In future work, we would like to strengthen this deterrent by enabling recipients to create *publicly verifiable proofs of inconsistency* in case of inconsistencies between the FssAgg authenticator and the snapshots produced by the author. This would enable recipients to publicly shame misbehaving authors, and presumably act as an even stronger deterrent. We think that recipient-specific verifiability coupled with publicly verifiable proofs of inconsistency is a promising approach for enforcing proper gossiping of snapshots.

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] A. Anagnostopoulos, M. T. Goodrich, and R. Tamassia. Persistent Authenticated Dictionaries and Their Applications. In *ISC*, volume 2200 of *LNCS*, pages 379–393. Springer, 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, A. Desai, D. Pointcheval, and P. Rogaway. Relations Among Notions of Security for Public-Key Encryption Schemes. In *CRYPTO*, volume 1462 of *LNCS*, pages 26–45, 1998.

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

[5] D. J. Bernstein. The Poly1305-AES Message-Authentication Code. In *FSE*, volume 3557 of *LNCS*, pages 32–49. Springer, 2005.

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

[7] D. J. Bernstein. Extending the Salsa20 nonce. In *Symmetric Key Encryption Workshop*, 2011.

[8] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. *J. Cryptographic Engineering*, 2(2):77–89, 2012.

[9] 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.

[10] D. Boneh, B. Lynn, and H. Shacham. Short signatures from the weil pairing. *J. Cryptology*, 17(4):297–319, 2004.

[11] 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.

[12] 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.

[13] 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.

[14] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. Private information retrieval. In *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*, pages 41–50. IEEE Computer Society, 1995.

[15] C. Coronado. On the security and the efficiency of the Merkle signature scheme. *IACR Cryptology ePrint Archive*, 2005:192, 2005.

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

[17] 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.

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

[19] J. E. Holt. Logcrypt: forward security and public verification for secure audit logs. In *Australasian Symposium on Grid Computing and e-Research (AusGrid) and Australasian Information Security Workshop (AISW)*, volume 54 of *CRPIT*, pages 203–211. Australian Computer Society, 2006.

[20] B. Laurie, A. Langley, and E. Kasper. Certificate Transparency. RFC 6962, June 2013.

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

[22] G. A. Marson and B. Poettering. Even More Practical Secure Logging: Tree-Based Seekable Sequential Key Generators. In *ESORICS*, volume 8713 of *LNCS*, pages 37–54. Springer, 2014.

[23] C. U. Martel, G. Nuckolls, P. T. Devanbu, M. Gertz, A. Kwong, and S. G. Stubblebine. A General Model for Authenticated Data Structures. *Algorithmica*, 39(1):21–41, 2004.

[24] 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.

[25] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Consulted*, 1(2012):28, 2008.

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

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

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

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

[30] S. Trabelsi, J. Sendor, and S. Reinicke. PPL: primelife privacy policy engine. In *Symposium on Policies for Distributed Systems and Networks*, pages 184–185. IEEE Computer Society, 2011.

APPENDIX

A. DETAILED EVALUATION

Where possible, we make use of the model of Pulls *et al.* [28], with minor modifications. Given the stronger adversarial setting of Insynd (untrusted server), and the fact that a state per recipient is kept at the author as opposed to the server, the `CorruptLogServer()` oracle is replaced by a `CorruptAuthor()` oracle.

A.1 Secrecy

Secrecy of ciphertexts in Insynd can be modeled as in [28], if the adversary is also given a decryption oracle for target ciphertexts. The latter models that adversaries must not gain any advantage toward learning plaintexts for ciphertexts other than those from selected events for which publicly verifiable proofs of message (or the values stored by the author for producing these proofs afterwards) are released.

