CoverDrop White Paper

Daniel Hugenroth[2], Sam Cutler[1], Dominic Kendrick[1], Mario Savarese[1], Zeke Hunter-Green[1], Philip McMahon[1], Marjan Kalanaki[1], Diana A. Vasile[2], Sabina Bejasa-Dimmock[1], Luke Hoyland[1], and Alastair R. Beresford[2]

[1]The Guardian
[2]University of Cambridge

The free press fulfills an important function in a democracy. It can provide individuals with a mechanism through which they can hold powerful people and organizations to account. In previous work, the University of Cambridge developed CoverDrop: a set of extensions to a typical news app which provided a secure and usable method of establishing initial contact between journalists and sources. Since publication, The Guardian and the University of Cambridge have undertaken further work on the design, highlighting additional challenges and shortcomings which needed to be addressed before deployment. This white paper presents an updated design of the CoverDrop system which addresses these issues, and describes the version that The Guardian first deployed in April 2025. You can find this paper and more information at `https://www.coverdrop.org`.

1 Introduction

The Snowden leaks revealed that nation states develop and operate extensive mass-surveillance infrastructure [1]. These capacities have potential for abuse. They could for example be deployed against news organizations and potential sources who are attempting to expose wrongdoing within governments or other powerful organizations. As a consequence, significant care is required to ensure that potential sources are not exposed. However, many leading news organizations today suggest that sources use initial methods of contact that are insecure or are hard to use. Such methods can, for example, fail to adequately protect either the confidentiality of the communication itself, or have the potential to expose the relationship between a source and a journalist to a network observer capable of performing traffic analysis. In prior work, we conducted two workshops with British news organizations and surveyed source communication options at major media outlets. We found a need for a system that provides a secure and usable method of initial contact between potential sources and journalists.

Our central insight towards a practical solution is that news organizations already run a widely-available platform from which they can offer a secure, usable method of initial contact: the news app on a smartphone. The confidentiality and integrity of message content can be assured through widely-available cryptography on these platforms; journalists can be authenticated directly by the news organization; traffic analysis by a network operator or state actor can be thwarted by requiring all the installations of the news app to produce cover traffic, thus hiding whether any given user is in contact with a journalist or not; the system does not require users to install specialist software or tools; and the news app can provide a usable interface which is similar in style and operation to a typical messaging app.

In previous work, we described CoverDrop, a set of extensions to a typical news app which provides a secure and usable method of initial contact between journalists and sources [2]. This white paper presents a revised version of CoverDrop that The Guardian deployed for the first time in 2025. We describe the planned system, implementation details (§2, §4–§6, §8–§10), the requirement analysis (§3) that motivates our design decisions, and discuss the user interface (§7).

To make the document approachable for readers from a variety of backgrounds, we highlight the main areas of sections. GENERAL sections provide important overview of the system and its environment. The cryptographic details and anonymity guarantees are highlighted with PROTOCOL. Since the usability of the system is critical, several sections focus on UX. Software engineering best-practices and specific deployment considerations are marked with SWE.

Our implementation is available as an open-source repository under an Apache 2.0 license: `https://github.com/guardian/coverdrop`. The revised CoverDrop system has previously undergone an extensive audit sponsored by the Open Technology Fund [3]. We are very thankful to the academic experts who have reviewed

previous versions of this white paper and whose valuable feedback helped us to improve both the final system and this document.

2 Overview

GENERAL In this section we introduce the high-level goals, workings, and setting for CoverDrop.

2.1 Goals

The system should allow potential sources to send and receive messages to and from journalists via the news app installed on their own smartphone. The system should support a low-throughput, medium latency, text-only communication channel to permit initial contact. We must protect not only the confidentiality and integrity of message contents, but also hide the fact that any potential source is communicating with a journalist; this requirement must hold against adversaries who can view all network communication on the public Internet as well as compromise all third-party cloud infrastructure. We want a system that is easy to find and easy to use for both journalists and potential sources.

The system should not require a potential source to install custom software or tools. CoverDrop should not impose significant additional battery, network, memory, or processing requirements on the smartphone. The CoverDrop system needs to integrate into the existing news organization infrastructure and workflows.

2.2 CoverDrop overview

This section provides a conceptual overview of the CoverDrop system. The deployed implementation deviates from this high-level description somewhat, but this overview aides the understanding of, and motivates, the eventual design (§4).

CoverDrop consists of four major types of components: the CoverDrop module in the news app, the CoverNode server run "on-premises" (by which we mean that the hardware is housed at a physical location entirely under the control of the news organization), an API service run by the organization in a cloud environment, and an app the journalist can use to read and respond to messages.

The flow of messages in the CoverDrop system is shown in Figures 1 and 2. Messages are generated by potential sources using the CoverDrop module embedded in the news app (§4.1). The CoverDrop module sends these messages to the CoverNode (§4.2) which is operated by a news organization from a secure location such as the organization's main office. In order to prevent a network observer from differentiating between normal use of the news app from network communication caused by the use of the CoverDrop module, all installations of the news app send CoverDrop messages regularly. If the CoverDrop module has no actual message to send (the common case) then a *cover* message is sent which, from the perspective of the network observer, is indistinguishable from a real message. Any real message is stored locally on the smartphone by the CoverDrop module and sent as the next CoverDrop message, i.e. replacing the dummy message which would otherwise have been sent. Consequently a network observer cannot determine whether any communication is taking place and CoverDrop therefore provides the potential source with plausible deniability.

The CoverNode and each journalist has their own public-private key pair. These keys are published by the news organization and available to the CoverDrop module directly so the user does not need know about them. When the CoverDrop module is used for the first time, it generates a new, random public-private key pair for the user.

All real CoverDrop messages sent by the CoverDrop module to the CoverNode include the text written by the potential source as well as their own public key. The message is first encrypted using the public key of the journalist who will ultimately receive the message, then encrypted a second time using the public key of the CoverNode. All dummy CoverDrop messages are encrypted using the public key of the CoverNode. All messages, real or dummy, are arranged to be the same, fixed length. Encryption and length constraints ensure that only the CoverNode can distinguish between real and dummy messages. The CoverNode operates as a mix node: it collects a number of incoming messages together as a batch, decrypts them, places all real messages in the batch into a new random order and publishes them together to the dead drop. All incoming dummy messages are discarded. If there are insufficient real messages to fill the dead drop (the common case) then the CoverNode generates additional dummy messages so that the published dead drop always has the same size. If there are too many real messages to fit into the dead drop, then the excessive messages are buffered and used in the following rounds. The Journalist's app regularly downloads all the messages contained in the dead drop and tries to decrypt all messages using the journalist's private key. Decryption will only succeed if a message was intended for the journalist.

Every successfully decrypted message received by a journalist reveals a user message and the public key of the sending user. This public key allows the journalist to identify whether the message starts a new conversation, or if it belongs to an existing conversation. When replying, the journalist encrypts a message using the user's public key, and then a second time using the CoverNode's

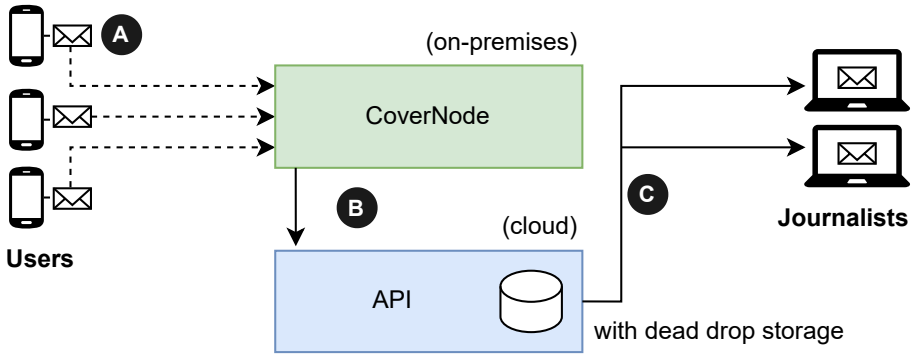

Figure 1: The logical flow of messages from the *user to journalists* in the CoverDrop system. A The users' news reader apps regularly send messages to the CoverNode. B The CoverNode mixes the messages and publishes the dead drops through the API service. C From there the journalists' apps can download them and try to find messages addressed to them by attempting to decrypt them. All communication between the components, i.e. all arrows in this diagram, is assumed to be observable by the adversary.

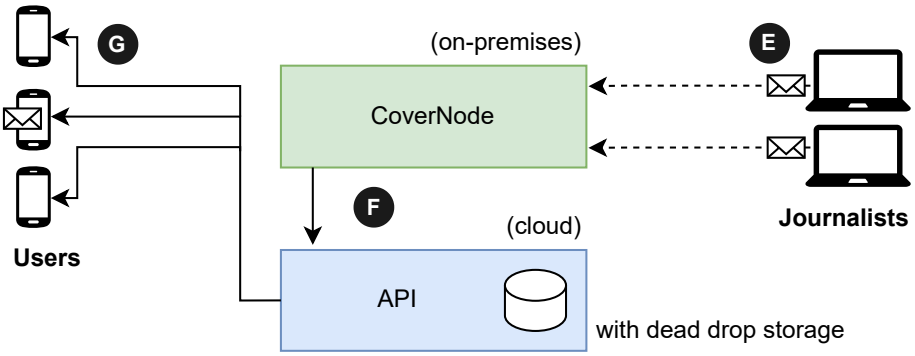

Figure 2: The logical flow of messages from the *journalist to users* in the CoverDrop system. E The journalists' apps regularly sent messages to the CoverNode. F The CoverNode mixes the messages and publishes the dead drops through the API service. G From there the users' news reader apps can download them regularly; when a user unlocks their session the phone tries to decrypt the cached dead drops to find messages addressed to them.

public key. The reply is then sent to the CoverNode which, analogously to the other direction, groups the messages into a batch and periodically publishes them to a user-facing dead drop endpoint.

The news app downloads a copy of recent dead drops on every app start and caches it in local storage on the smartphone. Having separate dead drops for each direction allows us to keep the downloads for the users small as each epoch will only contain few replies from journalists. When the user accesses the CoverDrop module, the module unlocks a private storage area that includes the user's public-private key pair. The CoverDrop module can then use the private key and try to decrypt all messages from the dead drop. If a message is successfully decrypted, it must be for the user, and the CoverDrop module therefore displays the response from the journalist as part of the chat conversation.

2.3 Threat model

We assume a powerful adversary that wants to learn whether a given person is likely to have been in contact with a journalist. For example, the adversary could be an organization that seeks to retaliate after someone exposed wrongdoings. We assume the adversary can monitor all public internet traffic to and from such an individual. Furthermore, we assume that the adversary can confiscate the mobile device of a suspected person—however, they cannot do so without the user noticing. In addition, the adversary can gain full access to all cloud-hosted infrastructure. The adversary can also confiscate on-premises hardware in a powered-off state, but not physically access it while powered on. We assume that the news organization is trusted by the source and that the news organization does not betray this trust by trying to deanonymize sources without their consent. Similar to other apps, CoverDrop only provides limited protection on smartphones that are fully compromised by malware, e.g., Pegasus, which can record the screen content and user actions.

3 Requirement analysis

GENERAL UX The original CoverDrop paper [2] derived its requirement analysis from two workshops with journalists and staff from major news organizations. We use its results as a starting point for the new implementation and augment it by interviewing further stakeholders at The Guardian.

3.1 Roles in the newsroom

We refer to "journalists" for any member of editorial staff who is involved in the production of the copy for a story. This category can then be further broken down into reporters and editors. A reporter generally performs more on-the-ground research into the story, while an editor is responsible for shaping the story and usually takes a more managerial role. We chose interview candidates based on three archetypes: dedicated investigative reporter, collaborative reporter, and investigative editor.

Dedicated investigative reporters spend the majority of their time on long-term, high-impact stories. In contrast, other reporters often spend most of their time writing up many different current events, whereas for an investigative journalist much more time is given to deeply examine individual topics. These reporters have experience with using dedicated tooling and software in a high-security context.

Collaborative reporters are assigned rolling news tasks but also sometimes join an investigation due to their expertise in a particular area. An example might be a banking correspondent who normally reports on the daily news from the finance sector but who has been drafted into an investigation on investment fraud. This archetype is useful to interview because they have experience in both the standard newsroom context, as well as the high security context.

Investigative editors take on more managerial roles including further consideration of the public interest, monitoring the development of investigations and discussing progress. Editors almost always have a reporting background. People in this role have broad overview and experience in many aspects of the newsroom.

