Designing the next generation of SecureDrop

Notes on a niche cypherpunk problem

8 May 2025 — "The Legacy of the Cypherpunks"

Cory Myers <cory@freedom.press>

Good afternoon. Thank you all for having me, and thanks so much to Nadia for inviting me. My name is Cory Myers, and for just about four years now I've had the privilege of working on the SecureDrop whistleblowing system at the Freedom of the Press Foundation. I should say something about that timeline. Like a lot of the other ideas and technologies on the syllabus for this class, SecureDrop has a lineage that begins in or even belongs to an earlier era of computing, an earlier era of the Internet. SecureDrop has a lot of concrete history that predates my joining the project. And the new work I'll also talk about today is very much joint work: with my colleagues at FPF, and with our collaborators at ETH, the Swiss Federal Technical Institute, in Zürich. But I'm speaking in a personal capacity; my comments here today are my own. So this is the usual disclaimer: anything interesting I say is because I have smart colleagues; anything I get wrong is my mistake alone.

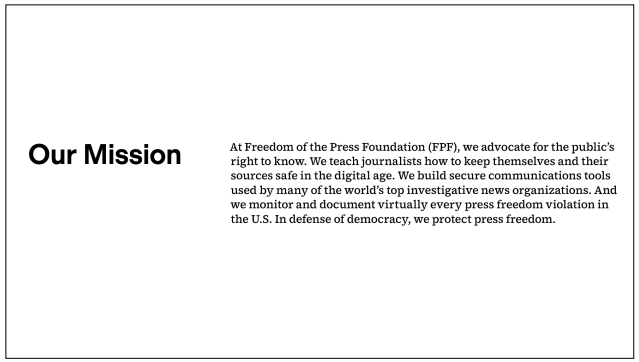
---

1. FPF in 3 minutes
2. Edward Snowden and the first-contact problem
3. How SecureDrop works today
4. Signal has entered the chat: what we can and can't learn from general end-to-end–encrypted messaging
5. The SecureDrop Protocol: designing for the first-contact problem today
6. Some lessons we've learned (and are still learning)

Here's a rough outline of what we'll cover in the next hour. I want to leave plenty of time for your questions and discussion, so I'll try to pause at the end of each section. But also feel free to interrupt me at any time with questions or clarifications.

FPF in 3 minutes

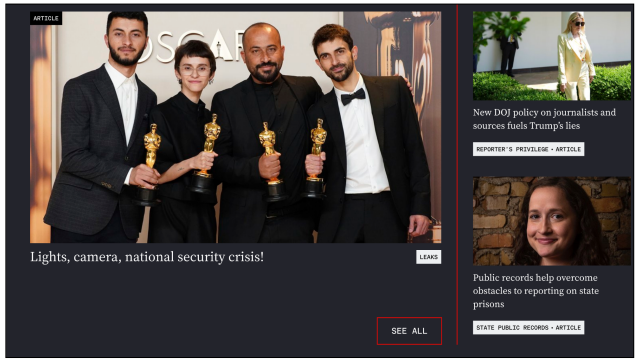
I want to start by telling you briefly about my organization, the work we do, and how we do it. Conway's law tells us that the structure of an organization influences the structure of the systems it produces. That might not be true about SecureDrop's structure, but it's definitely true about its values, as you'll see.



https://media.freedom.press/media/documents/FPF_2024ImpactReport.pdf

The Freedom of the Press Foundation was founded in 2013 by alumni of the Electronic Frontier Foundation. It began as a fiscal-sponsorship organization to circumvent what amounted to a blockade by payment processors who refused to process donations for WikiLeaks.

Since then FPF has evolved and grown significantly, and we now take a multi-prong approach to advocating for and defending the freedom of the press.

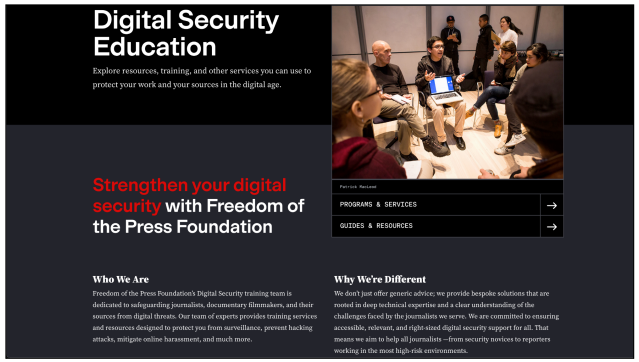


https://freedom.press/issues/

Our advocacy team lobbies for policies that protect journalists' rights to report and draws attention to cases where laws or the courts interfere with that right.

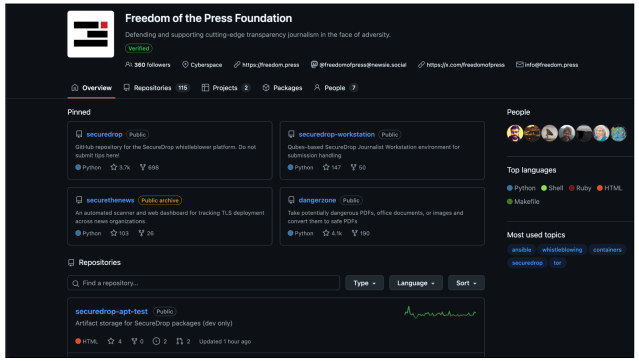
https://freedom.press/tracker/

We have our own reporting team that publishes the US Press Freedom Tracker, which is the canonical database of incidents in which court decisions or police actions obstruct or even harm journalists doing their job.



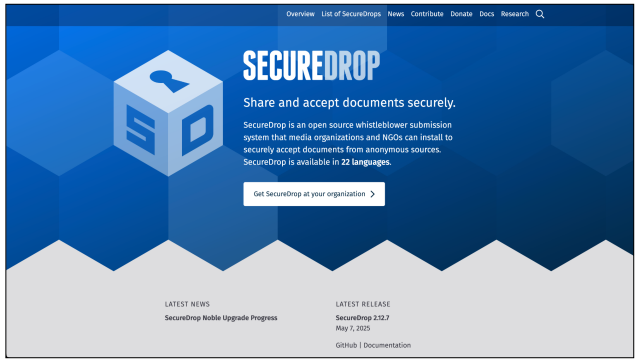
https://freedom.press/digisec/

Our Digital Security Training team teaches thousands of journalists every year how to keep themselves safe online and their sources safe in course of their reporting.



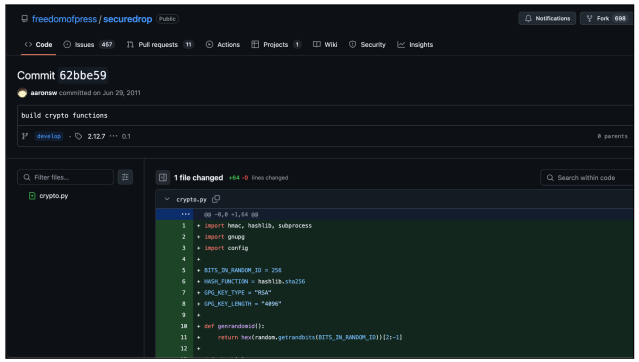
https://github.com/freedomofpress/

And we've maintained a variety of software projects over the years to help protect journalists and others. All of our software is open-source.



https://securedrop.org/

Of these projects, the one that's longest running and in the widest use is SecureDrop, which you've read about some and which I'll be talking about for rest of this talk.



https://github.com/freedomofpress/securedrop/commit/62bbe590afd77a6af2dcaed46c93da6e0cf40951

I had an error my abstract: Aaron Swartz made the first Git commit for what was then called the "DeadDrop" project in 2011.

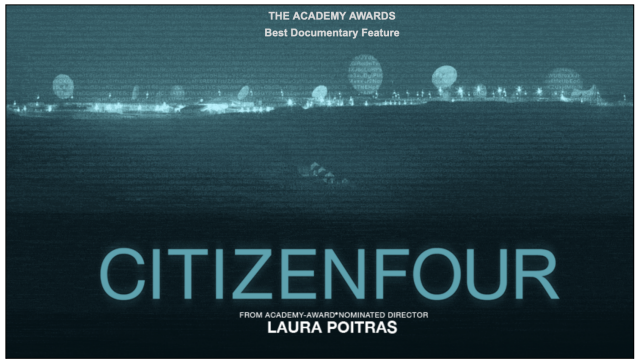
SecureDrop's first official release wasn't until two years later, after Aaron's death in 2013.

