Flexible Channels: Preventing Metadata Leakage in Communication over Public Channels

Jacob Lindahl lindahl@prg.is.titech.ac.jp, Hidehiko Masuhara masuhara@prg.is.titech.ac.jp

1 Abstract

Public blockchain ledgers are, at first glance, antithetical to privacy: all data are recorded permanently and publicly. While this is necessary, in many cases, to trustlessly verify the execution of the virtual machine, by the same token, blockchains are not often used to directly store sensitive information. However, popular blockchains do provide data distribution (data availability), historical execution auditability, and data accessibility that have interesting implications for encrypted messaging. The disadvantages for using a blockchain as the underlying middleman for an encrypted messaging system are clear and numerous: cost, privacy, efficiency, etc. We present a protocol that attempts to mitigate these issues while taking advantage of the unique mechanisms that blockchains do provide, and provide recommendations for similar projects.

2 Definitions

2.1 Metadata

The goal of this paper is to build upon the work of previous protocols to hide even more metadata about conversations. In particular, we will hide the following information, in addition to hiding of the payload itself:

- Sender's identity.
- Sender's location (geographical and network).
- Timestamp of transmission.
- Receiver's identity.
- Receiver's location (geographical and network).
- Timestamp of receipt.
- Payload size.
- Conversation history.
- Forward secrecy.

However, privacy of these data is not sufficient to make a usable protocol. Therefore, we will also ensure the following properties that make the protocol usable:

- Users can easily use the service across multiple devices, including message synchronization.
- Group messaging is efficient and scalable.
- The service is inexpensive to run as a server and as a user/client.

2.2 Conversation history

One of the issues experienced by many protocols in this sector is that while the messaging protocol may be clearly cryptographically and mathematically sound, correctly implementing such fancy techniques as deniability, if transcripts of the conversation are revealed, those hard mathematical evidences do very little to effectively recuse a conversant from a conversation. This has led some protocols to discount such techniques entirely. (Jefferys 2020)

The experimental techniques presented in this paper do not endeavor to implement complete deniability in the traditional sense, due in large part to the nature of the invariants required by the infrastructure upon which they depend. That is to say, it would violate the fundamental contract of an "append-only public ledger" if two plausible transcripts could be provided that purport a different sequences of appends.

Rather, we take a different approach. One of the problems with simply implementing something like the Signal Double-Ratchet algorithm is that while it hides the *content* of the messages between conversants Alice and Bob, it does not hide the fact that Alice and Bob are 1) conversing, or 2) conversing with each other. The flexible channels protocol in itself does attempt to conceal this information. However, it should be duly noted that the protocol as presented assumes the existence of some sort of public-key infrastructure (PKI). PKIs are usually publicly-accessible, so the presence of a user's public key in the PKI could belie their usage of the protocol. This issue can be mitigated somewhat by 1) using a PKI that has sufficient quantity of users for a diverse variety of applications, or 2) not using a public PKI, and instead manually facilitating public key exchanges (e.g. by meeting in person, scanning QR codes, etc.).

3 Prior art

4 Protocol

The key insights of this paper are the channel and sequence hash constructions.

A channel is a total-ordered stream of messages. It consists of membership list (one or more members of a group) and a shared secret (derivable by Diffie-Hellman or any other method of establishing a shared secret).

A channel has a deterministic sequence hash generator. Hashing the fields of the channel, together with a nonce, produces a "sequence hash." This sequence hash can only be recreated by parties privy to the channel's shared secret, yet it does not reveal the shared secret. The sequence hash serves as the identifying key of a message sent to the message repository.

The message repository is a simple construct, consisting only of a key-value store from which anyone can read, and to which anyone can write. The only restriction is that existing keys cannot be overwritten.

Using a series of abstractions, the channel construct can be composed in a number of different ways, supporting $1 \leftrightarrow 1$, $1 \leftrightarrow N$, and $N \leftrightarrow N$ messaging.

Of particular interest is $N \leftrightarrow N$ messaging. If all members of a particular group share the same secret, they can all generate the same sequence of hashes, and decrypt all of the messages in the channel. This allows for posting a single message, encrypted with the shared secret, to the public message repository, and all of the members of the group will be able to read it, regardless of the number of members in the group. Thus, we have $O(1)$ space complexity for broadcast transmissions.