3.2 Journalist interviews

Early in the process of developing CoverDrop we ran a series of interviews with editors and reporters at The Guardian. Our interviews were conducted by CoverDrop team members and always included at least one UX team member and a developer. The interviews were semi-structured around a set of questions that ensured coverage of all important areas while allowing the journalist to add extra information as they saw fit. Importantly, this allowed us to discover some unforeseen issues in journalists' existing workflows.

We interviewed five people in total over the span of three months. The recruitment was carried out on the basis of the existing relationships between the team members and the journalists. The interviewers took notes during the conversations and recorded them if the interviewee agreed. Afterwards, the notes were discussed with other team members and common themes were collected in an online document.

Preferred communication channels. Each interviewee had slightly different preferences regarding which

channels they prefer when communicating with sources. However, interviewees across the board highlighted in-person meetings as the preferred way to build mutual trust.

Signal was a clear favorite for digital communication. This was mainly due to the disappearing messages feature and the level of trust the interviewed journalists have in the Signal organization. Journalists often ask sources to move to Signal from other channels such as SecureDrop. In lower risk contexts WhatsApp was sometimes preferred in situations where sources were already using that app, meaning there was no need to ask the source to download another app. Other end-to-end encrypted (E2EE) messaging services that did not require the exchange of phone numbers were generally not used because of low public adoption.

For secure email, journalists we spoke to said that they had had occasion to use end-to-end encrypted email services such as ProtonMail. These were used principally to permit asynchronous communication with sources, or because one or other party did not want to share their phone numbers. No journalist that we spoke to liked PGP and they only used it to communicate with sources in very rare circumstances. In particular manual key management was seen as burdensome and difficult.

Several journalists also spoke of the usefulness of social media direct messaging such as X/Twitter, BlueSky, and LinkedIn. The ability to find people working in a certain context also allowed them to reach out to potential new sources.

SecureDrop was described as involving a large amount of overhead in communicating across the newsroom. Due to its technical complexity, reporters do not have direct access to The Guardian's SecureDrop instance. Instead a small team of more technically adept editorial staff access the server and facilitate the sharing of incoming and outgoing SecureDrop communication across the newsroom. Naturally, this introduces a large amount of latency when responding to sources.

Fear of the cloud. All of our interview candidates expressed reservations around tools that stored data in ways that were not within the journalists' control. Examples of online tools journalists are reticent to use include: cloud storage (Google Drive, DropBox, etc.), collaborative document services (Google Docs, Office365, etc.), and voice transcription services (Otter.ai, etc.). In particular, many of these services have end-user agreements and privacy policies that the journalists considered may not provide sufficient assurances that data will be kept private.

Source anonymity. Anonymity of sources is a topic that came up with both reporters and editors. A journalist has responsibility to protect their sources, and this may require that the source remains unidentifiable by other parties. For sources that reach out to journalists, while the initial contact may be anonymous, as an investigation progresses it is unusual for the source to remain completely anonymous to the journalist. This is because the journalist will need to verify the claims that the source is making. For example, if a source is making claims about a company they work for, then the journalist may need to confirm that the source does actually work for that company. There may be circumstances under which the full identity of a source remains unknown. But in all cases it is important that we clearly separate between anonymity towards an external adversary and anonymity towards the investigative reporter—both in our system design and in communication with the user.

Document upload. Document upload was another topic that came up during our interviews: sources frequently have documents[1] that they wish to share with journalists. While CoverDrop does not allow upload of documents, once the initial contact has been established, the journalist can guide the source on how to use secure document handover channels such as mailing hard copies or using SecureDrop. The use of high-latency channels for one-way document sharing is seen as more acceptable than for messaging. As such, CoverDrop and SecureDrop complement each other.

3.3 UX workshops

Based on our interviews with journalists, the UX researchers built a model that captures the full process and end-to-end user journeys that occur when journalists work with a source. For this we held a series of workshops where we developed the user journeys with stakeholders. These were interleaved with the aforementioned interviews so that insights from each process could be integrated and challenged in the other one. The purpose of the UX workshop and user journey was to understand the scope and boundaries of CoverDrop so that we can focus our development on the most impactful features. A simplified user journey is illustrated in Figure 3 in which we highlight the scope of CoverDrop.

Through these workshops, we concluded that the CoverDrop project needs to be primarily concerned with initial contact from sources, moderation of messages, confirming the validity of the information from the source, and deciding if it is a story worth pursuing. The moderation of messages includes filtering spam and abusive

[1] News organizations are required to have robust policies for the circumstances under which such documents should or should not be retained, and if retained, for how long. The details of these are beyond the scope of this paper.

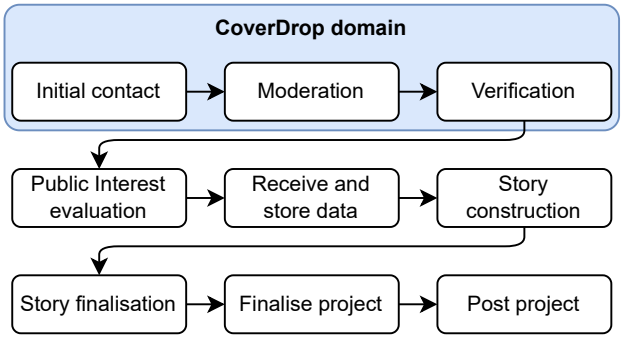

Figure 3: A simplified user journey that shows the lifecycle of an investigation.

messages, and then supporting the journalists in further verification work with the source. Activities beyond this point in the workflow, such as writing public interest memos or receiving any data leaks, are outside the scope of CoverDrop, and provide opportunity for other solutions that can be either digital or non-digital in the newsroom.

4 Architecture overview

GENERAL PROTOCOL This section discusses all involved components including the back-end infrastructure, the module within the news reader app, and the interactions between these components. As such this section is more technical than the conceptual overview (§2.2). The back-end infrastructure consists of the *CoverNode* that provides anonymity for incoming and outgoing messages and various *web services* that publish dead drops and key hierarchies. The existing news reader applications (app) for Android and iOS are extended by a *CoverDrop module*. These app modules are responsible for deniable storage of the local state, sending of cover traffic, and the user interface. Journalists use a *Journalist Client* app on their devices to interact with sources. Figure 4 illustrates all components of the CoverDrop system and the logical flow of data. Based on our threat model some services are run by third-parties, such as the AWS Kinesis message streams and the CDN, while the sensitive ones are deployed on-premises.

4.1 App module

We integrate the CoverDrop functionality in The Guardian's newsreader apps for iOS (§8.3) and Android (§8.4) such that it is deployed through the iOS App Store and Google Play to all the newspaper's app users. Readers can access CoverDrop from multiple entry points. It is advertised through banners in the app, designated call-outs in articles, and dedicated pages on the newspaper's website which link to detailed instructions on how to download the newsreader app and use CoverDrop. In addition, CoverDrop can always be accessed via a dedicated entry in the main menu of the newsreader app. The app module is loosely coupled with the main app allowing us to ignore the other aspects of the app for the protocol discussion. The open-source code allows customizing the branding so that it can be integrated in other news reader apps as well.

Deniable storage (§8.6) is automatically created for *all* users using a randomly selected passphrase when they start the news application for the first time. On every subsequent start of the app, the last-modified timestamp of the file is updated. The contents are always encrypted and padded to a fixed size. Hence, the presence of the file and its observable metadata do not leak information about the active usage of the CoverDrop functionality. We note that, in general, our deniable storage is not resistant against multi-snapshot adversaries who can compare the content of the file at two occasions. However, we rule this out in our threat model (§2.3) as we assume that users notice when their device has been captured. In that case they should follow up with a full device reset.

4.2 CoverNode

The CoverNode, which runs on dedicated on-premises hardware (§10.2), operates as a threshold mix node [4] and provides anonymity for the users towards the journalists and network adversaries. For this it decrypts the outer layer of the incoming messages and then adds them to the mixing process. The CoverNode cannot decrypt the inner layer containing the end-to-end encrypted message between the user and the journalist. This mixing process then yields dead drops which are signed and published via the API. Journalists later download these dead drops to find new messages from users. The opposite direction, from journalists to users, works analogously. We operate multiple CoverNodes so that in case of failures or maintenance, a fail-over can take over. Section 10.1 covers more details.

As the CoverNodes are operated by the news organization, it requires that all users trust that organization to operate honestly. This assumption is covered by our threat model (§2.3). In future designs this trust requirement could be reduced by routing messages through multiple CoverNodes operated by different news organizations. However, it remains out of scope for this version.

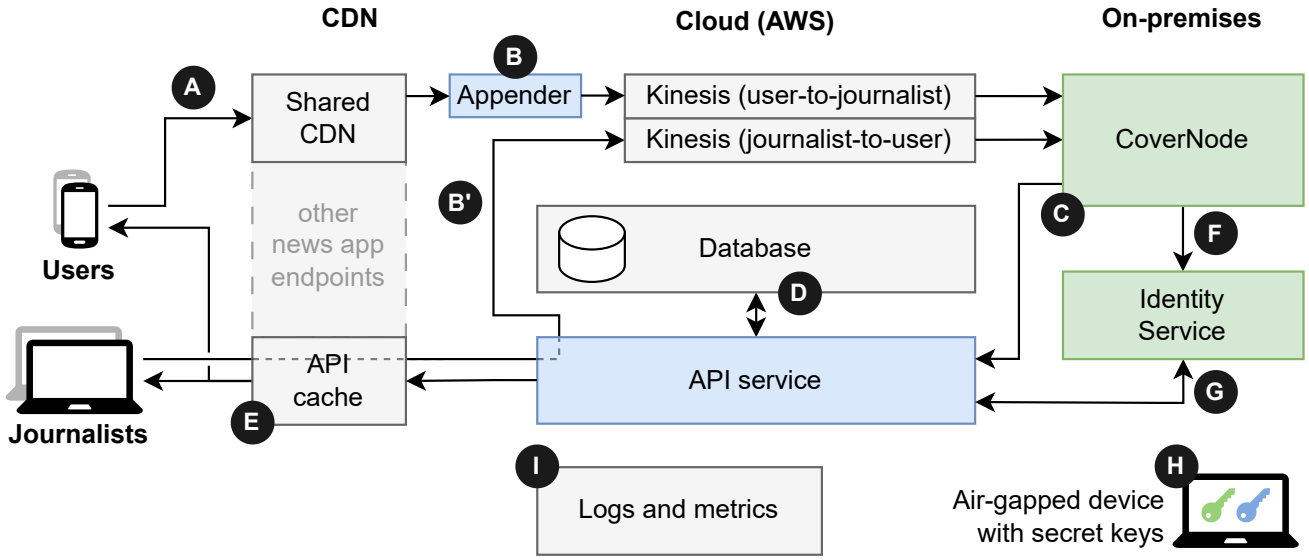

Figure 4: Overview of the full architecture including mobile devices and the back-end components. The web services are either provided by third-parties (gray); written by us and running on third-party cloud infrastructure (blue); or services running on on-premises hardware (green). The arrows in this chart indicate logical flow of messages that include messages and cryptographic key information. The on-premises services do not allow any incoming connection and instead use a pull-based approach.

A The CoverDrop module within the news app on the users' devices sends cover and real messages to the CDN which then B writes them to the Kinesis stream using the Appender microservice. The journalist messages are routed via the API service and then written to the respective stream B'. This allows to authenticate the journalist clients. C The CoverNode mixes the incoming messages and publishes dead drops to the API service. D The dead drops and the key hierarchies are persisted in a database. E User and journalist apps download dead drop and key hierarchies through the CDN which caches the responses. F The CoverNode interacts with the Identity Service to rotate its messaging and identity keys. G The Identity Service protects special provisioning keys that sign key rotation requests including those published by the journalist clients. H All long-term secrets, including the organization key, are kept on an air-gapped device which can create signed requests to install new provisioning keys. I All services generate high-level log information that are collected centrally for monitoring. Those emitted by on-premises services are carefully restricted to limit side-channels.

4.3 Web services

Web services comprise all components that are run on third-party cloud infrastructure. As such we assume that they operate under an honest-but-curious model, i.e. follow the protocol, but without any confidentiality guarantees. The access to these web services is routed through the regular CDN of the news organization. This allows the app to rely on the organization's already trusted SSL certificates and makes it harder to block access to the CoverDrop services without also disrupting access to the other parts of the app. The CDN also caches the published dead drops and key hierarchies to provide reliable and fast responses.

Messages from users are written into a message queue from a CDN endpoint via the Appender microservice. Journalist messages are routed via the API to enforce authentication. The CoverNode reads them from both message queues using a polling loop. It then posts the generated dead drops to an authenticated endpoint provided by the API service. The API service verifies the public signature of the proposed dead drops and then adds them to its long-term storage which in turn makes them available to the clients through the CDN.