One could give a whole other talk about Aaron's life and work, especially the transparency work that led to his prosecution and suicide. But that's another talk for another day. Suffice to say that SecureDrop grew out of Aaron's vision for it, and FPF was honored to become the project's steward after his death.

Questions? Comments?

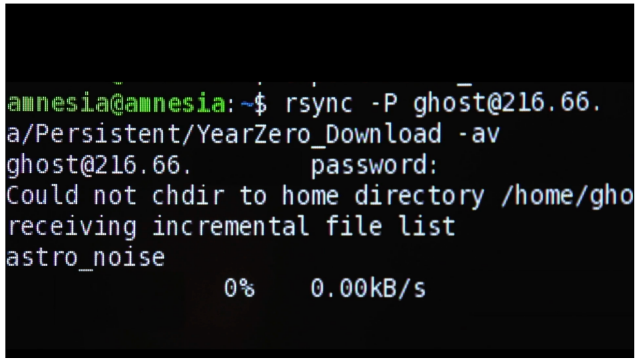
For today, though, let's turn to another transparency figure of 2013....

Edward Snowden
and the first-contact problem



https://citizenfourfilm.com

Show of hands: How many of you have seen "Citizenfour"? [...]



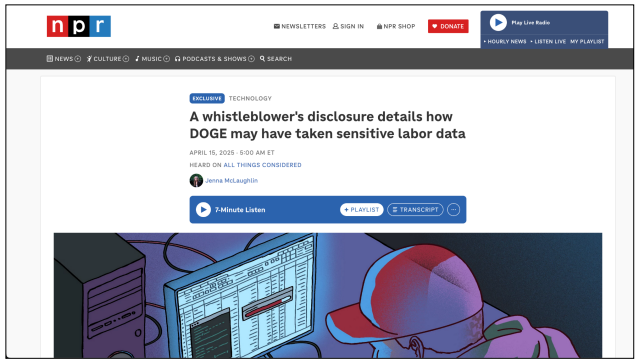
https://citizenfourfilm.com

The early scenes show the lengths Edward Snowden went to in 2013 to make contact with journalist Glenn Greenwald, then at “The Guardian”, and filmmaker Laura Poitras.

The rest of the film shows—in vivid, dramatic detail—what happened afterwards in terms of what we call *operational* security: the clandestine meetings in Hong Kong, paranoia about bugged hotel rooms.

https://security.stackexchange.com/questions/82362/in-citizenfour-what-was-edward-snowden-mitigating-with-a-head-blanket

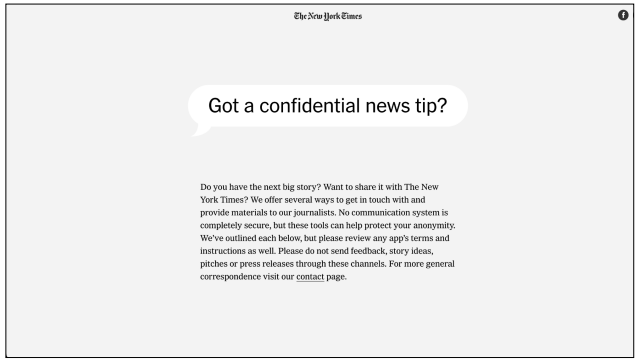
There's a great scene with Snowden with a blanket over the head when entering a password. But those first few minutes: let's break down what's happening there and why.



https://www.npr.org/2025/04/15/nx-s1-5355896/doge-nlrb-elon-musk-spacex-security

Let's say you're a government contractor—or a current federal worker—and you feel you have information the public needs to know.

If you're a political insider, you're probably in touch with lots of journalists anyway, so you just meet one of them for coffee or call them on the phone and relay whatever you have. In your situation, there's nothing especially noteworthy, never mind incriminating about that meeting or that phone call.



https://www.nytimes.com/tips

What if you're not? You probably go to a newspaper you trust—let's say "The New York Times" and look for their "tips" or "contact us" page.

Maybe you e-mail them.

https://x.com/zeynep/status/1018094555751682049

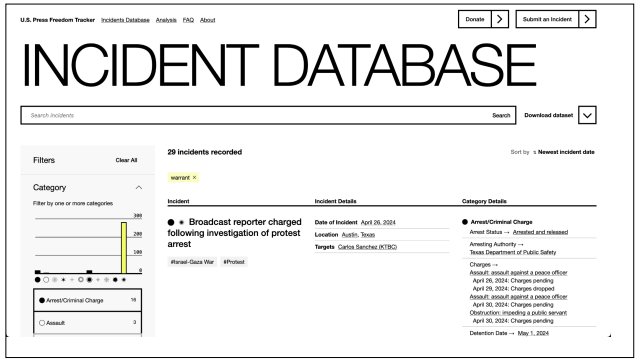
Or, since it's 2025, not 2013, you have Signal. You message a reporter on Signal and relay whatever you have. Okay, how do you get their contact information?

Let's say that what you have is a really big deal: a cache of documents about a mass surveillance program, for example, or the draft of a court decision that's about to overturn established precedent.

Even if your reporter and their editor are honorable, a lot of things could still go wrong from here:

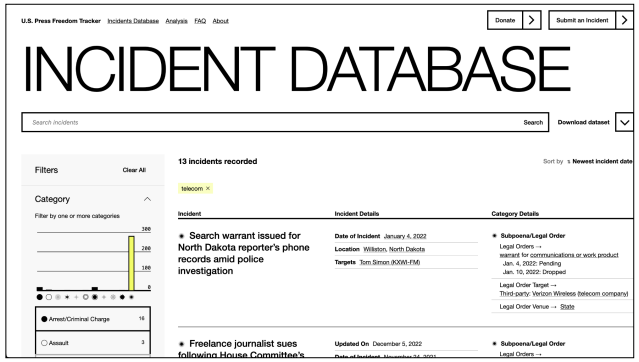
https://pressfreedomtracker.us/all-incidents/?legal_order_information_requested=TESTIMONY_ABOUT_SOURCE&categories=Subpoena%2FLegal+Order

The reporter could be subpoenaed for testimony.



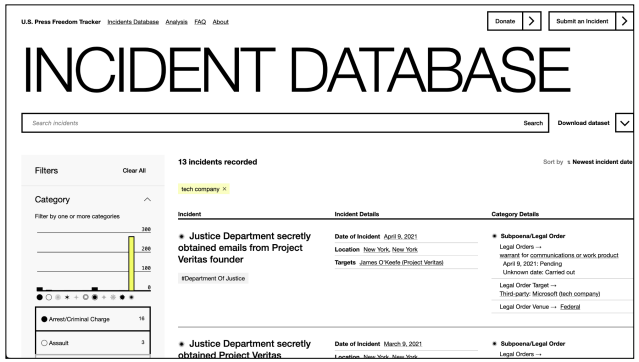
https://pressfreedomtracker.us/all-incidents/?legal_order_type=WARRANT&categories=Subpoena%2FLegal+Order

They could be searched, with a warrant.



https://pressfreedomtracker.us/all-incidents/?third_party_business=TELECOM&categories=Subpoena%2FLegal+Order

The authorities could go to their phone company for their calling and texting logs.

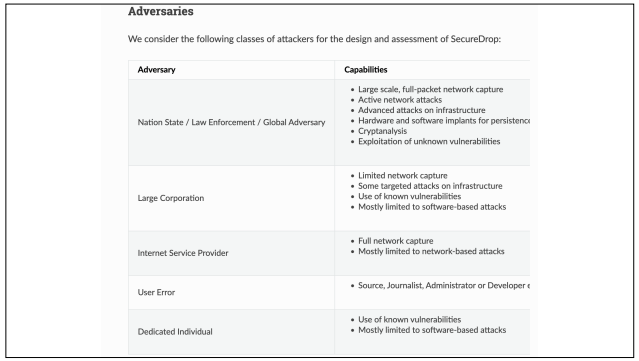


https://pressfreedomtracker.us/all-incidents/?third_party_business=TECH_COMPANY&categories=Subpoena%2FLegal+Order

Same thing at their e-mail provider.

In which case you'll quickly find yourself in trouble.

In other words, you might trust the journalist, and they might protect you to the best of their ability. But what if they're compromised or targeted themselves?



