# Securedrop Report

Michele Orrù

# SecureDrop Analysis Report

This document presents a preliminary review of SecureDrop's security, an open-source whistleblower submission system used by media organizations to securely accept documents from anonymous sources. This review is not exhaustive. The conclusions and observations within are based on the current state of analysis and should be regarded as an initial assessment rather than a definitive evaluation of SecureDrop's security posture.

**Audit Scope.** The focus of this audit was the scheme detailed in securedrop-poc/README.md, deducing a formal security model and attempting to find bugs in the cryptography use. In this report, we outline some formal security guarantees and we highlight some problems that were uncovered during the formalization.

**Overview of Findings**: Overall, the protocol could benefit from better authentication mechanisms for messages sent and received. We suggest techniques to reduce the risk of forgeries while allowing for plausible deniability. The whistleblower's key material is generated correctly but lacks forward secrecy and post-compromise security. It is also vulnerable to preprocessing attacks. We offer an overview of various mitigation techniques.

## Overview of protocol participants

- **FPF (Freedom of the Press Foundation)**: rusted to handle key material correctly. Acts as an authority over the key material used by newsrooms. While there is a technical or legal possibility of compromise, such risks are mitigated in practice through the use of canaries and certificate revocation.
- **Newsroom**: Manages a collection of journalists and acts as an authority over their key material. This includes key material submitted by the journalists themselves.
- **Server**: Responsible for handling messages posted by journalists and submissions. The server handles messages and submissions, with capabilities including delaying, blocking, and replaying messages. It can also copy its state at any given time. The only information at the server's disposal should be the journalists's public keys present in the system, the current number of submissions in the system, and their respective lengths.
- **Users (Journalist and Source)**: Manage their personal key pairs. Communication between them is private and authenticated. The recipient of a message and its content should be known only to the sender and recipient of that message. A compromise of their key material does not affect any messages sent by other participants.

- **External Network Adversary**: Observes the network passively. This adversary cannot access any information about the system's state, including the messages sent/received or the source/destination of a message.

# Cryptographic primitives

Differently from the initial description, we introduce a Key Encapsulation Mechanism that allows to abstract away the encryption of a symmetric key for the messages. This greatly simplifies the analysis and allows for a drop-in replacement with post-quantum primitives. A secure KEM implementation using Diffie-Hellman (as in the draft submitted) is available in RFC 9180.

- **Authenticated Encryption:**
  - Randomness can be generated with $k = \mathrm{Gen}()$.
  - Encryption is denoted as $c = \mathrm{Enc}(k, m)$.
  - Decryption is denoted as $m = \mathrm{Dec}(k, c)$.
- **Multi-key Existentially Unforgeable Signature Scheme:**
  - Key generation is denoted as $\mathrm{SK}, \mathrm{VK} = \mathrm{KeyGen}()$.
  - Signing is denoted as $\mathrm{sig}^{\mathrm{signer}}(m) = \mathrm{Sign}(\mathrm{SK}, m)$.
  - Verification is denoted as $\mathrm{true}/\mathrm{false} = \mathrm{Verify}(\mathrm{VK}, m, \mathrm{sig}^{\mathrm{signer}}(m))$.
- **Key Encapsulation Mechanism:**
  - Key generation is denoted as $\mathrm{sk}, \mathrm{pk} = \mathrm{KeyGen}()$.
  - Encapsulation is denoted as $k, \mathrm{ctx} = \mathrm{Encaps}(\mathrm{pk})$.
  - Decapsulation is denoted as $k = \mathrm{Decaps}(\mathrm{sk}, \mathrm{ctx})$.
- **Key Agreement:**
  - Key generation is denoted as $\mathrm{SK}, \mathrm{PK} = \mathrm{KeyGen}(s)$.
  - The Diffie-Hellman operation is denoted as $k = \mathrm{DH}(\mathrm{A}_{\mathrm{SK}}, \mathrm{B}_{\mathrm{PK}}) = \mathrm{DH}(\mathrm{A}_{\mathrm{PK}}, \mathrm{B}_{\mathrm{SK}})$.
- **Key Derivation Function:**
  - The function is denoted as $k = \mathrm{KDF}(m)$.

# Core Cryptographic scheme

We illustrate a core cryptographic scheme from the Securedrop protocol description. This scheme is more modular and allows for an easier formalization of the functionality desired from the new Securedrop protocol.

The scheme abstracts away the file upload and the choice of the recipients, and allows for a formal analysis of the anonymity requirement asked. Two minor changes tackling small issues are highlighted in red (when removed) and blue (when added).

