Breaking Bridgefy, again: Adopting libsignal is not enough

Martin R. Albrecht
Information Security Group, Royal Holloway, University of London

Raphael Eikenberg
Applied Cryptography Group, ETH Zurich

Kenneth G. Paterson
Applied Cryptography Group, ETH Zurich

Abstract

Bridgefy is a messaging application that uses Bluetooth-based mesh networking. Its developers and others have advertised it for use in areas witnessing large-scale protests involving confrontations between protesters and state agents. In August 2020, a security analysis reported severe vulnerabilities that invalidated Bridgefy's claims of confidentiality, authentication, and resilience. In response, the developers adopted the Signal protocol and then continued to advertise their application as being suitable for use by higher-risk users.

In this work, we analyse the security of the revised Bridgefy messenger and SDK and invalidate its security claims. One attack (targeting the messenger) enables an adversary to compromise the confidentiality of private messages by exploiting a time-of-check to time-of-use (TOCTOU) issue, side-stepping Signal's guarantees. The other attack (targeting the SDK) allows an adversary to recover broadcast messages without knowing the network-wide shared encryption key.

We also found that the changes deployed in response to the August 2020 analysis failed to remedy the previously reported vulnerabilities. In particular, we show that (i) the protocol persisted to be susceptible to an active attacker-in-the-middle, (ii) an adversary continued to be able to impersonate other users in the broadcast channel of the Bridgefy messenger, (iii) the DoS attack using a decompression bomb was still applicable, albeit in a limited form, and that (iv) the privacy issues of Bridgefy remained largely unresolved.

1 Introduction

The messaging space has witnessed a rapid transformation since the Snowden revelations in 2013. Since then, almost all major messaging services enabled at least some form of (optional) end-to-end encryption. In particular, many such offerings, e.g. the Signal app itself, WhatsApp, Google's Allo, Skype – adopted the Signal protocol [18]. This move, to strengthen privacy in light of reports of state surveillance, expresses a general ethos in the messaging app community to provide tools that can be relied upon and enable higher-risk users, such as human rights activists [23]. However, the same work [23] and follow up works [2, 37] also indicate a disconnect between what messaging app designers design for and the needs of higher-risk users.

A visible expression of this disconnect is Bridgefy, a mobile application and software development kit (SDK) that provides communication capabilities over Bluetooth. It allows users to form a mesh network to exchange messages without requiring a connection to the Internet. Its primary target applications are large events such as sports events where existing Internet infrastructure may not be able to cope with demand. Yet, its developers and others both report on the uptake of the application and also actively promote the application for use in protests and other situations of social unrest, where mobile telecommunications and Internet connections may be unreliable: during the 2019 Hong Kong protests [41, 57], during protests in Iran, Lebanon and Zimbabwe [32, 59], for Black Lives Matter protests in the US [58], after the military coup in Myanmar [13, 63, 64], during farmers' protests in India [62], for anti-lockdown protests in the US [60], after the Taliban retook Afghanistan [67]. According to the developers, the mobile application has been downloaded more than 6.5 million times [61], often from areas witnessing confrontations between protesters and agents of the state.

While the actual adoption of Bridgefy by protesters and activists seems to be somewhat limited [2], the spikes in downloads from areas witnessing conflicts suggest a need or desire for robust peer-to-peer offline communication; a need not catered to by more mainstream messaging platforms.[1]

On the other hand, the security track record of the Bridgefy app – so heavily advertised to higher-risk users – is less than stellar. In particular, in August 2020, a security analysis [1] demonstrated: (i) An adversary could track Bridgefy users, and produce a social graph of the mesh network, (ii) messages of users could be spoofed due to the lack of authentication mechanisms, (iii) an active attacker in the middle could

[1] Bridgefy is the market leader in the space of end-user mesh messaging [1].

impersonate two users to each other and eavesdrop on the communication, (iv) private messages were susceptible to a padding oracle attack, and (v) a carefully crafted message could either take down the entire network or prevent two particular users from communicating. In response, in October 2020, Bridgefy announced an overhaul of their security architecture [10] to address all these findings. The key technical change implemented by Bridgefy was the adoption of the Signal protocol [27]. In addition, all traffic – including metadata – is now also encrypted with a network-wide symmetric key using AES in ECB mode. Since then, no public independent security assessment of the Bridgefy application has been conducted, but Bridgefy started advertising their application again for higher-risk scenarios [11].

It is thus a natural and pressing question to ask whether these upgrades – adopting the Signal protocol and adding a layer of deterministic encryption – succeed in establishing a secure communication system. Indeed, we may ask more generally if 'we use Signal' is a sufficient, well, signal to indicate the security of applications produced by development teams inexperienced in defensive coding.

Contributions. In this work, we report severe, practically exploitable vulnerabilities in the Bridgefy messenger in version 3.1.3 and the SDK in version 2.0.2, i.e. versions featuring the above mentioned security enhancements.

In Section 3, we give an overview of the inner workings of Bridgefy in version 3.1.3. We provide an outline of the application architecture and the Bridgefy protocol.

For completeness, in Section 4 we first re-evaluate the vulnerabilities previously reported in [1] and find that they remain mostly entirely or insufficiently fixed. Specifically, for Bridgefy 3.1.3 we show: (1) The protocol persisted to be susceptible to an attacker in the middle. While the attack is now limited to the first exchange between a pair of users – it abuses the 'trust on first use' (TOFU) assumption – we note that Bridgefy offers users no option to verify the public keys of their contacts. (2) Broadcast messages continued to be unauthenticated; an adversary can exploit this to mount impersonation attacks. (3) The Denial of Service (DoS) attack remained applicable, albeit in a limited form. (4) Bridgefy users could still be tracked.

We then present two new attacks on Bridgefy. Our attack in Section 5 breaks the confidentiality of Bridgefy's Signal-secured private chats by associating an attacker's public key with the session between two targets. It exploits a difference in time that arises between queuing a message and fetching the encryption key and, as such, is a time-of-check to time-of-use (TOCTOU) vulnerability. We stress that our attack does not threaten the security of the Signal protocol itself but exclusively the way that libsignal is integrated into Bridgefy.

Our attack in Section 6 gives an adversary the ability to recover broadcast messages from a small set of possible plaintexts in the setting where the network-wide shared key is unknown to the adversary. While in case of the Bridgefy application itself we may assume the attacker knows this key, this assumption would not hold for a third party application utilising the Bridgefy SDK. It works because compression precedes encryption of packets. While it is well-known that this choice can leak some information about plaintexts [40, 47], it is non-trivial to exploit the leakage in the context of Bridgefy and to perform plaintext recovery.

In Section 7, we discuss our results. In particular, we discuss how our work highlights that secure offline messaging is still unsolved in practice.

Disclosure. We notified the Bridgefy developers about our findings from Section 4 and the attack from Section 5 on 2021-05-21. The developers confirmed receipt some days later and described their plans to remediate the vulnerabilities. On 2021-07-21, the developers informed us they would not publicly disclose the problems we reported, explaining they feared putting their users' safety at risk if they did. However, they promised to remove the term 'end-to-end' from all of their social media and blog posts.

In version 3.1.7 of the Bridgefy messenger, released on 2021-08-14, our exploit for the TOCTOU attack stopped working. Up until this point in time, the attack still worked as described here. We found that Bridgefy also deployed changes regarding the DoS attack from Section 4.3. The Bridgefy SDK was not updated at all throughout the course of our research, and continues to be vulnerable to the attacks described herein. However, a note was added to its website some time in 2021 indicating that the SDK was deprecated, contained 'notable security vulnerabilities' and was due to be replaced by December 2021 [12].

We disclosed the attacks on Bridgefy's broadcast encryption mechanism on 2021-09-07. On 2021-09-09, the developers informed us that they were aware of the vulnerability and were actively working on fixing it. They did not inform us of how they planned to do so.

We asked the developers to comment on the remediation progress on 2022-02-04. At the time of finalising this paper, two weeks later, the state of the remediation remained unclear.

2 Preliminaries

In this section, we introduce the concepts and technologies that the following sections build on. We also give an overview of related work in the broader context.

Terminology. We first briefly introduce non-standard terms we use consistently in this paper.

• *Messages and packets.* When a user types a string s and sends it to another user over the mesh network, the string passes by multiple nodes, i.e. it is transmitted over multiple hops. Obviously, s does not change over these hops but the bytes transmitted between the nodes on the way change because the metadata of the *packets* differ. In other words, the

user triggers a single message, which propagates in the network with the help of multiple packets.

• *Payload content.* When a user types a string s and sends it to another user, we call s the *payload content* of the message (and of the packets). This is to avoid confusion with the terminology used by Bridgefy: a *payload* in Bridgefy is a map of key-value pairs within a packet. The details of packet layouts are discussed in Section 3.5.

• *Simulation and attack samples.* In the broadcast message recovery attack, we have a simulation phase and an attack phase. Both require us to gather packet lengths to form a *sample*. The respective outputs will be a *simulation sample* and an *attack sample*.

Bluetooth Low Energy (BLE). Bluetooth Low Energy is a widely adopted wireless technology used in mobile and Internet of Things (IoT) devices. Ryan [48] conducted an early analysis of BLE security, demonstrating packet injection and breaking the key exchange as part of the encryption. Sivakumaran and Blasco [53] showed that pairing protected BLE data needs to be secured on the application layer in Android to prevent co-located applications on the device from accessing it. Wu, Nan, Kumar, Tian, Bianchi, Payer, and Xu [68] found a weakness in the BLE specification that enabled an attacker to impersonate a device to another. Zhang, Weng, Dey, Jin, Lin, and Fu [69] reported practically exploitable downgrade attacks on BLE.

Mesh Networks. A mesh network is based on a network topology where devices connect without following a hierarchical structure [17]: 'In mesh topologies, network nodes are directly and dynamically connected in a non-hierarchical way [...]. Moreover, mesh networks do not require an infrastructure, since they dynamically self-organise and configure themselves.' A mesh topology is especially useful when the goal is to build a decentralised network: devices route incoming traffic to their neighbours, such that each packet eventually reaches its destination. Popular protocols that use a mesh topology are Bluetooth Mesh [50], Zigbee [3], and Thread [36]. Note that Bluetooth Mesh is a dedicated technology that is not to be confused with mesh networks where the links are normal Bluetooth LE connections.