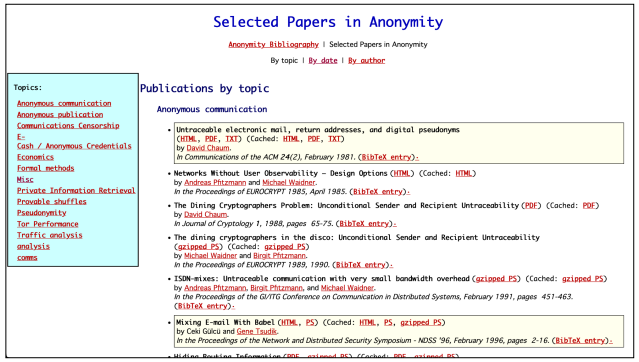
Adversaries

We consider the following classes of attackers for the design and assessment of SecureDrop:

Adversary	Capabilities
Nation State / Law Enforcement / Global Adversary	• Large scale, full-packet network capture • Active network attacks • Advanced attacks on infrastructure • Hardware and software implants for persistenc • Cryptanalysis • Exploitation of unknown vulnerabilities
Large Corporation	• Limited network capture • Some targeted attacks on infrastructure • Use of known vulnerabilities • Mostly limited to software-based attacks
Internet Service Provider	• Full network capture • Mostly limited to network-based attacks
User Error	• Source, Journalist, Administrator or Developer
Dedicated Individual	• Use of known vulnerabilities • Mostly limited to software-based attacks

https://docs.securedrop.org/en/stable/threat_model/threat_model.html#adversaries

So this is the first problem SecureDrop tries to solve: how to minimize the identifying information, both technical metadata and personal information, that a journalist or news organization ever sees about a source. If they don't know who you are, they can't reveal who you are, even with a warrant! And if the system doesn't know who you are or where you connected from, then it can't be searched to give you up either. So here it is concretely: FPF tries to solve press-freedom problems through advocacy and the law, in this case the prohibition against unlawful search and seizure under the Fourth Amendment. When that fails, or when those protections can't be relied upon, we try to solve or at least mitigate them with technology.

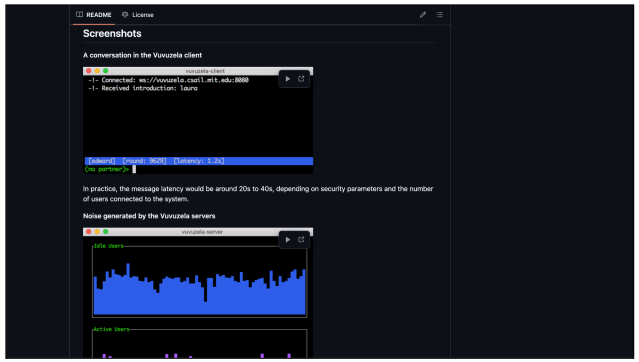


https://www.freehaven.net/anonbib/topic.html

Now, anonymous communication is not a new goal. There are a *lot* of systems that strive to provide anonymous communication, for different values of "anonymous".

But they all have limitations that make them difficult to use in this situation.

Some of them only work between two people. That would mean you couldn't send a message to multiple journalists, and you'd have to know already who you wanted to contact. What if you only know what newspaper, magazine, or broadcaster you want to leak to? Who do you contact? And why would they take you seriously?



https://github.com/vuvuzela/vuvuzela

Some of them only work if you and the other person are online at the same time, like an ephemeral real-time chat room. You can achieve some great security properties this way: for example, you can have all of your online users generate noisy fake traffic that covers one another's real traffic, so that it's hard to tell who's speaking when and to whom. But how does that work here? How do you coordinate this kind of online rendezvous if the source is, by definition, unknown to the journalist until they make contact?

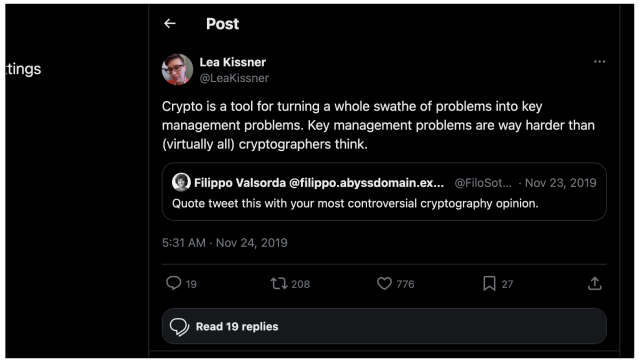
PKI for dialing. Vuvuzela's dialing protocol requires a public key infrastructure in two situations. First, to start a conversation, a user must know the public key of the other party. Looking up this key on-demand over the Internet via some key server would disclose who the user is dialing, so Vuvuzela clients should store public keys for contacts ahead of time. Second, when receiving a call via the dialing protocol, the recipient needs to identify who is calling, based on the caller's public key. Here, the caller can supply a certificate along with the invitation, if the recipient does not already know the caller; this avoids the need for the recipient to contact a key server.

https://dl.acm.org/doi/pdf/10.1145/2815400.2815417

And most of these systems require that each party has some way to receive the other's keys ahead of time. Same problem: how do you that without *coordinating* ahead of time?

So now we have a clearer view of the two problems we're trying to solve:

1. The first problem is anonymity. If you contact a journalist and send information or documents, you don't want the journalist to know who you are, so that they're not even *able* to reveal your identity to anyone else, even under investigation or coercion.



https://x.com/LeaKissner/status/1198595109756887040

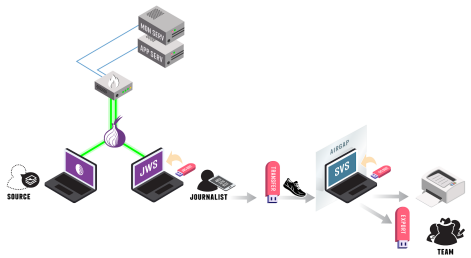
2. The second problem is key distribution. This is a *hard* problem in general.

But the good news is that SecureDrop doesn't need to solve it generally. In fact, we only need to solve it in one direction, since we only need a way for *sources* to contact *journalists*. So there's an asymmetry here. Sources need to be able contact journalists anonymously; journalists don't need to stay anonymous. That means journalists, and their keys, can be easy to find. And it doesn't matter if sources' keys are hard to find: they can just *send* them when they first contact a journalist!

Keep this asymmetry in mind—it will be relevant later.

Questions? Comments?

How SecureDrop works today



https://docs.securedrop.org/en/stable/what_is_securedrop.html

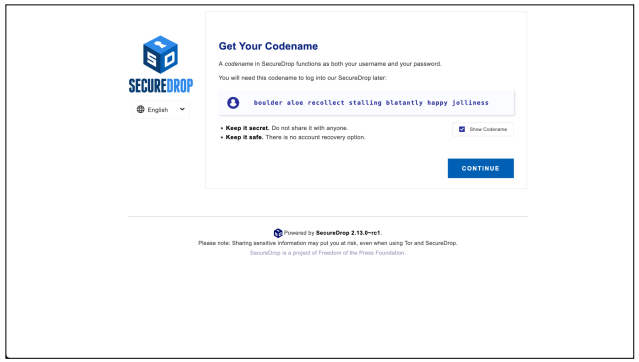
So here's the design SecureDrop has had more or less since 2013:

Newsrooms run their own SecureDrops, ideally in an office they control, so that they'd have to be served a warrant directly for a search. This is a transparency measure—no going around your back to the datacenter without your knowledge. It's also a publicity measure: it's a much bigger deal to try to serve a warrant to "The New York Times", or even your county newspaper, than a random datacenter.

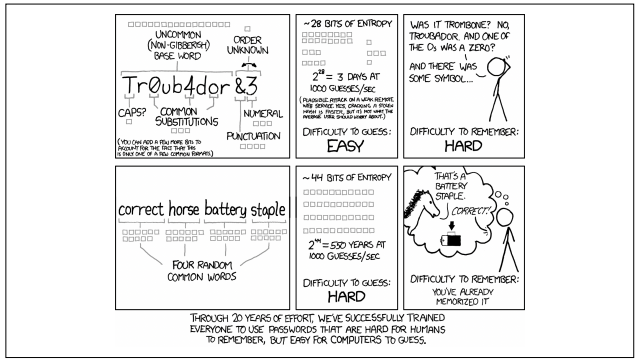
Once it's up and running, SecureDrop as a cryptographic system works in four steps:

1. A source uses Tor Browser to access a newsroom's SecureDrop. As you all know from reading about Tor, this means the network never sees that the source is connecting to SecureDrop, and the SecureDrop never sees the source's IP address.