## Constants

- `CHUNKS` : the size of a file being uploaded. (Not used in the core cryptographic protocol).
- `MAX_MESSAGES` : the number of messages withheld by the server.
- `MAX_TOKENS` : the length of the ciphered messages (in fixed-length tokens of size larger or equal than twice security parameter).
- `MAX_KEYS` : the number of ephemeral keys each user can deposit in the server.

## Keys Setup: `Setup()`

The setup algorithm creates the following key material for the parties participating to the protocol.

- **FPF**:
  - $\mathrm{FPF}_{\mathrm{SK}}, \mathrm{FPF}_{\mathrm{VK}}$: Long-term FPF signing key pair.
- **Newsroom**:
  - $\mathrm{NR}_{\mathrm{SK}}, \mathrm{NR}_{\mathrm{VK}}$: Long-term Newsroom signing key pair.
  - $\mathrm{sig}^{\mathrm{FPF}}(\mathrm{NR}_{\mathrm{VK}}) = \mathrm{Sign}(\mathrm{FPF}_{\mathrm{SK}}, \mathrm{NR}_{\mathrm{VK}})$.
- **Journalists**:
  - $\mathrm{J}_{\mathrm{SK}}, \mathrm{J}_{\mathrm{VK}}$: Long-term Journalist signing key pair.
  - $\mathrm{JC}_{\mathrm{SK}}, \mathrm{JC}_{\mathrm{PK}}$: Long-term Journalist message-fetching key pair.
  - $\mathrm{JE}_{\mathrm{SK}}^{[0-n]}, \mathrm{JE}_{\mathrm{PK}}^{[0-n]}$: Ephemeral per-message key-encapsulation key pair.
  - $\mathrm{sig}^{\mathrm{NR}}(\mathrm{J}_{\mathrm{PK}}) = \mathrm{Sign}(\mathrm{NR}_{\mathrm{SK}}, \mathrm{J}_{\mathrm{PK}})$.
  - $\mathrm{sig}^{\mathrm{J}}(\mathrm{JC}_{\mathrm{PK}}) = \mathrm{Sign}(\mathrm{J}_{\mathrm{SK}}, \mathrm{JC}_{\mathrm{PK}})$.
  - $\mathrm{sig}^{\mathrm{J}}(\mathrm{JE}_{\mathrm{PK}}^{[0-n]}) = \mathrm{Sign}(\mathrm{J}_{\mathrm{SK}}, \mathrm{JE}_{\mathrm{PK}}^{[0-n]})$.
  - $\mathrm{sig}^{\mathrm{J}}(\mathrm{J}_{\mathrm{PK}}) = \mathrm{Sign}(\mathrm{J}_{\mathrm{SK}}, (\mathrm{JC}_{\mathrm{PK}}, \mathrm{JE}_{\mathrm{PK}}^{[0-n]}))$.
- **Sources**:
  - $\mathrm{PW}$: Secret passphrase.
  - $\mathrm{S}_{\mathrm{SK}}, \mathrm{S}_{\mathrm{PK}}$: Long-term Source key-encapsulation key pair.
  - $\mathrm{SC}_{\mathrm{SK}}, \mathrm{SC}_{\mathrm{PK}}$: Long-term Source message-fetching key pair.
- **Messages**:
  - $\mathrm{ME}_{\mathrm{SK}}, \mathrm{ME}_{\mathrm{PK}}$: Ephemeral per-message key-agreement key pair.
- **Server**:
  - $\mathrm{RE}_{\mathrm{SK}}, \mathrm{RE}_{\mathrm{PK}}$: Ephemeral Server, per-request message-fetching key pair.

## Key Verification: `KeyVer()`

The key verification algorithms checks that the key material received is valid and certified by the FPF through a chain of trust.

**Input:**

- $\mathrm{FPF}_{\mathrm{VK}}$ (hardcoded).
- $\mathrm{NR}_{\mathrm{VK}}, \mathrm{sig}^{\mathrm{FPF}}(\mathrm{NR}_{\mathrm{VK}})$. (The actual implementation should embed additional information such as the website and common name)
- A list of tuples indexed by $i$, each tuple in the form:
  $\mathrm{J}^i_{\mathrm{VK}}, \mathrm{sig}^{\mathrm{NR}}(\mathrm{J}^i_{\mathrm{VK}}), \mathrm{JC}^i_{\mathrm{PK}}, \mathrm{sig}^{\mathrm{J}}(\mathrm{JC}^i_{\mathrm{PK}}), \mathrm{JE}^{ik}_{\mathrm{PK}}, \mathrm{sig}^{\mathrm{J}}(\mathrm{JE}^{ik}_{\mathrm{PK}})$.
  (Note: $k$ is the index of non-used, non-expired Journalist ephemeral keys.)

