TOKYO INSTITUTE OF TECHNOLOGY

MASTER THESIS

Flexible Channels: Preventing Metadata Leakage in Communication over Public Channels

Author:
Jacob LINDAHL

Student Number:
21M38053

Supervisor:
Prof. Hidehiko MASUHARA

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

in the

Programming Research Group
Department of Mathematical and Computing Science

January 22, 2025

"And so by doing just acts we become just, and by doing acts of temperance and courage we become temperate and courageous."

Aristotle, *Nicomachean Ethics*

TOKYO INSTITUTE OF TECHNOLOGY

Abstract

School of Computing
Department of Mathematical and Computing Science

Master of Science

Flexible Channels: Preventing Metadata Leakage in Communication over Public Channels

by Jacob LINDAHL

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.

Acknowledgements

The most heartfelt thanks I give to my supervisor, Prof. Hidehiko Masuhara, for his guidance, patience, and support. I am grateful for the opportunity to work with him and learn from him.

To my friends and family for their support and encouragement, I thank you.

To the members of the Programming Research Group for their feedback and support, I am grateful.

To my Lord and Savior, Jesus Christ, for His grace and mercy, I am awestruck.

Contents

List of Figures

List of Abbreviations

GWei	**G**iga **Wei** (10^9 units of the native Ethereum cryptocurrency)
HMAC	**H**ash-based **M**essage **A**uthentication **C**ode
HMAC-SHA3-512	**HMAC** using the **SHA3-512** hash function
PKI	**P**ublic **K**ey **I**nfrastructure
RPC	**R**emote **P**rocedure **C**all
SHA	**S**ecure **H**ash **A**lgorithm
TLS	**T**ransport **L**ayer **S**ecurity
URI	**U**niform **R**esource **I**dentifier
zk-SNARK	**Z**ero-**K**nowledge **S**uccinct **N**on-Interactive **Ar**gument of **K**nowledge
ZKP	**Z**ero-**K**nowledge **P**roof

For Aniyah

Chapter 1

Introduction

1.1 Motivation

The privacy of digital communications is a critical issue in the Internet age. The Snowden revelations (Greenwald, Poitras, and MacAskill, 2013) have shown that world powers are actively monitoring and collecting data on their citizens. This emphasizes the need for secure communication channels that are resistant to surveillance.

Not a few years prior, Satoshi Nakamoto had published the Bitcoin whitepaper (Nakamoto, 2008), which introduced the concept of a decentralized, trustless, and censorship-resistant currency. The potential of this technology rapidly expanded to include the trustless execution of general-purpose applications, notably in the form of smart contracts on the Ethereum blockchain (Buterin, 2014). While these platforms are widely regarded as censorship-resistant, public ledgers are eponymously antipodean to privacy in so far as their contents and execution are intrinsically revealed. Thus, an "anti-surveillance" public-ledger blockchain protocol is almost a contradiction in terms: decentralization, and consequently censorship-resistance, is achieved by "surveillance" of the network by the public.

Although there have been a few forays into privacy-first public-ledger cryptocurrency platforms (CryptoNote: Saberhagen, 2013; Zerocash: Ben Sasson et al., 2014), they have not achieved level of adoption of Bitcoin and Ethereum. Additionally, projects like Monero (CryptoNote derivative implementation) and Zcash (the active successor to Zerocash) are not general-purpose virtual machines like Ethereum, targeting only the peer-to-peer medium-of-exchange use case.

Mina Protocol (Bonneau, Meckler, and Rao, 2020) implements a general-purpose virtual machine by using recursive zk-SNARKs to compress the blockchain to a constant size proof-of-execution; the state root is included in the proof. In order to prove state transitions, therefore, the state must be obtained from a source external to the actual blockchain—a data availability layer. While the data could be sourced elsewhere (e.g. directly from the user interacting with an application), for the purposes of a messaging system, whose raison d'être data transmission from one party to another, the data availability infrastructure is not an improvement over the relatively simpler approach of a platform like Ethereum.

We attempt to combine some fundamental traditional cryptographic techniques with the modern blockchain technology to create a privacy-first messaging protocol that is resistant to surveillance and censorship.

1.2 Definitions

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.

However, privacy of these data is not sufficient to make a usable protocol. Therefore, we will also aim to fulfill 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.