Signal and libsignal. Signal [52] is a messaging application that enables end-to-end encrypted communication. Its security guarantees stem from the Signal cryptographic protocol, which was developed progressively as part of the Signal application. The protocol was subject to an extensive study by Cohn-Gordon, Cremers, Dowling, Garratt, and Stebila [18], who analysed the key agreement and the ratcheting mechanism of Signal. Their analysis revealed no significant flaws in its design.

The Signal protocol is available as an official implementation in Java called `libsignal-protocol-java` [51] under the GNU General Public License v3.0. The library can be used to provide end-to-end encrypted communication for applications other than Signal.

In the interface of the library, endpoints are identified by a `SignalProtocolAddress`. This type is a combination of a `name` that identifies the user and a `deviceId` that is unique for each device a user owns. Before two endpoints can communicate, one party needs to retrieve a 'prekey bundle' (PKB) of the other and use it to send an initial message. Here we may assume that the PKB acts like a public key: it contains all information to establish a secure session between the two parties, but it needs to be authentic. If an adversary was able to change the PKB for their own, the session would not be secure. In the Signal messenger, the server is hence trusted until the two communicating parties manually verify the authenticity of their session.

Time-of-Check to Time-of-Use (TOCTOU). Time-of-Check to Time-of-Use vulnerabilities [65, pg. 157] exploit a change in state between when a certain property is checked and used [56]. Bishop and Dilger [6] were among the first to describe this class of vulnerabilities and studied them in the context of file systems.

MessagePack. MessagePack [24] is a data format for object serialisation, similar to JSON, YAML, and TOML [8, 22, 44]. It supports various primitive types like integers, booleans, floats, strings, arrays, and maps. A key difference between MessagePack and its counterparts is that the format is binary, allowing for more compactness.

The specification of MessagePack is available on GitHub [30]. In general, an object is converted by sequentially lining up the respective *formats* for all values of an object. For example, given an object made of two boolean values, the serialised form is a concatenation of the formats for these two boolean values.

The format of a value is defined in the specification. For instance, the boolean values `false` and `true` convert to the fixed bytes `0xc2` and `0xc3` respectively. Values with variable length convert into formats that contain not just the value but also their size. A string with a length of up to 31 bytes converts into a leading byte b, followed by the ASCII encoding of the string. b is a composition of form `101XXXXX`, where the placeholder `XXXXX` refers to the size of the string. For example, the string 'id' converts to the bytes `0xa2 0x69 0x64`. Here $b = \texttt{a2}_{16} = \mathbf{101}00010_2$, followed by the ASCII representations of 'i' and 'd'.

Maps – which map from keys to values – work similar to strings. They also start with a byte b, for maps with up to 15 elements of form `1000XXXX`, where `XXXX` now refers to the number n of key-value pairs. b is followed by $2n$ formats: odd elements are keys, and even elements are the value for the preceding key.

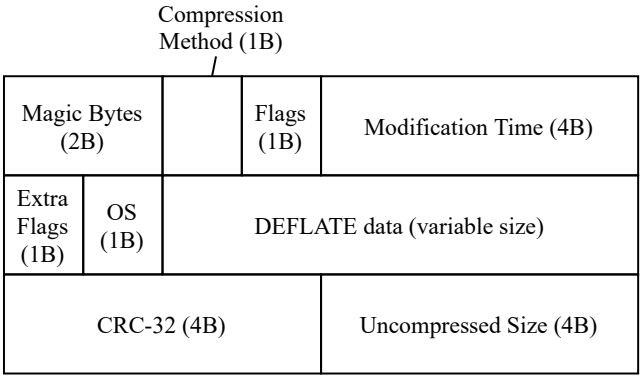


Figure 1: The file format of gzip. The 'Flags' field has influence over the structure of the file format after the 'Operating System' field. Here we assume that no flags are set.

Compression in Cryptography. The use of compression in combination with an encryption scheme was shown to be able to affect the security of that system through a side-channel in Kelsey [40]. Later Rizzo and Duong [47] showed with the CRIME attack that an attacker could recover secret web cookies based on a chosen-plaintext attack together with information leakage caused by compression in SPDY and TLS. Similarly, the BREACH attack, reported by Prado, Harris, and Gluck [43], demonstrated that the idea of CRIME was also applicable to compression in HTTP, which was not considered in the efforts to mitigate CRIME. Vanhoef and Van Goethem [66] showed with the HEIST attack that despite all efforts to mitigate CRIME and BREACH, an attacker-in-the-middle could still derive the length of the plaintext of a response and use the leakage of the compression to mount a plaintext recovery attack. Around the same time, Garman, Green, Kaptchuk, Miers, and Rushanan [26] reported an attack against Apple iMessage that exploits certain properties of DEFLATE compressed data.

Gzip [20] is a file format for lossless compressed data. In essence, it wraps DEFLATE [19] compressed data and attaches metadata fields around it. Figure 1 illustrates the high-level file format of gzip. The first two bytes are of the fixed values `0x1f` and `0x8b`. Then follows a field to indicate the algorithm used for compression: since only DEFLATE is defined in gzip, this is always a byte of value `0x08`. The other values of the header are commonly set to zero. The trailer consists of a CRC-32 value computed over the uncompressed data and the length of that data.

DEFLATE is based on LZ77 [70] and Huffman coding [38]. Overall, the algorithm replaces any repeating block of data with a reference to a previous occurrence. At the same time, it ranks bytes and references by occurrence and assigns them a code word accordingly. The compression effect, therefore, comes down to data deduplication at both byte and bit level.

The DEFLATE-compressed data can consist of several blocks. Each block starts with a header: the first three bits determine the type of the block and indicate if the block is the final block of the compressed data. Depending on the type, blocks can either be uncompressed or use fixed or dynamic Huffman codes. In this work, we focus on blocks with dynamic Huffman codes, for which the block then contains information about the Huffman table used for compression. The rest of the data in the block is the actual compressed data in the form of Huffman code words.

How this data is encoded is well-described in RFC 1951 [19]: '[...] encoded data blocks in the "deflate" format consist of sequences of symbols drawn from three conceptually distinct alphabets: either literal bytes, from the alphabet of byte values (0..255), or <length, backward distance> pairs, where the length is drawn from (3..258) and the distance is drawn from (1..32,768). In fact, the literal and length alphabets are merged into a single alphabet (0..285), where values 0..255 represent literal bytes, the value 256 indicates end-of-block, and values 257..285 represent length codes (possibly in conjunction with extra bits following the symbol code) [...]'

As described above, a block always ends with the code word that represents the value 256. Since code words are bits of variable length, a block is not necessarily byte-aligned. Still, the gzip trailer must be byte-aligned, which is why the block is padded to the next full byte.

Maximum Likelihood Estimation (MLE). Maximum Likelihood Estimation is a method to derive the parameter that is most likely to underlie the probability distribution of observed data. MLE has been used, e.g. by Bricout, Murphy, Paterson, and van der Merwe [9] and Garman, Paterson, and van der Merwe [25].

We give a brief introduction to MLE. We are given a random sample $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ where $x_i \sim X_i$ for some random variable X_i. We assume that the joint probability distribution of $\mathbf{x}$ depends on the unknown parameter $\theta \in \Theta$. We can make a 'best guess' $\hat{\theta}$ for θ based on the observation $\mathbf{x}$:

$$\hat{\theta} = \arg\max_{\theta \in \Theta} \mathcal{L}(\theta|\mathbf{x})$$

where the likelihood $\mathcal{L}(\theta|\mathbf{x})$ is defined as

$$\mathcal{L}(\theta|\mathbf{x}) := \Pr(\mathbf{x}|\theta) = \Pr(x_1 x_2 \ldots x_n|\theta).$$

Assuming that X_i, X_j with $1 \leq i, j \leq n$ and $i \neq j$ are pairwise independent, we can simplify this expression to

$$\mathcal{L}(\theta|\mathbf{x}) = \Pr(\mathbf{x}|\theta) = \prod_{i=1}^{n} \Pr(x_i|\theta).$$

Equivalently, we write:

$$\hat{\theta} = \arg\max_{\theta \in \Theta} \log \mathcal{L}(\theta|\mathbf{x}) = \arg\max_{\theta \in \Theta} \sum_{i=1}^{n} \log \Pr(x_i|\theta).$$

Instead of determining only one best guess, we can create a sequence $\hat{\Theta} = (\hat{\theta}_1, \hat{\theta}_2, \ldots, \hat{\theta}_n)$ of *candidates*, ordered by decreasing (log) likelihood.

2.1 Methodology

We analysed the Android application in version 3.1.3 and the SDK in version 2.0.2, dated 2021-04-27 and 2021-02-09. Since both the messaging application and the SDK are closed-source software, it was necessary to reverse engineer them. We retrieved the APK by installing the Bridgefy messenger from Google Play [33] on our Android phone and fetching the file via adb [45]. While the SDK is compiled into the messaging application, it is also available separately in a public Maven repository [15] as an AAR file.

Static Analysis. We decompiled Bridgefy to reconstruct Java source code for better readability. The APK file was directly decompiled using Jadx [54], but also converted into a JAR file using enjarify [34] for further processing. The AAR file was extracted to retrieve a JAR file. Both JAR files were then decompiled to Java source, leveraging multiple Java decompilers: CFR [5], Fernflower [39], Krakatau [35], and Procyon [55]. While the output was obfuscated, Bridgefy's code sometimes references class and method names in debug messages.

Dynamic Analysis. After manually inspecting the Java code, we instrumented the Bridgefy messenger with Frida [46] and objection [49]. This allowed us to hook into existing functions of the app, and thereby monitor method calls and change method behaviour. In particular, we could observe packets as they were being encrypted and decrypted. As part of this work, we produced several Frida scripts to extract information and modify the behaviour of Bridgefy. The source code of these scripts will be published in our public repository.[2]

Simulation. Our simulations in Section 6 were performed on several machines. In total, these were equipped with four Intel(R) Xeon(R) Gold 6252 CPUs, two Intel(R) Xeon(R) Gold 6138 CPUs, two Intel(R) Xeon(R) E5-2690 v4 CPUs, and two Intel(R) Xeon(R) E5-2667 v2 CPUs.

3 Bridgefy Architecture

In this section, we explain how the protocol underlying the Bridgefy application and SDK works, with a focus on its confidentiality and authenticity mechanisms.

3.1 Overview

Users that run an application using the Bridgefy SDK (such as the Bridgefy application itself) become part of a Bluetooth network that relays messages, i.e. they become a *peer* of the mesh network.