**Output:** true/false.

1. Check $\mathrm{Verify}(\mathrm{FPF}_{\mathrm{VK}}, \mathrm{sig}^{\mathrm{FPF}}(\mathrm{NR}_{\mathrm{VK}})) = \mathrm{true}$.
2. For every Journalist (indexed by $i$) in Newsroom:
   - Check $\mathrm{Verify}(\mathrm{NR}_{\mathrm{VK}}, \mathrm{sig}^{\mathrm{NR}}(\mathrm{J}^i_{\mathrm{VK}})) = \mathrm{true}$.
   - Check $\mathrm{Verify}(\mathrm{J}^i_{\mathrm{VK}}, \mathrm{sig}^{\mathrm{J}}(\mathrm{JC}^i_{\mathrm{PK}})) = \mathrm{true}$.
   - Check $\mathrm{Verify}(\mathrm{J}^i_{\mathrm{VK}}, \mathrm{sig}^{\mathrm{J}}(\mathrm{JE}^{ik}_{\mathrm{PK}})) = \mathrm{true}$ for every ephemeral key $k$.
   - Check $\mathrm{Verify}(\mathrm{J}^i_{\mathrm{VK}}, \mathrm{sig}^{\mathrm{J}}(J^{ik}_{\mathrm{PK}}), (\mathrm{JC}^i_{\mathrm{PK}}, \mathrm{JE}^{ik}_{\mathrm{PK}})) = \mathrm{true}$ for every ephemeral key $k$.

# Message Post: `Write()`

This function allows a user (either the source or the journalist) to post a message in the system.

## Client

**Input:**

- Sender static/ephemeral keypair $[(\mathrm{SE}_{\mathrm{SK}}, \mathrm{SE}_{\mathrm{PK}}), (\mathrm{SC}_{\mathrm{SK}}, \mathrm{SC}_{\mathrm{PK}})]$,
- Fixed-length message $t$ composed of of `MAX_TOKENS` tokens
- Recipient keys $[\mathrm{DE}^i_{\mathrm{PK}}, \mathrm{DC}^i_{\mathrm{PK}}]_i$.

**Output:** 1 if success, 0 if failure.

1. Source pads the resulting text to a fixed size: mp = Pad(message, metadata, SPK, $\mathrm{SC}_{\mathrm{PK}}$, [0-m]s, $[0\text{-}m]_t$)*
2. Journalist pads the text to a fixed size: *mp = Pad(message, metadata)* .
3. Create a message $m = (\mathrm{SE}_{\mathrm{PK}}, \mathrm{SC}_{\mathrm{PK}}, t)$.
4. For every recipient $\mathrm{DE}^i_{\mathrm{PK}}, \mathrm{DC}^i_{\mathrm{PK}}$, in random order:
   - Sample $\mathrm{ME}^i_{\mathrm{SK}} = \mathrm{Gen}(), \mathrm{ME}^i_{\mathrm{PK}} = \mathrm{GetPub}(\mathrm{ME}^i_{\mathrm{SK}})$.
   - Let $ke^i, \mathrm{CE}^i = \mathrm{Encaps}(\mathrm{DE}^i_{\mathrm{PK}})$.
   - Let $\mathrm{KC}^i = \mathrm{DH}(\mathrm{ME}^i_{\mathrm{SK}}, \mathrm{DC}^i_{\mathrm{PK}})$.
   - Discard $\mathrm{ME}^i_{\mathrm{SK}}$.

- Encrypt $\mathrm{ctx}^i = \mathrm{Enc}(ke^i, m)$.
- Send $(\mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \mathrm{KC}^i)$.

### Server

**Input**:

- Current internal store
- A list of tuples $(\mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \mathrm{KC}^i)$

**Output**

- a new store, and identifiers for the created slot

1. For every received tuple $(\mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \mathrm{KC}^i)$:
   - Sample 32 bits $\mathrm{mid}^i = \mathrm{Gen}()$ and internally store $(\mathrm{mid}^i, \mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \mathrm{KC}^i)$.
   - Yield $\mathrm{mid}^i$

## Message Poll: `Poll()`

Allows a client (source/journalist) to check if a new message has been produced for them.