4.4 Journalist Client

We deploy a dedicated Journalist Client app to the devices of the participating journalists. It continuously downloads dead drops for the journalist and decrypts incoming messages. Replies by the journalist are encrypted and added to the respective message stream via an authenticated API endpoint. The client also rotates the messaging and identity keys for the journalist by queueing such requests via the API from where they are regularly pulled and processed by the Identity Service.

Similar to the user-facing CoverDrop module in the apps, the Journalist Client encrypts all local storage under a passphrase. Since journalists are expected to have such software on their computers, we can avoid the added complexity of plausibly deniable storage. The storage contains the journalists long-lived identity keys $K_{journalist,id}$, the short-term messaging keys $K_{journalist,msg}$, and the chat histories.

In a next iteration we want to add group communication between these journalist clients. This will enable hand-over procedures (e.g., a source is being delegated to an expert in that area) and collaborative spam filtering. For this the clients will use a Local-First approach [5] and synchronize via an end-to-end encrypted messaging layer such as MLS [6].

5 Key management

PROTOCOL CoverDrop uses a dedicated key hierarchy to certify participants, authenticate messages, and facilitate key rotation. The full key hierarchy (see Figure 5) is stored, updated, and published by the API web service in a single document. We use Curve25519 keys throughout our protocol for ECC signing and key agreement.

5.1 Key hierarchy

In our system the news organization maintains a long-term organization key K_{org} which is generated on an offline computer and stored in a secure location[2]. During rare ceremonies this key is retrieved to sign two provisioning keys $K_{covernode}$ and $K_{journalist}$ which have a medium-term lifetime and are used as signing keys for all CoverNode and journalists keys respectively. This intermediate layer of provisioning keys allows the operators to rotate child keys more easily and frequently as well as onboard new journalists without accessing the organization key. The CoverNode has one identity key $K_{covernode,id}$ that is rotated every three months. It signs messaging keys $K_{covernode,msg}$ of which multiple can exist concurrently and that have a lifespan of two weeks and are used to decrypt and encrypt incoming messages. Similarly, each journalist has one identity key $K_{journalist_i,id}$ and a signed set of messaging keys $K_{journalists_i,msg}$ with a lifespan of two weeks. Journalists can rotate their identity keys by sending a message of a challenge and their new public key, both signed with their current key, to the Identity Service which holds the $K_{journalist}$ provisioning key.

Organization key. Our top-level and most protected key is the organization key K_{org} that acts as a trust anchor for the rest of the key hierarchy. The organization key is generated offline in a key generation ceremony and an initial set of provisioning keys are generated and signed. These secret keys are sealed in tamper-proof bags, and serial numbers are recorded. These sealed copies are then given to designated staff members and stored in secure locations.

The organization key is rotated every year by repeating the ceremony and publishing the new public keys. The relatively short rotation period ensures that processes are regularly practiced. By default, the currently valid organization public keys are included in the apps when they are distributed from the respective app stores. In addition, The Guardian can include hashes of currently

[2] In practice, a news organization might split the secret key into multiple shares under a threshold secret sharing scheme [7] and store these in different safe places to avoid a single point of failure.

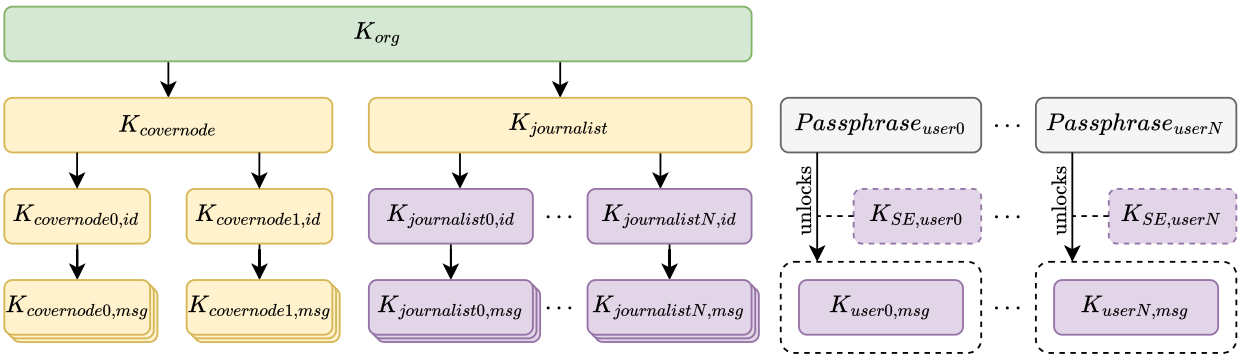

Figure 5: Overview of the full key hierarchy that that is rooted in the organization key K_{org}. Green indicates offline keys, yellow indicates keys kept on dedicated machines, and purple is used for keys on end-user devices. The independent user key hierarchies are stored in the encrypted deniable storage that is unlocked using the passphrase and (if supported) the non-extractable key $K_{user,SE}$ that is lives within the Secure Element.

valid organization keys in the printed newspaper to allow for out-of-band verification.

Journalist and CoverNode provisioning keys. The provisioning keys $K_{journalist}$ and $K_{covernode}$ are used to sign journalist and CoverNode identity keys, respectively, when they are rotated. Their use is closely monitored as part of the key rotation service (§5.2). The secret keys are used automatically by the key rotation service for signing new identity keys. In addition they are used by CoverDrop administrators for on-boarding new journalists or for manual rotations in case of suspected compromise.

CoverNode identity and messaging keys. The CoverNode identity keys are used to prove the identity of the CoverNode service, to verify messaging keys, and to sign the published dead drops. The CoverNode messaging keys are used by both the users and journalists to encrypt messages to the CoverNode. With the CoverNode key material, an adversary would be able to deanonymize sources by decrypting the sent messages. Since we operate multiple CoverNodes for fail-over purposes, each CoverNode runs with their own independent identity and messaging keys that are signed by the CoverNode provisioning key $K_{covernode}$.

We plan to use Trusted Execution Environments (TEEs), such as Intel SGX, as an additional defense-in-depth to ensure that the key material is never exposed in memory unencrypted.

Journalist identity and messaging keys. Identity keys are used to prove the identity of a journalist and to verify their messaging keys. Messaging keys are used by journalists for encrypting messages. They are relatively short-lived and are rotated frequently in order to minimize the damage if a given key is compromised (§6.3). The journalists' keys are stored in a passphrase protected *vault* which is managed by the dedicated Journalist Client app (§4.4) that automatically rotates identity and messaging keys for the journalist. The encrypted vault, which also stores existing messages, can be copied to external storage for backup purposes.

5.2 Key rotation

Journalist and CoverNode identity keys are automatically rotated using a key rotation service. Once a journalist's client decides to rotate a key it generates a new key and sends it to the rotation service along with a timestamp, all signed by their latest identity key—analogously for the CoverNodes. The rotation service verifies that the signature is valid and then returns a signature for the new public key using $K_{covernode}$ or $K_{journalist}$ respectively. Our shared code contains static definitions of the expected lifetimes of each of the keys and their expected rotation periods so that all clients and services follow the same behavior.

A monitoring service, inspired by Certificate Transparency [8], regularly polls the public key hierarchy snapshot to monitor correct rotation of keys. In particular, it sends a warning email to the development team if keys approach the end of their validity time and appear to have failed to rotate automatically. The administrators can also remove keys suspected of being compromised.

6 Protocol details

PROTOCOL This chapter describes the messaging protocol between the clients and its cryptographic operations in detail.

6.1 Cryptographic primitives

For the CoverDrop protocol and its messages we use four cryptographic primitives: a padding scheme for strings and three hybrid encryption schemes. We use the cross-platform cryptography library LibSodium for encryption, signatures, and passphrase hashing. LibSodium is written in C and used by all our components through language-specific bindings which ensures interoperability and allows us to use similar implementations. LibSodium is an "opinionated" library which means that it provides secure defaults, limits choice, and its API primarily exposes high-level primitives. For signatures we use the Ed25519 scheme.

PaddedCompressedString. We call our padding scheme for the text messages `PaddedCompressedString`. It takes a global target length (we use pad_{target}=512) and the user message. The scheme first compresses the user message, then adds a length prefix, and finally fills up the array with `0x00` bytes until it reaches pad_{target}. Natural language text compresses very well allowing the user to send text messages with more than pad_{target} characters. In our implementation we use GZip which typically allows more than 800 characters to fit into a single `PaddedCompressedString`. Other compression algorithms such as brotli or zStd with custom dictionaries might allow for even better compression ratios at the cost of additional library dependencies.

AnonymousBox. The `AnonymousBox` encrypts a message for a recipient using their public key. It maps to LibSodium's `SealedBox` which uses X25519 for key agreement and XSalsa20-Poly1305 for authenticated encryption of the payload.

`AnonymousBox` internally generates an ephemeral public-private key pair for each message and then uses ECDH for deriving the shared secret for encrypting the payload. The ephemeral public key is sent together with the ciphertext. Since a new ephemeral key is generated for each execution, the nonce is implicitly derived from the included public key. Since no long term key material of the sender is included, messages are unlinkable and the sender remains anonymous towards the recipient and third-parties with access to the cipher text. The fresh ephemeral key pair ensures CPA security. The total overhead for `AnonymousBox` is 48 Bytes.

MultiAnonymousBox. The `MultiAnonymousBox` allows encryption of a message for multiple recipients using their public keys. For this it encrypts the message under a fresh secret key k using XSalsa20-Poly1305. The secret key k is then encrypted for each of the recipients under their public key using `AnonymousBox`[3] which are prepended to the final ciphertext. The fresh ephemeral key pairs of the `AnonymousBoxes` ensure CPA security. Since k is not reused, XSalsa20-Poly1305 can use a constant nonce. The overhead for `MultiAnonymousBox` consists of 16 Byte for the Poly1305 MAC and 80 B for each recipient. As we always use it with two recipients (the active CoverNode and its fail-over), the total overhead is 176 Bytes.

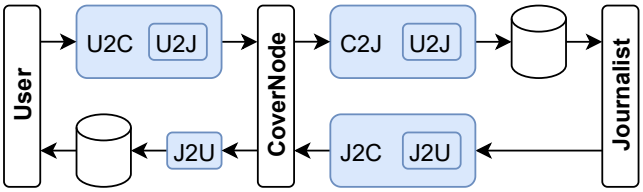

Figure 6: Overview of the message types sent between users, the CoverNode, and journalists. The cylindrical storage symbols represent the dead drops.

TwoPartyBox. The `TwoPartyBox` encrypts a message for a recipient using their recipient's public key and the sender's private key. It maps to LibSodium's `CryptoBox` which uses X25519 for key agreement and XSalsa20-Poly1305 for authenticated encryption of the payload.

Without the need for including an ephemeral public key it is more space efficient than `AnonymousBox`. This scheme also provides authenticity as the recipient can only decrypt the message by knowing and using the sender's public key. The ciphertext is randomized using an included nonce to ensure CPA security. In our `TwoPartyBox` we attach the nonce to the end of the message resulting in a total overhead of 36 Bytes.

6.2 Messages formats

In the CoverDrop protocol messages are exchanged between users and journalists via the CoverNode. For this purpose the inner messages between user and journalists are wrapped into an outer message that is decrypted inside the CoverNode. Figure 6 illustrates the logic flow of messages and their types between the parties. Figures 7 and 8 show the structure of all message types.

We call a message from the user to the journalist `UserToJournalistMessage` (U2J). The message contains the text padded with `PaddedCompressedString` and the user's public encryption key. This content is encrypted using `AnonymousBox` under the messaging key of

[3]The `AnonymousBox` comes with its own authentication tag that is not strictly needed here. Removing it would save 16 Byte per recipient. We opted for composing proven primitives of LibSodium instead of a more custom key agreement.