Chapter 2

Proposal

2.1 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). (Diffie and Hellman, 1976)

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 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. The public nature of the message repository is critical to ensuring the efficiency of group messaging as well as ensuring that a user accessing the system from multiple different terminals is easily able to retrieve previous messages.

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.

2.2 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.

2.3 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.

2.4 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.

2.5 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 $i|A| + \text{indexof}(A, S)$. 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.

2.6 Garbage messages

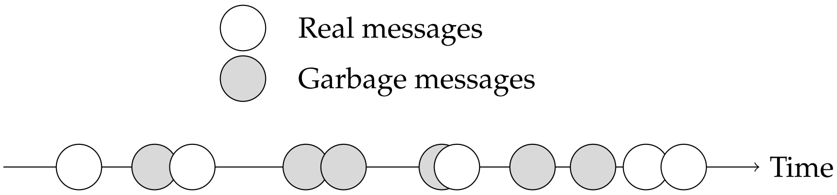
While any party without knowledge of the channel's shared secret is incapable of decrypting the messages within a channel, if a channel member's network activity is being monitored or if the message repository that a channel is using has very low traffic, it becomes possible to deduce increasingly detailed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



An observer cannot distinguish between real and garbage messages without knowledge of channel secrets.

FIGURE 2.1: Real and garbage messages as seen by outside observers.

2.7 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—many, many orders of magnitude beyond a commercial cloud provder like Amazon S3.[1]

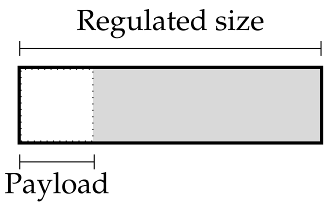


FIGURE 2.2: A smaller payload fits into a larger regulated-size envelope with padding.

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.

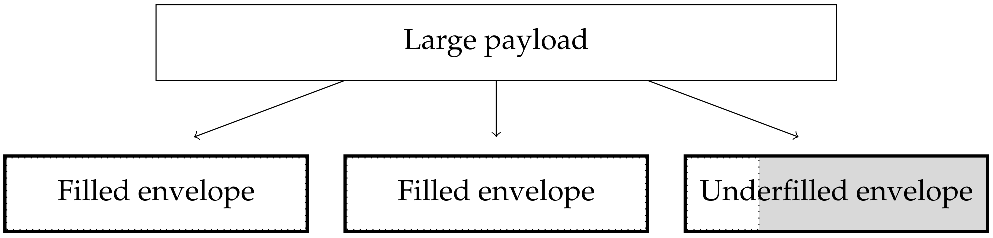


FIGURE 2.3: A large payload split into multiple chunks, padded to fit regulated-size envelopes if necessary.

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 three 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

[1] For instance, the Ethereum smart contract platform charges 20,000 units of gas for a cold 256-bit storage write (Wood, 2025, Appendix G). At current gas prices of about 30 GWei (3E-8 Ether) (etherscan.io, n.d.) and an Ether price of about \$3,400 (*Ethereum Price* n.d.), we calculate an approximate price of $3400\,\mathrm{USD/Ether} \cdot 3.1 \times 10^{-8}\,\mathrm{Ether/gas} \cdot 20000\,\mathrm{gas/slot} \cdot \frac{10^9\,\mathrm{bytes/GB}}{32\,\mathrm{bytes/slot}} = 6.6 \times 10^8\,\mathrm{USD/GB}$: over \$600 million per gigabyte.

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.

2.8 Proxies

Within the protocol itself we have already discussed many facets of metadata concealment. However, as other projects have noted in the past (ErCiccione, 2020; Dingledine, Mathewson, and Syverson, 2004), 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.

2.8.1 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. (Goldwasser, Micali, and Rackoff, 1985)

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: $(h^n_{\{S,R\}}, c^n, p^n)$—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.

2.9 Dandelion-style routing

Let us suppose there is an Observer Ollie of the network, including the message repository, the proxies, and the message sender. Ollie is able to observe the activity between each of the network participants, but not able to see the internal memory or computational activities of any of them. Because the message repository is publicly readable by anyone, Ollie is able to monitor the message repository for new messages. Therefore, Ollie can detect when a message sent by Steve appears on the message repository, and can therefore learn the sequence hash of a message known to have been sent by Steve. Continuing to monitor the network activity of all participants could further reveal to Ollie the identity of the message receiver Robin. (However, if we assume impenetrably encrypted network communication tunnels, this step may require additional techniques beyond simple surveillance.)