### Client

**Input:**

- Personal static key $\mathrm{SC}_{\mathrm{SK}}$.
- A list $\mathrm{sctxmids}$ of elements $(\mathrm{ME}^i, \mathrm{ctxmid}^i)$ indexed by $i$.

**Output:** A list $\mathrm{mids}$ of identifiers.

1. Initialize a list $\mathrm{mids} = []$.
2. For every $(\mathrm{ME}^i, \mathrm{ctxmid}^i)$:
   - Let $\mathrm{KC}^i = \mathrm{DH}(\mathrm{SC}_{\mathrm{SK}}, \mathrm{ME}^i)$.
   - Let $\mathrm{mid}^i = \mathrm{Dec}(\mathrm{KC}^i, \mathrm{ctxmid}^i)$.
   - If $\mathrm{mid}^i$ is not NULL (decryption successful), append it to $\mathrm{mids}$.
3. Return $\mathrm{mids}$.

### Server

**Input:** The list of message identifiers $\mathrm{smids}$.

**Output:** A list $\mathrm{sctxmids}$ of encrypted messages.

1. Set $\mathrm{sctxmids}$ to be the empty list.
2. For every entry $(\mathrm{mid}^i, \mathrm{c}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{KC}^i)$ in $\mathrm{smids}$:
   - Generate a new keypair $\mathrm{RE}^i_{\mathrm{SK}} = \mathrm{Gen}()$.
   - Let $\mathrm{KC}^i_{\mathrm{S}} = \mathrm{DH}(\mathrm{RE}^i_{\mathrm{SK}}, \mathrm{KC}^i)$.
   - Let $\mathrm{ME}^i = \mathrm{DH}(\mathrm{RE}^i_{\mathrm{SK}}, \mathrm{ME}^i_{\mathrm{PK}})$.
   - Discard $\mathrm{RE}^i_{\mathrm{SK}}$.
   - Encrypt $\mathrm{ctxmid}^i = \mathrm{Enc}(\mathrm{KC}^i_{\mathrm{S}}, \mathrm{mid}^i)$.
   - Append $(\mathrm{KC}^i_{\mathrm{S}}, \mathrm{ctxmid}^i)$ to $\mathrm{sctxmids}$.
3. For each $i$ in $[1, .., \mathrm{MAX\_MESSAGES} - |\mathrm{sctxmids}|]$:
   - Sample a new keypair $R_{\mathrm{SK}} = \mathrm{Gen}()$ and $R_{\mathrm{PK}} = \mathrm{GetPub}(R)$.
   - Sample $|\mathrm{sctxmid}^0|$ (length of all ciphertexts is fixed) random bytes called $\mathrm{ctxr}$.
   - Append $(R_{\mathrm{PK}}, \mathrm{ctxr})$ to $\mathrm{sctxmids}$.
4. Shuffle $\mathrm{sctxmids}$
5. Sort $\mathrm{sctxmids}$ lexicographically.
6. Return $\mathrm{sctxmids}$.

# Message Read: `Read()`

## Client

**Input**:

- KEM secret key $\mathrm{S}_{\mathrm{SK}}$.
- Message id $\mathrm{mid}$

**Output:** Message $m$ or NULL.

1. Send $\mathrm{mid}$ to the server and parse the response as $(c, \mathrm{CE})$
2. Let $ke = \mathrm{Decaps}(\mathrm{S}_{\mathrm{SK}}, CE)$.
3. Decrypt $m = \mathrm{Dec}(ke, c)$.
4. Return $m$. If decryption fails, return NULL.

## Server

**Input**:

- Store with elements of the form $(\mathrm{mid}^i, \mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \mathrm{KC}^i)$

**Output**: empty.

1. Receive $\mathrm{mid}$ from the user

2. If there is an $i$ such that $\mathrm{mid}^i = \mathrm{mid}$
    - Respond with $\mathrm{ctx}^i, \mathrm{CE}^i$
3. Otherwise, respond the empty string.

# High-level protocol

## Key Setup

Use the key generation procedure described above to generate the key

## Material upload

1. For all *i* indices over the *c* chunks:
    - Generate a random encryption key ${}^{i}s = KeyGen()$
    - Encrypt ${}^{i}u$ using *s*: ${}^{i}ctx = Enc({}^{i}s, {}^{i}u)$
    - Upload ${}^{i}ctx$ to the server obtaining a unique token ${}^{i}t$
2. Return $[{}^{i}s, {}^{i}t]_i$

1. Generate a random encryption key $s = KeyGen()$
2. Encrypt *u* using *s*: $ctx = Enc(s, u)$
3. For all *i* indices over the *c* chunks:
    - Upload *ctx* to the server obtaining a unique token ${}^{i}t$
4. Return $(s, [{}^{i}t]_i)$