2. A source sends a message, and maybe uploads some files, to the SecureDrop server, which encrypts them to the news organization's public key.



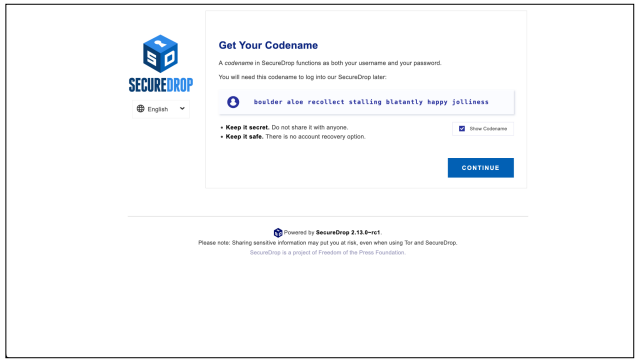
Along the way, they get a seven-word passphrase that is both their username and their password to this SecureDrop.



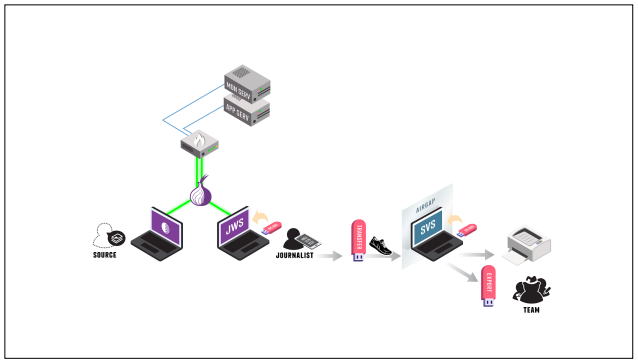
https://xkcd.com/936/?correct=horse&battery=staple

Passphrases like these are easier for people to remember.

You'd think that would make them easier for computers to crack, too, but they're harder, for this reason. From a computational perspective, normal passwords are made up of 26 uppercase letters, 26 lowercase letters, 10 digits—if we forget about the symbols, that's 62 characters. Passwords like these are made of up "characters" that are actually individual words, drawn from a dictionary of seven *thousand* possibilities.



So they get this password and have to memorize it or write it down. This is the only piece of information they get or need that links them back to this SecureDrop.



https://docs.securedrop.org/en/stable/what_is_securedrop.html

3. Someone at the news organization, usually a journalist or an editor, uses Tails and Tor Browser to check SecureDrop regularly and download new submissions.

4. They manually transfer these downloaded files to a completely separate computer, which we call the secure viewing station, to decrypt them using the news organization's private key. This computer also runs Tails, but it never connects to the Internet. This creates what's called an "air gap". Malware from the Internet can't reach it. And even if a submission to SecureDrop were to contain malware that could run on the secure viewing station, it would have no way to connect back to the Internet. That means a document can't use a PDF vulnerability, for example, to reveal that it's been opened, which would be interesting for some attackers to know. It can't do anything else, either, like take over the system or exfiltrate data from it.

If need be, the journalist can reply to the source for more information. The source can log back in later, using the same seven-word passphrase they got earlier, to decrypt the message and reply themselves.

Does anything stand out to you about this design? What limitations can you think of?

---

2. Formal definition of end-to-end encryption

An end-to-end-encryption service provides confidentiality, integrity, authenticity and forward secrecy between ends.

In the context of messaging, confidentiality implies that a system that uses "end-to-end [...] encryption would conceal communications between one user's instant messaging application through any intermediate devices and servers all the way to the recipient's instant messaging application." [dkg] Confidentiality is broken if content can be decrypted at any intermediate point.

https://datatracker.ietf.org/doc/html/draft-knodel-e2ee-definition-11#name-formal-definition-of-end-to

[Not end-to-end–encrypted!
- Tor provides encryption in transit between the source and the server, and again between the server and the journalist. This is comparable to Transport Layer Security that provides the "https" in your Web browser.
- PGP provides encryption at rest on the server's hard disk, and these files are also sent to the journalist to decrypt.
- But it's not end-to-end encrypted. That would require doing the encryption in JavaScript in the source's browser, and it's a fundamental security principle of SecureDrop that we ask sources to connect with JavaScript *disabled*. We'll talk about this more later if we have time.
]

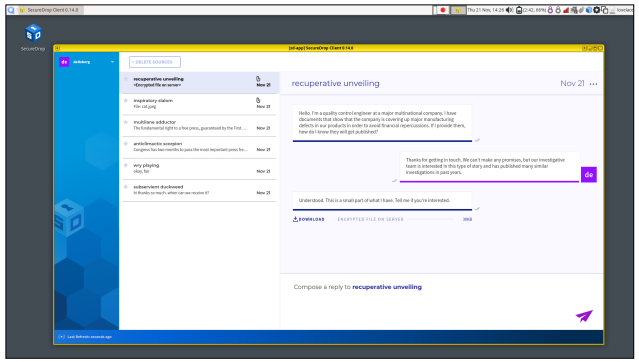
What I want to highlight—and credit—about this design is that it's both conservative and opinionated.

It's *conservative* in that it's built out of other people's tech, in particular tried-and-true open-source projects. Tor secures the connections. PGP—yes, *PGP*—secures the message and submission files on disk. SecureDrop itself is a couple of simple Web applications written in Python, using the Flask Web framework. If you're a programmer, and you've ever worked on a Web application, you'll be either amused or horrified to discover that there's no database server; it's just a SQLite database sitting on disk. Why? Because SecureDrops don't get a lot of traffic, and Tor guarantees that any traffic a SecureDrop does get will be slow. A SecureDrop doesn't need to scale or be fast. It needs to be simple, and slow is fine. Keeping it simple makes it easier for us to maintain and easier for news organizations to run—and harder for someone to attack. Letting it be slow—not deliberately, mind you, but at least not going out of our way to make it fast—makes it harder for someone who doesn't like *The New York Times*, say, to crash it by filling up the disk with junk submissions.

This design is *opinionated* in that it sets strict rules for how users can interact with it. With a very narrow exception for system administrators, the only way to talk to a SecureDrop is over Tor. The only way for a source to connect to a SecureDrop is via Tor Browser. In addition to encrypting and anonymizing connections over the Tor network, Tor Browser clears everything about your session when you quit, so a source's visit to a SecureDrop leaves no trace on their computer. That is, connecting and submitting to a SecureDrop must leave no incriminating clues on a source's computer. (Remember, the seven-word passphrase they received doesn't get saved or stored anywhere. The source has to remember it themselves, or write it down, and reenter it the next time they want to log into that SecureDrop.)

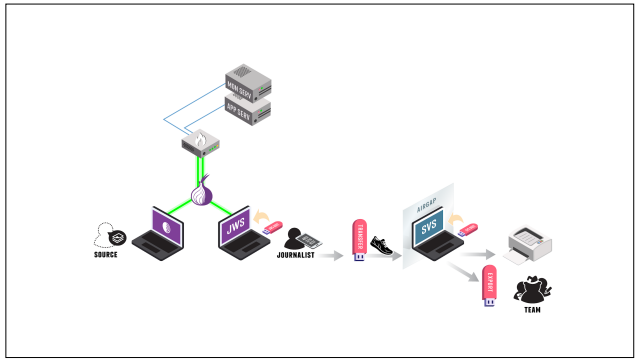
The only way for a journalist to connect to a SecureDrop is also using Tor Browser running on Tails. Tails, or The Amnesiac Incognito Live System, is a version of Linux that extends this "amnesia" property of Tor Browser to the entire operating system. And we encourage sources to use Tails if they can, too, ideally from a public place like a café or library rather than their own home Internet connection. Finally, the secure viewing station—the "air gap"—is the only place journalists can ever decrypt submissions they download from SecureDrop.

To be clear: Most systems can't get away with a design like this. SecureDrop isn't *difficult* to use, but it's *clunky* to use, because it does one thing and makes you do it in a rigid way. There are very few settings, and there are almost no shortcuts. There are very few systems—outside of regulated domains like military, medical, and financial technology—that let you be this strict about your users' experience.



https://workstation.securedrop.org/en/stable/journalist/sources.html

But we *do* want to make it easier, and that's one of the things we've started working on in the last few years. I thought I would have time to talk about it today and I don't. But we have more details available online, or come talk to me afterwards.