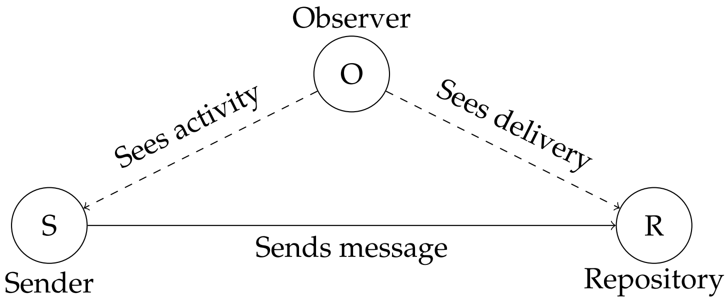


FIGURE 2.4: Network observer monitoring message sender and repository.

Therefore, the protocol is in need of a feature to break (or at least obfuscate) the network activity link between a sender distributing a message payload and that message payload appearing on the message repository.

When attempting to conceal the activity of network participants, many projects have adopted similar measures to that which we propose here: that of passing messages amongst multiple peers before delivering it to its final destination, and concealing the true origin in the process. Most prominently among such projects may be The Onion Router, more commonly known as Tor. Put simply, a user chooses a path through his peers and wraps his request in multiple layers of encryption which each subsequent peer in the path is able to decrypt. A similar proposal was made for the Bitcoin network (Venkatakrishnan, Fanti, and Viswanath, 2017), and a variation of it (Fanti et al., 2018) was adopted by Monero in 2020 (ErCiccione, 2020).

Introducing this feature to our protocol requires the increase from a single proxy service to multiple, unaffiliated proxy services. "Unaffiliated" is an important qualifier: the proxy network members (relays) must have sufficiently disparate interests and controlling influences to ensure that they are not incentivized to cooperate with each other to reveal the payload paths throughout the network. (Paganini, 2014)

This feature is not without its downsides: not only does it add nontrivial amounts of complexity to the development and maintenance of the protocol—properly implementing the Dandelion-style routing is a significant up-front cost, and maintaining a large, decentralized network of routing proxies is a significant maintenance cost—it also adds a noticeable amount of latency to the message sending process. From a privacy perspective, this is not entirely bad, since high and highly-variable latency between dispatch and receipt by the message repository can reduce the correlations in network activity timings, but it has an exclusively negative impact on the user-friendliness of the protocol.

Even assuming an honest network of relays, timing attacks are a further deanonymization technique. (DEFCONConference, 2013) Because each member of the proxy network should be publicly accessible, this allows for sidechannel deanonymization attacks: though the technique presented by Evans and Grothoff relies on a malicious JavaScript injection to "phone home," simple observation of the message sender along with the message repository over a long enough period could serve the same purpose of latency measurement.

2.10 Message notifications

The life of a message receiver is rather drab and repetitive. Receiver Robin has a simple responsibility: check the message repository to see if the next sequence hash exists. It does exist? Great, download the message! It does not exist? Go back and check again, repeating until it shows up. Robin simply plies the message repository with "message notification" requests until he receives the message (then he starts looking for the next message) or he goes offline.

Unfortunately, upon closer examination, some issues with this model become apparent. In the same vein as the previous section, wherein we addressed the potential for network activity analysis to deanonymize message senders, similar analysis can serve to deanonymize receivers. However, it is arguably slightly more risky for the receivers, because they check for the same sequence hash multiple times, meaning that a network observer need only be observing the network for long enough to witness a single message notification request. (Note that in the case of a blockchain environment, an RPC node operator could easily be this "network activity observer," for whom observation is significantly easier to perform than a traditional man-in-the-middle sniff.) The knowledge that a user searching for a particular (as of yet undelivered) message is likely to inquire about the sequence hash multiple times again in the future enables attackers to execute more targeted deanonymization attempts.

Thus, the first privacy-enhancing option available to the message receiver is to adopt the same dandelion-style network that the message senders are using, except to use it for sending message notification requests instead of for transmitting message payloads. While this would conceal the origin of the requester on a per-request basis, it is not as clean of a solution as it is in the case of the message sender: a network observer can still detect when a user (that is, for example, via activity originating from an IP address) joins the network, and if requests for a certain sequence hash are made in elevated frequency when a certain user is also active on the network, it may not be necessary to trace the exact route of a request back to its origin to establish with reasonably high confidence that this certain user is interested in this certain sequence hash.