[2] https://github.com/eikendev/breaking-bridgefy-again

Bridgefy supports Bluetooth Low Energy (BLE) and Classic Bluetooth, with BLE being the default mode of operation. Under certain conditions, messages can also be transmitted over the Internet, however, if a device is offline, it will naturally only communicate over Bluetooth. In this work, we focus exclusively on BLE-based communication.

Messages can either be sent publicly to everyone nearby or to a specific user. Public messages are sent in the broadcast room, while private messages are sent in a private chat. A private chat can only be instantiated with users whose device has previously been within Bluetooth reach. This is done by clicking on the name of another user in the broadcast room.

For private chats, the application will indicate visually when the other user is within Bluetooth reach. If this is the case, then the messages to that user will not be relayed over the mesh network, but sent directly to that user.

Users are identified by a universally unique identifier (UUID) of 128 bit called *userId*. This UUID is randomly generated on each device when the application is launched for the first time. Users must also pick a *display name* when they install the app, however, it is not unique and can be arbitrarily chosen. When a new broadcast message is received, the display name of the sender is displayed along with the message.

3.2 Software Components

As mentioned above, the Bridgefy application makes use of Bridgefy's SDK. While the application is responsible for the user interface and chat management, the SDK provides the necessary mechanics to (i) establish trust between devices, (ii) encrypt and decrypt packets, and to (iii) transmit packets via the Bluetooth functionality offered by the underlying operating system.

For the SDK to work, it needs to be initialised, which happens when the application starts. This process and the general use of the SDK is documented in a GitHub repository together with official sample applications [16]. Additionally, a description of all exposed functionality is available in the official SDK documentation [14].

To summarise, the application calls `Bridgefy.initialize()` of the SDK with a registration callback and an API key. The SDK will then validate the API key and notify the application of the result via the callback. On success, the application next calls `Bridgefy.start()` with two different callbacks:

- a **message listener** that is called when a new message is received, and
- a **state listener** that is called when a connection with a nearby peer is established or closed.

Finally, if the application wants to send a message, it calls `Bridgefy.sendMessage()` or `Bridgefy.sendBroadcastMessage()`. Depending on

its use case, the application can set a profile for the SDK to control the lifetime of messages in the network.

The SDK outsources some cryptography-related operations to libsignal. When instantiating a `SignalProtocolAddress`, Bridgefy sets the `deviceId` to 0 while using a peer's userId in the addresses `name` field. libsignal maintains state for all established sessions in a `SignalProtocolStore`. When a new PKB is received from a peer, Bridgefy instantiates a `SessionBuilder` which is supplied with the protocol store and the peer's protocol address. A new session is then created by passing the PKB to `SessionBuilder.process()`. When the SDK needs to encrypt data using Signal for a particular peer, it instantiates a `SessionCipher` and supplies it with the protocol store and the peer's protocol address. The data is then passed to `SessionCipher.encrypt()`.

3.3 Packet Types

Users can decide between sending broadcast messages and private messages. However, since private messages can either be sent directly to the other peer or over the mesh network, there are three different settings to consider:

- A **broadcast packet** propagates a broadcast message from one peer to multiple other peers over the mesh network.
- A **multi-hop packet** transmits a private message from one peer to another over the mesh network.
- A **one-to-one packet** transmits a private message from one peer to another directly. Note that this setting is only applicable when the two peers are within Bluetooth reach.

On the network layer, Bridgefy associates only two different packet types with these settings: those that are routed through the mesh network, and those that are sent directly. The former packets are referenced as type `ForwardMessage`, while the latter are of type `BleEntityContent`.

3.4 Handshake

When two devices get physically close enough to establish a Bluetooth connection, they perform a handshake (assuming that they have not performed a handshake previously). This process is handled by the SDK, meaning it is not visible to the application.

In the handshake, each party generates a PKB and sends it to the other party. Based on the exchanged PKBs, a Signal session is established, enabling the parties to encrypt and authenticate packets.

Assuming Alice A and Bob B come within range of one another for the first time, the handshake proceeds as follows:

$$A \to B: \quad \texttt{ResponseTypeGeneral}(\text{userId}_A) \tag{1}$$
$$B \to A: \quad \texttt{ResponseTypeGeneral}(\text{userId}_B) \tag{2}$$
$$A \to B: \quad \texttt{ResponseTypeKey}(\text{PKB}_A) \tag{3}$$
$$B \to A: \quad \texttt{ResponseTypeKey}(\text{PKB}_B) \tag{4}$$

Here, userId_A denotes the userId of peer A and PKB_A denotes the PKB generated by A. After (1), B checks if any Signal session has already been established for userId_A, and aborts the handshake if this is the case. Peer A may also abort the handshake after (2).

Note that we have made some simplifications here that are not relevant to our analysis. For example, the packets also contain CRC checksums and version information. Further, all four packets of the handshake are wrapped in a `BleHandshake` packet, which itself is wrapped in a `BleEntity` packet.

The handshake is not performed over the mesh network, but only over a direct Bluetooth connection. As a result, only peers that have previously met can later exchange messages privately over the mesh network.

Because no further authentication is involved, the handshake follows the trust on first use (TOFU) principle: in (3) and (4), the parties implicitly trust the PKB they receive. In contrast to messengers like Signal, users cannot verify the keys of peers manually, as Bridgefy's user interface offers no way to do so.

3.5 Packet Encoding

On the lowest layer, Bridgefy encapsulates all packets into the type `BleEntity`. Its `et` field (presumably for 'entity type') indicates the type of packet it contains.

The type `ForwardPacket` represents multi-hop packets and broadcast packets. For efficiency, multiple objects of type `ForwardPacket` are bundled into a packet of type `ForwardTransaction` on the network layer. Going forward and for ease of exposure, we will assume that a `ForwardTransaction` contains only a single `ForwardPacket`, as would be the case in a low-traffic mesh network.

The type `ForwardPacket` features fields necessary to route the packet through the mesh network. Among other things, it contains a time to live (TTL) field named `hops`. This field is a single byte value that decrements whenever the packet is forwarded by a node. The purpose of the field is to prevent packets from circulating in the mesh network indefinitely: once the value reaches 0, the packet is discarded.

The `track` field is a list that contains the CRC-32 sums of userIds that have been involved in the delivery of a packet. More precisely, its length is limited to the last n nodes, where n varies depending on the profile of the connection.The field appears unused otherwise.

Both the `ForwardPacket` and the `ForwardTransaction` have their own `sender` field. The former holds the userId of the *message* sender, while the latter holds the userId of the *packet* sender. The message sender originally typed the message into the chat window, whereas the packet sender was the most recent peer to relay the message. The two fields are equal exactly at the very first hop of the message.

While the overall structure of broadcast and multi-hop packets is similar, there are important differences:

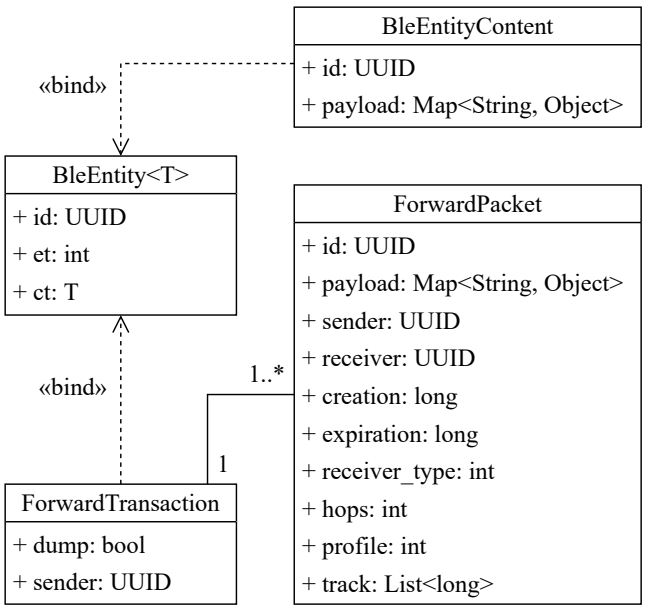


Figure 2: A `ForwardPacket` is always encapsulated in a `ForwardTransaction`, which itself is encapsulated in a `BleEntity`. A `BleEntityContent` is also encapsulated in a `BleEntity`.

- The `receiver_type` field is used to differentiate broadcast packets from multi-hop packets: the value 1 indicates a broadcast packet, and the value 0 a multi-hop packet.
- Since broadcast packets do not have a designated receiver, they do not contain a populated `receiver` field.
- In multi-hop packets, the payload entry `nm` (presumably for 'name') refers to the name of the receiver, whereas in broadcast packets, it refers to that of the sender.
- While a `ForwardPacket` containing a broadcast packet is serialised and encrypted as a whole, it is handled differently for multi-hop packets: the `payload` field is removed from the `BleEntity` and processed separately. The remaining data in the `BleEntity` is considered metadata and encrypted in another way than the payload.

Because one-to-one packets are not carried over the mesh network, they do not carry routing information. For this reason, they are encoded in the more concise packet type `BleEntityContent`. Figure 2 illustrates the relations between all discussed types in a UML diagram.

The message typed by a user is referred to as 'payload content'. In a `ForwardPacket`, the payload content is stored as a string under the key `ct` in the `payload`. For one-to-one packets, it is encoded in the `payload` map of the `BleEntityContent` respectively.

3.6 Packet Encryption

Before a `BleEntity` is sent to another peer, it is (i) serialised using MessagePack, (ii) compressed using gzip, and then (iii) encrypted. The encryption step can involve Signal encryption in combination with AES in ECB mode with PKCS#7 padding, or AES-ECB with PKCS#7 padding only. Table 1 summarises which encryption method is used for the different packet types.

Data Category	*Metadata*	*Payload*
`BleHandshake`	AES-ECB	AES-ECB
`BleEntityContent`	AES-ECB	libsignal
`ForwardTransaction`	AES-ECB	libsignal

Table 1: Encryption of packets in Bridgefy by data category and packet type.

For AES-ECB, a symmetric key is shared between all peers in the network. In the case of the Bridgefy messenger, an adversary can easily obtain this symmetric key because the application is public. More generally, depending on the nature of the threats considered – inside and outside – the shared symmetric key may be considered known or unknown to the adversary.

Signal encryption is only used for the `payload` field of multi-hop and one-to-one packets. Broadcast packets and the metadata of any other packets are encrypted with AES-ECB. In Section 5 we will ignore this layer of encryption as the shared key of the Bridgefy messenger must be assumed as known to the adversary: it can be recovered using dynamic instrumentation. We then treat it in Section 6 and Appendix A.