THEOREM 1. *Insynd provides computational secrecy of ciphertexts under the DDH assumption on Curve25519.*

Specifically for the used `crypto_box` (in the default mode of operandus), we have to take into account that the adversary, by invoking `CorruptAuthor()`, would learn the author's private key which used to seal `crypto_box` and the current values of the nonces for each recipient[9]. By generating an ephemeral key pair for every encryption (effectively randomising the author's private key), this risk at the author is mitigated. The output of the ephemeral public key as part of the ciphertext obviously has no impact on the security of `crypto_box`. The fact that a publicly verifiable proof of message outputs the ephemeral private key, and the nonce used in `crypto_box`, does not affect the secrecy of other ciphertexts since this private key is randomised for every encryption.

A.2 Forward Integrity and Deletion Detection

Insynd provides forward integrity and deletion detection for individual recipient's events thanks to the use of the FssAgg authenticator by Ma and Tsudik [21]. The verification is "all or nothing", i.e., we can detect if all events are authentic or not, not which events have been tampered with.

THEOREM 2. *Given an unforgeable signature algorithm, an unforgeable MAC, and a collision and pre-image resistant hash function, Insynd provides computational forward integrity and deletion dection.*

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, as described in Sections 3.2 and 4.1.2. Furthermore, the author signs the initial key and value together with the `BSD` and public key of recipient. This forces the author to commit to the existence of either *state* (in the form of the initial key and value) or at least one event for the recipient (the last event with the stop marker and authenticator) assuming an unforgeable signature algorithm. These steps corresponds to the log file initialization in the Ma and Tsudik construct.

After registration, but before the last event, the author must keep state for the recipient. The state is in the form of the current key k and authenticator v. In Algorithm 1, steps 4–5 authenticates the entire event into v and overwrites the key k. This construction is identical to the Ma and Tsudik privately verifiable FssAgg construction that is provable forward-secure given a collision resistant hash function and an unforgeable MAC [21].

After the last event, the author has deleted state. The last event, as described in Section 4.2.3, contains a stop marker M_s and the authenticator v before the last event was generated. Since the verification algorithm, Algorithm 2, verifies the authenticator in the case of state (step 9) and no state (step 7), this is the same as in the previous case after registration but before the last event. Compared to the privately verifiable FssAgg construction, we store the authenticator in an event instead of state. This is irrelevant for security, since we still verify the authenticator.

One last complexity for us to take into account are proofs of messages, as described in Section 4.4.4. A proof of message consists of a membership query for an event, the nonce n, a public key `pk`, and the ephemeral secret key `sk'`. This enables an adversary to change the ciphertext for the event using `crypto_box` such that the decryption is successful, since the adversary learns `sk'` and n. This leaks no information about the k used to derive n (see Figure 2), due to the pre-image resistance of the hash function. Therefore, the adversary cannot forge the authenticator, even if it can replace the last event with the stop marker and authenticator with an arbitrary message.

A.3 Forward Unlinkability of Events

It should be hard for an adversary to determine whether or not a relationship exists between two events in one round of the `insert` protocol, even when given at the end of the `insert` protocol the entire state of the author: (`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 unlinkablilty 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 easily be modeled as in [28], where the adversary can only create users before each round of the insert protocol and is limited to finding a relationship between two events within one round. In their definition, the adversary has to output a guess for the bit b, which does not imply one has to know the recipient for each message. Note however, that this definition also covers that it is hard for the adversary to link a user to one particular event. For the following sequence of oracles calls:

- $vuser \leftarrow$ `Drawuser`(A,B)
- $e_1 \leftarrow$ `CreateEntry`$_b(vuser, message)$
- `Free`$(vuser)$
- $vuser \leftarrow$ `Drawuser`(A,C)
- $e_2 \leftarrow$ `CreateEntry`$_b(vuser, message)$

[9]Insynd generates nonces for encrypting events with a forward secure construction, however we do not rely on it for (forward) secrecy.

- `Free`($vuser$)

finding a relationship between e_1 and e_2 implies that the bit $b = 0$. The events for which a publicly verifiable proof of recipient or message is available (or for which the author stored data to be able to make these proofs) can be modeled by creating these event by invoking $\texttt{CreateEntry}_b()$ for a *vuser* that was drawn by selecting the same recipient as left and right argument, e.g. `Drawuser`(A,A). As such the adversary does not gain any information towards the value of the bit b. Before outputting its guess, the adversary is allowed to make a call to the `CorruptAuthor()` oracle.