Malicious network observations of message notification requests reveal (potentially) unused-yet-valid sequence hashes to attackers. If the zero-knowledge proof extension is not implemented, then this system requires assuming the trustworthiness of the message repository and all handlers of unencrypted (that is, by a TLS tunnel or similar) payload data between the message repository and the message receiver to not "snipe" the valid sequence hash and insert it with a malicious or garbage payload before the real message has a chance to land. But, the goal of this protocol is to minimize the trust users must have in these entities. With the zero-knowledge proof extension, the risk of sequence hash "sniping" is reduced, but the risk of correlating user activity with particular sequence hash read requests is not. Thus, we understand that the protocol is in need of a mechanism for allowing message receivers to check for the presence of a particular sequence hash without divulging to anyone on the network which sequence hash it is.

Furthermore, it is unlikely that a single receiver will only be interested in checking for the existence of a single message. Rather, a single user of the protocol is probably going to be listening for the existence of a multitude of different messages. This would impose a serious bandwidth requirement on even light users of the protocol. The reference implementation of this protocol uses a few types of control messages (messages containing instructions to the application, rather than directly human-readable plain text) which might be transmitted across designated channels. This practice in combination with a user who is only communicating with a few different users and/or groups already imposes a heavy burden of network usage on the user if they wish for their communications to be even vaguely real-time.

Therefore, while using the dandelion-style proxy network for checking for individual sequence hash notifications is certainly an improvement over direct requesting, there is a fundamental disconnect in the approach: Robin wishes the message repository to tell him whether specific sequence hashes exist without identifying those hashes.

2.10.1 Message notification filters

To solve this conundrum, we introduce a sequence hash Bloom filter. (Bloom, 1970) Upon writing a message to its storage, the message repository also inserts[2] the sequence hash into a Bloom filter. In order to maintain an acceptable false-positive rate, the current "message notification filter" is archived at regular intervals. Archived and current filters can be requested by users. This structure allows a message receiver awaiting the delivery of one or more sequence hashes to download a single payload within which he can check for the presence of any number of sequence hashes. If the user has not synchronized with the network recently, he can download archived filters to the same effect.

[2] It is not necessary that the Bloom filter is constructed immediately. It could, for example, be constructed on-demand if the cost of storage on the message repository is higher than that of compute.

In the case that the message repository generates the message notification filters on-demand, the message repository should *not* allow clients to specify a custom timeframe of sequence hashes to include, due to potential metadata leakage pertaining to when the client in question last synchronized with the network.

While this approach succeeds in concealing the sequence hashes for which a message receiver is intent on listening, it increases network load and worsens latency. Whereas a simple, direct query to the message repository would be nearly negligible in terms of network load—a request object containing merely a sequence hash and response merely a boolean—the size of a Bloom filter scales linearly with the maximum number of insertions given a false-positive rate, where the maximum number of insertions is the maximum number of messages that the message repository can process before archiving the current message notification filter. Where p is the desired false-positive rate, the number of bits required per element b is given by:

$$b = \frac{\ln\left(\frac{1}{p}\right)}{\ln^2(2)}$$

Meaning that a modest 1% false-positive rate commands about 9.6 bits per sequence hash. (Cortesi, 2010)

The apples-to-apples comparison with authenticated (albeit end-to-end encrypted) server messaging platforms is quite bleak, as they have no such scaling limitation. However, compared with other public ledger/private activity-style platforms which may require downloading and scanning the entirety of the ledger (monero.how, n.d.), the expense of ~10 bits per sequence hash is comparatively low.

2.10.2 Garbage message requests

While the Bloom filter approach does have the advantage of providing a quasi-offline means of checking whether a particular set of sequence hashes are reasonably likely to have been posted to the message repository, the inextractibility of elements contained within the Bloom filter actually prevent a mirror strategy to the "garbage messages" suggested previously.

If Receiver Robin is concerned that requesting a certain extant sequence hash may link his identity to that of Sender Steve, Robin might consider sending bogus requests for other extant sequence hashes to conceal the genuine request amongst false ones. However, if the primary means by which Robin learns about the state of sequence hashes in the message repository is via the message notification filter, there is not a reasonable way for Robin to be able to know other extant sequence hashes without constructing them himself (implying knowledge of the secrets necessary to calculate them) or plainly asking for them (implicitly confessing a lack of knowledge of the secrets necessary to calculate them).