4.1 Channels

Consider a channel with members Sender Steve and Receiver Robin $\mathcal{C}_{\{S,R\}}$, where the public keys v_S and v_R are mutually known. Using Diffie-Hellman or any other method of establishing a shared secret, Steve and Robin can establish a shared secret $k_{\{S,R\}} = \text{Diffie-Hellman}(S, R)$. In combination with other, optional, static metadata, a similarly secret channel identifier $i_{\{S,R\}}$ can be derived. An example of this process is demonstrated below:

$$i_{\{S,R\}} = H(k_{\{S,R\}}, v_S \parallel v_R)$$

where H is a keyed cryptographic hash function (such as HMAC-SHA3-512).

The channel identifier $i_{\{S,R\}}$ is used to identify the channel $\mathcal{C}_{\{S,R\}}$. However, this identifier should not be publicized because it can be used to derive the sequence hashes for the channel before they have been posted to a public message repository. Additions to the protocol relieve the need to keep individual sequence hashes secret, however, revealing a channel identifier would make it possible to generate all sequence hashes for a channel, proving that they are linked.

4.2 Generating sequence hashes

Once a channel has been established, the sender and receiver can generate a sequence hash $h^n_{\{S,R\}}$ for the nth message in the channel. This sequence hash is used to identify the message in the message repository. The sequence hash is generated as follows:

$$h^n_{\{S,R\}} = H(i_{\{S,R\}}, n)$$

where n is the sequence number of the message.

4.3 Posting and reading messages

Once a sequence hash has been generated, the sender can post a payload $(h^n_{\{S,R\}}, c^n)$ to the message repository, where c^n is the ciphertext of the nth message, encrypted with the shared secret $k_{\{S,R\}}$. Once the message has posted, Receiver Robin, knowing the shared secret $k_{\{S,R\}}$ and the index of last message he saw $n-1$, can find the message keyed by $h^n_{\{S,R\}}$ in the message repository, and decrypt it.

4.4 Group abstractions

When multiple actors have read and write access to a channel by virtue of knowing the channel shared secret, how do they synchronize which channel member is entitled to use which sequence numbers?

As long as all member of a channel have full knowledge of the other members of the channel, this is a trivial problem to solve. The protocol simply defines a deterministic sort order according to which the channel members' respective identifiers are ordered A. Then, the ith message sent by Steve uses sequence number $i|A| + \mathrm{indexof}(A, S)$. This ensures that each channel member has a predefined set of sequence numbers assigned to them which all other channel members know. This formulation also prevents the race condition where two different members from attempting to use a nonce simultaneously.

However, the downside to this approach is that it requires observers of the channel to check for $O(n)$ message receipts where n is the number of members in the channel. In the case of large channels, this can cause excessive network activity even when the channel itself has low activity. This can be somewhat mitigated with the message notification extension described below.

4.5 Garbage messages

While any party without knowledge of the channel's shared secret is incapable of decrypting the messages within a channel, if a channel member's network activity is being monitored or if the message repository that a channel is using has very low traffic, it becomes possible to deduce increasingly detailed metadata about channel participants even if the contents of the channel messages themselves remain uncompromised.

Channel members can leverage the notion of garbage messages to improve the privacy of themselves and other users of the same message repository. We take advantage of the fact that channel identifiers

cannot be derived from their constituent sequence hashes, nor can a link be proven between two sequence hashes while their channel identifiers remain unknown.

To produce a garbage message, a user simply generates a channel with a set of faux members. These members, while they do not exist in the real world, cannot be proven to not exist, and messages sent to channels containing them will appear indistinguishable from any other message sent to the repository, provided the other metadata is sufficiently anonymized as described in the following sections.

We have established that a user *can* send garbage messages, but *when* should the user send them? Ideally, the garbage messages would be sent continuously at some sort of Poisson-distributed frequency such that when the user chooses to send a real message, it completely blends in with the garbage. Of course, this once again imposes a significant network usage load onto the user, as well as requiring their client to be continuously online and broadcasting.

Requiring a client to be continuously online is a perfectly fine stipulation for a protocol to make, but in the case of this protocol, it well-nigh defeats the purpose of having a message repository in the first place. We will return to this point again when we discuss proxies.

4.6 Payload size

One of the most obvious pieces of metadata in an encrypted message is its size. A malicious actor observing network traffic over a long period of time may detect patterns in payload size which could allow the malicious actor to trace users across time.