Remark 1 *Previous versions of Bridgefy implemented a custom scheme based on RSA in place of the Signal protocol. With Bridgefy's adoption of the Signal protocol in place of RSA, the padding oracle attack reported in [1] is no longer applicable.*

3.7 Devices and Sessions

In the Bridgefy SDK, the `DeviceManager` is responsible for maintaining a list of nearby Bluetooth devices. Each device is associated with a session, which itself is managed by a `SessionManager`. Since the co-existence of devices and sessions appears arbitrary, we will in the following refer to sessions only.

During the handshake, a userId is received from the other peer and saved in the corresponding session. When the SDK is instructed to send a message to a userId, it looks for a session associated with that userId. The message is then queued in the `TransactionManager` together with the session. Once Android requests more Bluetooth data to send, the SDK pops the queued message, encrypts it for the userId saved in the session, and dispatches it.

When a Bluetooth packet is received, the SDK looks up the correct session based on the remote Bluetooth address. After assembling and decrypting the packet, it is passed to a generic message handler.

4 Re-evaluation of Previous Attacks

As outlined in Section 1, several vulnerabilities described in [1] remain unfixed. We discuss these here in more detail.

4.1 Active Attacker-in-the-middle (MITM)

Due to Bridgefy's architecture, any PKB received from a new peer is inherently trusted, following the TOFU principle. That implies that Bridgefy is vulnerable to a MITM attack similar to the one reported in [1]. However, with the adoption of libsignal, the conditions necessary to perform the attack have changed slightly: Mallory now needs to perform the handshake with Bob before Alice does, whereas in earlier versions of Bridgefy this was not required.

The updated attack proceeds as follows: Assume that Alice and Bob have not met before. Mallory performs a handshake with each of Alice and Bob and impersonates them to one another. Any message then sent from one party is then relayed by Mallory to the other party.

If Mallory tries to perform the attack after Bob has already run a handshake with Alice, the following would happen: Mallory would try to impersonate Alice by performing a full handshake with Bob, using Alice's userId but Mallory's own PKB. When the SDK tries to store Mallory's PKB under Alice's userId, libsignal would throw an exception since Alice has already established a Signal session with Bob and so a PKB is already present under Alice's userId.

Note that Alice and Bob will never be able to confirm if they are directly exchanging messages or if they are instead subject to a MITM attack. This is because, in contrast to popular messaging applications like Signal, Bridgefy does not provide any mechanism to allow users to verify the keys of other peers manually.

4.2 Impersonation in the Broadcast Chat

An adversary can forge arbitrary broadcast messages. The adversary can send messages under the name of any userId and freely choose a payload content and display name. The reason for this is the lack of authentication for broadcast messages.

We implemented a proof of concept for this attack to verify it. We found, however, that the app permanently saves the display name of other peers based on their userId. Any peer that once received a message from Alice will remember her display name and permanently associate it with her userId. When Mallory sends a message using Alice's userId but with a different display name, other peers will still show Alice's real display name for this message.

4.3 Denial of Service (DoS)

We confirmed that Bridgefy remains vulnerable to a ZIP bomb attack as reported in [1]. This attack exploits that all packets are decompressed using gzip after decryption. An adversary can inject a specifically crafted packet that decompresses to more bytes than are available in the memory of a target. The target's app will first freeze and become unresponsive, and eventually crash. That allows Mallory to prevent specific devices from participating in the mesh network.

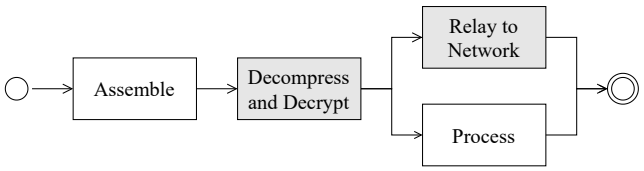


Figure 3: Since metadata is encrypted and compressed, a peer needs to decompress a packet before it can tell the type of message it received.

With its overhaul, Bridgefy now encrypts all metadata with the shared key. That makes it necessary for a peer to decompress a packet before being able to determine what type of message was received, as illustrated in Figure 3. Given the new flow to process incoming packets, the attack reported in [1] – where only a single message can shut down the entire network – no longer works, as it requires peers to forward mesh packets before decompression.

However, this vulnerability can be used to interfere with the correct functioning of the mesh network by shutting down several parts of the network. Specifically, all the peers that are one hop from the adversarially controlled peers can be taken offline. Given that resilience is a key requirement for Bridgefy's adoption in higher-risk environments, this attack invalidates one of Bridgefy's most central appeals.

4.4 Building a Social Graph

As reported in [1], Bridgefy previously transmitted the `sender` and `receiver` fields of multi-hop packets in plaintext. These are now encrypted under the shared network key. Thus, an adversary in the mesh network spanned by the Bridgefy messenger remains able to learn who is privately communicating with whom.

Using the `track` field of a `ForwardPacket`, an adversary can determine what nodes helped to deliver a packet. That permits building a model of the psychical topology of the mesh network. An adversary could also use this to trace back the location of a peer that repeatedly sends messages or relays such of other peers.

4.5 Historical Proximity Tracing

Bridgefy announced that they now protect against the historical proximity tracing method reported in [1]. However, our tests show that the attack is still possible: a full handshake is performed when two devices have not been near each other before, while only a partial handshake is performed otherwise.

An adversary can leverage this, e.g. to learn if a peer was physically present at a protest. Given that the timing and the approximate size of the handshake packets are known to the

adversary, the attack is even possible without knowledge of the shared symmetric key.

5 Breaking Confidentiality of One-to-One Messages

We identified a TOCTOU vulnerability in the SDK that can be leveraged to read private messages between two users of the Bridgefy application.

For simplicity, we assume that the communicating parties are not directly connected via Bluetooth. While this assumption is not strictly necessary, it makes the exploitation of this vulnerability easier.

Accompanying the textual description of the attack that follows, the packet flow used in the attack is illustrated in Figure 4. The numbering on the very left of the illustration matches the numbering in the individual steps in the following paragraphs.

Assume a setting where Alice and Bob's devices have already performed a handshake and have exchanged messages (e.g. M_0 in Figure 4). Bob's device then goes out of range of Alice's so that the Bluetooth connection is terminated (step 1 in Figure 4). If Alice's device was to now send a message to Bob's device, it would send it into the mesh network, as Bob's device is no longer a directly connected peer.

Next, Mallory performs a full handshake with Alice's device so that Alice's device registers Mallory's PKB (step 2 in Figure 4). Until this point, Mallory behaves normally as any honest peer would.

Mallory again sends the first packet of the handshake, this time using Bob's userId in place of Mallory's own (step 3 in Figure 4). No mechanism in Bridgefy prevents Mallory's message from being processed. Alice's device will now associate the established session with Bob. In particular, Alice's device will queue any subsequent packets intended for Bob in this session.

Because Mallory initiated a new handshake using Bob's userId, Alice's device will indicate to Alice that Bob's device is in range. Suppose then Alice types a message intended for Bob (M_1 in Figure 4). The SDK looks for any active session where the userId equals that of Bob's device as per our description in Section 3.7 (step 4 in Figure 4). Since Mallory provided the userId of Bob's device in its second handshake, Alice's session with Mallory yields a match. Hence, the message is queued in the `TransactionManager` for the session with Mallory. If the packet was dispatched at *this* point, the packet would be encrypted for Bob (this is because libsignal also uses the userId of the session to decide which key to use in the encryption). So Mallory would not be able to read it. However, instead of being dispatched, the packet is only queued.

Now, Mallory sends the first packet of the handshake for a third time, using Mallory's own userId (step 5 in Figure 4).

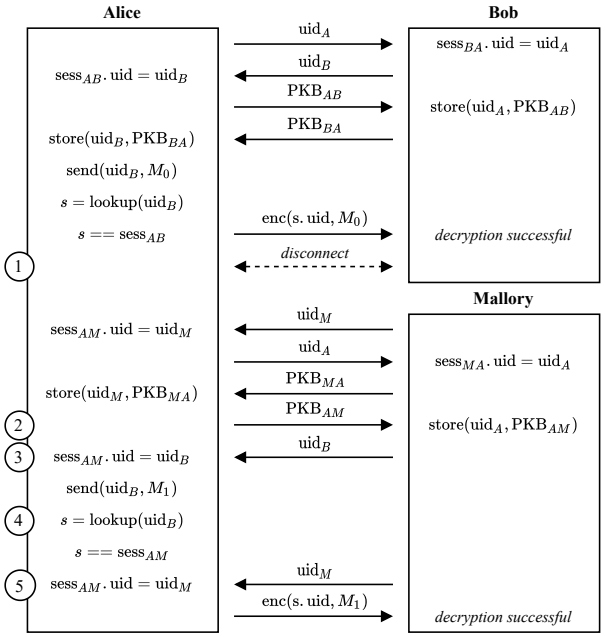


Figure 4: The packet flow of our TOCTOU attack on Bridgefy. Alice sends a message to Bob twice: the first message M_0 is sent to Bob only, but Mallory can decrypt the second message M_1, even though it was intended for Bob.

The userId of the session from the perspective of Alice's device now equals that of Mallory again. When the SDK on Alice's phone is asked for more data to transmit via Bluetooth, the packet is encrypted by Signal for Mallory and dispatched (again, libsignal uses the session's userId to decide which key to use in the encryption).

The above attack exploits a race condition: because Mallory sends the userId of Bob's device in its second handshake, Alice thinks she has a session with Bob. If she types a message for Bob, this message is then queued in a session with Mallory. But Mallory switches the userId back to its own userId in the third handshake so that when the message is dequeued and the libsignal encryption is performed, it is done using Mallory's public key.

Remark 2 *If no proper Signal session was established in the beginning, switching back to Mallory's real userId would require a full 2-round-trip handshake. Given that this attack exploits a race condition, it is hence important for Mallory to initiate an honest handshake before proceeding with the attack.*

We implemented a proof of concept for this attack to confirm that it works. We will publish the code in our public repository.[3] We sent 100 messages from Alice's phone, 56 of which were received and decrypted by Mallory in our tests. The fact that not all messages were received is explained

[3] https://github.com/eikendev/breaking-bridgefy-again

by the attack exploiting a race condition. What plays into the hands of Mallory is that Bridgefy reschedules a private message if it cannot be delivered to the receiver. If the SDK looks up a session matching the receiver's userId while the session is associated with Mallory's userId, it will be rescheduled. Still, packets can get 'lost' for Mallory when the packet is encrypted right after Mallory switches the userId back to Bob's.