## Submission

1. Download journalist keys (including ephemeral key encapsulation keys $JE_{PK}$)
2. Run the key verification procedure using the verification key of FPF hardcoded and the key material downloaded.
3. Run the (deterministic) key generation function for the whistleblower.
4. Run the material upload function. This produces a key *s* and tokens ${}^{i}t$ for $i = 1..|t|$.
5. For $i = |t| .. \texttt{MAX\_TOKENS} - 1$: set ${}^{i}t = 0$
6. Run the message posting algorithm using
    - Sender keys: $S_{SK}$, $SC_{SK}$ (nota bene: keys are re-used. No ephemeral keys)
    - Message blocks: ${}^{0}t = s, [{}^{i}t]$
    - Recipients keys: $JE_{PK}$

## Poll

**Source.**

1. Derive $SC_{SK}$ = *Gen(KDF(fetching_salt + PW))*
2. Download the `MAX_MESSAGES` *sctxmids* = $[^{i}ME_S, ^{i}enc\_mid]_i$ from *Server*
3. Run the message polling procedure with inputs:
   - $SC_{SK}$ the personal static key
   - *sctxmids* the list of encrypted server message ids

   Output whatever the procedure returns.

**Journalist**.

1. Download *sctxmids* = $[^{i}ME_S, ^{i}enc\_mid]_i$ from *Server*
2. Run the message polling procedure with inputs:
   - $JC_{SK}$ the personal static key
   - *sctxmids* the list of encrypted server message ids
3. Output whatever the procedure returns.

## Read

**Client** (Journalist and Source).
The client has input *mid*, and the key material.

1. Send *mid* to Server, obtain $(ctx, ME_{PK}, CE)$
2. For every unused key encapsulation secret $^{i}KE$ ($[^{i}KE] = [^{i}JE_{SK}]_i$ for the journalist, and $[^{i}KE] = [SE_{SK}]$ for the source):
   - Run the message read function with:
     - secret key $^{i}KE_{SK}$
     - message identifier *mid*

   If a message $m = (S_{PK}, SC_{PK}, s , t)$ is decrypted successfully, return it immediately.
3. Initialize empty file *f*. For every token $^{i}t$
   - Fetch from *Server* $^{i}f$ using token $^{i}t$ and adds it to *f*.
   - Decrypt $^{i}f$ using *s*
4. Decrypt *u = Dec(s, f)*
5. Output *u*

**Source**.

1. Download from *Server* using message-id *mid* the tuple (*mid*, *c*, $ME_{PK}$)
2. Derive $S_{SK}$ = *Gen(KDF(encryption_salt + PW))*
3. Run the message reading procedure using as input:
    - *mid* the message id requested
    - secret key $S_{SK}$

### Reply

**Source**.

Replies work the exact same way as a first submission, except the source is already known to the *Journalist*.

**Journalist**.
The journalist has a plaintext *mp*, and the recipient public keys $S_{PK}$ and $SC_{PK}$

1. Run the message upload procedure for *mp* obtaining a secret key *s* and a set of blocks ${}^{i}t$. Sample a new ephemeral keypair ${}^{k}JE_{SK}$ ${}^{k}JE_{PK}$.
2. Run the message posting procedure with:
    - Sender static/ephemeral keypair *[(${}^{k}JE_{SK}$, ${}^{k}JE_{PK}$), ($JC_{SK}$, $JC_{PK}$)]*,
    - Message blocks *t = [${}^{i}t$]* for all *i* indices from 0 to `MAX_TOKENS` .
    - Recipient keys *[${}^{i}S_{PK}$, ${}^{i}SC_{PK}$]$_i$*.

# Findings

We summarize below two attacks that emerged during the analysis the protocol, along with some possible avenues for countermeasures.

The use of re-randomizing Diffe-Hellman tuples as a means to ensure anonymity of the recipient is original work. Providing a re-randomizable KEM from post-quantum assumptions seems to be an open problem.

## Message Forgery

In the message posting process, $\mathrm{CE}$ and $\mathrm{ctx}$ are not authenticated, and the message tuple $(\mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \mathrm{KC}^i)$ can be tampered with by a third party sending a message to the same recipient, even though the recipient remains hidden.