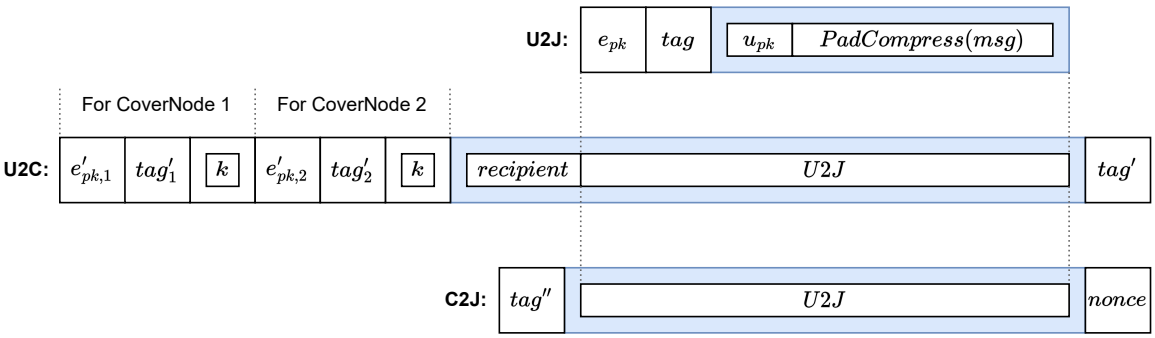

Figure 7: Format of the messages that are being exchanged from users to journalists via the CoverNodes. The U2J uses an `AnonymousBox` to encrypt the contents to the journalists. The U2J message is wrapped in an `MultiAnonymousBox` addressed to both CoverNodes before sent by the app. When exiting the CoverNode, the U2J message is wrapped in a `TwoPartyBox` addressed to the respective journalist.

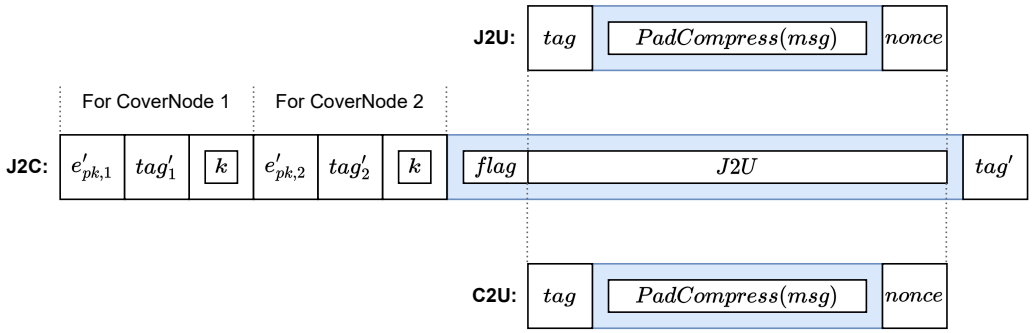

Figure 8: Format of the messages that are being exchanged from journalists to users via the CoverNodes. The J2U use an `TwoPartyBox` to encrypt the contents to the user. The J2U message is wrapped in an `MultiAnonymousBox` addressed to both CoverNodes before being sent by the Journalist Client. The message C2U leaving the CoverNode is identical to J2U.

the journalist. The resulting ciphertext and a recipient tag is wrapped in a `UserToCoverNodeMessage` (U2C) using `MultiAnonymousBox` with messaging keys of two CoverNodes. It is then sent to the Kinesis stream from which the active CoverNodes read the messages. Cover messages are created using the same approach, but set the recipient tag to an empty value and generate an arbitrary U2J.

The CoverNode decrypts the incoming U2C and learns the included recipient tag. If a recipient tag is associated with a journalist, i.e. it is not a cover message, it adds the message to the mixing process (§4.2). Once scheduled for sending, the outgoing message is wrapped in a `CoverNodeToJournalistMessage` which is encrypted as a `TwoPartyBox` using the public key under the recipient tag. This extra layer is required, as otherwise an adversary with network access could observe when a message that they have sent leaves the CoverNode. Such information would be helpful to them for performing n-1 attacks and similar deanonymisation attempts.

For the opposite direction the journalist pad their response using `PaddedCompressedString` and encrypt it using a `TwoPartyBox` under their own private key and the public key of the sender that they learned from the received U2J message. The resulting message is called `JournalistToUserMessage` (J2U). It is wrapped in a `JournalistToCoverNodeMessage` (J2C) using `MultiAnonymousBox` which is then sent to the CoverNode using an authenticated end-point. The J2C payload includes a flag to indicate cover messages. Similar to the messages from users, the CoverNode will unwrap the outer layer of the J2C messages, drop cover messages, mix real J2U messages, and publish user-facing dead drops. No extra layer of encryption is needed between the CoverNode and the users because the adversary cannot send J2U messages via the authenticated end-point. This allows to keep the size of the user-facing dead drops small.

6.3 Forward Security

The original CoverDrop protocol [2] used long-lived keys for encrypting all messages. This is inadvisable in practice because later compromise of key material, e.g., a device is stolen, would allow an adversary to decrypt previously recorded messages. Instead, modern E2EE messaging protocols support Forward Security (FS) [9] which protects previous messages by using short-lived secrets for the actual encryption.

Our protocol provides FS by rotating the messaging keys. For this each journalist generates a new messaging key pair $K_{journalist_i,msg}$ every day, each valid for two weeks. The public keys are signed with the respective identity key $K_{journalist_i,id}$ and published via the API, from which clients always pick the most recent one. The private keys are kept by the journalists and deleted after their expiry date. Similarly, users can rotate their messaging key pair K_{user} by including a new one in a messages signed by the current one.

While there are alternative approaches for achieving Forward Security, we think they are not practical in our setting. Most commonly, end-to-end encrypted messaging protocols derive short-term session keys through Diffie-Hellman key exchange. In our case this approach is unfavourable as the high latency of our system means that interactive key exchange could take up to multiple days if one party is not actively using the CoverDrop feature. In particular, users would have to unlock their CoverDrop session regularly to allow the protocol to make progress, as our threat model does not allow us to store message state outside the encrypted deniable storage. This is in addition to the high-latency mixing by the CoverNode.

We also considered puncturable encryption schemes such as the work by Green and Miers [10]. However, we have not found a mature implementation that is available for all targeted platforms. Also, an attacker can perform a denial-of-service attack against the journalist messaging key by sending many messages that in turn result in many punctures. Ultimately, such a design would also require support for key rotation as per our chosen solution.

7 UX Design

UX A messaging app enhanced with security protections is only secure if it is usable. We explored differences to other (secure) messaging apps that users might be familiar with so that we can give the right directives for correct usage and explain short-comings. In this section, we discuss the main app design and how it is based on the insights that we gained during our UX interviews and workshops.

7.1 Entry points

Users can navigate to the CoverDrop feature through banners in articles and the homepage, as well as a dedicated item in the main menu. Both the app and website already have similar banners that guide readers to existing solutions such as Signal and SecureDrop, and they have been used successfully for many years. Since CoverDrop has no implementation for the web, banners on the website explain how to install the app and navigate to the CoverDrop feature. The always visible item in the app's main menu is important to allow users to easily return to the CoverDrop functionality to check for responses without having to navigate back to the original article.

7.2 Start screen and on-boarding

The start screen of CoverDrop explains its purpose and allows users to browse further offline documentation that comes bundled with the app (see Figure 9). CoverDrop exclusively uses a dark color scheme that The Guardian also uses for their investigative content. The stark contrast to the lighter colors in the rest of the app emphasizes that CoverDrop is an independent part of the app and does not share state or login information with the main areas of the app. At the same time, the contrasting color scheme carries the risk of making it easier to spot CoverDrop screens for a nearby observer via shoulder surfing, but we think that this risk is acceptable in order to give the user confidence that they are in a different section of the app.

When starting a new session, the user is first given a short introduction on how CoverDrop works. This is done via a three-page tutorial (see Figure 10) that is common in many apps. The copy has been carefully chosen to manage the user's expectations, e.g., responses might take a few days or there might not be any, and the user's responsibilities, e.g., to memorize a passphrase that is required for later access to the inbox and keeping messages short and concise.

7.3 Creating a new session

The creation of a new session is the most critical step because it requires the user to memorize a multi-word passphrase. Therefore the first screen repeats the importance and shows all word fields blinded. This forces the user to interact with the "Reveal passphrase" button that shows the individual words. Using a separate "Hide passphrase" button the user can interactively learn all words of the passphrase. Once they feel confident, they are tested on the next page where they have to enter the full passphrase through which they also gain familiarity with the unlocking flow.

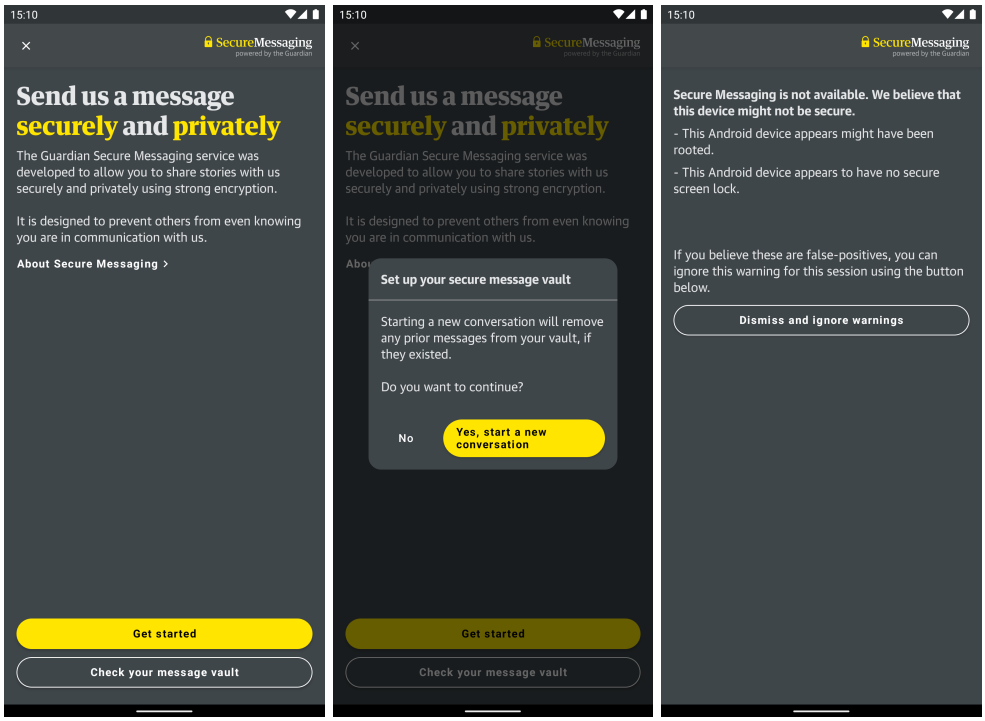

Figure 9: Left: when opening CoverDrop the distinctive color change reminds the user that they are entering a separate part of the app. Middle: when creating a new session, a clear warning explains that this will remove all previous messages. Right: the app shows a warning when it detects an unsafe configuration.

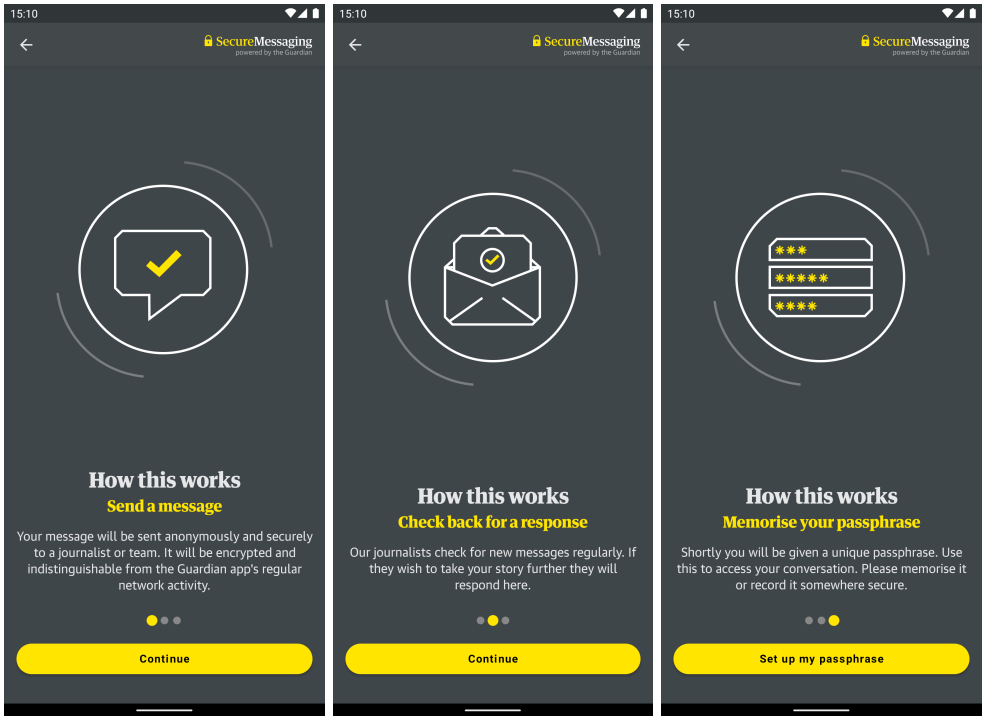

Figure 10: When creating a new CoverDrop session, three tutorial screen introduce the user to its operation and security guarantees.