Note that when Mallory intercepts a message, Bob will not receive it: Alice encrypts the packet for Mallory only, while Mallory cannot re-encrypt it for Bob in Alice's name. If Mallory was to encrypt and send a message to Bob while using Alice's userId during the handshake, Bob would fail to decrypt the packet. Instead, if Mallory used their real userId, Bob would process the packet before Mallory gets the chance to change the userId of the session again. In other words, the attack breaks confidentiality but not authentication.

6 Attacks on Broadcast Messages

Attacks on the confidentiality of broadcast messages are not an issue for the Bridgefy messenger, since the encryption key is assumed to be public knowledge anyway. However, they are relevant for other applications that use the Bridgefy SDK, where a per-application encryption key is used.

6.1 Distinguishing Attack

In Appendix A, we give a simple and efficient distinguisher that formally breaks the IND-CPA security of Bridgefy's broadcast message feature. That such an attack is possible seems intuitive given that the feature relies on AES-ECB, a deterministic encryption scheme. However, this intuition is not correct because, if we let the adversary only choose the payload content, then the scheme used is no longer deterministic (since broadcast packets also contain unpredictable fields such as the userIds, the sender's display name, and timestamps). Moreover these fields interact with the per-packet compression to produce variable plaintext inputs to AES-ECB, even if the payload contents are known. For this reason, our distinguisher breaking IND-CPA security does not rely on having controlled input blocks to AES-ECB but instead exploits Bridgefy's compression of the full plaintext before AES-ECB is applied. It is obtained by selecting in the IND-CPA security game pairs of equal-length payload contents: one payload content that compresses well and one that does not. AES-ECB aligns data on 16-byte block boundaries through padding, but we choose the payload contents so that the difference in compressed packet lengths is (sometimes) larger than one block in size.

6.2 Plaintext Recovery Attack

This compression-based side channel points the way to our plaintext recovery attack, which is the focus of the remainder of this section. We assume from here on that the broadcast message payload contents comes from a known set $\mathscr{P} = \{p_1, p_2, \ldots, p_n\}$ of possible payload contents. Additionally, we assume a network where a large number of devices participate, the senders of the broadcast messages have usernames of equal length, all broadcast messages contain the same payload content π (where $\pi \in \mathscr{P}$), and the adversary can capture M different packets containing payload content π at each of the first $H \leq 50$ hops. The attack we describe allows an adversary $\mathcal{A}$ to recover π given the set of $M \cdot H$ captured packets, without knowing the shared key.

In our attacks, we will work with different choices of $\mathscr{P}$. We define

$$\mathscr{P}_{\mathrm{b}} = \{p_i | 32 \leq i \leq 126\} \text{ and } \mathscr{P}_{\mathrm{w}}^{\ell,n} = \{p_1, p_2, \ldots, p_n\},$$

where $p_i \in \mathscr{P}_{\mathrm{b}}$ is a string that only consists of the single byte i – $\mathscr{P}_{\mathrm{b}}$ contains all printable ASCII characters that a user can type in the chat window of Bridgefy – and where $p_i \in \mathscr{P}_{\mathrm{w}}^{\ell,n}$ is the i-the most commonly used password of length ℓ in the rockyou password list [28] – a list of 32.6 million real-world passwords commonly used for password cracking. The payload contents in $\mathscr{P}_{\mathrm{w}}^{8,256}$, for instance, account for 2.2 % of all passwords in the whole rockyou data set. We do not anticipate Bridgefy users to be likely to broadcast passwords, but we use $\mathscr{P}_{\mathrm{w}}^{8,256}$ as a proxy for a set of likely broadcast messages. It could be replaced by a set of common English words, for example.

6.2.1 Core Idea. Our attack first simulates the broadcast of a large number of packets for each $p \in \mathscr{P}$, gathering statistics on the lengths of packets. The idea is that different values $p \in \mathscr{P}$ will produce different length distributions due to varying amounts of compression (and with the amount of compression possibly depending on the hop count and other header fields). This simulation step is done offline. Then, in the second step, we take the set of $M \cdot H$ captured packets, specifically, their lengths and use MLE to compute the likelihood of each candidate $p \in \mathscr{P}$, given the captured set of lengths. This allows us to rank all the candidates $p \in \mathscr{P}$ in order of likelihood.

6.2.2 Simulation Phase. The adversary runs its simulation to collect the lengths of a total of $N \cdot |\mathscr{P}| \cdot H$ packets. It builds $|\mathscr{P}|$ sets $\mathscr{X}_p$ of vectors, one set for each $p \in \mathscr{P}$. Each set $\mathscr{X}_p$ contains N vectors, and each vector has length H. A given vector $\mathbf{x} \in \mathscr{X}_p$ represents the sequence of packet lengths observed for a packet transporting payload content p being transmitted across H hops of the network. The lengths are all multiples of the AES block size of 16 bytes.

From this data, the adversary can derive an empirical value for $\theta_{\ell|h,p}$, this being the probability of observing a certain length ℓ given it is observed at hop h for a packet with payload content p. Note that at hop h the `hops` field has the value $50 - h + 1$.

Since this simulation phase is done offline, and the data reduction required to compute all the values $\theta_{\ell|h,p}$ can be done during the simulation itself, we can use large values of N and hence obtain accurate approximations to the corresponding distributions.

6.2.3 Attack Phase. The adversary observes M broadcast messages at each of the first $H \leq 50$ hops, yielding $M \cdot H$ packets. We assume all contain the same payload content $\pi \in \mathscr{P}$, where π is the target of the attack.

From the lengths $\ell_{i,h}$ of these $M \cdot H$ packets, the adversary computes counts $c_{\ell,h}$, where $c_{\ell,h}$ is the number of observed packets having length ℓ at hop h. Abstractly, we let $\mathbf{l}$ denote the collection of observed lengths and $\mathbf{c}$ the collection of observed counts.

Now we deploy MLE. For each choice of $p \in \mathscr{P}$, the adversary computes $\mathcal{L}(p|\mathbf{l})$, the likelihood that p is the correct payload content given the collection of observed lengths, and finds:

$$\hat{\pi} = \arg\max_{p \in \mathscr{P}} \mathcal{L}(p|\mathbf{l}).$$

If $\hat{\pi} = \pi$, then the adversary is able to successfully recover the payload content of the broadcast message. The adversary can also output a list of the top candidates by likelihood. It remains to describe how to compute $\mathcal{L}(p|\mathbf{l})$. Recall that

$$\mathcal{L}(p|\mathbf{l}) := \Pr(\mathbf{l}|p) = \prod_{i=1}^{M} \Pr(\ell_{i,1}\, \ell_{i,2} \ldots \ell_{i,H}|p).$$

We make the simplifying assumption that, in a given broadcast, the observed packet lengths are *independent* across the hops and depend only on the payload content p.[4] This enables us to write:

$$\begin{aligned}\mathcal{L}(p|\mathbf{l}) &= \prod_{i=1}^{M} \Pr(\ell_{i,1}|p) \cdot \Pr(\ell_{i,2}|p) \cdot \ldots \cdot \Pr(\ell_{i,H}|p) \\ &= \prod_{i=1}^{M} \prod_{h=1}^{H} \Pr(\ell_{i,h}|p) = \prod_{\ell} \prod_{h=1}^{H} \theta_{\ell|h,p}^{c_{\ell,h}}\end{aligned}$$

where, in the last step, we work with non-zero counts $c_{\ell,h}$ instead of individual packet lengths $\ell_{i,h}$. In practice, we work with the logarithm of the above expression, and compute

$$\hat{\pi} = \arg\max_{p \in \mathscr{P}} \log \mathcal{L}(p|\mathbf{l}) = \arg\max_{p \in \mathscr{P}} \sum_{\ell} \sum_{h=1}^{H} c_{\ell,h} \log(\theta_{\ell|h,p}).$$

Remark 3 *If a certain packet length ℓ is never observed for a pair (p,h) during the simulation phase, but that length appears in the attack sample, then we have $\theta_{\ell|h,p} = 0$ and then $\log(\theta_{\ell|h,p})$ will be undefined. In this case, we will use smoothing to set $\theta_{\ell|h,p}$ to a reasonable value. From the many smoothing methods available, we use Laplace smoothing [42, pg. 260] in our experiments. This is comparatively unsophisticated but easy to implement. We also experimented with Good-Turing smoothing, but it did not significantly improve our results.*

6.3 Attacking Single-Byte Payloads

We first discuss the instance of the attack where $\mathscr{P} = \mathscr{P}_{\mathrm{b}}$, meaning that π is a single byte. A realistic attack scenario for this would be when a protest leader surveys participants with the options to respond '`y`' for 'yes' and '`n`' for 'no'. If a significant majority answers with either option, an adversary could determine the answer of that majority with high probability.[5]

We now provide a possible mechanism to try to explain why our MLE attack based on compressed packet lengths can succeed in this case. The `hops` field in a `ForwardPacket` indicates the time to live (TTL) of the packet. For each hop in the network, this field decreases by 1, and the packet is eventually dropped before the value reaches 0. Because the fixed starting value 50 is used for all broadcast packets, the exact value of the field at each hop is known. When the `hops` field matches the byte value of π we can hope that a small amount of extra compression will be obtained, and the encrypted packet may end up being one block shorter than average.

Figure 5 shows the MessagePack-encoded `hops` field of a packet at the first hop, and its payload content. Because the fields do not share any surrounding bytes, the LZ77 compression alone cannot cause a change in the compressed packet length, but the Huffman coding can: in packets where the payload content π matches the `hops` field, we observed experimentally that the respective Huffman symbol is on average represented using 1 bit less than when they do not match. The root cause for this is the dynamic Huffman table in gzip, where more frequent symbols are assigned shorter code words.

The DEFLATE data is padded to a full byte, meaning that, in roughly 1 in 8 cases the single bit difference causes the gzip output to be one byte shorter when π matches the `hops` field than when it does not. Because the AES block size is 16 bytes, in a further fraction of 1 in 16 cases, this byte difference propagates through the AES-ECB layer and shows up as a packet that is one block shorter when π matches the `hops` field than when it does not.

Our MLE attack, which separates the packets by hop count, and which empirically estimates the distributions of packet lengths as a function of hop count and payload content, automatically makes use of any small signal arising through a compression leak of the type described above. In reality, based on experiments with single-byte payloads, the actual behaviour of the compression is significantly more complex. Neverthe-

[4] Instead of assuming that the observed packet lengths are independent across the hops, we could make a *first order Markov assumption* concerning the lengths. Because in our experimental results this only gave a negligible improvement, we here continue to assume independent lengths only.