## Description

We present the attack using the notation of the protocol provided in the initial submission.
In the submission version, a message deposited by the source for the journalist identified with keys $\mathrm{JE}^i_{\mathrm{PK}}, \mathrm{JC}^i_{\mathrm{PK}}$ is of the form $(\mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{KC}^i)$ where $\mathrm{ctx}^i$ is AES-encrypted with key $k = \mathrm{DH}(\mathrm{ME}^i_{\mathrm{SK}}, \mathrm{JE}^i_{\mathrm{PK}})$ and $\mathrm{KC}^i_{\mathrm{PK}} = \mathrm{DH}(\mathrm{ME}^i_{\mathrm{SK}}, \mathrm{JC}^i_{\mathrm{PK}})$.
First, the source runs the write function to submit a message formatted as mentioned. Then, the malicious server stores the message as per protocol description.
The journalist runs the message polling function.
The malicious server internally runs the polling protocol (acting honestly) and sends the encrypted message id $\mathrm{mid}$ corresponding to the created record. The message is encrypted with key: $\mathrm{KC}^i_{\mathrm{S}} = \mathrm{DH}(\mathrm{ME}^i_{\mathrm{SK}}, \mathrm{DH}(\mathrm{RE}^i_{\mathrm{SK}}, \mathrm{JC}^i_{\mathrm{PK}}))$ and is sent on the wire along with $\mathrm{DH}(\mathrm{RE}^i_{\mathrm{SK}}, \mathrm{ME}^i_{\mathrm{PK}})$.
The journalist correctly decrypts the message id $\mathrm{mid}$ and invokes the message reading function.
The malicious server at this point sends instead a forged message: it guesses one of the recently-used keys and sends $(\mathrm{ctx}^i_A, \mathrm{ME}^i_{\mathrm{A}})$ using a different ciphertext and ephemeral key. The ciphertext $\mathrm{ctx}^i_A$ is produced encrypting an arbitrary message with an ephemeral journalist key. The encrypted message is of the form:
$(\mathrm{SE}_{\mathrm{A}}, \mathrm{SC}_A, t)$
where the sender ephemeral and identity keys are of another journalist.

Also the current protocol is affected by the attack. Informally, the server logs ephemeral key encapsulation public keys sent to clients. Upon receiving a message posting request in the form $(\mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \mathrm{KC}^i)$, it replaces them with $(\mathrm{ctx}^i_A, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i_A, \mathrm{KC}^i)$, where:
- $\mathrm{k}^i, \mathrm{CE}^i_A = \mathrm{Encaps}(\mathrm{DE}^i_{\mathrm{PK}})$, and $\mathrm{DE}^i_{\mathrm{PK}}$ is guessed from previously sent keys to users.
- $\mathrm{ctx}^i_A = \mathrm{Enc}(\mathrm{k}^i, \mathrm{forged\_message})$.

## Mitigations

The classic way to prevent this attack would be to sign the message being sent using the sender key. That is, the tuple is still $(\mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \sigma^i, \mathrm{KC}^i)$
but $\mathrm{ctx}^i$ is an encryption of
$m = (\mathrm{SC}_{\mathrm{PK}}, \mathrm{SE}_{\mathrm{PK}}, t, \sigma\ )$
where
$\sigma = \mathrm{Sign}(\mathrm{SC}_{\mathrm{SK}}, (\mathrm{CE}^i, \mathrm{SC}_{\mathrm{PK}}, \mathrm{SE}_{\mathrm{PK}}, t))$

Another possible solution may involve *implicit authentication*, similar to x3DH of Signal: the *key* used for encrypting the ciphertext $\mathrm{ctx}^i$ is computed as
$k = \mathrm{H}(\mathrm{DH}(\mathrm{SC}_{\mathrm{SK}}, \mathrm{DE}_{\mathrm{PK}}),\ \mathrm{DH}(\mathrm{SE}_{\mathrm{SK}}, \mathrm{DC}_{\mathrm{PK}}), \mathrm{DH}(SE_{\mathrm{SK}}, \mathrm{DE}_{\mathrm{PK}}))$.
The computational Diffie-Hellman problem makes it hard for an adversary to provide a valid key

without knowing the discrete logarithm of the long-term sender key. We refer the reader towards the Signal documentation and this formal analysis of the Signal protocol for more information.

# Replay attack of source messages

The recipient key of a source is static, and (partially also due to the above problem) messages where the source is the recipient are susceptible to replay attacks.

## Description

Consider a network adversary that obtains (observing the network) a message record $(\mathrm{ctx}^i, \mathrm{ME}^i_{\mathrm{PK}}, \mathrm{CE}^i, \mathrm{KC}^i)$ with the source as recipient.
The pair $(\mathrm{ctx}^i, \mathrm{CE}^i)$ can be replayed and sent during the message reading function of another message.