The trivial solution to this is to enforce a regular payload size to which all messages must conform. This approaches a good solution to this problem, but it also introduces a number of drawbacks.

First, and most obviously, is that if the messages in question are much smaller than the standard size, it will result in a lot of wasted space on the message repository. If the message repository in question is a blockchain (a natural choice in many respects for this protocol), storage is quite expensive (TODO: cite), even orders of magnitude beyond a commercial cloud provder like Amazon S3.

A converse problem of a regulated message size is for messages where the contents are longer than the regular size. Not only does this impose an overhead in the necessity of breaking up large messages to fit into the regular size, but the metadata that must be included in order for the receiver to properly parse the incoming messages may be cumbersome (or large).

However, even worse than the inefficiency problem are the potential privacy implications. Let us say, for example, that the regulated message size were 1 kibibyte (1024 bytes), but the user wished to send a 10-kibibyte message? Of course, the user must send at least 10 standard-sized messages to transfer the data outright. Unfortunately, if the user wishes to do this with any degree of efficiency, he must send those messages all in quick succession, even simultaneously. From the perspective of a party

observing the user's network traffic, 10 simultaneous 1-kibibyte messages is not really much better than simply sending a single 10-kibibyte message.

Therefore, we expand this idea by introducing a range of regulated message sizes, e.g. 1-kb, 4-kb, 16-kb, 64-kb, to which each message must conform. It the responsibility of the client to choose the most effective set of message sizes to use when sending a particular message. The client could choose to send a 3-kb message in 3 1-kb messages, or in a single 4-kb message, or in two 64-kb messages, depending on how the client implements a size-selection algorithm: optimizing for efficiency or masking?

Some kinds of messages may simply be too large to make practical use of a protocol with standardized message lengths, such as photographs and HD video. In this case, we recommend providing the actual contents over a separate medium (e.g. a centralized hosting provider), and simply transmitting the URI for the contents over the channel.

Combining payload size manipulation with garbage messaging provides us with a good set of tools to use to emit noise into the protocol amongst which genuine communications can hide.

4.7 Proxies

Within the protocol itself we have already discussed many facets of metadata concealment. However, as other projects have noted in the past (cite: Monero (dandelion), Tor), the network transport layer has the potential to leak metadata about participants as well.

Since network activity to this protocol is easily identifiable, it becomes necessary to implement some sort of hiding technique to conceal additional information such as origin and destination IP addresses. If an implementation of the protocol only includes such features as described previously, a malicious actor could observe network traffic to and from the message repository to discover that Sender Steve is a user of the protocol. Or, even worse, if the message repository is hosted on a cloud service that does not respect the privacy of its clients, or if it is controlled by, for example, a malicious state actor, the message repository itself could discover the origin IP address of Steve. The privacy of a message recipient could be discovered in a similar manner.

Although it is not strictly required by the protocol, it is convenient to use a blockchain as the message repository for this protocol. This creates an even more pertinent problem related to the transport layer in that blockchain activity is both permanent and public, meaning that interactions with the message repository are permanently visible to the entire blockchain. It is a trivial task to open up a blockchain explorer and retrieve historical activity between different accounts, "pseudo"-anonymous as they may be.

Therefore, it becomes necessary to introduce a further layer of separation between the actual user of the messaging protocol and the message repository. We propose the use of "message proxies."

A message proxy is a service that sits in-between the user of the protocol and the message repository. It is the responsibility of the proxy service to ingest messages sent by users and forward them to the repository, effectively eliminating metadata from the message that might reveal the identity of the user. From a network transport layer perspective, as well as from the perspective of the blockchain, messages would then be coming from the proxy service instead of directly from the user.

Of course, this merely kicks the can down the road, per se, since now the trust issues associated with a compromised message repository are now conferred upon a compromised proxy service.

Furthermore, while we have previously assumed that a user may interface and transact directly and, most importantly, *immediately*, with a blockchain, that is not remotely true. In fact, the proxy service merely makes the problem more obvious: that a man-in-the-middle may intercept a message from a genuine user, replace the message payload with anything else, and submit the maliciously-constructed message to the blockchain before the genuine one has a chance to make it, thereby "sniping" a real sequence hash. If the proxy server is itself untrustworthy, it is even easier to steal sequence hashes from authentic payloads and never forward said payloads to the message repository in the first place.