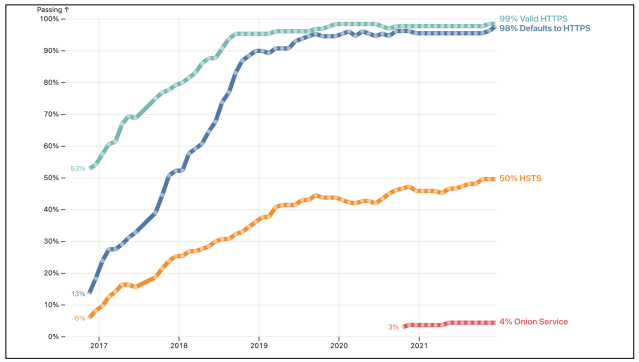


https://docs.securedrop.org/en/stable/what_is_securedrop.html

Questions? Comments?

Signal has entered the chat:
or, what we can and can't learn from general end-to-end–encrypted messaging.

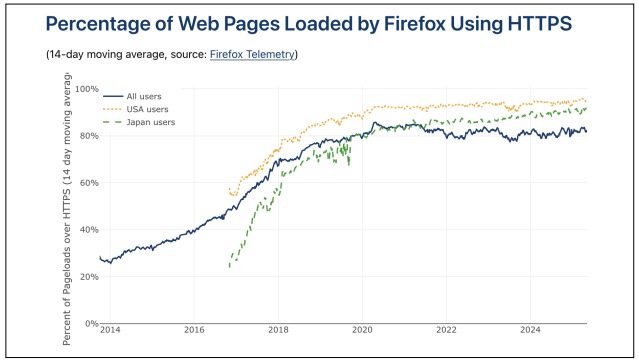
The past decade has been good for encryption.



https://freedom.press/about/announcements/five-years-of-secure-the-news/

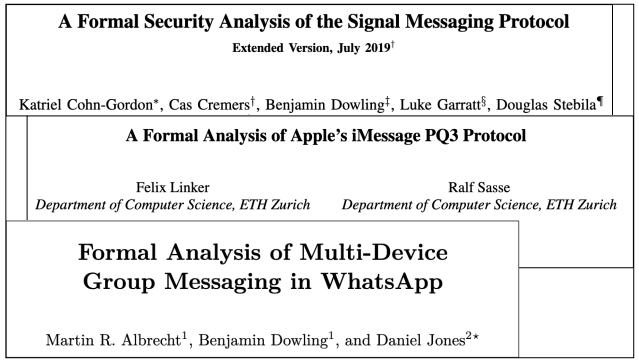
Most major Web sites now default to encrypted "https" by default. FPF used to run a project called "Secure The News". It was basically a bot that checked major news sites regularly and kept track of which ones use "https" encryption.

Now, why does this matter, since reading the news doesn't leak passwords or other confidential information? Until recently, it wasn't possible to hide from the network *that* you were connecting to "The New York Times". But there's no reason for the network to know exactly what articles you're reading too—that *could* be sensitive. So FPF, mostly before my time, ran this tracker as a technical advocacy project, to put pressure on news organizations to publish encrypted Web sites. We ran it until it looked like this: pretty much everything was encrypted by default. We declared victory and shut it down.



https://letsencrypt.org/stats/

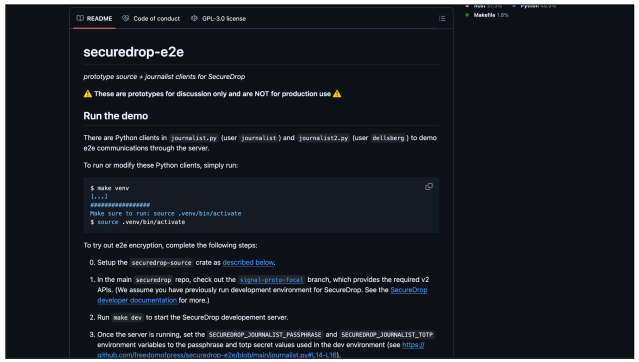
In fact, most Web traffic, period, is now encrypted by default.



https://eprint.iacr.org/2025/794.pdf
https://eprint.iacr.org/2024/1395.pdf
https://eprint.iacr.org/2016/1013.pdf

And major texting apps, including WhatsApp and iMessage, are encrypted within their networks. Things get more complicated in groups and across networks, especially between iOS and Android; I won't go into the details here. And generic SMS is still totally unencrypted. But there's been a lot of progress, reflecting a lot of great work being done in the research, standards, and engineering communities.

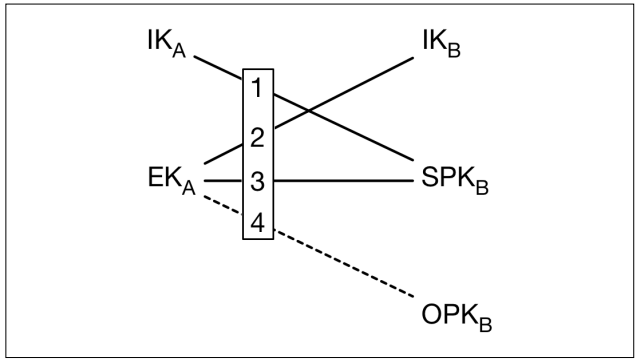
You may be wondering: If SecureDrop isn't end-to-end encrypted, and we want it to be, why can't we just use one of these messaging protocols? If Signal is state-of-the-art secure messaging already widely used by journalists, why not use it inside SecureDrop?



https://github.com/freedomofpress/securedrop-signal-poc

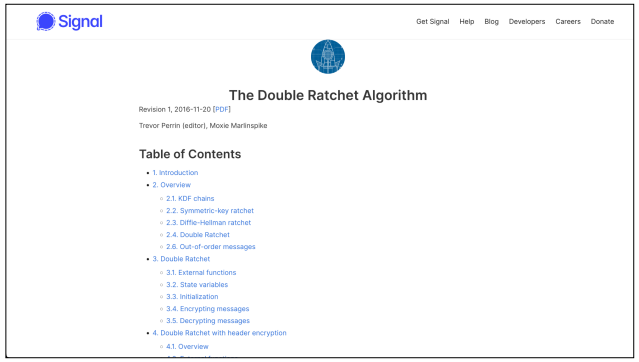
We tried! In fact, a couple of years ago, one of my predecessors prototyped exactly this approach.

But you already know that what we call "the Signal protocol" is actually the composition, or combination, of two different cryptographic protocols. You just read these papers a few weeks ago, so I'll just gesticulate at these two pictures:



https://signal.org/docs/specifications/x3dh/#sending-the-initial-message

1. X3DH gives you and your contact initial key agreement.

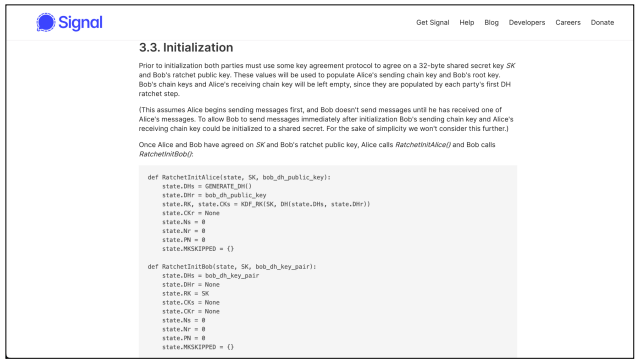


https://signal.org/docs/specifications/doubleratchet/

2. The double-ratchet algorithm keeps your keys rolling forward to preserve perfect forward secrecy.

But remember: a source needs to be able to come back to a SecureDrop using only the seven-word passphrase they first received when they submitted. SecureDrop isn't supposed to leave any other traces on a source's computer. And since they connect via Tor Browser, any other information—what we call "state", or the information the protocol currently needs to run—will be cleared when they quit.

How much state does the double-ratchet algorithm require a participant to retain and keep updated?



Signal

Get Signal Help Blog Developers Careers Donate

3.3. Initialization

Prior to initialization both parties must use some key agreement protocol to agree on a 32-byte shared secret key *SK* and Bob's ratchet public key. These values will be used to populate Alice's sending chain key and Bob's root key. Bob's chain keys and Alice's receiving chain key will be left empty, since they are populated by each party's first DH ratchet step.