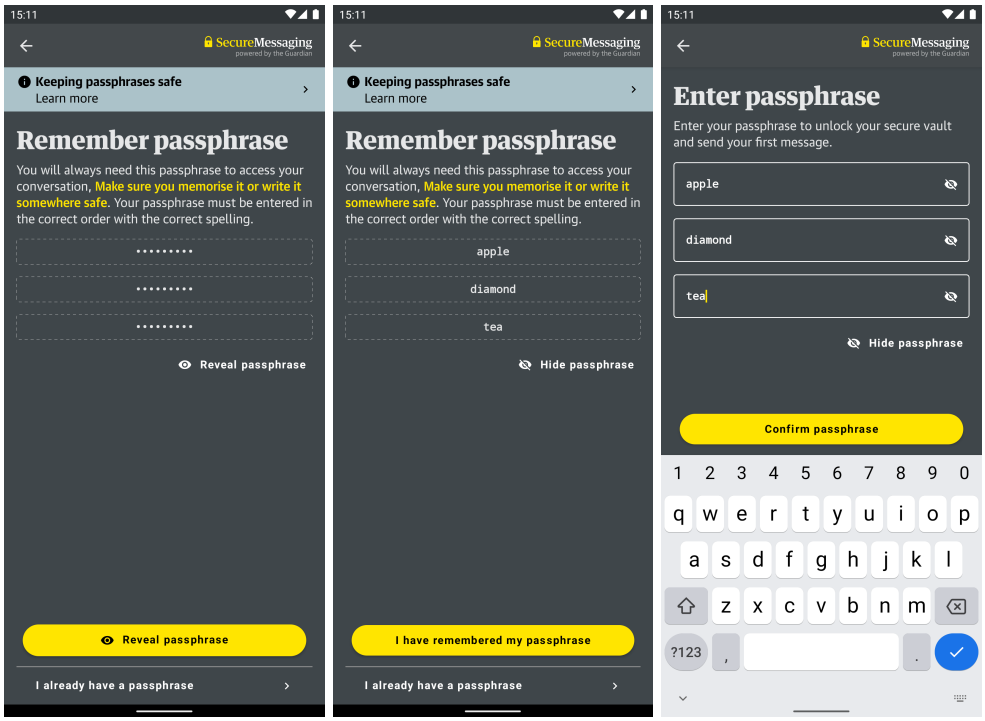

Figure 11: When creating a new CoverDrop session the user first memorizes a generated passphrase and then confirms it by typing it on a new screen.

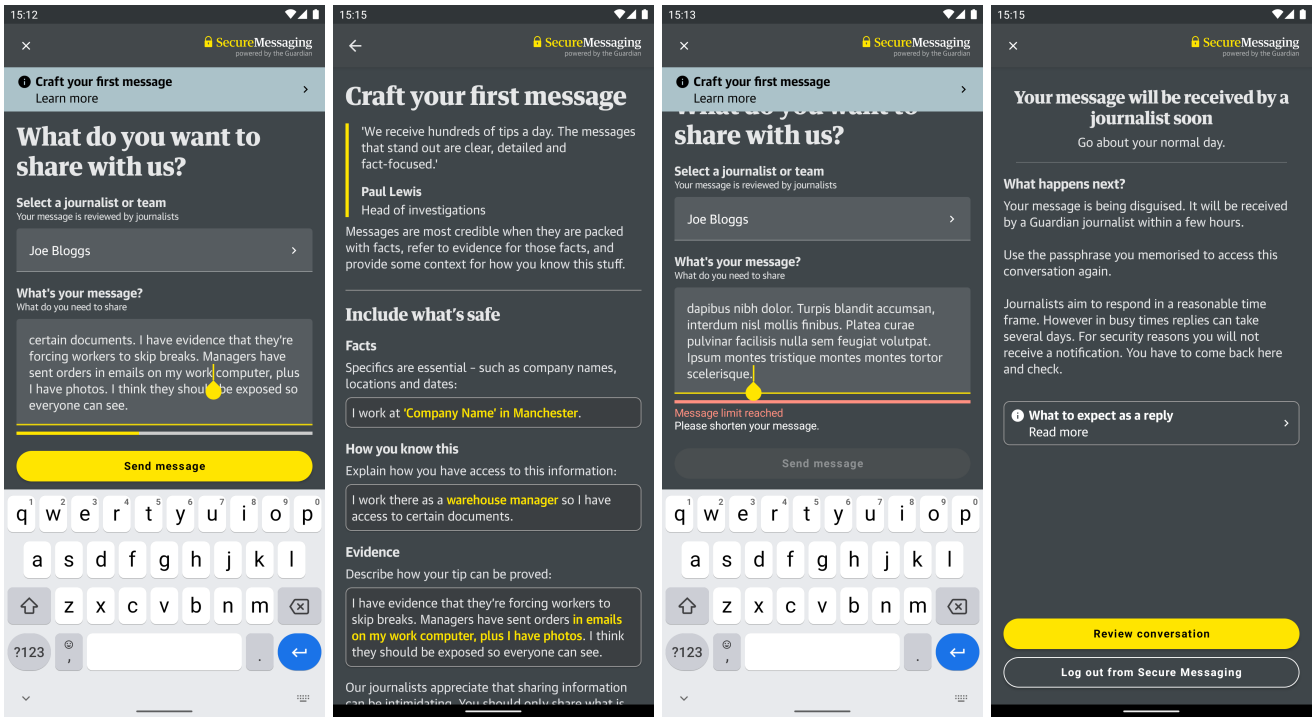

Figure 12: When sending the first message a form guides the user. The linked help screen explains what is important in an initial message and provides examples. If the text is too long, an error is shown. A confirmation screen repeats important information on how the system works.

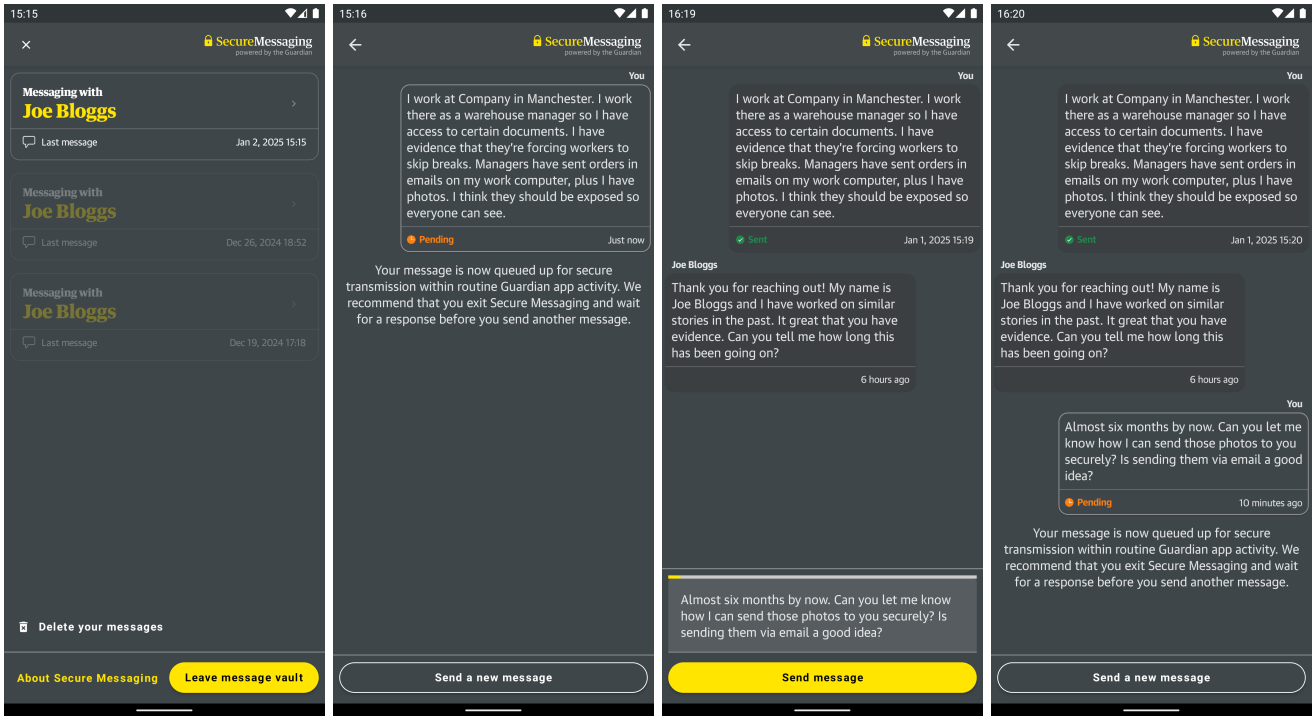

Figure 13: When returning to CoverDrop the inbox shows the currently active session. In the next screen we see the state of the thread right after the first message has been sent. The user then revisits the screen 24 hours later to find a response by the user. They then send a reply.

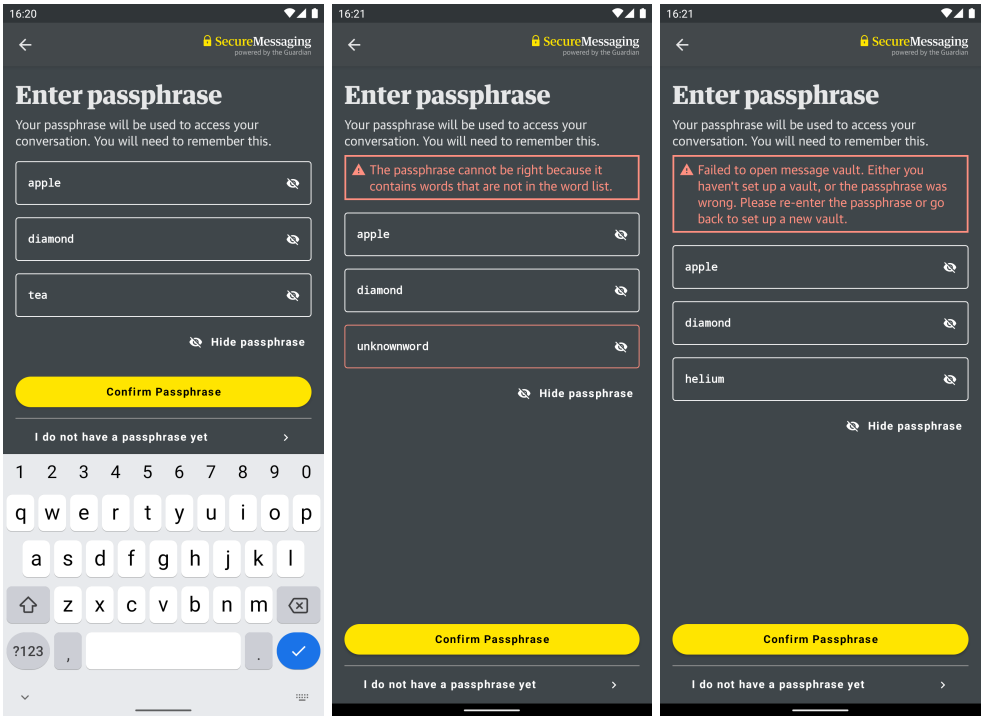

Figure 14: When entering the password we highlight words that are not in the word list and hence likely a result of a mistake. If all words are valid, but the passphrase is wrong, a generic error message is shown that makes clear that failing to unlock does not imply that CoverDrop has been used before.

7.4 Message composer

Our conversations with staff working with existing anonymous source communication channels highlighted that a common problem was a lack of useful information in the initial message sent by the source. This can make it difficult to ascertain if the source is worth speaking to, or which subject matter expert in the news organization should speak with them. Sources are often insufficiently specific, e.g., "I know about a major corruption story". Or they can wrongly assume the recipient knows the context of their claim, e.g., "Mr Smith was lying on the news yesterday. It was 43 people".

In order to help the source provide more helpful information up front, for the very first message that gets sent, we prompt the user to provide some context and provide examples of good messages via extra help screens. These have been described as helpful by reporters to quickly triage large numbers of incoming messages. This should help the journalist decide if the source is worth a follow-up. The journalists also stressed the importance of being able to take a conversational tone during interactions, in order to develop a human relationship with their sources.

Because all messages have to be the same size, the text entered by the user must be concise. Since we use an compression algorithm, we cannot display a simple number indicating how many characters are left as some text compresses better than other. Instead, we show an interactive progress bar that incorporates the compression step in its prediction (see Figure 12).