THEOREM 3. *Insynd provides computational forward unlinkability of events within one round of the* `insert` *protocol in the random oracle model.*

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 key for the recipient at the time of generating the event.

In the random oracle model, the output of a hash function is replaced by a random value. Note that the output of the random oracle remains the same when the same input is applied. By assuming the random oracle model, the key to the one-time unforgeable MAC function and the nonce as input of the encryption[10] 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.

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 [4, 29, 22]. Hence the adversary is not able to learn any past values of the current authentication key. 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 as assumed in Section 3.4, because the outputted ciphertext is encrypted using a symmetric key that is derived from both the Diffie-Hellman value and the nonce (which has sufficient entropy to base our security on). 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.

[10]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 `sk'` used to seal `crypto_box`. This implies that the adversary will also not gain any advantage from learning the corresponding ephemeral public key `pk'`, as part of the ciphertext.

Finally, 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. This follows directly from the used Forward-Secure Sequential Aggregate (FssAgg) authenticator [21].

A.4 Publicly Verifiable Proofs

We look at each proof individually.

A.4.1 Author

A proof of author for an event is a `B.MembershipQuery` in a Balloon for the event. A membership query in Balloon is an authenticated path in a Merkle tree from a signed snapshot by the author. The signature cannot be forged, since the signature algorithm is unforgeable, and the security of an authenticated path in a Merkle tree has been shown by Coronado to reduce to the collision resistance of the hash algorithm [15].

A.4.2 Time

A proof of time for an event consists of a proof of author from a snapshot that contains a time-stamp from a time-stamping authority. Forging a proof of author involves, as noted previously, forging the signature on the snapshot or the authenticated path to the event. In addition, a the proof of time depends on the time-stamping mechanism, which we consider out of scope. We note that a proof of time only proves that an event existed at a particular point in time as indicated by the time-stamp, not that the event was inserted or generated at that point in time.

A.4.3 Recipient

A proof of event recipient consists of a proof of author, a public key `pk`, and an event key k'. Forging a proof of author involves, as noted previously, forging the signature on the snapshot or the authenticated path to the event. The proof of author fixes the event, which consists of an event identifier e^{ID} and an event payload. The proof is verified by computing and comparing $e^{ID} \stackrel{?}{=} \texttt{MAC}_{k'}(\texttt{pk})$, Forging the event identifier is therefore forging the tag of the MAC since the tag is fixed by the proof of author, which is not possible, since the MAC is unforgeable.

A.4.4 Message

A proof of message consists of a proof of author, a public key `pk`, a nonce n, and a ephemeral secret key `sk'`. The proof of author, public key, and event key $k' \leftarrow \texttt{Hash}(n)$ (see Algorithm 1) results in a proof of recipient. The proof of author fixes the event, that consists of the event identifier and event payload. The proof of recipient fixes the public key, event key and nonce, since the prover provided a pre-image of the event key (the nonce). The payload consists of the ciphertext c and the ephemeral public key `pk'`. The prover provides `sk'`, such that $\texttt{pk'} \stackrel{?}{=} \texttt{pk}^*$, where $\texttt{crypto\_scalarmult\_base}(\texttt{pk}^*, \texttt{sk'})$. This fixes `sk'`, since there is only one `sk'` for each `pk'` for Curve25519[11]. This fixes all the input to `crypto_box_open`: $c, n,$ `pk` and `sk'`, and `crypto_box_open` is deterministic.

[11]There are two points on the curve for Curve25519 such that $\texttt{crypto\_scalarmult\_base}(\texttt{pk'}, \texttt{sk'})$ due to Curve25519 being a Montgomery curve, but Curve25519 only operates on the x-coordinate [6].