When sending messages, a user also sends out some garbage messages as noise. Assuming a network Observer Ollie who is attempting to deanonymize users based on inter-participant network and message repository activity, Ollie can observe certain messages that never get read. Ollie, having knowledge of the protocol, realizes that these messages are likely to be garbage messages, and therefore excludes them from his network analysis after some time.

Therefore, we make the following recommendation.

When this user sends out some garbage messages, the user retains the sequence hashes[3], additionally sending out garbage message *requests* later on to imitate a receiver coming on-line and detecting the delivery of an expected sequence hash. It should be noted that the delay between sending and “receiving” a garbage message should be highly variable, with some garbage messages not getting “received” at all.

[3] Or simply the minimum information necessary to reproduce them.

Chapter 3

Implementation

A reference implementation for this paper is available.

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.

Even without enabling many of the privacy-enhancing features, the simple application struggles to approach real-time communications, with regular latencies of 1-3 seconds.

Chapter 4

Prior Art

4.1 BitMessage

BitMessage (Warren, 2012) is a Bitcoin-inspired message transfer protocol. We highlight some notable differences in the functionality of the BitMessage protocol and our proposal.

1. Sending a message over the BitMessage network requires a proof-of-work, guaranteeing a latency floor in the protocol. Our protocol only requires such proofs as the underlying infrastructure, with an optional extension for zero-knowledge proofs.
2. All users connected to the BitMessage network receive all messages. Users of our protocol know beforehand which messages are intended for them and can retrieve only those in a privacy-preserving fashion. There is an optional extension for message notifications that incurs an $O(n)$ space cost on users where n is the number of messages sent to the message repository.
3. Broadcast messages may be sent on the BitMessage network, but they are visible to all users of the network who wish to view them. Our protocol supports arbitrarily large broadcast groups with $O(1)$ sending cost, simply by sharing a new channel key.
4. Messages on the BitMessage network are deleted after a period of two days. Our protocol uses an indelible append-only ledger (i.e. blockchain) from which messages cannot be erased.

4.2 Signal Double-Ratchet

Considered by many to be the gold standard in modern encrypted messaging, the Signal Double-Ratchet protocol (Perrin and Marlinspike, 2016) implements a foreboding trifecta of privacy properties: resilience, forward security, and break-in security. The sequence hashes from our protocol exhibit the first of these properties.

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.).

Chapter 5

Conclusion

We present Flexible Channels, an experimental foray into encrypted messaging over public, append-only message repositories such as blockchains. The key insights of this research are the channel and sequence hash constructions which allow for privacy-friendly messaging across public message repositories using simple, traditional cryptography.

We explore a number of benefits of this infrastructure, including easy cross-device message synchronization, uncensorability, and $O(1)$ broadcast messaging.

However, the protocol as described and implemented immediately suffers severe performance impairments that preclude its adoption. Regardless of its performance, the infrastructure does not provide significant improvements over existing practices of end-to-end encryption through authenticated servers, and the privacy implications of permanently etching ostensibly secret messaging history onto an inextirpable public record encrypted merely by means which are considered secure by the standards of today cannot be understated. The protocol as described does not enforce any sort of forward secrecy, although it is flexible enough to allow e.g. key rotation within channels, so the design does not preclude the implementation of this privacy-critical feature.

Therefore, while we cannot yet recommend using this protocol in non-experimental contexts, we are pleased to submit this novel combination of techniques for critique and edification.

Possible future applications of similar technology could become feasible in environments with a forcing need for public auditability juxtaposed with privacy (e.g. government operations or highly-regulated industries).

Bibliography

Ben Sasson, Eli et al. (2014). “Zerocash: Decentralized Anonymous Payments from Bitcoin”. In: *2014 IEEE Symposium on Security and Privacy*, pp. 459–474. DOI: 10.1109/SP.2014.36.

Bloom, Burton H. (July 1970). “Space/time trade-offs in hash coding with allowable errors”. In: *Commun. ACM* 13.7, 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: https://doi.org/10.1145/362686.362692.

Bonneau, Joseph, Izaak Meckler, and Vanishree Rao (Mar. 2020). “Mina: Decentralized Cryptocurrency at Scale”. en. In: URL: https://minaprotocol.com/wp-content/uploads/technicalWhitepaper.pdf.