[5] Our attack assumption was that all broadcast payload contents π are identical. This would not hold in the given scenario, but the minority answers can be regarded as noise in the likelihood computations.

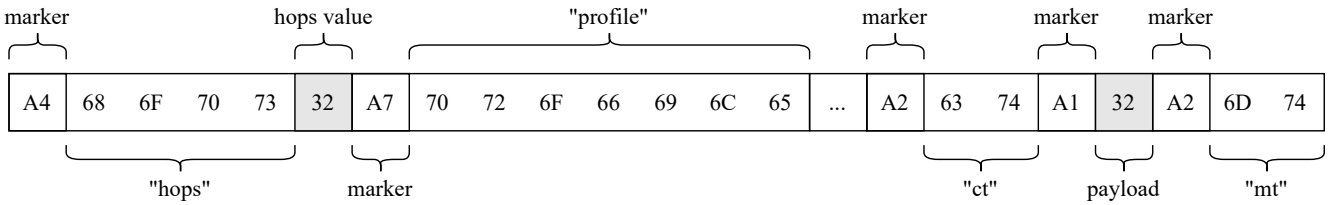


Figure 5: The `hops` field and the payload content in a MessagePack-serialised packet. All values are represented in hexadecimal, meaning this packet's `hops` value is 50.

less, the MLE attack still does improve plaintext prediction over random guessing, as we show in Section 6.5 below.

6.4 Equal-Length Payloads

We now discuss the case where $\mathscr{P} = \mathscr{P}_{\mathrm{w}}^{\ell,n}$. In our experiments, we take $\ell = 8$ and $n = 256$, implying that π is 8 bytes in length, so our attack will be trying to recover an 8-byte message from a set of 256 possibilities. We also look at $\mathscr{P} = \mathscr{P}_{\mathrm{w}}^{X,256}$, where we consider payload contents of mixed lengths. In the real world, when peers repeatedly share a message π with each other through the broadcast functionality, an adversary interested in this message could leverage the attack to recover π.

As two examples, $\mathscr{P}_{\mathrm{w}}^{8,256}$ contains the strings '11111111' and 'princess'. For the former, the effects of compression are easily visible in packet lengths because the string itself contains duplicate data. For the latter, the compression yields a signal because parts of the string can also appear in the metadata. As we will see in our experimental results, the signal is strong enough to give the MLE attack an advantage over random guessing.

6.5 Results

We implemented a proof of concept simulation for the attack to confirm that the compression leak is sensitive enough to provide statistically significant results. We perform the attack with $N = 2^{20}$, and with different H and M, using Laplace smoothing to account for lengths not encountered during the simulation.

We ran the attack for each of $\mathscr{P}_{\mathrm{b}}$, $\mathscr{P}_{\mathrm{w}}^{8,256}$, and $\mathscr{P}_{\mathrm{w}}^{X,256}$. For each such set $\mathscr{P}$, we ran the attack using each $\pi \in \mathscr{P}$ as a target, performing $n = 2^6$ attacks per target to remove noise. In each such run, we created a ranking of all possible candidates from $\mathscr{P}$ ordered by their log likelihoods. Before we present our results, we define the *rank* of an attack run.

Definition 1 *Let $\pi \in \mathscr{P}$ be the payload content used to generate a sample in an attack run. We call the **rank** of that attack run the index of π within the ordered ranking of all possible candidates from $\mathscr{P}$.*

So, for example, if the rank is 1, then the attack output the correct π as having the highest likelihood, if the rank is 2, then the attack output the correct π as having the second highest likelihood, etc.

Let $r_{\pi,i}$ be the rank of the i-th run for payload content π. We denote

$$\bar{r}_\pi = \frac{1}{n}\sum_{i=1}^{n} r_{\pi,i}$$

as the average rank over all runs for the payload content π, and

$$\bar{r}_{\mathscr{P}} = \frac{1}{|\mathscr{P}|}\sum_{\pi\in\mathscr{P}} \bar{r}_\pi$$

as the average rank over all runs for all payload contents in $\mathscr{P}$. To measure the overall accuracy of our attack, we look at the relative frequency of ranks among all measured payload contents. In particular, we are interested in the percentage of attack runs where the rank is less or equal to R, for increasing values of R. When randomly guessing π, we expect this relation to be linear – for example, half of the attacks would have a rank below the average value and the other half above. The plots in Figure 6 then highlight what difference our attack makes in comparison to random guessing.

Finally, we evaluate how the parameters H and M affect the accuracy of the attack. For this, we run simulations and attacks with various values for these parameters, while fixing $N = 2^{20}$ (in the simulation phase). Figure 7 shows the average rank across all payload contents for several combinations of parameters.

Our results demonstrate that the attack outperforms random guessing by a significant margin. As a point of comparison, $\mathscr{P}_{\mathrm{b}}$ is of size 95, so random guessing would result in an average rank of 47.5, whereas the attack achieves average ranks well below that. We note that increasing H (the number of hops over which packets are observed) leads to a better performance of the attack, as does increasing M (the number of packets available at each hop).

Remark 4 *Assuming an* a priori *non-uniform distribution on $\mathscr{P}$ is known, we can replace MLE by Maximum A Posteriori (MAP) estimation. This uses the* a priori *distribution to weight the candidates when computing likelihoods, resulting in a more powerful attack.*

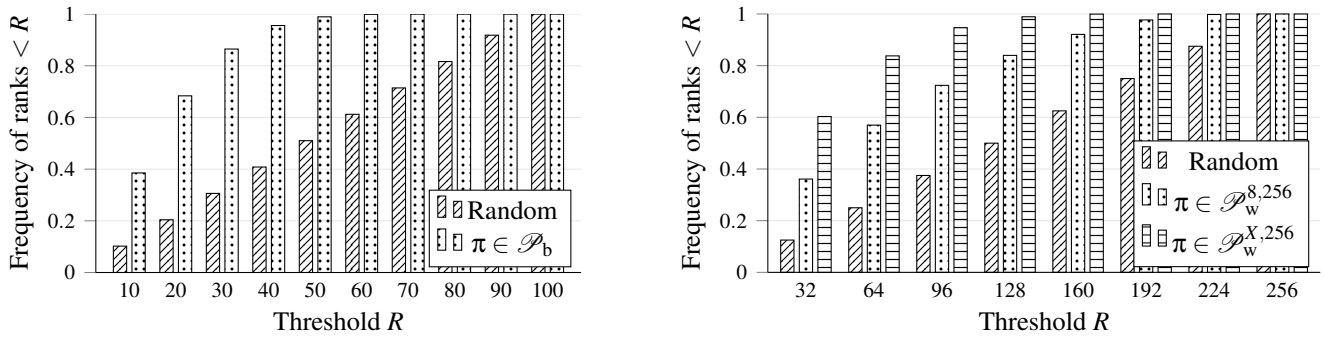
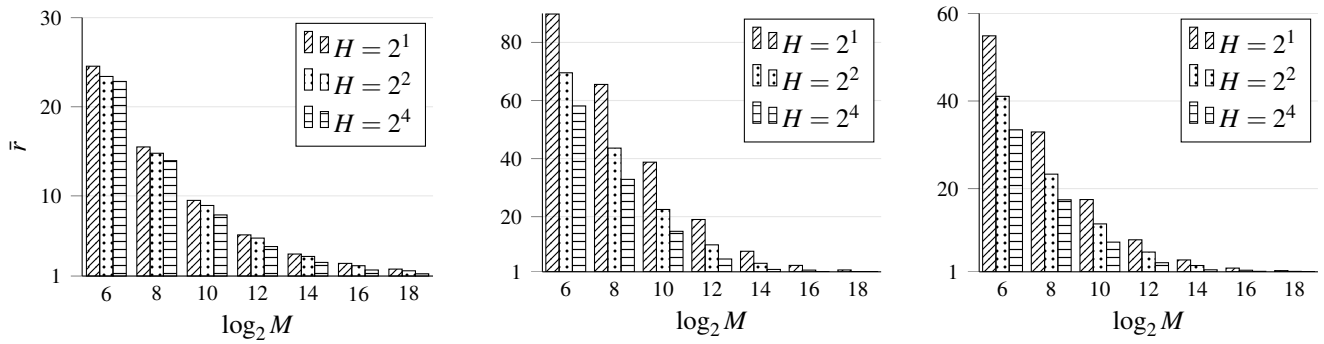


Figure 6: Cumulative frequency histogram of ranks, comparing our attack against random guessing for different $\mathscr{P}$. The plots show the portion of ranks less than R for $H = 2$, $N = 2^{20}$ and $M = 2^8$.



(a) $\pi \in \mathscr{P}_{\mathrm{b}}$ (b) $\pi \in \mathscr{P}_{\mathrm{w}}^{8,256}$ (c) $\pi \in \mathscr{P}_{\mathrm{w}}^{X,256}$

Figure 7: Average rank $\bar{r}$ for different (M,H).

7 Discussion

We gave a practical attack which enables an attacker to read private messages sent in the Bridgefy messenger because of an improper integration of the Signal protocol. Indeed, the most restricting requirement is physical presence: an adversary must be in sufficiently close proximity to the target to establish a direct Bluetooth connection. This does not pose an obstacle in a protest, where an adversary can disguise themselves as a protester. This highlights the dangers of 'bolting on' security to an otherwise unchanged design and suggests the need to carefully redesign Bridgefy to integrate tightly and fully its security mechanisms, here provided by the Signal protocol.

We also gave an attack to recover plaintexts of broadcast messages without knowing the shared encryption key. This attack is only relevant to the Bridgefy SDK but not to the Bridgefy application and its users, where this shared key can be assumed to be known to the adversary, as mentioned earlier. The attack is enabled by the decision to perform compression before encryption. This exposes information about the underlying data based on the output length. Our attack assumes a small space of possible plaintexts. This is reasonable in many contexts. For example, in a protest setting, a small group of protest leaders [4] may issue typical messages to the group. We illustrated the attack by recovering plaintext from a small set of possible passwords. Preventing compression oracle attacks is a known problem in the literature. The natural solution is to remove compression and to introduce an appropriate randomised padding mechanism to prevent length-based attacks. However, this needs to be done carefully so as to avoid traffic analysis attacks, cf. [21], and such an approach would reduce Bluetooth performance while enhancing security.