7.5 One active conversation at a time

Traditional instant messaging chat applications allow a user to open as many chat sessions with other users as they need. However, CoverDrop conversations are expected to be scarce and short-lived, as well as intended for a limited audience—one journalist or one team. Hence, we decided that the CoverDrop implementation will only allow one active session at a time (see Figure 13), with the others becoming invalidated and read-only as soon as a new session is started or if the user is handed over to a different point of contact. This is shown clearly in the inbox by highlighting the currently active conversation and displaying the inactive conversations smaller and after a header that reads "previous conversations". We believe that this helps the user to understand who they are currently talking to and disincentivizes them from contacting multiple persons in parallel and thus causing extra coordination and deduplication work for the journalists . Even if a source would have two independent topics they want to talk about, it is preferred that this is moderated by one dedicated point of contact.

7.6 Chat threads and message state

Once users have sent their initial message, the source will access the message history through a chat thread user interface (see Figure 13), similar to popular messaging apps such as WhatsApp or Signal. This more familiar interface also allows journalists to take a conversational tone with sources, helping them to build trust with the users. The design of the Private Sending Queue (§8.7) allows us to highlight which messages are still pending to be sent or have been sent already.

7.7 Wrong passphrases do not delete the messaging vault

The original paper [2] suggested that incorrect passphrases should delete the user's vault, regardless of whether there was a pre-existing used mailbox or the generic empty one. However, this adds additional stress to the unlocking procedure and might yield accidental data loss. Most importantly, deleting the mailbox would not add security in our threat model as an adversary could abort the algorithm before the deletion takes place.

Instead we now show a generic warning when the user enters a passphrase that fails to unlock CoverDrop (see Figure 14). In particular, we make it clear in that message that a failed attempt does not imply that CoverDrop has been used before. We believe that this is important in case a user is forced to try to unlock CoverDrop by people who do not understand the security architecture of our system. At the same time we can highlight words that are not in the word list as potential spelling mistakes (see Figure 14). This does not leak any information, as the word list is considered public.

7.8 Disappearing messages

Disappearing messages are a common feature in mainstream secure messaging apps. They provide privacy and plausible deniability since the record of a conversation is removed. We include disappearing messages as a privacy feature in the implementation of CoverDrop as follows: any messages that are sent from or received to the user's CoverDrop account have a limited availability window of two weeks. After this time, the message is considered to have expired and its contents are removed from the chat history. Within the app implementation, this concept is explained to users at multiple points.

8 Mobile apps

SWE PROTOCOL Users and potential sources access CoverDrop through a dedicated section in the newsreader apps for Android and iOS. We chose to develop

CoverDrop only for mobile to benefit from stronger app isolation, a more homogeneous platform, and access to strong hardware security. However, it also means less control over the platform.

8.1 Development approach

During the development process, all app code was developed in a repository independent of the main news reader application. In particular, we created a sample reference application to test the functionality and UI. This allowed to minimize dependencies on the developer team of the main news reader app and thus allow for more agile development. On both platforms the main CoverDrop logic is encapsulated in a core library that provides a simple API to the main app. The API hides the internal state from the integrating developers to minimize the risk of wrong use as it internally checks that methods and data are accessed in the intended manner.

While the core library is managed separately, the UI code is part of the main news reader app repository. Changes to these files will require review by the main app developers to ensure that it fits well with the other UI. Direct integration allows us to re-use resources such as fonts and color definitions in the CoverDrop UI code. The core library remains in an external repository that is included as a sub module. This allows the CoverDrop team to update the protocol without having to go through the UI review process of the main application.

We internally evaluated the implementations for both iOS and Android using the OWASP Mobile Application Security Verification Standard (MASVS) [11]. The app code including both the core and the UI modules has been part of the third-party security audit [3].

8.2 Repository architecture

The persistent storage of the state of the CoverDrop module is divided into a `PublicDataRepository` and a `PrivateDataRepository` as summarized in Table 1. This division enforces clear separation between the public data and sensitive private data.

The `PublicDataRepository` caches the latest published dead drops and published key hierarchy from the API. It is updated on every app start, but not more often than every 24 hours, and old entries are evicted after 14 days. The app downloads of these end-points in parallel to the existing download of the news articles allowing us to make use of the already active network connection and hence reduce our impact on battery life. Since the `PublicDataRepository` is updated by every user regardless of whether they used CoverDrop, its contents leak no information to an adversary about active CoverDrop usage.

`PublicDataRepository`
– Cached dead drops API responses – Cached key hierarchy API responses – Private Sending Queue
`PrivateDataRepository`
– User messaging key pair K_{user} – Current threads and messages – sk for the Private Sending Queue

Table 1: All app state is stored in a `PublicDataRepository` that is publicly readable and a `PrivateDataRepository` that is stored in the encrypted vault under the user's passphrase.

The `PrivateDataRepository` stores all sensitive information tied to an active CoverDrop session[4]. The stored data includes the user's private messaging key pair and all current conversations and message threads. This `PrivateDataRepository` is stored inside the plausibly deniable storage (§8.6) which is padded to a fixed size and encrypted with a passphrase.

8.3 iOS

Following our development approach (§8.1), the iOS code consists of a core and a UI module. The core module also exposes hooks that integrate with the app's lifecycle. On iOS we use swift-sodium as a wrapper library around LibSodium.

Our iOS implementation for the deniable storage utilizes the SecureElement (SE) that is common in modern iPhones. Apple first introduced the SE with the iPhone 5S (released 2013) and provides an API to developers on iOS 13 (released 2019) and newer. For our target audience we estimated near perfect availability. We expand on the implementation details of the deniable storage in Section 8.6.

All local files, such as the data from the `PublicDataRepository` and `PrivateDataRepository`, are stored in the `applicationSupportDirectory` which makes sure they are not visible to the user through the Files app. We also set the `isExcludedFromBackup` flag so that the files are excluded from any iCloud backup and remain locally on the device.

[4] An active CoverDrop session begins once a user has created a storage with a passphrase and includes all subsequent uses under the same passphrase

8.4 Android

We implemented the Android modules using Kotlin and the core module enforces a clear separation with a minimal API interface (§8.1). The code follows the current best-practices from the Android developer guide and adopts a reactive architecture using data repositories, co-routines, and state flow. The reactive design further incentivizes that all logic is encapsulated in the core module and that the UI only displays the derived UI state and emits events.

On Android, Secure Elements (SE) are less widely supported than on iOS. Using the Google Play Console we estimate the support for SEs to be around 50% by filtering for devices with `FEATURE_STRONGBOX_KEYSTORE` support. Due to a large number of devices without SEs, CoverDrop on Android requires a fallback mode for key stretching. We expand on the implementation details of the deniable storage in Section 8.6.

Like most Android apps, The Guardian news reader app integrates with third-party libraries to monitor crashes and performance issues of the deployed application. As such they could potentially leak whether a user is using the CoverDrop feature, e.g., when the name of an activity is included in a crash report. Therefore, we require the integrating app to implement life-cycle callbacks to stop those monitoring libraries when CoverDrop is started. In addition, all CoverDrop files are stored in directories that are excluded from device backups using the respective XML directives.

CoverDrop employs best-effort checks to warn the user when it detects potentially insecure circumstances. A (dismissible) warning screen is shown if, for instance, the smartphone is rooted, another window overlays the CoverDrop activity, or the device has no screen lock. In addition, all CoverDrop windows are marked as `FLAG_SECURE` so that the OS prevents screenshots and hides the activity preview when switching applications.

8.5 Message sending strategy

Mobile operating systems limit the control of mobile apps to schedule background tasks. This allows platforms to batch operations and block infrequently used apps in order to save energy. We confirmed this on both platforms and therefore implemented a custom message sending strategy using a single event after the app was exited. In addition, we added a fallback re-try mechanism that runs the missed event when the app is started the next time and the logs indicate that the background worker has not been executed successfully.

In the original paper, we described a strategy where the news reader app would send a message from an outgoing queue every hour. This simple approach does not work well in practice: it leads to at least one hour delay for outgoing messages, it wastes resources when the phone is not being used, and it cannot be reliably implemented due to the aforementioned restrictions of background processes.

In the new protocol, the news reader app schedules sending multiple messages after a random delay when the user leaves the app. This random delay is drawn from an exponential distribution with a mean delay of a few minutes (typically 10 minutes). This does not leak any additional information to the adversary, as they can already observe the regular traffic of the news app and thus expect that CoverDrop messages will be sent when the user leaves the app. The random delay helps to hide usage of the CoverDrop feature at the end of the session as it introduces an additional unpredictable pause between the last regular news story download and sending the messages which are both traffic events observable by the adversary. Also, this strategy avoids sending messages when the user has not used the app recently and thus limits the burden on inactive users.

8.6 Plausibly Deniable Storage

In the client apps, the CoverDrop library stores all sensitive data, i.e. the content of the `PrivateDataRepository`, in an encrypted container that provides plausible deniability against single-snapshot adversaries. This means that an adversary that captures a smartphone and gains full access to the storage cannot tell whether the user has been using the CoverDrop functionality. We achieve this by padding the content to a fixed length before encrypting and storing it. Also, on the first app start, the app creates a fake instance of this storage by encrypting an empty repository with a randomly chosen passphrase.

A typical approach would derive a key from the user passphrase using, e.g., Argon2, and then use this with an authenticated encryption scheme, e.g., AES-GCM, to encrypt the padded content. However, this approach is vulnerable to offline brute-force attacks, as an adversary can copy the ciphertext and then independently try passphrases on many machines in parallel. Instead in our approach we make the key stretching routine dependent on an unextractable secret that is stored inside the Secure Element (SE). Hence, an adversary is limited by the effective bandwidth of the SE. In turn, this allows us to generate short passphrases for the user.

For the CoverDrop mobile apps we use the Sloth scheme [12] that we have pursued as an independent project and offers important improvements compared to the approach presented in the CoverDrop paper. For iOS we designed a new variant called RainbowSloth that works with the limited API of the Secure Enclave

on Apple devices. For Android, an optional ratchet mechanism provides plausible deniability against multi-snapshot adversaries. The implementations are available as open-source libraries for both platforms.

We choose our parameters as follows to guarantee that an exhaustive search of the password space requires an adversary at least 100 years. We use the EFF word list that consists of $|w| = 7\,776$ words.

On all iPhones and Android devices with SE-support we use Sloth with a targeted key derivation time of $t_{sloth} = 1\,\text{s}$. By setting the passphrase length to three words, we have a passphrase entropy of $\log_2(7\,776^3) \approx 38.77$ bits which translates into $t_{sloth} \times 2^{38.77} \approx 14\,000$ years for an exhaustive search.

For devices without SE-support we have to rely on Argon2 combined with a longer passphrase. Since Argon2 is a memory-hard function, the adversary advantage is defined by their available memory and the chosen memory parameters. We assume that the adversary has access to a cluster with 1000 TiB of RAM and enough CPUs to compute a hash in 10 ms. We set the passphrase length to five words (passphrase entropy: 64.62 bits) and the memory parameter to 256 MiB. In this setting the adversary would need $\frac{256\,\text{MiB}}{1\,000\,\text{TiB}} \times 2^{64.62} \times 10\,\text{ms} \approx 2\,201$ years for an exhaustive search.

8.7 Private Sending Queue

In CoverDrop the messages written by the user are not sent immediately. Instead, a sending event is scheduled after each app exit regardless of whether CoverDrop was used or not. A background worker then reads the user-composed messages from a queue when the sending event fires. In addition, we want to ensure that an adversary, who captures a devices while there are messages in the queue, is not able to gain any information about the number of real and cover messages.

The `PrivateSendingQueue` provides these properties and it also improves efficiency by ensuring that real messages are sent before any cover messages. Its persisted data structure consists of a constant-size queue where the elements are pairs of messages msg_i and tags tag_i. Messages are U2C ciphertexts and the tags are 128 bit long hints that allow the app to identify which messages are real messages only when given access to a secret key sk that is stored in the `PrivateDataRepository`. This is illustrated in Figure 15. If there are no real messages, e.g., when CoverDrop has not been used yet, all items of the queue will be cover messages and all hints will be randomly generated byte sequences. At all times, the queue maintains the invariant that the tag_i is equal to $hash(sk, msg_i)$ if and only if the message is a real message. Let $hash$ be a keyed hash function, such as HMAC, whose output is indistinguishable from random

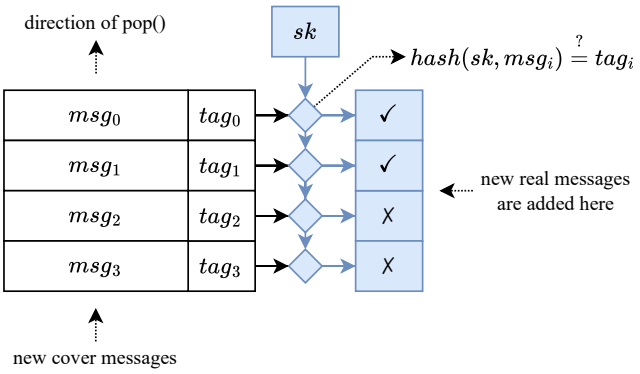

Figure 15: Sample of a private sending queue containing two real and two cover messages. Blue indicates secrets and computations that are only possible with the secret sk that is stored in the `PrivateDataRepository`.

so that the invariant can only be verified with sk.