## Mitigations

A simple fix to this problem may be to add timing information as the authenticated data within the message, and provide this information to the final user to check. This solution is sub-optimal and depends on the final user being attentive at the time displayed. A more modern solution would be to use ephemeral keys also on the side of the source and (just as it happens for the journalist) rotate them over time. This can happen in two ways:

- have an additional, per-connection password that is used to generate the ephemeral state.
- adopting password-authenticated key exchange mechanisms and store the key material on the server (preventing rollbacks).

The use-case here is that, while preprocessing attacks / password guessing here will completely compromise the long-term keys of the user, in the case above an adversary would *also* need to break the key material stored in the server or the ephemeral password.

## Asymmetric PAKE

**Context.** Allow a user to log in only if their password is correct. Moreover, we aim to avoid storing the password, thus ensuring a much stronger security model. The State-of-Art scheme is currently an RFC draft OPAQUE.
The protocol offers a crucial security property: no dictionary attack possible on this vast amount of data, and no information about the password being stored.

This protocol must rely on an additional cryptographic primitive called VOPRF (Verifiable Oblivious Pseudorandom Function).

VOPRFs are currently described in a RFC draft.

**Protocol Overview**. During the registration phase, the user stores a record $(k_s, \mathrm{S_{SK}}, \mathrm{S_{PK}}, \mathrm{U_{PK}}, c)$, where:

- $k_s$ is an ephemeral VOPRF key.
- $\mathrm{S_{SK}}$ is the ephemeral private key of the server, with $\mathrm{S_{PK}}$ being the corresponding public key.
- $\mathrm{U_{PK}}$ is the user's public key.
- $c = \mathsf{Enc}(F_{k_s}(\mathrm{pw}), (\mathrm{U_{SK}}, \mathrm{U_{PK}}, \mathrm{S_{PK}}))$ represents the authenticated encryption of the record. Here, $F_{k_s}(\mathrm{pw})$, which is a VOPRF evaluation of the password $\mathrm{pw}$ with key $k_s$, serves as the encryption key. Although $(\mathrm{U_{PK}}, \mathrm{S_{PK}})$ can be public, they need to be authenticated within this context.

To log in, the user identifies a specific record. The server assists in proving a VOPRF evaluation to the user, who then uses this evaluation to decrypt the record. The decrypted record contains information necessary for initiating a key exchange.

**Possible integration.** In order to provide a valid security notion that avoids rewinding of the state, the metadata of the authenticated encryption must also compose of a timestamp and a counter denoting the last access. This information can be checked by the user upon logging in.

The log-in phase is very similar to the file download process, except for the VOPRF evaluation.
It might be worth considering adding a VOPRF evaluation also for the document retrieval part to make the login flow indistinguishable from the downloading of a small document from the server.

# Informative comments

## Denial of Service Attacks

Besides obvious denial of service attacks that can be carried by the server, dropping or tampering with protocol messages, constants such as `MAX_MESSAGES` and `MAX_KEYS` are delicate and should be set carefully. While this was partially addressed in the initial writeup, we now provide additional concrete countermeasures and recommended bounds; here we also provide some concrete countermeasures and bounds to respect.

- `MAX_MESSAGES` determines a trade-off between public-key operations to be performed on the side of the client (during polling) and the number of submissions that can be accepted into the server. A low value will make the server easily susceptible to DoS (an adversary flooding the server with submissions), a high value will impact performance of the client when polling.
  A number of countermeasures here can be used. A simple solution here would be to increase the cost of the message writing functions using *proofs of work* / *grinding* techniques to make it

expensive to spam the server.
You might consider using verifiable oblivious random functions as described in the RFC draft for the Privacy Pass protocol rather than IP-based logging and blocking. A general rule of thumb is to consider 1e-5 seconds per Diffie-Hellman operation on the side of the client.

- `MAX_KEYS` determines the trade-off between public-key operations to be performed on the side of the journalist (when reading) and the number of submissions that a journalist can receive. In the current implementation (where each submission is sent to every journalist), it must be checked that `MAX_KEYS > MAX_MESSAGES` for correctness.

In addition, *exhausting the keys registered may leak information about the number of keys present in the system.*

### Typos

Some minor typos have been corrected and can be cherry-picked from mmaker/securedrop-poc.

### Canaries

A section dedicated to how keys should be handled will help preventing errors in deployments. For example, the FPF could implement canaries to alert users of potential compromises; Newsrooms should adhere to a minimal set of rules of key rotation and identity validation for journalists, and accountability that they still possess the key from time to time.