(This assumes Alice begins sending messages first, and Bob doesn't send messages until he has received one of Alice's messages. To allow Bob to send messages immediately after initialization Bob's sending chain key and Alice's receiving chain key could be initialized to a shared secret. For the sake of simplicity we won't consider this further.)

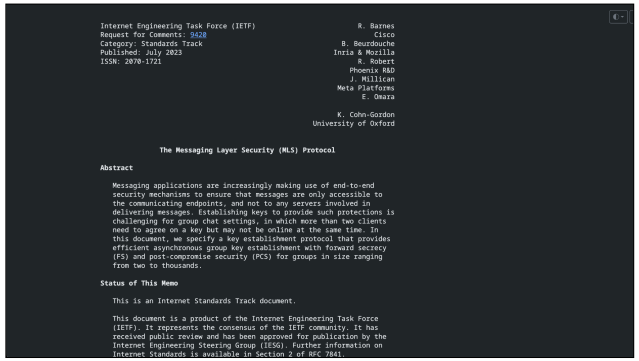
Once Alice and Bob have agreed on *SK* and Bob's ratchet public key, Alice calls *RatchetInitAlice()* and Bob calls *RatchetInitBob()*:

```
def RatchetInitAlice(state, SK, bob_dh_public_key):
    state.DHs = GENERATE_DH()
    state.DHr = bob_dh_public_key
    state.RK, state.CKs = KDF_RK(SK, DH(state.DHs, state.DHr))
    state.CKr = None
    state.Ns = 0
    state.Nr = 0
    state.PN = 0
    state.MKSKIPPED = {}

def RatchetInitBob(state, SK, bob_dh_key_pair):
    state.DHs = bob_dh_key_pair
    state.DHr = None
    state.RK = SK
    state.CKs = None
    state.CKr = None
    state.Ns = 0
    state.Nr = 0
    state.PN = 0
    state.MKSKIPPED = {}
```

https://signal.org/docs/specifications/doubleratchet/#initialization

An awful lot. That's fine for Signal running in an app on your phone.



Internet Engineering Task Force (IETF)
Request for Comments: 9420
Category: Standards Track
Published: July 2023
ISSN: 2070-1721

R. Barnes
Cisco
B. Beurdouche
Inria & Mozilla
R. Robert
Phoenix R&D
J. Millican
Meta Platforms
E. Omara

K. Cohn-Gordon
University of Oxford

The Messaging Layer Security (MLS) Protocol

Abstract

Messaging applications are increasingly making use of end-to-end security mechanisms to ensure that messages are only accessible to the communicating endpoints, and not to any servers involved in delivering messages. Establishing keys to provide such protections is challenging for group chat settings, in which more than two clients need to agree on a key but may not be online at the same time. In this document, we specify a key establishment protocol that provides efficient asynchronous group key establishment with forward secrecy (FS) and post-compromise security (PCS) for groups in size ranging from two to thousands.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

https://datatracker.ietf.org/doc/html/rfc9420

It's also fine for an even more general-purpose protocol like Messaging Layer Security running in apps on your phone or Web apps in your Web browser. MLS is the latest and greatest thing to come out of the Internet Engineering Task Force, and it's designed to generalize and standardize secure messaging, including between messaging ecosystems like Google's RCS.

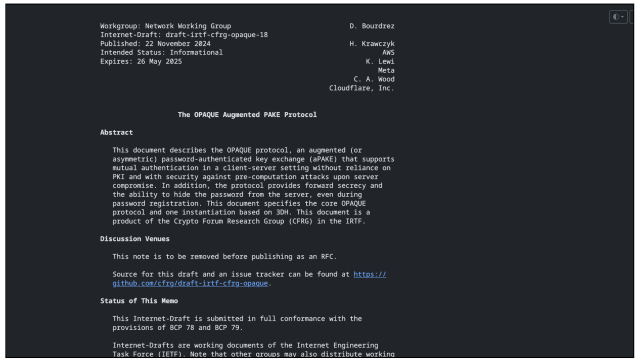
But keeping this much state around doesn't work for SecureDrop, at least not if the goal is to leave no trace on a source's computer.

To be clear: Signal and MLS are great protocols. They are the absolute state of the art for general-purpose secure messaging, in which two or more parties are running normal apps that can store the state of the conversations and groups they're part of. We just can't use them for SecureDrop! We wish we could.

Questions? Comments?

The SecureDrop Protocol: or,
designing for the first-contact problem today.

So here you have the conundrum we began thinking about a few years ago. SecureDrop, as an anonymous first-contact system, is characterized by this double-edged asymmetry. On the one hand, you only have to preserve anonymity for one half of the conversation, which means you only have to solve half of the key-distribution problem. On the other hand, only one half of the conversation can keep state, so you can't use a generic secure-messaging protocol—or indeed any messaging protocol that assumes that both halves of the conversation have the same resources and properties.



Workgroup: Network Working Group
Internet-Draft: draft-irtf-cfrg-opaque-18
Published: 22 November 2024
Intended Status: Informational
Expires: 26 May 2025

D. Bourdrez
H. Krawczyk
AWS
K. Lewi
Meta
C. A. Wood
Cloudflare, Inc.

The OPAQUE Augmented PAKE Protocol

Abstract

This document describes the OPAQUE protocol, an augmented (or asymmetric) password-authenticated key exchange (aPAKE) that supports mutual authentication in a client-server setting without reliance on PKI and with security against pre-computation attacks upon server compromise. In addition, the protocol provides forward secrecy and the ability to hide the password from the server, even during password registration. This document specifies the core OPAQUE protocol and one instantiation based on 3DH. This document is a product of the Crypto Forum Research Group (CFRG) in the IRTF.

Discussion Venues

This note is to be removed before publishing as an RFC.

Source for this draft and an issue tracker can be found at https://github.com/cfrg/draft-irtf-cfrg-opaque.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

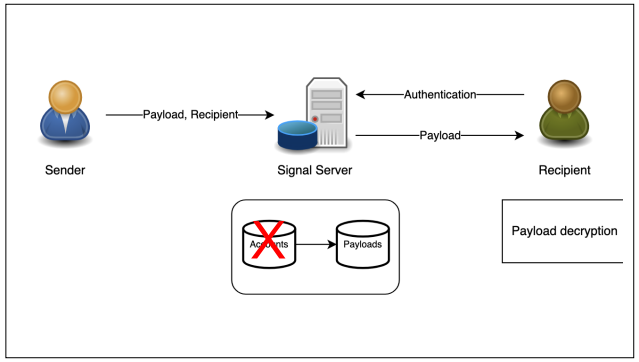
Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working

Now, there are various clever approaches you could take to solving the state problem. For example, you could use a password-authenticated key exchange, or PAKE, on the client, taking exactly the long passphrase we just talked about, to encrypt all that state and store it on the server. Then, when the source comes back, they enter that passphrase again, download the encrypted state, decrypt it, and start running something like Signal's double-ratchet algorithm again. Great.

And you absolutely could do this. We looked into it.

However, once you've accepted that you're going to need a custom protocol, you can begin to think creatively, or paranoiacally, about what other properties you might want to build in while you're at it. For us, this has meant using this asymmetry as our prompt to think more strictly and more rigorously about how much information we can hide from the server itself, so that it can never reveal it even if it's compromised. That opens the door to hosting more SecureDrops in more places. In the future, these might include cloud environments, where you don't trust the underlying infrastructure. But at a minimum we want to offer better protections to SecureDrops hosted outside the US, without Fourth Amendment protections against search and seizure. (Or in the US, for that matter.)

I'm not going to have time to go into the cryptographic details, but I'll sketch a few of the moves this asymmetry has let us make in the SecureDrop Protocol.