4.7.1 Ensuring proxy honesty

Therefore, it becomes necessary to enforce some sort of safeguard that prevents a malicious actor from stealing genuine sequence hashes from authentic payloads while they are in flight. Until this point in the description of the protocol, the message repository has been a low-complexity service, merely enforcing that values associated with stored keys may not be overwritten.

Now we introduce another criterion: a proof that the value to be stored under a key actually *belongs* in that storage slot. Since the message repository does not necessarily impose authentication measures, each proof must be self-contained. This use-case dictates that the proof must show that some property of the payload links it to the preimage of the sequence hash. Recall that the sequence hash is composed of, among other things, the shared secret among channel members. At first glance, including some sort of digital signature seems to accomplish a great deal of our goal, but closer inspection reveals that a digital signature must be verified against a known public key, the use of which would compromise the identities of participants and/or link multiple messages from the same channel together, eliminating many of the protocol's desirable qualities.

Therefore, instead of relying on traditional signature verification, we turn to a new player on the modern cryptographic stage: zero-knowledge proofs (ZKPs). A ZKP proves to a verifier that a prover is in possession of some knowledge without revealing to the verifier anything about that knowledge.

There are many different systems that can be used to generate zero-knowledge proofs, but the specifics of choosing a proof system is well beyond the scope of this paper. Rather than overconstrain this

protocol description with a specific implementation, we recommend optimizing for proof verification efficiency when choosing a system, since the use-case involves verifying many small proofs.

In our case, Sender Steve wants to prove that he knows the secret channel identifier $i_{\{S,R\}}$ necessary to both produce the sequence hash and encrypt the accompanying ciphertext without revealing it to the proxy, the message repository, or anyone with read access to the message repository. The ZKP circuit to prove such a statement looks something like the following code listing.

```
def verify(
    public image,
    public ciphertext,
    private key,
    private preimage_rest,
    private message):

    preimage = concat(key, preimage_rest)

    assert digest(preimage) == image
    assert encrypt(message, key) == ciphertext
```

Sender Steve will be responsible for generating a proof by constructing a set of public and private inputs which satisfy the circuit. In the case of the circuit pseudocode in the listing above, Steve would be responsible for providing public inputs of the sequence hash $h^n_{\{S,R\}}$ and ciphertext c^n, as well as the private inputs of the shared secret $k_{\{S,R\}}$, the rest of the information required to compute the sequence hash (such as the message index n), and the cleartext of the message. Since these latter items are part of the *private* inputs, they will not be revealed in the proof that is generated.

Upon constructing this proof, Sender Steve packages the three items—sequence hash, ciphertext, and proof—all together and sends them off to the message distributor (either directly to the message repository, or to the proxy if that service is in effect).

When the message repository receives this triplet, it first verifies the proof with the well-known circuit and the provided sequence hash and ciphertext. If the proof can be successfully verified, only then does the message repository write the ciphertext into its key-value store.

If a malicious third-party, Malicious Max, intercepted the triple and attempted to replace the ciphertext payload with one of his own design, he would not be able to also generate a ZKP proving foreknowledge of the private inputs that Steve knows: in particular, the shared secret. Thus Max, regardless of whether he is merely a man-in-the-middle or the veritable operator of a proxy, is unable to “snipe” sequence hashes from genuine payloads.

4.8 Dandelion-style routing

4.9 Message notifications

4.10 Message receiving

5 Development

A reference implementation for this paper is currently in development. The following is a proposal for the development of the protocol.

5.1 Phase 1: Proof of Concept

The basic proof of concept has already been completed and shown to work in a limited capacity. The proof of concept is a simple command-line application that allows for 1:1 messaging between two parties. The proof of concept is written in Rust, and uses the *Dalek* libraries for cryptographic primitives, as well as *ChaCha20-Poly1305* for encryption.

5.2 Phase 2: Multi-Device & Group Messaging

The next phase of development will focus on multi-device messaging and group messaging. This will be implemented in the proof of concept, and will be written in Rust.

5.3 Phase 3: Library

The final phase of development will be to create a library that can be used by other applications. This will be written in Rust, and will be published to crates.io.

References

Jefferys, Kee. 2020. “The Session Protocol: What’s Changing — and Why - Session Private Messenger.” *Session*. https://getsession.org/session-protocol-explained.