## Other informative comments

This writeup already integrated minor modifications to the initial protocol for a stronger system and a more robust security proof. A reasoning for these choices is provided below.

### Journalist/whistleblower anonymity

The previous protocol uses a different submission length algorithm for the whistleblower and the source.
In order to extend the anonymity set to include both journalists and sources, the protocols have been merged into one.

### Use of standard key encapsulation (KEM) primitives

One of the targets for the review was to also study the resilience of data at rest. The formalization under a generic KEM will ease post-quantum security migration.

### Message trimming attacks

The current implementation of Securedrop-NG AES-encrypts multiple files with multiple keys, and each file is encrypted in chunks. On the receiving end *length of the encrypted file must be checked* in order to prevent the server dropping chunks of the message. Empty files must also carefully handled as well. A simpler and more space-efficient alternative is to use one single key for encrypting the entire message (as presented in the protocol). Message-trimming attacks will be mitigated by the guarantees of authenticated encryption.

# Formalization of the System

## Correctness

We were able to check formally that the protocol was correct in the following aspects:

- *Key material*: All honestly generated key materials should pass the key verification protocol.
- *Message posting*: All messages posted for a recipient key should be successfully posted and read.
- *Message polling*: All posted messages can be retrieved successfully via polling.

The correctness of key material is assured by the signature scheme, message posting follows from the KEM and encryption scheme, message polling from the correctness of the encryption scheme and remarking that:

$$\mathrm{KC}^i_{\mathrm{S}} = \mathrm{DH}(\mathrm{ME}^i_{\mathrm{SK}}, \mathrm{DH}(\mathrm{RE}^i_{\mathrm{SK}}, \mathrm{JC}^i_{\mathrm{PK}})) = \mathrm{DH}(\mathrm{JC}^i_{\mathrm{SK}}, \mathrm{DH}(RE^i_{\mathrm{SK}}, \mathrm{ME}^i_{\mathrm{PK}})) = \mathrm{DH}(\mathrm{JC}^i_{\mathrm{SK}}, \mathrm{ME}^i_{\mathrm{S}})$$

## Anonymity

We were able to formally study the following properties of the system:

- *Message hiding*: An adversary asks a honest user to upload two different messages of the same length. The adversary canot tell which one was uploaded.
- *Polling anonymity*: The adversary cannot distinguish between two different polling queries made over different server state.
- *Sender/Recipient anonymity*: An adversary server selects two (honestly-generated) sender public keys $[\mathrm{SC}^b_{\mathrm{PK}}, \mathrm{SE}^b_{\mathrm{PK}}]_{b=0,1}$, two (honestly-generated) recipient public keys $[\mathrm{DC}^b_{\mathrm{PK}}, \mathrm{DE}^b_{\mathrm{PK}}]_{b=0,1}$, and two messages $t_0, t_1$. With those it can query challenge oracle to invoke the message write procedure on either of the two. send to one of two recipient.
  The dentities are concealed from the server.

Message hiding is straightforward from the IND-CPA properties of the encryption scheme.
Polling anonymity is *computational* and relies on the decisional Diffie-Hellman problem. It is possible, via the self-reducibility property of the Decisional Diffie-Hellman problem, to make all polling responses indistinguishable from random group elements. The advantage here is equal to DDH in the considered group.
Sender/Recipient anonymity: follows from IND-CPA encryption of the authenticated encryption and the key encapsulation mechanism.

## Authenticity

We study the existential unforgeability under chosen message attacks. An adversary asks the oracle to generate multiple honest users indexed in $i$. The adversary can instruct via an oracle to write new messages in the database from honest users. The plaintext messages are stored in a list of queries $Q_i$ associated to each user, while the server holds the encrypted messages in its internal state. The adversary can insert new messages in the server or replace previous records identified by some message id $\mathrm{mid}$. At the end, the challenger checks if the message reading procedure is successful: for all honest users, check if the message reading is successful and the plaintext is not in $Q_i$ and yet the user $i$ is declared as sender.

This has led to an attack discussed in the section above.

## Secrecy

Indistinguishability under chosen ciphertext attacks is provided by the authenticated encryption scheme and the key encapsulation function. Claims about forward secrecy can be made only for the journalists.
Once the main issue is solved, it should be possible to move Cohn-Gordon et al.'s analysis also here and already prove the forward-secrecy of a generalization of the Signal protocol. Their abstraction has a unified security notion of confidentiality + authenticity, and therefore the formal model will have to depend on the mitigation chosen.