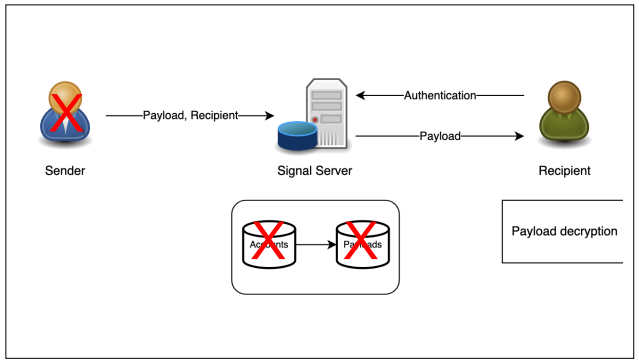


https://securedrop.org/news/introducing-securedrop-protocol/

Let's say you do this PAKE-encrypted-state thing. That's fine if you run a system that has thousands or millions of users. The state's encrypted. So long as you trust the encryption, there's nothing interesting about *having* a bunch of encrypted state lying around, right?

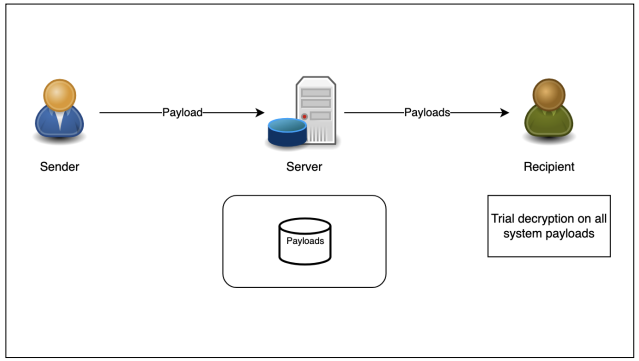
[Can anyone think of any problems with this approach? What could go wrong?]

What if you have a sensitive system that has only one or two users a day? What if those users are, by definition, whistleblowers trying to contact a journalist? Suddenly that state becomes a lot more interesting. Suddenly the *existence* of that state becomes interesting itself. New bit of encrypted state pops up? Guess "The New York Times" is getting a new leak.



https://securedrop.org/news/introducing-securedrop-protocol/

So we don't want to do that. Not doing the PAKE-encrypted-state thing means that sources have *no* place to store state between visits, since they can't store it locally in the amnesiac Tor Browser either.



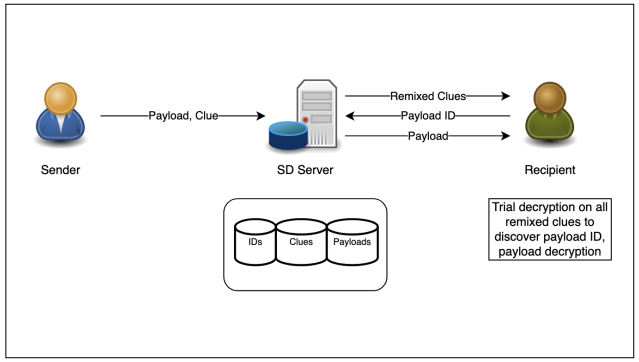
https://securedrop.org/news/introducing-securedrop-protocol/

So let's try something else. We can design a system where let sources fetch *all* the messages that are available on the server at any given time, and we trust that they can only decrypt the ones that are intended for them. This is called trial decryption, and it works handily when you don't have server-side accounts to track who your users are.

[Can anyone think of any problems with this approach? What could go wrong?]

But there are two problems with this. The first is that now we're letting anyone on the open Internet see how many messages we have, even if we're confident that only the intended recipients can decrypt them. Again, for WhatsApp, this is just a trade secret; but for a particular SecureDrop, this is pretty interesting information in itself.

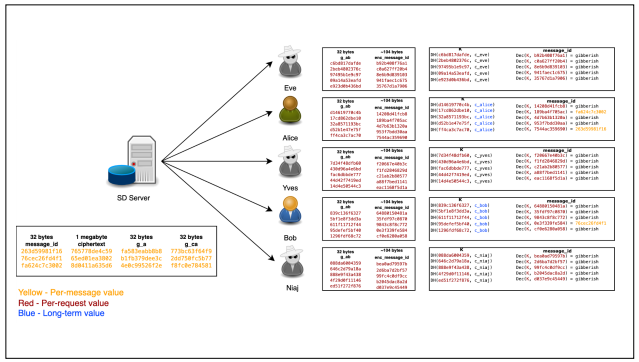
The second problem is that this puts enormous pressure on the strength of our encryption scheme. Let's say SecureDrop used this. If you are say, the NSA, there's nothing to stop you from taking a drive-by snapshot every day of every SecureDrop in the world and waiting until a cryptographically-relevant quantum computer comes around to decrypt all those messages. Maybe by then all the leaks have been published, but sources, the whistleblowers, are probably still anonymous. No good.



https://securedrop.org/news/introducing-securedrop-protocol/

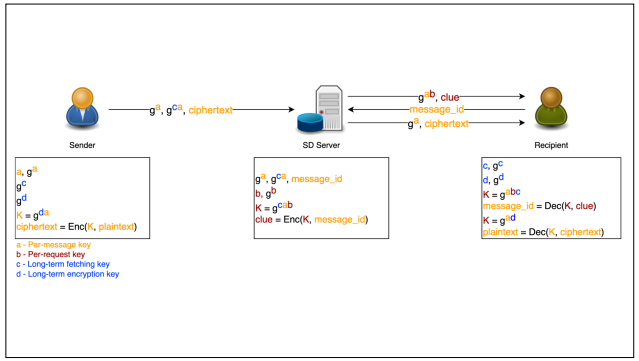
So now we add another layer. Let's make it so that the only thing anyone on the open Internet can download is a set of "clues" or "challenges".

You'll always get the same number; let's say five hundred. If you have a message waiting for you, one of those five hundred clues will be encrypted to you, and you can decrypt it. This is trial decryption again.



https://securedrop.org/news/introducing-securedrop-protocol/

But what you actually decrypt will be the information you need to go back to the server and download exactly that message, which you can then decrypt normally. But if you don't have any new message, you'll just get five hundred clues you can't decrypt, so you don't learn anything about how many messages the server actually has right now. And the clues are regenerated every time you request them, so you don't learn anything about how many messages the server has over time, either. That means you can't tell when a SecureDrop might have a new leak submission—or even when it's actively relaying a conversation for a source.



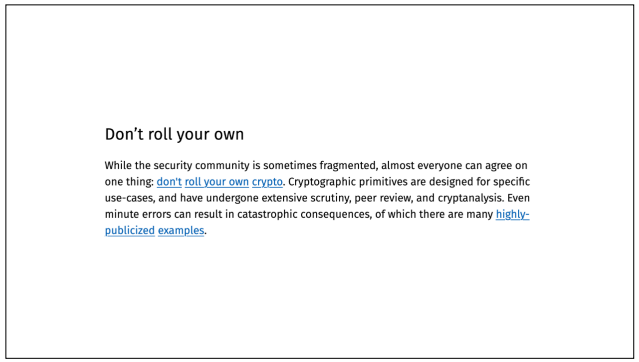
https://github.com/freedomofpress/securedrop-protocol

I'm glossing over a lot of details—they're in the readings and online if you want them.

[Questions? Comments?]

Some lessons we've learned
(and are still learning)

I've burned through most of the time I have and I want to make sure we have time for discussion. So let me conclude with some of my own observations from getting to work on this over the past few years.

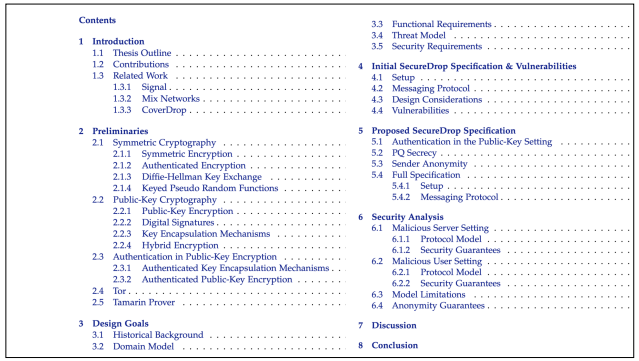


https://securedrop.org/news/how-to-research-your-own-cryptography-and-survive/

My colleagues and I are not cryptographers.