When the user has written a new message and clicks "Send" the message is first encrypted as a U2C message as described in Section 6.2. Then the sk is used to find the location of the first cover message in the queue, i.e. the lowest index j where the tag does not match the hash. We then replace msg_j with the new message and update $tag_j \leftarrow hash(sk,\ msg_j)$. Compared to simply enqueuing the message at the end of the queue, this ensures that we send real messages in order and before any cover messages. If the message queue is full, i.e. there is no cover message that we can replace, the send operation will fail. However, we believe that this is unlikely for reasonable queue capacity. In our apps we set the queue capacity to 8.

When the background worker is started by the sending event, it removes the front-most message, sends it, and enqueues a freshly generated cover message with a random hint to maintain the size invariant. We note that generating cover messages does not require access to any private information. Hence, dequeueing a message and filling the queue up with cover messages can be done without accessing the `PrivateDataRepository`. In our implementation we repeat this process twice for each sending event and therefore always send the first two messages. This makes sure that we include any clarification or updates that the user might have sent right after their initial message.

By storing the tag_i information with the threads and messages in the `PrivateDataRepository`, we can distinguish whether a recently composed message is still *pending* or has already been *sent*. Note, as with the plausibly-deniable encrypted storage, multi-snapshot adversaries are excluded from our threat model.

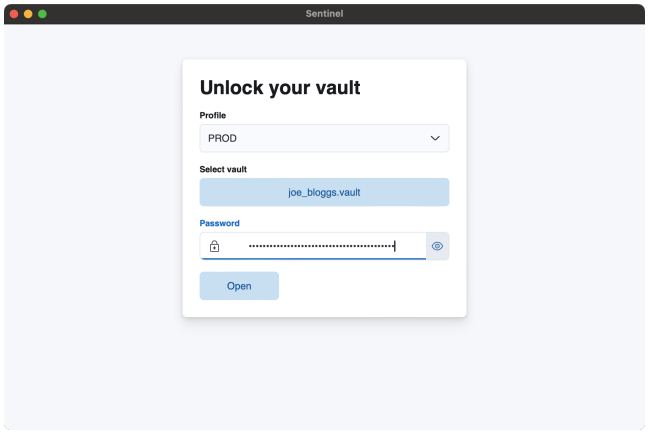

Figure 16: The Journalist Client login screens allows choosing between different vaults.

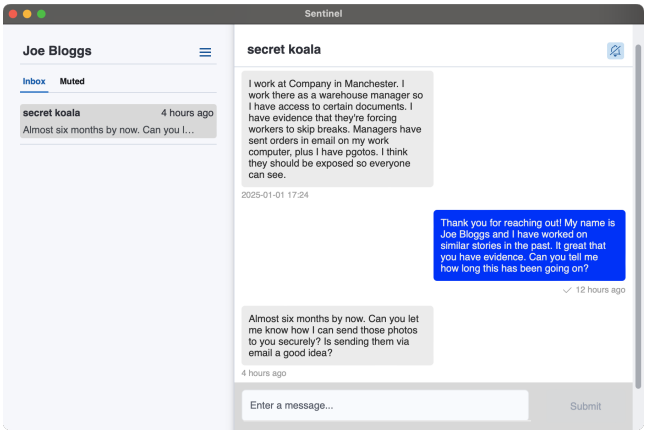

Figure 17: The main user interface draws on familiar UX patterns and expected chat behaviors.

9 Journalist Client

GENERAL SWE UX The Journalist Client provides the user interface for journalists and manages their encrypted local state, the vault. It is implemented using the Tauri framework which allows to re-use a lot of shared Rust code for API interactions, message handling, and cryptographic operations. The front-end is built using React.

9.1 User interface

When starting the Journalist Client, the journalist selects an encrypted vault and enters their passphrase (see Figure 16). By having separate vaults, journalists can manage multiple identities on a single machine. The main user interface (see Figure 17) is similar to messaging apps that journalists are familiar with and have mentioned in our UX workshops (§3.3). Each source is represented by a separate thread and the identifier is derived pseudo-randomly from their long-term public key. Journalists can rename these as conversations progress. The user interface includes typical features such as highlighting unread messages, preview of the content, and the ability to mute conversations.

9.2 Encrypted vault

All state is stored in an encrypted vault using a long, generated passphrase. It uses SQLCipher, an extension of the SQLite database with encryption support, to ensure robustness against crashes while writing. We derive the encryption key from a five word passphrase using Argon2 with conservative parameters. Optionally, the vault can be backed up by copying it to an external storage medium or storing it in a folder that is already being backed up by the news organization's IT team.

9.3 On-boarding and intervention

Journalists receive their original vault after an onboarding procedure during which a member of the admin team will create a fresh vault on an air-gapped machine. This process also generates the passphrase (to be written down on paper) and a digital form for the Identity Service that is signed by the offline journalist provisioning key. The vault and form are copied onto a USB stick and brought over to an internet-connected machine. The form is then presented to the Identity Service via the API which counter-signs the initial identity key and publishes it to the publicly available key hierarchy. Now, the journalist is registered in the system and the vault can be given to the journalist together with the passphrase that is shared out-of-band.

The journalist provisioning can also be used for later interventions. For instance, it can sign new identity keys. This might be required if the journalists machine has been declared compromised or if the app has not been used for so long that the latest identity key expired. It also allows to create signed requests for changing the displayed journalist's name or description.

10 Backend services

GENERAL SWE PROTOCOL The backend comprises the CoverNode (§10.1) and Identity Service that run on hardened on-premise machines (§10.2) as well as the CDN, API service, database, and Appender that run on third-party cloud infrastructure (§10.3).

10.1 CoverNode

The CoverNode is implemented as a Rust application and runs on a hardened on-premises machine (§10.2). As introduced in Section 4.2, it acts as a mix node to guarantee anonymity of the sources towards journalists and external adversaries.

During operations, the CoverNode decrypts the outer layer of all incoming messages to learn the included recipient tag and ciphertext. For most messages the recipient tag will be an empty value which indicates that it is a cover message and should be discarded. The few messages with active recipient tags are added to an internal queue. All messages (cover and real) increase the counter for received messages c.

The mix node has two thresholds t_{max} and t_{min}. It "fires" if the number of messages seen exceeds the maximum threshold $c > t_{max}$ *or* if $c > t_{min}$ and the duration d passed since the previous dead drop was published. The former condition ensures a maximum ratio of incoming messages to outgoing messages while the latter condition ensures that latency is kept low if there are only few messages. We present plausible parameters ranges for production usage in Appendix B.

In both cases, a new batch of size b is created, signed, and added as an output. If the number of real messages in the queue exceeds b, some messages remain in the queue for the next batch. If the number of real messages in the queue is lower than b, the batch is filled up with cover messages. Afterwards, the counter c and duration d are reset. After the dead drop has been published, the CoverNode also persists a Kinesis checkpoint to indicate that all prior messages have been fully processed. We expect that there will be many more cover messages than real messages. Hence, we can choose $b \ll t_{max}$ so that the output dead drops are small.

The CoverNode requires constant access to its key material for decrypting incoming messages and signing outgoing dead drops, which increases the risk of the keys being recovered through physical access. With access to the key material, an adversary could read or send arbitrary messages within the key validity dates. However, the critical operations including decryption, mixing, and signing, are well contained.

Therefore, we plan to run these critical operations including the generation of key material inside a TEE, such as Intel SGX, to ensure that the raw key material is never exposed to the main memory. This provides important defense-in-depth in case of compromised machines and physical access. However, it also makes recovery of failed instances harder which is why we added redundancy in the form of a secondary fail-over CoverNode instance to our architecture.

10.2 On-premises hardware

In our threat model we require that the CoverNode and the Identity Service run on trusted infrastructure that is wholly under the control of the news organization. This ensures that no third party can access key material or learn about the inner state of the CoverNode mixing algorithm. Therefore we deploy these components on our own (rather than cloud) machines that are stored in secure rooms within the news organization's office properties. Such facilities already exist and are regularly used for handling sensitive data during investigations. They are secured with access control and CCTV.

The machines are passphrase protected and run a hardened Linux operating system. All access is logged and the login details are shared only between members of the CoverDrop development team. Overall, our setup is very similar to the existing SecureDrop deployment in the same organization.

All on-premises machines are connected to a dedicated internet line such that they do not share any network infrastructure with the rest of the news organization. In addition, a hardware firewall blocks all incoming TCP/UDP connections. Instead, all on-premises services are using a poll-based strategy where they connect to external services, e.g., fetching entries from the Kinesis streams.

10.3 Cloud services

The API is a public-facing web service which serves public keys for the organization, journalists, and desks. It also stores dead drops and journalist information in a Postgres database. The journalist information includes their id, their display name, a description, and a special flag to mark the virtual desk accounts. Virtual desk accounts are not tied to an individual journalist, but shared among a small group of staff members working on a specific topic. In the app's recipient selection screen they are listed separately, but otherwise messages are encrypted and processed in the same way as messages to regular journalist accounts. The API is versioned to allow evolution of the backend endpoints and message formats. The `GET` endpoints for public key and dead drop retrieval are cached by the CDN for faster access. On the first start and after each update, the API verifies all keys following the chain of trust to avoid serving out-dated, invalid, or unverified keys. However, this is just a convenient verification for the system state and all clients independently verify the keys locally as well. Additionally, a regular scheduled task periodically deletes dead drops older than 14 days from the database.

The API is deployed as an EC2 instance behind a load balancer as the public entry point that accepts requests

from clients. Traffic to the API is only allowed from this entry point, which has a public DNS record associated with it, and the API server configuration disallows all outbound connections with the exception of the Postgres database which lives in the same security group as the API. The Postgres instance only allows connections from resources existing in the same security group, i.e. the EC2 instance for the API. We deploy our cloud-based services using the AWS Cloud Development Kit (CDK) framework. This allows us to provision cloud-based resources using TypeScript code which is committed to version control. All roles and resources live in a dedicated, CoverDrop specific AWS account.

We serve the `POST /user/messages` endpoint as a separate microservice, called Appender, which sits behind the shared CDN domain and a load balancer. The service first performs a basic sanity check to ensure that the base64-encoded payload is a serializable `MultiAnonymousBox` and is of the expected length. All valid looking payloads are then added to the user-to-journalist Kinesis message queue. In case of errors, e.g., we are exceeding the maximum throughput rate, an error is returned to the client app which then keeps the message in its outbound queue and retries later. Messages persist in the stream for 14 days, after which they are automatically deleted. This retention policy has the added benefit of persisting messages in case of a short CoverNode outage. The `POST /journalist/messages` endpoint is provided by the API to allow for authentication using the journalists' identity keys. Otherwise, it behaves analogously to the Appender service and writes the messages to the journalist-to-user Kinesis stream.

11 Development Practices

SWE During development we follow relevant software engineering best-practices. For this we rely on automated continuous integration (CI) and integration tests across the stack. We use Rust as a strongly-typed language to minimize logic errors and avoid memory safety issues when processing untrusted input.

11.1 Source code management

We require all code commits to be signed and implemented mandatory code review for all code commits to ensure the quality and reliability of our codebase. By requiring multiple developers to review each code change, we were able to catch errors and identify areas for improvement before the code was merged into the main branch. This helped us maintain a high standard of code quality and reduce the risk of bugs and other issues that could impact the functionality of our software. It also ensured that everyone in the small team is kept up-to-date on the full architecture as people are implementing the individual components concurrently, e.g., Android app module and API service.

In addition to manual code review, all commits are ran through a CI pipeline based on GitHub Actions. Our CI performs automated static checks for problems with code styling, detects unaudited and known-vulnerable libraries, and runs our full test suite (§11.3). These automated checks already helped us many times to quickly identify regressions and vulnerabilities.