Buterin, Vitalik (Dec. 2014). “Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform.” In: URL: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

Cortesi, Aldo (Aug. 2010). *3 Rules of thumb for Bloom Filters*. URL: https://corte.si/posts/code/bloom-filter-rules-of-thumb/ (visited on 01/17/2025).

DEFCONConference (Nov. 2013). *DEF CON 16 - Nathan Evans & Christian Grothoff: de-Tor-iorate Anonymity*. URL: https://www.youtube.com/watch?v=zpin0TtawpE (visited on 01/15/2025).

Diffie, W. and M. Hellman (Nov. 1976). “New directions in cryptography”. In: *IEEE Transactions on Information Theory* 22.6, pp. 644–654. ISSN: 0018-9448, 1557-9654. DOI: 10.1109/TIT.1976.1055638. URL: https://ieeexplore.ieee.org/document/1055638/.

Dingledine, Roger, Nick Mathewson, and Paul Syverson (Aug. 2004). “Tor: The Second-Generation Onion Router”. In: *13th USENIX Security Symposium (USENIX Security 04)*. San Diego, CA: USENIX Association. URL: https://www.usenix.org/conference/13th-usenix-security-symposium/tor-second-generation-onion-router.

ErCiccione (Apr. 2020). *Blog: Another privacy-enhancing technology added to Monero: Dandelion++*. en. URL: https://www.getmonero.org/2020/04/18/dandelion-implemented.html (visited on 01/14/2025).

Ethereum Price (n.d.). *Ethereum Price: ETH Live Price Chart, Market Cap & News Today*. en. URL: https://www.coingecko.com/en/coins/ethereum (visited on 01/19/2025).

etherscan.io (n.d.). *Ethereum Gas Tracker | Etherscan*. en. URL: https://etherscan.io/gastracker (visited on 01/19/2025).

Fanti, Giulia et al. (June 2018). “Dandelion++: Lightweight Cryptocurrency Networking with Formal Anonymity Guarantees”. In: *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, pp. 1–35. DOI: 10.1145/3224424.

Goldwasser, S, S Micali, and C Rackoff (1985). “The knowledge complexity of interactive proof-systems”. In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*. STOC ’85. Providence, Rhode Island, USA: Association for Computing Machinery, 291–304. ISBN: 0897911512. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178.

Greenwald, Glenn, Laura Poitras, and Ewen MacAskill (Sept. 2013). "NSA shares raw intelligence including Americans' data with Israel". en-GB. In: *The Guardian*. ISSN: 0261-3077. URL: https://www.theguardian.com/world/2013/sep/11/nsa-americans-personal-data-israel-documents (visited on 01/22/2025).

Jefferys, Kee (Dec. 2020). *The Session Protocol: What's changing — and why - Session Private Messenger*. en. URL: https://getsession.org/session-protocol-explained.

monero.how (n.d.). *A low-level explanation of the mechanics of Monero vs Bitcoin in plain English*. URL: https://www.monero.how/how-does-monero-work-details-in-plain-english (visited on 05/02/2024).

Nakamoto, Satoshi (Oct. 2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*. URL: http://www.bitcoin.org/bitcoin.pdf.

Paganini, Pierluigi (July 2014). *Attacks to compromise TOR Network and De-Anonymize users*. en-US. URL: https://securityaffairs.com/27193/hacking/attacks-against-tor-network.html (visited on 01/15/2025).

Perrin, Trevor and Moxie Marlinspike (Nov. 2016). "The Double Ratchet Algorithm". en. In: URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

Saberhagen, Nicolas van (2013). "CryptoNote v 2.0". In: URL: https://api.semanticscholar.org/CorpusID:2711472.

Venkatakrishnan, Shaileshh, Giulia Fanti, and Pramod Viswanath (Jan. 2017). "Dandelion: Redesigning the Bitcoin Network for Anonymity". In: *ACM SIGMETRICS Performance Evaluation Review* 44. DOI: 10.1145/3143314.3078528.

Warren, Jonathan (Nov. 2012). *Bitmessage: A Peer-to-Peer Message Authentication and Delivery System*. en. URL: http://bitmessage.org/bitmessage.pdf.

Wood, Dr Gavin (Jan. 2025). "ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER". en. In: URL: https://ethereum.github.io/yellowpaper/paper.pdf.