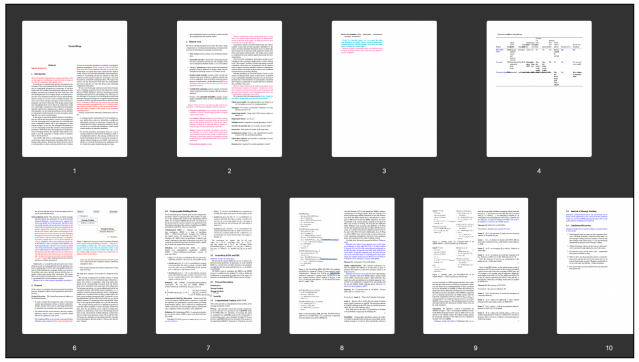
What we have—our blessing and our curse—is that we have a cryptographically interesting problem. Specifically, we have a problem that belongs to a domain that lots of cryptographers have already thought a lot about—secure messaging—plus some interesting constraints that break some of the general assumptions of that domain. To our luck, some of these same cryptographers find these constraints interesting too. So we've been lucky to have the resources, and increasingly the network, to pose a cryptographic problem and get help finding cryptographic solutions.

Software development is always iterative, and there's a particular version of iterative design that's taken shape here that I think is worth articulating. We have this cryptographic problem. We find that we can't use something off the shelf to solve it. So we start thinking about the properties we'd ideally like to have, and we start to sketch systems that achieve those properties in different ways.

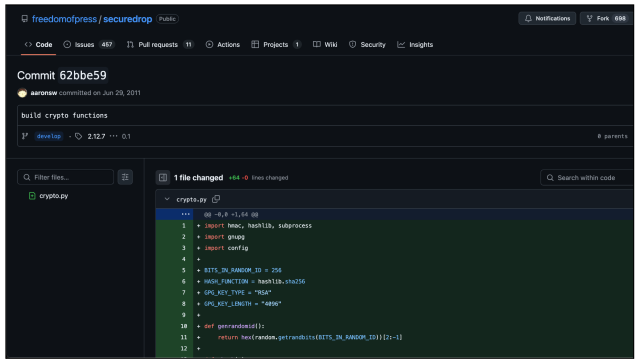


https://github.com/lumaier/securedrop-formalanalysis

Then we try to prove that it in fact does achieve those properties. If you did the optional reading for today, this is the master's thesis that Luca Maier completed in January at ETH Zürich: proving that the SecureDrop Protocol, with some improvements, can do the things we want it to do.



Now we go back to the beginning. If these properties are achievable, if this system is feasible, then can we find other ways to get there? In particular, can we reuse cryptographic primitives that are better understood, more standardized, and better proven? Then we have to prove that *those* work for us. That's where we are now. We're working with Felix Linker, Shannon Veitch, and Kenny Paterson at ETH Zürich to finish a formal model and proofs of the SecureDrop Protocol using slightly more standardized primitives and clearer security guarantees.



https://github.com/freedomofpress/securedrop/commit/62bbe590afd77a6af2dcaed46c93da6e0cf40951

Now, thinking about this talk in the context of this class, I've found myself wondering: Would this approach come as a surprise to Aaron Swartz?

I mean this with no disrespect and a lot of humility: Would this surprise a lot of cypherpunks?

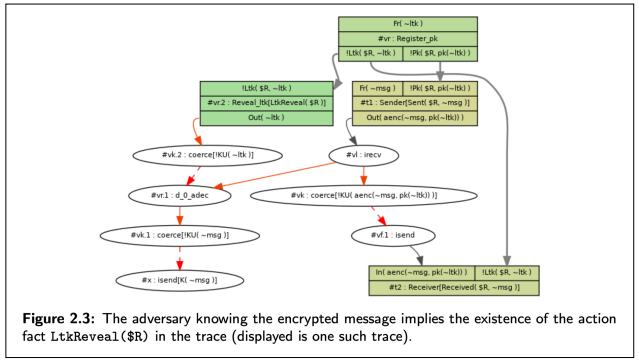
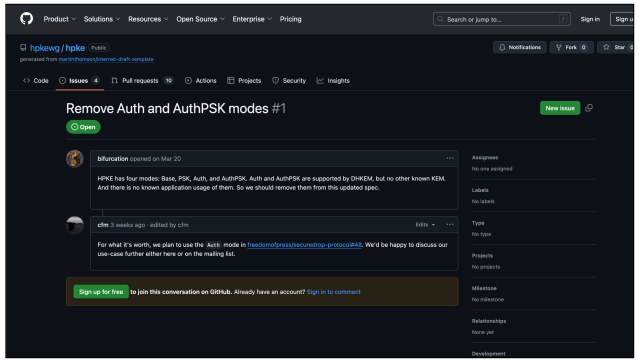


Figure 2.3: The adversary knowing the encrypted message implies the existence of the action fact `LtkReveal($R)` in the trace (displayed is one such trace).

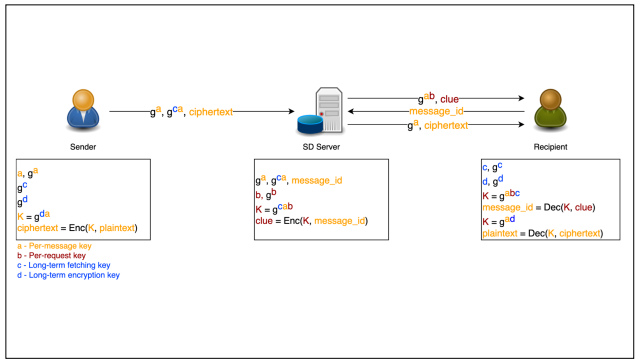
https://github.com/lumaier/securedrop-formalanalysis/blob/main/thesis.pdf

It's not very punk to rely on formal models and formal proofs.



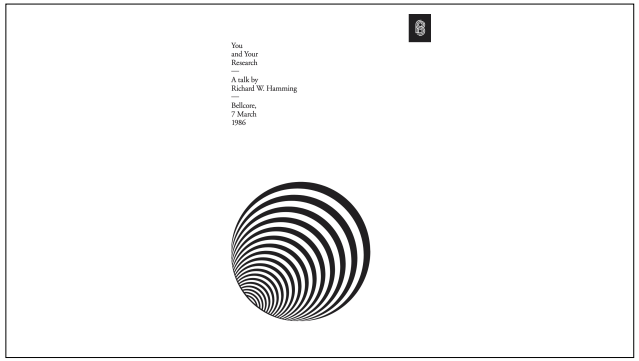
https://github.com/hpkewg/hpke/issues/1

It's also not very punk to spend time worrying about whether a standards body is going to take away an encryption mode you were hoping to use.



But it is my belief that a responsibly designed communications system in 2025 must be end-to-end–encrypted. I see no reason why SecureDrop should be an exception to that obligation. Indeed, I believe this obligation is all the more critical for us in our high-stakes niche.

Meanwhile, it's become clear that building meaningful end-to-end–encryption for SecureDrop means building something custom. And so doing *that* meaningfully means using every technique and collaboration at our disposal. That's all the more true for us as an organization that's lucky to have the resources to do so.



https://d37ugbyn3rpeym.cloudfront.net/stripe-press/TAODSAE_zine_press.pdf

The last note I want to hit is this. A mentor of mine many years age introduced me to a talk the mathematician Richard Hamming gave at Bellcore in the eighties entitled "You and Your Research". I won't quote directly from it, because it has a lot of language about "average scientists" and "great scientists", and in this room I have the liability of not being a scientist at all. But I will recommend it, with this endorsement. One of the privileges of this project has been discovering how much good work there is to do on a small, narrow, niche problem. Broadly speaking, secure messaging is a solved problem, so why work on it? Well, it isn't a solved problem for us, for all the reasons I've just sketched.

Now, that's interesting to me, both as an engineer and as a recovering philosopher. What does it mean when a conversation is fundamentally asymmetrical? What does that mean about the cryptographic objects you can use to secure it? How can you hide an asymmetric, low-volume, relatively rare event on a network? Put another way: So much of cryptography, and so much of computer science in general, is rightfully concerned about how to do things fast and at enormous scale. What changes, for good and for ill, if you have to do them at a very small scale? What techniques can you use that you couldn't use otherwise? What are the fundamental information-theoretic limits you hit anyway?

Maybe not so cypherpunk. Maybe more cypher than punk. But we are definitely trying to use "strong cryptography [...] as a means of effecting social and political change".

Things I didn't have time to talk about (but we still could!):

- Usable security
- Code verification and authentication in Web applications
- Quantum resistance
- Applications versus protocols

How to reach us:

https://securedrop.org

https://github.com/freedomofpress/securedrop

https://freedom.press

How to reach me:

cory@freedom.press

Signal: `cfm.38`

(With thanks to: Rowen S. at FPF for feedback!)

Thanks for your time.