The privacy issues of Bridgefy remain largely unresolved. While sender and receiver identities are no longer transmitted in plaintext, any peer in the mesh can decrypt the relevant fields. In the case of the Bridgefy messenger, the userIds of sender and receiver hence continue to be publicly visible. An adversary can leverage this to build communication graphs and thereby identify protest leaders. Moreover, an adversary could use the `track` field of a `ForwardPacket` to approximate the physical location of a peer in the network. It remains an open research question how (flooding based) mesh networking can efficiently provide sender and receiver privacy even against passive adversaries.

Bridgefy's decision to adopt the Signal protocol to replace its previous home-grown RSA-based solution – heeding the often-repeated 'don't roll your own crypto' advice – is to

be applauded. Yet, our ability to still mount practical attacks against Bridgefy's flagship application highlights that this common mantra is insufficient. What else do we expect inexperienced developers to do other than call out to a third-party cryptographic library when faced with the task of securing their application? That is, while progress has been made in simplifying cryptographic APIs, the task of cryptographically securing an application is still non-trivial. Exploring whether this is inherent, i.e. whether a certain defensive development mindset and training will remain required, or not, is a fascinating and pressing question for usable, developer-friendly cryptography and security.

Overall, we note that Bridgefy's track record on security contrasts with the fact that it continues to be advertised to higher-risk users. Ours is the second work within a year to present significant and practically exploitable security vulnerabilities in Bridgefy's flagship application. Moreover, the long response time and lack of transparency of the development team when responding to reports of security vulnerabilities suggest a team inexperienced with handling security-critical issues. Yet, this track record does not seem to impact Bridgefy's popularity among those who offer advice on how to stay connected in face of government-mandated Internet shutdowns, as discussed in the introduction. This highlights that there is, seemingly, no good alternative to Bridgefy,[6] i.e. that the problem of secure offline messaging remains unsolved in practice.

Acknowledgements

The work of Albrecht was supported by the EPSRC grants EP/S020330/1, EP/P009417/1, EP/L018543/1. The work of Paterson was supported in part by a gift from VMware.

References

[1] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Mesh messaging in large-scale protests: Breaking Bridgefy. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 375–398. Springer, Heidelberg, May 2021. doi: 10.1007/978-3-030-75539-3_16.

[2] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in large-scale urban protests: the case of Hong Kong. In *30th USENIX Security Symposium (USENIX Security '21)*. USENIX Association, 2021.

[3] Connectivity Standards Alliance. https://zigbeealliance.org/solution/zigbee/, no date.

[4] Evronia Azer, G Harindranath, and Yingqin Zheng. Revisiting leadership in information and communication technology (ICT)-enabled activism: a study of Egypt's grassroots human rights groups. *New Media & Society*, 21(5):1141–1169, 2019.

[5] Lee Benfield. https://www.benf.org/other/cfr/, no date.

[6] Matt Bishop and Mike Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Mar. 1996. ISSN 0895-6340.

[7] Dan Boneh and Victor Shoup. *A Graduate Course in Applied Cryptography*. 2020. URL https://toc.cryptobook.us/.

[8] Tim Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 8259, December 2017. URL https://rfc-editor.org/rfc/rfc8259.txt.

[9] Rémi Bricout, Sean Murphy, Kenneth G. Paterson, and Thyla van der Merwe. Analysing and exploiting the mantin biases in RC4. *Des. Codes Cryptogr.*, 86(4):743–770, 2018. doi: 10.1007/s10623-017-0355-3. URL https://doi.org/10.1007/s10623-017-0355-3.

[10] Bridgefy. Press release – major security updates at bridgefy! https://bridgefy.me/press-release-major-security-updates-at-bridgefy/, October 2020.

[11] Bridgefy. https://twitter.com/bridgefy/status/1356603238674538496, February 2021. https://web.archive.org/web/20210514094051/https://twitter.com/bridgefy/status/1356603238674538496.

[12] Bridgefy. Bridgefy SDK (legacy). https://bridgefy.me/sdk-legacy/, 2021. https://web.archive.org/web/20220204151515/https://bridgefy.me/sdk-legacy/.

[13] Bridgefy. https://twitter.com/bridgefy/status/1359200080700600322, February 2021. https://web.archive.org/web/20210209175856/https://twitter.com/bridgefy/status/1359200080700600322.

[14] Bridgefy. https://www.bridgefy.me/docs/javadoc/, no date.

[15] Bridgefy. http://104.196.228.98:8081/artifactory/libs-release-local/, no date.

[16] Bridgefy. https://github.com/bridgefy/bridgefy-android-sdk-sample, no date.

[6] See [1] for a survey on mesh messaging.

[17] Antonio Cilfone, Luca Davoli, Laura Belli, and Gianluigi Ferrari. Wireless mesh networking: An iot-oriented perspective survey on relevant technologies. *Future Internet*, 11(4), 2019. ISSN 1999-5903. doi: 10.3390/fi11040099. URL https://www.mdpi.com/1999-5903/11/4/99.

[18] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *J. Cryptol.*, 33(4):1914–1983, 2020. doi: 10.1007/s00145-020-09360-1. URL https://doi.org/10.1007/s00145-020-09360-1.

[19] L. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, May 1996. URL https://rfc-editor.org/rfc/rfc1951.txt.

[20] L. Peter Deutsch. GZIP file format specification version 4.3. RFC 1952, May 1996. URL https://rfc-editor.org/rfc/rfc1952.txt.

[21] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE Symposium on Security and Privacy*, pages 332–346. IEEE Computer Society Press, May 2012. doi: 10.1109/SP.2012.28.

[22] Ingy döt Net. https://yaml.org/, no date.

[23] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. Can johnny build a protocol? co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols. In *European Workshop on Usable Security*, 2017.

[24] Sadayuki Furuhashi. https://msgpack.org/, no date.

[25] Christina Garman, Kenneth G. Paterson, and Thyla van der Merwe. Attacks only get better: Password recovery attacks against RC4 in TLS. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015: 24th USENIX Security Symposium*, pages 113–128. USENIX Association, August 2015.

[26] Christina Garman, Matthew Green, Gabriel Kaptchuk, Ian Miers, and Michael Rushanan. Dancing on the lip of the volcano: Chosen ciphertext attacks on apple iMessage. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016: 25th USENIX Security Symposium*, pages 655–672. USENIX Association, August 2016.

[27] GitHub – Bridgefy. https://github.com/bridgefy/bridgefy-android-sdk-sample/blob/56ad2acc7c8893cb2ba53f0aa5839b867ebea446/CHANGELOG.md, no date.

[28] GitHub – danielmiessler. https://github.com/danielmiessler/SecLists/blob/master/Passwords/Leaked-Databases/rockyou-withcount.txt.tar.gz, no date.

[29] GitHub – msgpack. https://github.com/msgpack/msgpack-java, no date.

[30] GitHub – msgpack. https://github.com/msgpack/msgpack/blob/master/spec.md, no date.

[31] GitHub – vmihailenco. https://github.com/vmihailenco/msgpack/, no date.

[32] Dan Goodin. Bridgefy, the messenger promoted for mass protests, is a privacy disaster. Ars Technica, https://arstechnica.com/features/2020/08/bridgefy, August 2020.

[33] Google. https://play.google.com/store/apps/details?id=me.bridgefy.main, no date.

[34] Robert Grosse. https://github.com/Storyyeller/enjarify, no date.

[35] Robert Grosse. https://github.com/Storyyeller/Krakatau, no date.

[36] Thread Group. https://www.threadgroup.org/, no date.

[37] Harry Halpin, Ksenia Ermoshina, and Francesca Musiani. Co-ordinating developers and high-risk users of privacy-enhanced secure messaging protocols. In Cas Cremers and Anja Lehmann, editors, *Security Standardisation Research - 4th International Conference, SSR 2018, Darmstadt, Germany, November 26-27, 2018, Proceedings*, volume 11322 of *Lecture Notes in Computer Science*, pages 56–75. Springer, 2018. doi: 10.1007/978-3-030-04762-7_4. URL https://doi.org/10.1007/978-3-030-04762-7_4.

[38] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952. doi: 10.1109/JRPROC.1952.273898.

[39] JetBrains. https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine, no date.

[40] John Kelsey. Compression and information leakage of plaintext. In Joan Daemen and Vincent Rijmen, editors, *Fast Software Encryption – FSE 2002*, volume 2365 of *Lecture Notes in Computer Science*, pages 263–276. Springer, Heidelberg, February 2002. doi: 10.1007/3-540-45661-9_21.

[41] John Koetsier. Hong Kong protestors using mesh messaging app China can't block: Usage up 3685%. https://web.archive.org/web/20200411154603/https://www.forbes.com/sites/johnkoetsier/2019/09/02/hong-kong-protestors-using-mesh-messaging-app-china-cant-block-usage-up-3685/, September 2019.

[42] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, Cambridge, UK, 2008. ISBN 978-0-521-86571-5. URL http://nlp.stanford.edu/IR-book/information-retrieval-book.html.

[43] Angelo Prado, Neal Harris, and Yoel Gluck. Ssl, gone in 30 seconds: A breach beyond crime. *Black Hat USA*, 2013, 2013.

[44] Tom Preston-Werner. https://toml.io/, no date.

[45] The Android Open Source Project. https://developer.android.com/studio/command-line/adb, no date.

[46] Ole André V. Ravnås. https://frida.re/, no date.

[47] Juliano Rizzo and Thai Duong. The crime attack. In *Ekoparty*, volume 2012, 2012.

[48] Mike Ryan. Bluetooth: With low energy comes low security. In *7th USENIX Workshop on Offensive Technologies (WOOT 13)*, Washington, D.C., August 2013. USENIX Association. URL https://www.usenix.org/conference/woot13/workshop-program/presentation/ryan.

[49] SensePost. https://github.com/sensepost/objection, no date.

[50] Bluetooth SIG. https://www.bluetooth.com/learn-about-bluetooth/recent-enhancements/mesh/, no date.

[51] Signal. https://github.com/signalapp/libsignal-protocol-java, no date.

[52] Signal. https://signal.org/, no date.

[53] Pallavi Sivakumaran and Jorge Blasco. A study of the feasibility of co-located app attacks against BLE and a large-scale analysis of the current application-layer security landscape. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019: 28th USENIX Security Symposium*, pages 1–18. USENIX Association, August 2019.

[54] skylot. https://github.com/skylot/jadx, no date.

[55] Mike Strobel. https://github.com/mstrobel/procyon, no date.