11.2 Using Rust types

We chose Rust to implement the main CoverDrop back-end functionality due to its performance, productivity, safety, and security features. For this all back-end services use types and protocol functions from a shared `common` crate. Rust's focus on memory safety and strong type system makes it an ideal choice for developing secure systems. In particular, Rust's ownership and borrowing system allows for the creation of secure, low-level abstractions without sacrificing performance. This helps prevent common vulnerabilities, such as buffer overflows and data races, which can compromise the security of a system. In addition, Rust's static typing and compiler checks help prevent bugs and catch logic errors before they can cause problems.

We use the *Typestate Pattern* [13] extensively to ensure that only valid objects are reachable. An example usage of this pattern is when a service interacts with untrusted key material such as public keys from the API. Before the keys can be used they must be converted from the `PublishedPublicKeys` type into `VerifiedKeys`. This can only be done by combining the `PublishedPublicKeys` with a `AnchorOrganizationPublicKey` which is compared against the API's provided `OrganizationPublicKey` to confirm the client device is talking to the expected server, and from there we verify the whole key hierarchy. As there are no other accessible constructors or casting operations, we ensure that whenever a `VerifiedKeys` object exists it has been properly verified.

In addition, we use `PhantomData` to tag cryptographic material, such as keys, with their semantic context. For example, the `AnchorOrganizationPublicKey` is an alias for a `SigningPublicKey` tagged with a `Role` type parameter of `TrustedOrganization`. Other examples of other `Role`s include `JournalistIdentity` which are used to verify journalist messaging keys and `CoverNodeMessaging` which are used to encrypt messages to the CoverNode. By enforcing type constraints on all methods returning and accepting the keys, we can make sure that we, for example, never encrypt an out-

going `J2U` message with a key that is not a user public key. Thus this enforces domain separation at compile time at the type level using method type signatures.

11.3 Testing strategy

In order to be confident in our changes we require a robust testing strategy. As is standard industry practice, all smaller components are covered by unit tests. This includes basic functional components such as `PaddedCompressedString` and cryptographic primitives such as `AnonymousBox`. These tests are implemented in all used languages which include Rust (for the backend services), Kotlin (for the Android module), and Swift (for the iOS module).

To test the system as a whole we created a suite of integration tests which run against containerized versions of the production infrastructure. Using this approach we can execute simulations of complex real-world scenarios such as on-boarding a new journalist or sending messages. For time-dependent features such as key expiry we implemented a "time travel" mechanism. This allowed the orchestrator to perform an action then move forward in time and check that the API responses have changed as expected. For instance, we use this to verify the correct behavior of the key rotation service. The same mechanism allows us to use a static set of keys for the testing journalists accounts without having to worry about these keys expiring. Thorough testing is particularly critical in our case as we have configured our clients to not report crashes and log information since these could otherwise reveal active use of the CoverDrop feature. Therefore, the automated tests and manual testing with dedicated debug builds are our only source to identify regressions and bugs.

In order to test the apps against up-to-date versions of API responses we added the ability for the integration tests to emit test vectors that can be copied across to the mobile apps' test modules. These test vectors allow the app developers to implement the same tests as the integration tests module without the added complexity of managing local containers from the iOS and Android test environment. This also ensures byte-level compatibility between all three platforms. In addition, a script within the repository exports constants from Rust, such as the fixed message sizes and key lifetimes, to Kotlin and Swift files so that the constants are always in-sync for all components. At the time of writing the project features 126 tests across all Rust components, 249 tests for the Android app modules, and 142 tests in the iOS workspace.

11.4 Rollout

The rollout of CoverDrop requires additional work outside the CoverDrop system.

In order for our readers to find the CoverDrop tool within the live app we need to update our existing call-outs. These are elements that sit within an article and ask readers to contribute anything they might have to add. The Guardian already has a mechanism for customizing these call-outs, offering a range of options for how a user might reach our journalists. For topics with lower security needs, CoverDrop might not be the best tool. As such, staff will have to be trained on when CoverDrop should be included. When the call-out is viewed on a non-app platform, such as the web, a message should appear prompting the user to use the phone app and find the same article in order to use CoverDrop.

A major part of the rollout will involve training staff on how to maintain the CoverDrop system. Actions such as on-boarding journalists will generally be done without developer involvement, so documentation will be created. Overall we expect that the amount of training required and daily time invested by non-technical staff will be lower than for similar platforms such as SecureDrop. This is mainly because CoverDrop delivers messages directly to journalists and does not require technical experts to facilitate the message transport.

12 Limitations and future work

GENERAL This document describes a practical implementation of CoverDrop. As such, there are known limitations and missing features.

As an open anonymity network, CoverDrop is vulnerable to Sybil attacks where adversaries control multiple clients simultaneously. This can be used to start denial-of-service attacks against the CoverNode by sending many real messages with bogus content. To mitigate this, the news organization can rate-limit the ingress end-point at the CDN per IP address. For this the CDN should generally disallow access via other anonymity networks such as Tor.

However, even a single client can cause harm by sending abusive messages to journalists. We are currently researching extensions to the CoverDrop protocol that allow mitigating these risks. We think that an effective solution will incorporate both traffic limiting measures, e.g., using blinded tokens, and reactive blocking where journalists can report specially-franked messages back to the CoverNode.

The current implementation also misses functionality that remains out of scope for the first version. For instance, users can only initiate conversations with one recipient. We plan to extend the Journalist Client with

collaborative features using a Local-First architecture which will enable multiple journalists to interact with a source. Also, there is no functionality for sharing of photographs, documents, and other files. For now, this is intended to reduce the potential for abuse. However, later versions could include hand-over mechanisms to SecureDrop that allow linking the uploaded files with existing conversations.

13 Conclusion

In this white paper we have presented a practical implementation of CoverDrop. It allows sources to securely and anonymously reach out to journalists. CoverDrop has been designed for allowing bi-directional, medium latency, text-only communication between source and journalists. As such it focuses on the establishing an initial contact, verifying the source and their information, and building trust. Importantly, CoverDrop was designed together with investigative journalists to understand their needs for the system. Where possible, CoverDrop does not require manual key management, but automates key rotation.

The CoverDrop system provides strong security guarantees for potential sources that include confidentiality of their messages, anonymity towards network adversaries, and plausible deniability in case they have to hand over their phone. For this the system relies on hardened on-premises services, such as the CoverNode, and untrusted web services running in a cloud environment for message distribution. The mobile applications use the secure hardware on modern phones to provide strong security while keeping the system usable. The implementation follows best-practices for secure software by using safe languages like Rust, relying only on established cryptographic libraries, incorporating extensive integration tests, and open-sourcing the code for inspection.

Acknowledgments

We are very thankful to the academic experts who have reviewed previous versions of this white paper and whose valuable feedback helped us to improve both the final system and this document. In particular: Richard Danbury, George Danezis, Martin Kleppmann, Leona Lassak René Mayrhofer, Jan Nold, Kenny Paterson, Felix Reichmann, Angela Sasse, and Carmela Troncoso. All errors remain our own. We thank The Guardian (especially Chloe Kirton, Paul Lewis, Bella Purchas, and Noemi Szantai) for their long-term support and important contributions. We thank the University of Cambridge, its Department of Computer Science and Technology, and Nokia Bell Labs for their long-term support. We thank the Open Technology Fund for their generous support and sponsoring a full audit of the CoverDrop system. We thank 7A Security for performing said audit.

References

[1] Ewen MacAskill and Gabriel Dance. NSA files decoded: what the revelations mean for you. *The Guardian*, 2013. `https://www.theguardian.com/world/interactive/2013/nov/01/snowden-nsa-files-surveillance-revelations-decoded`.

[2] Mansoor Ahmed-Rengers, Diana A Vasile, Daniel Hugenroth, Alastair R Beresford, and Ross Anderson. Coverdrop: Blowing the whistle through a news app. *Proceedings on Privacy Enhancing Technologies*, 2022(2):47–67, 2022.

[3] Open Technology Fund. CoverDrop Security Audit Public Report RC 1.2, 2023. `https://www.opentech.fund/security-safety-audits/coverdrop-security-audit/`.

[4] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[5] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 154–178, 2019.

[6] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[7] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[8] Ben Laurie. Certificate transparency. *Communications of the ACM*, 57(10):40–46, 2014.

[9] Colin Boyd and Kai Gellert. A modern view on forward security. *The Computer Journal*, 64(4):639–652, 2021.

[10] Matthew D Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320. IEEE, 2015.

[11] Open Worldwide Application Security Project. Mobile application security verification standard. `https://mas.owasp.org/MASVS/`.

[12] Daniel Hugenroth, Alberto Sonnino, Sam Cutler, and Alastair R Beresford. Sloth: Key stretching and deniable encryption using secure elements on smartphones. *Proceedings on Privacy Enhancing Technologies*, 2024.

[13] Cliff L. Biffle. The typestate pattern in Rust. `http://cliffle.com/blog/rust-typestate/`. Accessed: 2023-06-26.

A Changes

The protocol and implementation presented in this paper differs in important elements from the one that was presented in the original CoverDrop paper in 2022:

- We derive further requirements from interviews and UX workshops (§3).
- We introduce the Journalist Client (§4.4).
- We add a more complete key hierarchy and means for key rotation (§5).
- We modify the message formats and introduce a new wrapping message between CoverNode and journalists (§6).
- We add support for Forward Security through rotation of short-lived messaging keys (§6.3).
- We updated the UI (§7).
- We devise a new message sending strategy (§8.5) and a private sending queue (§8.7).
- We use the Sloth scheme to provide Deniable Storage for both Android and iOS (§8.6).
- We implement all components following best-practices and considered real-world deployment challenges (§11).

B Metric estimates

In Table 2 we present a baseline scenario for usage numbers of CoverDrop. Importantly, the presented numbers have been chosen independently of any existing news reader apps and are only for illustrative purposes. However, we believe that even such reasonably estimated number provider useful indications to the volume of traffic under our choice of CoverDrop parameters.

We assume that the app is installed by 30 million users (about half the UK population) of which up to 5 million are daily active users (DAUs). On average each DAU opens the app twice per day which will mean that they trigger up to two send events (§8.5) with two messages each. All users (DAU and other) download missed dead-drops when they open the app the next time. We expect there will be 10 to 50 active journalists sending on average up to 2 messages per hour.

In the direction user-to-journalist the CoverNode we choose the thresholds $t_{max} = 500\,000$ and $t_{min} = 100\,000$ with $d = 1$ hour and the output size $b = 500$. For the direction journalist-to-user our parameters are $t_{max} = 100$ and $t_{min} = 20$ with $d = 1$ hour and $b = 20$.

Under these assumptions, the system processes up to 850 000 U2C messages per hour sent by the users to the CoverNode resulting in an overall ingress of up to 650 MiB/hour (or 185 KiB/s). As the content of the downloaded information, i.e. public keys and dead-drops, rarely changes, the CDN can handle the vast majority of requests without reaching out to the AWS back-end services. Overall, the download burden for individual users is up to 500 KiB per day which is comparable to a single news story with images.

Metric	**User → Journalist**	**Journalist → User**
General parameters (IN)		
Active senders	$500\,000 - 5\,000\,000$	$5 - 40$
Messages per sender	0.17 msg/h	$1 - 2$ msg/h
Total real messages	$50 - 100$ msg/h	$5 - 15$ msg/h
Message size	800 byte	600 byte
Sent messages	*from user*	*from journalist*
Expected sending rate	$85\,000 - 850\,000$ msg/h	$5 - 80$ msg/h
Expected ingress at CDN	$64.85 - 648.50$ MiB/h	$0.01 - 0.05$ MiB/h
Ratio real messages	$0.01 - 0.06\%$	$19 - 100\%$
CoverNode parameters (IN)		
Parameter threshold t_{min}	100 000 msg	20 msg
Parameter threshold t_{max}	500 000 msg	50 msg
Parameter timeout d	1 h	1 h
Parameter output size b	500 msg	20 msg
CoverNode processing		
Firing rate	$1.00 - 1.70$ h^{-1}	$1.00 - 1.60$ h^{-1}
Mean message delay	$0.29 - 0.50$ h	$0.31 - 0.50$ h
Expected output rate	$500 - 850$ msg/h	$20 - 32$ msg/h
Ratio real messages in output	$10 - 12\%$	$25 - 47\%$
Ingress at the receiving side	*to journalist*	*to user*
Published batches	$1 - 2$ batches/h	$1 - 2$ batches/h
Size of batch	0.38 MiB	0.01 MiB
Total ingress	$0.38 - 0.65$ MiB/h	$0.01 - 0.02$ MiB/h
Download burden per month	$275 - 467$ MiB	$8 - 13$ MiB

Table 2: This table shows the ranges of the CoverDrop configuration and estimates for our expected usage scenario. These are then used to calculate estimates for number of messages and overall traffic.