[56] The MITRE Corporation. https://cwe.mitre.org/data/definitions/367.html, July 2021.

[57] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1168917929301348354, September 2019. https://web.archive.org/web/20211010115025/https://twitter.com/bridgefy/status/1168917929301348354.

[58] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1268015807252004864, June 2020. http://archive.today/uKNRm.

[59] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1216473058753597453, January 2020. http://archive.today/x1gG4.

[60] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1268905414248153089, June 2020. http://archive.today/odSbW.

[61] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1456292964750307330, November 2021. https://web.archive.org/web/20220208195010/https://twitter.com/bridgefy/status/1456292964750307330.

[62] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1356750830955884552, February 2021. https://web.archive.org/web/20210516231628/https://twitter.com/bridgefy/status/1356750830955884552.

[63] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1371507779299590144, March 2021. https://web.archive.org/web/20210514094031/https://twitter.com/bridgefy/status/1371507779299590144.

[64] Twitter – Bridgefy. https://twitter.com/bridgefy/status/1356680753338318859, February 2021. https://web.archive.org/web/20210516231655/https://twitter.com/bridgefy/status/1356680753338318859.

[65] Paul C. van Oorschot. *Computer Security and the Internet: Tools and Jewels*. Springer, Cham, 2020.

[66] Mathy Vanhoef and Tom Van Goethem. Heist: Http encrypted information can be stolen through tcp-windows. In *Black Hat US Briefings, Las Vegas, USA*, 2016.

[67] Deutsche Welle. Afghanistan: How can messaging work safely in an internet shutdown? https://www.dw.com/en/afghanistan-how-can-messaging-work-s

afely-in-an-internet-shutdown/a-59018976, August 2021. https://web.archive.org/web/20211008155150/https://www.dw.com/en/afghanistan-how-can-messaging-work-safely-in-an-internet-shutdown/a-59018976.

[68] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave (Jing) Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. BLESA: Spoofing attacks against reconnections in bluetooth low energy. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020. URL https://www.usenix.org/conference/woot20/presentation/wu.

[69] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking secure pairing of bluetooth low energy using downgrade attacks. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020: 29th USENIX Security Symposium*, pages 37–54. USENIX Association, August 2020.

[70] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977. doi: 10.1109/TIT.1977.1055714.

All links were last checked on 2022-02-13.

A Broadcast Message Distinguisher

We formalise the game IND-CPA(q) analogous to [7, Section 5.3] between an adversary $\mathcal{A}$ and a challenger $\mathcal{C}$ that acts as a Left-or-Right (LoR) oracle. In the following game, let $\mathsf{KGen}, \mathsf{Enc}$, and Dec be the key generation function, the encryption function, and the decryption function employed by Bridgefy respectively, and $\mathcal{SE} = (\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ the symmetric encryption scheme. Note that Enc includes the compression using gzip, and Dec the decompression. Further, let $\ell(x)$ denote the length of string x and x^y the repetition of x with y copies.

Game IND-CPA(q):

1. $\mathcal{C}$ generates a key $K \leftarrow_{\$} \mathsf{KGen}$ and a bit $b \leftarrow_{\$} \{0,1\}$ uniformly at random.
2. $\mathcal{A}$ submits at most q queries to $\mathcal{C}$. In the ith query, $\mathcal{A}$ chooses two payload contents $\pi_{i,0}, \pi_{i,1}$, such that $B = \ell(\pi_{i,0}) = \ell(\pi_{i,1})$, and submits $(\pi_{i,0}, \pi_{i,1})$ to $\mathcal{C}$. $\mathcal{C}$ computes $c_i = \mathsf{Enc}_K(\pi_{i,b})$ and returns c_i to $\mathcal{A}$.
3. $\mathcal{A}$ outputs a guess $\hat{b} \in \{0,1\}$.

We denote the advantage of $\mathcal{A}$ in this game as

$$\mathbf{Adv}^{\text{IND-CPA}(q)}_{\mathcal{SE}}(\mathcal{A}, B) = 2 \cdot |\Pr(\hat{b} = b) - 1/2|.$$

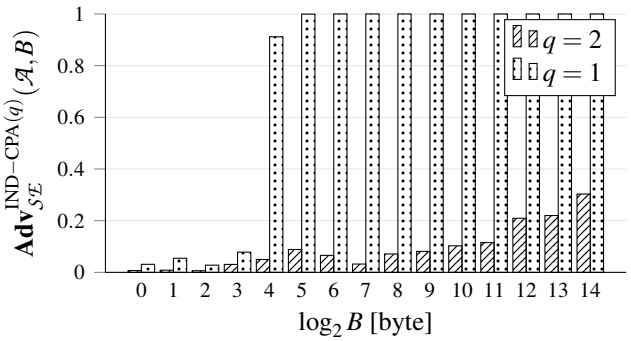


Figure 8: The advantage of $\mathcal{A}$ to win the game IND-CPA(q) for different B.

With $q \geq 2$, $\mathcal{A}$ can submit the pair $(\pi_{1,0}, \pi_{1,0})$ in the 2nd query, and compare $c_1 = \mathsf{Enc}_K(\pi_{1,b})$ to $c_2 = \mathsf{Enc}_K(\pi_{1,0})$: $\mathcal{A}$ can infer from matching blocks in c_1 and c_2 that the underlying gzip data also matches, suggesting that $b = 0$. Note that because we do not assume Enc to be deterministic, c_1 and c_2 do not match in every round of the game. We implemented this attack using a program that simulates a Bridgefy network with sufficient accuracy but without the physical setup. We measure $\mathbf{Adv}^{\text{IND-CPA}(2)}_{\mathcal{SE}}(\mathcal{A}, B)$ for different values of B in Figure 8 by playing the game $n = 2^{18}$ times each. The simulation confirms that Bridgefy's scheme is not IND-CPA secure.

But Bridgefy's scheme is even weaker: since compression precedes encryption, $\mathcal{A}$ can draw conclusions about the plaintext based on the ciphertext, if only the length of the plaintext is known. $\mathcal{A}$ can use this to win the game IND-CPA(1): $\mathcal{A}$ submits a single query (π_0, π_1) in Step 2, where π_1 contains duplicate data, but π_0 does not. In particular, we let π_0 be a random string, while π_1 is 0^B, where $B = \ell(\pi_0) = \ell(\pi_1)$.

Since π_1 is a string with duplicate data that can be compressed efficiently with gzip but π_0 is not, we can expect that $\ell(\mathsf{Enc}_K(\pi_0)) > \ell(\mathsf{Enc}_K(\pi_1))$ for increasing B. The difference in length of the compression output propagates to the length of the encryption output, such that b is leaked: in Step 3, $\mathcal{A}$ outputs

$$\hat{b} = \begin{cases} 0, & \text{if } \ell(c) > \frac{\ell(\mathsf{Enc}_{K'}(\pi_0)) + \ell(\mathsf{Enc}_{K'}(\pi_1))}{2} \\ 1, & \text{otherwise.} \end{cases}$$

Here, $\mathsf{Enc}_{K'}(\pi_0)$ and $\mathsf{Enc}_{K'}(\pi_1)$ are not derived by making a query to $\mathcal{C}$. Instead, $\mathcal{A}$ chooses an arbitrary key K' and runs $\mathsf{Enc}_{K'}$ locally. While K is unknown to $\mathcal{A}$, an arbitrary key K' can be chosen since only the output length is of interest. The userIds, sender's display name and timestamps used to derive c are also unknown to $\mathcal{A}$, and hence the values used by $\mathcal{A}$ will diverge from those used by $\mathcal{C}$. This introduces more noise to the compression length.

As before, we measure $\mathbf{Adv}^{\text{IND-CPA}(1)}_{\mathcal{SE}}(\mathcal{A}, B)$ for different B in Figure 8 by playing the game $n = 2^{18}$ times each. At

$B = 2^6$, $\mathcal{A}$ is already able to make a correct guess in each run of the game.

B Network Simulation Considerations

In this section, we discuss how we performed simulations with sufficient accuracy for our attacks. We implemented the program `ptxtrecov` in Go. `ptxtrecov` can repeatedly simulate Bridgefy networks, where a single node sends a broadcast message to other peers.

In the simulation phase, a large number of broadcast packets are generated for each payload content p at each hop h. The lengths of these packets are then recorded in a map which counts how often a length is observed for p at h.

While Bridgefy uses the official MessagePack library for Java [29], we used the Go library `vmihailenco/msgpack` [31]. To generate data representative for the Bridgefy application, we need to be especially careful in the serialisation step: MessagePack is somewhat flexible concerning the format used to encode a type. For instance, we found that Bridgefy converts timestamps with the `float64` format of MessagePack, although they could be converted with their dedicated timestamp extension format. Moreover, our Go library did not convert certain integers to their smallest possible format, as is intended by the MessagePack specification. To account for these differences, the types for these fields needed to be forced using our library.

The display name can have variable length, and so can the respective field of a packet. Because we draw conclusions based on the distribution of packet lengths, we need to ensure that the values we choose for this field do not cause a bias in the derived distributions. We decided to randomly choose the display names from a list composed of the 64 most common female and 64 most common male English names with a length of 5 characters.

We used a bespoke pseudo-random generator `aesrand` based on AES in counter mode to generate all the UUIDs needed in packets. We designed `aesrand` to be seeded and used it carefully so as to avoid both blocking and overlapping sequences of outputs in our multi-threaded simulations.

We also used `aesrand` to produce timestamps and time differences. In all experiments, we start at a constant base time T, given as a UNIX timestamp in microseconds. We can assume $\mathcal{A}$ to know T. The time at which a broadcast message is assumed to be sent (the `ds` field in the payload) is $t_0 = T + \Delta_0$, where Δ_0 is a random 24 bit integer. Δ_0 is drawn for each broadcast message individually. Note that using 24 bit for Δ_0 allow the attack to span over more than 4 h.

The packet's `creation` field is calculated as $t_1 = t_0 + \Delta_1$, where $4 \leq \Delta_1 < 64$. That is consistent with the behaviour of the Bridgefy messenger in the real world. The small delay occurs when the application passes the message on to the SDK for processing.

At each hop, the `added` field is set to the time when the packet is queued. The delay now reflects the time it takes to transmit the message via Bluetooth to the next hop and is, therefore, longer than Δ_1. We calculate the field as $t_2' = t_2 + \Delta_2$, where $128 \leq \Delta_2 < 512$, and t_2 is the field's value in the previous step.